Chapter 7 Files

7.1 Text Files & Binary Files

A computer file is a named collection of data that is kept on a disk or other device that holds data for an extended period of time. Most of the icons that appear on the screens of most computers represent files.

Files are divided into two categories by the way in which the data they hold is organized:

- text files — the kind you create by typing at the keyboard (or are created by the computer to look like such files). A very wide variety of programs can read and write text files.
- binary files — created by the computer with the intention that they will only be read by the computer (not conveniently read or written by humans). Binary files created by one program are often incompatible with other programs.

Java can create text files in a convenient way. Java can also conveniently read back the lines in the text file. Having Java extract individual words, punctuation, and numbers occurring in a single line of text is considerably more complex. Separating the parts of a line of text is discussed in section 12.3.

Java can easily create a binary file containing a representation of one or more objects. Java can just as easily read the file later and recreate the objects. The binary files that we will create are called "Object files"; these are the easiest to use for saving complex objects.

The classes that Java uses for reading and writing files are in the package *java.io*. To use these classes, place this line at the beginning of any class the reads or writes files.

```
import java.io.*;
```

Applets, Applications & Security

Programs that read from or write to files require security clearance to run as applets. Attempts to read or write files from applets generally result in security violation exceptions. It is much easier to run such programs as applications.

7.2 Text Files

Creating a Text File

Text is displayed using *System.out* with the same commands that would be used for placing text in a text file. The following is a large portion of the code which creates a text file saved on disk and called "letter.txt".

```
PrintWriter outfile;
outfile = new PrintWriter(new FileOutputStream("letter.txt"));
outfile.println("Dear John,");
outfile.println("    I'm sorry.");
outfile.println("        Love, Jane");
outfile.close();
```

The first line declares that within the program, we will use the name *outfile* for our file. Use of the class *PrintWriter* declares that:

- The file is a text file;
- Data may be placed in the file with the method *println*.

The second line creates a file named "letter.txt"; within the program we will use the name *outfile*.
The next three lines put text into the file.
The last line proclaims that we have finished creating the file.

Like *System.out.println*, *outfile.println* can output a textual representation of the values of variables. Thus, we can put any data from our program into a text file.

The text files we are creating here are 'ascii' text files. These files are not reliably portable containers for text that uses accent marks or letters other than the 26 letters used in English. Java can readily deal with 'unicode' text files which can handle text in almost any language. Unicode was not used here because unicode is not yet widely understood by non-Java computer programs and one of the principal uses for text files is interchange of data between programs.

The example above is incomplete. Special 'error catching' is required. Such 'error catching' must surround statements that can reasonably be expected to fail because of errors beyond the control of the programmer. The error prone statements must be surrounded by braces and preceded by the keyword *try*. This block of statements must be followed by a block of statements preceded by the keyword *catch*. Should an error occur, the statements in the try block stop executing, an error object (belonging to the class *Throwable*) is created, and the statements in the catch block are executed (having caught the error — the *Throwable* object).

The following creates a file if no error occurs and prints out an error message when one does.

```
try
    {
    PrintWriter outfile;
    outfile = new PrintWriter(new FileOutputStream("letter.txt"));
    outfile.println("Dear John,");
    outfile.println("    I'm sorry.");
    outfile.println("        Love, Jane");
    outfile.close();
    }
catch (IOException anError)
    {
    System.out.println("outfile error> " + anError); // display the error
    }
```

Reading a Text File

The following example reads the file "letter.txt" created in the last section and displays each line in the console window. You could just as well display the program file "MyPanel.java" by changing "letter.txt" to "MyPanel.java". You will not get reasonable results if you try to display the results of "MyPanel.class" because "MyPanel.class" is not a text file (it is a binary file, intended to be read only by computers).

```
try
    {
    BufferedReader infile;
    infile = new BufferedReader(new InputStreamReader(
                                                    new FileInputStream("letter.txt")));
    String line = infile.readLine();  // reads the first line so the while loop can test it
    while (line != null)
        {
        System.out.println(line);
        line = infile.readLine();      // reads subsequent lines which the while loop test before use
        }
    infile.close();
    }
catch (IOException anError)
    {
    System.out.println("infile error > " + anError); // display the error
    }
```

The following program allows the user to: create a list of names, see them on the screen, save the names in a text file, and retrieve the names from the created file – adding the names to the list:

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

public class StringProg extends EventPanel
   {
   private StringList aList;
   private TextField nameField;
   private AButton addButton, readButton, writeButton;

   public StringProg()
      {
      aList = new StringList();

      setLayout(null);
      nameField = new TextField("Carole");
      nameField.setLocation(300, 30);
      nameField.setSize(80, 25);
      add(nameField);

      addButton = new AButton("Add Name", 300, 70);
      readButton = new AButton("Read File", 300, 110);
      writeButton = new AButton("Write File", 300, 150);
      }

   public void mousePressed(MouseEvent e)
      {
      int x = e.getX(), y = e.getY();
      if (addButton.contains(x, y))
         {
         aList.addLine(nameField.getText());
         repaint();
         }
      else if (readButton.contains(x, y))
         {
         aList.readTextFile();
         repaint();
         }
      else if (writeButton.contains(x, y))
         {
         aList.writeTextFile();
         repaint();
         }
      }

   public void paint(Graphics g)
      {
      g.drawString("Name:", 300, 25);
      aList.paint(g);
      addButton.paint(g);
      readButton.paint(g);
      writeButton.paint(g);
      }
   }

class AButton
   {
   ... (from Chapter 4)
   }

class StringList
```

next page...

```
class StringList
   {
   private String list[];
   private int howMany;

   public StringList()
      {
      list = new String[30];
      howMany = 0;
      }

   public void addLine(String line)
      {
      list[howMany] = line;
      howMany++;
      }

   public void readTextFile()
      {
      try
         {
         BufferedReader infile;
         infile = new BufferedReader(new InputStreamReader(
                                                  new FileInputStream("names.txt")));
         list[howMany] = infile.readLine();
         while (list[howMany] != null)
            {
            howMany++;
            list[howMany] = infile.readLine();
            }
         infile.close();
         }
      catch (Throwable anError)
         {
         System.err.println("readFile> " + anError); // print the error
         }
      }

   public void writeTextFile()
      {
      try
         {
         PrintWriter outfile;
         outfile = new PrintWriter(new FileOutputStream("names.txt"));
         for (int h = 0; h < howMany; h++)
            {
            outfile.println(list[h]);
            }
         outfile.close();
         }
      catch (Throwable anError)
         {
         System.err.println("writeFile> " + anError); // print the error
         }
      }

   public void paint(Graphics g)
      {
      for (int h = 0, line = 20; h < howMany; h++, line += 15)
         {
         g.drawString(list[h], 20 ,line);
         }
      }
   }
```

If you wish to break the lines of text you are reading into words, punctuation and numbers, you will need to do additional programming. Separating the parts of a line of text is discussed in section 12.3.

Exercises — 7.2

1. Display the contents of several text files. Generally, all files whose names end with '.java', '.html' and '.txt' are text files. Word processors create binary files unless you have them 'save as text'. Sometimes the 'save as text' choice in a word processor is called 'save as ascii'.
2. Use a dialog box to get the name of the file for reading or writing.
3. Display the information from the file in the graphics window instead of the console window. Consider paging the information to allow viewing one screen full of information at a time. Allow for moving forward or backward in the file.
4. Select a program from chapter 6 that displays a list of names and/or numbers. Give the user the option of creating a text file containing the data.
5. Select a program from chapter 6 that displays a list of names. Give the user the option of having the program read names from a text file and add them to the list.

7.3 Object files

Creating and Reading an Object file

An *Object* file will save or restore a complex object with just one statement. An object to be placed in an *Object* file must be described by a class declaring that it *implements Serializable*. You don't need to do anything to implement *Serializable*, except write *implements Serializable* on the first line of the class. A small number of the classes that come with Java systems cannot be included in a *Serializable* class. If you encounter such an unserializable class, the Java compiler will inform you when you are compiling your program.

The following modification of the example in section 7.2 saves the list of names into the binary file "theList" when the method *writeObjectFile* is called. Calling the *readObjectFile* method in the program will replace the data in the object *aList* with the data in the file.

To read and write *Object* files instead of text files, the program of section 7.2 is modified as follows:

- The methods *writeObjectFile* and *readObjectFile* are added to the class *StringProg*.
- The method *mousePressed* (in StringProg) calls *writeObjectFile* and *readObjectFile* instead of *aList.writeTextFile* and *aList.readTextFile*.
- The words *implements Serializable* are added to the first line of the class *StringList*.
- The unused methods *writeTextFile* and *readTextFile* are removed from the class *StringList*.

```
class StringProg
    {
    ...
    private void writeObjectFile()        // Call this method when writeButton is pressed
        {
        try
            {
            ObjectOutputStream outfile;
            outfile = new ObjectOutputStream(new FileOutputStream("theList"));
            outfile.writeObject(aList);
            outfile.close();
            }
        catch (Throwable anError)
            {
            System.err.println("outfile error> " + anError);      // display the error
            }
        }

    private void readObjectFile()         // Call this method when readButton is pressed
        {
        try
            {
            ObjectInputStream infile;
            infile = new ObjectInputStream(new FileInputStream("theList"));
            aList = (StringList)infile.readObject();
            infile.close();
            }
        catch (Throwable anError)
            {
            System.err.println("infile error> " + anError);      // display the error
            }
        }
    }

class StringList implements Serializable
    {
    ...   // same as in section 7.2 – the methods readFile and writeFile are not used and may be omitted
    }
```

Writing an object writes the object and any objects that it may extend or contain. All objects that are to be written to an object file must be defined by a *Serializable* class; this includes classes defining objects which appear as instance variables inside a class to be written.

Object files can also contain several objects that are not nested within each other. Include calls to the method *writeObject* for each object not nested in any other. To read an object file containing several separate objects, include a call to the method *readObject* for each object. You must read the objects in the same order as they were written.

7.3 Exercises

1. Give the user of one of the stars programs in chapter 4 the option of creating a file containing the state of the stars. Also supply the option of using this file to restore the stars to this state.
2. Choose other programs in which to allow the user to save the state into a file. Allow the user to restore the state from the file.

7.4 File Dialogs

Sections 7.2 and 7.3 used a fixed name for the file. The user of a program should be able to use the computer's operating system to see existing files and then choose a name for the file. The following method will obtain file names in this way. Each method returns *null* when the user chooses to cancel the operation.

```
public String getFileName(boolean readingFromFile, String suggestedFileName)
   {
   FileDialog fd;
   if (readingFromFile)
      fd = new FileDialog((Frame)getParent(), "Open", FileDialog.LOAD);
   else
      fd = new FileDialog((Frame)getParent(), "Save", FileDialog.SAVE);
   fd.setDirectory(".");
   fd.setName(suggestedFileName);
   fd.setVisible(true);
   if (fd.getFile() == null)
      return null;
   return fd.getDirectory() + fd.getFile();
   }
```

Use the result from *getFileName* as in the following example:

```
String fileName = getFileName(true, "letter.txt");
if (fileName != null)
   try
      {
      BufferedReader infile;
      infile = new BufferedReader(new InputStreamReader(
                                                new FileInputStream(fileName)));
      ....
```

If you are constructing your programs as described in the appendices, this method should be part of the class *MyPanel*. It will not work as written if it is placed in another class, a class other than the one added to a *Frame*.

As written, the method *getFileName* will only work if it is placed in a *Component* that has been added to a *Frame*. The method uses *getParent* to reference the *Frame*. The method can be used in general locations in an application, but a reference to a *Frame* must be provided in the constructor of the *FileDialog*. In particular, if *getFileName* is to be placed in a *Frame*, it will work if the expression *(Frame)getParent()* is replaced by *this* in the two places *(Frame)getParent()* appears.

The method *getFileName* will not work in *MyPanel* with an applet created as described in the appendices — a *Frame* would have to be created. File reading or writing generally will not work with applets without special security settings anyway.