PREVIOUS Chapter 4: The Java Platform NEXT

4.9. Files and Directories

The `java.io.File` class represents a file or a directory and defines a number of important methods for manipulating files and directories. Note, however, that none of these methods allow you to read the contents of a file; that is the job of `java.io.FileInputStream`, which is just one of the many types of input/output streams used in Java and discussed in the next section. Here are some things you can do with `File`:

```
import java.io.*;

// Get the name of the user's home directory and represent it with a File
File homedir = new File(System.getProperty("user.home"));

// Create a File object to represent a file in that directory
File f = new File(homedir, ".configfile");

// Find out how big a file is and when it was last modified
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());

// If the file exists, is not a directory, and is readable,
// move it into a newly created directory.
if (f.exists() && f.isFile() && f.canRead()) {        // Check config file
  File configdir = new File(homedir, ".configdir");  // A new config directory
  configdir.mkdir();                                 // Create that directory
  f.renameTo(new File(configdir, ".config"));        // Move the file into it
}

// List all files in the home directory
String[] allfiles = homedir.list();

// List all files that have a ".java" suffix
String[] sourcecode = homedir.list(new FilenameFilter() {
  public boolean accept(File d, String name) { return name.endsWith(".java"); }
});
```

The `File` class provides some important additional functionality as of Java 1.2:

```
// List all filesystem root directories; on Windows, this gives us
// File objects for all drive letters (Java 1.2 and later).
File[] rootdirs = File.listRoots();

// Atomically, create a lock file, then delete it (Java 1.2 and later)
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
  // We successfully created the file, so do something
  ...
```

```
    // Then delete the lock file
    lock.delete();
  }
  else {
    // We didn't create the file; someone else has a lock
    System.err.println("Can't create lock file; exiting.");
    System.exit(0);
  }

  // Create a temporary file to use during processing (Java 1.2 and later)
  File temp = File.createTempFile("app", ".tmp");  // Filename prefix and suffix
  // Make sure file gets deleted when we're done with it (Java 1.2 and later)
  temp.deleteOnExit();
```

The `java.io` package also defines a `RandomAccessFile` class that allows you to read binary data from arbitrary locations in a file. This can be a useful thing to do in certain situations, but most applications read files sequentially, using the stream classes described in the next section. Here is a short example of using `RandomAccessFile`:

```
// Open a file for read/write ("rw") access
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100);                    // Move to byte 100 of the file
byte[] data = new byte[100];    // Create a buffer to hold data
f.read(data);                   // Read 100 bytes from the file
int i = f.readInt();            // Read a 4-byte integer from the file
f.seek(100);                    // Move back to byte 100
f.writeInt(i);                  // Write the integer first
f.write(data);                  // Then write the 100 bytes
f.close();                      // Close file when done with it
```

PREVIOUS 4.8. Threads | HOME | BOOK INDEX | NEXT 4.10. Input and Output Streams

PREVIOUS Chapter 4: The Java Platform NEXT

4.10. Input and Output Streams

The `java.io` package defines a large number of classes for reading and writing streaming, or sequential, data. The `InputStream` and `OutputStream` classes are for reading and writing streams of bytes, while the `Reader` and `Writer` classes are for reading and writing streams of characters. Streams can be nested, meaning you might read characters from a `FilterReader` object that reads and processes characters from an underlying `Reader` stream. This underlying `Reader` stream might read bytes from an `InputStream` and convert them to characters.

There are a number of common operations you can perform with streams. One is to read lines of input the user types at the console:

```
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.print("What is your name: ");
String name = null;
try {
  name = console.readLine();
}
catch (IOException e) { name = "<" + e + ">"; }  // This should never happen
System.out.println("Hello " + name);
```

Reading lines of text from a file is a similar operation. The following code reads an entire text file and quits when it reaches the end:

```
String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  String line;
  while((line = in.readLine()) != null) {  // Read line, check for end-of-file
    System.out.println(line);              // Print the line
  }
  in.close();    // Always close a stream when you are done with it
}
catch (IOException e) {
  // Handle FileNotFoundException, etc. here
}
```

Throughout this book, you've seen the use of the `System.out.println()` method to display text on the console. `System.out` simply refers to an output stream. You can print text to any output stream using similar techniques. The following code shows how to output text to a file:

```
try {
  File f = new File(homedir, ".config");
```

```
  PrintWriter out = new PrintWriter(new FileWriter(f));
  out.println("## Automatically generated config file. DO NOT EDIT!");
  out.close();  // We're done writing
}
catch (IOException e) { /* Handle exceptions */ }
```

Not all files contain text, however. The following lines of code treat a file as a stream of bytes and read the bytes into a large array:

```
try {
  File f;                              // File to read; initialized elsewhere
  int filesize = (int) f.length();     // Figure out the file size
  byte[] data = new byte[filesize];    // Create an array that is big enough
  // Create a stream to read the file
  DataInputStream in = new DataInputStream(new FileInputStream(f));
  in.readFully(data);  // Read file contents into array
  in.close();
}
catch (IOException e) { /* Handle exceptions */ }
```

Various other packages of the Java platform define specialized stream classes that operate on streaming data in some useful way. The following code shows how to use stream classes from `java.util.zip` to compute a checksum of data and then compress the data while writing it to a file:

```
import java.io.*;
import java.util.zip.*;

try {
  File f;                            // File to write to; initialized elsewhere
  byte[] data;                       // Data to write; initialized elsewhere
  Checksum check = new Adler32();    // An object to compute a simple checksum

  // Create a stream that writes bytes to the file f
  FileOutputStream fos = new FileOutputStream(f);
  // Create a stream that compresses bytes and writes them to fos
  GZIPOutputStream gzos = new GZIPOutputStream(fos);
  // Create a stream that computes a checksum on the bytes it writes to gzos
  CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

  cos.write(data);             // Now write the data to the nested streams
  cos.close();                 // Close down the nested chain of streams
  long sum = check.getValue(); // Obtain the computed checksum
}
catch (IOException e) { /* Handle exceptions */ }
```

The `java.util.zip` package also contains a `ZipFile` class that gives you random access to the entries of a ZIP archive and allows you to read those entries through a stream:

```
import java.io.*;
import java.util.zip.*;
```

```
String filename;     // File to read; initialized elsewhere
String entryname;    // Entry to read from the ZIP file; initialized elsewhere
ZipFile zipfile = new ZipFile(filename);        // Open the ZIP file
ZipEntry entry = zipfile.getEntry(entryname);   // Get one entry
InputStream in = zipfile.getInputStream(entry); // A stream to read the entry
BufferedInputStream bis = new BufferedInputStream(in);  // Improves efficiency
// Now read bytes from bis...
// Print out contents of the ZIP file
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
  ZipEntry zipentry = (ZipEntry) e.nextElement();
  System.out.println(zipentry.getName());
}
```

If you need to compute a cryptographic-strength checksum (also knows as a message digest), use one of the stream classes of the `java.security` package. For example:

```
import java.io.*;
import java.security.*;
import java.util.*;

File f;          // File to read and compute digest on; initialized elsewhere
List text = new ArrayList();  // We'll store the lines of text here

// Get an object that can compute an SHA message digest
MessageDigest digester = MessageDigest.getInstance("SHA");
// A stream to read bytes from the file f
FileInputStream fis = new FileInputStream(f);
// A stream that reads bytes from fis and computes an SHA message digest
DigestInputStream dis = new DigestInputStream(fis, digester);
// A stream that reads bytes from dis and converts them to characters
InputStreamReader isr = new InputStreamReader(dis);
// A stream that can read a line at a time
BufferedReader br = new BufferedReader(isr);
// Now read lines from the stream
for(String line; (line = br.readLine()) != null; text.add(line)) ;
// Close the streams
br.close();
// Get the message digest
byte[] digest = digester.digest();
```

So far, we've used a variety of stream classes to manipulate streaming data, but the data itself ultimately comes from a file or is written to the console. The `java.io` package defines other stream classes that can read data from and write data to arrays of bytes or strings of text:

```
import java.io.*;

// Set up a stream that uses a byte array as its destination
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");            // Write some string data out as bytes
```

```
out.writeDouble(Math.PI);         // Write a floating-point value out as bytes
byte[] data = baos.toByteArray(); // Get the array of bytes we've written
out.close();                      // Close the streams

// Set up a stream to read characters from a string
Reader in = new StringReader("Now is the time!");
// Read characters from it until we reach the end
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

Other classes that operate this way include `ByteArrayInputStream`, `StringWriter`, `CharArrayReader`, and `CharArrayWriter`.

`PipedInputStream` and `PipedOutputStream` and their character-based counterparts, `PipedReader` and `PipedWriter`, are another interesting set of streams defined by `java.io`. These streams are used in pairs by two threads that want to communicate. One thread writes bytes to a `PipedOutputStream` or characters to a `PipedWriter`, and another thread reads bytes or characters from the corresponding `PipedInputStream` or `PipedReader`:

```
// A pair of connected piped I/O streams forms a pipe. One thread writes
// bytes to the PipedOutputStream, and another thread reads them from the
// corresponding PipedInputStream. Or use PipedWriter/PipedReader for chars.
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new PipedInputStream(writeEndOfPipe);

// This thread reads bytes from the pipe and discards them
Thread devnull = new Thread(new Runnable() {
  public void run() {
    try { while(readEndOfPipe.read() != -1); }
    catch (IOException e) {}  // ignore it
  }
});
devnull.start();
```

One of the most important features of the `java.io` package is the ability to *serialize* objects: to convert an object into a stream of bytes that can later be deserialized back into a copy of the original object. The following code shows how to use serialization to save an object to a file and later read it back:

```
Object o;  // The object we are serializing; it must implement Serializable
File f;    // The file we are saving it to

try {
  // Serialize the object
  ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(f));
  oos.writeObject(o);
  oos.close();

  // Read the object back in:
  ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f));
  Object copy = ois.readObject();
  ois.close();
```

```
    }
    catch (IOException e) { /* Handle input/output exceptions */ }
    catch (ClassNotFoundException cnfe) { /* readObject() can throw this */ }
```

The previous example serializes to a file, but remember, you can write serialized objects to any type of stream. Thus, you can write an object to a byte array, then read it back from the byte array, creating a deep copy of the object. You can write the object's bytes to a compression stream or even write the bytes to a stream connected across a network to another program!

PREVIOUS | HOME | NEXT
4.9. Files and Directories | BOOK INDEX | 4.11. Networking