Crossroads
The ACM Student Magazine

Java I/O and Compression

by ***Matt Tucker***

The Objective Viewpoint column provides practical programming information. Last year's column gave instruction on the Java programming language. This year's column will build on that foundation and extend it by providing Java programming tips that are relevant to the focus of each *Crossroads* issue.

In this column we will look at several built-in Java classes for performing compression. Before learning to use these classes to create Zip and GZip archives, we will review basic Java input and output (I/O).

Basics of Java Input and Output

Beginning Java users, especially C++ users accustomed to cin and cout, bemoan the seeming complexity of doing input and output in Java. In fact, after learning a few basic concepts, the Java I/O package is elegant and easy to use.

Because of the size of the Java I/O library, our own look at it will be slightly abbreviated. Readers looking for a more thorough explanation may wish to consult Sun's Java Tutorial [1] or another reference.

Concept 1: Class Hierarchy

Input and output classes in Java are organized in a hierarchical manner. Readers and Writers are used for reading and writing characters whereas InputStreams and OutputStreams are primarily used for reading and writing binary data. Figure 1 shows most of the Reader and OutputStream class hierarchies.

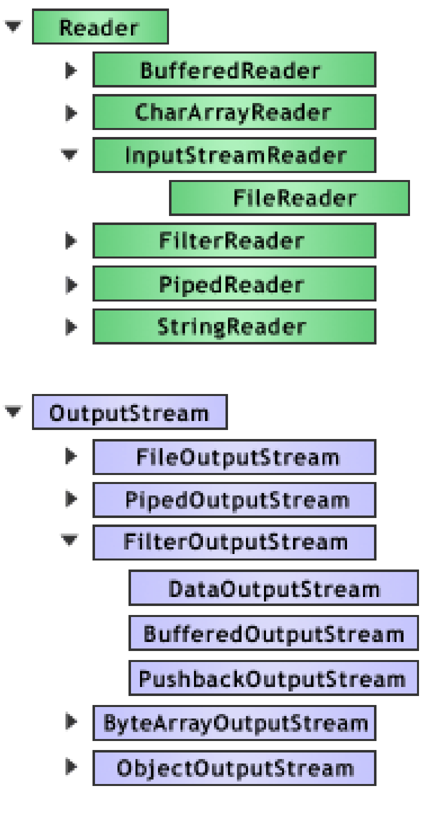

Figure 1: I/O Class Hierarchy

This class design allows us to easily perform some fancy tricks. Suppose we must write a program to read data out of a text file. To do this, we might construct a FileReader as follows:

```
FileReader fin = new FileReader("someFile.txt");
```

This will give us a stream that we can read from. However, to read the text file one line at a time and to improve performance, we need to wrap our FileReader within a BufferedReader:

```
BufferedReader bin = new BufferedReader(fin);
String line;
//Keep reading lines until there are none left and
//print each line to standard output as we go.
while((line = bin.readLine()) != null) {
    System.out.println(line);
}
//Close input stream
bin.close();
```

It is the hierarchical nature of the I/O class library that allows us to build up streams this way. To see how, we can take a peek at the BufferedReader constructor:

```
public BufferedReader(Reader in)
```

Because FileReader extends Reader, it is legal to create a BufferedReader out of a FileReader. The possibilities of linking together various types of Readers, Writers, and Streams are almost endless.

One further example illustrates the power and flexibility of this design. It is possible that our program requirements in the above example might suddenly change so that we have to read text data from an Internet address rather than a local file. Luckily, we can simply redefine our BufferedReader object and leave the rest of the program alone:

```
URL u = new URL("http://www.aHost.com/someFile.txt");
InputStreamReader isr = new InputStreamReader(u.openStream());
BufferedReader bin = new BufferedReader(isr);
```

In this example, we create an URL object with the address of our text file and then construct an InputStreamReader using that URL. Since InputStreamReader is a subclass of Reader, we create our BufferedReader using an InputStreamReader object.

One of the largest difficulties of performing I/O in Java is figuring out which class or set of classes is best suited for your task. With a bit of practice you will soon be proficient!

Concept 2: Basic Algorithms for Reading and Writing

Now that we have explored how to create Readers, Writers, and Streams, we must learn how to actually use them in order to become proficient in performing input and output. Happily, this task can be reduced to a simple algorithm:

1. Open Streams or Readers and Writers.
2. Read and write data until you are done.
3. Close Streams or Readers and Writers.

The second step is the most interesting; it may mean reading lines out of a text file until there are no more to read, or reading the first ten bytes from a binary file.

We have already seen an example of the input/output algorithm above that uses a BufferedReader to read lines from a text file and then print them to System.out. The equivalent task with Streams looks very similar. For our example, suppose that there is an image named java1.jpg that we want to copy into a new file java2.jpg:

```
//Step 1: open streams
BufferedInputStream in = new BufferedInputStream(new FileInputStream("java1.jpg"));
BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("java2.jpg"));
//Step 2: read and write until done
byte[] buf = new byte[1024];
int len;
while ((len = in.read(buf)) >= 0) {
   out.write(buf,0,len);
}
//Step 3: close streams
in.close();
out.close();
```

Reading binary data is different than reading text. Instead of reading one line into a String, we read an array of bytes. In the example code, we have arbitrarily defined the number of bytes to read in each time to be 1024. The integer variable `len` is also required to keep track of how many bytes *actually* get read. Consider the case of a 3024-byte file. If we read 1024 bytes during the first and second reads, there would be 976 bytes left. `in.read(buf)` would attempt to read 1024 bytes, but would only be able to read 976 bytes. The `len` variable keeps track of this fact so that we write the correct number of bytes to the OutputStream.

Compression in Java

Building on our knowledge of performing input and output, we can explore Java's built-in utilities for creating compressed Zip and GZip[2] files. Both Zip and GZip compress binary data. The primary difference between the two formats from a user's perspective is that Zip archives can include multiple files while GZip compresses only one file. To illustrate how to use the Java compression libraries, we will examine a few sample programs.

GZip Archives

The first program, SimpleGZip, demonstrates the use of the Java libraries for creating GZip archives. The program takes the name of a file as an argument and then compresses that file and gives it a .gzip extension.

Because GZip archives are limited to containing only one file, the format is often used in conjunction with Tar -- multiple files are joined into one Tar file and then compressed using GZip. As an exercise, you may want to use one of the available open source Java Tar libraries [3] to modify the SimpleGZip program to create .tar.gzip archives.

Code Listing: SimpleGZip.java

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

/**
 * Compresses a file into a GZip archive. Users of the class
 * supply the name of the file as an argument.
 */
public class SimpleGZip {

   public static void main(String[] args) {
      //User must specify a file to compress
      if (args.length < 1) {
         System.out.println("Usage: java SimpleGZip fileName");
         System.exit(0);
      }
      //Get the name of the file to compress.
      String fileName = args[0];
      //Use the makeGZip method to create a gzip archive.
      try {
         makeGZip(fileName);
      }
      //Simply print out any errors we encounter.
      catch (Exception e) {
         System.err.println(e);
      }
   }

   /**
    * Creates a GZip archive using the file name passed in as a parameter.
    */
   public static void makeGZip(String fileName)
         throws IOException, FileNotFoundException
   {
      File file = new File(fileName);
      FileOutputStream fos = new FileOutputStream(file + ".gzip");
      GZIPOutputStream gzos = new GZIPOutputStream(fos);
      //Create a buffered input stream out of the file we are
      //trying to add into the gzip archive.
      FileInputStream fin = new FileInputStream(file);
      BufferedInputStream in = new BufferedInputStream(fin);
      //Create a buffer for reading raw bytes from the file.
      byte[] buf = new byte[1024];
      //The len variable will keep track of how much we
      //are able to actually read each time we try.
      int len;
      //Read from the file and write to the gzip archive.
      while ((len = in.read(buf)) >= 0) {
         gzos.write(buf,0,len);
      }
      in.close();
      gzos.close();
   }
}
```

Notice that the important parts of the program are similar to the examples given in the I/O section of this column. First, we want to write data to a new file, so we create a FileOutputStream:

```
FileOutputStream fos = new FileOutputStream(file + ".gzip");
```

Then, we tell Java that we want to compress whatever we write to that file using the GZip format by wrapping the FileOutputStream with a GZipOutputStream:

```
GZIPOutputStream gzos = new GZIPOutputStream(fos);
```

We read data from a file using a FileInputStream, and then follow the read/write algorithm to transfer data from the uncompressed file to the new compressed file. After the process is done, we close the streams and the program is finished.

Mastering GZip compression in Java requires only a knowledge of the GZipOutputStream class, the GZipInputStream class, and basic Java I/O.

Zip Archives

Because Zip archives can contain multiple files, the ZipOutputStream is a bit more complex than the GZipOutputStream class. The second code example, SimpleZip, demonstrates how to add multiple entries to a Zip stream. Users of the program supply the name of a file or a directory to compress as an argument. The case of supplying a directory as an argument is more interesting since the program will recursively add all of the directory's contents (including subdirectories) to the Zip archive.

Code Listing: SimpleZip.java

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

/**
 * Compresses a file or directory into a Zip archive. Users of the
 * class supply the name of the file or directory as an argument.
 */
public class SimpleZip {

   private static ZipOutputStream zos;

   public static void main(String[] args) {
      //User must specify a directory to compress
      if (args.length < 1) {
         System.out.println("Usage: java SimpleZip directoryName");
         System.exit(0);
      }
      //Get the name of the file or directory to compress.
      String fileName = args[0];
      //Use the makeZip method to create a Zip archive.
      try {
         makeZip(fileName);
      }
      //Simply print out any errors we encounter.
      catch (Exception e) {
         System.err.println(e);
      }
   }

   /**
    * Creates a Zip archive. If the name of the file passed in is a
    * directory, the directory's contents will be made into a Zip file.
    */
   public static void makeZip(String fileName)
         throws IOException, FileNotFoundException
   {
      File file = new File(fileName);
      zos = new ZipOutputStream(new FileOutputStream(file + ".zip"));
      //Call recursion.
      recurseFiles(file);
      //We are done adding entries to the zip archive,
      //so close the Zip output stream.
      zos.close();
   }

   /**
    * Recurses down a directory and its subdirectories to look for
    * files to add to the Zip. If the current file being looked at
    * is not a directory, the method adds it to the Zip file.
    */
   private static void recurseFiles(File file)
```

```
      throws IOException, FileNotFoundException
   {
      if (file.isDirectory()) {
         //Create an array with all of the files and subdirectories
         //of the current directory.
         String[] fileNames = file.list();
         if (fileNames != null) {
            //Recursively add each array entry to make sure that we get
            //subdirectories as well as normal files in the directory.
            for (int i=0; i<fileNames.length; i++)  {
               recurseFiles(new File(file, fileNames[i]));
            }
         }
      }
      //Otherwise, a file so add it as an entry to the Zip file.
      else {
         byte[] buf = new byte[1024];
         int len;
         //Create a new Zip entry with the file's name.
         ZipEntry zipEntry = new ZipEntry(file.toString());
         //Create a buffered input stream out of the file
         //we're trying to add into the Zip archive.
         FileInputStream fin = new FileInputStream(file);
         BufferedInputStream in = new BufferedInputStream(fin);
         zos.putNextEntry(zipEntry);
         //Read bytes from the file and write into the Zip archive.
         while ((len = in.read(buf)) >= 0) {
            zos.write(buf, 0, len);
         }
         //Close the input stream.
         in.close();
         //Close this entry in the Zip stream.
         zos.closeEntry();
      }
   }
}
```

For every file we add to the archive, we must create a Zip entry, add it to the ZipOutputStream, and then close that entry:

```
//Create a ZipOutputStream named zos
ZipEntry zipEntry = new ZipEntry("Name of a File");
zos.putNextEntry(zipEntry);
//Write data to zos
. . .
zos.closeEntry();
```

All other aspects of creating Zip files are very similar to the GZip program described above, so we will forgo a detailed explanation. However, there are many options associated with Zip streams that are not covered in the sample program. You should consult the Java API documents [4] to explore all of them.

Wrapping It All Up

Both of the example programs deal with creating Zip and GZip archives rather than reading them in. Still, you should now be armed with enough basic knowledge to explore all aspects of compressed streams in Java yourself. Happy coding!

References

1
Sun Microsystems, The Java Tutorial. **http://thejavatutorial.com/**.

2
GZip Homepge. **http://www.gzip.org/**.

3
Ice.com, Java Tar Package -- **http://www.ice.com/java/tar/index.shtml**.

4
Sun Microsystems, Java Platform 1.2 API Specification, 1999.

Class Hierarchy image by Bill Lynch.

Biography

Matt Tucker is a senior majoring in Computer Science at the University of Iowa. He held a summer internship in the Java Division of Sun Microsystems and in his spare time runs **CoolServlets.com**, a site that gives away open-source Java servlets. Matt thinks that Java is pretty neat and finds it strange to write in third-person.