Java Programming 2 – Lecture #15 – Jeremy.Singer@glasgow.ac.uk

Persistent Data

Data stored in RAM is volatile – it disappears when the virtual machine quits, or when someone pulls the power plug. Persistent data may be stored as files in a filesystem or using an alternative backing store abstraction (e.g. a database). In this lecture, we will concentrate on reading data from input files.

Symmetry in the Libraries

The major Java library for file handling is the `java.io` package. Advanced features are available in the `java.nio` package. The basic abstraction for input and output is the *stream*, which is an ordered sequence of data (raw bytes or characters). For input, the `InputStream` is the basic abstract class. For output, there is a corresponding `OutputStream`. Concrete input classes include `BufferedReader` and `FileReader`. Concrete output classes include `BufferedWriter` and `FileWriter`.

The simple source code example below takes a single filename argument and counts how many bytes of data the file contains. Note that the abstract method `InputStream.read()` is overridden by subclasses.

```
public class FileSize {
  public static void main(String [] args) {
    InputStream in = new FileInputStream(args[0]);
    int total = 0;
    while (in.read() != -1) {
      total++;
    }
    System.out.printf("size of file %s is %d bytes\n", args[0], total);
  }
}
```

Filesystem Operations

Actually there is a simpler way to calculate file size, via the `File` class[1], e.g. in above code.

```
total = new File(args[0]).length();
```

Other standard filesystem interactions (e.g. setting permissions, listing directories, creating, renaming and deleting files) are all supported by methods in the File class[2].

[1] See http://docs.oracle.com/javase/7/docs/api/java/io/File.html
[2] Or by static methods in java.nio.file.Files in Java 7

Reading from a File

We will consider reading data from plain text files in a line-by-line fashion. There are several library classes in Java to support this operation – we will use the `BufferedReader`[3] class. Notice how the *`currentLine`* variable is assigned as a side-effect in the `while` statement condition. Also notice the double checking for `IOException` – first when the `BufferedReader` is constructed and used, second when the `BufferedReader` is closed in the `finally` clause.

```
String currentLine;
BufferedReader br;
try {
  br = new BufferedReader(new FileReader(FILENAME));
  while ((currentLine=br.readLine()) != null) {
    // echo line to standard output
    System.out.println(currentLine);
  }
}
catch (IOException e) {
  e.printStackTrace();
}
finally {
  try {
    if (br != null) {
      br.close();
    }
  }
  catch (IOException ee) {
    ee.printStackTrace()
  }
}
```

Java 7 try-with-resources construct

This double try/catch for `IOException`s is particularly inelegant. Java 7 introduces a new try-with-resources construct[4] as syntactic sugar to achieve the same effect without requiring an explicit finally clause.

Java 7 try-with-resources construct

1) Why is it important to `close()` files when we have finished using them?
2) Different methods have different ways of signaling that we have reached the end of an input file, when we have opened a file for reading. List as many of these different end-of-file cues as you can.

[3] See http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html
[4] See http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html