File IO

Lecture 20

I/O Package Overview

- Package java.io
- Core concept: streams
  - Ordered sequences of data that have a source (for input) or a destination (for output)
- Two major flavors:
  - Byte streams
    - 8 bits at a time, data-based (binary) information
    - Input streams and output streams
  - Character streams
    - 16 bits at a time, text-based information
    - Readers and writers
- See Java API documentation for details

Byte Streams

- Two abstract base classes: InputStream and OutputStream
- InputStream (for reading bytes) defines:
  - An abstract method for reading 1 byte at a time
    `public abstract int read()`
    - Returns next byte value (0-255) or -1 if end-of-stream encountered
    - Concrete input stream overrides this method to provide useful functionality
  - Methods to read an array of bytes or skip a number of bytes
- OutputStream (for writing bytes) defines:
  - An abstract method for writing 1 byte at a time
    `public abstract void write(int b)`
    - Upper 24 bits are ignored
  - Methods to write bytes from a specified byte array
- Close the stream after reading/writing
    `public void close()`
  - Frees up limited operating system resources
- All of these methods can throw IOException

Example 1: Measuring File Size


```
import java.io.*;
class CountBytes {
  public static void main(String[] args)
                        throws IOException {
    InputStream in = new FileInputStream(args[0]);
    int total = 0;
    while (in.read() != -1) {
      total++;
    }
    in.close();
    System.out.println(total + ” bytes”);
  }
}
```

Standard Streams

- Three standard streams for console IO
  - System.in
    - Input from keyboard
  - System.out
    - Output to console
  - System.err
    - Output to error (console by default)
- These streams are byte streams!
  - System.in is an InputStream, the others are PrintStreams (inherit from OutputStream)
  - Would be more logical for these to be character streams not byte streams, but they predate the inclusion of character streams in Java

Example 2: Console Streams


```
import java.io.*;
class TranslateBytes {
  public static void main(String[] args)
                        throws IOException {
  byte from = (byte)args[0].charAt(0);
  byte to = (byte)args[1].charAt(0);
  int x;
  while((x = System.in.read()) != -1)
    System.out.write(x == from ? to : x);
  }
}
```

- If you run “java TranslateBytes b B” and enter text bigboy via the keyboard the output will be: BigBoy

Character Streams

- Two abstract base classes: Reader and Writer
- Similar methods to byte stream counterparts
- Reader abstract class defines:
  - public int read()
    - Returns value in range 0..65535 (or -1)
  - public int read(char[] cbuf)
    - Returns number of characters read
  - public void skip(int n)
- Writer abstract class defines:
  - public void write(int c)
  - public void write(char[] cbuf)
  - public abstract void flush()
    - Ensures previous writes have been sent to destination
    - Useful for buffered streams
- Both classes define:
  - public void close()

Converting Byte/Character Streams

- Conversion streams: InputStreamReader and OutputStreamWriter
  - Subclasses of Reader and Writer respectively
- InputStreamReader

  ```
  public InputStreamReader(InputStream in)
  public InputStreamReader(InputStream in, String encoding)
  ```

  - An encoding is a standard map of characters to bits (eg UTF-16)

  ```
  public int read()
  ```

  - Reads bytes from associated InputStream and converts them to characters using the appropriate encoding for that stream
- OutputStreamWriter

  ```
  public OutputStreamWriter(OutputStream out)
  public OutputStreamWriter(OutputStream out, String enc)
  public void write(int c)
  ```

  - Converts argument to bytes using the appropriate encoding and writes these bytes to its associated OutputStream
- Closing the conversion stream also closes the associated byte stream – may not always desirable

The File Class

- Useful for retrieving information about a file or a directory
  - Represents a *path*, not necessarily an underlying file
  - Does not open/close files or provide file-processing capabilities
- Three constructors

```
public File(String name)
public File(String pathToName, String name)
public File(File directory, String name)
```

- Main methods

```
boolean canRead() /  boolean canWrite()
boolean exists()
boolean isFile() / boolean isDirectory()
String getAbsolutePath() / String getPath()
String getParent()
String getName()
long length()
long lastModified()
```

Working with Files

- A file can be identified in one of three ways
  - A String object (file name)
  - A File object
  - A FileDescriptor object
- Sequential-Access file: read/write at end of stream only
  - FileInputStream, FileOutputStream, FileReader, FileWriter
  - Each file stream type has three constructors
- Random-Access file: read/write at a specified location
  - RandomAccessFile
  - A *file pointer* is used to guide the starting position
  - Not a subclass of any of the four basic IO classes (InputStream, OutputStream, Reader, or Writer)
    - Supports both input and output
    - Supports both bytes and characters

Example: A Random Access File


```
public static void main(String args[]) {
    RandomAccessFile fh1 = null;
    RandomAccessFile fh2 = null;

    try {
      fh1 = new RandomAccessFile(args[0], “r”);
      fh2 = new RandomAccessFile(args[1], “rw”);
    } catch (FileNotFoundException e) {
      . . .
    }

    try {
      int bufsize = (int) (fh1.length())/2;
      byte[] buffer = new byte[bufsize];
      fh1.readFully(buffer, 0, bufsize); //read half of file
      fh2.write(buffer, 0, bufsize);     //write all of array
    } catch (IOException e) {
      . . .
    }
}
```

Efficient IO

- Buffering greatly improves IO performance
- Example: BufferedReader for character input streams

  `public BufferedReader(Reader in)`
  - The buffered stream “wraps” the unbuffered stream
- Example declarations of BufferedReaders
  - An InputStreamReader inside a BufferedReader

    ```
    Reader r = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(r);
    ```
  - A FileReader inside a BufferedReader

    ```
    Reader fr = new FileReader(“fileName”);
    BufferedReader in = new BufferedReader(fr);
    ```
  - Then you can invoke in.readLine() to read from the stream line by line

Example


```
public static void main (String[] args) {
    try {
      Reader fr = new FileReader(args[0]);
      BufferedReader br = new BufferedReader(fr)
      String line = br.readLine();

      while (line != null) {
        System.out.println("Read a line:");
        System.out.println(line);
        line = br.readLine();
      }
      br.close();
    } catch(FileNotFoundException e) {
      System.out.println(“File not found: ” + args[0]);
    } catch(IOException e) {
      System.out.println(“File unreadable: ” + args[0]);
    }
}
```