File I/O in Java

File Basics

- Recall that a file is **block** structured. What does this mean?
- What happens when an application **opens** or **closes** a file?
- Every OS has its own EOF character and, for text files, its own EOL character(s).

Streams

- Java file I/O involves **streams**. You write and read data to streams.
- The purpose of the stream abstraction is to keep program code independent from physical devices.
- Three stream objects are automatically created for every application: **`System.in`**, **`System.out`**, and **`System.err`**.

Types of Streams

- There are 2 kinds of streams
  - byte streams
  - character streams

Character Streams

- Character streams create **text** files.
- These are files designed to be read with a text editor.
- Java automatically converts its internal unicode characters to the local machine representation (ASCII in our case).

Byte Streams

- Byte streams create **binary** files.
- A binary file essentially contains the memory image of the data. That is, it stores bits as they are in memory.
- Binary files are faster to read and write because no translation need take place.
- Binary files, however, cannot be read with a text editor.

Classes

- Java has 6 classes to support stream I/O
- **File**: An object of this class is either a file or a directory.
- **OutputStream**: base class for byte output streams
- **InputStream**: base class for byte input streams

- **Writer**: base class for character output streams.
- **Reader**: base class for character input streams.
- **RandomAccessFile**: provides support for random access to a file.
- Note that the classes **InputStream**, **OutputStream**, **Reader**, and **Writer** are *abstract* classes.

File class

```
File myDir = new File(“C:\\CS311”);

File myFile = new File(“C:\\CS311\\junk.java”);

File myFile = new File(“C:\\CS311”, “junk.java”);

File myFile = new File(myDir, “junk.java”).
```

File methods

- **`exists()`**
- **`isDirectory()`**
- **`isFile()`**
- **`canRead()`**
- **`canWrite()`**
- **`isHidden()`**
- **`getName()`**

- **`getPath()`**
- **`getAbsolutePath()`**
- **`getParent()`**
- **`list()`**
- **`length()`**
- **`renameTo( newPath )`**
- **`delete()`**
- **`mkdir()`**
- **`createNewFile()`**

Reading and Writing a Text File

Reading and Writing Text Files

- To write to a text file, use a PrintWriter
- To read from a text file use
  - InputStreamReader: to read one char at a time
  - BufferedReader: read one line at a time
  - StreamTokenizer: read one word at a time

FileWriter Class

- The **`FileWriter`** class is a convenience class for writing character files.
- One version of the constructor take a **`string`** for a file name, another version takes an object of the **`File`** class.
- Both versions of the constructor above have forms that take an additional **`boolean`**. If **`true`**, the data is *appended* to the file; if **`false`**, the file is overwritten.

PrintWriter

- **`PrintWriter`** is a useful class for making text files because it has methods **`print()`** and **`println()`**
- One version of the constructor takes an **`FileWriter`** object and a **`boolean`**.
- If the boolean it true, then the stream is flushed whenever a **`println()`** is called.

Example

```
Disk =   new PrintWriter(
         new ( FileWriter (
         “my_file.txt” ),
         true);

Disk.println( “Hello World” );
Disk.close();
```

See **`FileWrite.java`**

Reading One Char at a Time

- See **StreamReader.java**
- The **`read()`** method returns an integer
- This integer should be cast to a **`char`**
- A value of -1 indicates the end of the stream has been reached.

Reading One Line at a Time

- See **LineReader.java**
- Use a **`BufferedReader`**
- The **`readLine()`** method returns a **`String`**.
- If the **`String`** is null, then the end of the stream has been reached.

Reading One Word at a Time

- See **WordReader.java**
- Use a **`StreamTokenizer`**
  - **ttype**: an int that contains the type of the current token. Values are TT_EOF, TT_EOL, TT_WORD, TT_NUMBER, or a character.
  - **sval**: String containing the current token if it is a word
  - **nval**: double containing the current token if it is
  - a number

Reading and Writing Binary Files

Reading and Writing Binary Files

- To read and write binary files, use **`DataInputStream`** and **`DataOutputStream`**

DataOutputStream Methods

- **`writeByte( int value )`**
- **`writeBoolean( boolean value )`**
- **`writeChar( int value )`**
- **`writeShort( int value )`**
- **`writeInt( int value )`**
- **`writeLong( long value )`**
- **`writeFloat( float value )`**
- **`writeDouble( double value )`**

String Output

- Writing Strings
  - **`writeBytes( String s ) //for Strings`**
  - **`write( byte[] b, int offset,`**
    `int length )  //partial strings`
  - A **`String`** may be converted to a byte array by using the **`String`** method **`getBytes()`**
  - If you use **`writeChars( String s )`** you will get Unicode characters (2 bytes).

Other Methods

- **`flush()`**
- **`size() //number of bytes written`**
- **`close()`**

- The constructor for this class takes an object of the **`outputStream`** class.

Filter Input Streams

- Derived from the abstract class **`InputStream`**
- Some methods
  - **`read()`** reads single byte of data and returns it as type **int**
  - **`read( byte [] b )`** reads enough data to fill the array or until the end of the steam is reached. Returns the number of bytes read.

- `read( byte [] b,`
  `        int offset,`
  `        int length )`

  reads **`length`** bytes into array **`b`** beginning at position **`b[ offset ]`** returns the number of bytes read.
- **`skip ( long n )`**: reads and discards **`n`** bytes for the stream
- **`markSupported()`**
- **`mark( int limit )`**
- **`reset()`**
- **`close()`**

DataInputStream

- Extends **`FilterInputStream`**.
- The methods in this class are mostly a mirror of the methods in the **`DataOutputStream`** class.
- This class does throw an **`EOFException`** when the end of the stream is found.
- See the example **`BinaryStreamTest.java`** and **`BinaryReadWrite.java`**.

Random Access Files

- A random access file allows you to read and write a file at any point.
- Methods that move the file pointer
  - **`seek( long position )`**
  - **`getFilePointer()`**: returns long
  - **`length()`**: returns long

- Constructor for a random access file
  - the constructor takes two arguments.
  - The first identifies the file
  - The second is “**`rw`**” or “**`r`**”

```
RandomAccessFile F =
new RandomAccessFile( “myFile”, “rw” )
```

- To use a random access file successfully, the data must be broken into fixed size units.
- See **RandomFileTest.java**

Object Streams

- To read and write objects, do the following
  - make the class *serializable* by adding **`implements Serializable`** to the class definition
  - Use the **`ObjectInputStream`** and **`ObjectOutputStream`** classes along with the **`writeObject()`** and **`readObject()`** methods.

Object Streams II

- Class instance variables can be marked as `transient` to avoid having their values written to a file. For example the `next` field in a linked list object or a current time field would normally be `transient`.
- See **ObjectFile.java** for an example.

Random Access with Objects

- A random access file must have each “slot” in the file the same length. This is fine if I only want to read and write a primitive type, but what if I want to read or write an object?
- In this case, I must do my own serialization. I must also make all strings fields a fixed size.
- See **RandomObject.java**