JAVA.IO

What is java.io?

- JAVA represents everything as Objects
- java.io is set of Objects that are an abstraction for streaming system resources such as files
- java.io also defines input and output mechanisms for things other than files

Streams

- Streams are the "fundamental element" of the java.io package
- The simplest streams are the abstract classes InputStream and OutputStream
  - You cannot use them directly
  - They define i/o in terms of bytes

File Streams

- **FileInputStream**

```
FileInputStream Fptr =
            new FileInputStream("/etc/passed");
int x = Fptr.read();
Fptr.close;
```

- **FileOutputStream**

```
FileOutputStream Fptr =
            new FileOutputStream("/tmp/blah");
Fptr.write(64);
Fptr.close;
```

Filtered Streams

- These take a stream as input during the constructor and add functionality:

```
FileInputStream Fptr = new FileInputStream("/etc/passwd");
FilteredInputStream FIS = new FilteredInputStream(Fptr);
int x = FIS.read();
FIS.close();
Fptr.close();
```

- You would never actually use **Filtered**[**In**|**Out**]**putStream** directly... you would use classes that extend it

Buffered[In|Out]putStream

- Buffered Streams cache operations...
    - Devices perform better when working with blocks
    - Use memory as a "buffer" for the i/o
  - Input:
    - A large block of the input is read ahead of time and stored until needed in the buffer memory
    - The stream can be reversed to a previous state provided that the desired state is still in the buffer
  - Output:
    - Writes are not committed immediately
    - Use flush if things need to be committed right away

Checksums

- A checksum is a first order approximation to to the problem of verifying data integrity
- Parity count:

  ```
  0100 0100 1101 0100 :: 1
  0111 1111 0000 0101 :: 0
  ```
- If the receiver gets the new message with a single bit "wrong" we know there was a problem in transit
- If two bits are wrong, we may not detect it

Checked[In|Out]putStream

-
  ```
  Checksum CK = new CheckSum();
  CheckedOutputStream COS =
      new CheckedOutputStream(CK, Fptr);
  COS.write(x);
  ```

-
  ```
  CheckSum CK = new CheckSum
  CheckedInputStream CIS =
      new CheckInputStream(CK, Fptr);
  x = CIS.read();
  if (CK != oldCK) {
    // throw some error
  }
  ```

Message Digests

- Message digests (one way hash functions) are a much better data verification methods
- Message digests are strong
  - Used for "Digital Signatures" (and passwords)
  - SHA1 and MD5 are the most common algorithms
- The idea is that you have some data x, put that through a function f to get y [y = f(x)]
  - You cannot recompute x from y
  - Any change (small or large) in x creates a wildly different (and probabilistically unique) y

Digest[In|Out]putStream

- These work as you would expect...
-
  ```
  MessageDigest md =
     MessageDigest.getInstance("SHA");
  DigestOutputStream DOS = new
     DigestOutputStream(Fptr, md);
  DOS.write(x);
  ```
- Reading is symmetric

Compressed Streams

- **Deflator**[**In**|**Out**]**putStream**
  GZIP[**In**|**Out**]**putStream**
  Zip[**In**|**Out**]**putStream**
- Realtime on-the-fly compression and decompression of all data in the stream
- GZIPOutputStream GOS =

```
GZIPOutputStream GOS =
   new GZIPOutputStream(Fptr);
GOS.write(x);
```

Progress Monitoring

- Puts a GUI object up with a progress bar
- You must specify where you want the GUI object to show up as well as a message
- InputStream in =

```
InputStream in =
   new BufferedInputStream(
      new ProgressMonitorInputStream(
parentComponent,
            "Reading " + fileName,
new FileInputStream(fileName)));
```

Lists of Streams

- **SequenceInputStream** allows you to concatenate streams
- After one stream is completely read, the next one in the list will be read
- You can construct a **SequenceInputStream** using an **Enumeration** or a pair of **InputStream** objects

Bytes, why bytes?

- Streams use bytes as the unit which can be read and written
- Bytes are good for hardware, but not good for software
- We need some abstractions...

Readers and Writers

- These provide the ability to perform input and output using characters (Strings)
  - Readers work with InputStreams
  - Writers work with OutputStreams
- **Reader** and **Writer** are abstract classes
- You will once again need to use extensions of these classes to do useful work

Streams to Readers/Writers

- **InputStreamReader** convers an **InputStream** into a **Reader**
  - InputStreamReader ISR = new InputStreamReader(new FileInputStream("..."));
- **OutputStreamWriter** converts an **OutputStream** into a **Writer**
  - OutputStreamWriter OSW = new OutputStreamWriter(new FileOutputStream("..."));

Buffered I/O

- **BufferedReader** and **BufferedWriter** are equivalent to **BufferedInputStream** and **BufferedOutputStream**
- In addition, **BufferedReader** has the ability to read a whole line of text (stripping the newline character(s)) with **readLine()**
- **BufferedWriter** has a platform independent **newLine()** method for outputting a newline character into the destination stream

Using Strings as Devices

- Sometimes you want to use a **String** rather than a physical device for input and output
- This is particularly useful for debugging
- **StringReader** and **StringWriter** let you read and write to a **String** in the same way that you would to a device
- The **String** can them be examined/modified by hand or by the computer

First Bytes, Then Strings...

- The next step is to be able to work with JAVA Objects as the data for I/O
- The **ObjectOutputStream** and **ObjectInputStream** classes are designed to allow us to do this
- Setup your I/O session as usual
- Use **readObject** / **writeObject** to do I/O
- **Be careful of casting!**

ObjectOutputStream

```
MySpecialObject me = // construct the object
FileOutputStream Fptr = new
   FileOutputStream("/tmp/blah");
ObjectOutputStream OOS = new
   ObjectOutputStream(Fptr);
OOS.writeObject(me);
OOS.close();
Fptr.close();
```

ObjectInputStream

```
MySpecialObject me = null;
FileInputStream Fptr = new
   FileInputStream("/tmp/blah");
ObjectInputStream OOS = new
   ObjectInputStream(Fptr);
me = (MySpecialObject)OOS.readObject();
OOS.close();
Fptr.close();
```

Serialization

- If you want to read/write an object, it needs to implement the **java.io.Serializable** interface
- This interface declares no abstract methods
- **So why have it at all?**
- Example: RMI... **Serializable** objects are passed by value whereas **Remote** objects are passed by reference

One Final Note

Don't forget to try and catch!