Lecture 1

In the previous lecture, we saw that there are three main types of file (sequential/binary, sequential/text and random-access/binary).

We looked at one of these three: sequential files containing (binary) data. We start the lecture by looking at these again (briefly). We give an overview of Java's facilities for writing/reading whole objects.

Then we look at sequential files that contain text.

And we conclude with a brief look at random-access files.

Writing/reading objects: `ObjectInput/OutputStream`

- If reading/writing objects, it is painful to have to write/read each instance variable yourself
- `ObjectOuputStream` and `ObjectInputStream` write & read whole objects at a time
- Any class whose objects are to be written/read must implement the `Serializable` interface, e.g.:

```
public class Country
   implements Serializable
{  // etc.
}
```

- The contents of each instance variable are written/read. If they refer to other objects, these other objects are written/read too (if `Serializable`) in a way that preserves the original structure

Writing/reading objects

- For example, if `c` is a variable that references a `Country` object, we can write the object to file in a single operation:

```
oos = new ObjectOutputStream(
  new BufferedOutputStream(
    new FileOutputStream(file)));
oos.writeObject(c);
```

  All of its instance variables are written to the file.

- Similarly, we can read an object in a single operation:

```
ois = new ObjectInputStream(
  new BufferedInputStream(
    new FileInputStream(file)));
Country c = (Country) ois.readObject();
```


Writing/reading complex objects

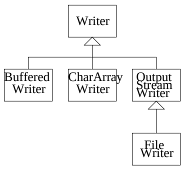
- Assume `v` is a `Vector` and that it contains `Student` objects; assume `Student`s have an instance variable of type `Department`; assume `Vector`, `Student` and `Department` all implement `Serializable`:

```
stream = new ObjectOutputStream(
   new BufferedOutputStream(
      new FileOutputStream(...));
stream.writeObject(v);
...
stream = new ObjectInputStream(
   new BufferedInputStream(
      new FileInputStream(...));
Vector w = (Vector) stream.readObject();
```

  — writes/reads whole `Vector`s and their contents!

- Many more details (e.g. exceptions, static/transient variables,. . .)

Writing text: `Writer`



Writing text to files: `FileWriter`

- You write *characters* to a `FileWriter`:

```
stream = new FileWriter(...);
char c = 'a';
stream.write(c);
```

  This writes one character at a time (using a specified encoding that would be understood by, e.g., text editors)

Writing text to files: `BufferedWriter` & `PrintWriter`

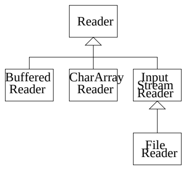
- For efficiency, attach a `BufferedWriter`; for greater convenience, attach a `PrintWriter`, which gives you `print` and `println` methods:

```
stream = new PrintWriter(
 new BufferedWriter(new FileWriter(...)))
stream.println("The answer is " + 42);
```

Using this in `CountryFile`

```
pw = new PrintWriter(
   new BufferedWriter(
      new FileWriter(file)));
pw.println(theCountry.getName() + "|" +
   theCountry.getPopulation() + "|" +
   theCountry.getCurrency());
```

Reading text: `Reader`



Reading text from files: `FileReader`

- You read *characters* from a `FileReader`:

```
stream = new FileReader(...);
int i = stream.read(); // read a char
char c = (char) i;
```


Reading text from files: `BufferedReader`

- For efficiency & convenience, attach a `BufferedReader`:

```
stream = new BufferedReader(
            new FileReader(...));
int i = stream.read(); // read a char
char c = (char) i;
String str = stream.readLine();
```

(If end of file is reached, `read` returns -1 but `readLine` returns `null`)

Using this in `CountryFile`

```
br = new BufferedReader(
      new FileReader(file));
while (true)
{  String str = br.readLine();
   if (str == null)
   {  throw new EOFException();
   }
   String name =
      str.substring(0, str.indexOf('|'));
   int pop = Integer.parseInt(
      str.substring(str.indexOf('|') + 1,
                  str.lastIndexOf('|')));
   String curr = str.substring(
      str.lastIndexOf('|') + 1);
   if (name.equals(theTarget))
   {  return new Country(name, pop, curr);
   }
}
```

(A better way to break up `String`s is to use a `StringTokenizer`)

Random-access files

- Java's `RandomAccessFile`s are binary files

- To open for reading & writing:

```
RandomAccessFile raf =
   new RandomAccessFile(..., "rw");
```

- Java keeps track of your 'position' in the file (the number of bytes from the start of the file). This is where read/writes will occur

- Move the 'pointer' (your position) using `seek`, e.g. to move to mid-point:

```
long address = raf.length() / 2;
raf.seek(address);
```


Random-access files

- Random-access files offer the same methods offered by `DataInput/OutputStream`s, e.g.:

```
int i = raf.readInt();
raf.writeUTF("Some text");
```

- To make effective use of random-access files, it is conventional to keep another structure in main memory, an index to the file

- E.g., for each country name, the index will say where in the file the corresponding data begins. When you want to retrieve the data about a particular country, find the entry in the index and then `seek` and `read` the file

- Efficiency will depend partly on the way you store the index (e.g. a hashtable, an ordered tree)