File Output

So far…

- So far, all of our output has been to `System.out`
  - … using `.print()`, `.println()`, or `.printf()`
- All input has been from `System.in`
  - … using the `Scanner` class to turn typed characters into numbers, words, etc.
- These are both special cases of Java's input/output capabilities.

`java.io`

- The `java.io` package contains classes related to input and output (I/O).
  - to/from the user, files, other code, …
- In particular, there are classes that can be used to read and write data from files.
  - There are separate classes for binary and text data.
  - We will only be using the text file ones.

Writing Text

- The `Writer` class is the (abstract) superclass for character output.
  - It provides methods for sending characters (`char[]` or `String`) to an output stream.
- Then are many subclasses that add more methods to do output in more useful ways.
- An "output stream" can include anything you can send characters to.
  - Typically a file or the screen.

Writing Text Files

- `FileWriter` is a subclass of `Writer`.
  - It connects a character output stream to an actual file.
  - The constructor takes either a `File` object or a string with a filename.
- Constructing creates an empty file.

```
Writer fileout = new
       FileWriter("outfile.txt");
```

Working with `Writer`

- You can use the `write` method to write characters to the end of the file.
  - argument can be a `char`, a `char[]`, or a `String`.
  - eg. `fileout.write(firstName);`
- File must be closed be closed before the program finishes.
  - Ensures that all contents actually make it to the disk.
  - eg. `fileout.close();`

Example

Create `Writer` object by giving filename

```
FileWriter out = new FileWriter("out.txt");

out.write("This is my text file.\n");
for( int i=0; i<20; i++ ) {
    out.write( Integer.toString(i) );
    out.write( '\n' );
}
out.close();
```

Write some strings … and individual characters

Close the file to finish

`IOException`

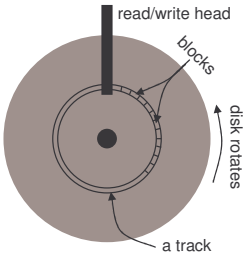
- All of the methods on `Writer` objects can throw (subclasses of) `IOException`.
- `IOException` is a checked exception.
  - So, you either have to catch it or declare your function with `throws IOException`.

How Disk I/O Works

- Disks are much slower than the processor or memory.
  - ~10^5 times: less disk access makes everything faster
- The storage on a disk is divided into "blocks".
  - often 4kB (4096 bytes)
  - can be 1, 2, 4, 8, 16, … kilobytes.
- The operating system assigns files to blocks and keeps track of what files are in what blocks.

Disks

- As the disk spins, the read/write head scans the blocks that pass under it.
- To read a particular block, the head must move to the right track
- … and the disk must spin so the right block is under the head.
- Must read a whole block.



Disk I/O

- When the disk reads/writes a particular block, the head must have moved to the track and the disk is always spinning.
  - So, it's very easy to read/write the next several blocks as the spinning continues.
- Therefore, reading or writing large segments of a file at one time is faster.
  - We can take advantage of this and combine many small reads/writes into one.

Convenient Output

- It's very common in programs to actually want to output small chunks of text at a time.
  - Small parts of the file are produced in a loop.
  - eg. single characters, numbers, lines of text.
- But this is inefficient.
  - It would be possible, to combine many small writes into a few large ones.
  - But, it would be tricky to get right.

Buffered Output

- The `java.io` package contains the `BufferedWriter` class that does this for us.
  - It collects multiple `write()` operations in memory (a "buffer") and actually sends them in a batch.
- The `BufferedWriter` "wraps" another `Writer` object.
  - It takes a bunch of small writes, stores them in memory and sends them to the other `Writer` together.

Buffered Output

- So, you need two `Writer` objects to do this:

- This can be much faster, because there is less disk acccess.
  - several times faster, depending on the size of the original writes.

Using `BufferedWriter`

- The constructor takes the `Writer` that's being wrapped.

```
Writer fileout = new FileWriter("out.txt");
Writer out = new BufferedWriter(fileout);
```

- Or, more compactly:

```
Writer out = new BufferedWriter(
                 new FileWriter("out.txt") );
```

- Then, `out` works like any other Writer.
  - … but faster.

Example

Put a `Writer` in `BufferedWriter`.

```
BufferedWriter out = new BufferedWriter(
             new FileWriter("output.txt") );

out.write("This is my text file.\n");
for( int i=0; i<20; i++ ) {
    out.write( Integer.toString(i) );
    out.write( '\n' );
}
out.close();
```

Importance of Closing

- Some of the output could be buffered by the `BufferedWriter`.
  - The last things you've written might not have made it to disk.
- Calling the `close()` method sends **all** of the data to disk.
  - Exiting the program without it might lose data.

Buffering by the OS

- Even if you're not using `BufferedWriter`, the operating system may buffer writes.
  - Commonly done for speed. Writes are actually done later, when the disk isn't doing anything else.
- Without the OS's buffer, `BufferedWriter` would make even more of a difference.
- You must still explicitly close even non-buffered streams because of these buffers.

Formatted Output

- The basic `Writer` class doesn't have the print methods that we're used to using.
  - The `write()` method only prints strings.
  - It won't convert other data types or use `toString`.
- For that, you need a `PrintWriter` object.
  - another subclass of `Writer`.
  - adds `print()`, `println()`, `printf()` methods.
  - These work like the ones in `System.out`.

`PrintWriter`

- Traditionally, `PrintWriter` worked like `BufferedWriter` and wrapped an existing `Writer`.
- But buffering is still nice, so the declaration becomes:

```
PrintWriter out = new PrintWriter(
    new BufferedWriter(
    new FileWriter("file.txt") ) );
```

`PrintWriter`

- With Java 5.0, this is simplified.
  - The constructor can take a filename.
  - A `BufferedWriter` and `FileWriter` are automatically created.
- e.g.

```
PrintWriter out = new
     PrintWriter("out.txt");
```

Example

Create a `PrintWriter`.

```
PrintWriter out = new
                  PrintWriter("output.txt");

out.println("This is my text file.");
for( int i=0; i<20; i++ ) {
    out.printf( "%d\n", i );
}
out.close();
```

Then use the `print` methods that we're used to from `System.out`.