Prepared by Linan Qiu <lq2137@columbia.edu>

File IO

Reading a File

```
BufferedReader br = new BufferedReader(new FileReader("pikachu.txt"));
String line;

while((line = br.readLine()) != null) {
    System.out.println(line);
    // use StringBuilder if you want to store this data
}

br.close(); // remember to close!
```

Woah that's a huge chunk right there. Let's break it down.

First, what the hell is the difference between `BufferedReader` and `FileReader` and why are we using the two wrapped around each other? Well, think of your FileReader as the water fountain at the ground floor of your dorm. It has access to an external stream (literally a stream). You are living on the 10th floor of the building.

Now you have two choices

- Hold a mug, and every time you need some water, run downstairs and get it. That means you'll have to wear some decent clothes, take a lift, get the water, go back up, and go into your PJs.
- Or… you can grab a big ass bucket, go fetch a lot of water at one go, bring it to your room, and fetch water from that bucket every time you need water using your mug.

Obviously the second is going to be a lot faster since you make only one trip to the ground floor. Same concept here.

The file you're reading resides on the hard disk. The hard disk is slow as f*** compared to the memory of your computer. So you want to minimize the number of times you query the hard disk. So yes, you could just read from `FileReader` directly, but every time you do that, you'll have to access the hard disk once. Instead, by wrapping the `FileReader` in `BufferedReader` you are essentially telling `BufferedReader` to grab data in bigger chunks from the hard disk. Then, you can read from the `BufferedReader`. When the `BufferedReader` is out of stuff (just like your bucket is empty), it will go grab more data. In this way, we minimize the number of times we query the hard disk, saving you precious time.

Then, what about this line `while((line = br.readLine()) != null)`? That seems like a huge chunk. Well, what's happening is really simple. `line = br.readLine()` reads the next line of text (until `\n`) and passes that data to `line`. That's what's happening in the `(line = br.readLine())` bracket. This bracket actually returns a variable (every `=` assignment returns the variable on the left). `line` getes returned, and we test if the newly polished `line` is equal to `null`. What does this mean? Well, `BufferedReader` simply returns null to `readLine()` if there's nothing else to be read. Hence, in the last run, you would have assigned `line` to null. Then we would test for it in the while condition. Then, we would stop reading. So basically it's just a mechanism to read till the end of the file.

Reading from `stdin`

Hang on, what the hell is `stdin`? Basically your keyboard. Remember when you used to like `Scanner s = new Scanner(System.in)`? That `System.in` is basically `stdin` in comp-sci-speak.

So how do we read from your keyboard instead of a file? Well, turns out its pretty much the same.

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String line;

while((line = br.readLine()) != null) {
    System.out.println(line);
}

br.close(); // remember to close!
```

You're simply specifying an `InputStreamReader` that listens to `System.in`, just like you did in `Scanner`.

The rest of the code logic stays the same.

Writing to File

How would you write to a file then?

```
BufferedWriter bw = new BufferedWriter(new FileWriter("file.txt"));
bw.write("I'm sick of being in a pokeball");
bw.close();
```

Pretty much the same stuff. Again, we're wrapping the output stream in a buffer. Just like you won't want to go down to the professor's office to hand in a new sheet of paper every time you finish writing one (you'd probably wait till you're done writing everything), we use a buffer to wrap the output stream.

However, do remember that **you need to close the stream**. I didn't emphasize this too much for the reader because it doesn't face the same issue, but do remember that if you don't close the BufferedWriter, some stuff could still be left in the buffer. What do I mean?

Imagine you're writing homework. You leave pages on your table till you're done with a certain amount, then you hand those in to the professor. Without close, even after you're done writing everything, if the pile on your table isn't high enough, you won't go and hand those in. That's silly isn't it? So closing the `BufferedWriter` forces the writer to empty its buffer, essentially spilling out that last chunk of stuff.