File Paths

A file is a permanent storage of data.
Absolute file name: The full name of the location of the file, with drive letter and path.
Relative file name: Location of the file in relation to the directory the computer is currently opened on.

c:\arjun\helloworld,java Absolute file name	helloworld.java Relative File name when the computer is opened to the "arjun" directory

File Objects

`import java.io.File;` needed.

The file object can be used with other classes for reading from and writing to files.

Ex: set up a file object so you can access a previously existing file in the directory titled "Jason.txt"

```
File f = new File("Jason.txt");
```

Exceptions with File Processing

Many of the classes used will throw certain IOExceptions and require a try-catch block to operate. This can be annoying, and although it's generally considered bad practice, assume that for the most part, your code will run exception free. The compiler will still not let you run the code unless you handle the exceptions in some way, though. So, you can throw an `IOException` on the method in use.

Ex:

```
public static void main(String[] args) throws IOException { }
```

PrintWriter

The PrintWriter class is used to add extra data to a file.

Ex: Create a file called "Anna.txt" and add the numbers 1 through 10 in it, separated by new lines.

At the end of the code, the printwriter object might be closed, otherwise the data might not actually get saved in the file.

```
PrintWriter out = new PrintWriter(new File("Anna.txt"));
for(int i = 0; i < 10; i++) {
      out.println(i);
}
out.close();
```

Scanner

The Scanner class can be used to read input from the terminal or a file.

From terminal:

```
Scanner scan = new Scanner(System.in);
//scanning from a java file
Scanner scan = new Scanner(new File("filenamehere.java"));
```

The scanner class has some useful methods:
`next()` -- returns the next sequence of characters in the input stream up until a whitespace.
`nextInt()` -- returns the next integer in the input stream
`nextDouble()` -- returns the next double in the input stream
`nextLine()` -- returns the sequence of characters in the input stream until a newline.
The following return true if there if the specified type exists in the remaining part of the input stream, false otherwise:

```
boolean hasNextInt()
boolean hasNextDouble()
boolean hasNext()
boolean hasNextLine()
```

"White space" and "new line" are just simplifications of what is actually going on. The scanner class will separate the input stream based on a **delimiter** which is by default all white space characters. `useDelimiter(String a)` will change the delimiter.

```
public static void main(String args[]) throws
IOException{
        Scanner scan = new Scanner(new
File("hi.dat"));
        int i = scan.nextInt();
        String x = scan.nextLine();
        System.out.println(i + " x:" + x );
        while(scan.hasNextDouble()) {
         System.out.println(scan.nextDouble());
        }
    }
```

```
hi.dat:
5
1 2 3.2
```

```
Output:
5 x:
1.0
2.0
3.2
```

The variable i gets the value 5. The variable x gets the newline separator (the scanner continues to read past 5 until the end of the line). Then, a while loop prints out all doubles on the next line.

String Tokenizer

A StringTokenizer object will take a string and separate it into tokens (based on the whitespace characters).

```
StringTokenizer t = new StringTokenizer("hi my name is Vihan");
while(t.hasMoreTokens()) { System.out.println(t.nextToken()); }
Output:
hi
my
name
is
Vihan
```

The `nextToken()` method returns a string, so to convert it to a primitive data type, use the wrapper class converters. Ex:

```
int a = Integer.parseInt(t.nextToken());
```

```
double b = Double.parseDouble(t.nextToken());
```

BufferedReader

The BufferedReader has the same uses as Scanner, but it is notably faster. Unfortunately it does not have as many methods so it needs to be used in conjuction with the StringTokenizer.
BufferedReader, when reading from the terminal, must be instantiated in the following way:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

Instantiate in this way to read from a file called, for example, “sprime.in”.

```
BufferedReader in = new BufferedReader(new FileReader("sprime.in"));
```

BufferedReader methods:
`readLine()` -- reads a line of text
`ready()` -- boolean for if the BufferedReader object ready to be used again (is there more input to read)

Ex: Write a program that accepts an unknown number of lines from the terminal and for each line it returns the sum of the integers on that line.

```
import java.util.*;
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
        while(in.ready()) {
            StringTokenizer t = new StringTokenizer(in.readLine());
            int total = 0;
            while(t.hasMoreTokens()) {
                total += Integer.parseInt(t.nextToken());
            }
            System.out.println(total);
        }
    }
}
```

BufferedWriter

This is a class that makes for fast creation of files and adding data to a file. It helps the PrintWriter go faster.

```
PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(new File("x.out"))));
```

Again, use the `close()` method at the end of the code to ensure the data actually transfers.