**Files Input and Output (I/O) and Streams**

Almost all programs need to get data either from the outside world or to put data into the outside world. One way is to store data permanently in files that can be read and written from the program. Other ways are to connect to databases or to connect to web pages and servers through URLs, but those will not be discussed in this course.

In Java, all IO has the unifying concept that it happens through Streams. A Stream is a sequence of data, either binary data or character data, that can be read by the program from any data source and written by the program to data destination. The programs uses these Streams one item at a time.

Java has a number of Streams, starting from low level streams that use bytes and higher level streams that use characters. The latter streams are called Readers and Writers and are the ones that we will use for files. In addition, there are streams that buffer the data as it is either put into the stream or taken out of the stream, and other streams with other performance characteristics. These streams are built on top of each other to add functionality or properties to the process of reading and writing data.

Standardly for reading and writing files, we will use a BufferedReader on top of the FileReader class and a BufferedWriter on top of a FileWriter class. These streams have methods for reading and writing either one character at a time or one line at a time.

An example for reading a file one line at a time was given in the Exception document.

To use a Reader for input, create a FileReader, as in that example, or a BufferedReader:
Example:

```
        BufferedReader inputStream = new BufferedReader(new FileReader ("xanadu.txt");
```

You may put just the name of the file, if it is located in the "current directory" of the program. For NetBeans programs, this directory is the NetBeans project directory. Otherwise, you may put the path of the file.

The inputStream has methods like read(), to read one character, or readLine() to read a line from the file.

```
Example:    int ch; String text;
            ch = inputStream.read();              // returns the int encoding of the character
            text = inputStream.readLine();        // returns a string
```

A BufferedWriter can be created similarly to a read, and it will have method write. There is not an equivalent writeLine method; instead you can either put newline characters in the string or use the newLine method. The newLine method writes a line separator that is appropriate for your system (whether it's a Mac or a PC).

Note that if a file with that name is not already in the current directory, it will be created. If there is already a file with that name, it will be overwritten.

Example:

```
	BufferedWriter outStream = new BufferedWriter(new FileWriter (“xanaduwords.txt”);

	outStream.write ( text );
	out.Stream.newLine ( );
```

Data is typically read or written to a file in a loop. While reading, if there is no more data, the next character from a read() is -1, and the next line from a readLine() is null.

**Scanners**

When we read or write data to a file, we commonly don’t want to deal with only characters, and it is difficult to work with a whole line as it comes in as a String that may represent any number of Strings, ints or doubles. For example, suppose that each line of a file represents an employee:

```
	Harry Hacker,M,50000,x4567,200 Hinds Hall
```

There are several classes that can help to break up the whole line of text from the file in order to convert it to other data types. These classes include StringTokenizer and StreamTokenizer, but we will use the Scanner class.

The Scanner class can be put on top of an input stream. It will break the string making up the line from the file according to a “delimiter” character. The default is whitespace, but it can also be set to characters like “,” or “|”.

Example (from the Sun Java Tutorial):

Suppose that we have a file with the text of a verse of the poem Xanadu, which starts

    In Xanadu did Kubla Khan
    A stately pleasure-dome decree:
    …

Here is an example of reading from a file of text and separating the words by white space:

```
import java.io.*;
import java.util.Scanner;

public class ScanXan  {
	public static void main(String[] args)  {
		Scanner s = null;
		try {
			s = new Scanner(new BufferedReader(new FileReader(“xanadu.txt”)));

			while( s.hasNext())  {
				System.out.println(s.next());
			}
```

```
                    }
                    catch (IOException e)   {
                              System.out.println(e.getMessage());
                    }
                    finally  {
                              if (s != null)  {  s.close(); }
                    }
          }
}
```

Here we use the methods hasNext() to test whether there is any more input and next() to get that input. There are additional methods for each type to test if the next item could be converted to that type and to get it.

```
Examples:     // assume that s is a Scanner
              s.hasNext()                    // returns a boolean
              s.next()                       // returns a String up to the next delimiter
              s.hasNextInt()                 // returns a boolean
              s.nextInt()                    // returns an int that was converted
              s.hasNextDouble()              // returns a boolean
              s.nextDouble()                 // returns a double
```

To read files that have the data separated by characters other than whitespace, the useDelimiter method passes a string regular expression that is used to match characters to separate the items in the file.

For example, suppose that we have a comma-separated file, often with file extension .csv as generated by a spread sheet program or other applications:

```
          Harry Hacker,M,50000,x4567,200 Hinds Hall
```

For this, we want to have the comma and any end-of-line characters to be delimiters and not a simple space. For example, we want "Harry Hacker" to be one String returned by s.next(). Here is a delimiter regular expression for this case:

```
          s.useDelimiter(",|(\\n|\\r)+");
```

We won't go into regular expressions in this class, but if you're interested, this regular expression can be expressed in English as:

Match either a "," or a sequence of 1 or more characters
consisting of "\n" (newline) or "\r" (carriage return).

Omitted from the matching definition of a delimiter are any occurrences of "\s" (space) or "\t| (tab).

Note that the Scanner class can be used on Streams other than those coming from a file or on a String.

**Formatting**

There is no equivalent to the Scanner class for writing out different items of output. Instead, we use the various formatting functions that we already know to convert an int or a double to a String. Here are some examples.

Examples:
```
int number;
double amount;

String.valueOf (number)
String.valueOf (amount)

// suppose that df has been created as a DecimalFormat
df.format (amount)

// suppose that cf has been created as a CurrencyFormat
cf.format (amount)
```

**File Chooser**

A nice way to allow the user to specify an input file is to use a JFileChooser. The showOpenDialog method will pop up a file chooser dialog window in which the user can select a file from anywhere on their hard drive. Here is an example that displays the path to the file that the user selected.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
        JFileChooser fileChooser = new JFileChooser();
      if( fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
      {
        File file = fileChooser.getSelectedFile();
        String inputFilePath = file.getAbsolutePath().toString();
        jTextField1.setText(inputFilePath);
      }
   }
```

There is also a File Chooser method to select output files, where the output file does not have to exist already.

```
        fileChooser.showSaveDialog(this);
```

This method is used in the same way as the showOpenDialog method.