# Processing Delimited Text Files – Supplemental Information

## Reading and parsing CSV files

In this assignment you will need to open and read data from a CSV file. CSV stands for Comma Separated Values. CSV files are text files where for each record, data is delimited by a comma character (‘,’). Each record is ended with a new-line character (‘\n’). There are a number of advantages to this. First, it is a text file, and therefore human readable and easily editable. Secondly, CSV files can be opened and edited by many spreadsheet applications such as Excel. It is also easy to read these files, and parse the data in each record since data is separated by a known character, a comma.

There are a couple of ways to read in record data from a CSV file. You may use either method in your assignment, but are responsible for understanding both.

**Method 1: Using Scanner’s useDelimiter method**

The lecture covering Character-based File I/O reads in a file that is delimited by semi-colons (‘;’). To read this file an entire line, which represents one record, is read in one at a time by a Scanner associated with the file. The read String is then given to a new Scanner object:

String record = new Scanner(readLine);

Scanner’s setDelimiter method was then used to set which characters should be used to separate data when reading from the String. In the case of a CSV file, this would be a comma:

record.setDelimiter(“,”);

From this point, the various Scanner methods such as nextInt, nextDouble, etc, could be used to parse the record piece by piece. The programmer would need to know the types that were expected for each piece of data in the record in order to call upon the appropriate method. If the parsed data does not conform to the Scanner method being used, an InputMismatchException would occur. For example, if nextInt were called when the next token were actually a double, this Exception would occur.

**Method 2: Using String’s split method**

The second method begins similar to the first. A Scanner is created to read entire records, one at a time. The parsing of the individual pieces of data is done by using String’s split method. The split method can be called upon by a String. The argument to split is a String listing the delimiters to divide the calling String upon. This will return an array of Strings, where each element is a token in the data, in the order it appeared.

For example:

String record = “abc;123;DEF;45.6”;

String[] tokens = record.split(“;”);

The tokens array would store the Strings: “abc”, “123”, “DEF”, and “45.6” in indices 0 through 3, respectively.

Be aware that these tokens are stored as Strings, and not numerical types.

## Parsing Numbers from Strings

If there is a String which contains characters that represent a numerical value, that value can be extracted by using static methods found in the Primitive Wrappers. Each of the Numerical Primitive Wrappers have a parse method that when given a String, will return the associated Primitive Wrapper type.

For example:

String sDouble = “12.34”;

Double d = Double.parseDouble(sDouble);

The call to parseDouble through the Double class will attempt to parse the String into a Double value, and return it. In this case, the reference to the new object is stored in the Double reference, “d”. If the String that is passed cannot be correctly parsed, it will throw a NumberFormatException.

For example:

Integer.parseInt(“1.2”);

Since “1.2” is not a valid Integer value, this call would fail, throwing a NumberFormatException.

When using Method 2 to read CSV files, since the data is split into Strings, these parsing methods will need to be used to extract the numerical data. Note that with this method, as well as the first, the programmer must know the order, and types of each piece of data in a record. For Method 1, to call upon the correct Scanner method, and for Method 2, to parse the String into the correct type.

## Clean Data Files

Be wary when processing data sets. In both methods of reading CSV files, it is noted that the type and order of data must be known to accurately (and easily) read and parse the file. Many sets of data are not “clean”, meaning that the format may not be consistent throughout all records. This could be due to additional or missing columns for some records, blank data, extra blank lines, a delimiter being used in a piece of data, and any number of other possibilities.

When writing an application to process data files, ensure that your input files are clean by fixing any present issues if possible. If your application can assume the data is clean, it will make writing the code much easier. This may not always be possible if you do not have control over the input files. In this case, Exception handling code may be required to resolve potential issues.

## Javadoc for Type Parameters and Thrown Exceptions

If a method declares a type parameter, it should be documented in the Javadoc header. Type parameters are treated similar to other parameters in Javadoc. If a generic method creates a type parameter:

```
<E extends Comparable<E>>
```

it should have a corresponding tag:

```
@param <E> Description of type here
```

Likewise, any Exceptions that a method may throw should also be noted in the Javadocs. If a method were to throw a NullPointerException, for example, the documentation should contain the tag:

```
@throws NullPointerException Description explaining why the Exception is thrown
```

## JUnit Test for Exceptions

It is possible to create JUnit tests to ensure that a method throws expected Exceptions when called under circumstances that should yield that Exception. In this assignment, there are a few methods that will throw Exceptions, and this will be tested, so students should test for this as well.

Example:

Math's floorDiv method takes two values and divides them, resulting in the floor of the division. This method throws an ArithmeticException if the divisor is 0. The following JUnit test will check that the method is throwing an ArithmeticException under that circumstance.

```
//Create an ExpectedException Rule, this rule can be used
//for all Exception tests
@Rule
public ExpectedException thrown = ExpectedException.none();

//Test to ensure Math.floorDiv throws an ArithmeticException when given
//a zero denominator
@Test
public void floorDivAETest() throws ArithmeticException //Test throws the Exception
{
	//Set the Rule's expected Exception, the one we are expecting to be thrown
	thrown.expected(ArithmeticException.class);
	//Invocation that should throw the Exception
	Math.floorDiv(5, 0);
}
```

This test will pass as long as the currently expected Exception occurs within the test. If the Exception is not thrown by anything in the test, it will fail