Programming Design and Implementation

Lecture 7: Input / Output

Updated: 3rd May, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

"Zero" Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Exceptions - Reminder

- Error handling is a necessary task, but how do you do it elegantly?
  - Errors aren't "normal", you don't make a system that *expects* errors
  - But you **must** handle error situations
  - One solution: return an error code (see UCP `COMP1000`)
- Object-Oriented languages (such as Java) solve error handling with **exceptions**
  - An independent "return path" designed specifically for notifying the caller of an exceptional situation (error)
  - On an error, a method "throws" an exception
  - The calling method can "catch" the exception
    - If the caller doesn't catch it, the exception is thrown to the next highest caller
    - If no one catches it, the exception causes the program to crash

Exceptions - Reminder (2)

- Java only lets objects of type **`Exception`** or its descendants to be thrown
  - Java has a range of classes descending (inheriting, extends) from **`Exception`**
    - e.g., **`IllegalArgumentException`** and **`ArrayIndexOutOfBoundsException`**
  - You may define your own **`Exception`** class, as long as it inherits from **`Exception`** (or one of its subclasses)
  - This will be covered in DSA (`COMP1002`)

Catching Exceptions

- Exceptions from different methods in different objects are often all caught at the one place in the calling method
- Somewhere close to **`main`**
  - Convenient: all error handling happens in one place
- Most languages use `try`, `catch()`, `finally` blocks
  - `try`: Define the set of statements whose exceptions will all be handled by the `catch` block associated with this `try`
  - `catch()`: Processing to do if an exception is thrown in the `try`
  - `finally`: Processing to always do, regardless of whether an exception occurs or not
    - Good for cleanup, e.g., closing files
    - This block is **optional** and executed *after* the `try` and `catch` blocks

Input With Exception Handling

```
public static double realInput(String prompt, double lower, double upper)
{
   double num;
   Scanner sc = new Scanner(System.in);
   String outputPrompt = prompt;
   do
   {
      try
      {
         System.out.print(outputPrompt + " between " + lower " and " +
                          upper);
         num = sc.nextDouble();
      }
      catch(InputMismatchException e)
      {
         sc.next();     //clear the buffer of the problematic input
         num = lower - 1.0; //set num to invalid so we don't exit
      }
      outputPrompt = "ERROR: please enter a valid value \n" + prompt;
   } while ((num < lower) || (num > upper));
   return num;
}
```

Why Files?

- RAM (**R**andom **A**ccess **M**emory) is volatile and private to an application, so it's not a good match for the following purposes:
  - Storing application data long-term (between runs)
  - Sharing information between applications
  - Reading in bulk data provided by a user
- In contrast, files stored on disk are (semi)permanent, can be shared between applications and can be manipulated by the user outside the application

File Input/Output

- Unlike RAM, files are effectively an input to and/or an output from the application
  - Hence the term File I/O, for Input/Output
- Three basic steps in File I/O: (these apply to any platform)
  - Open the file
  - Read data from a file and/or write data to a file
  - Close the file

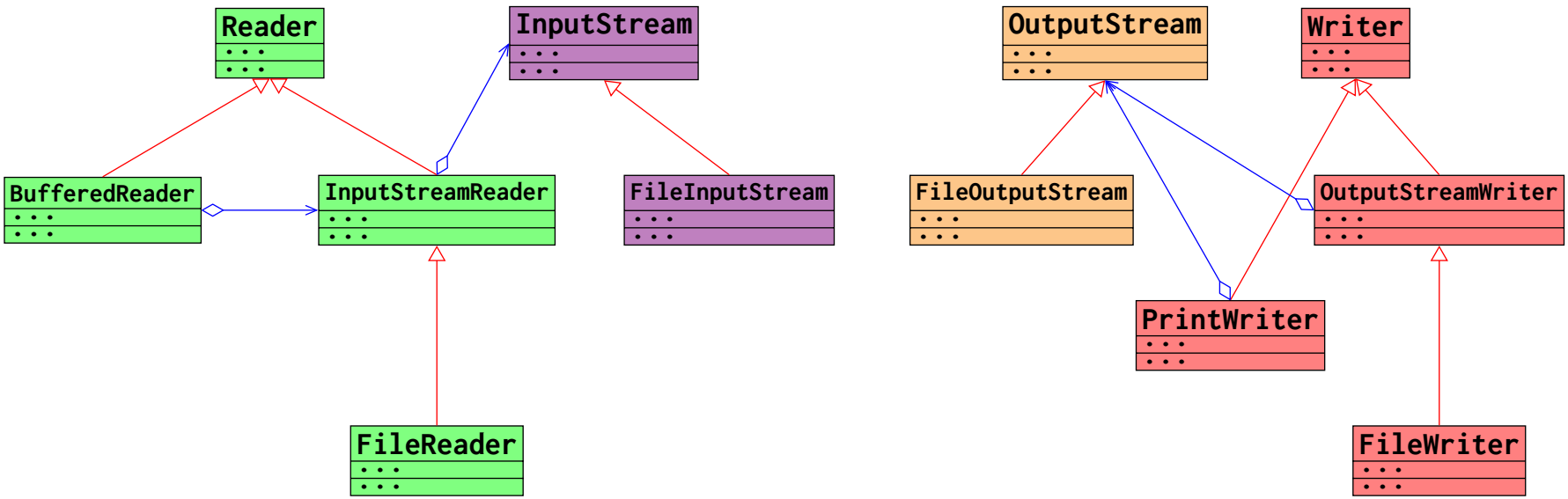
File I/O in Java

- Java encapsulates File I/O in a set of classes
  - All File I/O classes support the three basic steps
  - Different classses are used for manipulating different kinds of files or to perform I/O in different ways
    - Depends on what you need
  - Other kinds of I/O are also part of the same set of classes
    - e.g., Keyboard input, screen output, network I/O, device I/O, etc.
    - This was done to unify the handling of all the different I/O streams into a single consistent programming interface

Java I/O Classes

- The I/O classes can be split into three groups:
  - Classes that represent an I/O stream (e.g., file, device, etc.)
  - Classes to read from a stream
  - Classes to write to a stream
- There are specialised versions of the basic classes for each I/O type (file, network, device, etc.)
- We will concentrate on the file I/O classs
  - I am just making you aware of the broader scope to help explain why there are so many classes involved

Java I/O Class Hierachy



- **FileReader** and **FileWriter** are more for convenience and we wont be talking about them, or using them

Steps in Java for Reading a File

- 1. Create a stream object for a file
  - `FileInputStream`
- 2. Create an object that can read that stream
  - `InputStreamReader`
- 3. Read and process data from a file
- 4. Close the `FileInputStream`
- Note: A `FileReader` combines the first two steps into one

Steps in Java for Writing a File

- Basically the same as reading
- 1. Create a stream object for a file
  - `FileOutputStream`
- 2. Create an object that can write to that stream
  - `OutputStreamWriter` or `PrintWriter`
- 3. Write data to the file
- 4. Close the `FileOutputStream`
- Note: Again, `FileWriter` combines the first two steps into one

Efficient File I/O

- If you look at the methods for **`InputStreamReader`**, you'll see that it has two **`read()`** methods
  - One for reading a `byte` at a time
    - This returns `-1` if no more bytes to read (i.e., end-of-file reached)
  - A second for reading an array of `byte`'s at a time
- Since it's rare that we ever want just a single `byte` of data, the first method seems less useful
- However, when trying to read in text data there are conceptual difficulties with reading chunks at a time

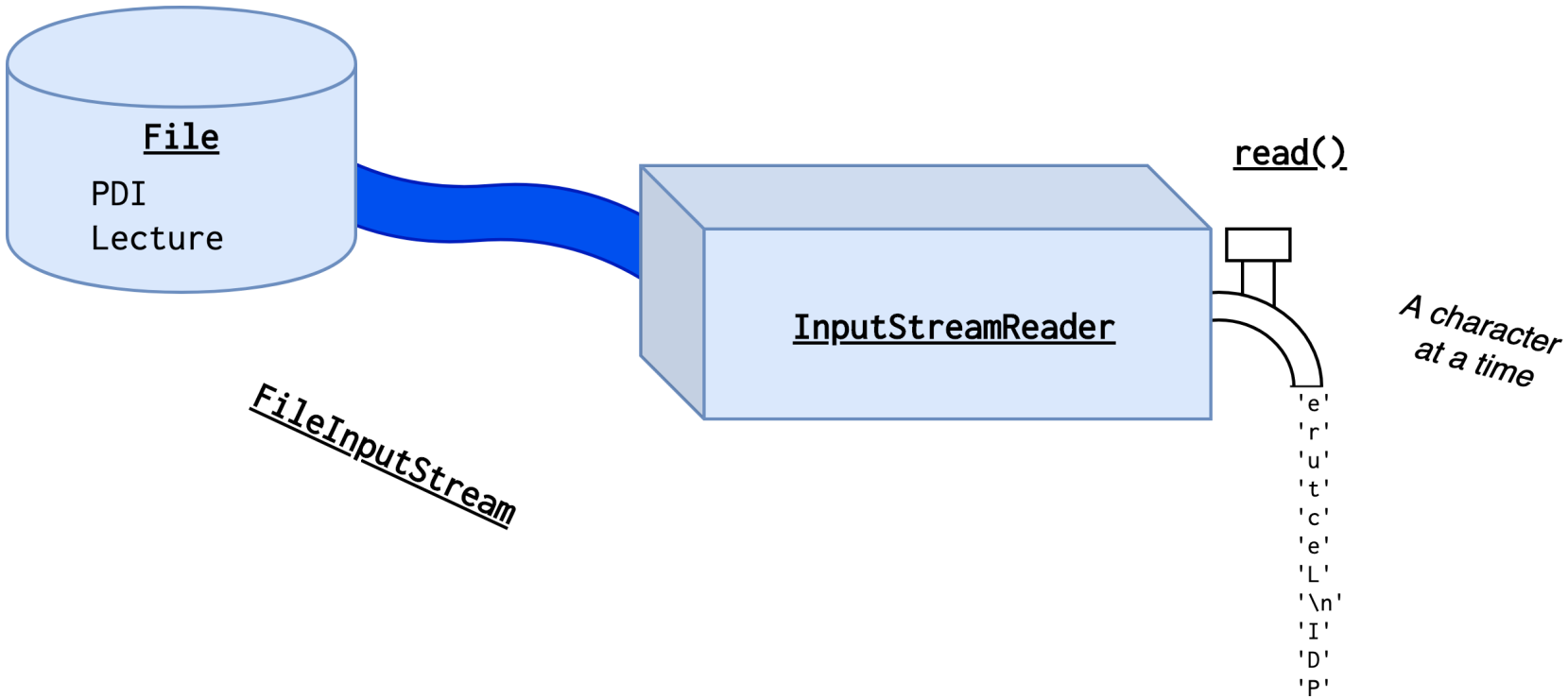
Text vs. Binary Data

- Data files generally fall into two broad categories:
  - Files containing **`text`** data (relatively unstructured)
  - Files containing **`binary`** data (everything else)
- Binary data is highly structured
  - e.g., images, databases, executable files, etc
  - There is a lot of prior knowledge on precisely where information exists in the file and how large blocks are
- Text data is unstructured, so it is usually impossible to know beforehand how many bytes to read
  - e.g., how many characters in an arbitrary line of text?

Efficient File I/O Processing

- So reading blocks of *N* `byte`'s from a file is okay
  - You will know beforehand how many bytes are needed
- But when reading text, you have to constantly be ready for the end of data since you can't predict it
  - e.g., end of word, end of line, or end of file
- One approach is to read data in a `byte` at a time, and check each byte for end-of-**X** (' ', '**n**' or `-1`)
  - The problem is that this will be very slow since hard disks are fastest at reading blocks of data at a time
    - Its like filling a bucket one drop at a time vs. opening the tap and letting the water flow freely into the bucket

InputStreamReader

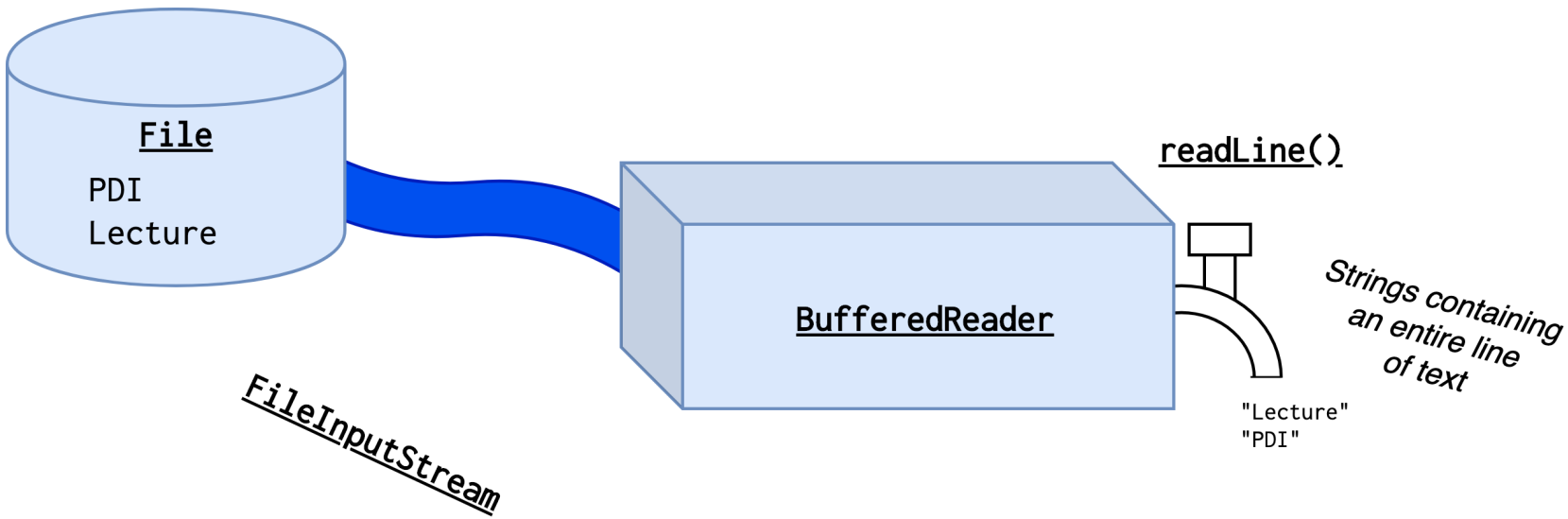


- Note: **read()** returns -1 when end-of-file (i.e., no more data) occurs
  - Also, note the '**\n**' (newline) is returned; just like any other character in the file

BufferedReader

- **BufferedReader** reads in chunks of data, buffering it in memory to search for end-of-**X**
  - In particular, the **readLine**() method looks for the end-of-line
  - The idea is to read in ('buffer') chunk of 1024 bytes, then search these bytes in RAM one at a time for the '**\n**'
    - '**\n**' = newline character, marking the end of the line in text
    - Since RAM is much faster than disk, this is much more efficient
  - When a line is found it is extracted and returned, but the rest of the buffer is kept in memory
    - Then for subsequent **readLine**() calls, it first checks the buffer for the '**\n**' before reading another 1024-byte chunk from the file
  - Note: you create a **BufferedReader** from another **Reader**

BufferedReader (2)



- Note: **readLine()** does **not** include the '**\n**' in the returned lines and returns `null` when end-of-file occurs

File Reading: One Line at a Time - Pseudocode

```
SUBMODULE: readFile
IMPORT: filename (String)
EXPORT: none
    theFile := OPENFILE filename
    lineNum := 0
    INPUT line FROM theFile

    WHILE NOT (theFile = EOF)    // EOF = end of file.
                                 // Detecting this is language-specific
        lineNum := lineNum + 1
        processLine(line)        // Whatever line processing is required

        INPUT line FROM theFile  // Read the next line
    ENDWHILE

    CLOSEFILE theFile            // Close the file
END readFile
```

File Reading: One Line at a Time - Java

```
public void readFile(String inFileName)
{
    FileInputStream fileStream = null;
    InputStreamReader rdr;
    BufferedReader bufRdr;
    int lineNum;
    String line;
    try
    {
        fileStream = new FileInputStream(inFileName);
        rdr = new InputStreamReader(fileStrm);
        bufRdr = new BufferedReader(rdr);
        lineNum = 0;
        line = bufRdr.readLine();
        while(line != null)
        {
            lineNum++;
            processLine(line);
            line = bufRdr.readLine();
        }
        ...
```

File Reading: One Line at a Time - Java (2)

```
            ...

            fileStream.close();
        }
        catch(IOException e)
        {
            if(fileStream != null)
            {
                try
                {
                    fileStream.close();
                }
                catch(IOException ex2)
                { }
            }
            System.out.println("Error in fileProcessing: " +
                                e.getMessage());
        }
}
```

Notes on File Handling

- Make sure you close the file as **soon** as possible
  - The operating system must track what files are open
    - The OS remembers where you were in the file, etc.
  - The resources available for this tracking are limited
    - Run out and the OS will terminate your program
  - Thus don't leave files open – clean them up early
    - Java doesn't free objects immediately - it waits for the garbage collector, so **always** explicitly `close()` a file once finished with it

File Handling - Exceptions

- You **must** handle **IOException**
  - **IOException**'s must be caught
    - Checked **Exception**
    - The compiler will complain otherwise
    - This also forces us to do the `try{} catch(){}` around **close**
  - Note: that you can add a `throws` **IOException** clause to the method signature (method contract) to avoid having to catch **IOException**'s
    - It just means that now the calling method must catch them.

Multiple catch Clauses

- Previous example caught **all** IO Exceptions
- What if we want to handle various exceptions in a different way?
  - e.g., if a file doesn't exist, ask the user for a different file name, but if anything else goes wrong with the file, terminate the program.
- Every `try` clause **must** be followed by one or more `catch` clauses
  - Order is important, it will attempt to catch the first **`Exception`** written first, if that is a less specific type then it may not catch a more precise type later
  - This concept will be explained more when you cover Inheritance in DSA (`COMP1002`)

File Reading: One Line at a Time - Java

```java
public void readFileMultipleException() throws IOException
{  // Why would it be bad to write "throws Exeception" here?
    Scanner sc = new Scanner(System.in);
    String inFileName;  // File Code omitted - see slide 22
    boolean noFile;
    do
    {
        noFile = true;
        inFileName = sc.next();
        try
        {
            while(line != null)
            {
                processLine(line);
            }
            noFile = false;
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Couldn't find your file " + e.getMessage() + " Try again!");
        }
        catch(IOException e)
        {
            throw e;
        }
        catch(Exception e)
        {
            throw new Exception("I am a bad programmer: ", e);
        }
    } while(noFile);
}
```

Parsing

- When dealing with text, it's often necessary to take it apart and organise it ready for processing/storage
- This is called **parsing** - to determine and extract the structure of a piece of text
  - The word originally comes from syntax analysis of written language
- Examples of where parsing is needed:
  - Natural language processing (e.g., spelling/grammar checks)
  - Building Web search indexes
  - Compilers - must parse the code to detect statements and variables

Tokenizing

- **Tokenizing** is the first step in parsing, the process of breaking up a stream of text into basic elements
  - We will use the US spelling (with a 'z') to avoid confusion
- These elements are called tokens, and what they are depends on what the application is parsing
  - e.g., Single words, entire lines, equation terms, etc.
- Tokens are broken up by searching for character(s) that delimit the boundary of a token
  - e.g., lines are separated by a '\n' newline character
  - e.g., words are separated by spaces, commas and periods
  - e.g., equation operands are separated by operators `+ / * -`

Tokenizing with Java

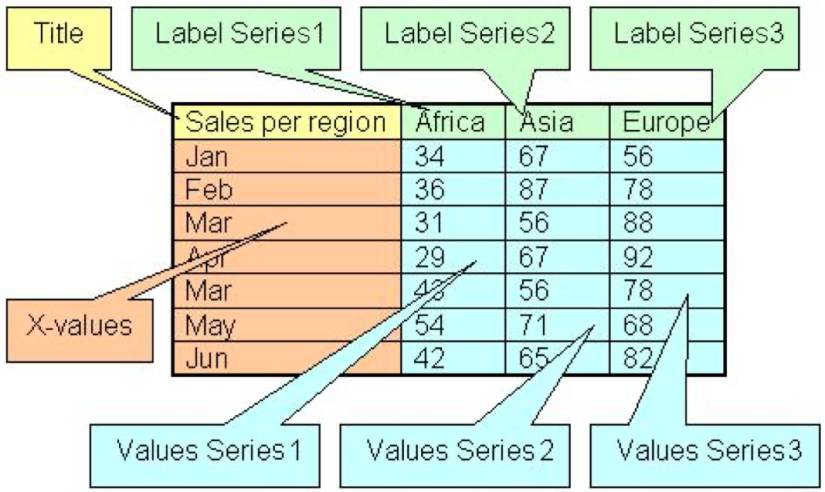
- Java provides two classes to assist in tokenizing:
  - **`StringTokenizer`**
  - **`StreamTokenizer`**
  - They both do a similar job, except one works on **`String`**'s while the other works on **`Reader`**s
    - For this unit we will focus on **`split(String regex)`** (below)
- Java also provides the **`split(String regex)`** method in the **`String`** class for simple tokenizing of a **`String`**
  - It is easier to use than the tokenizers, but wont return the delimiting character (which is something to remember)
  - We also don't know what regex is yet, but for the purpose of this unit, we can use basic regex to split on

Comma Separated Values

- We'll take a little detour and introduce comma separated values
  - We'll use this as an example for **`split`**`()` later on
- We often need to store data to a file
  - The question is, in what form should we store it?
- If the data is in table or matrix form, one can write it out as a set of rows and columns in a certain format:
  - One row is written per line
  - Each row contains multiple fields, one per column
  - Each field's value is separated by a comma `','`

CSV Example

- Contents of the file:



- The CSV file:

```
Sales per region,Africa,Asia,Europe
Jan,34,67,56
Feb,36,87,78
Mar,31,56,88
Apr,29,67,92
Mar,43,56,78
May,54,71,68
Jun,42,65,82
```

CSV Notes

- The delimiting commas have no trailing space
  - It may look better to have space after the comma, but it just makes it harder to parse when reading the file later!
- Column sizes don't have to be consistent across rows
  - e.g., the first row (headings) has much longer fields than the same columns in subsequent rows
    - It would be a waste of space to pad out the fields
    - Parsing/tokenizing must handle these variable length fields
- Numeric data is converted into its textual equivalent
  - If we saved integers directly, we might get things that look like text '\n' (ASCII 13) or ',' (ASCII 44) but are merely part of the data

Text vs. Binary

- You don't have to save table data in CSV format
  - In fact, dumping raw binary data is often more efficient, and you know how large each field is (e.g., ints = 4 bytes)
- However, CSV text data has some advantages:
  - Easy for humans to read and edit
  - Highly portable to different platforms
    - Big endian, vs. little endian
  - Fields are explicitly separated with commas
  - CSV is a widely-known format
    - XML has surpassed CSV as the standard format for data interchange between companies, but CSV is still used.

String.split() and CSV Data

- So lets parse a CSV file with String.split()
- The idea is you call a String variable's split method with what you want to "split" it on.
- It will return a **String**[] containing each part of the split in each element of the array
- Then we can iterate over the array to get the "parts" that we need out.

Parsing a Single CSV Row

```
private void processLine(String csvRow)
{
    String[] splitLine;
    splitLine = csvRow.split(",");

    for(int ii = 0; ii < splitLine.length; ii++)
    {
        System.out.print(splitLine[ii] + " ");
    }
    System.out.println("");
}
```

Output

- Given the following CSV data file:

```
97452,James,88,96,82,86
99576,Alan,6,46,34,38
9888,Geoff,100,68,72,75
```

- The following would be the output of having **readFileExample()** call the method **processLine()**:

```
97452 James 88 96 82 86
99576 Alan 6 46 34 38
9888 Geoff 100 68 72 75
```

Writing Text Files

- Writing files is actually simpler than reading them
  - Since you don't have to worry about parsing
- The overall approach is the same:
  - Open a **`FileOutputStream`**
  - Create a **`Writer`**
  - Output data to the file using the **`Writer`**
- One thing to be careful with is that you must ensure the newlines and commas are put in the right place
  - Assuming you are outputting CSV format of course

PrintWriter

- You can use **`OutputStreamWriter`**, but it is a but clunky
  - The **`write()`** method requires an array of bytes so you have to copy your data into an array before writing
- It would be easier if you could write to a file in the same way you can print messages to the terminal
  - Fortunately you can use a **`PrintWriter`**
  - In fact, **`System.out`** is an instance of a **`PrintWriter`**
  - Thus writing to files can be identical to printing to the screen
    - Just don't forget to **`close()`** the file after you are done

PrintWriter Example

- The next slide shows an example of writing to a file
- It assumes you have passed in the student marks info that was read in earlier from the earlier CSV file
  - e.g., we are saving one line of the data to a new CSV file
  - Note that the commas are inserted between the fields, and there are no spaces added.

Writing a CSV Row

```
private void writeOneRow(String filename, int ID, String name, double assign,
                         double test, double exam, double overall)
{
    FileOutputStream fileStrm = null;
    PrintWriter pw;
    try
    {
        fileStrm = new FileOutputStream(filename);
        pw = new PrintWriter(fileStrm);
        pw.println(id + "," + name + "," + assign + "," + test + "," + exam + "," + overall);
        pw.close();
    }
    catch(IOException e)
    {
        if (fileStrm != null)
        {
            try
            {
                fileStrm.close();
            }
            catch(IOException ex2)
            { }
        }
        System.out.println("Error in writing to file: " + e.getMessage());
    }
}
```

Things Can be Simpler

- We have explored the 'long' way to open files for I/O
- Java provides other classes and constructors that do some of the steps for you
  - e.g., **`FileReader`**, **`FileWriter`**
  - e.g., the **`FileOutputStream`** creation can be done by an alternate constructor in **`PrintWriter`** that accepts the fileName
- I didn't show you the convenience classes / constructors so as to explicitly show you how Java is put together
  - But you must use the way we have shown you in your programs

Next Week

- The next Lecture will address the following:
  - Basic Sorting Algorithms