



**Concordia University**  
**Comp 249 – Fall 2020**  
**Object-Oriented Programming II**  
**Assignment 3**

**Due 11:59 PM - Friday Nov 13, 2020**

---

**Purpose:** The purpose of this assignment is to allow you practice Exception Handling, and File I/O, as well as other previous object-oriented concepts.

---

JavaScript Object Notation or **JSON** is a well-known, lightweight data-interchange format, which is easily readable/writable by humans [1][2]. It is also easy for machines to parse and generate. JSON is based on data objects consisting of attribute–value pairs and array data types. It is widely used for asynchronous browser–server communication, including as a replacement for XML in some AJAX-style systems. As such, many journals and scientific servers are supporting this format. This format is easily extendable. The following is a typical sample (modified from an existing IEEE paper).

@ARTICLE{

```
8247289,  
author={J. Park and J. N. James and Q. Li and Y. Xu and W. Huang},  
journal={IEEE Transactions on Vehicular Technology},  
title={Optimal DASH-Multicasting over LTE},  
year={2018},  
volume={PP},  
number={99},  
pages={15-27},  
keywords={Forward error correction;Long Term Evolution;Maintenance engineering;Multicast  
communication;Resource management;Static VAR compensators;Streaming media;DASH;LTE;convex  
optimization;eMBMS;multicasting},  
doi={10.1109/TVT.2018.2789899},  
ISSN={0018-9545},  
month={January},  
}
```

In JSON, the fields however do not need to be placed in specific order. For example, in the above example, you can have the “number” field for instance, be written above the “year” field, and so on.

There are many parsers available for JSON developed in many programming languages as library, API, etc. Mendeley [3] is one example that uses this format. With Mendeley you can create your own library from all articles that you have read so far and you can use them when you want to write an article. Consequently, this is very good for inserting the needed references on a published paper and managing them afterwards. In particular, such tools can import the article(s) information from a bibliography file (.bib) and generate the reference(s) in particular format according to the conference or journal publisher standard.

For example, the representation of this file in IEEE format would be (this is a slightly simplified version than the actual one; however you need to follow this format for the scope of this assignment):

J. Park, J. N. James, Q. Li, Y. Xu, W. Huang. "Optimal DASH-Multicasting over LTE", IEEE Transactions on Vehicular Technology, vol. PP, no. 99, p. 15-27, January 2018.

Or in **ACM** format would be:

- [1] J. Park et al. 2018. Optimal DASH-Multicasting over LTE. IEEE Transactions on Vehicular Technology. PP, 99 (2018), 15-27. DOI:<https://doi.org/10.1109/TVT.2018.2789899>.

And finally, in **Nature Journal (NJ)**, which is one of the most famous journals in natural science, would be

J. Park & J. N. James & Q. Li & Y. Xu & W. Huang. Optimal DASH-Multicasting over LTE. IEEE Transactions on Vehicular Technology. PP, 15-27(2018).

You should have a “very” detailed look at these formats to see how they are mapped from the original record of the article.

In this assignment, you will be designing and implementing an alternative tool (to the existing ones), called **BibliographyFactory**. The main task of this tool is read and process a given .bib file (which has one or more articles) and create **3** different files with the correct reference formats for IEEE, ACM and NJ.

In short, you are given 10 files, called *Latex1.bib* to *Latex10.bib*. Your BibliographyFactory application will need to read all 10 input files (in one execution), determine whether each of these files is valid or not (details given below). If a file is valid, then BibliographyFactory will create 3 different files based on the articles in this file, one for IEEE format, one for ACM format and the last for NJ format. To distinguish our application from other existing similar software, we will call these files *IEEE*i*.json*, *ACM*i*.json*, and *NJ*i*.json* (although in fact, the created files contain the references to the articles, and not json records!), where *i* is the Latex file #. For instance, *Latex3.bib* will result in the creation of 3 files called: *IEEE3.json*, *ACM3.json* and *NJ3.json*. If a file is invalid, then none of the 3 output reference files (for this invalid file) is created. So, in best case, BibliographyFactory execution will result in the creation of 30 output files if all given Latex files are valid, and in worst case it will create 0 files (when all 10 input files are invalid).

The fine details of what you need to do and how your BibliographyFactory should work are given below.

1. For the purpose of this assignment, and to provide little simplifications, the following should be assumed in relation to the input files and the articles (records) in these input file:
  - a. Each file may have one or more articles; the number is unknown before processing, and your code must assume that;
  - b. An article starts with `@ARTICLE` followed by the body of the article (between “{” and “}”). It is assumed that all articles have enclosing bodies;
  - c. Inside each of these articles, there exists few fields; i.e. author, volume, year, etc. These fields are assumed to always start with the field name, followed by an “=” sign and a “{” character. For instance: `pages={ , month= , etc.`
  - d. It also assumed that each of the bodies of these fields have a closing character for its body. In specific, it is assumed that each of these fields end with “},”;
  - e. For simplicity, it is also assumed that there is a doi field, which maps to <https://doi.org/>
  - f. The order of the article is NOT important. i.e., it is okay to have any of these fields above or below other fields;
  - g. It is also assumed that empty lines can be there within the body of the articles, as well as between different articles;
  - h. HOWEVER (read carefully), any of the input files may have some of these fields as empty; i.e. `number={}`, `title={}`, etc. This is what we classify as an Invalid File. In other words, for a file to be valid, the file cannot have an empty field at any of its articles. Below is an image of an invalid file.

```

@ARTICLE{
8247289,
author={J. N. James and Q. Li and Y. Xu and W. Huang},
journal={IEEE Transactions on Vehicular Technology},
title={},
year={2018},
volume={PP},
number={},
pages={15-27},
keywords={Forward error correction;Long Term Evolution;Maintenance engineering;Multicast commun
doi={10.1109/TVT.2018.2789899},
ISSN={0018-9545},
month={January},
}

@ARTICLE{
2380090,
author={},
journal={IEEE Transactions on Computer Science},
title={Detecting Security Vulnerabilities in Binary Code},
year={2017},
volume={QQ},
number={85},
pages={},
keywords={Security attacks;Binary code processing;Security error detection;Deep machine learnin
doi={14.2408/TCS.2017.4746889}.

```

This is also empty, but processing will not reach here as the file has already been determined to be invalid!

Figure 1. Example of an Invalid Input File

2. Write an exception class called **FileInvalidException** exception. The class should have sufficient constructors allow:
  - a. A default error message “*Error: Input file cannot be parsed due to missing information (i.e. month={}, title={}, etc.)*” to be stored in the thrown object; and
  - b. The passing of any different error message if desired. This is actually the constructor that you will be using throughout the assignment (see Figure 3 below).
3. In the main() method of the BibilographyFactory class, attempt to open all 10 input files (Latex1.bib to Latex10.bib) for reading. You need to use the **Scanner** class for the reading these files. If “any” of these files does not exist, the program must display an error message indicating “*Could not open input file xxxxx for reading. Please check if file exists! Program will terminate after closing any opened files.*”, and then exits. You MUST however, close all opened files before exiting the program. For example, if *Latex3.bib* does not exist, then the following image shows the behavior of the program.

```

eclipse-workspace - BibCreator/src/BibilographyFactory/BibilographyFactory.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

<terminated> BibilographyFactory [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (Feb 27, 2020, 8:06:31 PM)
Welcome to BibilographyFactory!

Could not open input file Latex3.bib for reading.

Please check if file exists! Program will terminate after closing any opened files.

```

Figure 2. Example of Program Termination – One of the Input Files does not Exist

4. If all 10 input files can successfully be opened, the program will attempt to open/create all 30 output files (IEEE1.json to IEEE10.json, ACM1.json to ACM10.json, and NJ1.json to

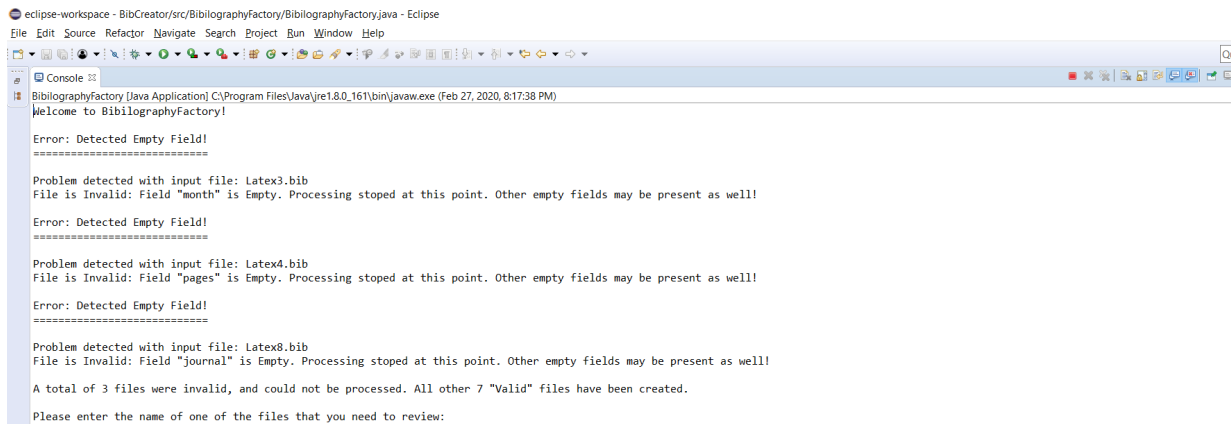
NJ10.json). You need to use PrintWriter to open these output files. If “any” of these output files cannot be created, then you must:

- Display a message to the user indicating which file could not be opened/created;
- Delete all other created output files (if any). That is, if you cannot create all of these output files, then you must clean the directory by deleting all other created files;
- Close all opened input files; then exit the program.

If you reach this step, then all 10 input files have been opened and all 30 output files have also been created (however, they are surely empty).

- Write a method (you should take advantage of static throughout the entire assignment!) called **processFilesForValidation**. This method will represent the core engine for processing the input files and creating the output ones. You can pass any needed parameters to this method, and the method may return any needed information. This method however must NOT declare any exceptions. In other words, all needed handling of any exceptions that may occur within this method, must be handled by the method. In specific:
  - The method should work on the already opened files;
  - A method must process each of these files to find out whether it is valid or not;
  - If a file is valid, then the method must create the proper records for each of the 3 formats (IEEE, ACM and NJ) and store them in these files;
  - If a file is invalid, then the method must stop the processing of this file only, throws **FileInvalidException** to display the exception error message, then display a message indicating which file was detected as invalid, and where the “first” problem in that file was detected (See Figure 3). The corresponding output file MUST then be deleted;
  - The method will then continue with the processing of the following file.

For instance, let us assume that the given input files (these files can be any files and they are not restricted to the ones provided with the assignment; in fact, the marker will execute your assignment with different files; so your code must work correctly for any given files) have 3 invalid files, Latex3.bib, Latex4.bib and Latex8.bib. Your program must detect these invalid files and show the following:



```
eclipse-workspace - BibCreator/src/BibliographyFactory/BibliographyFactory.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Console
BibliographyFactory [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (Feb 27, 2020, 8:17:38 PM)
Welcome to BibliographyFactory!

Error: Detected Empty Field!
=====
Problem detected with input file: Latex3.bib
File is Invalid: Field "month" is Empty. Processing stopped at this point. Other empty fields may be present as well!

Error: Detected Empty Field!
=====
Problem detected with input file: Latex4.bib
File is Invalid: Field "pages" is Empty. Processing stopped at this point. Other empty fields may be present as well!

Error: Detected Empty Field!
=====
Problem detected with input file: Latex8.bib
File is Invalid: Field "journal" is Empty. Processing stopped at this point. Other empty fields may be present as well!

A total of 3 files were invalid, and could not be processed. All other 7 "Valid" files have been created.

Please enter the name of one of the files that you need to review:
```

Figure 3. Example of Processing – Number of Invalid Files and Particular Fields are Indicated

- Again, once the processing is done, all unsuccessfully created files MUST be deleted. Here is how the directory would look like at this point based on the above scenario:

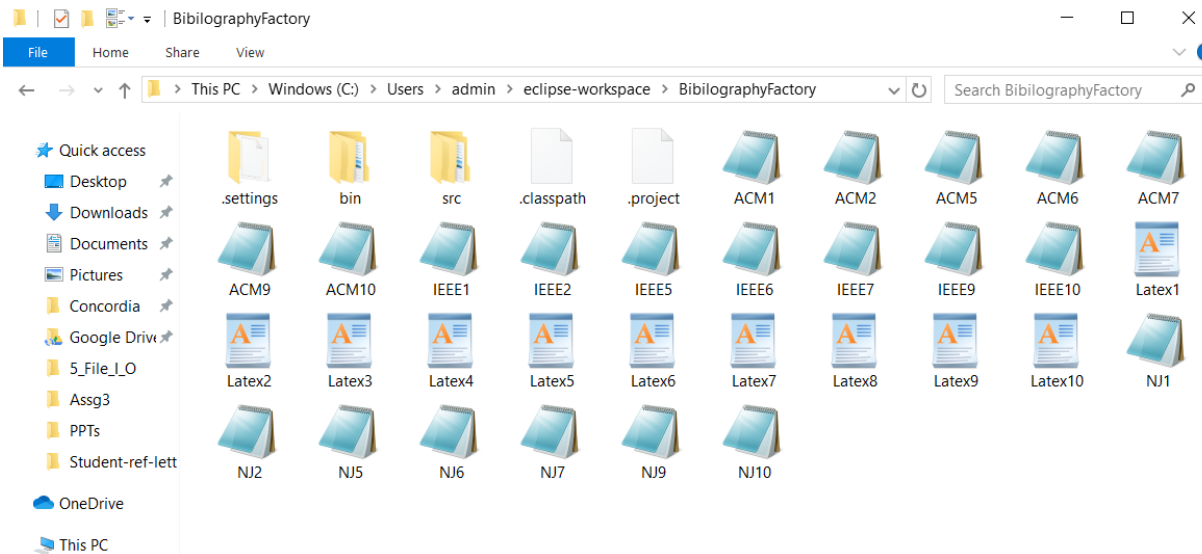


Figure 4. Contents of Current Directory after above processing Example

7. Finally, at this point, the program needs to ask the user to enter the name of one of the created output files to display. If the user enters an invalid name, a `FileNotFoundException` should be thrown; however, the user is allowed a second and final chance to enter another name. If this second attempt also fails, then the program exits. Figure 5 and Figure 6 below show the behavior of this program. You must however apply the following:
  - If the entered file is valid, then your program must open this file for reading using the **BufferedReader** class. Do not use the `Scanner` class to read the file for that task.
8. Finally, here are some general information:
  - a. It may assist you greatly if you take advantage of static variables/attributes and static attributes throughout the assignment; in fact, it is not necessary to utilize other aspects such as Inheritance, Polymorphism, etc.
  - b. You must exactly match the format and look of the expected output files. For instance, use & or et al. for the authors as expected, follow the exact order/format of the contents, use vol. instead of volume when as expected. In other words, a small difference in the expected output will surely result in mark deduction;
  - c. For the processing of the authors, you may want to use the **StringTokenizer** class;
  - d. You should minimize opening and closing the files as much as possible; a better mark will be given for that;
  - e. Do not use any external libraries or existing software to produce what is needed; that will directly result in a 0 mark!
  - f. Again, your program must work for any input files. The files provided with this assignment are only one possible version, and must not be considered as the general case when writing your code.
  - g. To make sure that the requirements are very clear to you, Figure 7 given an image (partially) of a sample input file, and Figures 8, 9 & 10 show the output of this sample file in the 3 formats. These files are also provided with the assignment.

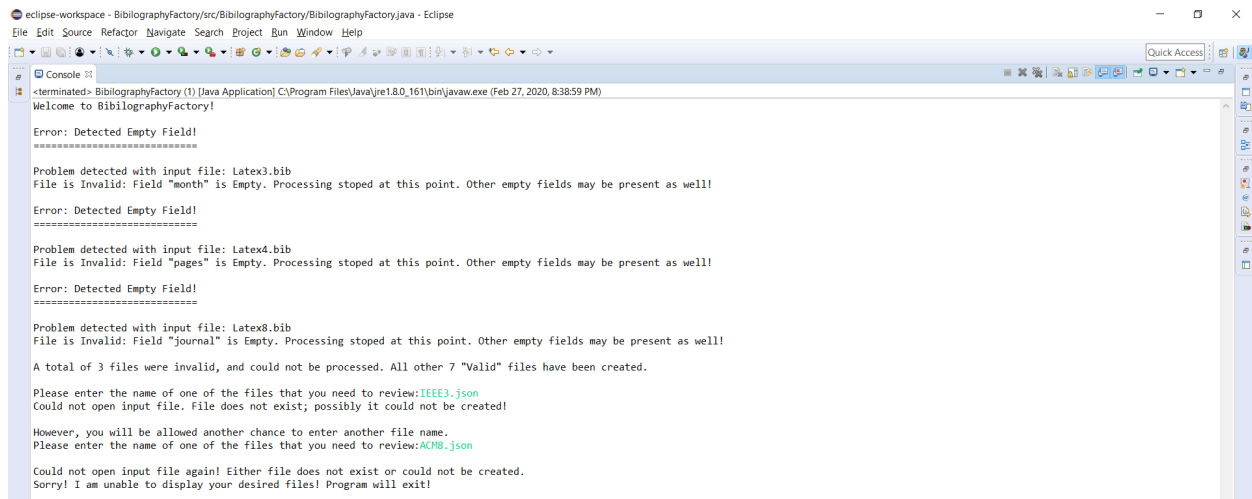


Figure 5. Example of Double-Fault for Displaying an Output File

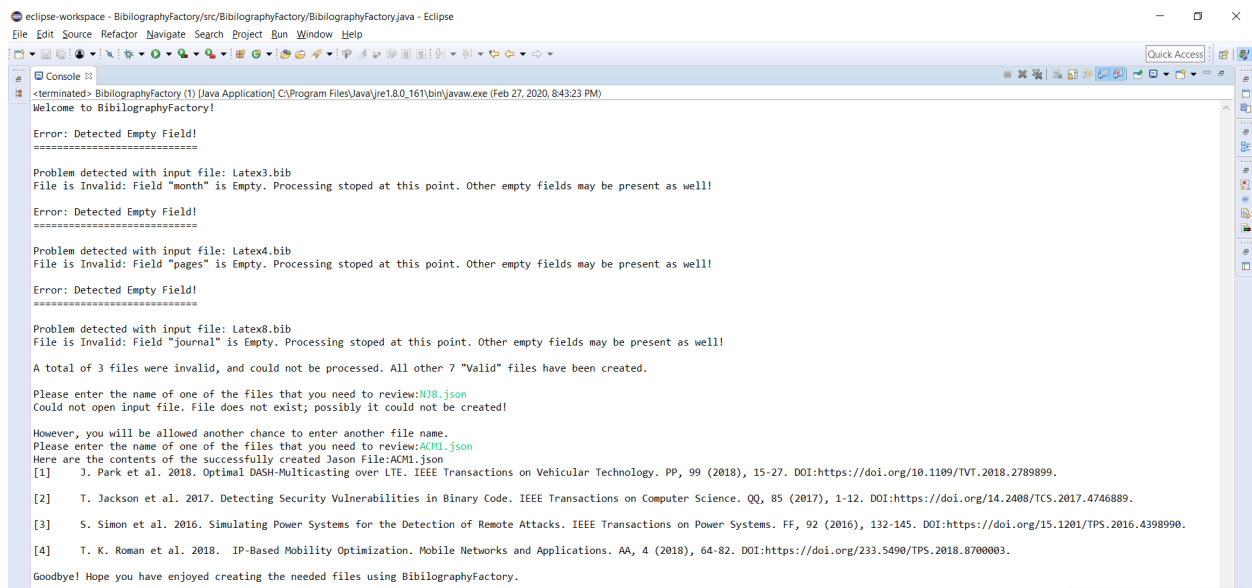


Figure 6. Example of Displaying the Contents of a Successfully Created Output File

```

LatexSample - Notepad
File Edit Format View Help
@ARTICLE{
8247289,
author={J. Park and J. N. James and Q. Li and Y. Xu and W. Huang},
journal={IEEE Transactions on Vehicular Technology},
title={Optimal DASH-Multicasting over LTE},
year={2018},
volume={PP},
number={99},
pages={15-27},
keywords={Forward error correction;Long Term Evolution;Maintenance engineering;Multicast communication;Resource management;Static VAR compensators;Streaming media;DASH;LTE;co
doi={10.1109/TVT.2018.2789899},
ISSN={0018-9545},
month={January},
}

@ARTICLE{ |
2380090,
author={T. Jackson and A. H. Peterson and N. Wang},
journal={IEEE Transactions on Computer Science},
year={2017},
volume={QQ},
number={85},
pages={1-12},

title={Detecting Security Vulnerabilities in Binary Code},
keywords={Security attacks;Binary code processing;Security error detection;Deep machine learning;Static analysis},
doi={14.2408/TCS.2017.4746889},
ISSN={0018-9545},
month={May},
}

@ARTICLE{
7628761,
author={S. Simon and K. Tomson},
journal={IEEE Transactions on Power Systems},
title={Simulating Power Systems for the Detection of Remote Attacks},
year={2016},
volume={FF},
}

```

Figure 7. Example of a Sample Bib Input File – Partial Image

```

IEEE Sample - Notepad
File Edit Format View Help
J. Park, J. N. James, Q. Li, Y. Xu, W. Huang. "Optimal DASH-Multicasting over LTE", IEEE Transactions on Vehicular Technology, vol. PP, no. 99, p. 15-27, January 2018.

T. Jackson, A. H. Peterson, N. Wang. "Detecting Security Vulnerabilities in Binary Code", IEEE Transactions on Computer Science, vol. QQ, no. 85, p. 1-12, May 2017.

S. Simon, K. Tomson. "Simulating Power Systems for the Detection of Remote Attacks", IEEE Transactions on Power Systems, vol. FF, no. 92, p. 132-145, November 2016.

T. K. Roman, C. Henry Jr., L. Fevens. "IP-Based Mobility Optimization", Mobile Networks and Applications, vol. AA, no. 4, p. 64-82, February 2018.

```

Figure 8. The Created IEEE File for the above Sample Bib File

```

ACMSample - Notepad
File Edit Format View Help
[1] J. Park et al. 2018. Optimal DASH-Multicasting over LTE. IEEE Transactions on Vehicular Technology. PP, 99 (2018), 15-27. DOI:https://doi.org/10.1109/TVT.2018.2789899.

[2] T. Jackson et al. 2017. Detecting Security Vulnerabilities in Binary Code. IEEE Transactions on Computer Science. QQ, 85 (2017), 1-12. DOI:https://doi.org/14.2408/TCS.2017.4746889.

[3] S. Simon et al. 2016. Simulating Power Systems for the Detection of Remote Attacks. IEEE Transactions on Power Systems. FF, 92 (2016), 132-145. DOI:https://doi.org/15.1201/TPS.2016.

[4] T. K. Roman et al. 2018. IP-Based Mobility Optimization. Mobile Networks and Applications. AA, 4 (2018), 64-82. DOI:https://doi.org/233.5490/TPS.2018.8700003.

```

Figure 9. The Created ACM File for the above Sample Bib File

```

NISample - Notepad
File Edit Format View Help
J. Park & J. N. James & Q. Li & Y. Xu & W. Huang. Optimal DASH-Multicasting over LTE. IEEE Transactions on Vehicular Technology. PP, 15-27(2018).

T. Jackson & A. H. Peterson & N. Wang. Detecting Security Vulnerabilities in Binary Code. IEEE Transactions on Computer Science. QQ, 1-12(2017).

S. Simon & K. Tomson. Simulating Power Systems for the Detection of Remote Attacks. IEEE Transactions on Power Systems. FF, 132-145(2016).

T. K. Roman & C. Henry Jr. & L. Fevens. IP-Based Mobility Optimization. Mobile Networks and Applications. AA, 64-82(2018).

```

Figure 10. The Created Natural Journal File for the above Sample Bib File

### **General Guidelines When Writing Programs:**

- Include the following comments at the top of your source codes  

```
// -----
// Assignment (include number)
// Question: (include question/part number, if applicable)
// Written by: (include your name and student ID)
// -----
```
- In a comment, give a general explanation of what your program does. As the programming questions get more complex, the explanations will get lengthier.
- Include comments in your program describing the main steps in your program.
- Display a welcome message which includes your name(s).



- Display clear prompts for users when you are expecting the user to enter data from the keyboard.
- All output should be displayed with clear messages and in an easy to read format.
- End your program with a closing message so that the user knows that the program has terminated.

### **JavaDoc Documentation:**

Documentation for your program must be written in **javaDoc**.

In addition, the following information must appear at the top of each file:

```
Name(s) and ID(s)      (include full names and IDs)
COMP249
Assignment #           (include the assignment number)
Due Date               (include the due date for this assignment)
```

### **Submitting Assignment 3**

- For this assignment, you are allowed to work individually, or in a group of a maximum of 2 students (i.e. you and one other student). You and your teammate must however be in the same section of the course. Groups of more than 2 students = zero mark for all members!
  - Only electronic submissions will be accepted. Zip together the source codes.
  - Students will have to submit their assignments (one copy per group) using the EAS (<https://fis.encs.concordia.ca/eas/>) system. Assignments must be submitted in the right folder of the assignments. **Assignments uploaded to an incorrect folder will not be marked and result in a zero mark. No resubmissions will be allowed.**
  - Naming convention for zip file: Create one zip file, containing all source files and produced documentations for your assignment using the following naming convention:  
The zip file should be called *a#\_StudentName\_StudentID*, where # is the number of the assignment and *StudentName/StudentID* is your name and ID number respectively. Use your “official” name only - no abbreviations or nick names; capitalize the usual “last” name. Inappropriate submissions will be heavily penalized. For example, for the first assignment, student 12345678 would submit a zip file named like: *a1\_Mike-Simon\_123456.zip*. if working in a group, the name should look like: *a1\_Mike-Simon\_12345678-AND-Linda-Jackson\_98765432.zip*.
  - Submit only ONE version of an assignment. If more than one version is submitted the first one will be graded and all others will be disregarded.
  - If working in a team, only one of the members can upload the assignment. Do NOT upload the file for each of the members!
- ⇒ Important: Following your submission, a demo is required (please refer to the courser outline for full details). The marker will inform you about the demo times. Please notice that failing to demo your assignment will result in zero mark regardless of your submission.

### **Evaluation Criteria for Assignment 3 (10 points)**

| <b>Total</b>                      | <b>10 pts</b> |
|-----------------------------------|---------------|
| <b>JavaDoc</b> documentations     | 1 pt          |
| Task # 2, Task 3 & Task # 4       | 1 pt          |
| Task # 5                          | 4 pts         |
| Task # 6                          | 2 pts         |
| Task # 7                          | 1 pt          |
| General Quality of the Assignment | 1 pt          |