

EECS 1021: Lab J

Accessing and Analyzing Hydrometric Engineering Data Online *in Java*

Before the lab: have you done the H5P and VPL activities on eClass? There are a dozen or so such activities that, if you do them before the lab, will help you with executing the lab. Also check out GitHub: <https://github.com/rr-codes/LabJ>

Introduction

Climate change has increased the severity of weather and is having a noticeable impact on the bodies of water in and around cities, including Toronto, as well as European cities such as Strasbourg, on the border between France and Germany (where James did his sabbatical in 2018/19; <https://bit.ly/3ouiWRu>). To better understand the general trends, but also to make decisions on how to regulate the flow of water (in and around our cities, engineers and city officials measure the height and flow (“streamflow”) of our rivers using hydrometric sensors such as the ones shown here: <https://bit.ly/2Qjn5iY>



Figure 1 York's Professor Usman Khan showing us a weir, which can be used to examine streamflow. (c/o Prof. Khan and CIVL 3220 class notes). More on his research here: <https://bit.ly/2O7KS2v>

Engineers like the Lassonde School's Professor Usman Khan use hydrometric data to examine the state of our watersheds. The information allows informs us about how water flows in and around first nations, cities, towns and rural areas so that we can make engineering and public policy decisions, like where to build dams, as well as how we should attempt to control water flow in our lakes and rivers. (<https://bit.ly/2O7KS2v>)




		
The Rhine River on the edge of Strasbourg, France ¹	Toronto's Don River @ Todmorden (historical painting) ²	Attawapiskat First Nation ³

Figure 2 Most cities and towns are built on bodies of water. Strasbourg, Toronto and Attawapiskat First Nation are examples of these.

¹ Rhine River photo: James Andrew Smith (<http://drsmith.blog.yorku.ca/>)

² Don River @ Todmorden historical painting (<http://citiesintime.ca/toronto/story/remnants-tor/>)

³ Attawapiskat First nation photo c/o Wikimapia (http://photos.wikimapia.org/p/00/02/11/09/06_full.jpeg)

In EECS 1021 we talk about “computational thinking”, using sensors and measurement as a platform to explore the concept. In all engineering disciplines sensors applied to data logging and data analysis applications are ubiquitous. The sensors used in Hydrometric applications are typical of the kinds of sensors found in data logging applications, using SONAR, RADAR or encoders to measure depth, distance and flow. The output of these sensors needs to be processed and analyzed prior to reacting with actuators like the ones found in the dams or lock of our waterways.

Engineers need to monitor the water levels, sometimes on a daily or hourly basis. This can be done by

- Visiting the water gauge
- Phoning for water gauge information
- Using the internet to monitor live data

In this lab we will explore how to access some of these water monitoring stations remotely, by telephone and internet. This is like we did with MATLAB in EECS 1011, but this time we’re doing it with Java.

Marking guide

- | | | |
|---|-----|-----------------------|
| 1. Provide the average depth value for the water station: | 0.2 | (0.1 if unsuccessful) |
| 2. Complete graph for daily water depth value: | 0.8 | (0.4 if unsuccessful) |

WATER LEVEL INSTRUMENTS

MODELS 6541C AND 6547A

SPECIFICATIONS
ORDERING
DOCUMENTS

OVERVIEW

The 6541 precision water level instrument is a high accuracy float and pulley based shaft encoder instrument for measuring the level of water in many different applications. Float-operated instruments can be the most accurate way to monitor water levels, and they are the most common method to measure river levels. The Unidata 6541 precision water level instrument can achieve operating accuracy and resolution of 0.2mm with high stability and minimum drift.

This accuracy is maintained for the service life of the instrument without calibration or maintenance, apart from battery changes. The 6541 has the range to monitor surface and underground waters and the precision to monitor rainfall and evaporation. The water level instrument is normally connected to the surface of the water by a float system. As the water level changes, the input shaft rotates. An optical encoder is mounted on the input shaft. On installation, the instrument is set to display the water level.

The encoder is continuously monitored as the instrument tracks water level changes. These changes update the LCD display and the readings can be recorded by an associated datalogger. The very low mechanical friction and inertia of the instrument means that it can produce data with high precision and accuracy. A replaceable battery pack powers the instrument for more than twelve months. Practical design and rugged construction ensures easy operation and long service life.




Figure 3 A example of an industrial water depth sensor (c/o

<https://www.unidata.com.au/products/water-monitoring-modules/precision-water-level-instrument/>)

York University acknowledges its presence on the traditional territory of many Indigenous Nations. The area known as Tkaronto has been care taken by the Anishinabek Nation, the Haudenosaunee Confederacy, the Huron-Wendat, and the Métis. It is now home to many Indigenous Peoples. We acknowledge the current treaty holders, the Mississaugas of the Credit First Nation. This territory is subject of the Dish With One Spoon Wampum Belt Covenant, an agreement to peaceably share and care for the Great Lakes region.

This lab activity includes an examination of water resources from a number of locations around Ontario, including the Attawapiskat First Nation, which you may have heard about in the news. You are encouraged to further explore online resources and news sites to inform yourself about the issues involving the First Nations.



Figure 4 use your telephone to obtain the water level at an automated hydrometric station.⁴

Part 1: Phone the Water Level Sensor

Before the lab, use your phone to obtain water level at a variety of locations around Canada using the Canadian Hydrographic service and the [waterlevels.gc.ca](https://waterlevels.gc.ca/eng/info/bulletin) “Bulletin” site (<https://waterlevels.gc.ca/eng/info/bulletin>):

1. Section 1: Water level @ St. Lawrence River, above the lock at Iroquois:
 - a. (613) 652-4426
2. Section 2: Water level @ St. Lawrence River, below the lock at Iroquois:
 - a. (613) 652-4839
3. Section 3: Water level @ Lake Huron at Tobermory
 - a. (519) 596-2085
4. Section 4: Water level @ Lake Ontario at Port Weller
 - a. (905) 646-9568
5. Section 5: Water level @ Lake Ontario at Burlington
 - a. (905) 544-5610
6. Section 6 Water level @ Sault Ste. Marie, above the lock
 - a. (705) 949-2066
7. Section 7 Water level @ Sault Ste. Marie, below the lock
 - a. (705) 254-7989
8. Section 8: Water level @ St. Lawrence River at Kingston
 - a. (613) 544-9264
9. Section 9: Water level @ the Detroit River at Amherstburg
 - a. (519) 736-4357
10. Section 10: Water level @ St. Lawrence River at Cornwall
 - a. (613) 930-9373
11. Section 11: Water level @ Lake Superior at Thunder Bay
 - a. (807) 344-3141
12. Section 12 or 13: Water level @ Lake Superior at Gros Cap
 - a. (705) 779-2052

Phone the particular location and [for the lab report] **write down (a) the current depth of the water, (b) the date and (c) the time of day.** No need to report on this, just try it out. It will only take a minute or two.

Note that your classmates *may* be phoning *at the same time*. **If it's busy**, wait and try again.

⁴ Icons courtesy of the Noun Project.

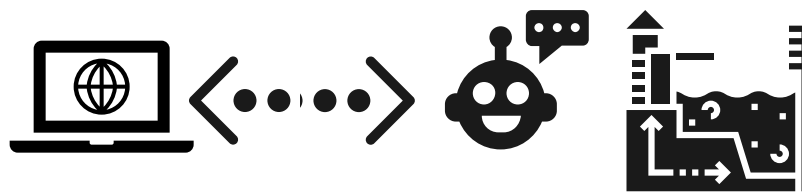


Figure 5 Use your computer and Java to obtain the water datasets at an automated hydrometric station.

Part 2: Getting Water Depth Information via Online Datasets.

Obtain the **daily water depth** for four different gauges, as listed below. Use the **Java code from GitHub** to get the data, stored on a Canadian government server as a “CSV” file. Assume that each new data point represents a new daily value.

Section	Station 1	Station 2	Station 3	Station 4
1 & 2	"DON RIVER AT TODMORDEN" (02HC024)	"BLACK CREEK NEAR WESTON" (02HC027)	"ATTAWAPISKAT RIVER BELOW ATTAWAPISKAT LAKE" (04FB001)	"ATTAWAPISKAT RIVER BELOW MUKETEI RIVER" (04FC001)
3 & 4	"HUMBER RIVER AT ELDER MILLS" (02HC025)	"DON RIVER AT TODMORDEN" (02HC024)	"ATTAWAPISKAT RIVER BELOW ATTAWAPISKAT LAKE" (04FB001)	"ATTAWAPISKAT RIVER ABOVE LAWASHI CHANNEL" (04FC002)
5 & 6	"LITTLE ROUGE CREEK NEAR LOCUST HILL" (02HC028)	"WINDIGO RIVER ABOVE MUSKRAT DAM LAKE" (04CB001)	"DON RIVER AT TODMORDEN" (02HC024)	"ATTAWAPISKAT RIVER BELOW MUKETEI RIVER" (04FC001)
7 & 8	"ETOBICOKE CREEK BELOW QUEEN ELIZABETH HIGHWAY" (02HC030)	"LITTLE ROUGE CREEK NEAR LOCUST HILL" (02HC028)	"HUMBER RIVER AT ELDER MILLS" (02HC025)	"DON RIVER AT TODMORDEN" (02HC024)
9 & 10	"REDHILL CREEK AT HAMILTON" (02HA014)	"ETOBICOKE CREEK BELOW QUEEN ELIZABETH HIGHWAY" (02HC030)	"LITTLE ROUGE CREEK NEAR LOCUST HILL" (02HC028)	"BLACK CREEK NEAR WESTON" (02HC027)
11	"FRENCH RIVER AT PORTAGE DAM", (02DD016)	"REDHILL CREEK AT HAMILTON" (02HA014)	"ETOBICOKE CREEK BELOW QUEEN ELIZABETH HIGHWAY" (02HC030)	"DON RIVER AT TODMORDEN" (02HC024)
12 (or 13)	"WINDIGO RIVER ABOVE MUSKRAT DAM LAKE" (04CB001)	"FRENCH RIVER AT PORTAGE DAM", (02DD016)	"REDHILL CREEK AT HAMILTON" (02HA014)	"HUMBER RIVER AT ELDER MILLS" (02HC025)

The **names** of stations and their **IDs** can be found here:

https://dd.weather.gc.ca/hydrometric/doc/hydrometric_StationList.csv

The **datasets** are found here, identified by province, daily or hourly data frequency and their station ID:

<https://dd.weather.gc.ca/hydrometric/csv/>

When you look at individual CSV files for daily depth, they look like this:

ID	Date	Water Level / Niveau d'eau (m)	Grade	Symbol	Discharge / Débit (cms)	Grade	Symbol
02ED003	2021-02-17T00:05:00-05:00	5.693	7.67	1			
02ED003	2021-02-17T01:05:00-05:00	5.692	7.67	1			
02ED003	2021-02-17T01:15:00-05:00	5.691	7.65	1			
02ED003	2021-02-17T01:25:00-05:00	5.691	7.65	1			
02ED003	2021-02-17T01:30:00-05:00	5.690	7.64	1			
02ED003	2021-02-17T01:35:00-05:00	5.689	7.62	1			
02ED003	2021-02-17T01:40:00-05:00	5.688	7.61	1			
02ED003	2021-02-17T01:45:00-05:00	5.688	7.61	1			
02ED003	2021-02-17T01:50:00-05:00	5.688	7.61	1			
02ED003	2021-02-17T01:55:00-05:00	5.688	7.61	1			
02ED003	2021-02-17T01:00:00-05:00	5.687	7.59	1			
02ED003	2021-02-17T01:05:00-05:00	5.686	7.58	1			
02ED003	2021-02-17T01:10:00-05:00	5.686	7.58	1			
02ED003	2021-02-17T01:15:00-05:00	5.686	7.58	1			
02ED003	2021-02-17T01:20:00-05:00	5.685	7.56	1			
02ED003	2021-02-17T01:25:00-05:00	5.685	7.56	1			
02ED003	2021-02-17T01:30:00-05:00	5.685	7.56	1			
02ED003	2021-02-17T01:35:00-05:00	5.685	7.56	1			
02ED003	2021-02-17T01:40:00-05:00	5.684	7.54	1			
02ED003	2021-02-17T01:45:00-05:00	5.684	7.54	1			
02ED003	2021-02-17T01:50:00-05:00	5.684	7.54	1			
02ED003	2021-02-17T01:55:00-05:00	5.683	7.53	1			
02ED003	2021-02-17T02:00:00-05:00	5.683	7.53	1			
02ED003	2021-02-17T02:05:00-05:00	5.683	7.53	1			

Figure 6 Water station data at station "Ontario 02ED003" (ON_02ED003). These are the "daily" readings, as opposed to the "hourly" readings. Station 02EC002 is located at Nottawasag River near Baxter, Ontario.

https://en.wikipedia.org/wiki/Nottawasaga_River

You'll need to **find the average value** of the daily depth value (use the **daily** sets, *not* the hourly sets) in the Ontario folder. However, many sensors have *bad data* in them, represented by a **NaN** (not a number or "null") value in the dataset. "Bad data" is a reality of real-world sensing and engineers need to be able to handle that possibility.

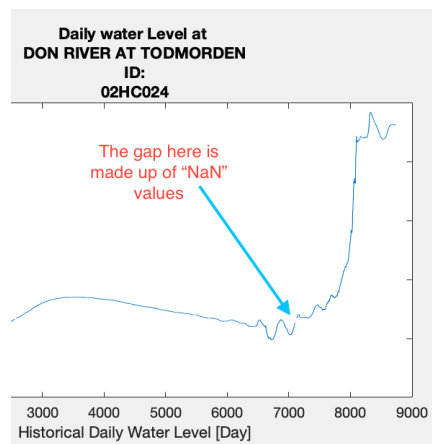


Figure 7 Some of the depth data is "bad" and is represented by NaN (not a number). This could be due to the sensor being faulty or turned off for a period of time.

You'll need to remove the "NaN" instances in the dataset before performing other operations. In your Java code consider using `orElse(0)` to replace those with numeric zero values.

Next, process **depth_data** to remove any NaN values before running the “mean” or any other function on it.

During the lab, **demonstrate to your TA** that you can graph the “Daily water level” at one of the four stations for your lab section in Table 1. The TA can choose any of the four stations. The TA will ask you for both the average depth and for you to show the graph that resulted from your Java program.

The average depth value is to be reported to the nearest centimeter (i.e. to two decimal points as depth is record in meters)

Java on GitHub

Go to GitHub: <https://github.com/rr-codes/LabJ>

In this lab you are to develop a “command line” Java program. Effectively, this means that you can either

1. Run the program from a command line or terminal program (like PowerShell in Windows or the Terminal application in macOS).
2. Run the program through IntelliJ as normal if you modify the “Program Arguments” in the same “run/debug configurations” menu that you accessed for JavaFX earlier. See the figure below.

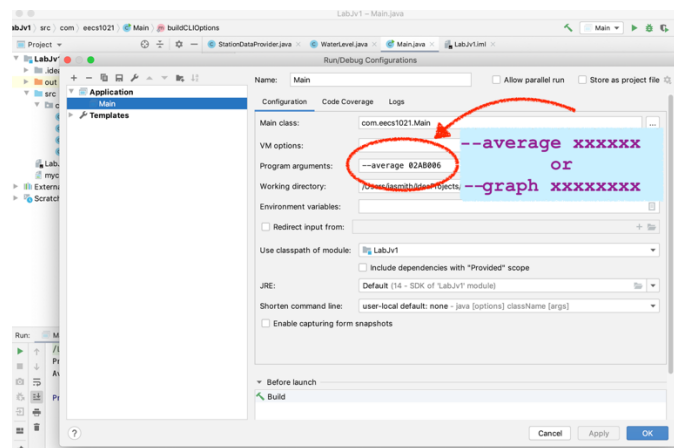


Figure 8 You can add “command line” arguments within IntelliJ if you don’t want to run the Java program directly from the Terminal.

When you modify the command line arguments you need to state whether you want a numeric value output (the “average” water depth) as an output or an image of a graph that shows the daily water levels for the station. The station IDs look like O3JK028 or 04FG002. The image will be outputted to a folder on your computer that also contains your source code, as shown in this figure:

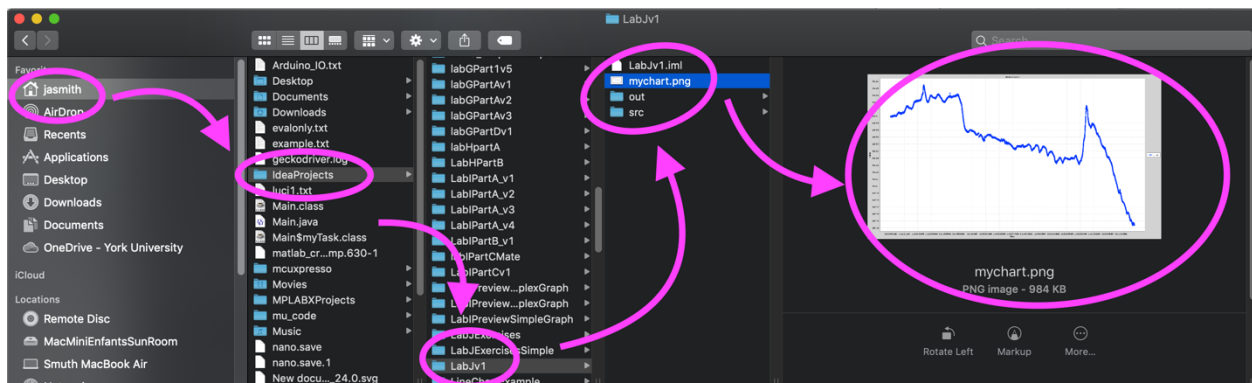


Figure 9 The graph of water depth will be stored in your project folder. The specific location will vary based on your machine and on the way that you set up IntelliJ.

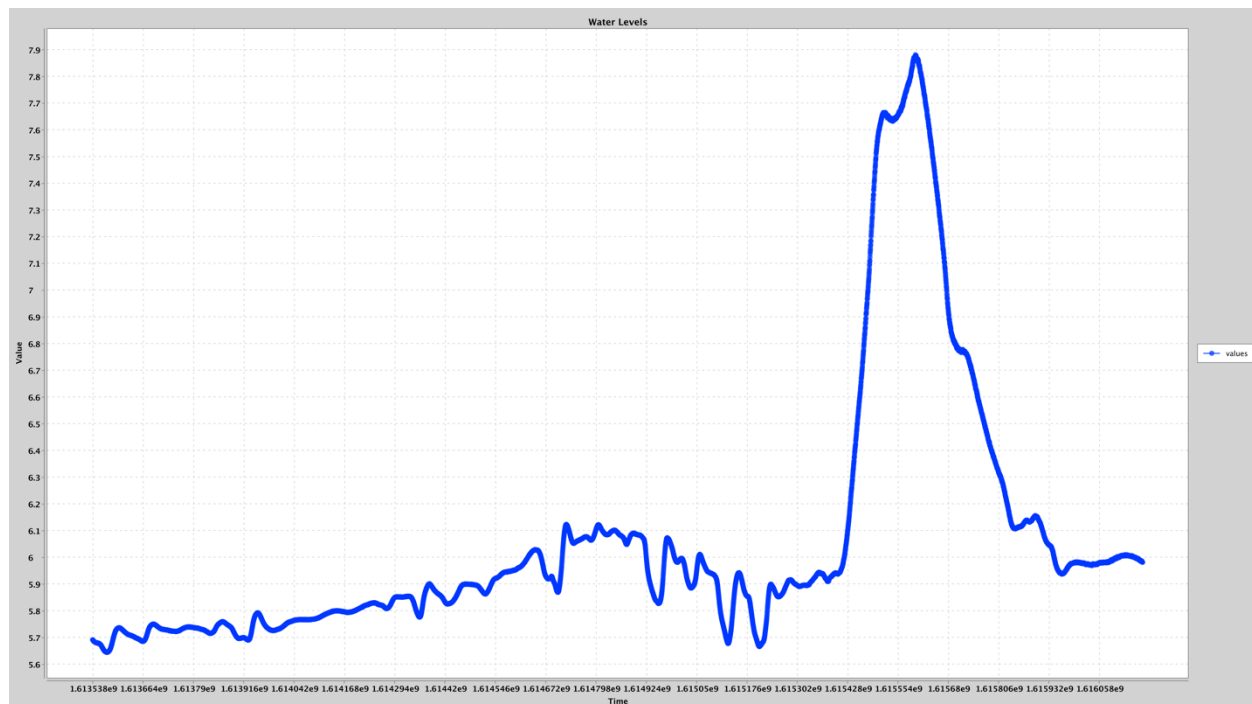


Figure 10 Water level (depth) information (daily) from station ON 02ED003. This is the graph obtained using the `--graph 02ED003` command line argument in IntelliJ. If we use `--average` we get a numeric value of 6.065358626198083.

The figure above shows one of the water depth graphs from the government website, as processed by the Java program that you are to build. Here is the original data:
https://dd.weather.gc.ca/hydrometric/csv/ON/daily/ON_02ED003_daily_hydrometric.csv

Importing Libraries using Maven

As we have seen before for `jSerialComm`, sometimes you need to add libraries to your project. They can be imported using Maven. In this project you'll need to import the following:

1. `org.apache.commons:commons-csv` (version 1.8)
2. `commons-cli:commons-cli` (version 1.4)
3. `org.jetbrains:annotations` (version 20.1.0)
4. `org.knowm.xchart:xchart` (version 3.8.0)

Modifying the source code on GitHub

In this folder you'll find the source code needed to run this lab:

- <https://github.com/rr-codes/LabJ/tree/master/src/com/eeecs1021>

There are **seven** places in the source code that you need to **focus on** for this lab:

1. "Compose a URL object" ...in Main.java
2. "If the user uses the 'a' flag" (i.e. `--average`) ...in Main.java

- | | |
|---|---------------------------------|
| 3. "If the user uses the 'g' flag" (i.e. --graph) | ...in Main.java |
| 4. Implement the averageWaterLevel() method | ... in StationAnalyzer.java |
| 5. Iterate waterLevelStream in createChart() method | ... In StationAnalyzer.java |
| 6. Implement the parseRecord() method | ... in StationDataProvider.java |
| 7. Implement getData() | ... in StationDataProvider.java |

Task 1: Compose a URL object in getUrl()

In the Main.java file you need to update the getUrl() method.
Finish it by

- Looking at the folder online where the CSV files reside. What are the addresses to these CSV files? (e.g. <https://bit.ly/38X6hHd>)
- Examining the image below. How could you reconstruct this getUrl() method?

Name	Last modified	Size	Description
Parent Directory	-	-	-
ON_02AB006_daily_hydrometric.csv	2021-03-19 15:43	478K	
ON_02AB008_daily_hydrometric.csv	2021-03-19 15:46	424K	
ON_02AB014_daily_hydrometric.csv	2021-03-19 15:43	479K	
ON_02AB017_daily_hydrometric.csv	2021-03-19 15:49	183K	
ON_02AB019_daily_hydrometric.csv	2021-03-19 15:44	465K	

Figure 11 Here are some of the water data files:
<https://dd.weather.gc.ca/hydrometric/csv/ON/daily/>

```

24 private static URL getUrl(String stationCode) throws MalformedURLException {
25     /*
26     TODO: Compose a URL object for the data URL of the specified station code.
27     If the code is 'ABC', then the URL string would be 'https://dd.weather.gc.ca/hydrometric/csv/ON/daily/ON_ABC_daily_hydrometric.csv'
28     */
29     var scheme = "https";
30     var domain = "dd.weather.gc.ca";
31     var path = "/hydrometric/csv/ON/" + stationCode + "_daily_hydrometric.csv";
32
33     return new URL(scheme + "://" + domain + path);
34 }
  
```

Figure 12 Edit the getUrl() method to reflect the location of the water resource files. Note that most of the final location name is common and only the station ID changes.

Task 2: Respond to the "average flag" (-a or --average)

Whether you run this program from the command line or from within IntelliJ you need to distinguish between providing a numeric or a graphical output. If the user chooses a **numeric** output then the average water depth will be provided to the user via the command window or terminal window.

```

36 public static void main(String[] args) throws ParseException {
37     var cliParser = new DefaultParser();
38     var options = buildCLIOptions();
39     var cmd = cliParser.parse(options, args);
40
41     if (cmd.hasOption('h')) {
42         var helpFormatter = new HelpFormatter();
43         helpFormatter.printHelp( cmdLineSyntax: "./labj [-ag] [station]", options);
44         System.exit( status: 0);
45     }
46
47     var stationCode = cmd.getArgList().get(cmd.getArgList().size() - 1);
48
63
64     /*
65     TODO: If the user uses the '-a' flag (ie, 'cmd.hasOption('a')'), then print the average water level
66     to the standard output, and 'return'.
67     */
68     if (cmd.hasOption('a')) {
69         var avg = analyzer.averageWaterLevel();
70         System.out.println(avg);
71         return;
72     }

```

print to screen with avg variable

Figure 13 Modify the main method so that you can capture the -a or --average input argument from the command line. This is done in the same way as the help (-h) argument is checked.

Task 3: Respond to the “graph flag” (-g or --graph)

Whether you run this program from the command line or from within IntelliJ you need to distinguish between providing a numeric or a graphical output. If the user chooses a **graphical** output then the average water depth will be provided to the user via the command window or terminal window.

```

74
75     /*
76     TODO: If the user uses the '-g' flag (ie, 'cmd.hasOption('g')'), then create and save the chart to a file.
77     To save the chart, use 'BitmapEncoder.saveBitmapWithDPI(chart, "mychart.png", BitmapEncoder.BitmapFormat.PNG, 300);',
78     where 'chart' is the name of the chart, 'mychart.png' is the name of the file, and '300' is the image DPI.
79     */
80     if (cmd.hasOption('g')) {
81         var chart = analyzer.createChart();
82         BitmapEncoder.saveBitmapWithDPI(chart, "mychart.png", BitmapEncoder.BitmapFormat.PNG, 300);
83     }
84

```

Figure 14 Modify the main method to be able to call the createChart() method if a graph is requested.

Task 4: Implement the averageWaterLevel() method

Go review the VPL activity, sumOfAllGPAs:

Interactive: Online Java Exercise 12b: Streams (sumOfAllGPAs)

Requested files: StreamOps.java ([Download](#))
Type of work: Individual work
Grade settings: Maximum grade: 1
Run: Yes. **Run script:** JAVA. **Evaluate:** Yes
Automatic grade: Yes.

This is a part of set of VPL exercises related to Streams. Streams are useful for data processing and data processing is important in engineering data logging applications. In this VPL exercise there is a hidden file in which a stream is created. It looks something like this:

```
var stream = Stream.of(
    new Student("Theresa", 3.0),
    new Student("George", 4.0),
    new Student("Yasmin", 5.0)
);
```

Hopefully, it's clear that there are three students, each with a unique name and an associated grade (GPA).

Complete the following method by copying and pasting the below code into the submission box and filling in the method

```
public static double sumOfAllGPAs(Stream<Student> students)
```

In this code we have an object called students formed from the Student class. This **second** method, sumOfAllGPAs() passes a Stream for students into the method and your task is to extract the grades (GPA) from all the students in the stream and to calculate the sum of those grades. To do so you create a single-line connected command like this:

```
return students.mapToDouble(s -> s.getGpa()).sum();
```

From left to right: apply the Map interface to the students stream. Extract the numeric GPA values from the students stream via the lambda expression in mapToDouble() and apply the sum() method.... That return on the left side will take that out of the sumOfAllGPAs() method.

How would we perhaps **call** this method from within a larger program, like in a lab activity? Like this:

```
var myOutput = myObject.sumOfAllGPAs(stream);
```

Figure 15 Review the sumOfAllGPAs activity on eClass. It provides the structure for what you need to do here.

Recall that in the VPL activity the solution looked like this:

```
public static double sumOfAllGPAs(Stream<Student> students) {
    return students.mapToDouble(s -> s.getGpa()).sum();
}
```

Figure 16 The solution to the VPL activity related to Task 4 in this Lab.

Note that the students stream had to be brought in as an the input parameter to this method, but in the lab file the stream is local to the class so we can use “this” and the stream name (waterLevelStream).

But, also go take a look at the first() VPL exercise and the use of the orElse(). For bad data values found in the CSV datasets change bad data to a value of zero (0) using the orElse().

Now, the actual solution looks something like this:

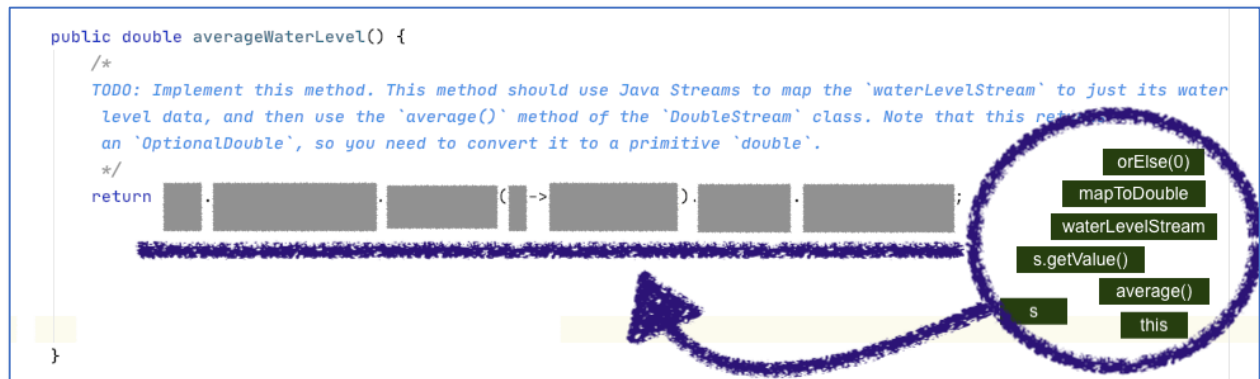


Figure 17 build up the command using the individual components on the right. This is a combination of the `sumOfAllGPAs()` and `first()` VPL assignments.

Task 5: Iterate `waterLevelStream` in `createChart()` method

Go review the VPL activity, `printStudents`. Notice how we iterate through all the stream by using the `forEach()` method.

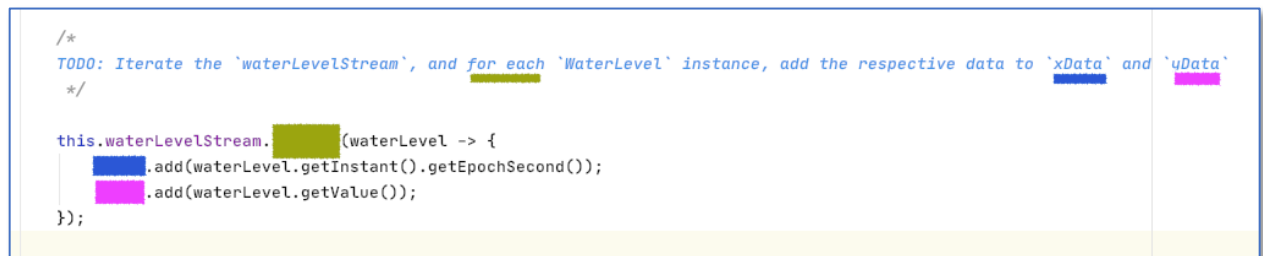


Figure 18 Modify this block of code so that we can iterate through the entire `waterLevelStream`. The "for each" component from the `printStudents` VPL activity will probably help. Also, where should the `xData` and `yData` go? (These refer to the horizontal and vertical data, respectively).

Task 6: Implement the `parseRecord()` method

Now, you have to break apart the CSV data. You need to look for date ("rawDate") and depth value ("rawValue"). Watch out for null values. Return the time (Instant) and depth (Value) values.

```

25 @ private static WaterLevel parseRecord(CSVRecord record) {
26     /*
27     TODO: Implement this method. Specifically, this method should:
28     1. Get the raw date and water level values of 'record', using 'record.get()'
29     2. If either value is 'null', return 'null'.
30     3. Parse both values. To parse an 'Instant', one can use 'Instant.parse()'. To parse a 'Double', one can use 'Double.parseDouble()'
31     4. Create and return a new 'WaterLevel' instance from these values.
32     */
33
34     var rawDate = (HEADERS[1]);
35     var rawValue = (HEADERS[2]);
36
37     if (rawDate == || rawValue == ) {
38         return ;
39     }
40
41     var parsedInstant = (rawDate);
42     var parsedValue = (rawValue);
43
44     return new WaterLevel(parsedInstant, parsedValue);
45 }

```

Task 7: Implement getData()

Convert the list of records of the parser to a stream of WaterLevel via a return from this method.

```

52 public Stream<WaterLevel> getData() throws IOException {
53     /*
54     TODO: Implement this method. This method should use Java Streams to convert the list of records of the parser
55     to a 'Stream' of 'WaterLevel's.
56     */
57
58     return ;
59 }

```

Diagram illustrating the implementation of `getData()` using Java Streams:

- `csvParser.stream()` (circled in green)
- `getRecords()` (circled in green)
- `parseRecord(s)` (circled in green)
- `s->this` (circled in green)

The implementation uses `map` to convert the stream of records into a stream of `WaterLevel` objects.

That should do it. Compile and run the program. Don't forget that you need to add command line arguments. You can do that from a terminal or you can do so from within IntelliJ. Just go into the same menu that you went into to configure JavaFX for runtime.

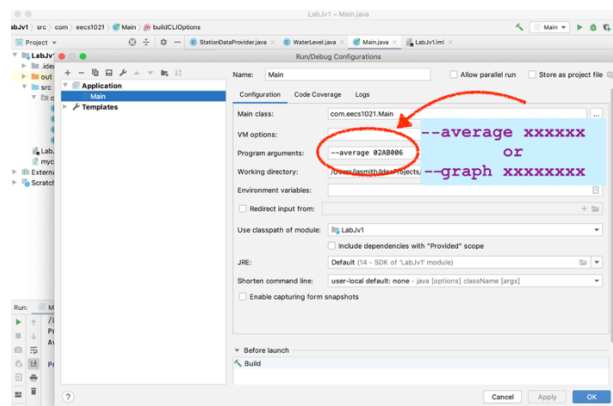


Figure 19 You can add "command line" arguments within IntelliJ if you don't want to run the Java program directly from the Terminal. Make sure to name the water station instead of just using "xxxxxx"

```
Packa
ge com.eecs1021;

import org.apache.commons.cli.*;
import org.knowm.xchart.BitmapEncoder;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

public class Main {
    private static Options buildCLIOptions() {
        var options = new Options();

        var modeGroup = new OptionGroup();
        modeGroup.addOption(new Option("g", "graph", false, "creates a
graph"));
        modeGroup.addOption(new Option("a", "average", false, "prints the
average level"));

        options.addOptionGroup(modeGroup);
        options.addOption("h", "help", false, "help");

        return options;
    }

    private static URL getUrl(String stationCode) throws
MalformedURLException {
        /*
        TODO: Compose a URL object for the data URL of the specified station
code.
        If the code is `ABC`, then the URL string would be
`https://dd.weather.gc.ca/hydrometric/csv/ON/daily/ON_ABC_daily_hydrometric.
csv`
        */
        return null;
    }

    public static void main(String[] args) throws ParseException {
        var cliParser = new DefaultParser();
        var options = buildCLIOptions();
        var cmd = cliParser.parse(options, args);

        if (cmd.hasOption('h')) {
            var helpFormatter = new HelpFormatter();
            helpFormatter.printHelp("./labj [-ag] [station]", options);
            System.exit(0);
        }

        var stationCode = cmd.getArgList().get(cmd.getArgList().size() - 1);

        URL url;

        try {
            url = getUrl(stationCode);
        }
```

```

        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }

        try (var provider = new StationDataProvider(url)) {
            System.out.println("Processing...");

            var data = provider.getData();
            var analyzer = new StationAnalyzer(data);

            /*
            TODO: If the user uses the '-a' flag (ie, `cmd.hasOption('a')`),
            then print the average water level
            to the standard output, and `return`.
            */

            /*
            TODO: If the user uses the `-g` flag (ie, `cmd.hasOption('g')`),
            then create and save the chart to a file.
            To save the chart, use `BitmapEncoder.saveBitmapWithDPI(chart,
            "mychart.png", BitmapEncoder.BitmapFormat.PNG, 300);`,
            where `chart` is the name of the chart, `mychart.png` is the
            name of the file, and `300` is the image DPI.
            */

            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

Main.java (complete the "to-do" parts)

```

package com.eecs1021;

import org.knowm.xchart.XYChart;

import java.util.ArrayList;
import java.util.stream.Stream;

public class StationAnalyzer {
    private final Stream<WaterLevel> waterLevelStream;

    public StationAnalyzer(Stream<WaterLevel> waterLevelStream) {
        this.waterLevelStream = waterLevelStream;
    }

    public double averageWaterLevel() {
        /*
        TODO: Implement this method. This method should use Java Streams to
        map the `waterLevelStream` to just its water
        level data, and then use the `average()` method of the
        `DoubleStream` class. Note that this returns
        an `OptionalDouble`, so you need to convert it to a primitive
        `double`.
        */
    }
}

```



```

        return 0.0;
    }

    public XYChart createChart() {
        var chart = new XYChart(1920, 1080);
        chart.setTitle("Water Levels");
        chart.setXAxisTitle("Time");
        chart.setYAxisTitle("Value");

        var xData = new ArrayList<Long>();
        var yData = new ArrayList<Double>();

        /*
        TODO: Iterate the `waterLevelStream`, and for each `WaterLevel`
        instance, add the respective data to `xData` and `yData`
        */

        chart.addSeries("values", xData, yData);

        return chart;
    }
}

```

StationAnalyzer.java (complete the “to-do” parts)

```

package com.eecs1021;

import org.apache.commons.csv.CSVFormat;
import org.apache.commons.csv.CSVParser;
import org.apache.commons.csv.CSVRecord;

import java.io.Closeable;
import java.io.IOException;
import java.net.URL;
import java.nio.charset.Charset;
import java.time.Instant;
import java.util.stream.Stream;

public class StationDataProvider implements Closeable {
    private static final String[] HEADERS = {
        "ID",
        "Date",
        "Water Level / Niveau d'eau (m)"
    };

    private static final CSVFormat CUSTOM_CSV_FORMAT = CSVFormat.DEFAULT
        .withHeader(HEADERS)
        .withSkipHeaderRecord();

    private static WaterLevel parseRecord(CSVRecord record) {
        /*
        TODO: Implement this method. Specifically, this method should:
        1. Get the raw date and water level values of `record`, using
        `record.get()`
        2. If either value is `null`, return `null`.
        */
    }
}

```

```
        3. Parse both values. To parse an `Instant`, one can use
`Instant.parse()`. To parse a `Double`, one can use `Double.parseDouble()`
        4. Create and return a new `WaterLevel` instance from these values.
        */
        return null;
    }

    private final CSVParser csvParser;

    public StationDataProvider(URL source) throws IOException {
        this.csvParser = CSVParser.parse(source, Charset.defaultCharset(),
CUSTOM_CSV_FORMAT);
    }

    public Stream<WaterLevel> getData() throws IOException {
        /*
        TODO: Implement this method. This method should use Java Streams to
convert the list of records of the parser
        to a `Stream` of `WaterLevel`s.
        */
        return null;
    }

    @Override
    public void close() throws IOException {
        csvParser.close();
    }
}
```

StationDataProvider.java (complete the “to-do” parts)

```
package com.eecs1021;

import java.time.Instant;

public class WaterLevel {
    private final Instant instant;
    private final double value;

    public WaterLevel(Instant instant, double value) {
        this.instant = instant;
        this.value = value;
    }

    public Instant getInstant() {
        return instant;
    }

    public double getValue() {
        return value;
    }

    @Override
    public String toString() {
        return String.format("WaterLevel{instant=%s, value=%s}", instant,
value);
    }
}
```

```
}  
}
```

WaterLevel.java