INFO3139 Lab 2

Rev 1.0

In some of the lectures, we will start off with a short discussion that may or may not be part of the work for the lab but is more a reference to a generic JavaScript topic. Its purpose is to make you aware of some of the more esoteric parts of JavaScript. Note that the topics may be represented on the midterm and final exams or may be used later in the course.

General JavaScript Topic #1 – The rest operator (…)

The rest operator is used to **pass a variable number** of arguments to a function, and it must be **the last one** in the parameter list. If the name of the function parameter starts with the three dots, the function will get the rest of the arguments as **an array**:

1. Create 2 new folders called **week2\class1** and create a new file called **rest_example.js** with the following contents utilizing the **… rest** operator:

```
// sample function using REST operator ...
const calculateTotalCost = (id, name, ...costs) => {
  let totalCost = 0.0;
  // if no costs come in then we will get an empty an array
  costs.forEach((amount) => (totalCost += amount));
  // send the JSON back including the newly calculated total
  return {
    productId: id,
    productName: name,
    totalCost: totalCost,
  };
};

// define any number of costs related to a product
let mfgCost = 100.0;
let shipping = 12.99;
let taxes = 5.43;
let insurance = 3.22;

// Call the function and pass all variables
let productInfo = calculateTotalCost(
  1001,
  "Widget",
  mfgCost,
  shipping,
  taxes,
  insurance
);
let fmtCost = productInfo.totalCost.toLocaleString("en-US", {
  style: "currency",
  currency: "USD",
});
console.log(
  `product ${productInfo.productId} a ${productInfo.productName} has a total cost of ${fmtCost}`
);
```

2. Execute the new file from the terminal window

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node rest_example
product 1001 a Widget has a total cost of $121.64
```

So, we see the …costs parameter is treated like an array, and we can add any number costs as arguments when calling the function and it will handle it.

…now, back to the regular course material

Passing command line arguments in node

According to the site nodejs.org, **process.argv** is an array containing command line arguments. The first element will be 'node', the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments. Let's confirm this is the case:

3. Add a new JavaScript file called week2\class1**args_example1.js** with the following code:

```
// loop through command line arguments in process.argv array
let idx, val;
for ([idx, val] of process.argv.entries()) {
  console.log(`index is ${idx} : value is ${val}`);
}
```

4. Then change directories in the **terminal window** to today's folder (jsexercises\week2\class1) and execute the args_example1 code:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example1
index is 0 : value is C:\Program Files\nodejs\node.exe
index is 1 : value is E:\winter2023\info3139\programming\nodeexercises\week2\class1\args_example1
```

5. Then try the command again with more arguments:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example1 with some random arguments
index is 0 : value is C:\Program Files\nodejs\node.exe
index is 1 : value is E:\winter2023\info3139\programming\nodeexercises\week2\class1\args_example1
index is 2 : value is with
index is 3 : value is some
index is 4 : value is random
index is 5 : value is arguments
```

In our case the **process.argv** is the array we're interested in. Here we have confirmed that node.exe is the first parameter, the folder and file of execution is the second and arguments 3-6 are what would be considered data for a program.

6. Create another JavaScript file in the week2\class1 folder called **args_example2.js**.

7. Add some conditions in the code to check the arguments value. We are going to look for arguments that reflect some sort of **provincial code**. Insert the following code in args_example2.js:

```
let idx, val;
for ([idx, val] of process.argv.entries()) {
  if (idx > 1 && idx < process.argv.length) {
    // look any arguments after 2nd one
    switch (val) {
      case "on":
        console.log(`We've entered an argument for Ontario`);
        break;
      case "ab":
        console.log(`We've entered an argument for Alberta`);
        break;
      case "bc":
        console.log(`We've entered an argument for British Columbia`);
        break;
      default:
        console.log(`Argument ${val} is not a valid argument`);
        break;
    }
  }
}
```

8. Then run args_example2 with some valid and invalid arguments and see how the output varies:

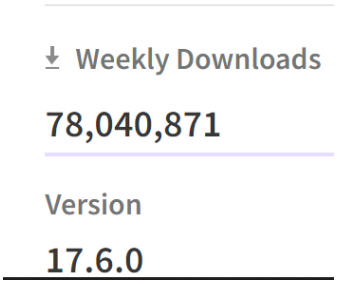
```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example2 on
We've entered an argument for Ontario
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example2 on ab bc
We've entered an argument for Ontario
We've entered an argument for Alberta
We've entered an argument for British Columbia
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example2 on ab bc qu
We've entered an argument for Ontario
We've entered an argument for Alberta
We've entered an argument for British Columbia
Argument qu is not a valid argument
```

With no arguments nothing is produced if we enter valid codes we get the expected results, and if we enter invalid codes we have that handled as well.An alternative syntax of the second example (uses something called **object literals**) could be something like:

```
// Alternative short-hand
const provinces = {
  on: "Ontario",
  bc: "British Columbia",
  ab: "Alberta",
};
let idx, val;
for ([idx, val] of process.argv.entries()) {
  if (idx > 1 && idx < process.argv.length) {
    provinces[val]
      ? console.log(`We've entered an argument for ${provinces[val]}`)
      : console.log(`Argument ${val} is not a valid argument`);
  }
}
```

yargs

We can do better than the default process.argv option. Next, we will next look at a popular npm package called **yargs.** At the time of this writing its statistics were:



So, like the got package we looked at last class, this package is a popular one. **yargs** will let us be a little more creative with our command line arguments. Before we look at using this package, we need to get it installed.

9. Back in the terminal pane issue the command **npm i yargs** and remember this must be done from the main **nodeexercises** folder (where the node_modules folder is, notice we don't need the full word install).

```
PS E:\winter2023\info3139\programming\nodeexercises> npm i yargs

added 16 packages, and audited 39 packages in 4s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
"dependencies": {
  "got": "^12.5.2",
  "yargs": "^17.6.0"
}
```

10. Return to the week2\class1 folder and create a new JavaScript file called **args_example3.js** with the following code:

```
import yargs from "yargs";
import { hideBin } from "yargs/helpers";
// Note: hideBin is a shorthand for process.argv.slice(2)
//       - bypass the first two arguments
const argv = yargs(hideBin(process.argv))
  .options({
    p1: {
      demandOption: true,
      alias: "province1",
      describe: "first province to compare transfer payments",
      string: true,
    },
    p2: {
      demandOption: true,
      alias: "province2",
      describe: "second province to compare transfer payments",
      string: true,
    },
  })
  .help()
  .alias("help", "h")
  .parse();

console.log(
  `you entered ${argv.p1} for province 1 and ${argv.p2} for province 2`
);
```

11. Then try running the new program with these options:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example3
Options:
      --version          Show version number                          [boolean]
      --p1, --province1  first province to compare transfer payments
                                                        [string] [required]
      --p2, --province2  second province to compare transfer payments
                                                        [string] [required]
  -h, --help             Show help                                    [boolean]

Missing required arguments: p1, p2
```

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example3 -h
Options:
      --version          Show version number                          [boolean]
      --p1, --province1  first province to compare transfer payments
                                                        [string] [required]
      --p2, --province2  second province to compare transfer payments
                                                        [string] [required]
  -h, --help             Show help                                    [boolean]
```

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example3 --p1 on
Options:
      --version          Show version number                          [boolean]
      --p1, --province1  first province to compare transfer payments
                                                        [string] [required]
      --p2, --province2  second province to compare transfer payments
                                                        [string] [required]
  -h, --help             Show help                                    [boolean]

Missing required argument: p2
```

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node args_example3 --p1 on --p2 ab
you entered on for province 1 and ab for province 2
```

To summarize, we have the ability to tell the user the arguments that the program requires, a little more powerful than the default process.argv.

Modules

We've briefly discussed using ESM modules (the system node is migrating to and the one that React uses), now we'll create one. This makes it easy to organize your code and make it re-usable.

12. The first routine we will create uses something called a **callback function**. An informal definition of a callback would be something like:

    *A callback is a function that is to be executed **after** another function has finished executing — hence the name 'call back'. A formal definition would be 'Any function that is passed as an argument to a function is called a **callback function**.'*

13. Create a JavaScript file called week2\class1**non_blocking_routines.js** with the following code:

```
// some valuable routine that we'll export so others can use it
const someRtnUsingOldSchoolCallback = (var1, callback) => {
  // make sure callback is a function
  if (callback && typeof callback === "function") {
    // check to see if var1 is err
    var1 === "err"
      ? // fire the callback either way
        callback("error happened in module", undefined) // simulate an error has occurred
      : callback("", { val1: "was", val2: "successful" });
  } else {
    console.log(
      "Error ==> no callback passed to someRtnUsingOldSchoolCallback"
    );
  }
};

// another valuable routine
const internalNameRtn = (callback) => {
  // make sure callback is a function
  if (callback && typeof callback === "function") {
    // we won't test for an error in this one
    callback(undefined, { val1: "was", val2: "successful" });
  } else {
    console.log("no callback passed to internalNameRtn");
  }
};

export { someRtnUsingOldSchoolCallback, internalNameRtn as anotherOldSchoolCallbackRtn };
```

14. Create another file called **old_school.js** with the following code, the bolded code is the callback function:

```
import {
  someRtnUsingOldSchoolCallback,
  anotherOldSchoolCallbackRtn,
} from "./non_blocking_routines.js";

let someParam = "no err";
// call module routine with no error and an
// anonymous callback function with 2 params
someRtnUsingOldSchoolCallback(someParam, (errorMessage, results) => {
  errorMessage // will be empty
    ? console.log(`Error ==> ${errorMessage}`)
    : console.log(`The call ${results.val1} ${results.val2}`);
});

// pass an err argument and see what happens
someParam = "err";
// same call as above
someRtnUsingOldSchoolCallback(someParam, (errorMessage, results) => {
  errorMessage // will have a value
    ? console.log(`Error ==> ${errorMessage}`)
    : console.log(`The call ${results.val1} ${results.val2}`);
});

// if don't pass a callback, an exception occurs
// so we need to catch it or the program stops
try {
  someRtnUsingCallback(someParam);
} catch (error) {
  console.log(`Error ==> ${error.message}`);
}

// now exercise the other library routine with just a callback
anotherOldSchoolCallbackRtn((errorMessage, results) => {
  errorMessage
    ? console.log(`Error ==> ${errorMessage}`)
    : console.log(`The other call ${results.val1} ${results.val2}`);
});
```

So, what is going on here?

- First in the non_blocking_routines.js, we have set the variable **someRtnUsingOldSchoolCallback** to equal an arrow function with **2 arguments**. The **var1** argument will be used to **receive some data** from the caller. The **callback** argument will be the **callback function** so we can **send data back** to the caller in a **non-blocking** manner.

- Next in the **oldSchool** file, the caller defines the actual callback function with 2 parameters. Having the error first and results second is known as a **callback interface**, but understand we are passing down the whole arrow function with the 2 parameters to the module

- The someRtnUsingOldSchoolCallback, then receives the arrow function from the caller and tests it to ensure that it is a function before invoking it. If it is a function, it then sees what the first parameter is. If the first parameter is the string **err** the callback is invoked with the string **'error happened'** else the callback is invoked with an undefined 1st parameter and a JSON object for the second parameter. If

the 2nd parameter is not a callback, we just log an error to the console indicating that.

- The caller then determines if there is a value in the returned data's 1st parameter. If an error condition has occurred an error message is written to the console. If there is no 1st parameter (undefined) a message is logged to the console using the data from the returned JSON.

- Note we imported using an alias in the **imports .. as**, this makes it available to others

Continuing with **internalNameRtn**

- It too is an arrow function that works in a similar fashion to the first routine. This time however, we import the routine **using an alias** in the syntax**.** The result of this, is that our code will only know about anotherOldSchoolCallbackRtn, someInternalName is never utilized inside the current module.
  .

15. Go to the terminal tab and change directories to the week2\class1 folder and test all 4 scenarios:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node old_school
The call was successful
Error ==> error happened in module
Error ==> someRtnUsingCallback is not defined
The other call was successful
```

Promises

Last class you were asked to do a bit of reading on Promises, we will add these now to our **non_blocking_routines.js** module

16. Return to the non_blocking_routines.js file and add the following code to it, **then update the exports to include the new method:**

```
// another routine, this time with a promise
 const someRtnWithAPromise = var1 => {
  return new Promise((resolve, reject) => {
    if (var1 === 'err') {
      // Reject the Promise with an error
      reject('some error');
    } else {
      // Resolve (or fulfill) the Promise with data
      let data = { val1: 'was', val2: 'successful' };
      resolve(data);
    }
  });
 };
```

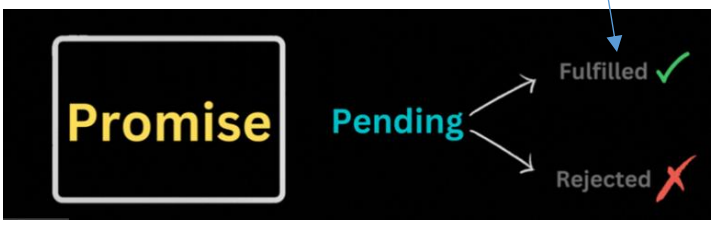
17. Create another file called **call_a_promise.js** with the following code:

```
import { someRtnWithAPromise } from "./non_blocking_routines.js";

let someParam = "no err";
// use promise
someRtnWithAPromise(someParam)
  .then((results) => {
    console.log(`The call ${results.val1} ${results.val2}`);
  })
  .catch((err) => {
    console.log(`Error ==> ${err}`);
  });

// call it again with an error
someParam = "err";
someRtnWithAPromise(someParam)
  .then((results) => {
    console.log(`The call ${results.val1} ${results.val2}`);
  })
  .catch((err) => {
    console.log(`Error ==> ${err}`);
  });
```

18. Execute **call_a_promise** to test the 2 scenarios:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node call_a_promise
The call was successful
Error ==> some error
```

Voila, we have re-worked the callback example with promises, it is a bit more elegant solution because it utilizes the intrinsic **resolve** function if it worked and **reject** function to indicate an error condition. From the homework, remember Promises have 3 states:



Also from the homework reading, Promises also give us the ability to **chain then(s)** and give you more control over the order the asynchronous calls will execute.

19. Create another method in the non_blocking_routines.js module with the following contents and **update the exports to include it**:

```
const anotherRtnWithAPromise = var1 => {
    return new Promise((resolve, reject) => {
      if (var1 === 'err') {
        // Reject the Promise with an error
        reject('yet another error');
      } else {
        // Resolve (or fulfill) the Promise with data
        let data = { val1: 'was', val2: 'more', val3: 'successful' };
        resolve(data);
      }
    });
  };
```

20. Create another file called **chain_of_promises.js** with the following content:

```
import { someRtnWithAPromise, anotherRtnWithAPromise } from "./non_blocking_routines.js";

let someParam = "no err";
// use promise rtn 1
someRtnWithAPromise(someParam)
  .then((results) => {
    console.log(`The 1st call ${results.val1} ${results.val2}`);
    // use promise rtn 2
    return anotherRtnWithAPromise(someParam);
  }) // here's the 2nd link in the chain
  .then((results) => {
    console.log(`The 2nd call ${results.val1} ${results.val2} ${results.val3}`);
    someParam = "err";
    return someRtnWithAPromise(someParam); // will fire catch
  })
  .then((results) => console.log("we never get here"))
  .catch((err) => {
    console.log(`Error ==> ${err}`);
    process.exit(1, err);
  });
```

21. Then, execute **chain_of_promises**:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node chain_of_promises
The 1st call was successful
The 2nd call was more successful
Error ==> some error
```

Note the way **we are exiting the routine** in both the successful and unsuccessful scenarios.

22. Add one more routine to the non_blocking_routines.js module called **ontarioTransferPaymentPromise** with the following code (do not forget to update the exports):

```
 // note we're using.then/.catch syntax here
const ontarioTransferPaymentPromise = () => {
  let srcAddr =
    "http://www.infrastructure.gc.ca/alt-format/opendata/transfer-program-programmes-de-transfert-bil.json";
  return new Promise((resolve, reject) => {
    got(srcAddr, { responseType: "json" })
      .then((response) => {
        let ont = response.body.ccbf.on["2022-2023"];
        resolve(ont);
      })
      .catch((err) => {
        console.log(`Error ==> ${err}`);
        reject(err);
      });
  });
};
```

Notice how we are using the got package in the promise as such you will need to **add the import at the top of the module** to access it. Also, notice the syntax is now using the **.then/.catch** syntax instead of try/catch and async/await that we used in week 1 (we will eventually go back and use try/catch after the next class when we cover the async/await in more depth, **but for today use the .then .catch syntax**)

23. Create a new file called **ontario_transfer_payment.js** with the following code:

```
import { ontarioTransferPaymentPromise } from "./non_blocking_routines.js";

// Create a currency formatter.
const currencyFormatter = (numberToFormat) =>
  new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 0,
  }).format(numberToFormat);

// call promise that utilizes the got package
ontarioTransferPaymentPromise()
  .then((ontariosPayment) => {
    console.log(
      `Ontario's transfer payment is: ${currencyFormatter(ontariosPayment)}`
    );
  })
  .catch((err) => {
    console.log(`Error ==> ${err}`);
  });
```

24. Finally execute the **ontarioTransferPayment**:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1> node ontario_transfer_payment
Ontario's transfer payment is: $853,621,164
```

Lab 2 – 2.5%

Part A - (2%)

Code a mini application that brings together numerous techniques we have covered last week and today:

- Create a JavaScript module file called **lab2_routines.js** that **starts out** with the following object array, currency formatter, and constant for the fiscal year (you will also be adding 3 promise-based functions as well):

```
import got from "got";

const provinces = [
  { code: "NS", name: "Nova Scotia" },
  { code: "NL", name: "Newfoundland" },
  { code: "NB", name: "New Brunswick" },
  { code: "PE", name: "Prince Edward Island" },
  { code: "QC", name: "Quebec" },
  { code: "ON", name: "Ontario" },
  { code: "MB", name: "Manitoba" },
  { code: "SK", name: "Saskatchewan" },
  { code: "AB", name: "Alberta" },
  { code: "BC", name: "British Columbia" },
  { code: "NT", name: "North West Territories" },
  { code: "NU", name: "Nunavut" },
  { code: "YT", name: "Yukon Territory" },
];

const FISCALYEAR = "2022-2023";

// Create a currency formatter.
const currencyFormatter = (numberToFormat) =>
  new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 0,
  }).format(numberToFormat);

export {
  provinces,
  currencyFormatter,
};
```

Create a JavaScript file called **lab2.js** that does the following:

1. Takes 3 **mandatory** arguments, **use yargs** to ensure all 3 arguments are passed. The arguments are to have the following characteristics:
    1. First Name
        - --firstname or --fname
        - Resident's first name
        - string
        - required
    2. Last Name
        - --lastname or --lname
        - Resident's last name
        - string
        - required
    3. Provincial Code
        - --province or --prov
        - Must be one of the code values in the module's provinces array. The provinces array **must only reside in the module**, not in app.js
        - Resident's home province
        - string
        - required

2. Execution of the program **without any of the required arguments** should result in the following:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1\lab2> node lab2
Options:
      --version             Show version number                        [boolean]
      --firstname, --fname  Resident's first name           [string] [required]
      --lastname, --lname   Resident's last name            [string] [required]
      --province, --prov    Resident's home province
   [string] [required] [choices: "NS", "NL", "NB", "PE", "QC", "ON", "MB", "SK",
                                 "AB", "BC", "NT", "NU", "YT"]
  -h, --help                Show help                                  [boolean]

Missing required arguments: firstname, lastname, province
```

3. Execution **with the correct parameters** should result in something like:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1\lab2> node lab2 --fname Evan --lname Lauersen --prov NB

Lab 2

Evan, Lauersen lives in New Brunswick. It received $47,147,924 in transfer payments.
```

And should be dynamic enough to handle entering a different first names, last names, and provinces:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1\lab2> node lab2 --fname Jane --lname Doe --prov BC

Lab 2

Jane, Doe lives in British Columbia. It received $293,162,621 in transfer payments.
```

4. Code **lab2.js** so it contains a single **chained function** that uses the 3 required arguments (first name, last name, and provincial code). This **chained function** will in turn call **3 promise-based** routines to be coded in the lab2_routines module **in succession** (use .then/.catch syntax). Using a series of console.log statements produce the output shown above in step 3.

The sequence for lab2.js execution will be as follows:

- Call a promise-based module function with all 3 parameters; first name, last name, and provincial code, and receive a string back, e.g.

  ```
  Evan, Lauersen lives in Quebec.
  ```

- Call another promise-based module function with no parameters and receive back a JSON object containing the GOC transfer data (utilize the got package in the module function for this one but remember to use the .then/.catch syntax).

- Call a 3rd module function, using the GOC JSON and the provincial code and receive a formatted transfer payment for that province.

- We will also have a need to confirm that the user has entered a valid provincial code argument using yargs. To supply an array of choices you can use the **choices property** as discussed in this article: http://yargs.js.org/docs/#api-reference-choiceskey-choices

3 Promise based functions for **lab2_routines.js**.

- **Promise function 1's** role is to build a string consisting of the full name and province of residence. it should receive 3 parameters and resolve to a template expression with the format: **firstname, lastname lives in province.** Make sure you **do not use concatenation** (e.g. the + sign), use **template expressions** and make sure you use the **province name not province code**

- **Promise function 2** is to resolve the GOC's transfer payment data as JSON. You can nest the required got call inside of the promise like we did in the ontarioTransferPaymentPromise example.

- **Promise function 3** is to resolve to a template literal string consisting of a formatted transfer payment amount for the province

```
26 > const fullNameAndProvincePromise = (fname, lname, provinceCode) => {···
35   };
36
37 > const transferPaymentsFromWebPromise = () => {···
49   };
50
51 > const transferPaymentForProvincePromise = (gocData, provCode) => {···
61   };
```

Submit 3 Screen shots in a single Word doc:

1. A screen shot of the execution of app.js with only **your first name** (should result in yargs complaining that the last name and province arguments are missing)

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1\lab2> node lab2 --fname Evan
Options:
      --version             Show version number                            [boolean]
      --firstname, --fname  Resident's first name            [string] [required]
      --lastname, --lname   Resident's last name             [string] [required]
      --province, --prov    Resident's home province
   [string] [required] [choices: "NS", "NL", "NB", "PE", "QC", "ON", "MB", "SK",
                                 "AB", "BC", "NT", "NU", "YT"]
  -h, --help                Show help                                      [boolean]

Missing required arguments: lastname, province
```

2. A screen shot of the execution of lab2.js with **your first name, last name,** and **MB** as the province

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1\lab2> node lab2 --fname Evan --lname Lauersen --prov MB

Lab 2

Evan, Lauersen lives in Manitoba. It received $75,806,775 in transfer payments.
```

3. Then repeat with **SK** as the province:

```
PS E:\winter2023\info3139\programming\nodeexercises\week2\class1\lab2> node lab2 --fname Evan --lname Lauersen --prov SK

Lab 2

Evan, Lauersen lives in Saskatchewan. It received $65,415,534 in transfer payments.
```

4. Also include the source from **lab2.js** and **lab2_routines.js in the same Word document**. Remember to convert any got processing to use the**.then/.catch syntax remove any async/await** from both files for this lab

Part B - (.5%) - Lab 2 Theory Quiz on FOL - 10 questions

Review

1. If a module is in the same folder as the routine that needs the module, what syntax would you use to utilize it?
2. How many arguments were in our "callback interface"?
3. What are they?
4. What does the process.argv array contain?
5. What array method did we use to process the entire process.argv array?
6. What elements are not passed as arguments?
7. What is yargs?
8. What command did we use to install yargs?
9. What yargs helper method did we use to bypass the first 2 arguments?
10. What do we pass to the **.options** property of yargs?
11. What option setting can I use to give an alternative name to an argument?
12. If I run my module without any arguments, and I've set the module up with yargs what will I see upon execution?
13. What yargs option did we use to limit the province code to ON or AB?
14. What are the 3 states of a promise?
15. What is the main benefit of using promises?
16. What 2 callbacks do we provide to the promise?
17. What code in the caller of the Promise handles the resolve callback?
18. What code in the caller of the Promise handles the reject callback?

Homework

Read over the following link:

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await