INF221 Lecture 2 - Lab Practice Exercises

1) Introduction

   Read more: conditioanl operators, loops, conditional statements and data value types.

2) Variables

   A variable is a name that can reference a specific value. Variables are declared using var,let or const followed by the variable's name. Here are some examples:

```
var example;

let example;

const example;
```

   The above variable is declared, but it isn't defined (it does not yet reference a specific value). Here's an example of defining a variable, making it reference a specific value:

```
var number = 12;

let myString = "some string";

const example = "some thing";
```

   NOTE A variable is declared using var, let, const and uses the equals sign to define the value that it references. This is known as variable initialization.

The challenge:

A. Create a file named index.html with the following content.

```
<!DOCTYPE html>

<html>

   <head>

       <title>My website</title>

   </head>

   <body>

       <h1>My website</h1>

       <p>This website contains nothing</p>

   </body>

</html>
```

B. We execute JavaScript in the browser by linking to it from an external file, with a script tag, like so:

```
<!DOCTYPE html>

<html>

   <head>

       <title>My website</title>

       <script src="script.js" lang="javascript"
type="text/javascript">

           // Javascript code...

       </script>

   </head>

   <body>

       <h1>My website</h1>

       <p>This website contains nothing</p>

   </body>

</html>
```

C. Else you can do the same in the `index.html`. In the `index.html` file between the `body` and `html` **closing** tags, put the following script tags `<script type="text/javascript">/* javascript code... */</script>`. Between the js tags declare the above variables. Make the variables equal to the value of choice. Then use `console.log()` to print each value to the console when the page loads.

3) Functions

Javascript supports both named and anonymous functions.

- Named functions:
  1. Normal way of declaring functions.

```
function myFunction() {
   // some code here
   console.log("My named function");
}
```

  2. Second way.

```
const myFunction = function() {
   // some code here
   console.log("My second named function");
}
```

  3. As arrow functions, es6.

```
// es6 way
const myFunction = () => {
   // some code here
   console.log("My second named function");
}
```

❖ Anonymous functions:

```
// anonymous function
(function() {
   // some code here
   console.log(confirm("Anonymous function?"));
})(); // note how the function is called
```

Note that functions can accept parameter values.

1. Passing values to functions

```
// find the maximum of two numbers
const max = (val1, val2) => {
    if(val1 !== 0 && val2 !== 0) {
        if(val1 > val2) {
            return val1;
        } else if(val1 === val2) {
            return 'equal';
        } else {
            return val2;
        }
    }
};
```

2. Passing functions as arguments.

```
// take two numbers and perform addition on them
```

```
const addition = (a, b) => {
   return a + b;
}
// take two numbers and perform addition on them
const subtraction = (a, b) => {
   return a - b;
}
// passing functions as parameters
const perform = (callback) => {
   return callback;
}
```

Then call `perform` function with necessary parameter values. i.e.

```
console.log(perform(add(12, 78));
console.log(perform(subtraction(1, 8));
```

3. Recursion (read more)

   Is when a function calls itself many times. Recursion is a programming pattern that is useful in situations when a task can be naturally split into several tasks of the same kind, but simpler. Or when a task can be simplified into an easy action plus a simpler variant of the same task. Or, as we'll see soon, to deal with certain data structures. Ie. `fibonacci` sequence etc.

   ```
   const fibonacci = (element) => element <= 1 ? 1 :
   fibonacci(element-1) + fibonacci(element-2);
   ```

   I. How different is recursion from iteration? Write any two functions for each to show the difference (Hint: `loops`).

The challenge:

Inside the `script.js` file, modify the `perform` function to accept two parameter values, the first one a function and the second one a list of integers (array). The function should look as below:

```
/**
* Here we have created a function that takes
* some list of something and a function as argument.
* It will go through the list and apply the provided function on each element,
* storing the returned value from the function in results.
*/
const perform = (callback, list) => {
    let results = [];
    for (let index = 0; index < list.length; index++) {
        results.push(callback(list[index]))
    }
    return results;
}
```

I. Write a function to find the factorial of a number when passed to it. Then call the `perform` function on: the `factorial` and any number of elements in an `array` passed as arguments.
II. Use the function to find the `fibonacci` sequence of each element in an `array`.

4) Data structures: Arrays and Objects

Numbers, Booleans, and strings are the basic atomic values from which data structures are built. In javascript `Objects` and `Arrays` are some of the best ways of representing these data structures. Arrays stores sequences of values, written as a list between square brackets, separated by commas. JavaScript arrays are zero-indexed: the first element of an array is at index `0`, and the last

element is at the index equal to the value of the array's `length` property minus 1. Using an invalid index number returns `undefined`.

```
1.    // array definition and accessing elements

let names = ['John', 'James', 'Joyce', 'Jane'];

// length

console.log(names.length); // log  4

// access element at index 0

console.log(names[0]); // log John

// access element at index 2

console.log(names[2]); // log Joyce

// access element at index 5

console.log(names[5]); // undefined, why?


2.    // accessing elements using some of the default methods

let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];

console.log(vegetables);

// ["Cabbage", "Turnip", "Radish", "Carrot"]

let pos = 1, n = 2;

let removedItems = vegetables.splice(pos, n);

// this is how to remove items, n defines the number of items
to be removed, note the use of splice.

// from that position(pos) onward to the end of array.

console.log(vegetables);

// ["Cabbage", "Carrot"] (the original array is changed)

console.log(removedItems);

// ["Turnip", "Radish"]
```

`Objects` allow us to group values, including other objects, to build more complex structures. These are a collection of key-value pairs.

```
// object definition
let course = {
   id: 0,
   name: 'INF221',
   description: 'Web Design and Development',
}
// access a variable
console.log(course.name);
// change value of variable
console.log(course.name = 'Web Development');
```

We can at any time add new variables to the object.

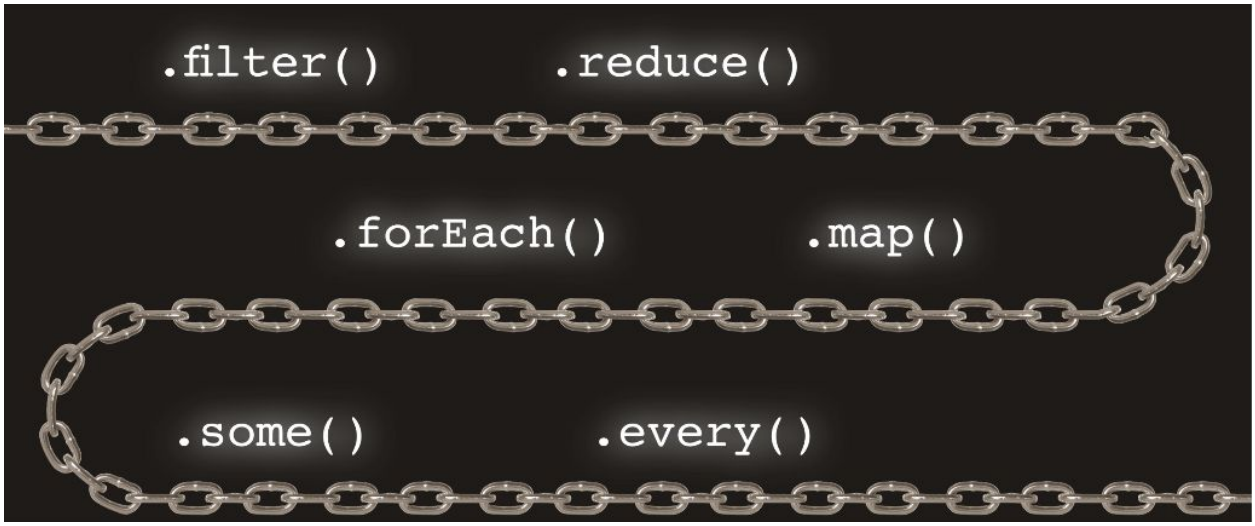
```
// add a new variable # of students and initialize it with 62
course.numberOfStudents = 62;
console.log(course);
```

You can define even more complex objects or arrays. The following is an array of objects.

```
// list of course objects
let courses = [
   {
       id: 0,
       name: 'INF221',
       description: 'Web Design and Development',
       numberOfStudents: 62
```

```
    },
    {
        id: 1,
        name: 'COM121',
        description: 'Intro. to Computer Programming',
        numberOfStudents: 200
    },
    {
        id: 2,
        name: 'COM222',
        description: 'Advanced Programming',
        numberOfStudents: 80
    },
];
```

In javascript, you will mostly work with `Arrays` or `Objects` and a combination of both to create very complex objects or arrays. ***Objects can also contain functions***. The following are some of the useful methods that can be called on arrays and objects.



The challenge:

Read more on the following ***before coming to class and lab tomorrow***: `forEach`, `map`, `reduce`, `filter higher-order functions` and `rest parameters, JSON, closures` and `array` or `object destructuring`.

I. Using `map` function print each of the objects contained in the array object.
II. Using a function, determine which course object has more students than the rest.
III. Calculate the total number of students for all courses by summing them together.
IV. Add a new function to `courses` and call it `summary`, the function should return a combination or concatenation of any variables in each object as brief summary.

5) Higher-order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. They allow us to abstract over actions, not just values. They come in several forms. For example, you can have functions that create new functions. Note in the following example.

```
// higher order function that returns an object of functions

const CustomMath = (() => {

   /**

    * Here we have created a function that takes

    * some list of something and a function as argument.

    * It will go through the list and apply the provided function on
each element,

    * storing the returned value from the function in results.
```

```
 */
const perform = (callback, list) => {
    let result = [];
    for (let index = 0; index < list.length; index++) {
        result.push(callback(list[index]))
    }
    return result;
};

// fibonacci
const fibonacci = (element) => element <= 1 ? 1 :
fibonacci(element-1) + fibonacci(element-2);

// find the maximum of two numbers
const max = (val1, val2) => {
    if(val1 !== 0 && val2 !== 0) {
        if(val1 > val2) {
            return val1;
        } else if(val1 === val2) {
            return 'equal';
        } else {
            return val2;
```

```
        }
    }
};


// factorial of a number
var factorial = function(number) {
    if (number <= 0) { // terminal case
        return 1;
    } else { // block to execute
        return (number * factorial(number - 1));
    }
};


// min between two numbers
const min = (val1, val2)  => {
    if (val1 < val2) {
        return val1;
    } else {
        return val2;
    }
};
```

```
    // take two numbers and perform addition on them
    const addition = (a, b) => {
        return a + b;
    };

    // take two numbers and perform addition on them
    const subtraction = (a, b) => {
        return a - b;
    };
    // return new functions
    return {
        perform,
        fibonacci,
        max,
        factorial,
        min,
        addition,
        subtraction,
    }

})();
// try to log the following to console
```

```
console.log(CustomMath.addition(23, 34));

console.log(CustomMath.factorial(4));

console.log(CustomMath.max(34, 12));

console.log(CustomMath.perform(CustomMath.fibonacci, [2,3,5,9,10]));
```

The challenge:

What is the difference between Higher order functions and closures?

6) The Document Object Model(The DOM)

When an HTML page or file is loaded in the browser, the document object structure is created. Forming a tree like structure of objects where every HTML-element is store in nodes. It is through these nodes that we define the behaviour of our page. Through javascript we can:

- Change the content of the html elements.
- Change the style(css) of html elements.
- React to html DOM events.
- Add and delete html elements.

The HTML DOM standard defines methods(actions) which act on html elements and properties(values of elements) that can be set or changed. Look at the following html structure.

```
<html>

   <head>

       <title>My Site</title>

       <script>

        // the statement changes the content(innerHtml) of the <div>

        // element with id='myDiv'
```

```
        document.getElementById('myDiv').innerHTML = 'My div hello';
    </script>
</head>
<body>
    <div id="myDiv"></div>
    <ol class='items'></ol>
    <ul id="list"></ul>
    <input type="button" id="btn_id" value="Register" />
</body>
</html>
```

In this example, **getElementById** is an action or method and **innerHtml** is a property. The method in this case is used to get the html element using its id. And the property is used to get the content of an element; thus useful for getting or replacing the content of html elements including <html> and <body>.

Note that the **document** object represents your web page. The document object helps us to access any element in the html page, by always starting with it. The examples below demonstrates the point.

- `let ul = document.getElementById('list');`
- `let items = document.getElementsByClassName('items');`
- `let tag = document.getElementsByTagName('ul');`

If you want to find all html elements matching some CSS selector(id, class names, types, attributes, values of attributes, etc) use `document.querySelectorAll(selector)` method.

```
const selector = document.querySelectorAll('ul.list') // returns a list of elements matched
```

Also the **document** object helps us to add and change values of html elements. The examples below demonstrates the point.

- ```
  let ul = document.getElementById('list');

  // creating new element
  ```
  ```

```
let li = document.createElement('li');

// changing values of elements

li.style.color = 'red'; // change style of element

li.innerHTML = "John"; // change inner html of an element

li.setAttribute('id', 'name'); // change attr. value

// then append li element to the ul

ul.appendChild(li); // add html element
```

- ```
  // write into the html output stream

  document.write(text);
  ```
- ```
  // replace an element

  document.replaceChild(newChild, oldChild);
  ```

We so far seen how to access, add and change html elements in the document structure. Next we shall add behaviour to this stucture we have defined. The following syntax defines the signature of adding event handlers to elements:

```
// adding event handlers to an event:onclick,onload,onmouseover etc.

document.getElementById(id).onclick = function() { /** code here */ }
```

We can also add event listeners that fires when a user interacts with html elements.

```
// add event listeners to html elements

// listener can be any callback function

document.getElementById('btn_id').addEventListener('click',
listener);

// or

const button = document.getElementById('btn_id');

button.addEventListener('click', listener);
```

The `document.addEventListener()` method attaches an event handler to the specified element. Many event listeners(*whether of the same type*) can be set to the same html element without overwriting each other. Using ths method controls how event reacts to **bubbling** or **capturing**(*Read about events propagation*). Events can also be removed using `document.removeEventListener()`. For more information on events look up the complete guide HTML DOM Event Object Reference.

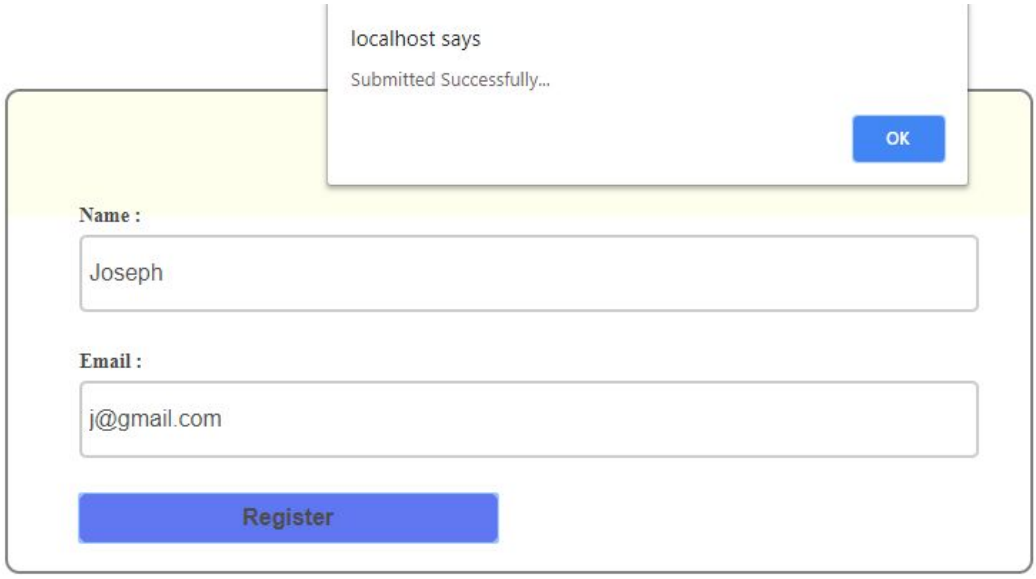
Node Relationship: the nodes in the node tree have a hierarchical relationship to each. These relationships a described by parent, child, and sibling(***Read more***).

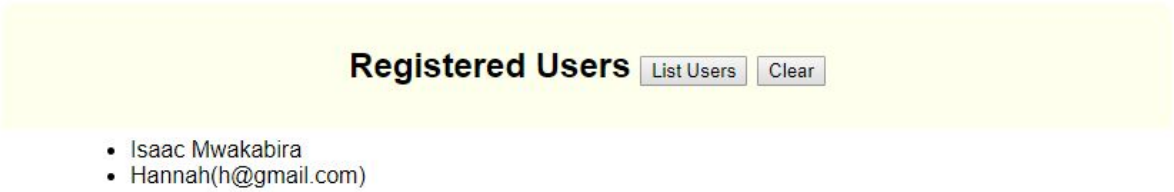
The challenge:

In this weeks challenge we will be developing a form and adding some behaviour to it using javascript. We have provided you the initial program code that contains the basic html form structure plus some styling applied to it. Your job is to define the necessary behaviour by faithfully following this tutorial step by step until all required functionality has been developed. All javascript functions **must** be written in the `form.js` file.

Note that `index.html` contains the html structure, `form.css` contains CSS styles and `script.js`, `module.js` and `form.js` contains the javascript functionality for our respective pages.

We will be developing a form that allows any user to input the user's name and email, then store those details to the local storage and finally load all details, for all users added, to the page as an unordered list of items. The picture below shows how the final product shall look like.





1. **Step zero.**

   Make sure you have pulled `git pull` all files from the `course-material` repository on github, and inside the index.html you have the following content structure.

```
<!DOCTYPE html>

<html>

    <head>

        <title>Registration form</title>

        <!-- Include CSS File Here -->

        <link rel="stylesheet" href="form.css"/>

        <!-- Include JS File below Here -->
```

```
    </head>

    <body>

        <div class="container">

            <div class="section" style="width: 80%">

                <div class="main">

                    <h2>Registration Form</h2>

                    <form action="#" method="post" name="form_name"
id="form_id" class="form_class">

                        <label>Name :</label>

                        <input type="text" name="name" id="name"
placeholder="Name" />

                        <label>Email :</label>

                        <input type="text" name="email" id="email"
placeholder="Valid Email" />

                        <input type="button" id="btn_id" value="Register"/>

                    </form>

                </div>

            </div>

            <div class="section" style="width: 80%; margin-top: 5%;">

                <h2>

                    Registered Users

                    <span>

                        <button>List Users</button>

                        <button>Clear</button>

                    </span>

                </h2>

                <ul id="list_of_users">
```

```
                    <!--first item is not coming from the localStorage:
note-->
                    <li>Isaac Mwakabira</li>
                </ul>
            </div>
        </div>
    </body>
</html>
```

2. **Step one**

   Include the `form.js` file within the `<head>` below the css file by typing the following: `<script src="form.js" type="text/javascript" lang="javascript"></script>`. All javascript functions must be put in here.

3. **Step Two**

   Write the following function to check if both form input fields are not empty before they can be submitted.

```
// form input fields validation
const validation = () => {
    // get element's values using their ids
    var name = document.getElementById("name").value;
    var email = document.getElementById("email").value;
    if( name === '' || email === ''){
        alert("Please fill all fields...!!!!!!");
        return false;
    } else {
        return true;
    }
```

```
    }
```

4. **Step Three**

   Write the following function that creates our store, to which all users are be stored. Then stores a single added user to local storage.

```
// function stores user to storage

const storeUser = (user) => {

   // get localStorage

   var storage = window.localStorage.getItem('users');

   // check first if storage exist

   // else create new and call it `users`

   var users = [];

   if (storage === undefined && storage === null) {

       users.push(user);

       window.localStorage.setItem('users', JSON.stringify(users));

   } else {

       // create an array where to store the added values

       // otherwise we do not want to replace the old values with
new ones.

       if(storage !== undefined && storage !== null) {

           if(storage.length !== 0) {

               users =
JSON.parse(window.localStorage.getItem('users'));

           }
```

```
            }

            // then add new user to the list

            users.push(user);

            // here we are storing the users to local storage

            window.localStorage.setItem('users', JSON.stringify(users));

        }

    }
```

Then write a function called `nullify(fieldIds)` that clears and set all previously typed input text values to null so that the we can register another person.

```
// sets all form fields to null

// make sure that the type of para fields is an Array

const nullify = (fieldIds) => {

   // here write the logic to clear the fields...

}
```

5. **Step Four**

   The following function creates an `<li>` element to be appended to the list of users items in the `<ul id="list_of_users">` (unordered list) element.

```
// function creates a single li element and set the innerHtml property to user's details ie. <li>Isaac(imwakabira@ymail.com)</li>

const singleUserListElement = (details) => {

   // create li element which will hold the user's details(name and email)

   let item = document.createElement('li');
```

```
    // then set user details to innerHtml property

    item.innerHTML = details;

    // then finally return this li element ie
<li>Isaac(imwakabira@ymail.com)</li>

    // to be appended to the parent ul element when loaded to the document
page

    return item;

}
```

Then create a function called `loadAndListAllUsers()` that loads all added users from local storage to the `<ul id="list_of_users">` element in the page using the above `singleUserListElement(details)` function.

```
const loadAndListAllUsers = () => {

      // here logic to load users from local storage...

}
```

And add this function as an `onclick` event listener handler on the button `List Users` as follows: `<button onclick="loadAndListAllUsers()">List Users</button>` in the index.html file.

6. **Step Five**

Write the following function that should be called upon submittion of the registration form by first validating the form using the function we defined above, .

```
// register a single user and load to page list items

const register = () => {

   var name = document.getElementById("name").value;
```

```
    var email = document.getElementById("email").value;
    if (validation()) {
        // define the javascript object to be stored in local
storage
        const user = {
            name: name,
            email: email
        }
        // store the added user to local storage
        storeUser(user);
        // alert user that submission was a success
        alert("Submitted Successfully...");
        // then nullify all fields in the form using thier ids
        nullify(['name', 'email']);
        // load all users to page from local storage
        loadAndListAllUsers();
    }
}
```

Then inside the `index.html` set an onclick event listener as follows on the form register button, `<input type="button" name="submit_id" id="btn_id" value="Register" onclick="register()" />`.

7) AJAX(`Reading assinment`)

Refer to the code given in `fetch()` function inside the `module.js` file.