CMPS 350 Web Development Fundamentals
Lab 5-6 –JavaScript Fundamentals and Unit Testing

Objective

The objective of this lab is to practice the following JavaScript Fundamental Programming features including:

- **Functional programming** including:
  - **Arrow functions**
  - **Array methods** (map, reduce, filter, flat, splice, sort…)
  - **Spread operator**
- **Object literals**: comma-separated list of name-value pairs and associated functions wrapped in curly braces.
- **Classes**: create classes and use them to instantiate objects.
- **Inheritance**
- **Modules**: export and import modules.
- **Unit Testing**

This Lab has three parts:

- **PART A:** Practice arrays and control structures.
- **PART B**: Banking App (duration: 1h20mins).
- **PART C:** Unit Testing

PART A – Practice Exercises

Exercise 1 : Arrays and Control Structures

```
let matrix = [ [2, 3], [34, 89], [55, 101, 34], [34, 89, 34,
   99]];
```

Use the above array and implement and test the following functions:

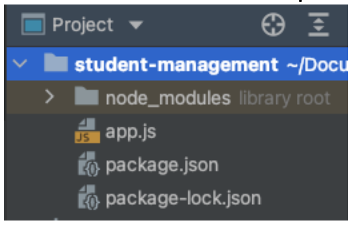
- **`flatten`:** gets a matrix (i.e., array of arrays) and returns a single dimensional flat array.
- **`max`**: gets an array and returns its maximum value (tip: use reduce function).
- **`sort`:** gets an array and returns a sorted array in descending order (from big to small).
- **`square`:** gets an array and returns an array with squared values.
- **average:** gets an array and returns its average.
- **`removeDuplicates`:** gets an array and returns an array without duplicate elements.
- **`filter`:** Find the sum of all the number in the array that are greater than 40. You should write everything as one single statement.

Expected output:

```
Original array:
[ [ 2, 3 ], [ 34, 89 ], [ 55, 101, 34 ], [ 34, 89, 34, 99 ] ]

Flattened:
[ 2, 3, 34, 89, 55, 101, 34, 34, 89, 34, 99 ]

Max value:
101

Sorted in descending order:
[ 101, 99, 89, 89, 55, 34, 34, 34, 34, 3, 2 ]

Sum of elements greater than 40:
433

Unique elements:
[ 2, 3, 34, 89, 55, 101, 99 ]

Sum of unique elements:
383

Square of unique elements:
[ 4, 9, 1156, 7921, 3025, 10201, 9801 ]
```

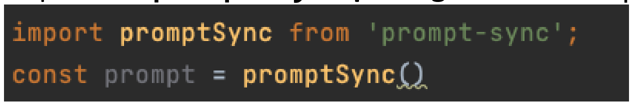
Exercise 2 : Objects

1. Create a new project named **student-management**
2. Create a JavaScript file named **`app.js`**
3. Open your terminal and type the following command **`[ npm i prompt-sync ]`** in order to get the **`'prompt-sync'`** NPM package, that allows you to read user input from the terminal. This command will create multiple files inside your project. We will talk about those files in-depth when we discuss about Node.js.



```
student-management ~/Docu
  node_modules library root
  app.js
  package.json
  package-lock.json
```

4. Open the `package.json` and add the following line `[`**`"type" : "module"],`**
5. Import the **`prompt-sync` package** inside the `app.js`



```
import promptSync from 'prompt-sync';
const prompt = promptSync()
```

6. Create an array named **students**
   Create a loop that asks the user to add 5 students. You should only ask the user the **name** and **gender** of the student. The age and grade of the students should be randomly generated. The age should be between [**17 - 35**] and the grade should be between [**0 and 100**] inclusive.

Sample : the students array after adding the five students should look something like this

```
[
  { name: 'Hani', gender: 'Male', grade: 92, age: 19 },
  { name: 'Ahmed', gender: 'Male', grade: 81, age: 30 },
  { name: 'Sara', gender: 'Female', grade: 99, age: 24 },
  { name: 'Emany', gender: 'Female', grade: 50, age: 26 },
  { name: 'Issa', gender: 'Male', grade: 70, age: 24 }
]
```

7. The write the code that answers the following questions
   a. Find the student with the highest grade
   b. Find both the youngest and oldest student in the class
   c. Find the average and median student age in the class
   d. Calculate the mean and variance of the student grades.
   Note : Do not use the traditional loops to solve the above questions. You should use the arrow function.

Mean	**Variance**
$\boldsymbol{\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i}$	$\sigma^2 = \frac{1}{n-1}\sum_{i=0}^{n-1}(x_i - \bar{x})^2$

PART B – Classes and Modules

In this exercise you will build a simple banking app according to the design shown in Figure 1.

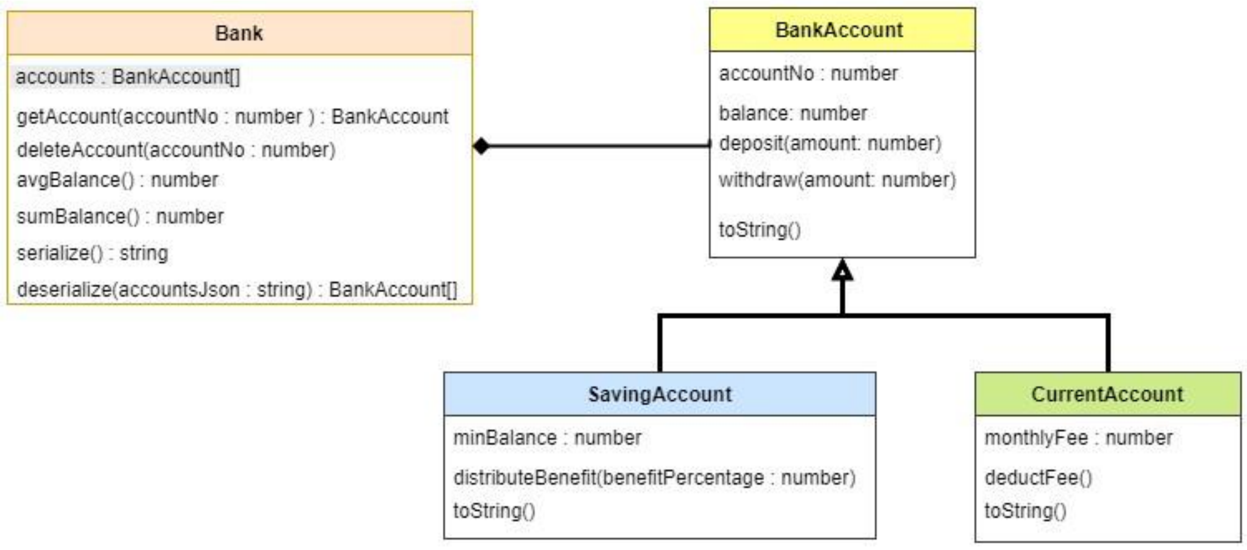


Figure 1. Banking App Class Diagram

1) Create **BankAccount** class with the following private properties: `accountNo` and `balance`. The account no, should be randomly generated. But the balance must be initialized through the constructor of the class. In addition, the class should have the following methods:

- getters for both accountNo and balance
- `deposit(amount)`: this method adds the amount to the balance
- `withdraw(amount)`: this method subtracts the amount from the balance
- `toString()`: this method return Account # **accountNo** has QR **balance**. e.g., Account #123 has QR1000.

Export the **`BankAccount`** class as a module.

2) Create `app.js` program. Declare **accounts** variable array and initialize it with the following accounts:

accountNo	**balance**
123	1000
234	4000
345	3500

Display the content of the **accounts** array.

3) Create **`SavingAccount`** class that extends `BankAccount` with an extra property: minBalance and an extra method `distributeBenefit(benefitPercentage)`. This method computes the monthly benefit using the balance += (balance * benefitPercentage). The constructor should extend BankAccount to initialize the `minBalance`. Also, extend the `toString()` to indicate that this is a Saving Account. e.g., e.g., **Saving** Account #123 has QR1000.

Test SavingAccount in app.js using the same table above and use a minimum balance of 500 for all accounts.

4) Create **`CurrentAccount`** class that extends `BankAccount` with an extra property: `monthlyFee` and an extra method `deductFee()`. This method subtracts the `monthlyFee` from the account balance. The constructor should extend BankAccount to initialize the monthlyFee. Also, extend the `toString()` to indicate that this is a Current Account. e.g., e.g., **Current** Account #123 has QR1000.

   Test **`CurrentAccount`** in app.js using the same table above and use a monthly fee of 10 for all accounts.

5) Create **Bank** class to manage accounts. It should have **`accounts`** property to store the accounts. Also, it should have the following methods:

Method	**Functionality**
`add(account)`	Add account (either Saving or Current) to accounts array.
`getAccount(accountNo)`	Return an account by account No

`deleteAccount(accountNo)`	Delete an account by account No
`avgBalance()`	Get the average balance for all accounts
`sumBalance()`	Get the sum balance for all accounts
`toJson()`	Return accounts as a JSON string
`fromJson(accountsJson)`	Takes JSON string representing accounts and returns an array of accounts.

6) Create app.js program. Declare an instance of Bank class then add the following accounts:

accountNo	balance	type	minimumBalance	monthlyFee
123	500	Saving	1000	
234	4000	Current		10
345	35000	Current		15
456	60000	Saving	1000	

- Test all the Bank methods described above.
- Display the total balance of all accounts.
- Increase by 5 the monthly fee of all the **Current** accounts then charge the monthly fee.
- Display the total balance of all accounts after charging the monthly fee.
- For all the **Saving** accounts distribute the benefit using a 5% benefit.
- Display the total balance of all accounts after distributing the benefits.

Part B – Unit Testing Using Mocha and Chai

1. Sync cmps350-lab repo to get the Lab files.
2. Copy ***Lab6-JS OOP*** folder from cmps350-lab repo to your repository.
3. Open ***Lab6- JS OOP \UnitConverter.js*** in Webstorm. You should see a JavaScript file named *UnitConverter.js*. In this exercise, you will create a spec file to unit test the function of the *UnitConverter* class.
4. First, create the package.json file using **npm init**. This file is used to define dependencies by listing the npm packages used by the app.

   Refresh your project to see the **package.json** file.
5. Install mocha and chai using *node package manager* (npm):
   ```
   npm install mocha -D
   npm install chai -D
   ```
   This will add 2 dev dependencies to package.json file.
6. Create a JavaScript file named **UnitConverter.spec.js**
7. Import an instance the *UnitConverter* class to be tested and the *chai expect* package.

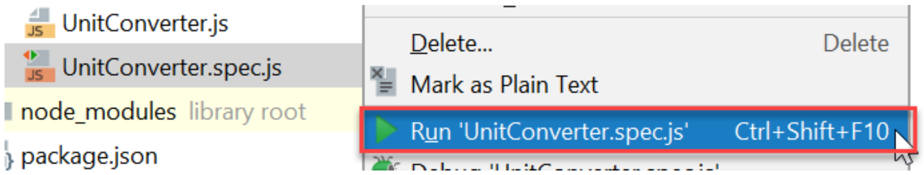
```
import unitConverter from './UnitConverter.js';
import {expect} from 'chai';
```

8. Write 2 test cases for each method of **UnitConverter** class.

You may start with the following inputs and expected results. Then use search for “google unit converter” to compute the expected results for more input values.

Method	**Input**	**Expected Result**
kgToOunce	1	35.274
kgToPound	2	4.4092
meterToInch	1	39.3701
meterToFoot	2	6.5617

9. Run the unit tests as you develop them using WebStorm:



Also run the unit tests from the command line using: **npm test**

But first, you should have the following in package.json file.

```
"scripts": {
  "test": "mocha **/*.spec.js"
},
```

After you complete the lab, push your work to your GitHub repository.