

# CMPS 350 Web Development Fundamentals

## Lab 6 –Object Oriented Programming and Unit Testing with JavaScript

---

### Objective

The purpose of this laboratory exercise is to strengthen your skills in Object Oriented Programming in JavaScript. You will practice working with the following features:

- **Object literals**, which consist of a series of name-value pairs and associated functions enclosed in curly braces.
- **Classes**, which involve creating classes and using them to create objects.
- **Inheritance**, which is the ability of a class to inherit properties and methods from a parent class.
- **Modules**, which allow for the exporting and importing of code between files.
- **Unit Testing**, which involves writing code to test the functionality of your program.

This Lab is divided into two parts:

- **PART A:** Banking App using object literals
- **PART B:** Banking App using classes
- **PART C:** Unit Testing

First create a subfolder in your repository named `lab6-js-oop` then open it in VS Code.

### PART A – Banking App using object literals

In this exercise, you will build a simple banking app to manage accounts including performing various operations, such as deposit/withdraw an account, adding, deleting, and retrieving accounts.

- 1) Create `bank.js` file. Declare an `accounts` array to store the following accounts:

accountNo	balance	type
123	500	Saving
234	4000	Current
345	35000	Current
456	60000	Saving

- 2) Implement and export the following functions in `bank.js`:

Functions	Functionality
<code>deposit(accountNo, amount)</code>	adds the amount to the account balance.
<code>withdraw(accountNo, amount)</code>	subtracts the amount from the account balance.

<code>add(account)</code>	Add either Saving or Current account to accounts array.
<code>getAccount(accountNo)</code>	Return an account by account No
<code>deleteAccount(accountNo)</code>	Delete an account by account No
<code>avgBalance()</code>	Get the average balance for all accounts
<code>sumBalance()</code>	Get the sum balance for all accounts
<code>distributeBenefit(benefitPercentage)</code>	Loops through savings accounts and updates the balance to: <code>balance += (balance * benefitPercentage).</code>
<code>deductFee(monthlyFee).</code>	Loops through current accounts and updates the balance to: <code>balance -= monthlyFee.</code>
<code>toJson()</code>	Return accounts as a JSON string

3) Create `app.js` to test `bank.js` functions.

- Create couple of new saving/current accounts.
- Do couple of deposit and withdraw operations.
- Charge a monthly fee of 10QR to current accounts.
- Display the total balance of all accounts after charging the monthly fee.
- Distribute 6% benefit to all the **Saving** accounts.
- Display the total balance of all accounts after distributing the benefits.
- Display the list of accounts in json format

### **PART A – Banking App using classes**

In this exercise, you will build a simple banking app. The app will consist of several classes, each with its own set of properties and methods to represent different types of bank accounts, such as saving accounts and current accounts. You will also create a Bank class that will manage all the accounts and perform various operations, such as adding, deleting, and retrieving accounts as shown in Figure 1.

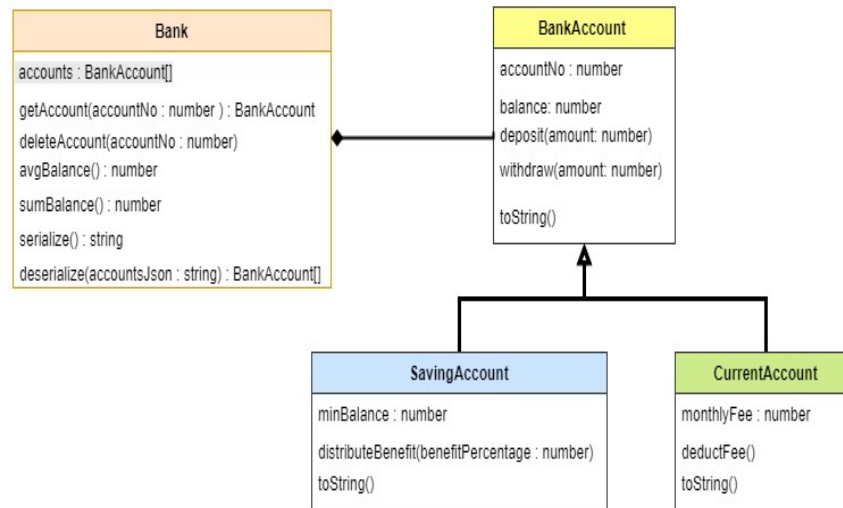


Figure 1. Banking App Class Diagram

- 4) Create a class called **BankAccount** with two private properties: **accountNo** and **balance**. The account number should be **randomly generated**, and the balance should be initialized through the class constructor. The class should also have the following methods:

- get properties for both accountNo and balance.
- deposit(amount): adds the amount to the balance.
- withdraw(amount): 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 so that it can be reused outside the BankAccount.js file.

- 5) Create **app.js** program to test the **BankAccount** class. Declare an array called accounts and initialize it with the following accounts:

accountNo	balance
123	1000
234	4000
345	3500

Then display the content of the **accounts** array.

- 6) Create **SavingAccount** class that extends **BankAccount** with extra static properties: **minBalance** (set the value to 500) and **benefitPercentage** (set the value to 0.05). Also, add an extra method **distributeBenefit()**. This method computes the monthly benefit using the **balance += (balance \* SavingAccount.benefitPercentage)**. Additionally, 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.

- 7) Create **CurrentAccount** class that extends **BankAccount** with an extra static property: **monthlyFee** (set the value to 10) and an extra method **deductFee()**. This method subtracts the **monthlyFee** from the account balance. 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.

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

Method	Functionality
<b>add(account)</b>	Add account (either Saving or Current) to accounts array.
<b>getAccount(accountNo)</b>	Return an account by account No
<b>deleteAccount(accountNo)</b>	Delete an account by account No
<b>avgBalance()</b>	Get the average balance for all accounts
<b>sumBalance()</b>	Get the sum balance for all accounts
<b>toJson()</b>	Return accounts as a JSON string
<b>fromJson(accountsJson)</b>	Takes JSON string representing accounts and returns an array of accounts.

- 9) Modify **app.js** program. Declare an instance of **Bank** class then add the following accounts:

accountNo	balance	type
123	500	Saving
234	4000	Current
345	35000	Current
456	60000	Saving

- 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 6% benefit.
- Display the total balance of all accounts after distributing the benefits.

### **Part B – Unit Testing Using Mocha and Chai**

Unit testing is an essential part of modern software development. It involves testing individual units of code to ensure that they function as expected and meet the requirements of the application. Unit tests help to catch bugs and errors early in the development process, reducing the likelihood of issues arising in production.

In this section, we will cover the basics of unit testing with Mocha and Chai. We will explore how to write and run tests, how to use the various test styles provided by Mocha, and how to write assertions using Chai.

1. Sync cmpts350-lab repo to get the Lab files.
2. Copy **Lab6-JS OOP** folder from cmpts350-lab repo to your repository.
3. Open **Lab6- JS OOP \UnitConverter.js** in VS Code. 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. Create 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. Add the following inside package.json file.

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

7. Create a JavaScript file named **UnitConverter.spec.js**
8. Import an instance the *UnitConverter* class to be tested and the *chai expect* package.

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

9. 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.

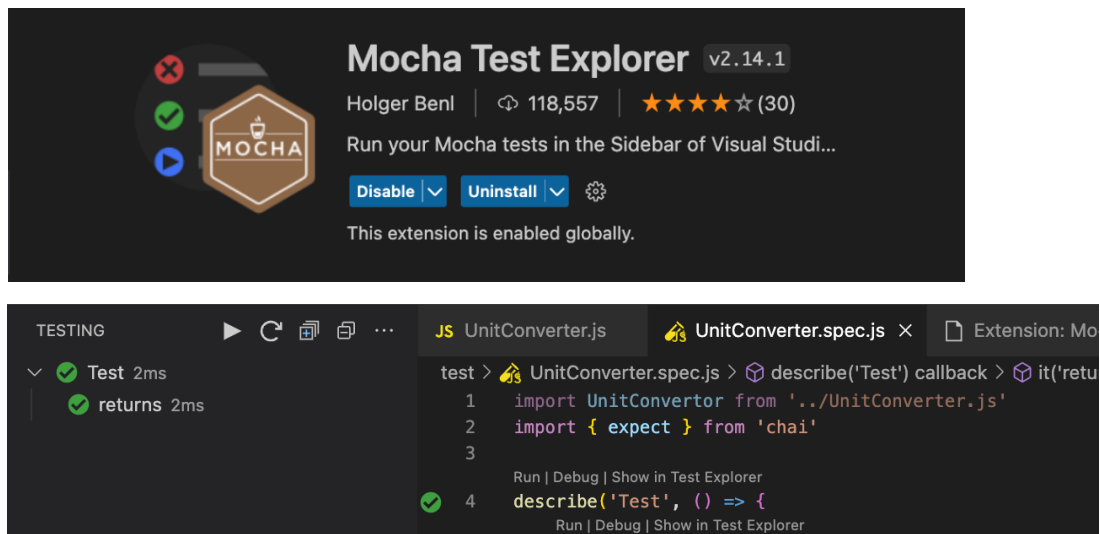
Method	Input	Expected Result
kgToOunce	1	35.274
kgToPound	2	4.4092
meterToInch	1	39.3701
meterToFoot	2	6.5617

### Tips

- Use **expect** to make assertions about the output of the functions.
- Use **describe** and **it** functions to structure your tests.
- Use console.log statements to debug your tests if necessary.

10. Run the unit tests from the command line using: **npm test** –watch to automatically re-run them as you make changes.

11. You can also install the following extension and run the tests using the Test Explorer UI.



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