Lab Practice: Advanced Javascripts Concepts

Part 1: Scope

Scope in JavaScript refers to the context in which variables are accessible and defined. It determines the visibility of variables, affecting how and where they can be used within your code.

1. Global Scope:
    - A variable declared in the global scope is accessible from anywhere in the code, including inside functions. Global variables are created when they are defined outside of any function.
2. Function Scope:
    - Variables declared within a function are only accessible within that function. They cannot be accessed from outside the function.
3. Block Scope:
    - Variables declared with let or const inside a block (i.e., within curly braces {}) are only accessible within that block. This was introduced with ES6 (ECMAScript 2015).

Objective:

Understand how scope works by creating functions that demonstrate global and local variables.

Step 1: Scope

1. Create a new folder for the lab practice.
2. Inside the folder, create a ***script.js*** file.

Step 2:

1. Write a code snippet that demonstrates global and function scopes.

```
let globalVar = 'I am global';

function testScope() {
    let localVar = 'I am local';
    console.log(globalVar); // Should print: I am global
    console.log(localVar);   // Should print: I am local
}
testScope();

console.log(localVar); // This should throw a ReferenceError
```

Step 3: Run the JavaScript File with Node.js

1. Open Terminal/Command Prompt:
    - Navigate to the directory where your script.js file is saved using the cd command. For example:

      cd path/to/your/file

2. Run the JavaScript File:
    - Execute your JavaScript file by typing:

      node script.js

Example of Function Scope

Step 1: Create an HTML File.

1. Create a new file named index.html.
2. Write the following code into index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Function Scope Example</title>
</head>
<body>
   <h1>Function Scope Example</h1>
   <button id="myButton">Click me!</button>

   <script src="script.js"></script> <!-- Link to the JavaScript file -->
</body>
</html>
```

Step 2: Create a JavaScript File

1. In the same folder, create a new file named script.js.
2. Write the following code into script.js:

```
function showMessage() {
    var message = "Hello from inside the function!"; // Function scoped variable
    alert(message); // This will work and show the message
}

document.getElementById('myButton').addEventListener('click', showMessage);

// Trying to access 'message' outside the function will result in an error
console.log(message); // This will throw a ReferenceError: message is not defined
```

Part 2: Closure

A closure allows a nested function to access variables from its parent function even after the parent function has finished executing.

Objective:

Learn how closures work in JavaScript by creating functions that can keep track of information.

Step 2: Create a Simple Closure

1. Create an outerFunction that contains a variable.
2. Define an innerFunction inside it that logs this variable.
3. Return the innerFunction and call it.

```
function myOuterFunction() {
    let secret = "I am a secret message!";

    function myInnerFunction() {
        console.log(secret);
    }

    return myInnerFunction;
}

const myClosure = myOuterFunction();
myClosure(); // Should log: "I am a secret message!"
```

Example of Closures

Step 1: Create an HTML File.

1. Write the following code into index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Counter Example</title>
</head>
<body>
    <h1>Counter Example</h1>
    <p id="counterDisplay">Count: 0</p> <!-- Element to display the count -->
    <button id="incrementButton">Increment Count</button> <!-- Button to increment
    the count -->
    <script src="script.js"></script> <!-- Include external JavaScript file -->
</body>
</html>
```

Step 2: Create a JavaScript File

1. Write the following code into script.js:

```
function createCounter() {
    let count = 0;

    return {
        increment: function() {
            count++;
            // Update the content of the HTML element to display the current count
            document.getElementById("counterDisplay").innerText = "Count: " + count;
        }
    };
}

const counter = createCounter();

// Get reference to the button and set up the event listener
document.getElementById("incrementButton").addEventListener("click", function() {
    counter.increment();
});
```

Part 3: Asynchronous JavaScript

Objective:

Students will learn important ideas about asynchronous JavaScript, such as callbacks, promises, and the async/await style. They will practice how to work with tasks that happen at different times.

I. **Understanding Callbacks**

A callback is a function that is passed as an argument to another function and is executed after some operation has completed.

Step 1:

1. Write an example using a setTimeout function to create a delay before executing a callback function.

```
function fetchData(callback) {
    setTimeout(() => {
        const data = 'Data fetched successfully!';
        callback(data);
    }, 2000); // Simulating a 2-second delay
}

// Using the callback
fetchData((data) => {
    console.log(data); // Should log "Data fetched successfully!" after 2 seconds
});
```

II. **Understanding Promises**

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Step 1: Create a Promise Example

1. Convert the previous callback example into a promise.

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const success = true; // Simulate success or failure
            if (success) {
                resolve('Data fetched successfully!');
            } else {
                reject('Error fetching data');
```

```
        }
      }, 2000);
    });
}

// Using the promise
fetchData()
    .then((data) => {
        console.log(data); // Should log "Data fetched successfully!" after 2 seconds
    })
    .catch((error) => {
        console.error(error);
    });
```

III. **Understanding Async/Await**

Step 1: Create an Async/Await Example

1. Convert the previous promise example to use async/await.

```
async function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const success = true; // Simulate success or failure
            if (success) {
                resolve('Data fetched successfully!');
            } else {
                reject('Error fetching data');
            }
        }, 2000);
    });
}

async function displayData() {
    try {
        const data = await fetchData();
        console.log(data); // Should log "Data fetched successfully!" after 2 seconds
    } catch (error) {
        console.error(error);
    }
}

displayData();
```

Example of Asynchronous JavaScript

Step 1: Create an HTML File.

1. Write the following code into index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Asynchronous JavaScript Example</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
        }
        #output {
            margin-top: 20px;
            font-weight: bold;
        }
    </style>
</head>
<body>
    <h1>Asynchronous JavaScript</h1>
    <button id="fetchDataButton">Fetch Data</button>
    <div id="output"></div> <!-- Element to display fetched data -->

    <!-- Link to the external JavaScript file -->
    <script src="script.js"></script>
</body>
</html>
```

Step 2: Create a JavaScript File

1. Write the following code into script.js:

```
// Example of using a Promise with setTimeout
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const success = true; // Simulate success or failure
            if (success) {
                resolve('Data fetched successfully!');
            } else {
```

```
                reject('Error fetching data');
            }
        }, 2000); // Simulate a 2-second delay
    });
}

// Using async/await to fetch and display data
async function fetchAndDisplayData() {
    try {
        const data = await fetchData();
        document.getElementById('output').innerText = data; // Display the fetched data
    } catch (error) {
        document.getElementById('output').innerText = error; // Display error message
    }
}

// Fetch data from an API when the button is clicked
document.getElementById('fetchDataButton').addEventListener('click',
fetchAndDisplayData);
```

2. To include a real API fetch example, you can update the script.js file as follows:

```
// Using fetch API to get data from a placeholder API
async function fetchAPIData() {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        const data = await response.json();
        document.getElementById('output').innerText = JSON.stringify(data); // Display
the fetched data
    } catch (error) {
        document.getElementById('output').innerText = 'Fetch error: ' + error; // Display
error message
    }
}

// Fetch data from the API when the button is clicked
document.getElementById('fetchDataButton').addEventListener('click',
fetchAPIData);
```

Part 4: Object-Oriented Programming in JavaScript

Objective:

Students will learn and implement the concepts of OOP in JavaScript, creating objects and classes, using inheritance, and applying encapsulation and polymorphism.

I. Introduction to Objects

What is an object?

An object is a collection of related data and functionalities (called properties and methods) bundled together. You can think of it like a real-world object, such as a car or a person, which has characteristics (properties) and actions it can perform (methods).

Example:

Imagine a car. It has properties like make, model, and year, and methods like start() or stop().

Step 1: Create a Simple Object a Simple Object Representing a Car.

1. Write the following code into script.js:

```
const car = {
    make: 'Toyota',
    model: 'Camry',
    year: 2020,
    start: function() {
        console.log(`${this.make} ${this.model} started.`);
    }
};

car.start(); // Should log "Toyota Camry started."
```

II. Constructor Functions and Prototypes

What is a constructor function?

A constructor function is a special type of function used to create multiple objects that share the same properties and methods. It's like a blueprint or template for making objects.

How does it work?

1. Define a function with a capitalized name (by convention) that initializes the properties of an object.
2. Inside the constructor, "***this***" refers to the new object being created.
3. Create new objects using the new keyword with the constructor function.

Step 1: Create a Car Constructor Function

1. Write the following code into script.js:

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}

Car.prototype.start = function() {
    console.log(`${this.make} ${this.model} started.`);
};

// Creating instances of Car
const car1 = new Car('Toyota', 'Camry', 2020);
const car2 = new Car('Honda', 'Accord', 2021);

car1.start(); // Should log "Toyota Camry started."
car2.start(); // Should log "Honda Accord started."
```

III. JavaScript Classes (ES6)

What is a class?

A class is a blueprint for creating objects. It defines properties and methods that all objects created from the class will have. Think of it like a template for building multiple similar objects.

How to create a class?

You use the ***class*** keyword followed by the class name (by convention, starting with a capital letter). Inside the class, you define a special method called ***constructor()***, which initializes the object's properties. You can also add other functions (***methods***) to define what actions the object can perform.

Step 1: Create a Class for Car

1. Write the following code into script.js:

```
class Car {
    constructor(make, model, year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    start() {
```

```
        console.log(`${this.make} ${this.model} started.`);
    }
}

const car1 = new Car('Toyota', 'Camry', 2020);
const car2 = new Car('Honda', 'Accord', 2021);

car1.start(); // Should log "Toyota Camry started."
car2.start(); // Should log "Honda Accord started."
```

IV. Inheritance

What is inheritance?

Inheritance allows you to create a new class that inherits properties and methods from an existing class. This means the new class (called a subclass) can use everything the original class (called a superclass) has, but you can also add more features or customize behaviors.

How does it work in JavaScript?

You use the ***extends*** keyword to make a class inherit from another class, and inside the subclass, you call ***super()*** to invoke the constructor of the parent class.

Step 1: Create a Subclass

1. Write the following code into script.js to create a ElectricCar subclass that inherits from Car.

```
class ElectricCar extends Car {
    constructor(make, model, year, batteryCapacity) {
        super(make, model, year); // Call the parent class constructor
        this.batteryCapacity = batteryCapacity;
    }

    charge() {
        console.log(`${this.make} ${this.model} is charging.`);
    }
}

const tesla = new ElectricCar('Tesla', 'Model S', 2022, '100 kWh');
tesla.start(); // Should log "Tesla Model S started."
tesla.charge(); // Should log "Tesla Model S is charging."
```

V. Encapsulation

What is Encapsulation?

Encapsulation is a way of hiding the internal details (data) of an object so that it can't be changed or accessed directly from outside the object. Instead, you provide controlled access through methods (functions), ensuring the data remains safe and consistent.

Why is it important?

- It protects the internal state of an object from unintended or harmful changes.
- It makes code easier to maintain because the internal workings are hidden, and you control how data is accessed or modified.
- It promotes data security and integrity.

How to implement it?

In JavaScript, with newer syntax (ES6+), you can declare private properties using the # symbol. These properties can only be accessed inside the class. To allow controlled access, you create methods like ***get*** and ***set.***

Step 1: Implement Encapsulation

1. Add the following code into script.js to modify the Car class to use private properties.

```
class Car {
    #make; // Private instance variable
    #model; // Private instance variable
    #year; // Private instance variable

    constructor(make, model, year) {
        this.#make = make;
        this.#model = model;
        this.#year = year;
    }

    start() {
        console.log(`${this.#make} ${this.#model} started.`);
    }

    getDetails() {
        return `${this.#make} ${this.#model}, Year: ${this.#year}`;
    }
}
```

```
const car1 = new Car('Toyota', 'Camry', 2020);
car1.start(); // Should log "Toyota Camry started."
console.log(car1.getDetails()); // Log details
```

VI. Polymorphism

Polymorphism in JavaScript allows us to define methods in a base class and override them in derived classes. This way, the same method can behave differently based on the object that it is being called on.

Step 1: Implement Polymorphism

1. Modify the ElectricCar subclass to override the start method from the Car class.

```
class Car {
  #make; // Private instance variable
  #model; // Private instance variable
  #year; // Private instance variable

  constructor(make, model, year) {
    this.#make = make;
    this.#model = model;
    this.#year = year;
  }

  getMake() {
    return this.#make;
  }

  getModel() {
    return this.#model;
  }

  start() {
    console.log(`${this.#make} ${this.#model} started.`);
  }

  getDetails() {
    return `${this.#make} ${this.#model}, Year: ${this.#year}`;
  }
}

class ElectricCar extends Car {
  #batteryCapacity; // Private instance variable
```

```
  constructor(make, model, year, batteryCapacity) {
    super(make, model, year); // Call the parent class constructor
    this.#batteryCapacity = batteryCapacity;
  }

  // Overriding the start method
  start() {
    console.log(`${this.getMake()} ${this.getModel()} (Electric) started silently.`);
  }

  charge() {
    console.log(`${this.getDetails()} with battery ${this.#batteryCapacity} is charging.`);
  }
}

const tesla = new ElectricCar('Tesla', 'Model S', 2022, '100 kWh');
const toyota = new Car('Toyota', 'Camry', 2020);

tesla.start();  // Outputs: Tesla Model S (Electric) started silently.
toyota.start(); // Outputs: Toyota Camry started.
```

Example of OOP in Javascript:

Step 1: Create an HTML File.

1. Write the following code into index.html to visualize OOP concepts in a practical setting:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Object-Oriented Programming Example</title>
</head>
<body>
<h1>Vehicle Information</h1>
<button id="showCarDetails">Show Car Details</button>
<button id="showElectricCarDetails">Show Electric Car Details</button>

<div id="output"></div>

<!-- Link to the external JavaScript file -->
```

```
<script src="script.js"></script>
</body>
</html>
```

Step 2: Create a JavaScript File

1. Write the following code into script.js:

```
class Car {
  #make;
  #model;
  #year;

  constructor(make, model, year) {
    this.#make = make;
    this.#model = model;
    this.#year = year;
  }

  getMake() {
    return this.#make;
  }

  getModel() {
    return this.#model;
  }

  start() {
    console.log(`${this.#make} ${this.#model} started.`);
  }

  getDetails() {
    return `${this.#make} ${this.#model}, Year: ${this.#year}`;
  }
}

class ElectricCar extends Car {
  #batteryCapacity;

  constructor(make, model, year, batteryCapacity) {
    super(make, model, year);
    this.#batteryCapacity = batteryCapacity;
  }
```

```
  start() {
    console.log(`${this.getMake()} ${this.getModel()} (Electric) started silently.`);
  }

  charge() {
    console.log(`${this.getDetails()} with battery ${this.#batteryCapacity} is charging.`);
  }
}

// Instantiate objects
const toyota = new Car('Toyota', 'Camry', 2020);
const tesla = new ElectricCar('Tesla', 'Model S', 2022, '100 kWh');

// Add event listeners
document.getElementById('showCarDetails').addEventListener('click', () => {
  document.getElementById('output').innerText = toyota.getDetails();
});

document.getElementById('showElectricCarDetails').addEventListener('click', () => {
  document.getElementById('output').innerText = tesla.getDetails();
});
```