ADVANCED JAVASCRIPTS CONCEPTS

OBJECTIVES

By the end of this chapter, students should be able to:

- explain JavaScript scope, distinguish global and local scopes, and understand closures and their use in modular code.
- implement asynchronous programming with callbacks, promises, and async/await for handling API calls and events.
- grasp key object-oriented principles in JavaScript, such as prototypes and inheritance, as well as functional programming concepts like first-class functions.
- use modern JavaScript features and design patterns to write clean, efficient, maintainable code while learning performance optimization techniques.

BASIC VS. ADVANCED CONCEPTS

- **Basic Concepts:** Includes understanding variables, data types, operators, control structures (if, loops), and basic functions.

- **Advanced Concepts:** Involves mastery of topics such as scope and closures, asynchronous programming, OOP principles, functional programming techniques, and performance optimization.

SCOPE

- Scope refers to the availability and accessibility of variables and functions in different parts of the code. It determines what variables can be used where.

- Types of Scope:
  - Global Scope
  - Local Scope
  - Block Scope

TYPES OF SCOPE

- **Global Scope:** Variables declared outside any function or block. These are accessible anywhere within the code.

Example:

```
var globalVar = "I am global";
function showGlobal() {
    console.log(globalVar); // Accessible here
}
```

TYPES OF SCOPE

- **Local Scope:** Variables declared within a function. These can only be accessed inside that function.

Example:

```
function myFunction() {
    var localVar = "I am local";
    console.log(localVar); // Accessible here
}
console.log(localVar); // Error: localVar is not defined
```

TYPES OF SCOPE

- **Block Scope:** Introduced with let and const. Variables declared within a block (i.e., { }) are not accessible outside that block.

Example:

```
{
    let blockVar = "I am block scoped";
}
console.log(blockVar); // Error: blockVar is not defined
```

- Variables declared with the ***var*** keyword can NOT have block scope.
- Variables declared inside a { } block can be accessed from outside the block.

CLOSURES

- A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope.

- **How Closures Work:**
  - When a function is defined inside another function, it creates a closure that allows the inner function to access variables from the outer function's scope, even after the outer function has finished executing.

CLOSURES (CONT')

- **Benefits of Closures:**
    - **Encapsulation:** Closures allow for data hiding, creating private variables.
    - **State Retention:** Useful in situations where you want to maintain state in a callback function or asynchronous operations.

CLOSURES (CONT')

- **Example of a Closure:**

```
function outerFunction() {
    let outerVar = "I am from outer function";
    function innerFunction() {
        console.log(outerVar); // Accesses outerVar
    }
    return innerFunction; // Returns the inner function
}

const myClosure = outerFunction(); // Executes outerFunction
myClosure();  // Output: "I am from outer function"
```

USE CASES FOR CLOSURES

- **Data Privacy:** Protect sensitive data from being accessed directly.

```
function createCounter() {
  let count = 0; // Private variable
  return {
    increment: function() {
      count++;
      return count;
    },
    decrement: function() {
      count--;
      return count;
    }
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
```

ASYNCHRONOUS JAVASCRIPT

- Asynchronous JavaScript refers to the ability to execute operations in a non-blocking manner, allowing other code to run while waiting for external processes (like network requests) to complete.

- It enhances user experience by preventing web applications from freezing or becoming unresponsive while performing long-running operations.

THE EVENT LOOP

- The event loop is a key mechanism that allows JavaScript to perform non-blocking I/O despite being single-threaded. It continuously checks for messages in the queue and executes code accordingly.
- **Call Stack:** Executes JavaScript code line by line. When a function is called, it is added to the stack; once the function completes, it is popped off the stack.
- **Callback Queue:** Contains messages/events that are waiting to be processed after the current call stack is empty.

CALLBACK FUNCTIONS

- Callback functions are functions passed as arguments to other functions and are executed after certain conditions are met or events occur.

Example:

```
console.log("Start");

setTimeout(function() {
    console.log("This message is delayed");
}, 2000); // This callback runs after 2 seconds
console.log("End");
```

CALLBACK FUNCTIONS (CONT')

Output:

```
Start
End
This message is delayed
```

PROMISES

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

- States of a Promise:
    - ***Pending:*** Initial state, neither fulfilled nor rejected.
    - ***Fulfilled:*** The operation completed successfully.
    - ***Rejected:*** The operation failed.

PROMISES (CONT')

- **Creating a Promise:**

```
const myPromise = new Promise((resolve, reject) => {
    const condition = true; // Change based on desired outcome
    if (condition) {
        resolve("Operation Successful!");
    } else {
        reject("Operation Failed!");
    }
});
```

PROMISES (CONT')

- **Using Promises:**
  - Then and Catch:

```
myPromise
    .then(result => {
        console.log(result); // Output if resolved
    })
    .catch(error => {
        console.log(error); // Output if rejected
    });
```

PROMISES (CONT')

- **Chaining Promises:** Promises can be chained for better readability and handling multiple asynchronous operations.

```
fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("Error:", error));
```

ASYNC/AWAIT

- “async/await” is syntactic sugar built on top of Promises, allowing us to write asynchronous code that looks synchronous.

- **Using async:** Functions defined with the async keyword always return a Promise.

- **Using await:** The await keyword pauses the execution of an async function, waiting for the Promise to resolve or reject.

ASYNC/AWAIT (CONT')

Example:

```
async function fetchData() {
    try {
        const response = await fetch("https://api.example.com/data");
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchData();
```

OBJECT-ORIENTED PROGRAMMING (OOP) IN JAVASCRIPT

- Object-Oriented Programming is a programming paradigm that uses "objects" to represent data and methods to manipulate that data. It helps structure code in a way that makes it more modular, reusable, and easier to maintain.

- **Key Concepts:** Objects, Classes, Inheritance, Encapsulation, and Polymorphism.

JAVASCRIPT OBJECTS

- An object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be of any data type, including functions (methods).

Example:

```
const person = {
    name: "John",
    age: 30,
    greet: function() {
        console.log("Hello, my name is " + this.name);
    }
};
person.greet(); // Output: Hello, my name is John
```

PROTOTYPAL INHERITANCE

- In JavaScript, inheritance is achieved through prototypes. Every JavaScript object has a prototype, which is another object from which it can inherit properties and methods.

- **How Prototypal Inheritance Works:**
    - An object can act as a prototype for another object, creating a prototype chain.
    - When trying to access a property, JavaScript looks at the object itself first, then at its prototype, and continues up the chain until it finds the property or reaches the end (null).

PROTOTYPAL INHERITANCE (CONT')

Example:

```
const animal = {
    speak: function() {
        console.log("Animal speaks");
    }
};

const dog = Object.create(animal); // dog inherits properties from animal
dog.speak(); // Output: Animal speaks

person.greet(); // Output: Hello, my name is John
```

CLASSES IN ES6

- Introduced in ECMAScript 2015 (ES6), classes provide a clearer syntax for creating objects and handling inheritance.

- Class Syntax: A class is defined using the **class** keyword.

CLASSES IN ES6

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}


const dog = new Dog("Rex");
dog.speak(); // Output: Rex barks.
```

INHERITANCE AND THE PROTOTYPE CHAIN WITH CLASSES

- **Extending Classes:** Use the extends keyword to create a subclass, inheriting properties and methods from a parent class.

- **Using super():** Call the parent class's constructor and methods using the super() function.

CLASSES IN ES6

```
class Cat extends Animal {
    constructor(name) {
        super(name); // Call parent constructor
    }

    speak() {
        console.log(`${this.name} meows.`);
    }
}

const cat = new Cat("Whiskers");
cat.speak(); // Output: Whiskers meows.
```

ENCAPSULATION

- Encapsulation is the concept of bundling data (properties) and methods that operate on the data within one unit (class) and restricting access to some components.

- Data Privacy: JavaScript offers a way to create private properties using closures or the new class fields syntax with the # prefix in ES2022 (Stage 3).

ENCAPSULATION (CONT')

```
function Person(name) {
    let age = 0; // Private variable
    this.name = name;

    this.getAge = function() {
        return age;
    };

    this.setAge = function(newAge) {
        age = newAge;
    };
}

const john = new Person("John");
john.setAge(25);
console.log(john.getAge()); // Output: 25
```

POLYMORPHISM

- Polymorphism allows methods to do different things based on the object (class) calling them, typically achieved via method overriding.
- Example: The speak() method in different subclasses (Dog, Cat) provides different outputs based on the calling object.

FUNCTIONAL PROGRAMMING CONCEPTS

- Functional programming (FP) is a programming paradigm focused on writing functions that produce outputs based solely on their inputs, avoiding shared state and mutable data.

- **Key Characteristics**:
  - Higher-order functions
  - First-class functions
  - Pure functions
  - Immutability
  - Function composition

FIRST-CLASS FUNCTIONS

- In JavaScript, functions are first-class citizens. This means functions can be assigned to variables, passed as arguments, and returned from other functions.

```
function greet(name) {
    return `Hello, ${name}!`;
}

const greetUser = greet; // Assigning function to a variable
console.log(greetUser("Alice")); // Output: Hello, Alice!
```

HIGHER-ORDER FUNCTIONS

- A higher-order function is a function that takes one or more functions as arguments or returns a function as its result.
- *Function as Argument:*

```
function repeatAction(action, times) {
    for (let i = 0; i < times; i++) {
        action(); // Execute the passed function
    }
}

repeatAction(() => console.log("Hello!"), 3);
```

HIGHER-ORDER FUNCTIONS (CONT')

- *Function as Return Value:*

```
function multiplier(factor) {
    return function(x) {
        return x * factor; // Returning a new function
    };
}

const double = multiplier(2);
console.log(double(5)); // Output: 10
```

PURE FUNCTIONS

- *A pure function is a function that:*
    - *Returns the same output for the same input (deterministic).*
    - *Has no side effects (does not modify any external state).*

```
function add(a, b) {
    return a + b; // Pure function
}

console.log(add(2, 3)); // Output: 5
```

IMMUTABILITY

- *Immutability means that once a data structure is created, it cannot be changed. Instead, any change results in the creation of a new data structure.*

```
const originalArray = [1, 2, 3];
const newArray = [...originalArray, 4]; // Creates a new array
console.log(originalArray); // Output: [1, 2, 3]
console.log(newArray); // Output: [1, 2, 3, 4]
```

FUNCTION COMPOSITION

- *Function composition is the process of combining two or more functions to produce a new function.*

```
const addOne = x => x + 1;
const double = x => x * 2;

const addOneAndDouble = x => double(addOne(x));
console.log(addOneAndDouble(3)); // Output: 8 (3 + 1 = 4, then 4 * 2 = 8)
```

THE ‘THIS’ KEYWORD

- In JavaScript, “***this***” refers to the context in which a function is executed. It allows access to the object that is currently calling the function, providing a way to refer to that object.
- **Global Context**
    - In the global execution context, “***this***” refers to the global object (Window in browsers).

```
console.log(this); // In a browser, this refers to the Window object
```

THE ‘THIS’ KEYWORD (CONT’)

- **Function Context**
  - When a function is called as a regular function (not as a method), “***this***” refers to the global object (or undefined in strict mode).

```
function showThis() {
    console.log(this);
}
showThis(); // In non-strict mode, logs the global object; in strict mode, logs undefined
```

THE ‘THIS’ KEYWORD (CONT’)

- **Method Context**
  - When a function is called as a method of an object, “***this***” refers to the object invoking the method.

```
const person = {
    name: "Alice",
    greet: function() {
        console.log("Hello, " + this.name);
    }
};
person.greet(); // Output: Hello, Alice
```

THE ‘THIS’ KEYWORD (CONT’)

- **Arrow Functions**
  - Unlike traditional functions, arrow functions do not have their own “***this***” context; instead, they lexically bind “***this***” to the surrounding code where the arrow function is defined.

```
const person = {
  name: "Bob",
  greet: function() {
    const innerFunction = () => {
      console.log("Hello, " + this.name);
    };
    innerFunction();
  }
};
person.greet(); // Output: Hello, Bob
```

THE ‘THIS’ KEYWORD (CONT’)

- **Binding “*this*”**
  - The **bind()** method creates a new function that, when called, has its “***this***” keyword set to a specified value.

```
function show() {
   console.log(this.name);
}

const user = { name: "Charlie" };
const boundShow = show.bind(user);
boundShow(); // Output: Charlie
```

THE ‘THIS’ KEYWORD (CONT’)

- **Call and Apply:** Both allow you to set this when calling a function.
  - **Call:** Call a function with a given this value and arguments.

```
function greet() {
    console.log("Hello, " + this.name);
}

const user = { name: "Diana" };
greet.call(user); // Output: Hello, Diana
```

THE ‘THIS’ KEYWORD (CONT’)

- **Call and Apply:** Both allow you to set this when calling a function.
  - **Apply:** Similar to **call()** but accepts an array of arguments.

```
function sum(a, b) {
    return a + b;
}

const result = sum.apply(null, [5, 10]); // First argument is this (null in this case)
console.log(result); // Output: 15
```

ERROR HANDLING

- Error handling in JavaScript refers to the process of responding to and managing errors that may occur during program execution.

- Proper error handling ensures that applications can gracefully handle unexpected situations without crashing.

TYPES OF ERRORS

- **Syntax Errors:** Occur when the code is not written correctly according to the syntax rules of JavaScript. These errors are caught at compile-time.
- ***Example:***

```
const name = "Alice   // Missing closing quote
```

- **Output:** SyntaxError: Unexpected token

TYPES OF ERRORS (CONT')

- **Runtime Errors:** Happen during program execution, often due to invalid operations or references that cannot be resolved.
- **Example:**

```
const obj = null;
console.log(obj.name); // Trying to access a property of a null object
```

- **Output:** TypeError: Cannot read property 'name' of null

TYPES OF ERRORS (CONT’)

- **Logical Errors:** These errors occur when the program runs without crashing, but the output is incorrect due to flaws in the logic.

```
function add(a, b) {
    return a - b; // Intended to add, but uses subtraction
}
console.log(add(2, 3)); // Output: -1
```

- **Output:** TypeError: Cannot read property 'name' of null

TRY-CATCH STATEMENT

- The try...catch statement allows you to catch and handle errors gracefully during runtime.
- *Syntax:*

```
try {
   // Code that may throw an error
} catch (error) {
   // Code to handle the error
console.error(error.message); // Log or handle the error appropriately
} finally {
   // Optional: code that runs after try or catch, regardless of the outcome
}
```

TRY-CATCH STATEMENT (CONT')

- Example:

```
try {
    const result = riskyFunction(); // A function that may throw an error
console.log(result);
} catch (error) {
    console.error("An error occurred:", error.message); // Handle the error
} finally {
    console.log("Execution complete!"); // Always runs
}
```

THROWING ERRORS

- You can create your own custom errors using the “***throw***” statement.
- *Syntax:*

```
throw new Error("Custom error message");
```

THROWING ERRORS (CONT’)

- *Example:*

```
function checkAge(age) {
    if (age < 18) {
        throw new Error("You must be at least 18 years old.");
    }
    return "Access granted!";
}

try {
    console.log(checkAge(16));
} catch (error) {
    console.error(error.message); // Output: You must be at least 18 years old.
}
```

ASYNCHRONOUS ERROR HANDLING

- **Promises:** *Handle errors in Promises using the .catch() method.*

```
fetch("invalid-url")
    .then(response => response.json())
    .catch(error => {
        console.error("Fetch error:", error.message);
    });
```

ASYNCHRONOUS ERROR HANDLING (CONT')

- **Async/Await:** Use try-catch blocks with async functions to handle errors.

```
async function fetchData() {
    try {
        const response = await fetch("invalid-url");
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Error:", error.message);
    }
}
```

BEST PRACTICES FOR ERROR HANDLING

- **Use Specific Error Types:** Create specific error classes for different kinds of errors in your application.
- **Log Errors:** Use logging to record errors for debugging purposes, but avoid exposing sensitive information in error messages.
- **Graceful Degradation:** Provide fallback content or alternative functionality to enhance user experience when an error occurs.
- **User-Friendly Messages:** Show meaningful and readable error messages to the user instead of technical jargon.

MODERN JAVASCRIPT FEATURES

- Modern JavaScript (ES6 and later) introduces a range of new features that improve code readability, maintainability, and functionality.

- These features make it easier to write complex applications with cleaner syntax and enhanced capabilities.

TEMPLATE LITERALS

- Template literals provide a way to embed expressions within string literals using backticks (`). They allow for multi-line strings and string interpolation.

- **Features:**
  - String Interpolation: Easily include variables and expressions within strings.

  - Multi-line Strings: Maintain formatting across multiple lines without special characters.

TEMPLATE LITERALS (CONT’)

- *Example:*

```
const name = "Alice";
const age = 25;
const greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting);
// Output: Hello, my name is Alice and I am 25 years old.
```

DESTRUCTURING ASSIGNMENT

- Destructuring allows you to unpack values from arrays or properties from objects into distinct variables.

- ***Array Destructuring:***

```
const colors = ["red", "green", "blue"];
const [firstColor, secondColor] = colors;
console.log(firstColor); // Output: red
```

DESTRUCTURING ASSIGNMENT (CONT')

- ***Object Destructuring:***

```
const person = { name: "Bob", age: 30 };
const { name, age } = person;
console.log(name); // Output: Bob
```

- ***Default Values:*** You can assign default values when destructuring.

```
const { name, occupation = "unknown" } = { name: "John" };
console.log(occupation); // Output: unknown
```

SPREAD AND REST OPERATORS

- **Spread Operator (...):** Expands elements of an iterable (like an array) into individual elements.

- ***Example:***

```
const numbers = [1, 2, 3];
const moreNumbers = [...numbers, 4, 5];
console.log(moreNumbers); // Output: [1, 2, 3, 4, 5]
```

SPREAD AND REST OPERATORS (CONT')

- **Rest Operator (...):** Collects multiple elements and condenses them into a single array.

- ***Example:***

```
function sum(...args) {
    return args.reduce((acc, val) => acc + val, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

MODULES: IMPORT AND EXPORT SYNTAX

- Modern JavaScript supports modular programming, allowing developers to write reusable code by exporting and importing modules.

- ***Exporting:***

```
// module.js
export const pi = 3.14;
export function add(x, y) {
    return x + y;
}
```

MODULES: IMPORT AND EXPORT SYNTAX (CONT’)

- ***Importing:***

```
// main.js
import { pi, add } from './module.js';
console.log(pi); // Output: 3.14
console.log(add(2, 3)); // Output: 5
```

- ***Default Exports:*** A module can export a single value as the default.

```
// module.js
const defaultExport = () => {};
export default defaultExport;

// main.js
import myFunction from './module.js';
```

ARROW FUNCTIONS

- Arrow functions provide a concise syntax for writing function expressions. They do not have their this, arguments, or super, making them ideal for certain contexts, such as callbacks.

- ***Syntax:***

```
const add = (a, b) => a + b;
const square = x => x * x; // Parentheses can be omitted for single parameter
```

ARROW FUNCTIONS (CONT')

- ***Example:***

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6]
```