SLIIT

COMPUTING | BUSINESS | ENGINEERING

SE3040 – Application Framework

Worksheet 02 - JavaScript

01. Describe the features of JavaScript? high-level programming language primarily known for adding interactivity to web pages

Interpreted Language - JavaScript is executed line-by-line by an interpreter.

Client-Side Execution - Runs in the user's browser, enabling dynamic web content.

Event-Driven Programming - Responds to user actions (clicks, key presses) via event listeners.

Object-Oriented Programming (OOP) - Supports OOP with prototypes and, since ES6, classes.

02. What is NIO model? JavaScript NIO (Non-blocking I/O)

event-driven architecture used in JavaScript environments (like Node.js and browsers) to handle I/O operations efficiently without blocking the main execution thread.

03. What does it mean is dynamically typed?

if the type of a variable (e.g., number, string, object) is determined at runtime rather than at compile time, and you don't need to explicitly declare the type of a variable before using it.

```
let x = 42;       // x is a number
x = "Hello";      // Now x is a string
x = true;         // Now x is a boolean
```

04. Compare dynamically typed vs statically typed?

Aspect	Dynamically Typed (e.g., JavaScript)	Statically Typed (e.g., Java, C++)
Type Declaration	Not required. Variables can hold any type.	Required. You must declare the type (e.g., `int`, `String`).
Type Checking	Done at **runtime** (when the code executes).	Done at **compile time** (before the code runs).
Flexibility	High. Variable types can change during execution.	Low. Variable types are fixed once declared.
Example Code	`javascript let x = 5; x = "Hello"; // Valid`	`java int x = 5; x = "Hello"; // Error`
Error Detection	Errors (e.g., type mismatches) appear at runtime.	Errors caught during compilation, before runtime.
Performance	Slightly slower due to runtime type checking.	Faster, as types are resolved at compile time.
Ease of Use	Easier for quick scripting and prototyping.	More verbose, better for large, structured projects.
Debugging	Harder, as type errors may only show up when code runs.	Easier, as type errors are caught early.
Use Cases	Web development, scripting, rapid prototyping	System programming, large-scale apps (e.g.,

05. Describe how multi paradigms of programming work in JavaScript?

supports multiple programming paradigms—different approaches or styles of structuring and writing code. This flexibility allows developers to choose the paradigm (or mix of paradigms) that best suits their problem or preference.

06. What is an Eventing System in JavaScript?

An eventing system is the way JavaScript manages and responds to events—things that happen in a program, like a user clicking a button, a timer finishing, or data arriving from a server.

constantly checking (polling) for changes, JavaScript waits for events to occur and reacts by running specific code (callbacks) when they do. This makes it efficient and ideal for interactive applications, like web pages or Node.js servers.

07. What does it mean by “function as a Class” in JavaScript?

function as a constructor to create objects with shared properties and methods, mimicking the behavior of a class in traditional object-oriented programming.

Before ES6 introduced the class keyword, JavaScript used constructor functions and prototypes to achieve this.

```
// Constructor function (acting as a class)
function Person(name) {
    this.name = name; // Instance property
}

// Add a method to the prototype
Person.prototype.sayHello = function() {
    console.log(`Hello, I'm ${this.name}`);
};

// Create instances
const person1 = new Person("Alice");
const person2 = new Person("Bob");

person1.sayHello(); // "Hello, I'm Alice"
person2.sayHello(); // "Hello, I'm Bob"
```

08. What is object literal?

way to create an object by defining its properties and methods directly within curly braces {} using key-value pairs.

Keys are property names, and values can be anything: numbers, strings, functions, arrays, or even other objects.

```
const person = {
    name: "Alice",          // Property
    age: 25,                // Property
    sayHello: function() {  // Method
        console.log(`Hi, I'm ${this.name}`);
    }
};

console.log(person.name);     // "Alice"
person.sayHello();            // "Hi, I'm Alice"
```

09. What does it **“This”** means in JavaScript?

this represents the object that "owns" the current code at runtime.

Global Context - outside any function, this refers to the global object.In browsers, it’s window;

Constructor Context - In a constructor function or class, this refers to the newly created instance.

method - this is the object; in a standalone function, it’s the global object or undefined (strict mode).

10. Closure in JavaScript

a. What is Closure in JavaScript?

A closure is a function that retains access to variables from its parent (outer) scope, even after the parent function has closed.

```
<!--
    A closure is a function having access to the parent scope,
    even after the parent function has closed.
-->

<p id="demo"></p>

<script>

//a closure gives you access to an outer function's scope from an inner function.

function greeting() {
    let message = 'Hi';
    function sayHi() {
        console.log(message);
    }
    return sayHi;
}
let hi = greeting();
hi(); // still can access the message variable!
```

Normally, when a function finishes, its variables disappear. Closures break that rule by 'remembering' the outer scope.

Closures let you create private variables that can't be accessed directly from outside, mimicking private fields in object-oriented programming.

Why It's Useful: Protects data from being accidentally modified or exposed.

```
function createWallet() {
  let balance = 0; // Hidden variable

  return {
    deposit: (amount) => balance += amount,
    withdraw: (amount) => balance -= amount,
    checkBalance: () => balance,
  };
}

const myWallet = createWallet();
myWallet.deposit(100);
console.log(myWallet.checkBalance()); // 100
myWallet.withdraw(30);
console.log(myWallet.checkBalance()); // 70

// CANNOT directly access `balance`! It's protected.
```

What are real world use cases of Closures?

Problem: You want to store data securely so other code can’t modify it.
Solution: Use a closure to create "private" variables.

11. What is Callback in JavaScript?

A callback is just a function that is passed into another function as an argument, to be executed later when a task finishes.

Example:

javascript Copy

```
function greet(name, callback) {
  console.log("Hello, " + name);
  callback(); // Runs the callback function
}

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Alice", sayGoodbye); // Passing `sayGoodbye` as a callback
```

Output:

Copy

```
Hello, Alice
Goodbye!
```

Callbacks are used in JavaScript for two main reasons:

1. To Handle Asynchronous Operations -

Without Callbacks JavaScript waited for these tasks to finish, the app would freeze. because JavaScript is single-threaded,
With Callbacks pass a function (callback) that runs after the slow task finishes.
Meanwhile, JavaScript keeps running other code.

2. Controlling Execution Order

Sometimes, you need tasks to run in sequence (one after another).
Without Callbacks → Wrong Order

12. Understanding “Callback Hell” in JavaScript

- Callback Hell refers to a situation where multiple asynchronous functions are nested within each other, leading to deeply indented and hard-to-read code.
- This happens when we pass callbacks into callbacks, making the code difficult to maintain and debug.
- Often require executing one task after another. If we use callbacks for each operation, the code can become deeply nested.

Issues:
Code is hard to read (too many nested })).
Adding/removing steps is error-prone.
Debugging is difficult

```
function step1(callback) {
    setTimeout(function () {
        console.log("Step 1: Get bread 🍞");
        callback();
    }, 1000);
}

function step2(callback) {
    setTimeout(function () {
        console.log("Step 2: Add fillings 🥬🍅🧀");
        callback();
    }, 1000);
}

function step3(callback) {
    setTimeout(function () {
        console.log("Step 3: Serve the sandwich 🥪");
        callback();
    }, 1000);
}

// Callback Hell (Nested functions)
step1(function () {
    step2(function () {
        step3(function () {
            console.log("Sandwich is ready! 🎉");
        });
    });
});
```

13. Understanding Promises in JavaScript

- A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation.
- It was introduced to solve the callback hell problem.
- Asynchronous operations like API calls, file reading, and database queries take time. Instead of blocking the execution, JavaScript uses Promises to handle these operations asynchronously.

Basic Syntax of Promises

Marks the promise as successful.
Passes data to .then().

Marks the promise as failed.
Passes an error to .catch().

```
const myPromise = new Promise(function (resolve, reject) {
    let success = true;

    setTimeout(function () {
        if (success) {
            resolve("Operation successful! ✅"); // Resolves if success is true
        } else {
            reject("Operation failed! ❌"); // Rejects if success is false
        }
    }, 2000);
});

// Handling the Promise
myPromise
    .then(function (message) {
        console.log(message); // Runs if resolved
    })
    .catch(function (error) {
        console.log(error); // Runs if rejected
    });
```

3 states:
Pending (waiting for result).
Fulfilled (success → resolve).
Rejected (failure → reject).

14. Why do we need resolve and reject?

---

resolve(value) → Marks the promise as successful and passes a result.

---

reject(error) → Marks the promise as failed and passes an error.

---

15. How can the call back hell problem in part 12 can be solved using promises. Refactor the code.

****