Fundamentals of Software Development for Electronics

Prepared By: Dr. Rony Ibrahim

الجامعة اللبنانية
UNIVERSITE LIBANAISE

Course outline

- Session 1: Importance of Software Engineering in Electronics
- Session 2: Basic Concepts and Terminologies
- Session 3: Software Development Life Cycle (SDLC)
- Session 4: Integrated Development Environments (IDEs) and Basic Git Workflow
- Session 5 and 6: JavaScript & Introduction to TypeScript
- Session 7 and 8: Building Web Applications with ReactJS
- Session 9: CSS Basics and Responsive Design Principles
- Session 10: Basics of Flutter / React Native / Ionic
- Session 11: Project Presentation & Final Review

Advanced JavaScript and TypeScript

Objectives

- Deepen understanding of JavaScript and TypeScript fundamentals.
- Learn advanced JavaScript features like closures, callbacks, and promises.
- Explore TypeScript's advanced type system, generics, and interfaces.
- Implement TypeScript in electronics-based projects.

Exercise 1 – Implementing Closures

Task: Write a JavaScript function that keeps track of how many times it has been called using a closure.

Instructions:

- Create a function that returns another function.
- The returned function should increment and print a counter each time it's called.

Exercise 1 – Solution

```
function counter() {
    let count = 0;
    return function() {
        count++;
        console.log(`Called ${count} times`);
    };
}

const myCounter = counter();
myCounter();  // Called 1 times
myCounter();  // Called 2 times
```

JavaScript and Event-Driven Programming

Event Loop and Asynchronous Programming:

- How the event loop works in JavaScript.
- Handling multiple inputs (e.g., sensors) simultaneously using events.

Example:

- Implementing a non-blocking process to read sensor data every second while maintaining responsiveness.

JavaScript and Event-Driven Programming

JavaScript is single-threaded, meaning it can only execute one piece of code at a time. Despite this limitation, the event loop allows JavaScript to handle multiple operations (like user interactions, network requests, or timers) without blocking the execution of the program.

JavaScript and Event-Driven Programming

Key Components of the Event Loop:

- **Call Stack:** The call stack is a data structure that tracks the function currently being executed. Every time a function is invoked, it’s added to the top of the call stack, and when the function completes, it's removed from the stack (LIFO)
- **Web APIs (or Node.js APIs):** Certain tasks (like DOM events, setTimeout, network requests, etc.) are handled outside of the main JavaScript runtime by the browser (or Node.js) using Web APIs.
- **Callback Queue (or Task Queue):** The callback queue holds all pending callback functions waiting to be executed.
- **Event Loop:** The event loop continuously checks whether the call stack is empty. If it is, it takes the first function from the callback queue and pushes it to the call stack for execution.

Exercise 2 – Async Sensor Data Handling

Task: Simulate a system where multiple sensors are asynchronously sending data to a central server using Promises and ***async/await.***

Instructions:

- Create two functions that simulate sensor readings (using ***setTimeout*** to mimic delay).
- Use ***async/await*** to fetch readings from both sensors and process them.

Exercise 2 – Solution

```
function sensorA() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(15);
        }, 1000);
    });
}

function sensorB() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(25);
        }, 1500);
    });
}

async function fetchSensorData() {
    const dataA = await sensorA();
    const dataB = await sensorB();
    console.log(`${dataA}, ${dataB}`);
}

fetchSensorData();
```

TypeScript Type Annotations and Interfaces

Type Annotations:

- Define specific types for variables, parameters, and return values.
- Example: ***let temperature: number = 25***;

Interfaces:

- Create custom types for complex objects, such as sensor data or device configurations.

```
interface SensorData {
    temperature: number;
    humidity: number;
    waterLevel: number;
    timestamp: string;
}
```

Exercise 3 – TypeScript with Sensor Data

Task: Define an interface for a sensor object that includes *name*, *type*, and a function to return current data. Then, implement a function that accepts this interface as a parameter and outputs sensor data.

Instructions:

- Create a *Sensor* interface.
- Create an array of sensor objects and iterate over them to fetch data.

Exercise 3 – Solution

```
interface Sensor {
    name: string;
    type: string;
    getData(): string;
}

const tempSensor: Sensor = {
    name: "Temperature Sensor",
    type: "Thermometer",
    getData() {
        return "25°C";
    }
};
```

Exercise 3 – Solution

```
const humiditySensor: Sensor = {
    name: "Humidity Sensor",
    type: "Hygrometer",
    getData() {
        return "60%";
    }
};

function displaySensorData(sensor: Sensor) {
    console.log(`${sensor.name}: ${sensor.getData()}`);
}

const sensors: Sensor[] = [tempSensor, humiditySensor];
sensors.forEach(displaySensorData);
```

Advanced TypeScript: Generics and Utility Types

Generics:

- Allow functions and classes to operate on multiple types while remaining type-safe.
- Example: A generic function to handle sensor data of various types.

Utility Types:

- Built-in types like *Partial*, *Pick*, and *Omit* to manipulate interfaces and types.

Exercise 4 – TypeScript Generics

Task: Implement a generic function that takes a list of sensor readings (of any type) and returns the readings with a timestamp.

Instructions:

- Use a generic function to handle an array of readings from different types of sensors.

Exercise 4 – Solution

```
function logReadings<T>(readings: T[]): { reading: T, timestamp: string }[] {
    return readings.map(reading => ({
        reading,
        timestamp: new Date().toISOString()
    }));
}

const temperatureReadings = [22, 23, 25];
const loggedTemp = logReadings(temperatureReadings);
console.log(loggedTemp);
```

Real-World Project Example Using TypeScript

A home automation project where Raspberry Pi collects sensor data (temperature, humidity) and sends it to a server, ensuring type safety with TypeScript.

Project Components:

- TypeScript for defining sensor data types.
- Async/await for asynchronous data collection.
- Event-driven programming for handling multiple devices.

Project Setup (Hands-On)

Task: Set up a TypeScript project to simulate an IoT dashboard for an electronics project. Use classes and interfaces to define devices, sensors, and data. Implement a function to periodically log data from sensors and update the dashboard.

Instructions:

- Create classes for devices and sensors.
- Use interfaces to define data types.
- Use async/await to simulate asynchronous data collection from multiple devices.

Guidance for Solution:

- Each device has multiple sensors, and each sensor has a ***getData*** method.
- Use async/await to simulate periodic data collection and display it in the console or a web UI.

Recap & Key Takeaways

Key Concepts Reviewed:

- Advanced JavaScript concepts like closures, promises, and async/await.
- TypeScript fundamentals and advanced types (interfaces, generics).
- Practical applications of JavaScript and TypeScript in electronics projects.

Project Setup (Hands-On) - Solution Guide

Step 1: Define the Interfaces for Devices and Sensors

Start by defining interfaces for the devices and sensors. This will ensure that we adhere to a consistent structure when creating our devices and sensors.

```
// Define the interface for a Sensor
interface Sensor {
     name: string;
     type: string;
     getData(): Promise<number>;
}

// Define the interface for a Device
interface Device {
     id: string;
     sensors: Sensor[];
     getDeviceData(): Promise<void>;
}
```

Project Setup (Hands-On) - Solution Guide

Step 2: Implement the Sensor Class

```
// Sensor class simulating real-time data fetching
class TemperatureSensor implements Sensor {
    name: string;
    type: string;

    constructor(name: string, type: string) {
        this.name = name;
        this.type = type;
    }

    // Simulate fetching data asynchronously with a Promise
    async getData(): Promise<number> {
        return new Promise((resolve) => {
            setTimeout(() => {
                const reading = Math.random() * 100; // Random temperature reading
                resolve(reading);
            }, 1000); // Simulate 1-second delay for fetching data
        });
    }
}
```

Project Setup (Hands-On) - Solution Guide

```
class HumiditySensor implements Sensor {
    name: string;
    type: string;

    constructor(name: string, type: string) {
        this.name = name;
        this.type = type;
    }

    // Simulate fetching humidity data asynchronously
    async getData(): Promise<number> {
        return new Promise((resolve) => {
            setTimeout(() => {
                const reading = Math.random() * 100; // Random humidity reading
                resolve(reading);
            }, 1500); // Simulate 1.5-second delay for fetching data
        });
    }
}
```

Project Setup (Hands-On) - Solution Guide

Step 3: Implement the Device Class

```
// Device class containing multiple sensors
class SmartHomeDevice implements Device {
    id: string;
    sensors: Sensor[];

    constructor(id: string, sensors: Sensor[]) {
        this.id = id;
        this.sensors = sensors;
    }

    // Fetch data from all sensors asynchronously
    async getDeviceData(): Promise<void> {
        console.log(`Fetching data from device ${this.id}...`);

        for (const sensor of this.sensors) {
            const data = await sensor.getData();
            console.log(`${sensor.name} (${sensor.type}): ${data}`);
        }
    }
}
```

Project Setup (Hands-On) - Solution Guide

Step 4: Implement the Dashboard and Periodic Data Fetching

```
// Function to log data periodically from devices
async function logDataPeriodically(devices: Device[], interval: number) {
    while (true) {
        console.log("\n--- Fetching data from all devices ---\n");

        // Loop through all devices and fetch data
        for (const device of devices) {
            await device.getDeviceData();
        }

        // Wait for the given interval before fetching data again
        await new Promise(resolve => setTimeout(resolve, interval));
    }
}
```

Project Setup (Hands-On) - Solution Guide

Step 5: Simulate the IoT System

```
    // Instantiate sensors
    const tempSensor1 = new TemperatureSensor("Living Room Temp Sensor",
"Temperature");
    const tempSensor2 = new TemperatureSensor("Bedroom Temp Sensor", "Temperature");
    const humiditySensor = new HumiditySensor("Living Room Humidity Sensor",
"Humidity");

    // Create a SmartHomeDevice with multiple sensors
    const smartDevice1 = new SmartHomeDevice("Device-001", [tempSensor1,
humiditySensor]);
    const smartDevice2 = new SmartHomeDevice("Device-002", [tempSensor2]);

    // List of devices
    const devices: Device[] = [smartDevice1, smartDevice2];

    // Start logging data periodically (every 5 seconds)
```

Further Enhancements:

- You could expand this project by storing the fetched sensor data in a database or visualizing it using a front-end web application built with React.

- Adding error handling for potential failures during data fetching (e.g., network issues).