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

Basic Concepts and Terminologies

Objectives

- Understand key programming concepts and terminologies.
- Learn basic syntax and structures in software development.
- Apply these concepts to the development of software for electronics.

Introduction to Key Programming Concepts

- Programming is a way to instruct a computer or device to perform specific tasks.
- These instructions are written in code using different programming languages (Python, JavaScript, etc.).
- Understanding the foundational concepts of programming is essential to build software for electronics.

What is an Algorithm?

- Definition:
  - An algorithm is a sequence of well-defined steps or instructions to solve a particular problem or perform a task.
  - Algorithms form the core of software development.
- Example of an Algorithm:
  - Turning on a light if it's dark:
    1- Check if the room is dark.
    2- If yes, turn on the light.
    3- Otherwise, leave the light off.
- Algorithms control the functionality of electronic devices (e.g., sensors, actuators).

Data Types – Foundation of Variables

Data types define the kind of data a variable can hold.

Primitive Data Types:

- **Integer:** Whole numbers *(e.g., 5, 100)*.
- **Float/Double:** Decimal numbers *(e.g., 3.14, -0.001)*.
- **String:** Sequence of characters *(e.g., "Hello, World!")*.
- **Boolean:** True or false values *(true, false)*.

Composite Data Types:

- **Arrays:** Ordered lists of values.
- **Objects/Dictionaries:** Key-value pairs to store structured data.

Relevance in Electronics:

You'll use data types to store and manipulate sensor readings (e.g., temperature as an integer or float).

Variables – Storing Information

A **variable** is a named location in memory used to store data that can change during the execution of a program.

Declaring Variables:

- In Python: *temperature = 25*
- In JavaScript: *let sensorData = "on"*

Importance:

- Variables are essential to store data from electronic devices like sensors.
- You can store data such as temperature, pressure, and other sensor outputs.

```
temperature = 30  # stores the temperature value

fanState = "OFF"  # stores whether the fan is on or
off
```

Control Structures – Directing Program Flow

Control structures manage the flow of a program based on conditions or iterations.

Conditional Statements (IF-ELSE):

- Used to make decisions in a program.
- Example: If the temperature is greater than 30°C, turn on the fan.

Loops (FOR, WHILE):

- Repeat a block of code multiple times.
- Example: Continuously read sensor data and make decisions based on it.

Control Structures: Conditionals

IF Statements: Used to perform actions based on a condition.

```python
# Python

if temperature > 30:

    print("Turn on the fan")
```

```javascript
# JavaScript

if (temperature > 30) {

    console.log("Turn on the fan");

}
```

Use Case in Electronics: Automatically controlling a fan when the temperature exceeds a threshold.

Control Structures: Loops

While Loops: Repeat a block of code as long as the condition is true.

For Loops: Repeat a block for a defined number of iterations.

```python
# Python

while temperature > 30:

    print("Fan is running")
```

```javascript
# JavaScript

for (let i = 0; i < 5; i++) {

    console.log("Reading sensor data");

}
```

Functions – Reusable Code Blocks

A function is a block of reusable code that performs a specific task.

Benefits:

- Reduces redundancy by allowing the reuse of code.
- Makes code modular and easier to understand.

```
# Python

def turn_on_fan():

    print("Fan is ON")

# Call the function

turn_on_fan()
```

Functions are used to handle repetitive tasks, such as reading sensor data or controlling devices like motors.

Exercise 1 – Writing an Algorithm

Task: Write an algorithm to control a lightbulb using a motion sensor.

Instructions:

- If motion is detected, turn on the light for 10 minutes.
- If no motion is detected, keep the light off.
- Write the algorithm in plain English or pseudocode.

Exercise 1 – Solution

Task: Write an algorithm to control a lightbulb using a motion sensor.

- pseudocode:

```
IF motion_sensor detects motion
    TURN ON the light
    START TIMER for 10 minutes
ELSE
    KEEP the light OFF
```

Exercise 2 – Control Structures with Python

Task: Write Python code that:

- Turns on a light if the temperature is above 25°C and motion is detected.
- Turns off the light otherwise.

Instructions: Use an IF-ELSE control structure.

Exercise 2 – Solution

Task: Write Python code that:

- Turns on a light if the temperature is above 25°C and motion is detected.
- Turns off the light otherwise.

Instructions: Use an IF-ELSE control structure.

```
temperature = int(input("Enter temperature: "))

motion_detected = input("Is motion detected? (yes/no): ")

if temperature > 25 and motion_detected == "yes":

    print("Light is ON")

else:

    print("Light is OFF")
```

Exercise 3 – Loops in JavaScript

Task: Write a JavaScript program that simulates reading sensor data 5 times in a loop.

- Use a for loop to repeat the sensor reading process.
- Output the sensor data (e.g., "Reading temperature: X°C") in each iteration.

Exercise 3 – Solution

Task: Write a JavaScript program that simulates reading sensor data 5 times in a loop.

- Use a for loop to repeat the sensor reading process.
- Output the sensor data (e.g., "Reading temperature: X°C") in each iteration.

```
for (let i = 0; i < 5; i++) {

    Const temp = await getTemp(); // this function will return temp

    console.log("Reading temperature: " + (temp) + "°C");

}
```

Exercise 4 – Debugging Practice

Task: Identify and fix the bug in the following Python code:

```
temperature = input("Enter temperature: ")

if temperature > 30:

    print("Fan is ON")

else:

    print("Fan is OFF")
```

Exercise 4 – Solution

Task: Identify and fix the bug in the following Python code:

```
temperature = input("Enter temperature: ")
temperature = int(input("Enter temperature: "))
if temperature > 30:

    print("Fan is ON")

else:

    print("Fan is OFF")
```

Exercise 5 – TypeScript Data Types and Interfaces

Task: Define an interface **SensorData** for temperature and humidity sensors.

Instructions:

- The interface should include temperature (number), humidity (number), and status (a string representing if the sensor is "active" or "inactive").

- Write a TypeScript function *printSensorStatus(data: SensorData): void* that outputs the sensor status in the console.

Exercise 5 – Solution

Task: Define an interface **SensorData** for temperature and humidity sensors.

```
interface ISensorData {
    temperature: number;
    humidity: number;
    status: "active" | "inactive";
}
function printSensorStatus(data: ISensorData): void {
    console.log(`Sensor is ${data.status}. Temperature: ${data.temperature}°C, Humidity: ${data.humidity}%`);
}
const sensor: ISensorData = { temperature: 28, humidity: 45, status: "active" };
printSensorStatus(sensor);
```

Challenge Project

Task: Write a Python or JavaScript program that:

- Reads the temperature and humidity data (simulate using input functions).
- Turns on the fan if the temperature is above 30°C and humidity is below 50%.
- Outputs "Fan is OFF" if either condition is not met.

Discussion

Why are TypeScript types and interfaces important for code reliability, especially in embedded systems?

Exercise 6 – Advanced JavaScript Control Structures

Task: Create a JavaScript function to simulate reading sensor data 10 times, and implement error handling using ***try/catch*** to manage potential errors (e.g., when data is *null*).

Instructions:

- Use a *for* loop to simulate 10 readings.
- Randomly generate sensor data but throw an error if a null value is encountered.
- Ensure the program continues executing even after an error.

Exercise 6 – Solution

```
function readSensorData() {
    return Math.random() > 0.1 ? Math.floor(Math.random() * 100) : null;
}
for (let i = 0; i < 10; i++) {
    try {
        let data = readSensorData();
        if (data === null) throw new Error("Sensor data is null");
        console.log(`Reading ${i + 1}: Sensor data = ${data}`);
    } catch (error) {
        console.log(`Reading ${i + 1}: Error occurred - ${error.message}`);
    }
}
```

Discussion

How does error handling in JavaScript contribute to building reliable systems, especially when dealing with unpredictable hardware data?

Error Handling in JavaScript for Reliable Systems

- **Handling Unpredictable Inputs:**
  Hardware devices (e.g., sensors) can return unexpected or null values.
  Proper error handling ensures that the system gracefully manages these situations without crashing.
- **Preventing System Failures:**
  catches runtime errors, preventing system-wide failures.
  Errors can be logged and handled without interrupting the system’s overall functionality.
- **Ensuring Data Integrity:**
  Validates sensor data before processing (e.g., rejecting null or out-of-range values).
  Helps maintain accurate and reliable data processing.

Error Handling in JavaScript for Reliable Systems

- **Asynchronous Error Handling:**
  Promises and *async/await* handle errors in asynchronous operations, crucial when dealing with hardware delays or communication timeouts.

- **Fault-Tolerant Systems:**
  Error handling creates fault-tolerant systems that continue running under adverse conditions (e.g., sensor disconnections or intermittent failures).

  The system can retry failed operations or switch to backup resources.

Exercise 7 – TypeScript Classes for Sensor Management

Task: Create a TypeScript class Sensor that simulates **sensor** data readings and stores the last 5 readings.

Instructions:

- The class should include methods to read data and retrieve the last 5 readings.
- Data should be stored in an array, and older readings should be removed once the array exceeds 5 entries.
- Simulate multiple readings and output the last 5 readings after each new entry.

Exercise 7 – Solution

```
class Sensor {
    private readings: number[] = [];

    readData(): number {
        const newData = Math.floor(Math.random() * 100);
        this.readings.push(newData);
        if (this.readings.length > 5) this.readings.shift();
        return newData;
    }

    getLastReadings(): number[] {
        return this.readings;
    }
}

const sensor = new Sensor();
for (let i = 0; i < 10; i++) {
    console.log(`New reading: ${sensor.readData()}`);
    console.log(`Last 5 readings: ${sensor.getLastReadings()}`);
}
```

Exercise 8 – Promises and Async/Await for Sensor Data Fetching

Task: Create a simulated asynchronous function in JavaScript/TypeScript that fetches sensor data from a remote server, returning the result after a delay.

Instructions:

- Implement both a **Promise**-based version and an ***async/await*** version of the function.
- Simulate data fetching with a ***setTimeout()*** to delay the result by 1 second.
- Handle success (data fetched) and failure (error occurred) cases.

Exercise 8 – Solution (Promise version)

```
function fetchSensorData(): Promise<number> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = Math.random() > 0.1 ? Math.floor(Math.random() * 100) : null;
            data !== null ? resolve(data) : reject(new Error("Failed to fetch data"));
        }, 1000);
    });
}

fetchSensorData()
    .then(data => console.log(`Sensor data: ${data}`))
    .catch(error => console.error(error.message));
```

Exercise 8 – Solution (Async/Await version)

```
async function fetchSensorDataAsync(): Promise<void> {
    try {
        const data = await fetchSensorData();
        console.log(`Sensor data: ${data}`);
    } catch (error) {
        console.error(error.message);
    }
}

fetchSensorDataAsync();
```

Exercise 9 – Mapping Data Structures with TypeScript Generics

Task: Create a generic **SensorMap<T>** class that maps sensor IDs to their corresponding data.

Instructions:

- Use TypeScript generics to allow the map to store any type of sensor data (***number, string***, etc.).
- The class should include methods to add sensor data, retrieve data by ID, and list all sensors and their values.

Exercise 9 – Solution (Async/Await version)

```
class SensorMap<T> {
    private map: Map<string, T> = new Map();

    addSensorData(sensorId: string, data: T): void {
        this.map.set(sensorId, data);
    }

    getSensorData(sensorId: string): T | undefined {
        return this.map.get(sensorId);
    }

    listSensors(): void {
        this.map.forEach((value, key) => {
            console.log(`Sensor ${key}: Data = ${value}`);
        });
    }
}

const tempSensor = new SensorMap<number>();
tempSensor.addSensorData("sensor_1", 25);
tempSensor.addSensorData("sensor_2", 30);
tempSensor.listSensors();
```

Challenge Project - Build a Simple Sensor Dashboard

Task: Build a simple web dashboard using TypeScript and JavaScript to display data from multiple sensors. The dashboard should:

- Fetch sensor data asynchronously using ***fetchSensorDataAsync***() from Exercise 8.
- Display the last 5 readings for each sensor in a dynamic table.
- Update the table in real time as new sensor data is fetched.

Recommended Libraries and Framework:
React JS (html, tailwind css, axios), Node JS (express, REST API)