**Module 1**

**Lecture 1: Introduction to Programming**

1. **Introduction to Programming in the Modern World**

* **Role and Importance of Programming**
* Programming is the process of providing a computer with a structured set of instructions to perform specific tasks.
* It plays a vital role in automating everyday tasks and enabling digital services through software applications.
* **Purpose of Programming Languages**
* Programming languages, such as JavaScript, serve as a medium of communication between humans and computers.
* They allow developers to write code that can be interpreted or compiled into machine-readable instructions.

1. **Case Study: Practical Applications in Daily Life (Jane's Scenario)**

* **Grocery Shopping**
* Barcode scanners and cash registers use embedded programs to:
  + Read product information.
  + Calculate total cost, apply taxes and discounts.
  + Generate a printed receipt.
* **Financial Transactions**
* Credit card payments trigger backend programs in point-of-sale systems that:
  + Authenticate payment details.
  + Communicate with banking systems.
  + Update the user's account balance.
* **Health Monitoring**
* Smartwatches contain software to:
  + Monitor real-time biometric data (e.g., heart rate).
  + Sync information to fitness applications.
  + Enable user control over data sharing and privacy.
* **Online Purchases**
* Ordering products online involves interconnected programs that:
  + Manage inventory, payments, and delivery logistics.
  + May rely on satellite communication and other network services.

1. **Fundamentals of Computer Programs**

* A **computer program** is a structured set of instructions designed to perform specific operations under predefined rules.
* Programs respond to inputs and produce outputs based on logic and conditions defined by the developer.

1. **Software and Hardware Distinction**

* **Software**: Refers to the code, applications, and operating systems that run on computing devices.
* **Hardware**: Denotes the physical components (e.g., CPU, RAM, I/O devices) that execute software instructions.

1. **Understanding How Computers Process Code**

* **Machine Language (Low-Level Language)**
* Code must be stored in memory (RAM or disk) in binary form (0s and 1s).
* Binary represents electrical signals that the CPU interprets directly.
* This machine-readable format is known as a **low-level language**.
* **High-Level Languages**
* Languages like **JavaScript** are considered high-level because they are closer to human language and abstract away hardware details.
* They require **interpretation or compilation** into binary before execution by the CPU.

**Lecture 2: Welcome to Programming**

**1. Understanding JavaScript: The Language of the Web**

* **Role of JavaScript in Web Interactivity**
* JavaScript enables interactive behavior on web pages.
* Common uses include:
  + Watching videos.
  + Using maps.
  + Navigating social media feeds.
* It's used heavily in online courses and learning platforms.
* **Core Benefits of JavaScript**
* **Client-Side Execution**: JavaScript runs directly in the browser, making real-time updates possible.
  + E.g., interactive maps, form validation, dynamic content.
* **Ubiquity**: Nearly all websites incorporate JavaScript in some form.
* **Longevity**: In use since 1995; remains central to front-end development.
* **Alternatives and Compatibility**
* Alternatives like **VBScript** and **TypeScript** exist.
  + TypeScript compiles down to JavaScript to maintain browser compatibility.
* JavaScript is the only language natively supported by all browsers for client-side scripting.
* **Importance of JavaScript in Modern Browsers**
* **Backward Compatibility**: Browsers maintain JavaScript to support legacy websites.
* Removing JavaScript would break millions of existing sites.
* Acts as a **core pillar** of web development alongside HTML and CSS.
* **Why Developers Choose JavaScript**
* **Ease of Use**:
  + Accessible for beginners via browser developer tools (console).
  + No complex setup needed initially.
* **Learning Path**:
  + Helps developers transition into other technologies and languages.
  + Tools like Node.js and frameworks like React expand its utility.
* **Community Support**:
  + Extensive online resources and forums.
  + Rich ecosystem of libraries and frameworks (Vue, D3, etc.).
* **JavaScript on Both Client and Server**
* **Client Side**: Vanilla JS powers front-end interfaces.
* **Server Side**: Tools like **Node.js** and **Deno** enable backend development with JavaScript.
  1. **JavaScript in the Real World**
* **Where JavaScript Is Used**
* **Client-Side (Browser)**:
  + Adds interactivity (e.g., adding items to a cart on click).
  + Enhances user experience through dynamic behavior.
* **Server-Side**:
  + Powers web applications.
  + Connects to and communicates with databases.
  + Supports backend development via Node.js and Deno.
* **Cross-Platform Development**:
  + Builds mobile applications using **React Native**.
  + Programs hardware and sensors in **Internet of Things (IoT)** devices.
* **Browser Compatibility Challenges**
* In the early 2000s, multiple browser vendors (e.g., Internet Explorer, Firefox) caused:
  + **Inconsistent behavior** across browsers.
  + Developers writing different JS code for each browser.
  + Increased development time and poor user experience.
* **jQuery: A Game-Changer**
* **jQuery** was introduced to address browser compatibility issues.
* Allowed developers to:
  + Write simplified, unified code.
  + Ensure compatibility across all major browsers.
* Became the **most popular JavaScript library** for over a decade.
* **Evolution Beyond jQuery**
* As web applications grew more complex, jQuery’s limitations became clear.
* **React (2011)** emerged to:
  + Simplify the creation, update, and maintenance of dynamic UIs.
  + Change the way we structure and think about component-based development.
* **Modern JavaScript Frameworks & Libraries**

Following Reacts success, many frameworks emerged:

* **Knockout**
* **Backbone**
* **Angular**
* **Ember**
* **Vue**
* **AlpineJS**
* Each aims to address specific challenges in building scalable and maintainable web applications.
* **Legacy Code and jQuery Today**
* **Legacy Code**: Older JavaScript code using libraries like jQuery.
* jQuery is still found in many active projects but is **rarely used** in modern development.
* Important to understand it for maintenance and migration purposes.
* **Learning Path for Developers**
* **Start with Vanilla JavaScript** (plain JS without frameworks).
* Mastering core JavaScript helps in:
  + Understanding how frameworks work internally.
  + Building scalable apps with tools like React or Vue.
* No need to learn every framework—**focus on the fundamentals first**.
  1. **Writing Your First JavaScript Code**
* You’ll learn:
  + How to write **comments**
  + What **semi-colons** do
  + How to run JavaScript using the **browser console**
  1. **Comments in JavaScript**
* **Why start with comments?**
* Easy to write
* Helps explain your code to yourself and others
* **Types of comments:**
* **Single-line**:  
  // this is a comment
* **Multi-line**:

/\*

this is

a multi-line

comment

\*/

* Multi-line can also be used in one line:  
  /\* comment \*/
* **Why comments matter:**
* Explain code
* Leave notes for yourself or teammates
* Mark tasks (e.g., TODO)
* Turn off parts of code without deleting them
* Useful for:
  + Testing ideas
  + Debugging problems
* **Semi-colons in JavaScript**
* Used to **separate code statements**
* Like a period (.) in English
* JavaScript can **add them automatically** (Automatic Semicolon Insertion or ASI)
* **Optional** in many cases, but adding them improves clarity
  1. **Using the Developer Console**
* **What you need:**
* A **browser** (e.g., Google Chrome)
* Right-click → **Inspect** → Open **Console** tab
* Or press ESC to toggle console
* **How to use it:**
* Type JavaScript code directly
* Use SHIFT + ENTER to go to a new line without running the code
* Use ENTER to run the code
  + **Output to the Console**
* **Basic example:**

console.log("Hello, World");

* **Styled output:**

console.log("%cHello, World", "color: blue; font-size: 40px");

%c applies styles using CSS

* + **Output Multiple Words**

**Using + to join words:**

console.log("Hello " + "there, " + "World");

**Using , to separate:**

console.log("Hello", "there,", "World");

* + joins into one string
* , logs each part separately, adding space between
  1. **Variables in JavaScript**

Variables are used to **store data** that can be used and modified later.

* **Declaring Variables**

JavaScript uses **three keywords** to declare variables:

1. var – *Old way (function-scoped)*
2. let – *Modern way (block-scoped)*
3. const – *For constants (block-scoped)*

* **Syntax**

let name = "John"; // Using let

const age = 25; // Using const

var city = "Paris"; // Using var

* **Rules for Naming Variables**
* Can include: letters, numbers, \_, $
* Cannot start with a number
* **Case-sensitive** (Name and name are different)
* Use **camelCase** by convention: userName, totalAmount
* **Differences Between var, let, and const**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Keyword | Scope | Reassignable | Redeclarable | Hoisted |
| var | Function | ✅ Yes | ✅ Yes | ✅ Yes (undefined) |
| let | Block | ✅ Yes | ❌ No | ✅ Yes (TDZ\*) |
| const | Block | ❌ No | ❌ No | ✅ Yes (TDZ\*) |

*TDZ = Temporal Dead Zone (accessible only after declaration)*

* **Examples**

let score = 100;

score = 200; // valid

const pi = 3.14;

// pi = 3.15; // ❌ Error: cannot reassign

var greeting = "Hi";

var greeting = "Hello"; // ✅ valid with var

**5. JavaScript Data Types**

* 1. **Primitive Data Types**
* **Primitive types** are immutable
* Passed **by value** (copied when assigned)
* typeof operator returns the type (except null)

### **1. String**

Represents textual data enclosed in quotes.

let name = "John"; *// Double quotes*

let greeting = 'Hello'; *// Single quotes*

let template = `User: ${name}`; *// Template literals (ES6)*

### **2. Number**

Represents both integer and floating-point numbers.

let age = 25; *// Integer*

let price = 99.99; *// Floating point*

let scientific = 2.998e8; *// Scientific notation*

let hex = 0xFF; *// Hexadecimal (255)*

### **3. Boolean**

Logical entity with true or false values.

let isOnline = true;

let hasPermission = false;

### **4. Undefined**

Variable declared but not initialized.

let score; *// typeof score → "undefined"*

let total = undefined; *// Explicit assignment (avoid this)*

### **5. Null**

Intentional absence of any object value.

let data = null; *// typeof data → "object" (historical quirk)*

### **6. Symbol (ES6)**

Unique and immutable primitive value.

let id = Symbol("id"); *// Unique identifier*

let key = Symbol.for("sharedKey"); *// Global symbol registry*

### **7. BigInt (ES2020)**

Arbitrary precision integers (suffix with 'n').

let bigNumber = 123456789012345678901234567890n;

let huge = BigInt("9007199254740991"); *// Constructor syntax*

1. **JavaScript Operators**
   1. **Arithmetic Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Output |
| + | Addition | 5 + 2 | 7 |
| - | Subtraction | 5 - 2 | 3 |
| \* | Multiplication | 5 \* 2 | 10 |
| / | Division | 10 / 2 | 5 |
| % | Modulus | 10 % 3 | 1 |
| \*\* | Exponentiation | 2 \*\* 3 | 8 |
| ++ | Increment | x = 1; x++ | 2 |
| -- | Decrement | x = 2; x-- | 1 |

**6.2 Comparison Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Output |
| == | Equal (loose) | 5 == '5' | true |
| === | Equal (strict) | 5 === '5' | false |
| != | Not Equal (loose) | 5 != '5' | false |
| !== | Not Equal (strict) | 5 !== '5' | true |
| > | Greater Than | 10 > 5 | true |
| < | Less Than | 5 < 10 | true |
| >= | Greater Than or Equal | 10 >= 10 | true |
| <= | Less Than or Equal | 5 <= 10 | true |

**6.3 Logical Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Output |
| && | Logical AND | true && false | false |
| || | Logical OR | True || false | True |
| ! | Logical NOT | !true | false |

**6.4 Assignment Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Meaning |
| = | Assign | x = 10 | Assign 10 to x |
| += | Add and assign | x += 5 | x = x + 5 |
| -= | Subtract and assign | x -= 3 | x = x - 3 |
| \*= | Multiply and assign | x \*= 2 | x = x \* 2 |
| /= | Divide and assign | x /= 2 | x = x / 2 |
| %= | Modulus and assign | x %= 3 | x = x % 3 |
| \*\*= | Power and assign | x \*\*= 2 | x = x \*\* 2 |

**6.5 Ternary Operator**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Output |
| ? : | Ternary | x > y ? 'Yes' : 'No' | 'Yes' or 'No' |

**6.6 Type / Misc Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Output |
| typeof | Type Check | typeof "Hello" | "string" |
| delete | Delete property | delete obj.name | true |
| in | Property exists check | "name" in obj | true/false |
| instanceof | Instance check | arr instanceof Array | true |

1. **Numbers in JavaScript**

* Used to represent both **integers** and **floating-point** numbers.
* No separate types for int, float, etc.
* Example:

let a = 10;

let b = 3.14;

let sum = a + b;

* **Special values**:
  + Infinity, -Infinity, NaN (Not a Number)

1. **Strings in JavaScript**

* Used to store **text** data.
* Can be written with:
  + **Single quotes** '...'
  + **Double quotes** "..."
  + **Backticks** `...` (template literals)
* Example:

let name = "Alice";

let greeting = `Hello, ${name}`;

* Common string methods:
  + **.length, .toUpperCase(), .toLowerCase(), .slice(), .replace()**

1. **Booleans in JavaScript**

* Represents **true** or **false**
* Often used in conditions, comparisons, and logic
* Example:

let isAdult = true;

let hasID = false;

* Used in control flow:

if (isAdult && hasID) {

console.log("Access granted");

}

**Lecture 3: Conditionals and Loops**

* 1. **Introduction to Conditional Statements**
* Conditional statements allow code to execute based on whether a condition is **true or false**.
* These are essential for making decisions in code (e.g., "if it's raining, take an umbrella").

**1.2 Types of Conditional Statements**

Used to make decisions based on conditions.

|  |  |  |
| --- | --- | --- |
| Statement | Syntax Example | Description |
| if | if (condition) { // code } | Runs code **if** condition is true |
| if...else | if (condition) { // } else { // } | Runs one block if true, else another |
| else if | if (...) {...} else if (...) {...} | Tests multiple conditions |
| switch | switch (value) { case x: break; ... } | Compares a value with multiple cases |

* **if Statement**
  + Executes a block of code **only if** the condition is true.
  + Syntax:

if (condition) {

// code to run if condition is true

}

* **if-else Statement**
  + Executes one block if condition is **true**, and another if **false**.
  + Example:

if (score > 40) {

console.log("You passed the test");

} else {

console.log("You did not pass the test");

}

* **else-if Statement**
  + Used to test **multiple conditions**.
  + Executes the first true condition, then **skips the rest**.
  + Always end with an optional else block as a fallback.
  + Example:

if (place == "first") {

console.log("Gold");

} else if (place == "second") {

console.log("Silver");

} else if (place == "third") {

console.log("Bronze");

} else {

console.log("No medal");

}

* **switch Statement**
  + Ideal for **multiple exact match** cases.
  + Easier to read and manage than multiple else-if blocks.
  + Uses case, break, and default.
  + Example:

switch(place) {

case "first":

console.log("Gold");

break;

case "second":

console.log("Silver");

break;

default:

console.log("No medal");

}

**1.3 Comparison Operators for Conditions**

* == (equal to)
* != (not equal to)
* > (greater than)
* < (less than)
* >=, <= (greater than or equal to, less than or equal to)
  1. **Break Statement**
* Used inside switch and loops to **exit early**.
* Prevents further code in the block from executing.
* Example:

for (let i = 1; i <= 5; i++) {

if (i === 3) {

break;

}

console.log(i);

}

// Output: 1, 2

**1.5 When to Use What?**

* **Use if-else:**
  + For **binary conditions** or simple logical checks.
  + When conditions involve **ranges or inequalities**.
* **Use switch:**
  + For **multiple fixed values** (like menu options, grades, etc.).
  + When all checks are based on **exact value matches**.

### **What Are Loops in JavaScript?**

* Loops let you **repeat code** until a condition is no longer true.
* Similar to if statements, but loops run **multiple times**.
* They use a **counter variable** (usually i) to track iterations.
* Without a proper counter or condition, loops can become **infinite**.
* **Real-Life Examples of Repetition**
* Closing shirt buttons.
* Washing multiple plates.
* Counting down from 10 before New Year.

**2.1 Why Use Loops Instead of Writing Repeated Code?**

* Efficient and cleaner code.
* Avoids repetitive manual writing (console.log(1)...console.log(100)).
* Especially useful when tasks scale (e.g., counting to 100 or 1000).
  1. **Types of Loops in JavaScript**

**2.2.1 for Loop**

* Syntax: for (start; condition; increment) { code }
* All loop logic (start, condition, increment) is **in one place**.
* Common structure:

for (let i = 1; i <= 3; i++) {

console.log(i);

}

console.log("Go!");

* You can also **count down**:

for (let i = 10; i > 0; i--) {

console.log(i);

}

console.log("Happy New Year!");

* **Exit condition** is critical to avoid infinite loops.
  + 1. **while Loop**
* Syntax:

let counter = 3;

while (counter > 0) {

console.log(counter);

counter--;

}

console.log("Happy New Year!");

* The **counter is declared outside**, and increment/decrement is **inside** the loop.
* Only the **condition** goes inside the while() declaration.

**Module 2**

**Lecture 1: Arrays, Objects and Functions**

* 1. **JavaScript Functions**
* **Purpose of Functions**: Automate repetitive tasks by grouping code under a single reusable unit.
* **DRY Principle**: "Don’t Repeat Yourself" – functions help reduce code duplication.
* **Function Declaration**:
  + Use function keyword followed by a name (e.g., addTwoNums()).
  + Add parentheses () and curly braces {} for the function body.
* **Function Invocation**:
  + A function must be explicitly *called* using its name followed by ().
* **Static vs Dynamic Functions**:
  + Initial example used fixed values (e.g., 10 + 20).
  + To make functions reusable, use **parameters** in declarations.
* **Parameters & Arguments**:
  + **Parameters**: Placeholder variables in the function definition.
  + **Arguments**: Actual values passed when invoking the function.
* **Flexibility with Parameters**:
  + Functions can take dynamic input, enabling code reuse with various values.
  + Example: addTwoNums(a, b) can be used with any two numbers.
  1. **JavaScript Arrays**
* **What is an Array**: A collection of ordered values stored under one variable.
* **Why Use Arrays**:
  + Group related data together.
  + Easily access data using index numbers.
* **Array Declaration**:
  + Use square brackets [] (e.g., let train1 = ["wheat", "coal", "iron"]).
* **Array Indexing**:
  + Index starts at 0.
  + Access using arrayName[index] (e.g., train1[0] returns "wheat").
* **Arrays vs Individual Variables**:
  + More efficient than creating multiple uniquely named variables.
  + Arrays represent a logical grouping (e.g., carriages of a train).
* **Function Example: Iterating Through an Array**
* **Function Declaration Example**:

function listArrayItems(arr) {

// code to iterate through array

}

* **Understanding the for loop**:
  + **Initialization**: let i = 0
  + **Condition**: i < arr.length
  + **Increment**: i++
* **Purpose**: Loops through each array element and performs actions like logging or transforming data.
  1. **Why Use Objects in JavaScript**
* Objects group **related data** together.
* Represent **real-world entities** (e.g., game characters, products).
* Make code **cleaner and easier** to manage.
* Avoids using multiple separate variables for one entity.

**3.1 Creating Objects**

* **Using Object Literal Notation**

var assistantManager = {

rangeTilesPerTurn: 3,

socialSkills: 30,

streetSmarts: 30,

health: 40,

specialAbility: "young and ambitious",

greeting: "Let's make some money"

};

* Properties = key-value pairs.
* Keys are like variable names; values are the data.
* **Using Dot Notation to Build or Update**

var storeManager = {}; // Create empty object

storeManager.rangeTilesPerTurn = 4;

storeManager.socialSkills = 50;

storeManager.streetSmarts = 40;

storeManager.health = 60;

storeManager.nextAchievement = "Double Sales";

**3.2 Accessing and Updating Object Properties**

* Use **dot notation**:

console.log(storeManager.health); // 60

storeManager.health = 70;

* Use **bracket notation**:

storeManager["health"] = 80;

console.log(storeManager["health"]); // 80

* **Bracket Notation Use Cases**
* Useful when:
  + Property name has **spaces**:

storeManager["team leadership"] = "Excellent";

* + Property name is a **dynamic string or number**:

var year = "2022";

car[year] = 1234; // same as car["2022"]

* **Key Takeaways**
* Dot and bracket notation are both valid.
* Bracket notation is more flexible for **complex or dynamic property names**.
* You can **combine both notations** in your code.
* Objects help structure data in an organized and readable way.

**3.3 Arrays Are Objects in JavaScript**

* Arrays are a special type of object.
* They come with built-in **properties and methods** like .push() and .pop().

**➕ push() Method**

* Adds items to the **end** of an array.
* Example:

var fruits = [];

fruits.push("apple"); // ['apple']

fruits.push("pear"); // ['apple', 'pear']

**➖ pop() Method**

* Removes the **last item** from an array.
* Example:

fruits.pop(); // removes 'pear'

**🔧 Building Arrays with Functions**

* Example function:

function arrayBuilder(one, two, three) {

var arr = [];

arr.push(one, two, three);

return arr;

}

var simpleArr = arrayBuilder('apple', 'pear', 'plum');

console.log(simpleArr); // ['apple', 'pear', 'plum']

**🧮 JavaScript Math Object Cheat Sheet**

**📌 Constants**

* Math.PI ≈ 3.14159
* Math.E ≈ 2.718
* Math.LN2 ≈ 0.693

**🔢 Rounding Methods**

* Math.ceil(x) → Rounds **up**
* Math.floor(x) → Rounds **down**
* Math.round(x) → Rounds **to nearest**
* Math.trunc(x) → **Removes** decimal part

**✖️ Arithmetic & Calculus Methods**

* Math.pow(2, 3) → 8
* Math.sqrt(16) → 4
* Math.cbrt(8) → 2
* Math.abs(-10) → 10

**🔍 Logarithmic Methods**

* Math.log(x) → Natural log
* Math.log2(x) → Base 2 log
* Math.log10(x) → Base 10 log

**📉 Min & Max**

* Math.min(3, 7, 2) → 2
* Math.max(3, 7, 2) → 7

**🔁 Trigonometry**

* Math.sin(), Math.cos(), Math.tan(), etc.

**🎲 Math.random() + Math.ceil()**

* Math.random() → Returns a decimal between **0 and <1**
* Multiply by 10 to scale: let decimal = Math.random() \* 10
* Use Math.ceil() to round up to the next integer:

js

CopyEdit

var rounded = Math.ceil(decimal);

console.log(rounded); // Value between 1–10

**🔁 What is an Iterable in JavaScript?**

* An **iterable** is a data type that can be looped over using a for...of loop.
* **Examples** of iterables: Arrays, Strings.

**🧵 Strings are Array-Like**

* Strings can be looped like arrays.
* Example:

js

CopyEdit

for (let char of "ABC") {

console.log(char); // A, B, C

}

**🥗 Array Example**

* veggies = ["onion", "parsley", "carrot"]

js

CopyEdit

console.log(veggies.length); // 3

console.log(veggies[0]); // onion

console.log(veggies[1]); // parsley

for (let i = 0; i < veggies.length; i++) {

console.log(veggies[i]);

}

**👋 String Example**

* greeting = "howdy"

js

CopyEdit

console.log(greeting.length); // 5

console.log(greeting[0]); // h

console.log(greeting[1]); // o

for (let i = 0; i < greeting.length; i++) {

console.log(greeting[i]); // h, o, w, d, y

}

**⚠️ Strings ≠ Arrays**

* Strings are **not** true arrays:
  + Cannot use array-only methods like .pop() on strings.

js

CopyEdit

let greet = "hello";

greet.pop(); // TypeError: greet.pop is not a function

**➕ String Concatenation**

* Use + operator or .concat() method to join strings:

js

CopyEdit

let greet = "Hello ";

let user = "Lisa";

console.log(greet + user); // Hello Lisa

console.log(greet.concat(user)); // Hello Lisa

**📘 Understanding Object Methods in JavaScript**

* **Objects** in JavaScript are made of **key-value pairs**, called **properties**.
* Properties can store any data type, including **functions**.
* When a **function is stored as a property**, it becomes a **method** of the object.
* You can add properties/methods using **dot notation**:

js

CopyEdit

var car = {};

car.color = "red"; // property

car.turnKey = function() {

console.log("engine running"); // method

};

**🛠 Working Example: The car Object**

* Create an empty object: var car = {};
* Add basic properties:
  + car.mileage = 98765;
  + car.color = "red";
* Add methods:
  + car.turnTheKey = function() { console.log("The engine is running"); };
  + car.lightsOn = function() { console.log("The lights are on."); };
* Invoke methods using the object:
  + car.turnTheKey(); ➝ Logs: *The engine is running*
  + car.lightsOn(); ➝ Logs: *The lights are on.*

**📌 Important Notes**

* A **method** is just a function attached to an object.
* You must call methods using the **object name + dot + method name + parentheses**.
* Example: console.log() — here, log is a method of the console object.

**Lecture 2: Error Handling**

**🔧 JavaScript: Bugs vs Errors**

**✅ Bug**

* **Definition**: Code runs but behaves **unexpectedly**.
* **Example**: Mixing types leads to unexpected output.

javascript

Copy code

function add(a, b) {

console.log(a + b);

}

add("1", 2); // Output: "12" (String concatenation, not addition)

console.log("Still running"); // Program continues running

* No crash, but result is not as intended → **Bug**.

**❌ Error**

* **Definition**: Code **stops executing** due to an invalid operation.
* **Example**: Using undeclared variables.

javascript

Copy code

console.log(c + d); // ReferenceError: c is not defined

console.log("This line never runs"); // Not executed

* Program **throws** an error and **halts** execution.

**🧠 Common JavaScript Error Types**

1. **ReferenceError**
   * Using a variable that hasn't been declared.
   * Example: console.log(a); when a is not defined.
2. **SyntaxError**
   * JavaScript **cannot parse** the code.
   * Example:

javascript

Copy code

let str = "Hello; // Missing closing quote → SyntaxError

1. **TypeError**
   * Performing an operation on the wrong **data type**.
   * Example:

javascript

Copy code

let num = 5;

num.pop(); // TypeError: num.pop is not a function

**🛠️ JavaScript Error Handling: try, catch, throw**

**🧩 Structure**

javascript

Copy code

try {

// Code that may throw an error

} catch (err) {

// Handle the error

}

**📌 Basic Example with Try-Catch**

javascript

Copy code

try {

console.log(a + b); // Error: a is not defined

} catch (err) {

console.log("There was an error");

console.log(err); // Logs ReferenceError

}

console.log("My program does not stop"); // Continues execution

**💥 Using throw to Manually Create an Error**

javascript

Copy code

try {

throw new ReferenceError("Manually thrown reference error");

} catch (err) {

console.log(err); // Custom error message

console.log("There was a reference error");

}

console.log("My program does not stop"); // Continues execution

* throw: Manually **raises an error**.
* new ReferenceError(...): Instantiates a specific error type.

**🔹 Common JavaScript Error Types**

1. **ReferenceError**: Accessing variables that haven’t been declared.
2. **SyntaxError**: Code written with invalid syntax (e.g., missing quotation marks).
3. **TypeError**: Using a method on a value of an incompatible type (e.g., 5.pop()).

**🔹 JavaScript Error Handling with try, catch, and throw**

* try block: Code to test for errors.
* catch(err): Code to execute if an error is thrown. err is the error object.
* throw: Manually throw custom errors (e.g., throw new ReferenceError("Custom error message")).

**🔹 Behavior of Error Handling**

* Without try-catch: Error stops program execution.
* With try-catch: Program continues after the error is caught.
* Errors can be logged and handled without crashing the program.

**🔹 Best Practices in JavaScript Error Handling**

* Use try-catch to handle potential runtime errors gracefully.
* Always name the error object in catch() meaningfully (e.g., err).
* Log error details for debugging: console.log(err.message) or console.error(err).
* Avoid catching errors silently – always handle or report them.

**🔹 Key Takeaways**

* Bugs cause **unexpected results** but don’t stop execution.
* Errors cause **termination** unless handled.
* Use try-catch to **control flow** and maintain program stability.
* Use throw to raise **custom exceptions** when needed.

**🔹 null Data Type**

* Represents the **intentional absence** of any object value.
* Returned by built-in methods (e.g., match) when a result **can’t be constructed**.
* Example:
  + 'abc'.match('a') returns a **match object** (array).
  + 'abc'.match('d') returns **null**, since no match is found.

**🔹 undefined Data Type**

* Indicates a variable has been **declared but not assigned** a value.
* Can only hold **one value**: undefined.
* Common scenarios:
  + Function **returns nothing** → default return is undefined.
  + Declaring a variable without assigning a value:

js

CopyEdit

let noise;

console.log(noise); // undefined

noise = 'thunder';

console.log(noise); // 'thunder'

* + Accessing **nonexistent object properties**:

js

CopyEdit

let game = { score: 1000 };

console.log(game.Score); // undefined (property doesn’t exist)

* Using a **completely undeclared variable** causes a **ReferenceError**, not undefined.

**🔹 Empty Strings ("" or '')**

* A **string** data type with **no characters** inside.
* Still a valid string, but considered **“empty”**.
* Can be created with:

js

CopyEdit

let name = ""; // or ''

**✅ Quick Comparison**

| **Type** | **Represents** | **Common Causes** |
| --- | --- | --- |
| null | **Intentional absence** of object | Method fails to return an object |
| undefined | **Uninitialized** variable or property | No assignment, wrong property access |
| "" | A **valid but empty string** | Explicitly set to empty string |

**Module 3**

**Lecture 1: Introduction to Functional Programming**

1. **Programming Paradigms**:

* Similar to human language styles, programming has **paradigms** (styles).
* Two common paradigms:
  + **Functional Programming (FP)**
  + **Object-Oriented Programming (OOP)**
* **Functional Programming (FP)**:
* **Data and functions are separate**.
* Functions receive data via arguments, process it, and return results.
* Promotes **pure functions** without side effects.
* **Object-Oriented Programming (OOP)** (briefly mentioned):
* Combines **data and functions** into objects.
* **Coding Example (Currency Conversion using FP)**:
* Three variables: currencyOne = 100, currencyTwo = 0, exchangeRate = 1.2.
* CamelCase naming convention used.
* A function convertCurrency(amount, rate) returns amount \* rate.
* Result stored in currencyTwo by calling convertCurrency(currencyOne, exchangeRate).
* Console outputs: **120** (i.e., 100 × 1.2)

1. **What is a Recursive Function?**

* A **recursive function** is a function that **calls itself** within its own code.
* It's useful for repetitive tasks but must be written carefully to **avoid infinite loops**.

**Basic Function Creation Example**

* A simple JavaScript function named example is created.
* It includes:
  + Three console.log statements outputting "line one", "line two", and "line three".
* When the function is run, these lines are executed **in sequence**.

**Creating an Infinite Loop**

* Adding example(); inside its own body causes the function to call itself **indefinitely**, creating an **infinite loop**.

**Fixing the Infinite Loop with a Counter**

* A counter variable is introduced and set to 3.
* Original log statements are replaced with:
  1. console.log(counter);
  2. counter = counter - 1;
  3. An if statement: if (counter === 0) return;
* Now, when the function runs, it logs 3, 2, 1 and **stops**—preventing an infinite loop.

**Purpose and Importance of Recursion**

* **Recursion** is an alternative to **loops** for repeating code.
* Must include a **base case** (stopping condition) to prevent endless repetition.

**Functional Programming Paradigm**

* **Definition**: Functional programming (FP) is a coding style that emphasizes the use of functions and immutable values.
* **Contrast with OOP**: Differs from Object-Oriented Programming (OOP) by separating data and functionality versus grouping them in objects.

**Core Concepts**

1. **Immutability**: Variables are not changed after initial assignment.
2. **Function Usage**: Functions are used extensively, and new values are returned rather than modifying existing ones.
3. **Data Separation**: Data and functionality are kept separate, enhancing clarity and modularity.

**Additional Concepts**

* **First-Class Functions**: Functions are treated as values, allowing them to be passed as arguments, saved in variables, or returned from other functions.
* **Higher-Order Functions**: Functions that accept other functions as arguments or return them as results.
* **Pure Functions**: Always produce the same output for the same input, without side-effects.
* **Side-Effects**: Modifications outside the function's scope, such as changing external variables or interacting with I/O operations.

**🔹 Scope in JavaScript**

* **Scope** determines the accessibility of variables and functions in different parts of the code.
* Two main types of scope:
  + **Global Scope**: Accessible anywhere in the code.
  + **Local (Function) Scope**: Accessible only inside the function where the variable is declared.
* **Scope Chain**: Functions have access to variables in their parent scopes, forming a chain of references.

**🔹 Types of Scope**

* **Global Scope**: Variables declared outside functions.
* **Local/Function Scope**: Variables declared inside functions (available only within).
* **Block Scope (introduced in ES6)**:
  + Created with {} such as in if, for, or while.
  + Applicable when using let and const.
  + Variables declared with let or const are restricted to the block.

**🔹 Keywords: var, let, and const**

**var**

* Function-scoped.
* Can be **accessed before declaration** (due to **hoisting**) – outputs undefined.
* Can be **redeclared** and **reassigned** without error.
* **Does not create block scope** – accessible outside of {} blocks.

**let**

* **Block-scoped**.
* **Cannot be accessed before declaration** (throws **ReferenceError**).
* Can be **reassigned** but **not redeclared** in the same scope.
* Better for variables that may change over time.

**const**

* **Block-scoped**.
* **Must be initialized** during declaration.
* Cannot be **redeclared** or **reassigned**.
* Ideal for values that should remain constant.

**🔹 Best Practices**

* Use \*\*let\*\* if the value will change.
* Use \*\*const\*\* for fixed values.
* Avoid var in modern JavaScript due to its loose scoping rules and hoisting behavior.

**Lecture 2: Introduction to Object-Oriented Programming**

**Programming Paradigms Overview**

* Programming paradigms are styles of writing and organizing code.
* Two major paradigms: **Functional Programming** and **Object-Oriented Programming (OOP)**.

**🔹 Functional Programming (FP)**

* Data and functions are **separated**.
* Focuses on **pure functions** and immutability.
* Example:
  + Variables for data: shoes = 100, tax = 1.2.
  + A function totalPrice(shoes, tax) returns shoes \* tax.
  + Output via console.log.

**🔹 Object-Oriented Programming (OOP)**

* Organizes code using **objects** which bundle **data and methods**.
* Functions inside objects are called **methods**.
* Example:
  + purchase1 is an object with shoes, tax, and totalPrice() method.
  + purchase1.totalPrice() uses internal object data to compute total.
  + Dot notation (purchase1.totalPrice) is used to access members.

**🔹 Using this Keyword in OOP**

* this refers to the **current object instance**.
* Allows reusability of methods across multiple objects.
* Reduces duplication: method uses this.shoes instead of hardcoding purchase1.shoes.

**🔹 Improving OOP Efficiency with Classes**

* Avoid duplicating method definitions across multiple objects.
* Use **classes** as templates to define object structure and behavior.
* Enables creation of multiple object instances with **shared logic**.

**🔹 Conclusion**

* **Functional Programming**: Separates data and functions.
* **OOP**: Combines data and behavior in objects, uses this for reusability, and classes for scalability.
* OOP helps write **efficient, maintainable, and reusable** code.

### 🔑 **Key Points: JavaScript Classes**

* **Purpose of Classes**:
  + Used to efficiently create multiple objects with the same structure (properties and methods).
  + Act as **blueprints** for building object instances (e.g., hundreds of car objects in a game).
* **Class Declaration**:
  + Use the class keyword followed by the **class name** (capitalized).
  + Enclosed in {} braces.
* **Constructor Function**:
  + Declared inside the class using constructor(...).
  + Accepts parameters to initialize object properties.
  + Called automatically during **object instantiation**.
* **Methods in Classes**:
  + Declared without the function keyword.
  + Defined directly inside the class after the constructor.
  + Can accept parameters like regular functions.
* **Creating Object Instances**:
  + Use new ClassName(...) to instantiate.
  + Resulting object (instance) gets access to all class-defined properties and methods.
* **Accessing Methods and Properties**:
  + Use **dot notation**: objectName.methodName()
  + Methods operate on the instance’s data and can return outputs or perform actions.
* **Benefits of Using Classes**:
  + Promotes **code reuse** and **modularity**.
  + Enhances **readability** and **maintainability**.
  + Efficient for building large-scale applications with many similar objects.

**✅ Benefits of OOP**

* **Real-world modeling:** Mimics real-life relationships between objects.
* **Modular code:** Encourages separation into independent modules.
* **Flexible design:** Easier to modify and extend.
* **Code reusability:** Enables reuse through inheritance and composition.

**🔑 Four Core OOP Principles**

1. **Inheritance**
   * Enables a class (sub-class) to inherit properties and methods from another class (super-class).
   * Promotes **code reuse** and **hierarchical structuring**.
   * Syntax in JavaScript: class B extends A.
2. **Encapsulation**
   * Hides internal implementation details from users.
   * Promotes **information hiding** and **interface-based usage**.
   * Example: Using "abc".toUpperCase() without needing to understand its implementation.
3. **Abstraction**
   * Focuses on **relevant data and behavior**, ignoring complex details.
   * Helps in designing **generalized solutions**.
   * Often confused with encapsulation, but abstraction is conceptual, while encapsulation is technical.
4. **Polymorphism**
   * Allows the **same method name** to behave **differently** across classes.
   * Achieved through **method overriding** and **method overloading**.
   * Example: ringTheBell() acts differently for bicycle and door objects.

**🔧 Practical JavaScript OOP Examples**

* **Object creation:**
  + Object.create() method
  + new keyword with classes
* **Inheritance hierarchy example:**

js

CopyEdit

class Animal {}

class Mammal extends Animal {}

class Elephant extends Mammal {}

* **Polymorphism Example:**

js

CopyEdit

class Bird {

useWings() { console.log("Flying!"); }

}

class Eagle extends Bird {

useWings() {

super.useWings();

console.log("Barely flapping!");

}

}

**🧠 Summary**

* OOP **structures and simplifies** software development by:
  + Modeling real-world entities.
  + Promoting **cleaner, more maintainable**, and **scalable** code.
  + Utilizing **inheritance, encapsulation, abstraction,** and **polymorphism** to achieve modularity and flexibility.

**What Are Constructors?**

* Special functions to create and initialize objects.
* Often used with the new keyword.
* Built-in constructors include Date, Array, RegExp, etc.

**🏗️ Using Built-in Constructors**

* new Date() → Creates a date object with current date/time.
* Math is not a constructor → new Math() throws an error.
  + Use static access: Math.pow(2, 5) → 32.

**🧊 Custom Constructor Functions**

javascript

CopyEdit

function Icecream(flavor) {

this.flavor = flavor;

this.meltIt = function() {

console.log(`The ${this.flavor} icecream has melted`);

};

}

let kiwi = new Icecream("kiwi");

let apple = new Icecream("apple");

* Creates instances of Icecream with unique properties.

**⚠️ Constructor vs. Literals – Performance Tip**

* Avoid using constructors for **primitive types**:
  + ❌ new String("apple") → Object
  + ✅ "pear" → Primitive (more efficient)
* Comparing objects: new String('a') !== new String('a') → false

**🧱 Use Literals Instead**

* Object: {} instead of new Object()
* Array: [] instead of new Array()
* Function: function() {} instead of new Function()
* RegExp: /d/ instead of new RegExp("d")

**✅ Constructors You Can Use Safely**

* new Date()
* new Error()
* new Map()
* new Set()
* new Promise()
* new WeakMap(), new WeakSet()

**🧠 Key Takeaways**

* Use constructor functions for custom and certain built-in objects.
* Prefer literals for performance and readability.
* Understand when constructors are necessary and when they’re

### **🔑 Key Concepts of JavaScript Inheritance**

* **Inheritance Definition**:
  + In JavaScript, inheritance allows one object to access properties and methods of another object.
  + It is implemented using the **prototype** system.
* **Prototype**:
  + A **prototype** is an object from which other objects inherit properties.
  + Acts as a shared reference for methods and properties.
* **Prototype Inheritance Model**:
  + JavaScript uses a **prototype-based inheritance** rather than class-based inheritance (like in Java or C++).
  + Objects inherit directly from other objects using their prototype chain.

**🧱 Using Object.create() Method**

* Object.create(proto) creates a new object with the specified prototype object.

**Example:**

javascript

CopyEdit

var bird = {

hasWings: true,

canFly: true,

hasFeathers: true

};

var eagle1 = Object.create(bird);

* eagle1 is an empty object but **inherits** properties from bird.
* Logging eagle1.hasWings returns true through its prototype.

**🦅 Creating Multiple Objects with Same Prototype**

* You can create multiple objects like eagle2 using the same prototype:

javascript

CopyEdit

var eagle2 = Object.create(bird);

* Each new object accesses the prototype’s properties independently.

**🐧 Overriding Prototype Properties**

* Properties can be **overridden** in the child object:

javascript

CopyEdit

var penguin1 = Object.create(bird);

penguin1.canFly = false;

* Now penguin1.canFly is false because it's set directly on the object, overriding the prototype's value.

**🔄 Property Lookup Mechanism**

* JS checks properties:
  1. On the object itself.
  2. If not found, it **delegates to the prototype**.
* Once a property is found, **lookup stops**.

**✅ Key Benefits & Notes**

* Objects can share behavior without duplication.
* Inheritance using Object.create() is simple and clear.
* **Overridden properties** only affect the instance object.
* For complex use-cases, **use class syntax** which is syntactic sugar over prototypes but easier to manage and more readable.

**🔹 Basic Class Creation**

* Use class ClassName {} to define a new class.
* Capitalize the class name (e.g., Train).
* Inside the class, use a constructor() to define instance properties.

js

CopyEdit

class Train {

constructor(color, lightsOn) {

this.color = color;

this.lightsOn = lightsOn;

}

}

**🔹 Creating Object Instances**

* Use new to create an instance:  
  let train1 = new Train('red', false);
* Each instance has its own properties.

**🔹 Adding Methods to Classes**

* Methods go **outside** the constructor.
* Shared across all instances via the prototype.

js

CopyEdit

toggleLights() { this.lightsOn = !this.lightsOn; }

lightsStatus() { console.log('Lights on?', this.lightsOn); }

getSelf() { console.log(this); }

getPrototype() { console.log(Object.getPrototypeOf(this)); }

**🔹 Prototype Concept**

* constructor assigns **instance-specific data**.
* Methods are stored on the **prototype**, shared by instances.

**🔹 Inheritance**

* Use extends to create a subclass:  
  class HighSpeedTrain extends Train {}
* Use super() in subclass constructor to call parent constructor.

js

CopyEdit

class HighSpeedTrain extends Train {

constructor(passengers, highSpeedOn, color, lightsOn) {

super(color, lightsOn); // Call parent constructor

this.passengers = passengers;

this.highSpeedOn = highSpeedOn;

}

}

**🔹 Polymorphism (Overriding Methods)**

* Subclasses can override inherited methods.
* Can also add new methods unique to the subclass.

**🔑 Key Concepts**

* **extends**: Sets up inheritance between classes.
* **super**: Calls the parent class's constructor or methods.

**🧱 Class Hierarchy**

markdown

CopyEdit

Animal

/ \

Cat Bird

/ \ \

HouseCat Tiger Parrot

**🐾 Animal Class**

* **Properties**: color, energy
* **Methods**:
  + isActive(): Decrease energy; if 0, calls sleep().
  + sleep(): Increases energy by 20.
  + getColor(): Logs color.

**🐱 Cat Class (extends Animal)**

* **Properties**: sound, canJumpHigh, canClimbTrees
* **Method**: makeSound()

**🐦 Bird Class (extends Animal)**

* **Properties**: sound, canFly
* **Method**: makeSound()

**🐈 HouseCat Class (extends Cat)**

* **Property**: houseCatSound
* **Overridden Method**:
  + makeSound(option)
    - If option === true: calls super.makeSound() + logs houseCatSound
    - Else: just logs houseCatSound

**🐅 Tiger Class (extends Cat)**

* **Property**: tigerSound
* **Method**: Same logic as HouseCat but with tigerSound

**🦜 Parrot Class (extends Bird)**

* **Property**: canTalk
* **Method**:
  + makeSound(option)
    - Logs sound
    - If option === true and canTalk === true, logs "I'm a talking parrot!"

**🧪 Example Usage**

js

CopyEdit

var leo = new HouseCat();

leo.makeSound(false); // meow

leo.makeSound(true); // purr, meow

var cuddles = new Tiger();

cuddles.makeSound(false); // Roar!

cuddles.makeSound(true); // purr, Roar!

var polly = new Parrot(true);

var fiji = new Parrot(false);

polly.makeSound(true); // chirp, I'm a talking parrot!

fiji.makeSound(true); // chirp

**Lecture 3: Advanced JavaScript Features**

**🎯 Destructuring in JavaScript (Objects & Arrays)**

**🔹 Concept Overview:**

* Destructuring = extracting values from objects/arrays into distinct variables.
* Similar to copying formatted text style from one portion to another in a word processor.
* Destructured values are **independent copies** of the original data.

**🔹 Object Destructuring Example:**

let { PI } = Math;

* Copies Math.PI into a variable named PI.
* Only works if property exists **and** spelling is exact (case-sensitive).

**🔹 Common Mistake:**

let { pi } = Math; // returns undefined

* Fails because Math has PI, **not** pi (case matters).

**🔹 Proof of Independence:**

PI === Math.PI; // true

PI = 1;

PI === Math.PI; // false

* Reassigning PI doesn’t affect Math.PI.

**🔹 Key Takeaways:**

* Destructuring creates **separate variables**, **not references**.
* Ideal for cleanly extracting specific data from large structures.

**🔹 Understanding for...of Loop**

* for...of **cannot iterate over objects directly** because objects are **not iterable**.
* It is designed for iterating over **iterable data types** like:
  + Arrays
  + Strings
  + Maps
  + Sets

**🔹 Example**

const colors = ['red', 'orange', 'yellow'];

for (const color of colors) {

console.log(color); // red, orange, yellow

}

**🔹 Error Example with Object**

const car = { speed: 100, color: 'blue' };

for (const prop of car) {

console.log(prop); // ❌ TypeError: car is not iterable

}

**🔹 How to Iterate Over Objects with for...of**

Use object transformation methods:

* Object.keys(obj) → Returns array of property names
* Object.values(obj) → Returns array of property values
* Object.entries(obj) → Returns array of [key, value] pairs

**✅ Example using Object.keys():**

for (const key of Object.keys(car)) {

console.log(key, ":", car[key]);

}

**✅ Example using Object.entries():**

for (const [key, value] of Object.entries(car)) {

console.log(key, ":", value);

}

**🔹 Dynamic Property Access**

* Use **bracket notation** for dynamic keys:

let key = 'speed';

console.log(car[key]); // Accesses car.speed

**🔹 for...in Loop vs for...of Loop**

| **Feature** | **for...in** | **for...of** |
| --- | --- | --- |
| Iterates over | Object keys (including inherited) | Values of iterables (arrays, etc.) |
| Use case | Looping through object properties | Looping through array-like structures |

**Example:**

js

CopyEdit

for (let key in car) {

console.log(key); // Outputs: speed, color

}

**🔹 Summary**

* for...of is **not for objects** directly—use transformation methods.
* Use for...in to loop through **object keys**.
* Use Object.entries() with for...of for **both keys and values**.
* Bracket notation helps with **dynamic key access**.

**🧵 ES5 String Limitations**

* Strings are enclosed in **single ('')** or **double ("")** quotes.
* **Multi-line strings** are **not supported**.
* Concatenation uses the **plus (+) operator**.
* Splitting strings over multiple lines causes **syntax errors** (unterminated string literal).

**✅ ES6 Template Literals**

* Introduced in **ES6** using **backticks (`)** instead of quotes.
* **Support multi-line strings** naturally.
* No syntax errors when spanning lines.
* **Backtick location**: Usually above the **Tab key** on your keyboard.

**🧠 Variable Interpolation**

* Combine variables directly inside strings using:

js

CopyEdit

`Hello, ${variableName}!`

* No need for + operator.
* **Quotes inside template literals** don’t break the string—they're treated as characters.

**🗂️ Importance of Data Representation**

* How data is **structured** influences how it is **processed**.
* Good data representation leads to a more **efficient** and **readable** solution.

**🔢 Inefficient Data Representations**

* Using a **string**: Hard to extract numeric values for calculation.
* Using **individual variables**: Less scalable, more error-prone.

**✅ Efficient Data Representation**

* Using an **array**: Groups related values (e.g., grades) under one label.
  + Easier to **iterate**, calculate sum, and derive average.

**🧠 Data Structures in JavaScript**

JavaScript supports a limited but useful set of data structures:

**1. Object**

* **Unordered**, non-iterable collection of **key-value pairs**.
* Useful for accessing data via a **key**.
* Common in **object-oriented programming**.

**2. Array**

* **Ordered**, iterable collection of values.
* Access via **index** (auto-assigned).
* Ideal for **grouping** and **looping** over similar data.

**3. Map**

* Like an object but:
  + **Iterable** collection of **key-value pairs**.
  + **Keys can be of any type** (not limited to strings or symbols).
* Useful when key types vary or order matters.

**4. Set**

* **Collection of unique values**.
* Prevents duplicate entries.
* Adding duplicates is ignored (no error, no update).

**🧮 Applying Arrays for Average Calculation**

* Loop over array (e.g., for loop).
* Sum all grades.
* Divide sum by array.length to get average.

**💡 Key Takeaway**

* The **choice of data structure** directly impacts the **logic and efficiency** of your code.
* **Choose based on task** needs: uniqueness, order, key-value access, or iteration.

**📌 JavaScript Data Structures: Key Points**

**🔹 Overview**

* Focus on JavaScript’s built-in data structures: **Object, Array, Map, Set**
* Excludes advanced structures like Queues, Linked Lists, Trees, Graphs (can be custom-coded in JS)

**🔸 Working with Arrays**

**1. forEach()**

* Used for **iterating** over array elements.
* Syntax: array.forEach(callback)
* Callback receives element and index.
* Does **not return** a new array.

js

CopyEdit

fruits.forEach((fruit, index) => console.log(`${index}. ${fruit}`));

**2. filter()**

* Creates a **new array** with elements that pass a **test condition**.
* Useful for **conditional selection** of items.

js

CopyEdit

const result = nums.filter(num => num > 20); // [30, 40, 50]

**3. map()**

* Transforms each array element and returns a **new array**.
* Often used for **value transformation**.

js

CopyEdit

const result = nums.map(num => num / 10); // [0, 1, 2, 3, 4, 5]

**🔸 Working with Objects**

* JavaScript object: key-value data structure.
* Example: Convert object to array using Object.keys() and forEach().

js

CopyEdit

const result = [];

Object.keys(drone).forEach(key => result.push(key, drone[key]));

* Tip: If you're converting objects frequently, you might need a different structure (e.g., Map).

**🔸 Working with Maps**

* Use new Map() to create a map.
* Stores key-value pairs **without inheritance**.
* Methods:
  + set(key, value) – adds entry
  + get(key) – retrieves value

js

CopyEdit

let bestBoxers = new Map();

bestBoxers.set(1, "The Champion");

bestBoxers.get(1); // 'The Champion'

**🔸 Working with Sets**

* Use new Set() to create a set.
* Stores **unique values only**.
* Can be initialized with an array to remove duplicates.

js

CopyEdit

const uniqueFruits = new Set(['apple', 'pear', 'apple']); // {'apple', 'pear'}

**🔸 Other Data Structures (Advanced - Not Covered Here)**

* Custom-implementable in JS, but not built-in:
  + Queues
  + Linked Lists
  + Trees
  + Graphs

**✅ Spread Operator (...)**

* **Purpose**: Expands arrays, objects, or strings into individual elements or properties.
* **Syntax**: ...array or ...object
* **Common Use Cases**:
  + **Pass array elements as arguments to functions**:

js

CopyEdit

showItinerary(...top7);

* + **Clone arrays/objects** (shallow copy):

js

CopyEdit

const newArr = [...oldArr];

const newObj = {...oldObj};

* + **Concatenate arrays**:

js

CopyEdit

const combined = [...arr1, ...arr2];

* + **Merge objects**:

js

CopyEdit

const merged = {...obj1, ...obj2};

* + **Convert string to array**:

js

CopyEdit

const chars = [..."hello"];

**✅ Rest Operator (...)**

* **Purpose**: Gathers multiple elements into a single array or object.
* **Used In**:
  + **Array destructuring**:

js

CopyEdit

const [first, ...rest] = [1, 2, 3];

* + **Function parameters**:

js

CopyEdit

function sum(...nums) {

return nums.reduce((a, b) => a + b, 0);

}

**🆚 Spread vs Rest**

| **Feature** | **Spread** | **Rest** |
| --- | --- | --- |
| Function | Expands | Collects |
| Use Case | Copy, merge, pass args | Destructuring, collect args |
| Placement | Anywhere | Must be **last** in parameter list |

**✅ Examples**

* **Join Arrays**:

js

CopyEdit

const allFruits = [...fruits, ...berries];

* **Join Objects**:

js

CopyEdit

const flyingCar = {...flying, ...car};

* **Destructuring with Rest**:

js

CopyEdit

const [first, second, ...others] = top7;

* **Function with Rest Parameter**:

js

CopyEdit

function addTaxToPrices(tax, ...prices) {

return prices.map(price => price + tax);

}

**📝 Key Takeaways**

* **Spread** simplifies passing multiple array items or merging arrays/objects.
* **Rest** simplifies collecting leftovers during destructuring or capturing unlimited function arguments.
* Both operators use the same ... syntax but differ by **context**.

**Lecture 4: JavaScript in the Browser​**

**Introduction to JavaScript Modules**

* JavaScript programs can be complex and used across multiple applications.
* Like camera lenses in photography, reusable code components (modules) offer flexibility without redundancy.
* Since **ES6**, JavaScript supports **native modules** for modular code architecture.

**What Are JavaScript Modules?**

* **Modules** are self-contained blocks of code that can be imported/exported across files.
* They enable **code reuse**, **encapsulation**, and **maintainability**.
* Modules can be **added**, **removed**, or **replaced** without breaking the program.

**JavaScript Before Modules**

* In older JavaScript versions, all functions declared globally resided on the window object.
* This global scope often caused **naming collisions** and **conflicts** between scripts.
* Workarounds were developed but had **limitations and risks**.

**CommonJS – Early Attempt at Modularity**

* Introduced by **Kevin Bangor (Mozilla)** through the **ServerJS project**, later renamed to **CommonJS**.
* Widely adopted in **Node.js** for server-side JavaScript.
* **Limitation**: Not supported natively by browsers (e.g., require, module.exports not recognized).

**ES6 Modules in the Browser**

* Uses <script type="module"> to define a module in HTML.
* Example: import a module like greeting.js in a script tag.
* **Benefits**:
  + Cleaner syntax
  + Built-in support by modern browsers
* **Console output** can be verified using browser developer tools.

**CORS and Local Server Requirement**

* Running ES6 modules locally may trigger **CORS (Cross-Origin Resource Sharing)** errors.
* To avoid this, serve HTML via a **local server**.
* Reference provided resources for **setting up a local server**.

**Conclusion**

* ES6 modules provide a **standard, secure, and scalable** way to organize JavaScript code.
* Developers can now build more **modular**, **maintainable**, and **robust** web applications using built-in module functionality.

### **1. DOM as a Control Interface for Webpages**

The Document Object Model (DOM) functions like a remote control for webpages, enabling developers to update and modify individual elements (e.g., images, headings) without reloading the entire page. Unlike a traditional TV remote, the DOM offers granular control over web content.

**2. What is the DOM?**

The DOM is a JavaScript object representation of the entire HTML document. It is automatically created by the browser when a webpage loads and is structured in a tree format with nested nodes for elements, attributes, and text.

**3. Accessing the DOM**

Developers can access and interact with the DOM using the browser’s built-in Developer Tools:

* **Elements Tab**: A visual, GUI-based interface to inspect and modify the DOM.
* **Console Tab**: A JavaScript console to manipulate the DOM programmatically.

You can access Developer Tools by right-clicking on a webpage and selecting **“Inspect”**.

**4. Manipulating the DOM Using JavaScript**

Common DOM manipulation techniques include:

* **Creating Elements**:

javascript

CopyEdit

const h2 = document.createElement('h2');

* **Setting Text Content**:

javascript

CopyEdit

h2.innerText = 'This is an h2 heading';

* **Adding Attributes**:

javascript

CopyEdit

h2.setAttribute('id', 'sub-heading');

h2.setAttribute('class', 'secondary');

* **Appending to the DOM**:

javascript

CopyEdit

document.body.appendChild(h2);

These methods allow dynamic updates to webpage content in real time.

**5. Understanding the Scope of DOM Changes**

Any modifications made through the browser's console are **local and temporary**. Reloading the webpage will revert all changes, as the DOM resets to match the HTML file downloaded from the server.

**6. Example: Adding a Heading Element**

A live demonstration involves:

* Creating an <h2> element using createElement().
* Adding text and attributes (innerText, setAttribute).
* Appending it to the page using appendChild().

This showcases how developers can use the DOM to inject custom content into a live webpage locally.

**7. Summary**

The DOM is a powerful tool that enables developers to programmatically interact with and modify web content. Understanding how to manipulate the DOM using JavaScript is fundamental for building dynamic and responsive web applications.

**1. Power of JavaScript DOM Manipulation**

* JavaScript enables real-time updates to HTML content (e.g., changing text color, showing popups).
* Interaction with HTML elements is dynamic and driven by DOM manipulation.

**2. Accessing the DOM in JavaScript**

* The DOM (Document Object Model) represents the webpage loaded in browser memory.
* Access it using the document keyword.
* You can log the DOM to the console with console.log(document).

**3. Selecting Elements with querySelector**

* **document.querySelector('tag')** returns the **first** matching HTML element.
  + Example: document.querySelector('p') returns the first <p> element.
  + Works with any valid HTML tag (e.g., 'a' for anchor tags).

**4. Selecting Multiple Elements with querySelectorAll**

* **document.querySelectorAll('tag')** returns **all** matching elements as a NodeList.
  + Example: document.querySelectorAll('p') returns all <p> tags.

**5. Selecting Elements by ID**

* **document.getElementById('id')** fetches a single element with a specific ID.
  + Example: document.getElementById('heading') returns the element with id="heading".

**6. Selecting Elements by Class Name**

* **document.getElementsByClassName('class')** returns all elements with a specific class.
  + Example: document.getElementsByClassName('txt') returns all elements with class="txt".

**7. Important Notes for Beginners**

* Use singular: getElementById (returns one element).
* Use plural: getElementsByClassName (returns a collection).
* If no match:
  + **ID selector** returns null.
  + **Class selector** returns an **empty collection** ([]).

**1. Introduction to JavaScript Events**

* **Events** are user interactions such as button clicks or icon taps.
* JavaScript allows you to **listen** for these events and **execute code** in response.
* Events can be targeted to specific elements on a webpage.

**2. Common Use Case: Add to Cart Interaction**

* Example: Clicking an "Add to Cart" button triggers an event.
* Event handler updates UI (e.g., increases the item count beside a shopping cart icon).
* Repeated clicks trigger the handler again, updating the UI accordingly.

**3. Event Handlers in JavaScript**

* An **event handler** is a function that runs in response to an event.
* Events must be **captured** before they can be **handled**.

**4. Using addEventListener Method**

* Steps to add an event listener with addEventListener:
  1. Select the target element using document.querySelector('element').
  2. Assign it to a variable (e.g., const target = document.querySelector('body');).
  3. Create a function (e.g., function handleClick() { console.log('clicked the body'); }).
  4. Add the event listener: target.addEventListener('click', handleClick);.
* Clicking the body will now log a message to the console.

**5. Alternative: HTML Event Attributes**

* Events can also be handled directly within HTML using event attributes like onclick.
* Steps:
  1. Write a second function (e.g., function handleClick2() { console.log('clicked the heading'); }).
  2. Edit HTML: Add onclick="handleClick2()" directly to the desired element (e.g., <h1>).
* Clicking the heading will now trigger both the body and heading event handlers.

**6. Multiple Listeners on a Single Event**

* A single click can trigger multiple event handlers if different elements are targeted (e.g., body and heading).
* Clicking outside a specific element only triggers the broader event handler (like on the body).

**7. Summary: Event Listening in JavaScript**

* Two primary ways to handle events:
  1. **addEventListener method** (JavaScript-based).
  2. **HTML event attributes** (inline in HTML).
* Both methods effectively respond to user interactions, enhancing interactivity on webpages.

### **Moving Data Around on the Web**

* The web consists of interconnected pages, services, and databases.
* Websites communicate using data from other sources (feeds, providers).
* A common data format is required for this data exchange.

**Introduction to JSON**

* JSON (JavaScript Object Notation) was created by Douglas Crockford in 2001.
* It replaced XML as the primary data interchange format.
* **Why JSON became popular:**
  + **Lightweight**: Less verbose than XML.
  + **Easy in JavaScript**: Directly usable in JS code.

**JSON vs JavaScript**

* JSON is a strict subset of JavaScript.
  + All JSON is valid JavaScript.
  + Not all JavaScript is valid JSON.
* JSON must follow stricter rules (e.g., keys in double quotes).

**Using JSON in Web Applications**

* Often used to fetch data from external APIs or services.
* Example: A file like stockPrices.json may contain stock market data.
* Once fetched, JSON is parsed into JS objects for processing.

**Accessing Parsed JSON**

* Example structure:

js

CopyEdit

const currencyInfo = {

USD: { ... },

GBP: { ... },

EUR: { ... }

}

* You can directly access specific data like currencyInfo.USD.

**Rules of JSON Format**

* **Keys**: Must be double-quoted strings.
* **Properties**: Comma-separated.
* **Strings**: Always in double quotes.
* **Numbers**: Use normal JS number format (5, 1.2, etc.).
* **Booleans**: true and false.
* **Null**: Represented as null.
* **Allowed types**:
  + **Primitive**: strings, numbers, booleans, null.
  + **Complex**: objects and arrays.

**What JSON Doesn’t Support**

* Functions are ignored during stringification.
* BigInt (e.g., 123n) causes errors: TypeError.

**JSON Examples**

* Simple object:  
  '{"color":"red"}'
* Nested object:  
  '{"color":"red", "nestedObject": { "color": "blue" }}'
* Array of strings:  
  '["one", "two", "three"]'
* Array of objects:  
  '[{ "color": "blue" }, { "color": "red" }]'

**🔄 Understanding JSON and JavaScript Objects**

* JSON (JavaScript Object Notation) is essentially a **string representation of an object**.
* It has a specific format but is fundamentally just a string.
* To **work effectively with JSON**, you typically **convert it to a JavaScript object**.

**📥 Converting JSON String to JavaScript Object**

* Use the **JSON.parse()** method to convert a JSON string into a JavaScript object.
* Example:

js

CopyEdit

const JSONstr = '{"greeting": "hello"}';

const aPlainObj = JSON.parse(JSONstr);

* After conversion, you can **manipulate the object** as usual:

js

CopyEdit

aPlainObj.greeting = "hi";

**📤 Converting JavaScript Object to JSON String**

* Use the **JSON.stringify()** method to convert a JavaScript object to a JSON string.
* Example:

js

CopyEdit

const data = { name: "John", age: 30 };

const jsonString = JSON.stringify(data);

* The result will include **double-quoted keys and values**, compliant with JSON format.

**⚠️ Important Limitations of JSON**

* JSON strings:
  + **Cannot include functions**.
  + **Cannot contain comments**.
* When stringifying an object:
  + Any **function properties will be omitted** in the resulting JSON string.

**🧰 Practical Applications**

* Parsing JSON is a **standard practice when working with APIs**.
* After parsing, you can **access object properties programmatically**.
* Mastering JSON.parse() and JSON.stringify() is essential for **efficient web development workflows**.

### **Module 4**

### **🚗 Historical Context and Innovation**

* **1886**: Karl Benz invents the first internal combustion engine-powered car.
* The combustion engine was soon used in cars, planes, motorbikes, and boats — beyond the inventor’s original vision.
* **Parallel to JavaScript**: Originally created for browsers, JavaScript has evolved to serve broader purposes.

**🌐 Evolution of JavaScript: From Browser to Server**

* JavaScript was **initially a front-end language**, running only in browsers.
* In **2009**, **Ryan Dahl** introduced **Node.js** by leveraging Google’s V8 JavaScript engine for server-side use.
* This breakthrough allowed JavaScript to become a **full-stack language**, used for both front-end and back-end development.

**⚙️ Introduction to Node.js**

* **Node.js** is a **standalone runtime environment** that enables JavaScript execution outside the browser.
* It can run:
  + In the **command line (terminal/shell)**
  + In **desktop apps**
  + On the **backend of web applications**
* Previously, backend development relied on languages like **PHP, Python, Ruby, C#, and Java**.

**📦 Understanding npm (Node Package Manager)**

* **npm** is the **package manager** that comes with Node.js.
* It manages **Node modules** or **npm packages** — reusable pieces of code shared by the community.
* There are over **11 million open-source packages** available via npm.

**💻 Using Node.js and npm Locally**

* Node.js and npm can be **installed** on your local machine (or may be pre-installed).
* Common commands:
  + node — Runs JavaScript code or files.
  + npm — Installs packages from the npm registry.

**🧰 Starting a Node Project**

* Begin by creating a folder for your project.
* Run npm init or npm install to set up the project.
* This generates a **package.json file** that:
  + Tracks all installed packages (dependencies).
  + Makes the project **easily portable and shareable**.

**🔁 Managing Dependencies**

* Installed packages are saved in the **node\_modules** folder.
* package.json includes metadata about:
  + The project
  + Installed packages
  + Version control
* Team members can **recreate the environment** by running npm install using the shared package.json.

**📘 Summary**

* **Node.js** transformed JavaScript into a **full-stack language**.
* **npm** provides access to millions of reusable modules.
* You learned:
  + The **history and purpose** of Node.js
  + How to **use Node and npm**
  + The **importance of package.json** for project consistency and collaboration

**1. Importance of Testing in JavaScript**

* Ensures the code works as intended before deployment.
* Provides confidence in the functionality and reliability of applications.

**2. Manual vs Automated Testing**

* **Manual Testing**: Involves using comments to document expectations.
  + Downside: Ambiguity and lack of structure.
* **Automated Testing**: Uses testing frameworks (e.g., Jest) to formally define and verify expectations.

**3. Advantages of Testing Frameworks**

* Eliminates ambiguity by replacing comments with executable tests.
* Acts as documentation and verification tool.
* Allows repeatable and automated validation of code behavior.

**4. Practical Example: concatStrings Function**

* Joins two strings and returns the result (e.g., "abc" + "def" → "abcdef").
* Incorrect behavior (e.g., 1 + 2 returning 3 instead of "12") can be identified and fixed through tests.

**5. Writing Tests Using Jest**

* expect() is used to define expected output.
* Tests verify actual output against expected results.
* E.g., expect(concatStrings("abc", "def")).toBe("abcdef").

**6. Benefits of Automated Tests**

* **Concise**: Minimal syntax required to define expectations.
* **Clear**: Explicit inputs and outputs improve understanding.
* **Repeatable**: Tests can be run multiple times with consistent results.

**7. Handling Failing Tests**

* Failed tests indicate unmet expectations.
* Guide developers to identify and correct issues in code logic.

**8. The Red-Green-Refactor Cycle**

* **Red**: Write a failing test (indicating what’s missing).
* **Green**: Implement code to pass the test.
* **Refactor**: Optimize code without altering behavior.
* Core principle of **Test-Driven Development (TDD)**.

**9. Advantages of Test-Driven Development (TDD)**

* Encourages cleaner, modular, and maintainable code.
* Helps prevent regression and improves code quality.
* Enhances collaboration through shared understanding of requirements.

**10. Final Summary**

* Testing validates the intended behavior of code.
* It is a critical part of development that ensures reliability, maintainability, and clarity in software projects.

**1. Understanding Software Testing**

* **Definition:** Software testing is the process of confirming that software behaves according to its defined requirements.
* **Purpose:** Ensures that the software is bug-free, maintains existing functionality, and meets stakeholder expectations.

**2. Stakeholder Perspectives on Testing**

* **UX Designers:** Ensure the interface looks and behaves as intended.
* **Project Managers:** Verify software integration within the overall system.
* **Software Engineers:** Confirm code meets requirements and doesn't introduce new bugs.

**3. Approaches to Software Testing**

* There are multiple ways to test software, depending on the project scope and objectives.
* The three most common types of testing are:
  + **End-to-End (E2E) Testing**
  + **Integration Testing**
  + **Unit Testing**

**4. End-to-End (E2E) Testing**

* **Definition:** Simulates real user behavior by testing the entire application flow.
* **Example:** Opening a browser, logging in, and adding items to a cart.
* **Tools:** WebdriverJS, Protractor, Cypress.
* **Pros:** Validates complete system behavior from the user’s perspective.
* **Cons:** Slowest to run and most expensive to set up.
* **Note:** Can be performed by non-developers as well.

**5. Integration Testing**

* **Definition:** Tests how different components or modules of an application work together.
* **Example Tools:** React Testing Library, Enzyme.
* **Position in Pyramid:** Middle layer—faster and cheaper than E2E, but slower and more complex than unit tests.

**6. Unit Testing**

* **Definition:** Tests individual functions or methods in isolation from the rest of the codebase.
* **Goal:** Verify that small, single units of code work as expected.
* **Benefits:** Fast, easy to write, low cost, and highly reliable.
* **Common Use:** JavaScript functions or methods.

**7. The Testing Pyramid**

* **Structure:**
  + **Base:** Unit Tests (fastest, cheapest).
  + **Middle:** Integration Tests (moderate speed and cost).
  + **Top:** E2E Tests (slowest, most expensive).
* **Purpose:** Encourages a balanced approach by combining different testing levels for maximum efficiency and coverage.

**8. Conclusion**

* Combining all three testing types ensures thorough validation of code functionality.
* Helps deliver stable, user-friendly, and high-quality software products.

**Introduction to JavaScript Testing**

* JavaScript lacks built-in testing capabilities.
* External libraries and frameworks are essential for writing tests in JavaScript.

**Popular JavaScript Testing Frameworks**

* Commonly used testing libraries include:
  + **Jasmine**
  + **Mocha**
  + **Karma**
  + **QUnit**
* **Jest** is a popular testing framework, particularly for React and other modern JavaScript frameworks (Babel, TypeScript, Node, Angular, Vue).

**Overview of Jest**

* Maintained by Meta and community contributors.
* Supports testing for various environments and technologies.
* Built-in features include:
  + Code coverage
  + Mocking
  + Snapshot testing

**Code Coverage in Jest**

* **Definition**: Measures the percentage of code executed by tests.
* **Example**: 80% code coverage means 20% of code is untested.
* **Benefit**: Helps identify untested areas, reducing chances of bugs.
* **Limitation**: Even 100% coverage doesn’t guarantee all expectations are tested.
* **Productivity Tip**: Higher code coverage often reduces time spent on writing future tests.

**Mocking with Jest**

* **Purpose**: Isolates code under test from external dependencies.
* **Use Case**: Enables front-end testing by simulating backend responses.
* **Team Benefit**: Prevents development bottlenecks by allowing parallel front-end/back-end work.
* **Efficiency**: Helps developers work independently and deploy features faster.
* **Tooling**: Jest provides built-in mocking without the need for extra libraries like Sinon.

**Snapshot Testing**

* **Definition**: Captures the current state of the DOM structure.
* **Usage**: Ensures no unexpected changes or regressions occur in the UI after updates.
* **Common Use Case**: Visual testing of React components.

**Advantages of Jest for JavaScript Testing**

* Easy setup and usage, especially for beginners.
* Built-in support for asynchronous code testing.
* Comprehensive tooling in a single framework (no need for multiple plugins).

**Conclusion**

* Jest is a powerful and user-friendly framework for JavaScript testing.
* With features like code coverage, mocking, and snapshot testing, it provides robust tools for developing stable and reliable applications.

**1. Project Setup for Testing**

* Create a JavaScript file (e.g., addFive.js) containing a basic function to test (addFive).
* Export the function using module.exports to enable importing it in test files.

**2. Environment Check**

* Confirm Node.js and NPM installation using:
  + node --version
  + npm --version
* Check for Jest installation using:
  + jest --version (returns error if not installed)

**3. Installing Jest Locally**

* Initialize a Node project with:
  + npm -y (auto-generates package.json)
* Install Jest as a project dependency:
  + npm install --save-dev jest
* Node modules and Jest entry will appear in package.json.

**4. Configuring Jest in Package.json**

* Update the scripts section in package.json:

json

CopyEdit

"scripts": {

"test": "jest"

}

* This setup allows running tests via npm run test.

**5. Creating a Test File**

* Create a test file using the naming convention filename.test.js (e.g., addFive.test.js).
* Import the target function using:

js

CopyEdit

const addFive = require('./addFive');

**6. Writing a Basic Test**

* Use Jest’s test() method:

js

CopyEdit

test("returns the number plus five", () => {

expect(addFive(10)).toBe(15);

});

* The string parameter describes the test and appears in the command line output.

**7. Running Tests**

* Execute tests using:
  + npm run test
* Jest will run the test file and display **pass/fail** results with corresponding test descriptions.

**8. Outcome**

* Successful test run confirms correct setup and validates that Jest works as intended.
* This forms the foundation for writing and managing unit tests in JavaScript projects.

**1. Purpose of Software Requirements**

* Every software is built on either formal or informal requirements.
* Requirements describe what the software should do in human language.
* TDD is a methodology directly influenced by how these requirements are handled.

**2. Introduction to Test-Driven Development (TDD)**

* TDD stands for **Test-Driven Development** or RTDD (Red-Green-Refactor).
* It is a coding approach where tests are written before the actual implementation.
* Helps ensure that the code meets the specified requirements.

**3. TDD Workflow: High-Level Process**

* Receive a requirement for a new feature.
* **Write a failing test** for the feature before implementing it.
* **Implement code** to make the test pass.
* Refactor the code for optimization and maintainability.
* The approach is summarized as **Red (Fail) → Green (Pass) → Refactor**.

**4. TDD Illustrated Through Real-Life Analogy**

* Scenario: Forgetting car keys when heading to work.
* Traditional method: Check for keys after reaching the car (reactive).
* TDD method: Test for keys before leaving home → fails → retrieve keys → test again → pass.

**5. Practical Example of TDD with Code**

* Start by writing a test for a function (e.g., statusOfKeys) that hasn’t been implemented yet.
* Run the test → it fails.
* Write minimal code (declare the function) to pass the test.
* Add more functionality based on new requirements (e.g., accept a keys variable and log it).
* Each new feature follows the fail → implement → pass cycle.

**6. Code Maintenance and Refactoring**

* After passing tests, clean up unnecessary code and correct formatting.
* Re-run tests to confirm that the refactoring hasn't broken functionality.

**7. Full TDD Cycle Recap**

* Read the requirement.
* Write a failing test (**Red**).
* Write code to pass the test (**Green**).
* Refactor the code for better structure (**Refactor**).
* Repeat the cycle for every new requirement.

**8. Benefits of TDD**

* Enables **automated testing** across all platforms.
* Prevents **regressions** (accidental bugs in previously working code).
* Provides **proof of stability** for both new and old code.
* Facilitates easier **onboarding** of new developers with test-based documentation.
* Ensures **robust and reliable** software development.