SEMESTER 1: HTML, CSS AND JAVASCRIPT

SEMESTER 1: HTML, CSS AND JAVASCRIPT

Month 1, Week 1: Introduction to AltSchool and Onboarding

🔹 Overview

The first week at AltSchool focused on familiarizing students with the platform, support systems, and academic expectations. This onboarding process ensured smooth navigation of learning resources and prepared students for success in the School of Engineering.

1 Introduction to AltSchool

AltSchool Africa is a structured, career-focused learning platform that equips students with industry-relevant skills in engineering and technology. During onboarding, students explored the learning model, available resources, and the journey ahead.

2 Accessing Content on the LMS

Students were guided on how to use the Learning Management System (LMS), which serves as the central hub for coursework, assignments, and discussions.

Key Features:

- **Course Materials** – Structured modules and lessons
- **Progress Tracking** – Monitoring course completion status
- **Discussion Forums** – Interaction with peers and instructors

3 Learning Circles & Community Engagement

AltSchool encourages collaboration through Learning Circles, which enable peer discussions, problem-solving, and shared learning experiences.

Community Benefits:

- Networking with fellow students and mentors
- Access to group study sessions
- Support in grasping challenging concepts

4 Communication & Support Services

Understanding how to reach AltSchool for guidance was emphasized. Students were introduced to:

- **Email & Help Desk** – For inquiries and troubleshooting
- **Live Chat & Forums** – Quick resolutions and peer support
- **Mentorship & Career Guidance** – Personalized professional assistance

5 Live Classes & Interactive Learning

AltSchool integrates live classes where students engage with instructors and classmates in real time.

Live Class Features:

- Scheduled sessions with expert instructors
- Interactive Q&A and discussions
- Recorded lectures for review

6 AltSchool Terms of Use & Payments

Students were informed about AltSchool policies, including:

- Ethical learning practices and platform conduct
- Payment structure & installment options for tuition
- Refund policies and financial assistance opportunities

7 Choosing a Career Path

Selecting the right engineering track is crucial for long-term success. Students explored various paths within software development, such as:

- **Frontend Engineering** – UI/UX design & interaction
- **Backend Engineering** – Server-side logic & databases
- **Cybersecurity** – Protecting Digital Assets
- **Cloud Computing** – Scalable Web Applications

8 Expected Outcomes After Course Completion

AltSchool provides students with practical skills, industry connections, and certifications to increase employability. Upon completing the program, students will:

- Possess hands-on experience in engineering projects
- Be equipped for real-world problem-solving in tech
- Receive guidance for job placements & career growth

Conclusion

Week 1 provided a solid foundation in navigating AltSchool's learning platform, communication channels, and structured curriculum. This onboarding phase ensured students understand their career direction, learning expectations, and community resources—paving the way for a productive and impactful learning experience.

(Month 1, Week 2) Introduction to Computer programming

1. Introduction to Programming

Programming is the practice of writing instructions (code) in a language that computers can interpret and execute. These instructions are designed to solve problems or perform tasks. Programming languages such as Python, Java, C++, and JavaScript are commonly used for this purpose.

- **Code** refers to the actual instructions written by a programmer.
- A **Program** is a complete set of instructions designed to perform a specific task.
- **Compilers** and **interpreters** are tools used to translate human-readable code into machine-readable code.

Example:

```
console.log("Hello, world!");
```

Displays "Hello, world!" on the screen.

2. Compilers vs. Interpreters

- **Compiler**: Translates entire code before execution. It detects all errors at once.
- **Interpreter**: Translates code line by line and stops at the first error.

Feature	Compiler	Interpreter
Translation	Whole program at once	One line at a time
Execution Speed	Faster	Slower
Error Handling	After compilation	On first error
Examples	C, C++	Python, JavaScript

Example in C (Compiler):

```
int c = a / b; // Error if b = 0
```

Example in JavaScript (Interpreter):

```
let c = a / b; // Returns Infinity if b = 0
```

3. Bits, Bytes, Variables, Data Types, and Type Checking

- **Bit**: Smallest data unit (0 or 1).
- **Byte**: 8 bits. Used to represent characters like 'A'.

Variables:

Used to store data.

```
let name = "Alice";
let age = 25;
```

Data Types in JavaScript:

Type	***Example***	***Description***
Number	`42, 3.14`	Integer/Decimal numbers
String	"Hello"	Text
Boolean	`true, false`	Logical values
Object	`{key: value}`	Key-value structure
Array	`[1,2,3]`	List of values
Null	`null`	No value
Undefined	`undefined`	Not assigned

Type Checking:

```
typeof 25; // "number"
typeof "Bob"; // "string"
```

4. Data Structures, Control Flow, Loops, and Recursion

Data Structures:

- **Array**: `let arr = [1, 2, 3];`
- **Object**: `let obj = {name: "Alice"};`
- **Set**: `new Set([1,2,3]);`
- **Map**: `new Map();`

Control Flow:

```
if (age >= 18) console.log("Adult");
```

Loops:

- For Loop:

```
for (let i = 0; i < 3; i++) {}
```

- While Loop:

```
while (condition) {}
```

- For…of Loop:

```
for (let val of array) {}
```

Recursion:

```
function factorial(n) {
  if (n === 0) return 1;
```

```
  return n * factorial(n - 1);
}
```

(Month 1, Week 3-4): HTML & Web Fundamentals

Overview

During Weeks 3 and 4, we were introduced to **HTML (Hypertext Markup Language)** and the foundational concepts of how the **World Wide Web** works. Emphasis was placed on creating basic web pages, understanding the client-server model, HTML structure, tags, elements, formatting, media, and semantic layout.

1 Understanding the Web

Internet vs. World Wide Web

- **Internet**: Infrastructure for global communication (e.g., email, file transfer).
- **WWW**: Hyperlinked documents accessed via browsers, built on top of the Internet.

Client-Server Architecture

- **Client**: Web browsers that send requests.
- **Server**: Hosts and serves content on request.

HTTP Protocol & Status Codes

- **GET**: Retrieve resources.
- **POST**: Send data to server.
- **Common Status Codes**:
  - `200 OK`: Success
  - `404`: Not Found
  - `500`: Server Error

2 Core Web Technologies

Technology	Role	Description
HTML	Structure	Defines layout and content
CSS	Style	Controls visual appearance
JavaScript	Behavior	Adds interactivity and dynamic behavior

3 HTML Fundamentals

What is HTML?

- **Hypertext Markup Language**
- Not a programming language
- Uses tags to define content
- HTML5 is the current version

Basic HTML Document Structure:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <!-- Content here -->
  </body>
</html>
```

Key Sections:

- `<!DOCTYPE html>`: Tells browser to use HTML5
- `<head>`: Metadata (title, scripts, styles)
- `<body>`: Visible content (headings, paragraphs, images, etc.)

4 HTML Syntax & Best Practices

- Tags wrap content:

  ```
  <p>This is a paragraph.</p>
  ```

- Self-closing for empty elements:

  ```
  <img src="pic.jpg" alt="A picture" />
  ```

- Proper **nesting** is required:

  ```
  <p>This is <strong>important</strong>.</p>
  ```

5 Common HTML Elements

Text Elements:

- Headings: `<h1>` to `<h6>` (use one `<h1>` per page)
- Paragraphs: `<p>`, line breaks: `<br />`
- Formatting: `<strong>`, `<em>`, `<b>`, `<i>`

Lists:

- Unordered: `<ul>` with `<li>`
- Ordered: `<ol>` with `<li>`

- Nested lists supported

Links:

```
<a href="https://example.com" target="_blank">Visit Site</a>
<a href="mailto:someone@example.com">Email Me</a>
```

Images:

```
<img src="image.jpg" alt="Description" width="500" height="300" />
```

- Important to include `alt` text for accessibility

Tables:

```
<table>
  <tr><th>Name</th><th>Age</th></tr>
  <tr><td>Alice</td><td>22</td></tr>
</table>
```

6 Semantic HTML5 Layout

Tag	Meaning
`<header>`	Top of page/section
`<nav>`	Navigation links
`<main>`	Main content
`<article>`	Independent content
`<aside>`	Sidebar info
`<footer>`	Bottom info

These enhance SEO and accessibility.

7 HTML Attributes

Attribute	Use
`id`	Unique identifier for one element
`class`	Assigns styling/grouping
`style`	Inline CSS (discouraged for large apps)
`title`	Tooltip info on hover

8 Developer Tools & Validation

- **Browser Dev Tools**: Inspect, edit, debug HTML/CSS
- **Validation**: Use W3C Validator to ensure standards-compliance

Practical Assignment

Create a Personal Web Page with:

- Name in a heading

- A paragraph about yourself
- A list of hobbies/interests
- A hyperlink to your favorite website
- An image with alt text
- Valid and well-formatted HTML

Next Steps

- Begin learning **CSS** to style web pages
- Study **HTML forms**
- Learn **JavaScript** for interactivity
- Continue practicing through mini-projects

Resources

- MDN Web Docs
- W3Schools
- Free Editors: VS Code, Atom, Sublime

Git & GitHub Summary

Version Control Basics

- Version control helps track file changes, collaborate, and revert changes.
- Git is **distributed**, **fast**, supports **branching**, and is **open-source**.
- Git vs SVN/Mercurial: Git is more flexible, distributed, and widely adopted.

Installing & Configuring Git

- **Install** via `git-scm.com`, Homebrew (`macOS`), or `apt` (`Linux`).
- Configure identity:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Basic Git Workflow

```
mkdir my-project && cd my-project
git init
echo "Hello Git" > hello.txt
git add hello.txt
git commit -m "Initial commit"
```

Branching & Merging

```
git branch feature
git switch feature
# After changes
git switch main
git merge feature
```

GitHub Integration

- GitHub hosts Git repositories with tools for collaboration.
- **Clone repo**:

```
git clone https://github.com/user/repo.git
```

- **Push to GitHub**:

```
git remote add origin https://github.com/user/repo.git
git push -u origin main
```

Team Workflow

- **Fork → Clone → Create Branch → Pull Request → Review → Merge**

Merge Conflicts

- Occur when simultaneous edits affect the same file lines.
- Resolve manually, then:

```
git add conflicted-file.txt
git commit
```

Hands-On Activities

- Students practice: repo creation, cloning, committing, pushing, and submitting pull requests.

(Month 2, Week 1): Basic CSS Report

During **Week 1 of Month 2**, the focus was on understanding and applying **Cascading Style Sheets (CSS)** to enhance the presentation layer of web pages. CSS allows for separation of content and design, making websites cleaner, more scalable, and easier to maintain.

1. Introduction to CSS

CSS (Cascading Style Sheets) defines how HTML elements are displayed on screen. It enhances user experience by adding layout control, colors, fonts, and visual effects. Key benefits discussed included: * Applying consistent design across web pages * Creating responsive designs for different devices * Reducing HTML clutter by offloading style rules to external files

2. CSS Syntax and Units

The basic CSS syntax uses selectors followed by a declaration block of property-value pairs. Example:

```
p {
  color: blue;
  font-size: 16px;
}
```

Units introduced included:

- `px`, `%`: Absolute and relative sizing
- `em`, `rem`: Relative to font sizes
- `vh`, `vw`: Relative to viewport dimensions

3. CSS Selectors

Students explored multiple selector types: * **Element selectors**: Target tags like `h1`, `p` * **Class selectors**: Use `.` to style grouped elements * **ID selectors**: Use `#` for unique elements * **Universal, group, and descendant selectors** for more advanced targeting * **Pseudo-classes/elements** for interaction-based or structural styling (`:hover`, `::first-line`)

4. Methods of Including CSS

Three ways to add CSS were covered: * **Inline** (e.g., `<p style="color:red">`) * **Internal** (via `<style>` tag in HTML head) * **External** (via linked `.css` file using `<link>`)

5. Best Practices and Tips

Best practices included:

- Using classes for reusability
- Avoiding inline styles
- Grouping related styles
- Using comments and consistent formatting
- Testing styles across browsers

6. CSS Variables

Students learned how to use custom properties:

```
:root {
  --main-color: #3498db;
}
```

This allowed centralized color or value definitions for easier maintenance and theme adjustments.

7. Glossary of Common CSS Properties

A detailed glossary of common CSS properties was introduced, covering: * Text styling (`color`, `font-size`, `font-family`) * Box model (`margin`, `padding`, `border`) * Layout (`display`, `position`, `flex`, `grid`) * Effects (`box-shadow`, `transition`) * Responsive tools (`width`, `overflow`, `z-index`, alignment)

Summary

This week laid a strong foundation in **web design styling with CSS**, enabling students to visually structure HTML elements, implement clean and modular designs, and build user-friendly web interfaces. The CSS knowledge acquired set the stage for deeper topics like responsive design, Flexbox, and Grid in the coming sessions.

(Month 3, Week 1): CSS Layout (Flexbox and Grid)

During Week 2 of the programming module, we explored **CSS layout systems**, focusing on **Flexbox** and **Grid**, which are essential for building responsive and well-structured web designs.

1. Flexbox Layout (1-Dimensional)

Purpose: Flexbox is used for one-dimensional layouts—either in a **row** or **column** direction.

Key Features:

- Use `display: flex;` on the container.
- Control layout with properties like:
  - `flex-direction`
  - `justify-content`
  - `align-items`
  - `flex-wrap`
  - `gap`

Example Code:

```
.flex-container {
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: center;
  gap: 10px;
}
```

Use Cases:

- Navigation bars
- Card layouts
- Centering content vertically or horizontally

2. Grid Layout (2-Dimensional)

Purpose: CSS Grid allows layouts in both **rows and columns**, ideal for more complex designs.

Key Features:

- Use `display: grid;` on the container.
- Define structure with:
  - `grid-template-columns`
  - `grid-template-rows`
  - `grid-area`
  - `gap`, `place-items`

Example Code:

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(2, 1fr);
  grid-template-rows: auto auto;
  gap: 10px;
}
```

Use Cases:

- Full page layouts
- Photo galleries
- Dashboards

Flexbox vs Grid Comparison Table

Feature	Flexbox	Grid
Layout Type	1D (row/column)	2D (row + column)
Best Use Case	Components	Entire page layouts
Item Placement	In source order	Placed using grid lines
Complexity	Simpler	More powerful, more complex

Combined Example

Using both Grid and Flexbox can provide powerful layout control. A layout can use Grid for structural layout and Flexbox for component alignment within sections.

Summary

- Use **Flexbox** for simpler, one-dimensional component layouts.
- Use **Grid** for two-dimensional, structured page layouts.
- Both systems improve responsiveness and reduce layout complexity compared to older techniques.

(Month 2, Week 3-4) CSS responsiveness and animations

Month 1, Weeks 3–4: More CSS Techniques Course: Frontend Web Development Topic: Responsive Design & Animations

1. Media Queries

Overview: Media queries allow web developers to build **responsive designs** by applying CSS conditionally based on screen/device characteristics such as width, resolution, and orientation.

Key Concepts: * **Syntax:** `css @media media-type and (condition) { /* CSS rules */ }` * **Use Cases:** * Making websites adapt to mobile, tablet, and desktop * Changing font sizes, layout direction * Supporting dark/light themes

Examples:

```
/* Mobile default */
body { background-color: lightblue; }
/* Tablet */
@media (min-width: 768px) {
  body { background-color: lightgreen; }
}
/* Desktop */
@media (min-width: 1024px) {
  body { background-color: lightcoral; }
}
```

Common Media Features: | Feature | Use Case Example | | ——————————- | ——————————————— | | `min-width` | `(min-width: 768px)` – tablets/desktops | | `max-width` | `(max-width: 600px)` – mobile-only | | `orientation` | `(orientation: portrait)` | | `prefers-color-scheme` | `(prefers-color-scheme: dark)` | | `resolution` | `(min-resolution: 2dppx)` |

Pro Tip: You can combine multiple queries:

```
@media (min-width: 768px) and (orientation: landscape) { ... }
```

2. CSS Animations

A. CSS Transitions

Used for smooth effects like color changes on hover or focus.

Example:

```
.button {
  transition: background-color 0.3s ease;
}
.button:hover {
```

```
  background-color: darkblue;
}
```

Properties:

- transition-property
- transition-duration
- transition-timing-function
- transition-delay

B. CSS Keyframe Animations

Used for complex or continuous animations like slide-ins, pulses, and loaders. **Key Example: Slide-In Animation**

```
@keyframes slide-in {
  from { transform: translateX(-100%); opacity: 0; }
  to { transform: translateX(0); opacity: 1; }
}
.box {
  animation: slide-in 1s ease-out;
}
```

Animation Properties Table:

Property	Function
animation-name	Name of the animation (@keyframes)
animation-duration	How long the animation lasts
animation-timing-function	Animation speed curve (ease, linear)
animation-delay	Wait time before starting
animation-iteration-count	How many times it repeats
animation-direction	Forward, reverse, alternate
animation-fill-mode	Persist styles before/after animation

Pulsing Button Example:

```
@keyframes pulse {
  0%, 100% { transform: scale(1); }
  50% { transform: scale(1.1); }
}
.pulse-button {
  animation: pulse 1.5s infinite ease-in-out;
}
```

Real-World Applications

Task	Media Queries Needed?	Animation Needed?
Mobile-friendly layout	✅ Yes	❌ No
Dark/light mode support	✅ Yes	❌ No
Button hover effects	Optional	✅ Yes
Loading spinner	❌ No	✅ Yes
Slide-in side menu	Optional	✅ Yes

Key Takeaways

- Use media queries to build **responsive** websites for all screen sizes.
- Use transitions and keyframe animations to enhance **user experience** with visual feedback.
- Combine both techniques for professional, adaptive, and interactive web design.

(Month 3, Week 1)JavaScript Fundamentals – Summary Report

1. Primitive Data Types

JavaScript offers **seven primitive types**, each representing a single value:

- **String** – for textual data: `"John"`
- **Number** – for integers and floating-point: `5`, `3.14`
- **Boolean** – true/false values: `true`, `false`
- **Undefined** – declared but not assigned: `let x;`
- **Null** – intentional absence of value: `let x = null;`
- **Symbol** – for unique identifiers
- **BigInt** – handles very large integers: `123456789123456789n`

2. Non-Primitive Types

- **Object** – collections of key-value pairs
- **Array** – ordered list-like structures
- **Function** – reusable blocks of code

3. Type Checking & Conversion

- `typeof` checks the type of a variable
- Implicit conversions (e.g., `"3" + 2` → `"32"`)
- Explicit conversions using `Number()`, `String()`, `Boolean()`

4. JavaScript Operators

- **Arithmetic**: `+`, `-`, `*`, `/`, `%`, `**`, `++`, `--`
- **Assignment**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- **Comparison**: `==`, `===`, `!=`, `>`, `<`, etc.

- **Logical**: `&&`, `||`, `!`
- **String Operators**: string concatenation with `+`
- **Type Operators**: `typeof`, `instanceof`

5. Conditional Statements

- `if`, `if...else`, `if...else if...else`
- **Ternary Operator**: `condition ? trueExpr : falseExpr`
- `switch` for multiple condition checks

6. Loops

- **for** – for known number of iterations
- **while** – continues while a condition is true
- **do…while** – runs at least once
- **for…of** – iterates over array values
- **for…in** – iterates over object keys

Loop Controls:

- `break` – exits loop early
- `continue` – skips current iteration

7. Functions

- **Function Declaration**: `function name() {}`
- **Function Expression**: `const name = function() {}`
- **Arrow Functions**: `const name = () => {}`
- Parameters vs. Arguments
- `return` statement returns values from functions
- **Default Parameters**: provide fallback values
- **Anonymous & Callback Functions**

8. Scope

Defines the accessibility of variables:

- **Global Scope** – accessible everywhere
- **Function Scope** – accessible only inside a function
- **Block Scope** – with `let` and `const` inside `{}`

Avoid using `var` due to its function-only scoping.

9. Closures

A **closure** is when a function retains access to its **outer scope** even after the outer function has finished executing.

Examples:

- Counter that remembers previous values

- Factory functions (`makeMultiplier`)
- Creating private variables

Best Practices:

- Prefer `===` over `==`
- Use `const` and `let`, avoid `var`
- Avoid global variables where possible
- Modularize code using functions
- Use closures to manage state and privacy

(Month 3, Week 2) Data structures, DOM and Events.

13. Browser Object Model (BOM) and DOM Interaction

JavaScript can interact not just with a webpage's content (DOM) but also with the browser environment itself using the **Browser Object Model (BOM)**.

What is BOM?

The **BOM** is a set of browser-provided objects allowing interaction beyond the page, including navigation, window control, alerts, and history.

Object	Description
`window`	The global object representing the browser window. All BOM and DOM objects are children of `window`.
`navigator`	Contains information about the browser (e.g., user agent, online status).
`location`	Provides information about the current URL and allows redirection.
`history`	Allows navigation back and forth through the user's browser history.
`screen`	Contains information about the user's screen (e.g., width, height).
`alert()`, `confirm()`, `prompt()`	Methods provided by `window` for user interaction.

Example:

```
alert("Hello!");
console.log(location.href); // Prints the current URL
```

14. Arrays in JavaScript

An **array** is a list-like data structure that stores multiple values in a single variable.

Declaration:

```
let fruits = ["Apple", "Banana", "Cherry"];
```

Accessing Elements:

```
console.log(fruits[0]); // Apple
```

Looping:

```
fruits.forEach(fruit => console.log(fruit));
```

Useful Array Methods : Modifying Arrays

Method	Description	Example
`push()`	Add item to end	`fruits.push("Kiwi")`
`pop()`	Remove last item	`fruits.pop()`
`shift()`	Remove first item	`fruits.shift()`
`unshift()`	Add item to start	`fruits.unshift("Lemon")`
`indexOf()`	Find index of an item	`fruits.indexOf("Banana")`
`includes()`	Check if an item exists	`fruits.includes("Apple")`
`join()`	Join all elements into a string	`fruits.join(", ")`
`slice()`	Get a portion of the array	`fruits.slice(1, 3)`
`splice()`	Add/Remove items from any position	`fruits.splice(1, 1, "Orange")`
`map()`	Create new array from each item	`numbers.map(x => x * 2)`
`filter()`	Filter items that match a condition	`numbers.filter(x => x > 2)`

Example:

```
let numbers = [1, 2, 3, 4, 5];
let evens = numbers.filter(n => n % 2 === 0); // [2, 4]
let squares = numbers.map(n => n * n);        // [1, 4, 9, 16, 25]
```

15. Sets in JavaScript

A **Set** is a collection of unique values.

Creating a Set:

```
let numbers = new Set([1, 2, 3, 3, 4]); // Set(4) {1, 2, 3, 4}
```

Common Set Methods

Method	Description	Example
`add(value)`	Adds a value to the set	`mySet.add(5)`
`delete(value)`	Removes a value from the set	`mySet.delete(3)`

Method	Description	Example
`has(value)`	Checks if a value exists	`mySet.has(2)`
`clear()`	Removes all values	`mySet.clear()`
`size`	Returns number of elements	`mySet.size`

Iteration:

```
for (let val of numbers) {
  console.log(val);
}
```

Remove Duplicates:

```
let nums = [1, 2, 2, 3];
let uniqueNums = [...new Set(nums)]; // [1, 2, 3]
```

When to Use Sets

- When you need a collection of **unique** values.
- When checking for existence (`has()`) needs to be **fast**.
- When you're removing duplicates from arrays.

Limitations of Sets

- No access by index like arrays (`mySet[0]` doesn't work).
- No direct `map()` or `filter()`—you need to convert to an array first:

```
let filtered = [...mySet].filter((x) => x > 2);
```

1. Array Destructuring

Array destructuring allows you to extract values from an array and assign them to variables in a concise syntax.

- **Basic Syntax:**

  ```
  const [a, b] = [10, 20];
  console.log(a); // 10
  console.log(b); // 20
  ```

- **Skipping Items:** You can skip unwanted elements:

  ```
  const [first, , third] = [1, 2, 3];
  console.log(third); // 3
  ```

- **Swapping Variables:** Destructuring simplifies variable swapping:

  ```
  let x = 1, y = 2;
  [x, y] = [y, x];
  console.log(x); // 2
  ```

2. Object Destructuring

Object destructuring extracts multiple properties from an object in a single line.

- **Basic Syntax:**

```
const person = { name: "Alice", age: 30 };
const { name, age } = person;
```

- **Renaming Variables:**

```
const { name: fullName } = person;
console.log(fullName); // Alice
```

- **Default Values:**

```
const { city = "Unknown" } = person;
console.log(city); // Unknown
```

3. Practical Use Cases of Destructuring

- **Function Parameters:**

```
function greet({ name, age }) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}
greet({ name: "Bob", age: 25 });
```

- **Fetching API Data:**

```
fetch("/api/user")
  .then(res => res.json())
  .then(({ id, email }) => {
    console.log(id, email);
  });
```

- **Returning Multiple Values:**

```
function getStats() {
  return [95, 88];
}
const [math, science] = getStats();
```

- **Looping with `Object.entries()`:**

```
const scores = { Alice: 90, Bob: 85 };
for (const [name, score] of Object.entries(scores)) {
  console.log(`${name}: ${score}`);
}
```

4. DOM – Document Object Model

The DOM represents the structure of a web page as a tree, allowing JavaScript to manipulate HTML elements dynamically.

- **Accessing Elements:**

```
document.getElementById("id")
document.querySelector(".class")
```

- **Modifying Elements:**

```
element.textContent = "New text";
element.style.color = "blue";
```

- **Creating Elements:**

```
const div = document.createElement("div");
document.body.appendChild(div);
```

- **Example – To-Do List:**
  - A simple UI lets users input tasks.
  - JavaScript dynamically adds list items and delete buttons using DOM manipulation.
  - Also supports pressing **Enter** to add a task.

5. BOM – Browser Object Model

The BOM gives JavaScript access to browser-specific features beyond the page content.

- **Key Objects:**

Object	Use
`window`	Global scope, alert/confirm/prompt
`navigator`	Browser metadata
`location`	URL manipulation
`history`	Back/forward navigation
`screen`	Screen resolution info

- **Examples:**

```
alert("Hello!");
console.log(location.href);
```

6. DOM Events and Event Handling

Events are user or browser actions that can be detected and responded to using JavaScript.

- **Common Events:** `click`, `mouseover`, `keydown`, `submit`, `change`, etc.

- **Event Listeners:**

```
button.addEventListener("click", () => alert("Clicked!"));
```

- **Event Object:** Provides metadata like:
  - `event.target`
  - `event.type`
  - `event.preventDefault()`
  - `event.stopPropagation()`
- **Propagation Phases:**
  - **Capturing Phase**
  - **Target Phase**
  - **Bubbling Phase** (default)

7. Event Delegation

Event delegation leverages event bubbling by placing a single event listener on a parent element to handle actions from many children.

- **Why it's useful:**
  - Fewer event listeners.
  - Handles dynamically added elements.
  - Improves performance and scalability.
- **Example:**

```
document.getElementById("menu").addEventListener("click", (event) => {
  if (event.target.tagName === "LI") {
    alert(`Clicked: ${event.target.getAttribute("data-action")}`);
  }
});
```

Conclusion

This section highlights JavaScript's powerful capabilities for interacting with both data (via destructuring) and the user interface (via DOM, BOM, and events). These concepts form the backbone of modern web development, empowering developers to create rich, interactive experiences that respond to user behavior and adapt dynamically.

(Month 3, Week 3)Asynchronous JavaScript: Callbacks, Promises, and async/await

Modern JavaScript relies heavily on **asynchronous programming** to handle tasks like API requests, file I/O, and time-based actions. The three primary approaches are:

Callbacks

A **callback** is a function passed into another function to be executed after the parent function completes.

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

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

greet("Alice", sayGoodbye);
// Output:
// Hello Alice
// Goodbye!
```

Callbacks are also common in asynchronous functions:

```
setTimeout(function () {
  console.log("This runs after 2 seconds");
}, 2000);

console.log("This runs first");
```

Pros:

- Easy to implement for small tasks ❌ **Cons**:
- **Callback Hell** when callbacks are deeply nested
- Hard to read and maintain

```
doTask1(function (result1) {
  doTask2(result1, function (result2) {
    doTask3(result2, function (result3) {
      console.log(result3);
    });
  });
});
```

Promises

A **Promise** represents a value that might be available now, later, or never. Promises use `.then()` and `.catch()` to handle successful results and errors.

```
fetch("https://api.github.com/users/octocat")
  .then((res) => res.json())
  .then((user) => fetch(user.repos_url))
  .then((res) => res.json())
```

```
  .then((repos) => console.log(repos))
  .catch((err) => console.error(err));
```

Pros:

- Chainable, readable
- Better error handling than callbacks

async/await

`async/await` is a cleaner syntax for writing asynchronous code based on promises.

```
async function fetchUser() {
  try {
    const res = await fetch("https://api.github.com/users/octocat");
    const user = await res.json();
    console.log(user);
  } catch (err) {
    console.error("Error:", err);
  }
}
```

Pros:

- Synchronous-like flow
- Easy debugging with `try...catch`
- Clean and modern syntax

Callback Hell Example: GitHub API Chain

```
getUser("octocat", (err, user) => {
  if (err) return console.error(err);

  getRepos(user, (err, repos) => {
    if (err) return console.error(err);

    const firstRepo = repos[0];
    getIssues(firstRepo, (err, issues) => {
      if (err) return console.error(err);

      const firstIssue = issues[0];
      getComments(firstIssue, (err, comments) => {
        if (err) return console.error(err);

        console.log("Comments on the first issue:", comments);
      });
    });
  });
});
```

The same flow with `async/await`:

```
async function fetchGitHubData() {
  try {
    const userRes = await fetch("https://api.github.com/users/octocat");
    const user = await userRes.json();

    const reposRes = await fetch(user.repos_url);
    const repos = await reposRes.json();

    const issuesRes = await fetch(
      `https://api.github.com/repos/${repos[0].full_name}/issues`
    );
    const issues = await issuesRes.json();

    const commentsRes = await fetch(issues[0].comments_url);
    const comments = await commentsRes.json();

    console.log("Comments on the first issue:", comments);
  } catch (err) {
    console.error("Something went wrong:", err);
  }
}
```

Summary Table

Feature	Callbacks	Promises	async/await
Readability	Poor (nested)	Better (chained)	Best (sequential, clean)
Error Handling	Manual `if (err)`	`.catch()`	`try...catch`
Debugging	Hard	Easier	Easiest
Use Case	Small async tasks	Most async operations	Modern, recommended for most cases

(Month 3, Week 4)(Object-Oriented Programming in JavaScript)

Object-Oriented Programming (OOP) in JavaScript is a programming paradigm based on the concept of "**objects**", which can contain data (***properties)*** and code (***methods***). JavaScript supports OOP through **prototypes** and **classes *(introduced in ES6),*** allowing you to create reusable and modular code.

🔹 Core OOP Concepts in JavaScript

1. **Classes and Objects**
2. **Encapsulation**
3. **Inheritance**

4. **Polymorphism**
5. **Abstraction**

1. Classes and Objects

In JavaScript, classes are syntactic sugar over the existing prototype-based inheritance.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  }
}

const person1 = new Person("Alice", 30);
person1.greet(); // Hi, I'm Alice and I'm 30 years old.
```

2. Encapsulation

Encapsulation means hiding internal details. In JS, you can simulate this using closures or the newer # syntax for private fields.

```
class BankAccount {
  #balance = 0; // Private field

  constructor(owner) {
    this.owner = owner;
  }

  deposit(amount) {
    if (amount > 0) this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount("Bob");
account.deposit(1000);
console.log(account.getBalance()); // 1000
// console.log(account.#balance); // Error: Private field
```

3. Inheritance

Inheritance allows a class to inherit from another class.

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

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

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

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

4. Polymorphism

Polymorphism allows different classes to be treated as instances of the same parent class, often overriding methods.

```
function makeAnimalSpeak(animal) {
  animal.speak();
}

const dog = new Dog("Buddy");
const animal = new Animal("Generic");

makeAnimalSpeak(dog); // Buddy barks.
makeAnimalSpeak(animal); // Generic makes a sound.
```

5. Abstraction

Abstraction means hiding complex implementation and showing only the necessary details.

While JS does not have abstract classes in the same way as some other languages, you can use base classes or interfaces (with TypeScript) to simulate it.

```
class Vehicle {
  startEngine() {
    throw new Error("Method 'startEngine()' must be implemented.");
  }
}

class Car extends Vehicle {
  startEngine() {
    console.log("Car engine started.");
```

```
  }
}

const car = new Car();
car.startEngine(); // Car engine started.
```

Summary Table

Concept	JavaScript Example
Class & Object	`class Person { ... }`
Encapsulation	Private fields `#balance`
Inheritance	`class Dog extends Animal { ... }`
Polymorphism	Method overriding (`speak()` in `Dog`)
Abstraction	Throwing errors in base methods

In JavaScript, the `constructor` is a **special method** used within a class to create and initialize objects created from that class.

🔹 What is a Constructor?

- It's a method with the name `constructor`.
- It is automatically called when you create a new instance of a class using the `new` keyword.
- It sets up **initial values** for object properties.

🔹 Syntax

```
class ClassName {
  constructor(parameters) {
    // initialization code
  }
}
```

🔹 Example

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
}

const p1 = new Person("Alice", 25);
p1.greet(); // Hi, I'm Alice
```

- Here, `new Person("Alice", 25)` calls the constructor with `"Alice"` and `25` as arguments.
- `this.name` and `this.age` set the object's properties.

🔹 Points to Remember

1. **One constructor per class**: You can't define multiple constructors in a class. However, you can make parameters optional or use default values.

```
constructor(name = "Guest") {
  this.name = name;
}
```

2. **Constructor is optional**: If you don't define one, JavaScript adds a default constructor automatically:

```
constructor() {}
```

3. **In inheritance**, if a class extends another class, the `super()` function must be called inside the constructor before using `this`.

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

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Calls Animal's constructor
    this.breed = breed;
  }
}
```

🔹 Summary

Feature	Description
Purpose	Initialize new objects
Syntax	`constructor(parameters) { ... }`
Called by	Automatically by `new ClassName()`
Inheritance	Must call `super()` before `this`

5. Big O Notation and Programming Paradigms

Big O Notation:

Describes algorithm efficiency.

Notation	Meaning	Example
O(1)	Constant	`arr[0]`

Notation	Meaning	Example
O(log n)	Logarithmic	Binary search
O(n)	Linear	Loop over array
O(n log n)	Linearithmic	Merge sort
O(n^2)	Quadratic	Nested loops
O(2^n)	Exponential	Recursive subsets

Programming Paradigms:

Paradigm	Description	Example
Imperative	Step-by-step	`for` loop
Declarative	Focus on what, not how	`arr.map(x => x * 2)`
Procedural	Organized with functions	`function greet() {...}`
Object-Oriented	Objects with methods and properties	`{ start() { console.log("Go") } }`
Functional	Pure functions, no side-effects	`const double = x => x * 2;`