JavaScript

Absolutely! Let's start with **1. Introduction to JavaScript** in detail:

---

1. Introduction to JavaScript

What is JavaScript?
JavaScript (JS) is a programming language used mainly to make websites interactive and dynamic. It runs in the browser (like Chrome, Firefox) but can also run on servers (using Node.js).

Why use JavaScript?

- Adds interactivity to web pages (like buttons, forms, animations)
- Can change content dynamically without reloading the page
- Works with HTML and CSS to create modern websites
- Widely used and supported by all browsers

Where does JavaScript run?

- **Client-side**: Runs inside your browser, interacting directly with the webpage.
- **Server-side**: With tools like Node.js, JS can run on servers to build backend systems.

JavaScript Characteristics:

- Interpreted language (runs directly without compilation)
- Loosely typed (no strict rules on variable types)
- Object-oriented but also supports functional programming
- Event-driven (reacts to user actions like clicks)

Hello World Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>JS Introduction</title>
</head>
<body>

<h1>Hello JavaScript!</h1>

<script>
  console.log("Hello, World!");  // Prints message to browser console
  alert("Welcome to JavaScript!"); // Shows popup message to user
```

```
</script>

</body>
</html>
```

How to run JS?

- Embed directly inside HTML using `<script>` tag (like above)
- Link to an external JS file using `<script src="filename.js"></script>`
- Use browser developer console to type and test JS commands instantly

---

Would you like me to explain how to include JS in HTML next or try some simple code examples?

Got it! Here's the next topic in detail:

---

2. Including JavaScript in HTML

To make your web pages interactive with JavaScript, you need to include your JS code in your HTML file. There are two main ways to do this:

a) Inline JavaScript (Inside HTML file)

You can write JavaScript directly inside your HTML file using the `<script>` tag.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline JS</title>
</head>
<body>

<h1>Welcome!</h1>

<script>
  alert("Hello from inline JavaScript!");
</script>

</body>
</html>
```

The `<script>` tag can be placed inside `<head>` or `<body>`. Usually, it's better to place it just before `</body>` so the page loads first.

---

b) External JavaScript File

You can write JavaScript in a separate file (with `.js` extension) and link it to your HTML using the `src` attribute.

Example:

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>External JS</title>
</head>
<body>

<h1>Welcome!</h1>

<script src="script.js"></script>

</body>
</html>
```

script.js

```
alert("Hello from external JavaScript file!");
```

This helps keep your code organized and easier to manage, especially for bigger projects.

Notes:

- When using the external file, make sure the `.js` file is in the correct folder relative to the HTML file.
- You can include multiple `<script>` tags if needed.
- By default, scripts block page loading, but you can add attributes like `defer` or `async` to change this behavior.

Want me to explain those `defer` and `async` attributes next, or should I move to the next topic?

Sure! Here's the next topic in detail:

---

3. Output Methods in JavaScript

JavaScript provides several ways to display output or show information to the user or developer.

a) `console.log()`

- Prints messages to the browser's **console**.
- Used mainly for debugging.
- You can open the console in browsers by pressing **F12** or **Ctrl+Shift+I** and navigating to the Console tab.

Example:

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

---

b) `alert()`

- Shows a popup message box to the user.
- Blocks the browser until the user clicks "OK".
- Not recommended for frequent use because it interrupts user experience.

Example:

```
alert("This is an alert message!");
```

---

c) `document.write()`

- Writes HTML content directly to the webpage.
- Usually used for simple tests or examples.
- If used after the page has loaded, it can overwrite the entire page content.

Example:

```
document.write("Hello, this text is written on the page.");
```

---

d) Other output methods (for later learning)

- Modifying HTML elements using JavaScript (like changing innerHTML, textContent)
- Using the DOM to update page content dynamically

Want me to continue with **4. Variables and Data Types** next?

Sure! Here's the next topic in detail:

---

4. Variables and Data Types

Variables

Variables are containers to store data values. You can create variables in JavaScript using three keywords:

- `var` — old way, function-scoped
- `let` — newer, block-scoped (recommended)
- `const` — block-scoped, cannot be changed after assignment (constant)

Example:

```
let name = "Alice";
const age = 25;
var city = "New York";
```

Data Types

JavaScript has several basic data types:

- **String** — text inside quotes
  Example: `"Hello"`, `'JavaScript'`
- **Number** — integers or decimals
  Example: `10`, `3.14`
- **Boolean** — true or false
  Example: `true`, `false`
- **Null** — empty or "no value"
  Example: `null`
- **Undefined** — variable declared but not assigned a value
  Example: `let x;` (x is `undefined`)
- **Symbol** — unique identifier (advanced)
- **Object** — complex data structure (will learn later)

Example:

```
let message = "Hi there!"; // String
let count = 100;           // Number
let isActive = true;       // Boolean
let emptyValue = null;     // Null
let notDefined;            // Undefined
```

Want me to explain **5. Operators** next?

Here's the next topic in detail:

---

5. Operators

Operators are symbols that perform operations on values or variables.

Types of Operators:

1. **Arithmetic Operators**
   Perform math calculations:

- `+` (add)
- `-` (subtract)
- `*` (multiply)
- `/` (divide)
- `%` (modulus - remainder)
- `++` (increment by 1)
- `--` (decrement by 1)

Example:

```
let x = 10;
let y = 3;
console.log(x + y);  // 13
console.log(x % y);  // 1
```

---

2. **Assignment Operators**
   Assign values to variables:

- `=`
- `+=` (add and assign)
- `-=` (subtract and assign)
- `*=`
- `/=`

Example:

```
let a = 5;
a += 3;  // same as a = a + 3; now a = 8
```

---

3. **Comparison Operators**
   Compare two values and return true or false:

- `==` (equal to)
- `===` (strict equal, checks value and type)
- `!=` (not equal)
- `!==` (strict not equal)
- `>` (greater than)
- `<` (less than)
- `>=` (greater or equal)
- `<=` (less or equal)

Example:

```
console.log(5 == "5");   // true (only value compared)
console.log(5 === "5");  // false (type differs)
```

---

4. **Logical Operators**
   Used with boolean values:

- `&&` (AND) — true if both sides true
- `||` (OR) — true if one side true
- `!` (NOT) — reverses boolean

Example:

```
console.log(true && false);  // false
console.log(true || false);  // true
console.log(!true);          // false
```

Want to go to **6. Conditional Statements** next?

Here's the next topic in detail:

---

6. Conditional Statements

Conditional statements let your program make decisions based on conditions.

---

a) `if` statement

Runs code only if a condition is true.

Example:

```
let age = 18;
if (age >= 18) {
  console.log("You are an adult.");
}
```

---

b) `if...else` statement

Runs one block if condition is true, another if false.

Example:

```
let age = 16;
if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
```

---

c) `else if` statement

Checks multiple conditions in order.

Example:

```
let score = 75;
if (score >= 90) {
  console.log("Grade A");
} else if (score >= 75) {
  console.log("Grade B");
} else {
  console.log("Grade C");
}
```

---

d) `switch` statement

Checks one variable against many values.

Example:

```
let color = "red";
switch(color) {
  case "red":
    console.log("Stop");
    break;
  case "yellow":
    console.log("Get ready");
    break;
  case "green":
    console.log("Go");
    break;
  default:
    console.log("Unknown color");
}
```

Want to continue with **7. Loops** next?

Here's the next topic in detail:

---

7. Loops

Loops help run the same code repeatedly until a condition is met.

---

a) `for` loop

Runs code a specific number of times.

Example:

```
for (let i = 0; i < 5; i++) {
  console.log("Count:", i);
}
```

---

b) `while` loop

Runs code as long as a condition is true.

Example:

```
let i = 0;
while (i < 5) {
  console.log("Count:", i);
  i++;
}
```

---

c) `do...while` loop

Runs code at least once, then repeats while condition is true.

Example:

```
let i = 0;
do {
  console.log("Count:", i);
  i++;
} while (i < 5);
```

Want to go on with **8. Functions** next?

Here's the next topic in detail:

---

8. Functions

Functions are reusable blocks of code that perform a task.

---

a) Function Declaration

Example:

```
function greet() {
  console.log("Hello!");
}
greet();  // Calls the function
```

---

b) Function with Parameters

You can pass values to functions.

Example:

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

---

c) Return Statement

Functions can return a value.

Example:

```
function add(a, b) {
  return a + b;
}
let result = add(3, 4);
console.log(result);  // 7
```

---

d) Function Expression

Functions can be stored in variables.

Example:

```
const sayHi = function() {
  console.log("Hi!");
};
sayHi();
```

---

e) Arrow Functions (ES6)

Shorter syntax for functions.

Example:

```
const multiply = (x, y) => x * y;
console.log(multiply(2, 3));  // 6
```

Want me to continue with **9. Objects**?

Here's the next topic in detail:

---

9. Objects

Objects store data in key-value pairs and represent real-world things.

Creating an Object

```
const person = {
  name: "Alice",
  age: 25,
  city: "New York"
};
```

Accessing Properties

- Dot notation:

```
console.log(person.name);  // Alice
```

- Bracket notation:

```
console.log(person["age"]);  // 25
```

Adding or Changing Properties

```
person.job = "Developer";
person.age = 26;
```

Methods (Functions inside objects)

```
const person = {
  name: "Alice",
  greet: function() {
    console.log("Hello, " + this.name);
  }
};
person.greet();  // Hello, Alice
```

Want to go on with **10. Arrays**?

Here's the next topic in detail:

---

10. Arrays

Arrays store multiple values in a single variable, in an ordered list.

Creating an Array
let fruits = ["apple", "banana", "cherry"];

Accessing Elements

Array elements are accessed by their index (starting from 0):

console.log(fruits[0]); // apple

Changing Elements
fruits[1] = "blueberry";

Array Properties and Methods

- `.length` — number of elements
- `.push()` — add to end
- `.pop()` — remove from end
- `.shift()` — remove from start
- `.unshift()` — add to start

Example:

fruits.push("orange");
console.log(fruits.length); // 4

Want to continue with **11. Scope**?

Here's the next topic in detail:

---

11. Scope

Scope determines where variables are accessible in your code.

Global Scope

Variables declared outside any function or block are **global** — accessible anywhere.

Example:

```
let globalVar = "I am global";
function test() {
  console.log(globalVar);  // Accessible here
}
test();
console.log(globalVar);    // Also accessible here
```

Local Scope (Function Scope)

Variables declared inside a function are local to that function only.

Example:

```
function test() {
  let localVar = "I am local";
  console.log(localVar);  // Works here
}
test();
console.log(localVar);    // Error! localVar is not defined here
```

Block Scope

Variables declared with `let` or `const` inside `{}` (like inside loops or if blocks) are limited to that block.

Example:

```
if (true) {
  let blockVar = "Block scoped";
  console.log(blockVar);  // Works here
}
console.log(blockVar);    // Error! blockVar not defined outside block
```

Want to continue with **12. Hoisting**?

Here’s the next topic in detail:

---

12. Hoisting

Hoisting means JavaScript moves **variable and function declarations** to the top of their scope before code runs.

How it works with variables:

- Variables declared with `var` are **hoisted** but initialized as `undefined` until assigned.

Example:

```
console.log(x);  // undefined (not error)
var x = 5;
```

- Variables declared with `let` and `const` are **not accessible** before declaration (called “temporal dead zone”).

Example:

```
console.log(y);  // ReferenceError
let y = 10;
```

How it works with functions:

- Function declarations are hoisted fully, so you can call a function before it appears in the code.

Example:

```
sayHello();  // Works fine

function sayHello() {
  console.log("Hello!");
}
```

- Function expressions (functions stored in variables) behave like variables and are hoisted differently.

Example:

```
sayHi();  // Error: sayHi is not a function

var sayHi = function() {
  console.log("Hi!");
};
```

Want me to continue with **13. DOM Manipulation**?

Absolutely! Let me explain **DOM Manipulation** in more detail with examples and explanations:

---

13. DOM Manipulation (Detailed)

What is DOM?
DOM stands for **Document Object Model**. It's a way browsers represent the HTML of a webpage as a structured tree of objects (elements). JavaScript can use the DOM to read or change anything on the page — text, images, styles, and even structure.

---

1. Selecting Elements from the DOM

Before you can change or read anything, you need to select the element(s).

- **By ID** (selects one unique element):

```
const heading = document.getElementById("main-heading");
```

- **By Class Name** (selects multiple elements as a collection):

```
const items = document.getElementsByClassName("list-item");
```

- **By Tag Name** (selects elements by tag, like all <p>):

```
const paragraphs = document.getElementsByTagName("p");
```

- **By CSS Selector** (more flexible, can select any element using CSS selectors):

```
const firstButton = document.querySelector("button");
const allButtons = document.querySelectorAll(".btn");
```

---

2. Changing Element Content

- Change **text** inside an element (without HTML tags):

```
heading.textContent = "Welcome to My Site!";
```

- Change **HTML** inside an element (can include tags):

```
heading.innerHTML = "Welcome to <em>My Site</em>!";
```

---

3. Changing Styles

You can modify the inline CSS styles of elements:

```
heading.style.color = "blue";
heading.style.backgroundColor = "yellow";
heading.style.fontSize = "24px";
```

---

4. Creating New Elements

You can create new elements and add them to the page:

```
const newDiv = document.createElement("div");  // Create a <div>
newDiv.textContent = "This is a new div";      // Add text inside it
document.body.appendChild(newDiv);             // Add it to the end of <body>
```

---

5. Removing Elements

You can remove elements from the page:

```
const oldDiv = document.getElementById("old-div");
oldDiv.remove();  // Removes the element from DOM
```

---

6. Changing Attributes

You can get or set element attributes (like `src`, `href`, `class`):

```
const link = document.querySelector("a");
```

```
console.log(link.getAttribute("href"));        // Get href value
link.setAttribute("href", "https://google.com");  // Change href value
```

---

7. Example: Change text on button click

HTML:

```
<button id="myBtn">Click me</button>
<p id="demo">Hello!</p>
```

JavaScript:

```
const button = document.getElementById("myBtn");
const paragraph = document.getElementById("demo");

button.onclick = function() {
  paragraph.textContent = "Button was clicked!";
};
```

If you want, I can explain **Event Handling** next (how to respond to clicks, typing, etc.), which often works closely with DOM manipulation.

Would you like me to continue?

Here’s the next topic in detail:

---

14. Event Handling

Events are actions like clicks, typing, or mouse moves that happen in the browser.
JavaScript can “listen” to these events and respond.

Adding Event Listeners

You can attach functions that run when events happen.

Example — respond to a button click:

```
const button = document.getElementById("myButton");

button.addEventListener("click", function() {
  alert("Button clicked!");
```

});

Common Events

- `"click"` — mouse click
- `"mouseover"` — mouse moves over an element
- `"mouseout"` — mouse moves out of element
- `"keydown"` — key pressed down
- `"keyup"` — key released
- `"submit"` — form submission

Using named functions

```
function showMessage() {
  console.log("Hello!");
}

button.addEventListener("click", showMessage);
```

Removing Event Listeners

```
button.removeEventListener("click", showMessage);
```

Want to continue with **15. ES6 Features**?

Here's the next topic in detail:

---

15. ES6 Features

ES6 (ECMAScript 2015) introduced many modern features that make JavaScript easier and cleaner.

---

a) `let` and `const`

Block-scoped variables and constants (instead of `var`).

Example:

```
let count = 1;
const PI = 3.14;
```

---

b) Arrow Functions

Shorter function syntax.

Example:

```
const add = (a, b) => a + b;
```

---

c) Template Literals

Strings with embedded variables, using backticks `.

Example:

```
const name = "Alice";
console.log(`Hello, ${name}!`);
```

---

d) Default Parameters

Set default values for function parameters.

Example:

```
function greet(name = "Guest") {
  console.log(`Hello, ${name}`);
}
```

---

e) Destructuring

Extract values from arrays or objects.

Example:

```
const [a, b] = [1, 2];
const {name, age} = {name: "Bob", age: 30};
```

---

f) Modules

`import` and `export` to organize code (used in bigger projects).

Want to keep going or stop here?

Here's the next topic in detail:

---

16. Promises & Async/Await

These help JavaScript handle **asynchronous operations** (actions that take time, like fetching data from the internet) without blocking the rest of the code.

What is a Promise?

A **Promise** represents a value that might be available now, later, or never.

It has three states:

- **Pending** — initial state, not finished yet
- **Fulfilled** — operation completed successfully
- **Rejected** — operation failed

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve("Operation succeeded!");
  } else {
    reject("Operation failed.");
  }
});
```

Using `.then()` and `.catch()`

```
myPromise
  .then(result => {
    console.log(result);  // "Operation succeeded!"
  })
  .catch(error => {
    console.error(error);
  });
```

Async/Await (Modern syntax)

Makes asynchronous code look like synchronous code, easier to read.

```
async function doTask() {
  try {
    let result = await myPromise;
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}
doTask();
```

Example: Fetch data from API

```
async function fetchData() {
  try {
    let response = await fetch("https://api.example.com/data");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
fetchData();
```

Want me to continue with **17. Classes & Objects** next?

Here's the next topic in detail:

---

17. Classes & Objects (ES6 Classes)

Classes provide a clear, simpler way to create objects and handle object-oriented programming in JavaScript.

Creating a Class

```
class Person {
  constructor(name, age) {
```

```
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

Creating Objects from Classes

```
const alice = new Person("Alice", 30);
alice.greet();  // Hello, my name is Alice
```

Inheritance

Classes can inherit properties and methods from other classes.

```
class Student extends Person {
  constructor(name, age, grade) {
    super(name, age);  // Call parent constructor
    this.grade = grade;
  }

  study() {
    console.log(`${this.name} is studying.`);
  }
}

const bob = new Student("Bob", 20, "A");
bob.greet();  // From Person
bob.study();
```

Want to continue with **18. Error Handling**?

Here's the next topic in detail:

---

18. Error Handling

Error handling helps your program respond to problems without crashing.

`try...catch` block

Use `try` to run code that might throw an error, and `catch` to handle the error.

```
try {
  let result = riskyOperation();
  console.log(result);
} catch (error) {
  console.error("Something went wrong:", error);
}
```

`finally` block

Code in `finally` runs after `try` and `catch`, no matter what.

```
try {
  // code
} catch (e) {
  // handle error
} finally {
  console.log("This runs always.");
}
```

Throwing Errors

You can create your own errors with `throw`.

```
function checkAge(age) {
  if (age < 18) {
    throw new Error("You must be at least 18.");
  }
  return true;
}

try {
  checkAge(15);
} catch (e) {
  console.log(e.message);
}
```

Want me to continue with **19. Local Storage & Session Storage**?

Here's the next topic in detail:

---

19. Local Storage & Session Storage

Both let you store data in the browser for later use.

Local Storage

- Stores data with **no expiration** (data stays until cleared).
- Data saved as **strings**.
- Use `localStorage` object.

Example:

```
localStorage.setItem("name", "Alice");      // Save data
let name = localStorage.getItem("name");    // Read data
localStorage.removeItem("name");            // Remove data
localStorage.clear();                       // Clear all data
```

Session Storage

- Stores data for the **current browser session only**.
- Data is cleared when the tab or browser closes.
- Use `sessionStorage` object.

Example:

```
sessionStorage.setItem("token", "abc123");
let token = sessionStorage.getItem("token");
sessionStorage.clear();
```

Use cases:

- Save user preferences
- Keep login status
- Store temporary data

Want to continue with **20. JSON**?

Here's the next topic in detail:

---

20. JSON (JavaScript Object Notation)

JSON is a lightweight data format used to exchange data between a server and a web app.

JSON Format

- Looks like JavaScript objects, but is **text** (string).
- Uses key-value pairs with double quotes " " for keys and strings.

Example JSON:

```
{
  "name": "Alice",
  "age": 25,
  "isStudent": true,
  "courses": ["Math", "English"]
}
```

Converting between JSON and JavaScript

- Convert JavaScript object to JSON string:

```
const obj = { name: "Alice", age: 25 };
const jsonStr = JSON.stringify(obj);
console.log(jsonStr);  // '{"name":"Alice","age":25}'
```

- Convert JSON string to JavaScript object:

```
const jsonStr = '{"name":"Alice","age":25}';
const obj = JSON.parse(jsonStr);
console.log(obj.name);  // Alice
```

Used for sending data in APIs, saving configs, and more.

Want to continue with **21. Fetch API / AJAX**?

Here's the next topic in detail:

---

21. Fetch API / AJAX

Used to get or send data to a server without reloading the page.

Fetch API (modern way)

Basic example to get data:

```
fetch('https://api.example.com/data')
  .then(response => response.json())  // Parse JSON response
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Using async/await with Fetch

```
async function getData() {
  try {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
getData();
```

AJAX = Asynchronous JavaScript and XML (older technique, Fetch is newer and better).

Want to continue with **22. Regular Expressions**?

Here's the next topic in detail:

---

22. Regular Expressions (RegEx)

Regular Expressions are patterns used to match or find text in strings.

Creating a RegEx

```
const pattern = /abc/;         // Matches 'abc' anywhere
const pattern2 = new RegExp("abc");  // Same as above
```

Common Methods with RegEx

- `test()` — checks if pattern exists

```
pattern.test("abcdef");  // true
```

- `match()` — gets matches

```
"abcdef".match(pattern);  // ['abc']
```

- `replace()` — replaces matched text

```
"abc123".replace(/\d+/g, "");  // 'abc'
```

Flags

- `g` — global search (all matches)
- `i` — case insensitive
- `m` — multiline

Example:

```
const re = /hello/gi;
```

RegEx is useful for validation (like email, phone), search, and replace tasks.

Want to continue with **23. Event Propagation**?

Here's the next topic in detail:

---

23. Event Propagation

Event propagation describes how events flow through the DOM tree when an event happens.

Two Phases:

- **Capturing phase:** Event travels from the top (window) down to the target element.
- **Bubbling phase:** Event bubbles up from the target element back to the top.

Example

```
element.addEventListener("click", function() {
  console.log("Clicked!");
}, true);  // true = use capturing phase
```

If `false` or omitted, the listener uses bubbling phase (default).

Event Delegation

Instead of adding event listeners to many elements, add one listener to a common parent and check event target.

Example:

```
document.getElementById("parent").addEventListener("click", function(e) {
  if(e.target && e.target.matches("button.className")) {
    console.log("Button clicked:", e.target);
  }
});
```

Want to continue with **24. Closures**?

Here's the next topic in detail:

---

24. Closures

A **closure** happens when a function remembers the variables from its outer (enclosing) scope, even after that outer function has finished running.

Example

```
function outer() {
  let count = 0;
```

```
  function inner() {
    count++;
    console.log(count);
  }

  return inner;
}

const counter = outer();
counter();  // 1
counter();  // 2
counter();  // 3
```

`inner` remembers `count` from `outer`'s scope even after `outer` is done.

Closures help with:

- Data privacy
- Creating function factories
- Maintaining state in async code

Want to continue with **25. Callbacks**?

Here's the next topic in detail:

---

25. Callbacks

A **callback** is a function passed as an argument to another function, to be called later.

Example:

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

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

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

Callbacks are used for tasks that happen later, like:

- Handling responses from servers
- Timers and delays
- Event listeners

Want to continue with **26. Modules & Bundling**?

Here's the next topic in detail:

---

26. Modules & Bundling

Modules help organize code into separate files and import/export functions, objects, or variables between them.

Exporting

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Importing

```
// main.js
import { add } from './math.js';
console.log(add(2, 3));  // 5
```

Bundling

When building big projects, tools like **Webpack**, **Rollup**, or **Parcel** bundle all modules into one file for browsers.

Want to continue with **27. JavaScript Design Patterns**?

Here's the next topic in detail:

---

27. JavaScript Design Patterns

Design patterns are common solutions to recurring coding problems.

Common Patterns:

- **Module Pattern**: Encapsulates code and exposes only what's needed.
- **Singleton Pattern**: Ensures only one instance of an object exists.
- **Observer Pattern**: Objects watch for changes and react (like events).
- **Factory Pattern**: Creates objects without specifying exact class.

Example — Module Pattern:

```
const Counter = (function() {
  let count = 0;
  return {
    increment() {
      count++;
      console.log(count);
    }
  };
})();

Counter.increment(); // 1
```

Want to continue with **28. Web APIs**?

Here's the next topic in detail:

---

28. Web APIs

Web APIs let JavaScript interact with browser features beyond basic HTML.

Examples:

- **Geolocation API** — get user's location

```
navigator.geolocation.getCurrentPosition(position => {
  console.log(position.coords.latitude, position.coords.longitude);
});
```

- **Canvas API** — draw graphics

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
ctx.fillStyle = 'red';
ctx.fillRect(10, 10, 100, 50);
```

- **Notification API** — show notifications

```
Notification.requestPermission().then(permission => {
  if(permission === 'granted') {
    new Notification('Hello!');
  }
});
```

Want to continue with **29. Functional Programming**?

Here's the next topic in detail:

---

29. Functional Programming

Functional programming uses functions as first-class objects and avoids changing data.

Key Concepts:

- **Pure Functions** — No side effects, same output for same input.
- **Immutability** — Don't change data, create new copies.
- **Higher-Order Functions** — Functions that take or return other functions.

Common Methods:

- `map()` — transform array

```
const nums = [1, 2, 3];
const doubled = nums.map(x => x * 2);  // [2, 4, 6]
```

- `filter()` — select elements

```
const evens = nums.filter(x => x % 2 === 0);  // [2]
```

- `reduce()` — reduce array to single value

```
const sum = nums.reduce((total, x) => total + x, 0);  // 6
```

Want to continue with **30. Generators & Iterators**?

Here's the next topic in detail:

---

30. Generators & Iterators

Generators are special functions that can pause and resume, useful for custom iteration.

Generators

Use `function*` and `yield` to create them.

```
function* countUp() {
  yield 1;
  yield 2;
  yield 3;
}

const counter = countUp();
console.log(counter.next().value);  // 1
console.log(counter.next().value);  // 2
console.log(counter.next().value);  // 3
```

Iterators

Objects that follow the iterator protocol by having a `next()` method returning `{ value, done }`.

Example:

```
const iterable = {
  data: [10, 20, 30],
  [Symbol.iterator]() {
    let index = 0;
    const data = this.data;
    return {
      next() {
        if (index < data.length) {
          return { value: data[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for (const value of iterable) {
  console.log(value);
}
```

That covers all main JavaScript topics! Want me to help with examples or practice exercises?