

Full Stack Development (23CSPE311)

Module-1: JavaScript and DOM Manipulation

Basic JavaScript Instructions, Statements, Comments, Var Loops, Functions, Methods & Objects, Functions & Methods, Objects & Arrays. DOM Manipulation, Selecting Elements, Working with DOM Nodes, Updating Element Content & Attributes, Events, Different Types of Events, How to Bind Delegation, Event Listeners.

Basic JavaScript Instructions

JavaScript is a **lightweight, cross-platform, and interpreted compiled programming language which is also known as the scripting language for webpages.**

2 way usage

1. Inside HTML
2. External File

Basic JavaScript Instructions

1. Write it inside an HTML file

Put your JavaScript code between `<script>` and `</script>` tags in an HTML page.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First JavaScript</h1>

<script>
    alert("Hello! This is JavaScript.");
</script>

</body>
</html>
```

Basic JavaScript Instructions

2. Put it in a separate file

Save your code in a file ending with .js (example: script.js) and connect it to your HTML.

Example:

HTML file:

```
html

<!DOCTYPE html>
<html>
<body>

<h1>External JavaScript Example</h1>
<script src="script.js"></script>

</body>
</html>
```

script.js:

```
javascript

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

Basic JavaScript Instructions

Statement

- A statement is a single instruction for the computer.
- Each statement should:
 - Start on a new line
 - End with a semicolon (;)
- Semicolons signal the end of a step so JavaScript knows to move to the next one.

Code Blocks

- Some statements are grouped inside curly braces { } — these are called code blocks.
- The closing curly brace is not followed by a semicolon.
- Code blocks group related statements together, making code organized and easier to read.

Basic JavaScript Instructions

Statement

Each of the lines of code in green is a statement.

The Black curly braces indicate the start and end of a code block. (Each code block could contain many more statements.)

The code in purple determines which code should run.

```
var today = new Date ();  
var hourNow = today.getHours();  
var greeting;  
if (hourNow > 18) {  
    greeting = 'Good evening';  
} else if (hourNow > 12) {  
    greeting = 'Good afternoon';  
} else if (hourNow > 0) {  
    greeting = 'Good morning';  
} else {  
    greeting = 'Welcome';  
}  
document.write(greeting);
```

Basic JavaScript Instructions

Comments

- In JavaScript, comments are notes you write in your code that the computer ignores.
- They help explain what the code does for you or others reading it.
- You can write a single-line comment using `//`.
- For multi-line comments, use `/* ... */`.
- Comments make code easier to understand and maintain.
- They are not executed, so they don't affect how the program runs.

Basic JavaScript Instructions

Comments

```
/* This script displays a greeting to the user based upon the current time.  
It is an example from JavaScript & jQuery book */
```

```
var today = new Date(); // Create a new date object  
var hourNow = today.getHours(); // Find the current hour  
var greeting
```

Basic JavaScript Instructions

Comments

```
/* This script displays a greeting to the user based upon the current time.  
It is an example from JavaScript & jQuery book */
```

```
var today = new Date(); // Create a new date object
```

```
var hourNow = today.getHours(); // Find the current hour  
var greeting
```

Basic JavaScript Instructions

Variables

A variable in JavaScript is a named storage for data. It acts like a container that holds values which can be used and modified in a program.

Before using a variable, you must declare it this means creating it and giving it a name.

Example

syntax:

```
var quantity;
```

var → Variable keyword (tells JavaScript you are creating a variable).

quantity → Variable name (identifier).

Basic JavaScript Instructions

- The JavaScript interpreter recognizes `var` as a keyword for variable creation.
- Every variable must have a name so it can be referred to later in code.
- If a variable name has more than one word, use **camelCase**:
- First word all lowercase Subsequent words start with uppercase (e.g., `totalAmount`).

Basic JavaScript Instructions

Variables: How to Assign Them a Value

After creating a variable, you can assign a value to it (store information in it).

Example:

quantity = 3;

quantity → Variable name

= → Assignment operator (tells JavaScript to assign a value)

3 → Variable value

Basic JavaScript Instructions

DATA TYPES

- JavaScript distinguishes between **numbers**, **strings**, and **true or false** values known as **Booleans**.

NUMERIC DATA TYPE

- Handles numbers.
- Example:
 0.75
- For counting or calculations,
- use numbers 0-9.
- Examples: 5272 (no commas between thousands and hundreds)
- Negative numbers: -23678
- Decimals: 0.75

Basic JavaScript Instructions

STRING DATA TYPE

Consists of letters and other characters.

Example:

'Hi, Hello!'

Strings are enclosed in quotes (single or double).

Opening and closing quotes must match.

Used for any kind of text.

Often used to add content to a page and can contain HTML markup.

Basic JavaScript Instructions

BOOLEAN DATA TYPE

Has two possible values: true or false.

Example:

For Arduino

True Acts like a light switch (on/off).

Useful for controlling which parts of a script run.

Basic JavaScript Instructions

1. Using a Variable to Store a Number

A number variable holds numeric values that you can use for calculations like addition, subtraction, multiplication, and division.

In JavaScript, numbers can be integers (like 5) or floating-point (like 3.14).

2. Using a Variable to Store a String

A string variable holds a sequence of characters like words, sentences, or symbols.

Strings must be enclosed in single quotes ('), double quotes ("), or backticks (`).

3. Using a Variable to Store a Boolean

A boolean variable holds only two values:

true → something is correct or "yes"

false → something is wrong or "no"

Basic JavaScript Instructions

```
// Number variable
let age = 25;

// String variable
let name = "Shwetha";

// Boolean variable
let isStudent = true;

// Using them together
console.log("Name: " + name);           // Output: Name: Shwetha
console.log("Age: " + age);             // Output: Age: 25
console.log("Is a student? " + isStudent); // Output: Is a student? true

// Example of logic using the variables
if (isStudent && age < 30) {
    console.log(name + " is a young student.");
} else {
    console.log(name + " is not a young student.");
}
```

Basic JavaScript Instructions

- Variables are declared and values assigned in the same statement.
- Three variables are declared on the same line, then values assigned to each.
- Two variables are declared and assigned values on the same line. Then one is declared and assigned a value on the next line.
- Here, a variable is used to hold a reference to an element in the HTML page. This allows you to work directly with the element stored in that variable.

```
JAVASCRIPT c02/js/shorthand-variable.js
```

- ①

```
var price = 5;
var quantity = 14;
var total = price * quantity;
```
- ②

```
var price, quantity, total;
price = 5;
quantity = 14;
total = price * quantity;
```
- ③

```
var price = 5, quantity = 14;
var total = price * quantity;
```
- ④

```
// Write total into the element with id of cost
var el = document.getElementById('cost');
el.textContent = '$' + total;
```

changing the value of variable

- Changing the value of a variable in JavaScript is simple, you assign it a new value after it has already been declared.

```
let age = 25;  
console.log(age); // Output: 25  
  
age = 30; // change value  
console.log(age); // Output: 30  
  
age = age + 5; // update using old value  
console.log(age); // Output: 35
```

Rules for Naming Variables in JavaScript

1. **Start with the right character** – Variable names must begin with a letter, _ (underscore), or \$ (dollar sign). They cannot start with a number.
2. **Allowed characters** – After the first letter, you can use letters, numbers, _, or \$. You cannot use spaces, - (dash), or . (dot).
3. **No keywords** – You cannot use JavaScript reserved words like var, function, if, etc., as variable names.
4. **Case-sensitive** – score and Score are treated as two different variables.
5. **Meaningful names** – Choose names that describe the data stored (e.g., firstName for a person's first name).
6. **Multi-word style** – For names with more than one word, use camelCase (firstName) or underscores (first_name). Do not use dashes.

JavaScript Arrays

What is an Array?

An array is a special variable that can hold multiple values in a single name.

Instead of creating many variables, you can store a list of values in one array.

```
let colors = ["red", "green", "blue"];  
console.log(colors);    // Output: ["red", "green", "blue"]
```

JavaScript Arrays

Creating Arrays

You can make an array in two ways:

Using square brackets (common way)

```
let fruits = ["apple", "banana", "mango"];
```

Using the Array constructor

```
let fruits = new Array("apple", "banana", "mango");
```

Accessing Array Values

- Each value has an **index** (position number), starting from **0**.

```
let fruits = ["apple", "banana", "mango"];  
console.log(fruits[0]); // apple  
console.log(fruits[1]); // banana  
console.log(fruits[2]); // mango
```

JavaScript Arrays

Changing array values in JavaScript just means **replacing an existing value at a certain position (index)** with a new one.

1. Arrays have indexes that start from 0 for the first element.
2. You can target an index directly and assign it a new value.
3. This updates the array in place ,you don't have to recreate it.

```
let fruits = ["apple", "banana", "mango"];  
console.log(fruits); // ["apple", "banana", "mango"]  
// Change "banana" to "orange"  
fruits[1] = "orange";  
console.log(fruits); // ["apple", "orange", "mango"]
```

Expressions and Operators in JavaScript

What is an Expression?

- An **expression** is any piece of code that produces a **value**.
It can be a number, a variable, a calculation, or even a function call.

5 // produces the value 5

x // produces the value stored in x

5 + 3 // produces 8

"Hello" + "!" // produces "Hello!"

Expressions and Operators in JavaScript

What is an Operator?

- An **operator** is a symbol that tells JavaScript to do something with one or more values (operands).

Arithmetic Operators (Math operations)

- These work with numbers to perform calculations.

Operator	Meaning	Example	Output
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 3	1
**	Exponentiation (power)	2 ** 3	8
++	Increment (add 1)	x = 5; x++;	6
--	Decrement (subtract 1)	x = 5; x--;	4

```
let a = 10, b = 3;
console.log(a + b);
// 13
console.log(a % b);
// 1
console.log(a * b); //
30
```

Expressions and Operators in JavaScript

Assignment Operators (Store values in variables)

- These assign values and can combine with arithmetic.

Operator	Meaning	Example	Result
=	Assign value	<code>x = 5</code>	<code>x</code> becomes 5
+=	Add and assign	<code>x += 3</code>	<code>x = x + 3</code>
-=	Subtract and assign	<code>x -= 3</code>	<code>x = x - 3</code>
*=	Multiply and assign	<code>x *= 3</code>	<code>x = x * 3</code>
/=	Divide and assign	<code>x /= 3</code>	<code>x = x / 3</code>
%=	Modulus and assign	<code>x %= 3</code>	<code>x = x % 3</code>
**=	Power and assign	<code>x **= 3</code>	<code>x = x ** 3</code>

```
let x = 5;  
x += 2; // x = 7  
x *= 3; // x = 21  
console.log(x); //  
21
```

Expressions and Operators in JavaScript

Comparison Operators (Check values → true/false)

Used in conditions (if, loops, etc.)

```
let age = 18;  
console.log(age >= 18); // true  
console.log(5 === "5"); // false
```

Operator	Meaning	Example	Result
==	Equal to (loose, ignores type)	5 == "5"	true
===	Equal to (strict, checks type)	5 === "5"	false
!=	Not equal to (loose)	5 != "5"	false
!==	Not equal to (strict)	5 !== "5"	true
>	Greater than	5 > 3	true
<	Less than	5 < 3	false
>=	Greater or equal	5 >= 5	true
<=	Less or equal	5 <= 3	false

Expressions and Operators in JavaScript

Logical Operators

Logical operators are used when you want to check multiple conditions at the same time.

They always return true or false.

AND (&&)

Returns true if both conditions are true.

If either one is false, result is false.

OR (||)

Returns true if at least one condition is true.

Only returns false if both are false.

NOT (!)

Reverses the result: If something is true, it becomes false.

If it's false, it becomes true.

```
let isAdult = true;
let hasTicket = false;

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

Expressions and Operators in JavaScript

String Operators

In JavaScript, strings are pieces of text, and string operators let you combine or add to them.

Concatenation (+): Join (combine) two strings into a new one.

Concatenation Assignment (+=) Append (add) new text directly to an existing string variable.

Operator	Name	Description	Example	Result
<code>+</code>	Concatenation	Joins two strings into one.	<code>"Hello" + " World"</code>	<code>"Hello World"</code>
<code>+=</code>	Concatenation Assignment	Adds (appends) new text to an existing string.	<code>let msg = "Hi"; msg += " there!";</code>	<code>"Hi there!"</code>

Functions in JavaScript

Functions in JavaScript are reusable blocks of code designed to perform specific tasks. They allow you to organize, reuse, and modularize code. It can take inputs, perform actions, and return outputs.

```
function sum(x, y)
{
return x + y;
}
console.log(sum(6, 9)); // Output=15
```

Functions in JavaScript

Function Syntax and Working

- A function definition is sometimes also termed a function declaration or function statement. Below are the rules for creating a function in JavaScript:
- Begin with the keyword **function** followed by,
- A user-defined function name (In the above example, the name is **sum**)
- A list of parameters enclosed within parentheses and separated by commas (In the above example, parameters are **x** and **y**)
- A list of statements composing the body of the function enclosed within curly braces **{}** (In the above example, the statement is “return **x + y**”).

Functions in JavaScript

- **Return Statement**

- In some situations, we want to return some values from a function after performing some operations.
- In such cases, we make use of the return. This is an optional statement. In the above function, “sum()” returns the sum of two as a result.

Function Parameters

- Parameters are input passed to a function. In the above example, sum() takes two parameters, x and y.

Calling Functions

- After defining a function, the next step is to call them to make use of the function. We can call a function by using the function name separated by the value of parameters enclosed between the parenthesis.

Functions in JavaScript

// Function Definition

```
function welcomeMsg(name) {  
return ("Hello " + name + " welcome "); }
```

```
let nameVal = "User";
```

// calling the function

```
console.log(welcomeMsg(nameVal));
```

Output

Hello User welcome

Functions in JavaScript

Function Invocation

- The function code you have written will be executed whenever it is called.
- Triggered by an event (e.g., a button click by a user).
- When explicitly called from JavaScript code.
- Automatically executed, such as in self-invoking functions.

Function Expression

- It is similar to a function declaration without the function name. Function expressions can be stored in a variable assignment.

Syntax:

```
let val= function(paramA,  
paramB) { // Set of statements }
```

Ex,

```
const mul = function (x, y) {  
return x * y;  
};  
console.log(mul(4, 5));
```

Output

2

Functions in JavaScript

Nested Functions

Functions defined within other functions are called nested functions. They have access to the variables of their parent function.

```
function outerFun(a) { function innerFun(b) {  
return a + b; }  
return innerFun; }  
const addTen = outerFun(10); console.log(addTen(5));
```

Output

15

Functions in JavaScript

Callback Functions

- A callback function is passed as an argument to another function and is executed after the completion of that function.

```
function num(n, callback) { return callback(n);  
}
```

```
const double = (n) => n * 2; console.log(num(5, double));
```

Output

10

Anonymous Functions

- Anonymous functions are functions without a name. They are often used as arguments to other functions.

```
let show = function() { console.log('Anonymous function');  
};  
show();
```

Functions in JavaScript

VARIABLE SCOPE

- The location where you declare a variable will affect where it can be used within your code. If you declare it within a function, it can only be used within that function. This is known as the variable's *scope*.

LOCAL VARIABLES

- When a variable is created inside a function using the `var` keyword, it can only be used in that function. It is called a local variable or function-level variable.
- It is said to have local scope or function-level scope. It cannot be accessed outside of the function in which it was declared. Below, `area` is a local variable.

Functions in JavaScript

- **GLOBAL VARIABLES**

- If you create a variable outside of a function, then it can be used anywhere within the script. It is called a global variable and has global scope.
- In the example shown, wallSize is a global variable.

```
function getArea(width, height) {  
    var area = width * height;  
    return area;  
}  
  
var wallSize = getArea(3, 2);  
document.write(wallSize);
```

Functions in JavaScript

How Memory & Variables Work

- **Global variables** stay in the browser's memory **as long as the web page is open** and uses more memory.
- **Local variables** only exist **while the function runs** and use less memory.

Memory usage

- Every variable you make takes up memory.
- More variables means more memory and it slower performance.
- The same value can be stored in more than one variable (e.g., two different variables can both be true).

```
var width = 15;    // 15
var height = 30;  // 30
var isWall = true; // true
var canPaint = true; // true
```

Functions in JavaScript

Naming collisions

- If two scripts (JavaScript files) both have a global variable with the same name, they can overwrite each other and causes bugs.

Example:

- Script 1 has `var width = 3;`
- Script 2 also has `var width = 25;`
- They fight over the same variable name.

Use function scope for variables declared inside functions don't interfere with each other.

```
// Show size of the building plot
function showPlotSize(){
  var width = 3;
  var height = 2;
  return 'Area: ' + (width * height);
}
var msg = showArea();
```

```
// Show size of the garden
function showGardenSize() {
  var width = 12;
  var height = 25;
  return width * height;
}
var msg = showGardenSize();
```

Objects in JavaScript

An object is like a box that stores related information and actions together.

Properties

The information about the object (like the object's facts or details).
Example: A hotel's name, number of rooms, location.

Methods

The actions the object can do (like the object's abilities). Example: A hotel can check room availability or book a room.

Objects in JavaScript

```
var hotel = {  
  name: 'Quay',      // string  
  rooms: 40,        // number  
  booked: 25,       // number  
  gym: true,        // boolean  
  roomTypes: ['twin', 'double', 'suite'], // array  
  checkAvailability: function() { // method  
    return this.rooms - this.booked;  
  }  
};
```

Properties

information about the object (key/value pairs).

Example:

name: 'Quay' ; name is the key, 'Quay' is the value

rooms: 40 ; rooms is the key, 40 is the value

Methods

actions the object can perform (key is the method name, value is a function).

Example:

checkAvailability : runs code to find available rooms.

Objects in JavaScript

Important Note:

- Keys are like labels to find values.
- An object can't have two keys with the same name.
- A property's value can be text, number, true/false, array, or another object.
- A method's value is always a function.

CREATING AN OBJECT: LITERAL NOTATION

- Literal notation is the easiest and most popular way to create objects.
- The object is inside curly braces { } and their contents.
- The object is stored in a variable called hotel, so you would refer to it as the hotel object.
- Separate each key from its value using a colon.
- Separate each property and method with a comma except the last value.

CREATING AN OBJECT: LITERAL NOTATION

In the `checkAvailability()` method, the `this` (this means "use the properties from this object.") keyword is used to indicate that it is using the `rooms` and `booked` properties of this object.

```
var hotel = {  
  name: 'Quay',  
  rooms: 40,  
  booked: 25,  
  
  checkAvailability: function() {  
    return this.rooms - this.booked;  
  }  
};
```

ACCESSING AN OBJECT AND DOT NOTATION

- You access the properties or methods of an object using dot notation. You can also access properties using square brackets.

objectName.propertyName or **objectName.methodName()**

- The dot (.) is called the member operator, it means **look inside this object for something**.

```
var hotelName = hotel.name; // get the hotel's name property
```

```
var roomsFree = hotel.checkAvailability(); // run the method
```

- You can also use **square brackets** to get a property:

```
var hotelName = hotel['name'];
```

CREATING MORE OBJECT LITERALS

- This example is just another hotel object, but with different details.
- The properties (name, rooms, booked) have new values for the Park hotel.
- The method (checkAvailability) stays the same, it still calculates free rooms.
- The HTML code updates automatically because it uses the same instructions, just with new data from the object.

```
var hotel = {  
  name: 'Park',           // new hotel name  
  rooms: 120,             // bigger hotel  
  booked: 77,             // rooms already booked  
  checkAvailability: function() {  
    return this.rooms - this.booked;  
  }  
};  
  
document.getElementById('hotelName').textContent = hotel.name;  
document.getElementById('rooms').textContent = hotel.checkAvailability();
```

CREATING MORE OBJECT LITERALS

- If you had 1,000 hotels, you don't have to rewrite the whole program, you just change the property values for each hotel. The rest of the code works the same.
- If you want more than one hotel on the same page, just make another object with a different variable name (e.g., hotel1, hotel2).

```
var hotel = {  
  name: 'Park',           // new hotel name  
  rooms: 120,             // bigger hotel  
  booked: 77,             // rooms already booked  
  checkAvailability: function() {  
    return this.rooms - this.booked;  
  }  
};  
  
document.getElementById('hotelName').textContent = hotel.name;  
document.getElementById('rooms').textContent = hotel.checkAvailability();
```

CREATING AN OBJECT: CONSTRUCTOR NOTATION

- Constructor notation is another way to make an object in JavaScript.
- First, create an empty object using

```
var hotel = new Object();
```

Then, add properties and methods one by one using dot notation

```
hotel.name = 'Quay';
```

```
hotel.rooms = 40;
```

```
hotel.booked = 25;
```

```
hotel.checkAvailability = function() { return this.rooms this.booked;};
```

CREATING AN OBJECT: CONSTRUCTOR NOTATION

- Literal notation way to make an empty object:

```
var hotel = {};
```

difference:

Literal notation

You can create the object and add all properties at once.

Constructor notation

You start with an empty object and then add properties/methods step by step

Updating an Object

You can change an object's property value using:

Dot notation:

```
hotel.name = 'Park';
```

Square brackets:

```
hotel['name'] = 'Park';
```

- If the property already exists in the object, JavaScript updates its value.
- If the property does not exist, JavaScript will create a new property in that object and assign the value to it.

Delete a property:

```
delete hotel.name;
```

Clear a property's value (but keep the property):

```
hotel.name = "";
```

Creating Many Objects with Constructor Functions

If you want **lots of similar objects** (e.g., many hotels), you can make a **template** using a special function called a **constructor**.

```
function Hotel(name, rooms, booked) {  
    this.name = name;  
    this.rooms = rooms;  
    this.booked = booked;  
    this.checkAvailability = function() {  
        return this.rooms - this.booked;  
    };  
}
```

`this` means “the property/method belongs to the object that’s being created.”

Use `new` to make a new object.

```
var quayHotel = new Hotel('Quay', 40, 25);  
var parkHotel = new Hotel('Park', 120, 77);
```

The constructor is the stamp, and each new object is a stamped copy with different details.

ARRAYS ARE OBJECTS

- Arrays are actually a special type of object. They hold a related set of key/value pairs (like all objects), but the key for each value is its index number.

ARRAYS OF OBJECTS & OBJECTS IN ARRAYS

- You can combine arrays and objects to create complex data structures:
- Arrays can store a series of objects and remember their order.
- Objects can also hold arrays as values of their properties.

ARRAYS IN AN OBJECT

The property of any object can hold an array.

On the left, each item on a hotel bill is stored separately in an array.

To access the first charge for room1, you would use:

```
costs.room1.items[0];
```

Ex:

room1 items[420, 40, 10]

room2 items[460, 20, 20]

room3 items[230, 0, 0]

room4 items[620, 150, 60]

PROPERTY:	VALUE:
room1	items[420, 40, 10]
room2	items[460, 20, 20]
room3	items[230, 0, 0]
room4	items[620, 150, 60]

OBJECTS IN AN ARRAY

- The value of any element in an array can be an object ,written using the object literal syntax.
- Here, to access the phone charge for room three, you would use:

```
costs[2].phone;
```

INDEX NUMBER:	VALUE:
0	{accom: 420, food: 40, phone: 10}
1	{accom: 460, food: 20, phone: 20}
2	{accom: 230, food: 0, phone: 0}
3	{accom: 620, food: 150, phone: 60}

BUILT-IN OBJECTS

THREE GROUPS OF BUILT-IN OBJECTS

JavaScript has three main groups of built-in objects, and each group gives you different tools to help you create and control web pages.

- **1. BROWSER OBJECT MODEL**

The Browser Object Model contains objects that represent the current browser window or tab. It contains objects that model things like browser history and the device's screen.

- **2. DOCUMENT OBJECT MODEL**

The Document Object Model uses objects to create a representation of the current page. It creates a new object for each element (and each individual section of text) within the page.

- **3. GLOBAL JAVASCRIPT OBJECTS**

The global JavaScript objects represent things that the JavaScript language needs to create a model of. For example, there is an object that deals only with dates and times.

The **DOM model** (Document Object Model)

The **DOM model** (Document Object Model) is a **programming interface** that represents a web page so that programs (like JavaScript) can interact with it. It turns the **HTML or XML document** into a structured, tree-like model that can be accessed and modified dynamically.

- When a web page loads in the browser, the browser parses the HTML and creates the DOM tree.
- Each element, attribute, and piece of text in the HTML becomes a node in this tree.
- JavaScript can use the DOM to add, remove, or change elements and their content, styles, or attributes while the page is running.

The **DOM model** (Document Object Model)

Structure of the DOM Model:

Document Node → The root of the tree (the entire HTML document).

Element Nodes → Represent HTML tags (like <p>, <div>, <h1>).

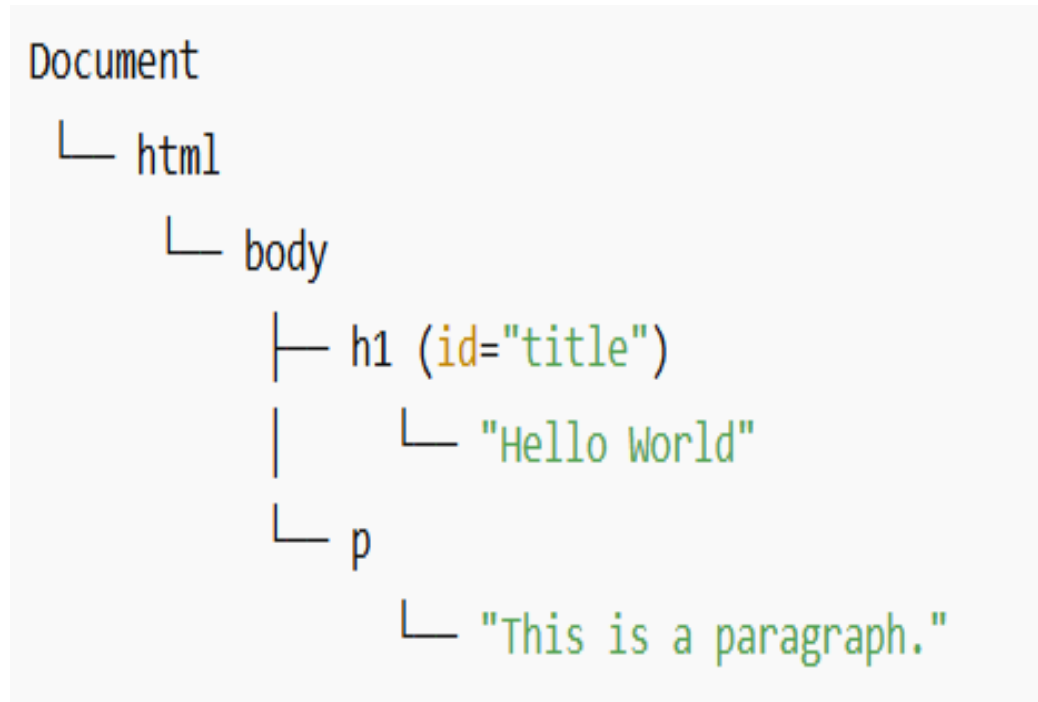
Attribute Nodes → Represent attributes of elements (like id, class, href).

Text Nodes → Represent the actual text inside an element.

The **DOM model** (Document Object Model)

Structure of the DOM Model:

```
<html>
  <body>
    <h1 id="title">Hello World</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```



The **DOM model** (Document Object Model)

How JavaScript uses the DOM:

// Accessing an element

```
let heading = document.getElementById("title");
```

// Changing its content

```
heading.innerHTML = "Welcome!";
```

// Adding new element

```
let para = document.createElement("p");
```

```
para.textContent = "New text added.";
```

```
document.body.appendChild(para);
```

The **DOM model** (Document Object Model)

DOM Manipulation

- **DOM manipulation** means using JavaScript to interact with and change the structure, style, or content of a webpage after it has loaded in the browser.
- Since the **DOM** represents the web page as a tree of nodes, JavaScript can add, remove, or modify these nodes at any time.

Ways to Manipulate the DOM

1. Selecting Elements

To change anything in the DOM, we first need to select an element.

```
document.getElementById("idName");    // Select by id
document.getElementsByClassName("cls"); // Select by class
document.getElementsByTagName("p");    // Select by tag
document.querySelector("div");        // Select first match
document.querySelectorAll("p");       // Select all matches
```

Ways to Manipulate the DOM

Changing Content

```
document.getElementById("title").innerHTML = "New Heading";  
document.querySelector("p").textContent = "Updated paragraph text.";
```

Changing Attributes

```
let img = document.querySelector("img");  
img.setAttribute("src", "newimage.jpg"); // Change image source
```

Changing Styles

```
let box = document.getElementById("box");  
box.style.color = "red";  
box.style.backgroundColor = "yellow";  
box.style.fontSize = "20px";
```

Ways to Manipulate the DOM

Adding and Removing Elements

// Create a new paragraph

```
let newPara = document.createElement("p");
```

```
newPara.textContent = "This is a new paragraph.";
```

// Add it to the body

```
document.body.appendChild(newPara);
```

// Remove an element

```
let oldPara = document.querySelector("p");
```

```
oldPara.remove();
```

Ways to Manipulate the DOM

Event-Based Manipulation

DOM manipulation often happens in response to events like clicks, input, or mouse movements.

```
let btn = document.getElementById("myBtn");  
btn.addEventListener("click", function() {  
  document.body.style.backgroundColor = "lightblue";  
  alert("Background changed!");});
```

DOM – Document Object Model

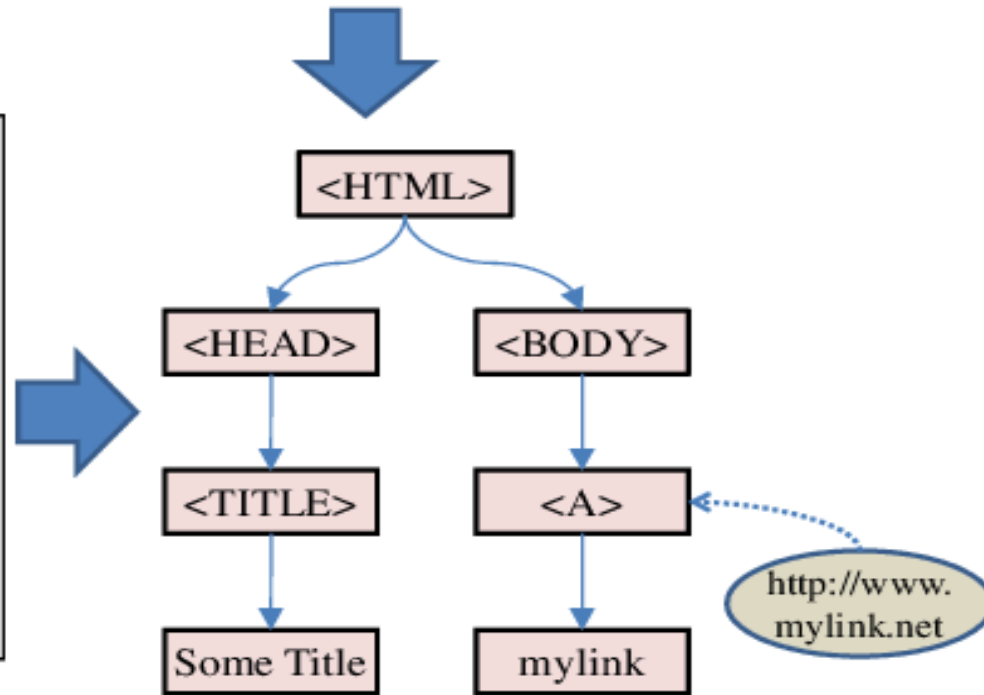
- When a web page is loaded, the browser creates a **Document Object Model** of the page.
- The **HTML DOM** model is constructed as a tree of **Objects**.
- The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:
 - The HTML elements as **objects**
 - The **properties** of all HTML elements
 - The **methods** to access all HTML elements
 - The **events** for all HTML elements

HTML5, JQuery and Ajax

DOM – Document Object Model

```
<html>  
<head> <title>Some Title</title> <!-- some text --> </head>  
<body> <a href=http://www.mylink.net>mylink</a></body>  
</html>
```

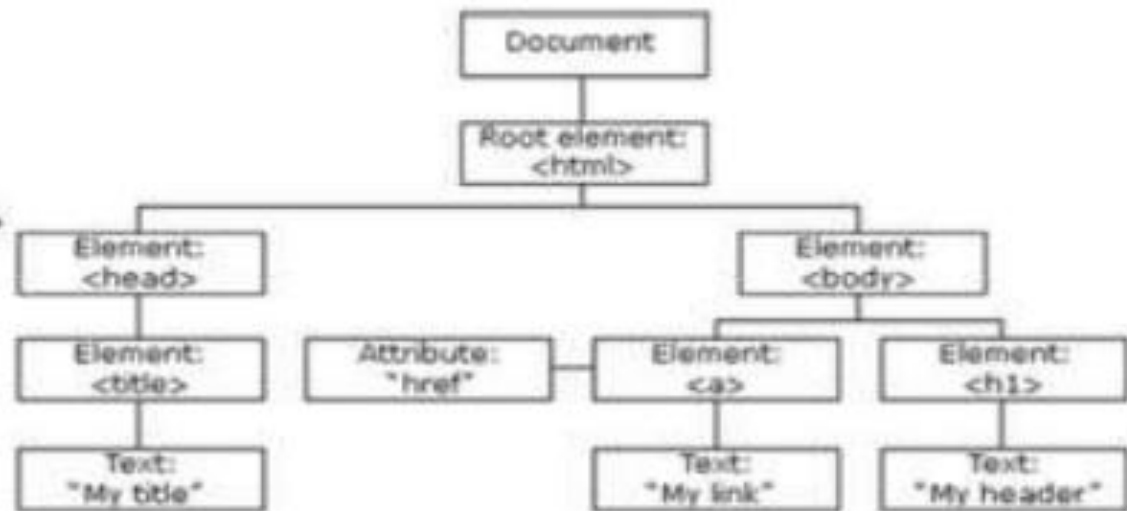
```
<HTML>  
<HEAD>  
<TITLE>Some Title  
</TITLE>  
<!-- other text -->  
</HEAD>  
<BODY>  
<A HREF=http://www.mylink.net>  
mylink</A>  
</BODY>  
</HTML>
```



DOM – Document Object Model

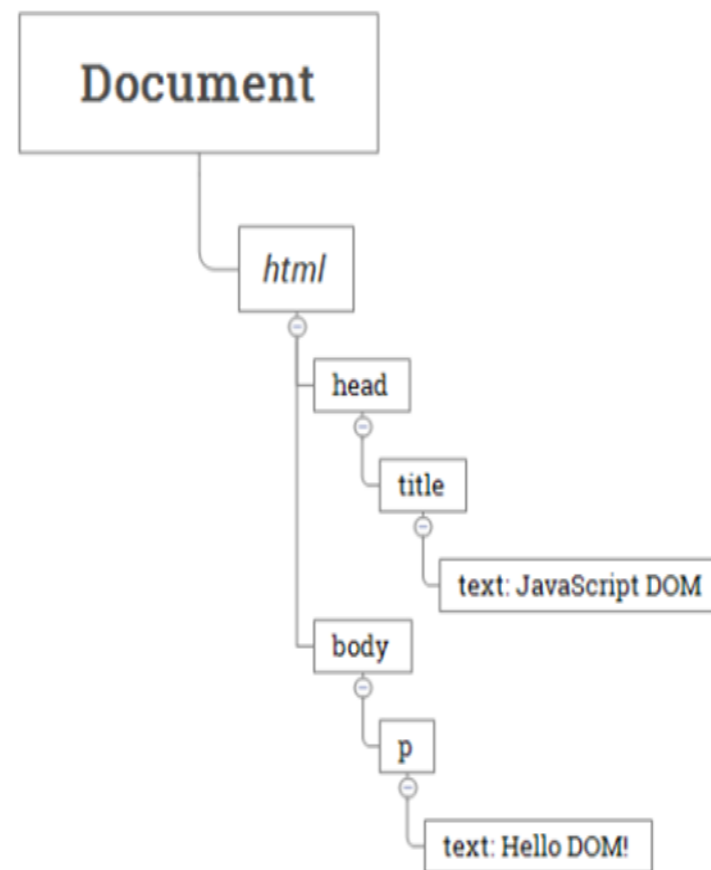
- The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.
- The DOM is not:
 - part of the JavaScript language
 - constructed from the browser
 - is globally accessible by JavaScript code using the document object

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="href">My link</a>
    <h1>My header</h1>
  </body>
</html>
```



DOM – Tree Representation

```
<html>
  <head>
    <title>JavaScript DOM</title>
  </head>
  <body>
    <p>Hello DOM!</p>
  </body>
</html>
```



In this DOM tree, the document is the root node. The root node has one child which is the `<html>` element. The `<html>` element

Document Object Model

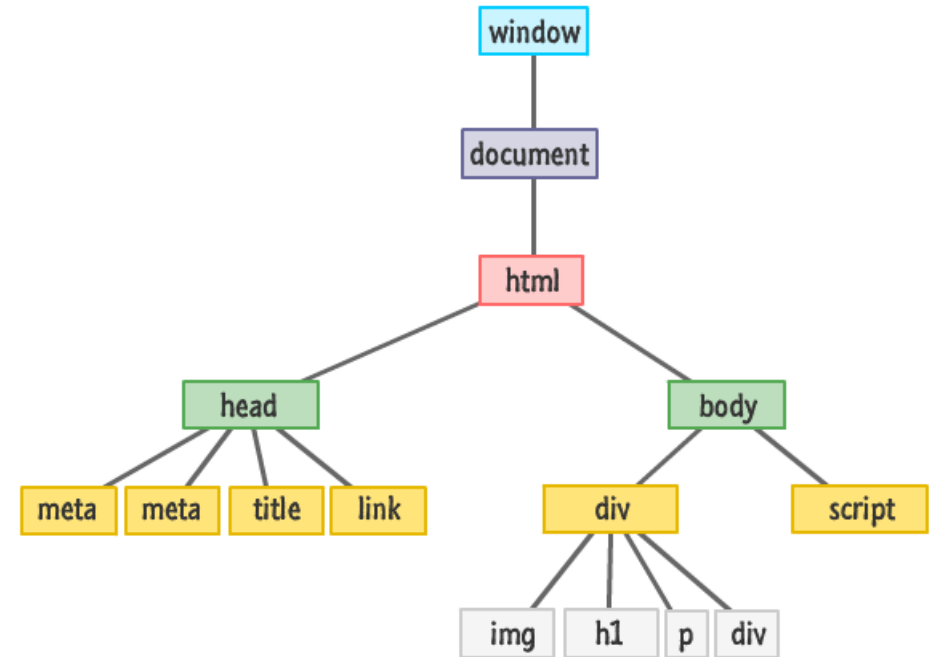
W3C Document Object Model (DOM)

- *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*
- Document.write executed after the page has finished loading will overwrite the page, or write a new page, or not work
- Document.write practically only appending to the page

Document Object Model

Introduction to DOM

- A Web page is a document. This document can be either displayed in the browser window or as the HTML source. But it is the same document in both cases.
- The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

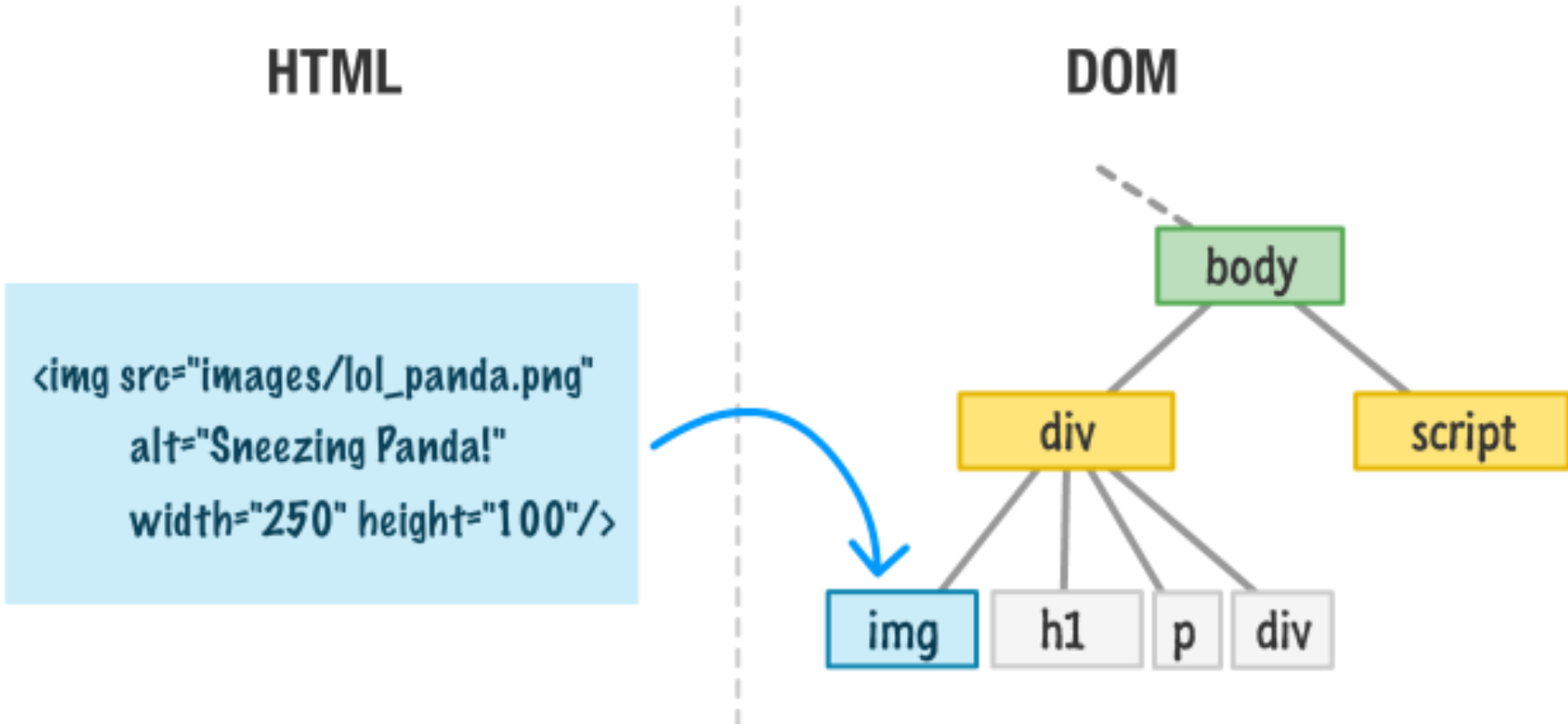


DOM

- Objects have properties and methods, and respond to events.
 - Properties – specify attributes or characteristic of object .
 - Methods – specify functions object can perform.
 - Events – methods corresponding to user actions.

Document Object Model

DOM Elements are Objects



DOM

Accessing DOM

- **write("string")**: writes the given string on the document.
- **getElementById()**: returns the element having the given id value.
- **getElementsByName()**: returns all the elements having the given name value.
- **getElementsByTagName()**: returns all the elements having the given tag name.

Document Object Model

Accessing Elements in DOM

Access Element By	Equivalent Selector	Method
ID	#demo	<code>getElementById("demo")</code>
Class	.demo	<code>getElementsByClassName("demo")</code>
Tag	<tag name> like p	<code>getElementsByTagName("p")</code>
Selector (single)	Any CSS Selector	<code>querySelector("selector")</code>
Selector (all)		<code>querySelectorAll("selector")</code>

getElementById()

- The **document.getElementById()** method returns the element of specified id.
- The parameter of *getElementById* can be any expression that evaluates to a string.

syntax-

```
document.getElementById("#id");
```

getElementsByTagName ()

- *getElementsByTagName* is used to access elements and attributes using tag name.
- This method will return an array of all the items with the same tag name as a NodeList object.

syntax-

```
document.getElementsByTagName(tagname)
```

getElementsByTagName()

- The *getElementsByTagName()* method returns a collection of all elements in the document with the specified name (the **value** of the name attribute), as a NodeList object.

Syntax-

```
document.getElementsByTagName (name) ;
```

Document.querySelector()

- The Document method **querySelector()** returns the first Element within the document that matches the specified selector, or group of selectors.
- If no matches are found, null is returned.

Syntax-

```
element = document.querySelector(selectors);
```

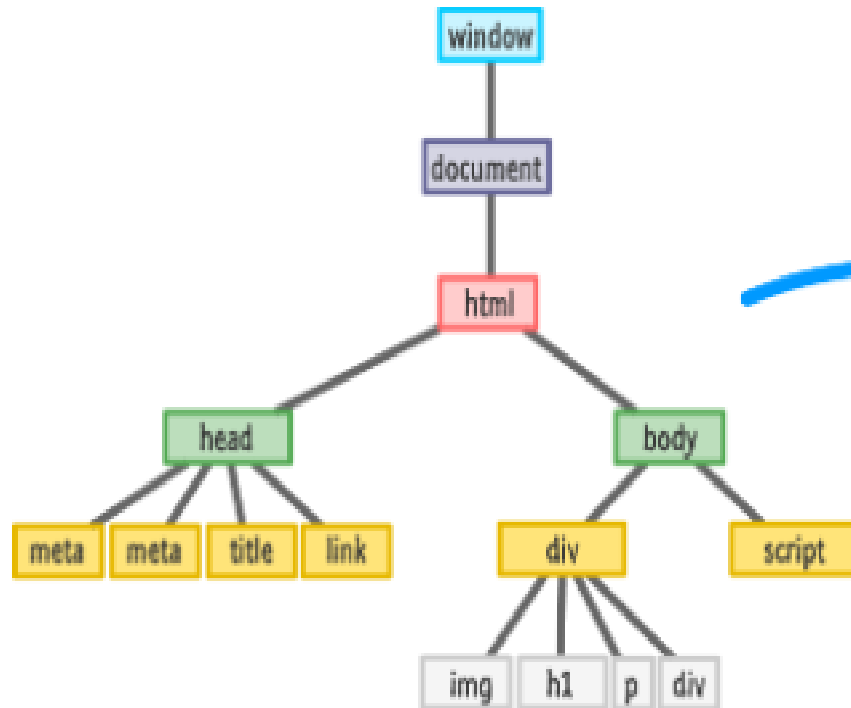
Document.querySelectorAll()

- The Document method **querySelectorAll()** returns a static (not live) NodeList representing a list of the document's elements that match the specified group of selectors.
- If no matches are found, null is returned.

Syntax-

```
elementList = parentNode.querySelectorAll(selectors);
```

Traversing the DOM

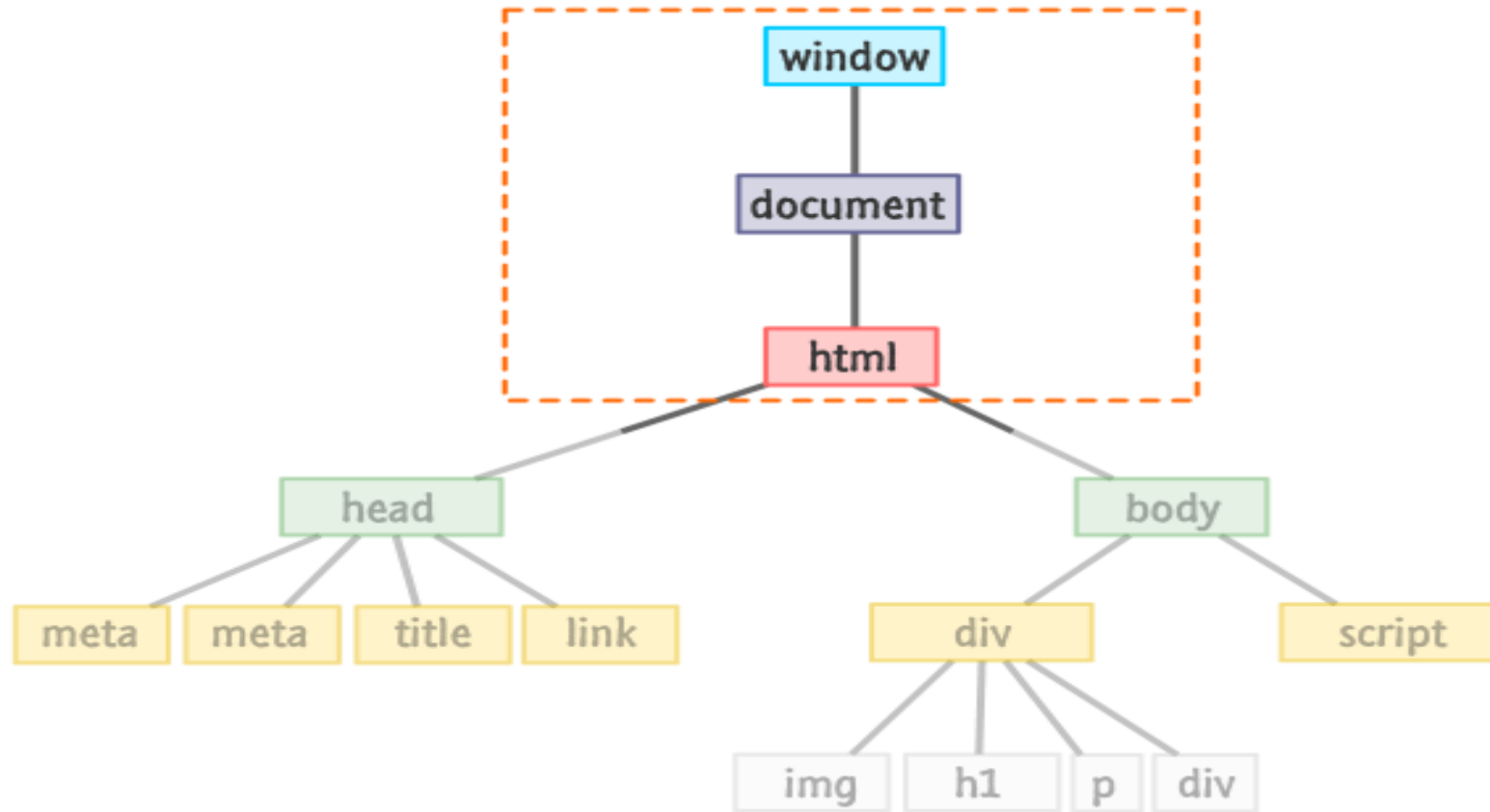


The DOM

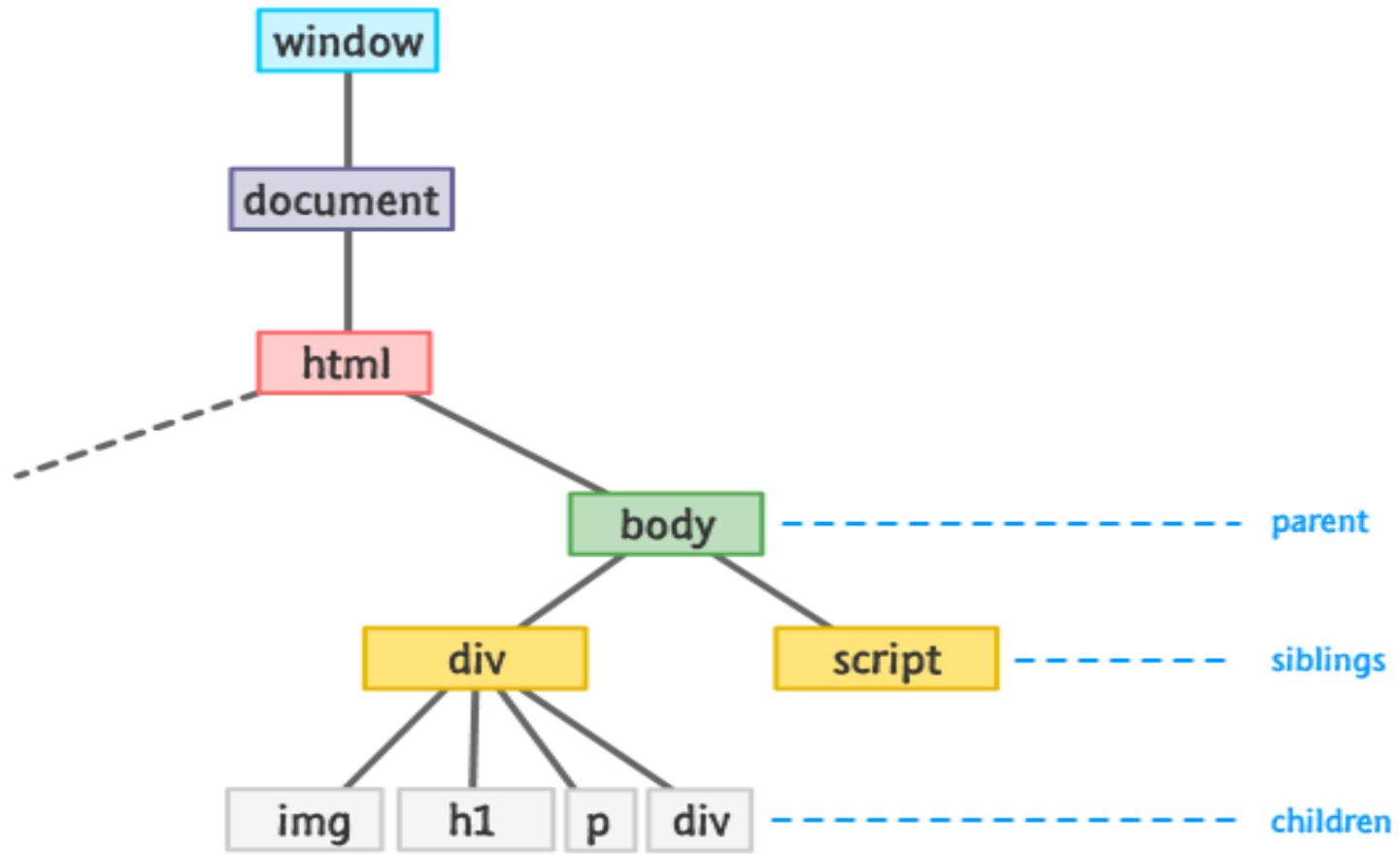


The Browser
(aka what you see)

Traversing the DOM

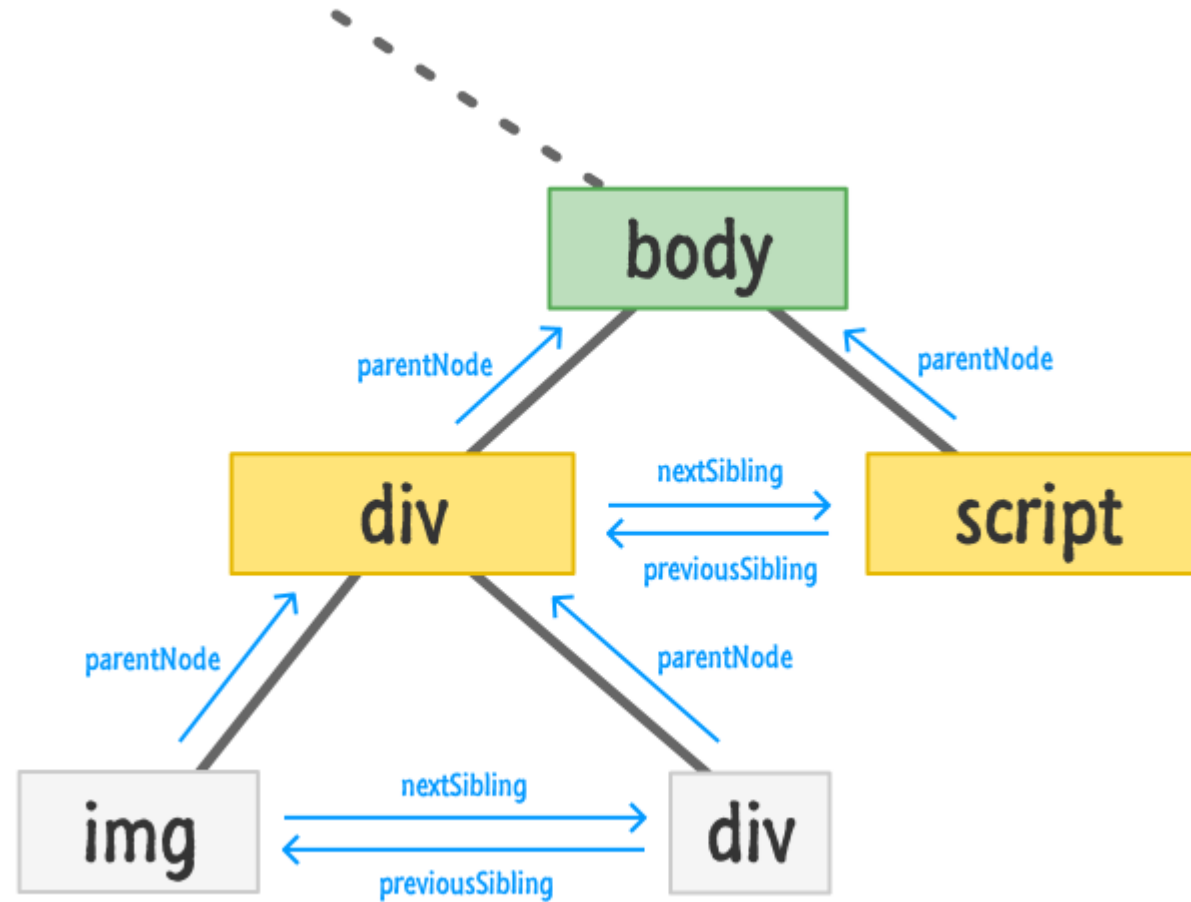


Traversing the DOM



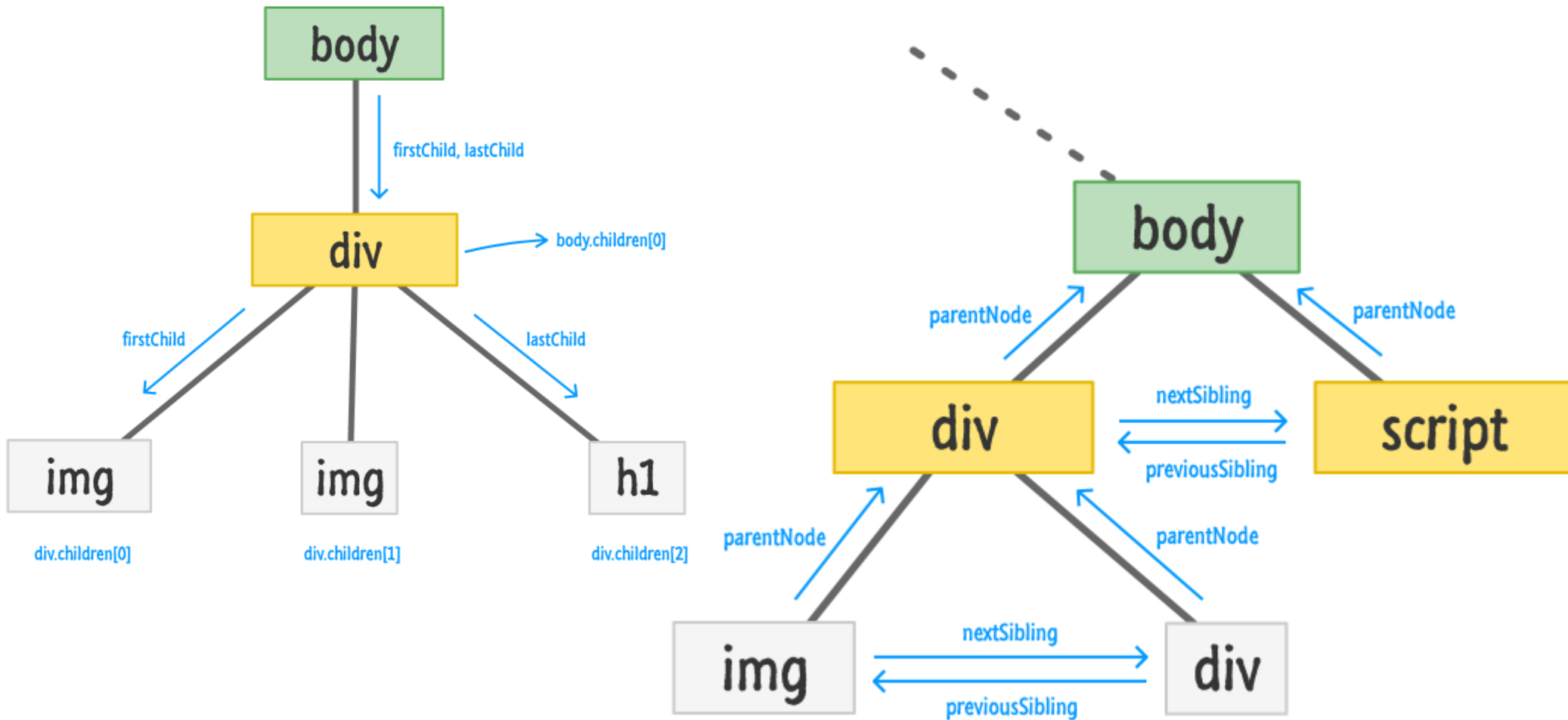
Document Object Model

Traversing the DOM



Document Object Model

Traversing the DOM



Document Object Model

Creating Element Objects

Method		Description
<code>document.createElement()</code>		Create a new element node using tag
<code>document.createTextNode()</code>		Create a new text node
Node property		Description
<code>node.textContent</code> or <code>node.innerText</code>		Get or set the text content of an element node (without HTML tags)
<code>node.innerHTML</code>		Get or set the HTML content enclosed in the element tag

Document Object Model

Manipulating Nodes in the DOM

Method	Description
<code>node.appendChild()</code>	Add a node as the last child of the parent element.
<code>node.insertBefore()</code>	Insert a node into the parent before a specific sibling node
<code>node.replaceChild()</code>	Replace an existing node with a new node
<code>node.removeChild()</code>	Removes child node
<code>node.remove()</code>	Removes a node

* **node** here can be `document.body` or any existing element in the DOM

Events

Events

- An **event** is an action or occurrence that happens in the browser, which JavaScript can respond to.
Examples: clicking a button, typing in a text box, loading a page, moving the mouse, etc.
- When an event happens, we can write JavaScript code to “listen” and **react** to it this is called **event handling**.

Events

Different Types of Events

Mouse Events

click → When the user clicks an element.

dblclick → Double-click.

mouseover → When the mouse pointer moves over an element.

mouseout → When the mouse leaves an element.

mousemove → When the mouse is moved.

Events

2. Keyboard Events

keydown → When a key is pressed down.

keyup → When a key is released.

keypress → When a key is pressed.

3. Form Events

submit → When a form is submitted.

change → When the value of an input element changes.

focus → When an input field gets focus (clicked into).

blur → When an input field loses focus.

Events

4. Window Events

load → When the page has fully loaded.

resize → When the window is resized.

scroll → When the user scrolls the page.

5. Touch & Mobile Events

touchstart → When a finger touches the screen.

touchend → When the finger leaves the screen.

touchmove → When the finger moves on the screen.

Mutation Events in the DOM

Mutation events were a way to detect when the structure of the DOM (HTML elements) **changed**.

For example:

- A new element was added.
- An element was removed.
- An attribute was changed.

Some common mutation events were:

DOMNodeInserted → The event is triggered when a new node was added.

DOMNodeRemoved → The event is triggered when a node was removed.

DOMAttrModified → The event is triggered when an attribute changed.

DOMCharacterDataModified :The event is triggered when the text content inside a node changed.

Binding an event to an element

In JavaScript, **binding an event to an element** means attaching some code that will run when a particular event happens on that element.

1. Inline Event Binding

You directly add the event inside the HTML tag.

```
<button onclick="alert('Button clicked!')">Click Me</button>
```

Here, when the button is clicked, it calls the `alert()` function.

Binding an event to an element

2. Using DOM Element Properties

You access the element in JavaScript and assign a function to its event property.

```
<button id="myBtn">Click Me</button>
```

```
<script>
```

```
let btn = document.getElementById("myBtn");
```

```
btn.onclick = function() { alert("Button clicked!"); };
```

```
</script>
```

- Only assign one function at a time to `.onclick`. If you assign another one, it will overwrite the previous.

Binding an event to an element

3. Using `addEventListener()` : - attach an event listener to an element.

```
<button id="myBtn">Click Me</button>
```

```
<script>
```

```
let btn = document.getElementById("myBtn");
```

```
// Bind event with addEventListener
```

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

```
// You can bind multiple functions to the same event
```

```
btn.addEventListener("click", function() {  
    console.log("Another function runs too!");  
});
```

Binding an event to an element

3. Using `addEventListener()`

- Can bind multiple functions to the same event.
- More flexible here you can remove listeners later using `removeEventListener`.

Event Delegation

- Event delegation is a **JavaScript technique** where instead of attaching an event listener to **multiple child elements**, you attach a single listener to their **parent element**.
- The parent listens for events that "bubble up" from its children, and you decide which child triggered it.

This is useful when:

- You have **many child elements** for e.g., 100 buttons.
- Child elements are **created dynamically** after page load.

Event Delegation

- When something happens like a click on a child element, the event **doesn't stop there**.
- It travels **upwards through its parent**, then grandparent, and so on, until it reaches **the top (document)**, This is called **event bubbling**.
- So if you click on a button inside a div, the button handles the event first, then the div can also "catch" it, and even the whole document can catch it too.

Event Delegation

Example: **Without Delegation**

```
<ul> <li>Apple</li>  
<li>Banana</li>  
<li>Cherry</li></ul>
```

```
<script>
```

```
let items = document.querySelectorAll("li");  
items.forEach(function(item) {  
item.addEventListener("click", function() {  
alert("You clicked on " + item.innerText);  });  
});
```

```
</script>
```

Event Delegation

```
<ul id="fruitList">
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
<script>
  let list = document.getElementById("fruitList");
  // Attach one event listener to parent <ul>
  list.addEventListener("click", function(event) {
    if (event.target.tagName === "LI") {
      alert("You clicked on " + event.target.innerText);
    }
  });
</script>
```

Event Listener

An event listener is like a “**watcher**” attached to an element in the DOM. It **waits for something to happen like a click, key press, mouse hover, etc.** and then **runs some code when that event occurs.**

Syntax:

```
element.addEventListener("event", function);
```

- **element** refers to The DOM element like a button or div.
- **event** refers the type of event like "click", "mouseover", "keydown", etc.
- **function** refers to The code or function that should run when the event happens.

Event Listener

```
<!DOCTYPE html>
<html>
<body>
  <button id="myBtn">Click Me!</button>
  <script>
    // Get the button element
    let btn = document.getElementById("myBtn");
    // Add an event listener to it
    btn.addEventListener("click", function() {
      alert("Button was clicked!");
    });
  </script>
</body>
</html>
```

Advantage of
addEventListener()
over using onclick:

1. You can add multiple listeners to the same element for the same event.
2. You can also remove listeners if needed.

Properties of DOM elements in javascript.

- They belong to the HTML`Element` interface, which is part of the DOM (Document Object Model) API.
- That means any HTML element node you select using JavaScript like with `getElementById`, `querySelector`, etc. will have these properties.

1. From Node Interface

These are available to all nodes (elements, text nodes, comments, etc.).

textContent → Gets/sets all text inside a node (including hidden).

nodeValue → Gets/sets the value of a text node or attribute node.

nodeName → Tag name for element nodes (DIV, P),

nodeType → Returns numeric type (1 = element, 3 = text, 8 = comment, etc.).

childNodes → `NodeList` of all children (elements, text, comments).

firstChild / **lastChild** → First/last child node.

Properties of DOM elements in javascript.

```
<p id="p1">Hello <b>World</b></p>
```

```
<script>
```

```
let p = document.getElementById("p1");
```

```
console.log(p.nodeName); // "P"
```

```
console.log(p.nodeType); // 1 (element)
```

```
console.log(p.firstChild); // "Hello " (text node)
```

```
</script>
```

Properties of DOM elements in javascript.

2. From Element Interface

- These apply to all elements (<div>, <p>, , etc.).

innerHTML → HTML inside the element.

outerHTML → HTML including the element itself.

children → HTMLCollection of child elements only not text nodes.

firstElementChild / **lastElementChild** → First/last child that is an element.

attributes → NamedNodeMap of all attributes.

id, **className**, **tagName** → Element-specific info.

Properties of DOM elements in javascript.

```
<div id="box" class="red"><p>Hi</p></div>
```

```
<script>
```

```
let div = document.getElementById("box");
```

```
console.log(div.innerHTML); // "<p>Hi</p>"
```

```
console.log(div.outerHTML); // "<div id="box" class="red"><p>Hi</p></div>"
```

```
console.log(div.children[0]); // <p>Hi</p>
```

```
console.log(div.tagName); // "DIV"
```

```
</script>
```

Properties of DOM elements in javascript.

3. From HTML Element Interface

- These apply only to HTML elements

innerText → Visible text, it ignores hidden, considers CSS.

style → Contents within the <style> tag.

title → Tooltip text (title="tip").

hidden → Boolean. You can set it to true/false to hide/show elements.

Properties of DOM elements in javascript.

```
<p id="p2" style="display:none" title="tip">Hidden Text</p>
```

```
<script>
```

```
let p = document.getElementById("p2");
```

```
console.log(p.innerText); // "" (empty, since hidden)
```

```
console.log(p.style.color); // "" (not set)
```

```
console.log(p.title); // "tip"
```

```
</script>
```

Properties of DOM elements in JavaScript.

```
<p id="p2" style="color:blue" title="tip">Visible Text</p>  
<script>  
  let p = document.getElementById("p2");  
  console.log(p.innerText); // "Visible Text" (text is visible now)  
  console.log(p.style.color); // "blue" (style property is set)  
  console.log(p.title);     // "tip" (tooltip text)  
</script>
```

string operations like slicing, concatenation, and replacement

String Slicing

- Slicing means extracting a part of the string using indexes.

In JavaScript, we use `.slice(start, end)` or `.substring(start, end)`.

```
let text = "Hello, World!";
```

```
// Extract substring from index 0 to 4
```

```
console.log(text.slice(0, 5)); // "Hello"
```

```
// Extract from index 7 till end
```

```
console.log(text.slice(7)); // "World!"
```

```
//Extract last 6 characters
```

```
console.log(text.slice(-6)); // "World!"
```

string operations like slicing, concatenation, and replacement

String Concatenation

- Joining two or more strings together.
- We can use + or template literals.

```
let str1 = "Good";
```

```
let str2 = "Morning";
```

// Using + operator

```
let result = str1 + " " + str2;
```

```
console.log(result); // "Good Morning"
```

// Repeating a string

```
console.log("Hi! ".repeat(3)); // "Hi! Hi! Hi! "
```

string operations like slicing, concatenation, and replacement

String Replacement

- Replacing a part of a string with another.
- In JavaScript, `.replace()` or `.replaceAll()` is used.

```
let sentence = "I like Java";
```

```
// Replace "Java" with "Python"
```

```
let newSentence = sentence.replace("Java", "Python");
```

```
console.log(newSentence); // "I like Python"
```

```
let text2 = "apple, apple, orange";
```

```
// Replace only first occurrence
```

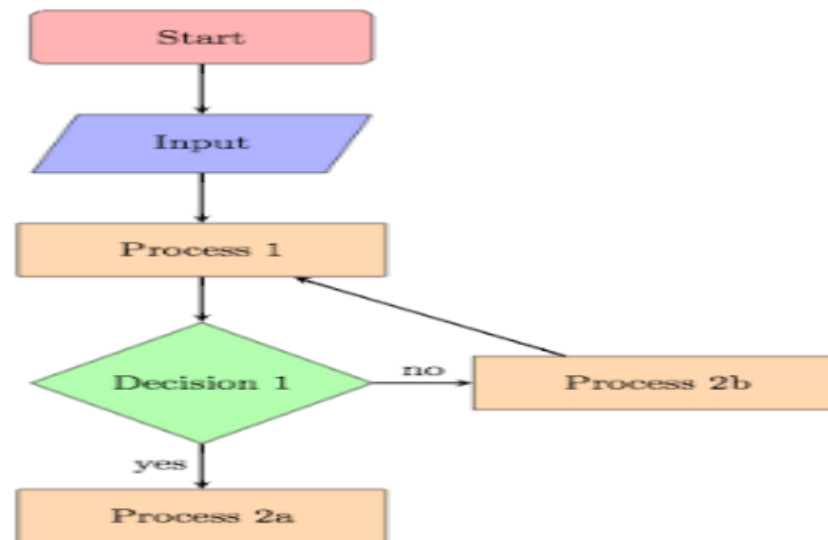
```
console.log(text2.replace("apple", "banana")); // "banana, apple, orange"
```

```
// Replace all occurrences
```

```
console.log(text2.replaceAll("apple", "banana")); // "banana, banana, orange"
```

Decision and Loops

- In programming, sometimes we need to decide which part of the code should run.
- A flowchart can help us plan decisions.
- The diamond shape represents a decision point.
- Each path can go in different directions depending on the condition.



Decision and Loops

This means:

- The program checks a condition.
- If it's true, one block of code runs.
- If it's false, another block runs.

We use comparison operators for conditions:

> greater than

< less than

== equal to

Evaluating Conditions & Conditional Statements

When making a decision in code:

- An **expression** is checked which gives **true** or **false**.
- A **conditional statement** decides what to do next.

```
if (score > 50) {  
    document.write("You passed!");  
} else {  
    document.write("Try again...");  
}
```

Evaluating Conditions & Conditional Statements

Comparison Operators

- They are used to **compare two values** i.e. numbers, strings, or Booleans.
- The result is always **true** or **false**.
- Programmers call this process **evaluating a condition**.

1. **Equal To (==)**

2. **Not Equal To (!=) :**

3. **Strict Equal To (===) :** Checks if both **value and data type** are the same
(`"3" === 3 → false`)

4. **Strict Not Equal To (!===):** Checks if either the value or data type is **different**.
(`3 !== "3" → true`)

5. **Greater Than (>)**

6. **Less Than (<)**

7. **Greater Than or Equal To (>=)**

8. **Less Than or Equal To (<=)**

Structuring Comparison Operators

A condition usually has:

One operator (like \geq , $=$, \neq , $<$, $>$)

Two operand values or variables being compared.

(score \geq pass)

Here:

score \rightarrow operand (the student's marks)

\geq \rightarrow operator (greater than or equal to)

pass \rightarrow operand (the passing marks)

Brackets () are often used to make conditions clear.

Using Expressions with Comparison Operators

- A comparison does not always need to be just single numbers or variables.
- The operands can also be expressions like math operations.
- Example:

((score1 + score2) > (highScore1 + highScore2))

Here:

Left side → (score1 + score2) → adds two scores.

Right side → (highScore1 + highScore2) → adds two high scores.

Operator → > checks which total is bigger.

Logical Operators

- Comparison operators return true or false for example: $5 < 2 \rightarrow \mathbf{false}$.
- Logical operators let you combine or modify these results.

AND (&&)

- Both conditions must be true.

OR (||)

- At least one condition must be true.

NOT (!)

- Reverses the result.

Logical Operators

Short-Circuit Evaluation

- When you write a logical expression (using AND `&&` or OR `||`), JavaScript checks it from left to right.
- If the first part is enough to decide the result, it does not check the second part.

```
let a = true;
```

```
let b = false;
```

```
// Logical AND (&&)
```

```
let andResult = a && b; // true AND false → false
```

```
// Logical OR (||)
```

```
let orResult = a || b; // true OR false → true
```

```
// Logical NOT (!)
```

```
let notResultA = !a; // NOT true → false
```

```
let notResultB = !b; // NOT false → true
```

Conditional Statements in JavaScript

Conditional Statements in JavaScript

- Conditional statements are used to **make decisions** in a program.

They allow the computer to choose **different actions** depending on whether a condition is **true** or **false**.

- The main conditional statements in JavaScript are:
 - **if**
 - **if-else**
 - **switch**

Conditional Statements in JavaScript

IF Statement

The if statement runs a block of code only if the condition is true. If the condition is false, nothing happens.

Syntax:

```
if (condition)
{ // code to run if condition is true }
```

Example:

```
<html><body><h3>IF Example</h3>
<p id="ifDemo"></p>
<script>
let age = 20;
if (age >= 18)
{ document.getElementById("ifDemo").innerHTML = "You are an adult.";}
</script></body></html>
```

Conditional Statements in JavaScript

2. IF-ELSE Statement

The if-else statement runs one block of code if the condition is true, and another block if the condition is false.

Syntax:

```
if (condition) { // code if true }  
else  
{ // code if false }
```

Example:

```
p id="ifElseDemo"></p>  
<script>  
let marks = 35;  
if (marks >= 40) { document.getElementById("ifElseDemo").innerHTML = "Pass ";}  
else { document.getElementById("ifElseDemo").innerHTML = "Fail ✖";}</script>
```

Conditional Statements in JavaScript

3. SWITCH Statement

The switch statement is used when you want to test a variable against many possible values.

Syntax:

```
switch(expression) {  
  case value1:    // code if expression == value1  
    break;  
  case value2:    // code if expression == value2  
    break;  
  ...  
  default:       // code if no match is found}
```

Conditional Statements in JavaScript

```
<script>
```

```
let day = 3;
```

```
let dayName;
```

```
switch(day) {
```

```
  case 1: dayName = "Monday"; break;
```

```
  case 2: dayName = "Tuesday"; break;
```

```
  case 3: dayName = "Wednesday"; break;
```

```
  case 4: dayName = "Thursday"; break;
```

```
  case 5: dayName = "Friday"; break;
```

```
  case 6: dayName = "Saturday"; break;
```

```
  case 7: dayName = "Sunday"; break;
```

```
  default: dayName = "Invalid Day";
```

```
}
```

```
document.getElementById("switchDemo").innerHTML = "Day is: " + dayName;
```

```
</script>
```

Type Coercion & Weak Typing

- JavaScript can automatically change data types behind the scenes to complete an operation.

Example:

```
"1" > 0 // true
```

- Here, the string "1" is converted to a number before comparison.
- This automatic type conversion is called **type coercion**.
- JavaScript is called **weakly typed because** a variable's type can change automatically.

Example:

```
var x = 5; // number
```

```
x = "hello"; // now string
```

In other languages like Java or C++, you must declare the type and it cannot change. That's called **strong typing**.

Type Coercion & Weak Typing

Common Data Types:

- **string** for text
- **number** for numbers
- **boolean** for true or false
- **null** for empty value
- **undefined** for declared variable with no value

Also, **NaN** (Not a Number) happens when you try invalid math,
for e.g.: `"ten" / 2 // NaN`

Truthy & Falsy Values

- Because of type coercion, everything in JavaScript can be treated as true or false.

Falsy Values count as false in conditions:

1. false → the boolean false
2. 0 → number zero
3. "" → empty string
4. null → empty value
5. undefined → variable with no value
6. NaN → Not a Number

```
if (0) {  
  console.log("True");  
} else {  
  console.log("False"); // this runs  
}
```

Truthy & Falsy Values

Truthy Values count as true in conditions:

Almost everything else is truthy.

Examples:

true any non-zero number (1, -5, 100, etc.)

any non-empty string ("hello", "0", "false")

objects { } or arrays []

Example:

```
if ("hello")  
{  
  console.log("This is true!"); // runs  
}
```

Checking Equality & Existence

Existence Check

- Objects (`{ }`) and arrays (`[]`) are always considered true (truthy).
- That means we can put them directly in an if statement to check if they exist.

Example:

```
let user = { name: "Sam" };  
if (user) { console.log("User exists!"); }
```

- Since user is an object, it's truthy → so the message prints.

With an array:

```
let items = [];  
if (items) { console.log("Array exists!"); }
```

Checking Equality & Existence

Equality in JavaScript

JavaScript has two types of equality checks:

== (loose equality) → allows type coercion.

=== (strict equality) → checks value and type .

LOOPS

1. for loop

The for loop is used when you know how many times you want to repeat a block of code.

It has three parts inside the parentheses:

Initialization → start value of the loop counter

Condition → loop runs as long as this condition is true

Update → increases or decreases the counter

Syntax:

```
for(initialization; condition; update)
{ // code to execute }
```

Example:

```
for(let i = 1; i <= 5; i++)
{ console.log("Number is: " + i);} //Prints numbers from 1 to 5.
```

LOOPS

2. while loop

- The while loop is used when you **don't know exactly how many times** to repeat, but want to keep looping **as long as a condition is true**.
- It checks the condition **before** running the loop body.

Syntax:

```
while(condition)
{ // code to execute
}
```

Example:

```
let i = 1;
while(i <= 5) {
  console.log("Number is: " + i);
  i++; // update
}
```

LOOPS

3. do...while loop

- The do...while loop is similar to while, but it **executes at least once** before checking the condition.
- Good for cases where you want the code to run at least one time, no matter what.

Syntax:

```
do {  
  // code to execute  
} while(condition);
```

Example:

```
let i = 1;  
do { console.log("Number is: " + i);  
  i++; }  
while(i <= 5);
```

ARRAY

- An **array** in JavaScript is a special type of object used to **store multiple values in a single variable**.
- It can hold **different data types** (numbers, strings, booleans, objects, functions).
- Array elements are **indexed starting from 0**.

Creating Arrays

Using square brackets [] :

```
let numbers = [10, 20, 30, 40];
```

Using new Array():

```
let colors = new Array("Red", "Green", "Blue");
```

ARRAY

Array Properties

- `length` → gives the number of elements.

```
let arr = [1, 2, 3, 4];
```

```
console.log(arr.length); // 4
```

Array Methods

- JavaScript provides many built-in methods for arrays:

1. Accessing / Changing Elements

```
let cars = ["BMW", "Audi", "Tesla"];
```

```
cars[1] = "Mercedes"; // change element
```

```
console.log(cars); // ["BMW", "Mercedes", "Tesla"]
```

ARRAY

2. Adding / Removing Elements

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

```
// Add at end
```

```
fruits.push("Mango");
```

```
console.log(fruits); // ["Apple", "Banana", "Mango"]
```

```
// Remove last element
```

```
fruits.pop();
```

```
console.log(fruits); // ["Apple", "Banana"]
```

```
// Add at beginning
```

```
fruits.unshift("Orange");
```

```
console.log(fruits); // ["Orange", "Apple", "Banana"]
```

```
// Remove first element
```

```
fruits.shift();
```

```
console.log(fruits); // ["Apple", "Banana"]
```

ARRAY

3. Concatenation

```
let a = [1, 2];
```

```
let b = [3, 4];
```

```
let c = a.concat(b);
```

```
console.log(c); // [1, 2, 3, 4]
```

4. Slicing

```
let nums = [10, 20, 30, 40, 50];
```

```
let part = nums.slice(1, 4); // from index 1 to 3
```

```
console.log(part); // [20, 30, 40]
```

ARRAY

5. Splicing

- **Add/Remove from anywhere**

```
let fruits = ["Apple", "Banana", "Mango"];  
fruits.splice(1, 1, "Grapes"); // remove 1 item at index 1, add "Grapes"  
console.log(fruits); // ["Apple", "Grapes", "Mango"]
```

6. Looping through Arrays

```
let fruits = ["Apple", "Banana", "Mango"];  
// For loop  
for (let i = 0; i < fruits.length; i++)  
{ console.log(fruits[i]);  
}
```

ARRAY (SORTING STRING ELEMENTS)

In JavaScript, arrays have a built-in method `.sort()` which sorts the elements.

By default, `.sort()` converts elements into strings and sorts them in Unicode (alphabetical) order.

If you want case-insensitive sorting, you can use `.toLowerCase()` inside a custom sort function.

Example 1: Simple String Sorting

```
let fruits = ["Banana", "Apple", "Mango", "Cherry"];  
console.log("Original Array: ", fruits); // Original Array: [ 'Banana', 'Apple', 'Mango',  
'Cherry' ]  
  
// Sort in alphabetical order  
fruits.sort(); console.log("Sorted Array: ", fruits); // Sorted Array: [ 'Apple',  
'Banana', 'Cherry', 'Mango' ]
```

ARRAY (SORTING STRING ELEMENTS)

case-insensitive sorting

```
let names = ["john", "Alice", "bob", "David"];  
console.log("Original Array: ", names);  
// Sort without case sensitivity  
names.sort((a, b) => a.toLowerCase().localeCompare(b.toLowerCase()));  
console.log("Case-Insensitive Sorted Array: ", names);
```

OUTPUT:

Original Array: ['john', 'Alice', 'bob', 'David']

Case-Insensitive Sorted Array: ['Alice', 'bob', 'David', 'john']

ARRAY (SORTING STRING ELEMENTS)

Reverse Sorting

```
let cities = ["Delhi", "London", "New York", "Tokyo"];
```

```
console.log("Original Array: ", cities);
```

```
// Sort alphabetically
```

```
cities.sort();
```

```
// Reverse order
```

```
cities.reverse();
```

```
console.log("Reverse Sorted Array: ", cities);
```

Output:

```
Original Array: [ 'Delhi', 'London', 'New York', 'Tokyo' ]
```

```
Reverse Sorted Array: [ 'Tokyo', 'New York', 'London', 'Delhi' ]
```