

Programming 'Javascript'

```
{ [JavaScript Basics:  
Mastering the DOM]
```

```
< JavaScript Fundamentals and the DOM >
```

```
}
```

Contents Of 'Javascript';

JavaScript is a programming language used to make websites interactive and responsive. It allows developers to add functionality like buttons, animations, and forms to web pages.

Why is JavaScript an Interpreted Language?

- * JavaScript is an interpreted language, meaning it doesn't need to be compiled before running.
- * Instead of compiling the entire code into machine language beforehand, like compiled languages such as C++, JavaScript code is executed line-by-line by a JavaScript engine in the web browser.
- * This makes it easier and quicker to write and test code because developers can see immediate results without waiting for a compilation step.

Importance:

- * JavaScript is essential for creating dynamic and engaging websites.
- * Its interpreted nature allows for faster development and easier debugging.
- * JavaScript code is executed by the web browser's JavaScript engine, making it accessible to developers and users alike.

What is 'Console' {

A **console** is a developer tool provided by the web browser, in which we can see the output of our JS code and what action the code is performing. Unlike CSS and HTML, output of JS is not displayed in the main body of this web browser.

```
> function isPalindromeNumber(num) {  
  let str = num.toString();  
  let str1 = str.split("").reverse().join("");  
  return str === str1;  
}  
console.log(isPalindromeNumber(121));  
  
true
```

}

Variables in Javascript {

- Variables are containers for storing values (data)
- Use keywords `var`, `let`, and `const` to declare them
- `let` and `const` are preferred for block-level scoping

```
JavaScript  
  
var oldWay = "This is a variable";  
let preferredWay = "Modern style";  
const fixedValue = 10;
```

}

Concepts < var > {

- * **Scope:** Function-scoped. A variable declared with var is accessible within the entire function where it's declared (or globally if declared outside any function).
- * **Hoisting:** var variables are hoisted to the top of their scope and initialized with undefined. This means you can access them before they are declared, but their value will be undefined before initialization.
- * **Re-declaration:** Can be re-declared with the same name within its scope.

}

Concepts < let > {

- * **Scope:** Block-scoped. A variable declared with let is accessible only within the block where it's declared (e.g., an if statement, a loop, or any block defined by curly braces {}).
- * **Hoisting:** let variables are also hoisted to the top of their scope, but they are not initialized. Trying to access a let variable before its declaration results in a ReferenceError.
- * **Re-declaration:** Cannot be re-declared with the same name within the same block.

}

Concepts of `< const > {`

- * **Scope:** Block-scoped (similar to `let`).
- * **Hoisting:** `const` variables are hoisted, but not initialized (similar to `let`).
- * **Re-declaration:** Cannot be re-declared within the same block.
- * **Re-assignment:** Cannot be reassigned to a new value after declaration. Values within objects and arrays declared with `const` can still be modified.

` Why use 'let' and 'const' over 'var': `

- * **Cleaner Coding:** `let` and `const` have block-level scoping. This means variables are only accessible within the specific code block they're declared in (like an if statement or a loop). This prevents accidental naming conflicts and makes code easier to manage.
- * **Prevent Errors:** `let` and `const` aren't accessible before they are assigned a value. This avoids confusing situations where `var` could be used before being properly initialized.
- * **Predictable Values:** `const` declares variables that cannot be reassigned. This helps maintain clearer data integrity, making it easier to reason about how your application's state works.

`<p>` In short, `let` and `const` promote cleaner, more predictable, and less error-prone JavaScript code! `</p>`

How to display 'Outputs' {

Contents 🖱️

- * `console.log()` for printing to the console
- * `document.write()` for writing directly into the HTML document (use with caution)
- * `alert()` for creating pop-up message boxes
- * Modifying DOM elements to display output

JavaScript

}



JavaScript

```
console.log("This will appear in the console");
alert("This will create a pop-up");
document.getElementById('output').innerHTML = "This changes a page element";
```

'If-else' in JavaScript {

Contents [🔍]

- `if`, `else if`, `else` statements control code execution based on conditions
- Comparison operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
- Logical operators: `&&` (AND), `||` (OR), `!` (NOT)

JavaScript []

```
let age = 25;
if (age >= 18) {
  console.log("You are eligible to vote");
} else {
  console.log("You are not eligible to vote");
}
```


'Loops' in JavaScript{

- **for loop:** Repeats code a fixed number of times
- **while loop:** Repeats while a condition is true
- **do-while loop:** Runs at least once, then repeats while a condition is true

JavaScript

```
JavaScript
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

```
JavaScript
let count = 1;
while (count <= 5) {
  console.log(count);
  count++; // Important to increment, or it
           // becomes an infinite loop!
}
```

```
JavaScript
let input;
do {
  input = prompt("Enter a number:");
} while (isNaN(input));
// Check if the input is not a number
```

Knowing about 'Arrays & Objects & Functions'

ARRAYS

- Used to store ordered lists of data
- Elements accessed using their index (starting from 0)
- Common array methods: push(), pop(), shift(), unshift(), length.

```
JavaScript
const myArray = ["apple", "banana", 30];
myArray.push("orange"); // Add to end
console.log(myArray[1]); // Access 'banana'
```

FUNCTIONS

- Blocks of code that perform a specific task
- Can take parameters as input
- Can optionally return output

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

function calculateArea(length, width) {
  const area = length * width;
  return area;
}
// Calling the function and using the returned value
const roomArea = calculateArea(10, 15);
console.log("The area of the room is:", roomArea);
```

OBJECTS

- Objects store properties (key-value pairs)
- Properties accessed using dot notation (object.property) or bracket notation (object['property'])
- Methods are functions associated with objects

```
JavaScript
const person = {
  firstName: "John",
  lastName: "Doe",
  greet: function() {
    console.log("Hello!");
  }
};
person.greet();
```

Knowing about 'Arrays & Objects & Functions'

ARRAYS

- Used to store ordered lists of data
- Elements accessed using their index (starting from 0)
- Common array methods: push(), pop(), shift(), unshift(), length.

```
JavaScript
const myArray = ["apple", "banana", 30];
myArray.push("orange"); // Add to end
console.log(myArray[1]); // Access 'banana'
```

FUNCTIONS

- Blocks of code that perform a specific task
- Can take parameters as input
- Can optionally return output

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

function calculateArea(length, width) {
  const area = length * width;
  return area;
}
// Calling the function and using the returned value
const roomArea = calculateArea(10, 15);
console.log("The area of the room is:", roomArea);
```

OBJECTS

- Objects store properties (key-value pairs)
- Properties accessed using dot notation (object.property) or bracket notation (object['property'])
- Methods are functions associated with objects

```
JavaScript
const person = {
  firstName: "John",
  lastName: "Doe",
  greet: function() {
    console.log("Hello!");
  }
};
person.greet();
```

Iterating Arrays: 'for of' and 'for in' in JS example {

- * for of: Iterates directly over the values of an array
- * for in: Iterates over the keys (indices) of an array

For of

```
JavaScript

const fruits = ["apple", "banana", "orange"];

for (const fruit of fruits) {
  console.log(fruit);
}

// Output:
// apple
// banana
// orange
```

For in

```
JavaScript

const colors = ["red", "green", "blue"];

for (const index in colors) {
  console.log("Color at index", index, "is", colors[index]);
}

// Output:
// Color at index 0 is red
// Color at index 1 is green
// Color at index 2 is blue
```

Array Methods: map, filter, reduce {

- ❖ **map():** Creates a new array by applying a function to each element of the original array
- ❖ **filter():** Creates a new array containing elements that pass a certain test (condition)
- ❖ **reduce():** Reduces an array to a single value (sum, product, etc.) by applying a function

Map()

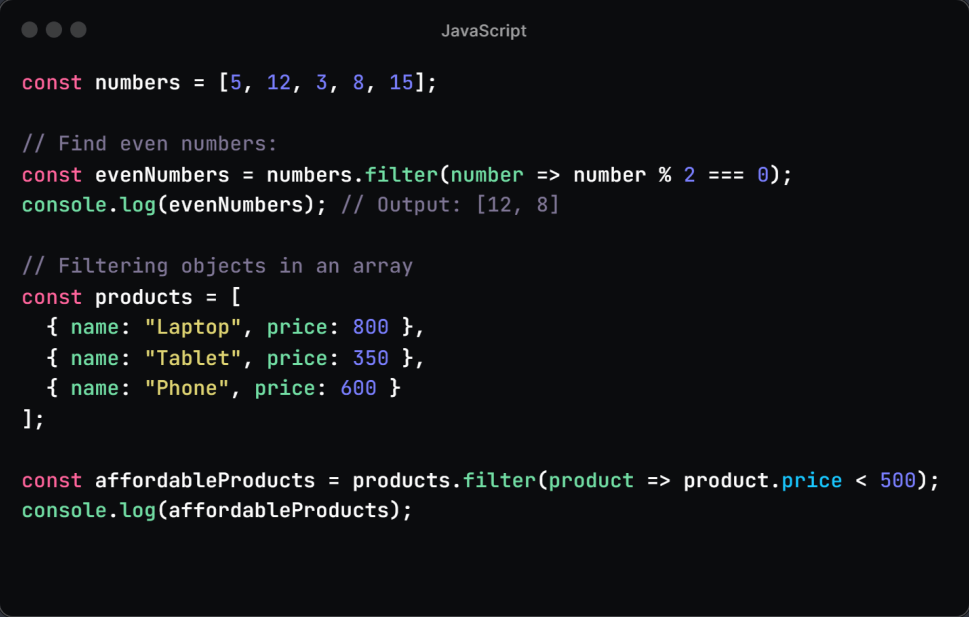
```
JavaScript

const numbers = [1, 2, 3, 4];

// Double each number
const doubledNumbers = numbers.map(number => number * 2);
console.log(doubledNumbers); // Output: [2, 4, 6, 8]

// Convert an array of strings to uppercase
const names = ["Alice", "bob", "Charlie"];
const uppercaseNames = names.map(name => name.toUpperCase());
console.log(uppercaseNames); // Output: ["ALICE", "BOB", "CHARLIE"]
```

```
1 { Filter()       
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```




```
JavaScript

const numbers = [5, 12, 3, 8, 15];

// Find even numbers:
const evenNumbers = numbers.filter(number => number % 2 === 0);
console.log(evenNumbers); // Output: [12, 8]

// Filtering objects in an array
const products = [
  { name: "Laptop", price: 800 },
  { name: "Tablet", price: 350 },
  { name: "Phone", price: 600 }
];

const affordableProducts = products.filter(product => product.price < 500);
console.log(affordableProducts);
```

```
1 { Reduce()  —  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

```
JavaScript  
  
const numbers = [2, 5, 8];  
  
// Sum of all numbers  
const sum = numbers.reduce((accumulator, currentNumber)  
=> accumulator + currentNumber, 0);  
// Note: 0 is the initial value of the accumulator  
console.log(sum); // Output: 15  
  
// Find the maximum number  
const maxNumber = numbers.reduce((accumulator, currentNumber) => {  
  return Math.max(accumulator, currentNumber);  
}, -Infinity);  
console.log(maxNumber); // Output: 8
```

DOM Manipulation {

Accessing Elements:

- ❖ **document.getElementById():** Retrieves an element based on its unique 'id' attribute.
- ❖ **document.getElementsByClassName():** Retrieves a collection of elements with the same 'class' attribute.
- ❖ **document.querySelector():** Retrieves the first element matching a specific CSS selector.
- ❖ **document.querySelectorAll():** Retrieves a collection of elements matching a specific CSS selector.

Modifying Contents:

- ❖ **element.textContent:** Changes the plain text content of an element.
- ❖ **element.innerHTML:** Changes the HTML content (including nested elements) within an element.

}

Basic 'DOM Manipulation' Statements{

}

```
JavaScript

// Access an element
const heading = document.getElementById('main-title');

// Change its text content
heading.textContent = 'Welcome to My Website!';

// Change its color
heading.style.color = 'purple';

// Add a button
const button = document.createElement('button');
button.textContent = 'Click Me!';
document.body.appendChild(button);

// Respond to clicks
button.addEventListener('click', () => {
  alert('You clicked the button!');
});
```

Events and Event Listeners{

Events are signals triggered by user interactions (clicks, hovers, form submits, etc.)

Event listeners: JavaScript functions that execute when a specific event occurs `addEventListener('event type', listenerFunction)`

}

```
JavaScript
<!DOCTYPE html>
<html>
<head>
  <title>Simple Event Example</title>
</head>
<body>
  <button id="myButton">Change Text</button>
  <p id="changeableText">This text will change!</p>
  <script src="script.js"></script>
</body>
</html>
```

```
JavaScript

const button = document.getElementById('myButton');
const paragraph = document.getElementById('changeableText');

function changeText() {
  paragraph.textContent = 'The text has been changed!';
}

button.addEventListener('click', changeText);
```

‘Synchronous’ vs ‘Asynchronous’ JavaScript{

- ❖ **Synchronous:** Operations happen one after another, in sequence. Code blocks until a task completes before moving on.
- ❖ **Asynchronous:** Operations can be delayed or occur without blocking the main thread (e.g., network requests, setTimeout, event listeners).

```
JavaScript

console.log("Synchronous Task 1: Starting");
const result = 10 + 20;
console.log("Synchronous Task 1: Finished. Result:", result);

console.log("Asynchronous Task 2: Starting");

// Simulate a delayed operation like a network request
setTimeout(function() {
  console.log("Asynchronous Task 2: Finished after delay");
}, 3000); // 3 second delay

console.log("Synchronous Task 3: Starting");
console.log("Synchronous Task 3: Finished");
```

Callbacks, Promises {

CALLBACKS

- Functions passed as arguments to other functions.
- Executed (called) at a later time when the operation completes.
- Example: Simulating a Slow Network Request

```
function fetchData(url, callback) {  
  setTimeout(() => {  
    // Simulate network delay  
    const data = { message: 'Data from the server' };  
    callback(data);  
  }, 2000); // 2 second delay  
}  
  
fetchData('https://api.example.com', (data) => {  
  console.log(data);  
});
```

PROMISES

- Objects representing the eventual outcome (success or failure) of an asynchronous operation.
- Provide .then and .catch for handling success and errors.
- **States:** Pending, Fulfilled, Rejected
- Example: Refactoring with Promises

```
function fetchData(url) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      const data = { message: 'Data from the server' };  
      resolve(data);  
    }, 2000);  
  });  
}  
  
fetchData('https://api.example.com')  
  .then(data => console.log(data))  
  .catch(error => console.error(error));
```

Async/Await {

```
JavaScript

// Old way (with Promises)
function getWeather() {
  fetch('https://api.openweathermap.org/
data/2.5/weather?q=London')
    .then(response => response.json())
    .then(data => console.log("Temperature:",
data.main.temp));
}

// New way (with async/await)
async function getWeather() {
  const response = await
fetch('https://api.openweathermap.org/data/2.5/weather
?q=London');
  const data = await response.json();
  console.log("Temperature:", data.main.temp);
}
```

- **'async'**: Labels a function as asynchronous, meaning it might have to wait for things to happen before finishing.
- **'await'**: Works inside an 'async' function. Pauses the function until a Promise is settled (has its final result or error).
- **Why use them?** Makes your code dealing with Promises cleaner and easier to follow, as it looks more like normal step-by-step code.

See the difference? With async/await, it almost reads like a recipe:

- `await fetch(...)`: "Get the weather data (and wait for it to arrive)."
- `await response.json()`: "Decode the weather data (and wait for that too)."
- `console.log(...)`: "Finally, show the temperature!"

APIs (Application Programming Interfaces){

- APIs provide a way for applications to communicate with each other
- Fetch API: Built-in, Promise-based tool for making network requests.
- Examples: Weather APIs, Map APIs, Translation APIs, etc.

```
JavaScript

async function getWeather(city) {
  const response = await fetch(`https://api.openweathermap.org/data/2.5/weather?
q=${city}`);
  const data = await response.json();
  return data.main.temp;
}

async function displayWeather() {
  const temp = await getWeather('London');
  console.log(`The temperature in London is ${temp}`);
}
```

Review 'Concepts' {

JavaScript:

Powers Interaction: JavaScript makes web pages dynamic, handling user input and changing what you see.

DOM: The Webpage Map: The DOM is the browser's way of representing the structure of a webpage. JavaScript uses it to manipulate elements.

Essentials:

Variables store data.
if/else and loops control how code runs.
Functions are reusable code blocks.
DOM Manipulation:

Async Code: Handling Delays

JavaScript can handle things that take time (fetching data).
async/await makes code that waits for these operations look cleaner.

}



```
1 Thanks For Tunning In; {
```

```
2  
3 'Do you have any questions?'💡
```

```
4  
5 < Feel free to ask >
```

```
6  
7  
8 <p> Speakers :
```

```
9  
10 Arya Basu : https://www.linkedin.com/in/aryabasu21/
```

```
11 Soham Das : https://www.linkedin.com/in/soham-das-15ab07174/
```

```
12 </p>
```

```
13  
14 }
```



[phoenixnsec.in]