Table of Contents

1. Introduction to JavaScript

JavaScript is a programming language that is widely used in web development. It is a client-side scripting language, which means that it is executed on the client side (in a web browser) rather than on the server side. JavaScript is used to create interactive web pages and is an essential component of modern web development. It is a high-level, dynamically typed language that is interpreted, which means that it is not compiled into machine code before it is run. JavaScript is used to add functionality to websites, such as handling user events (such as clicks and form submissions), animating page elements, and making asynchronous requests to servers. It is a versatile language that can be used for a wide range of applications, from simple scripts that add simple interactivity to websites to complex single-page applications.

Real-life Example:
When you click "Add to Cart" on Amazon and the cart updates without reloading — that's JavaScript in action.

On Page Script & External File

<script type="text/javascript"> .... </script>

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

Variables: var, let, const

- **var**: Function-scoped and can be redeclared.
- **let**: Block-scoped and cannot be redeclared within the same scope.
- **const**: Block-scoped and cannot be reassigned.

Example:

```
var name = "govind";

let age = 25;

const country = "India";
```

Data Types

- **Primitive**: Number, String, Boolean, Null, Undefined, Symbol, BigInt
- **Non-Primitive**: Object, Array, Function

Example:

```
let str = "Hello";

let arr = [1, 2, 3];

let obj = { name: "Alice" };
```

Operators

- **Arithmetic: +, -, *, /, %**
- **Comparison: ==, ===, !=, !==, <, >**
- **Logical: &&, ||, !**
- **Assignment: =, +=, -=**

Example:

```
let a = 5;

let b = 10;

console.log(a + b); // 15
```

Control Structures

- **if/else**
- **switch**
- **for, while, do-while loops**

Example:

```
if (age > 18) {
        console.log("Adult");
} else {
        console.log("Minor");
}
```

2. Intermediate JavaScript

Functions

Function Declaration

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

Function Expression

```
const greet = function() {
  console.log("Hello!");
}
```

Arrow Functions

```
const greet = () => console.log("Hello!");
```

Scope: Block, Function, Global

- **Block Scope** (created using {}): Variables declared using let or const live here.
- **Function Scope**: Variables declared with var inside a function.
- **Global Scope**: Accessible from anywhere in the program.

Real-life analogy: Think of scopes like rooms in a house. A variable (item) in one room isn't necessarily visible in another.

Example:

```
let a = "global";
function test() {
  let b = "function scope";
  if (true) {
    let c = "block scope";
  }
}
```

Execution Context

When a function is called, the JavaScript engine creates an execution context.

1. **Global Execution Context**: Created for the whole script.
2. **Function Execution Context**: Created every time a function is invoked.

Real-life analogy: Like a call center creating a new workspace every time a customer calls.

Creation Phase & Execution Phase

- **Creation Phase**: Variables and functions are stored in memory. Variables are set to undefined.
- **Execution Phase**: Code runs line by line and variables are assigned actual values.

Example:

```
console.log(x); // undefined
var x = 10;
```

Hoisting

Variables and functions are moved to the top of their scope before code execution.

Closures

A closure is formed when an inner function accesses a variable from its outer function even after the outer function has executed.

Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count);
  }
}
const counter = outer();
counter(); // 1
counter(); // 2
```

Real-life analogy: A child (inner function) remembers their parent's (outer function) phone number even after moving away.

Callback Functions

A function passed as an argument to another function and executed after the parent function completes.

Example:

```
function greet(name, callback) {
  console.log("Hi " + name);
  callback();
}
function callMe() {
  console.log("I am callback");
}
greet("Govind", callMe);
```

First-Class & Higher-Order Functions

- **First-Class**: Functions can be assigned to variables, passed as arguments, or returned from another function.
- **Higher-Order**: Functions that take other functions as arguments or return them.

3. Advanced JavaScript

Lexical Environment & Scope Chain

- Lexical environment is the context in which variables are defined.
- Scope chain enables access to variables from parent scopes.
- **Example:**

```
function outer() {
  let name = "Govind";
  function inner() {
    console.log(name); // Accesses parent scope
  }
  inner();
}
outer();
```

Shadowing

Occurs when a variable in a local scope has the same name as a variable in a parent scope.

Example:

```
let a = 10;
function test() {
  let a = 5;
  console.log(a); // 5 (local variable shadows the outer one)
}test();
```

setTimeout with Closures

Closures maintain access to outer variables even inside asynchronous functions like setTimeout.

Example:

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function() {
    console.log(i); // Prints 4 three times due to closure with var
  }, 1000);
}
```

To fix:

```
for (let i = 1; i <= 3; i++) {
  setTimeout(function() {
    console.log(i); // 1, 2, 3
  }, 1000);
}
```

Asynchronous JavaScript

Allows non-blocking execution using:

- **Callbacks**
- **Promises**
- **Async/Await**

Real-life analogy: Ordering food at a restaurant (asynchronous). You place your order and continue chatting while the chef cooks in the background.

Promises & Async/Await

```
function getData() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Data received"), 2000);
  });}


async function fetchData() {
  let result = await getData();
  console.log(result);
}
fetchData();
```

Array Methods: map, filter, reduce

map: Transforms each element in an array.

```
const nums = [1, 2, 3];
const squares = nums.map(x => x * x);
```

filter: Filters elements based on a condition.

```
const even = nums.filter(x => x % 2 === 0);
```

reduce: Reduces array to a single value.

```
const sum = nums.reduce((acc, curr) => acc + curr, 0);
```