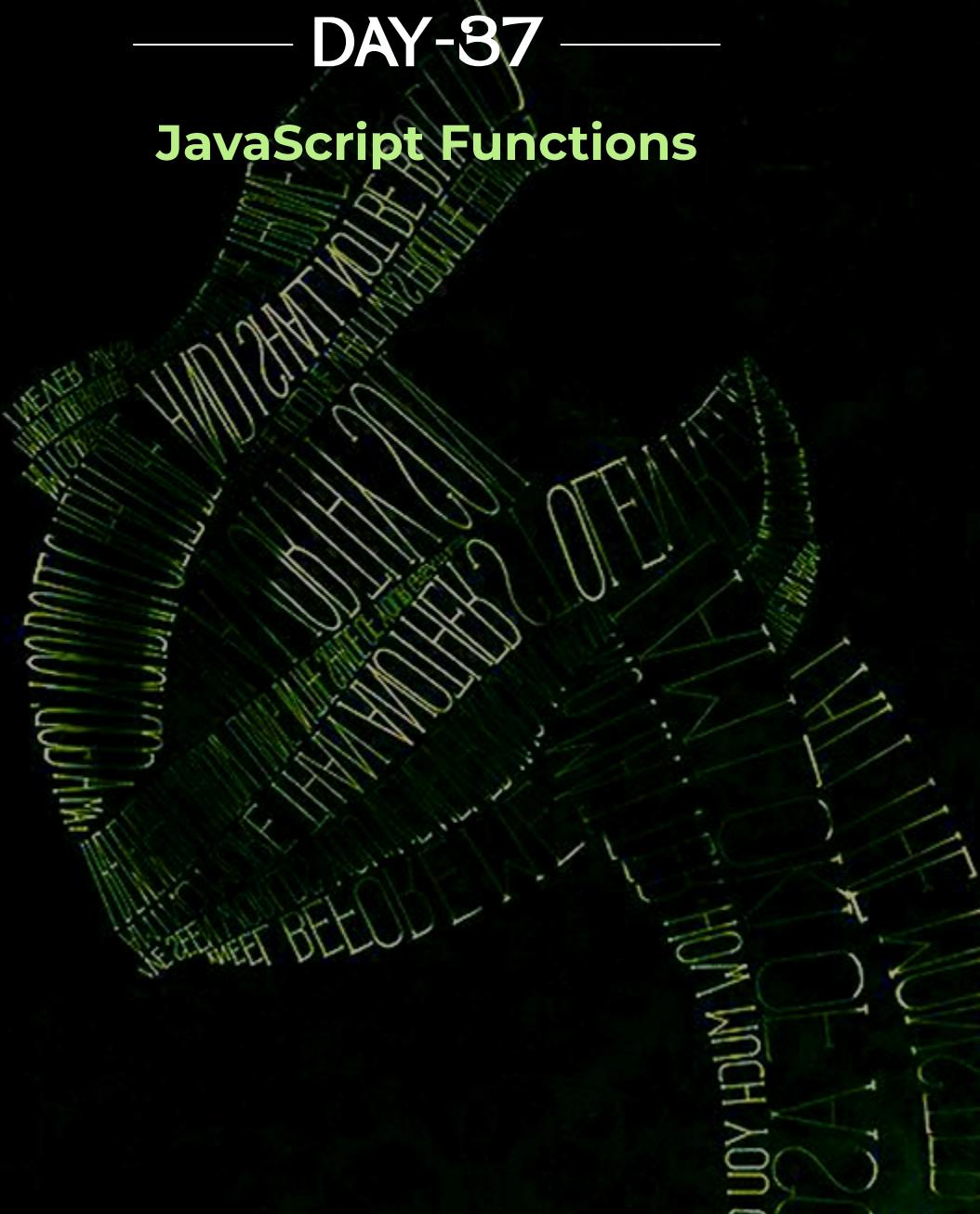


Live Cohort

DAY-37

JavaScript Functions



JavaScript Function

1. Parameters in JavaScript

Parameters are variables listed in a function's definition.
Arguments are the actual values passed to the function when it is called.

Required Parameters

These must be provided when calling the function.
If not passed, their value becomes `undefined`.

✿ Example:

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

Default Parameters

These parameters take a default value if no argument is provided.

✿ Example:

```
● ● ●  
function greet(name = "Student") {  
    console.log("Hello " + name);  
}  
  
greet(); // Hello Student
```

Default parameters help prevent errors when arguments are missing.

JavaScript Function

Rest Parameters

Sometimes we don't know how many arguments will be passed to a function.

Rest parameters collect all remaining arguments into an array.

✿ Example:

```
● ● ●  
function sum(...numbers) {  
    return numbers.reduce((acc, num) => acc + num, 0);  
}  
  
console.log(sum(1, 2, 3, 4)); // 10
```

Rest parameters are useful when building functions that accept a variable number of inputs (like `Math.max`).

Destructured Parameters

You can directly extract specific properties from objects or arrays passed into a function.

✿ Example:

```
● ● ●  
function display({ name, age }) {  
    console.log(` ${name} is ${age} years old`);  
}  
  
display({ name: "Shery", age: 25 });
```

Key Point:

Parameters define what kind of data a function will receive.

You can combine required, default, and rest parameters as needed.

JavaScript Function

2. Arguments in JavaScript

Arguments are the actual values passed to a function when it is called.

They replace the parameters inside the function during execution.

Positional Arguments

Matched to parameters based on their order.

❖ Example:

```
● ● ●  
function multiply(a, b) {  
    return a * b;  
}  
  
console.log(multiply(5, 2)); // 10
```

Default Arguments

When a parameter has a default value, it will be used if no argument is provided.

❖ Example:

```
● ● ●  
function greet(name = "Sheryian") {  
    console.log("Hello " + name);  
}  
  
greet(); // Hello Sheryian
```

JavaScript Function

Spread Arguments

The spread syntax (`...`) allows an array to be expanded into individual elements.

* Example:

```
● ● ●  
function add(a, b, c) {  
    return a + b + c;  
}  
  
const nums = [1, 2, 3];  
console.log(add(...nums)); // 6
```

Key Point:

The way arguments are passed (by order or using spread syntax) affects how the function behaves.

3. Variable Hoisting

Hoisting is the default behavior in JavaScript where variable declarations are moved to the top of their scope before execution.

However, only the declaration is hoisted — not the initialization.

* Example:

```
● ● ●  
  
console.log(x); // undefined  
var x = 5;
```

JavaScript Function

Key Points:

- `var` variables are hoisted and initialized as `undefined`.
- `let` and `const` are also hoisted but remain in the **Temporal Dead Zone (TDZ)** until their declaration line executes.
- Accessing them before declaration causes a **ReferenceError**.

4. Function Hoisting

Function declarations are hoisted completely both name and body.

You can call them before they are defined.

✿ Example:

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

Key Points:

- Function declarations are hoisted.
- Function expressions and arrow functions are **not hoisted** since they're treated as variable assignments.

JavaScript Function

5. Classic Functions and Nested Functions

Classic functions use the `function` keyword.

Nested functions are functions inside other functions.

✿ Example:

```
● ● ●  
function outer() {  
  let outerVar = "Outer";  
  
  function inner() {  
    console.log(outerVar); // Accessible due to scope chain  
  }  
  
  inner();  
}  
  
outer();
```

Explanation: Inner functions can access variables of outer functions — this is **lexical scoping**.

6. Scope Chain in JavaScript

The scope chain determines how JavaScript finds variables.
If not found locally, JS looks in parent and global scopes.

✿ Example:

```
● ● ●  
let a = "Global";  
  
function outer() {  
  let b = "Outer";  
  
  function inner() {  
    let c = "Inner";  
    console.log(a, b, c);  
  }  
  
  inner();  
}  
  
outer();
```

● ● ●
Output:
Global Outer Inner

Key Point:

- JS always searches variables from local → outer → global scope.

JavaScript Function

7. IIFE (Immediately Invoked Function Expression)

An IIFE runs immediately after it's defined — used to create a private scope.

✿ Example:

```
● ● ●  
(function() {  
  console.log("This runs immediately!");  
})();
```

Explanation: Parentheses make it an expression, and the second pair executes it instantly.

Commonly used before ES6 modules to isolate scope.

8. More Types of Functions in JavaScript

◆ Arrow Functions

Cleaner syntax introduced in ES6.

✿ Example:

```
● ● ●  
const add = (a, b) => a + b;  
console.log(add(2, 3)); // 5
```

◆ Fat Arrow Functions

Refers to the `=>` symbol.

✿ Example:

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

JavaScript Function

◆ Anonymous Functions

Functions without a name, often used temporarily.

✿ Example:

```
● ● ●  
setTimeout(function() {  
    console.log("Anonymous function executed");  
, 1000);
```

◆ Higher-Order Functions

Functions that take or return other functions.

✿ Example:

```
● ● ●  
function higherOrder(fn) {  
    fn();  
}  
  
higherOrder(() => console.log("This is a callback!"));
```

◆ Callback Functions

Passed as arguments and executed later.

✿ Example:

```
● ● ●  
function greet(name, callback) {  
    console.log("Hello " + name);  
    callback();  
}  
  
greet("Shery", () => console.log("Welcome to JavaScript!"));
```

JavaScript Function

◆ First-Class Functions

Functions are treated as values.

✿ Example:



```
const sayHi = () => console.log("Hi!");
const run = sayHi;
run(); // Hi!
```

Key Point:

- JavaScript functions are **first-class citizens**, meaning they can be stored, passed, or returned — enabling modular and dynamic code.