Introduction to JavaScript

a high-level, interpreted programming language

R.A: Assad Ullah Khan

in assadullahkhan

AssadUllahKhan

What is a Function?

Concept:

- A function in JavaScript is a **reusable block** of code designed to perform a specific task.
- Instead of writing the same **code multiple times**, we define it **once** and call it whenever needed.
- Functions make code **modular, readable,** and **easy to maintain.**

Functions make your code:

1. **Reusable:** write once, use many times
2. **Organized:** divide big programs into smaller parts
3. **Maintainable:** easy to debug and update

```
// Defining a function
function greet() {
  console.log("Hello, welcome to JavaScript!");
}

// Calling the function
greet();
```

Think of a function as a machine:You give it some input, it processes it, and returns an output — just like a coffee machine ☕

Why Use Function

- Code Reusability
- Modularity
- Readability
- Maintainability

What is a Function?

Concept:

- A function declaration is the most **common way** to define a function in JavaScript.
- It tells the JavaScript engine what the function does and allows it to be used anywhere in your code (because of hoisting).

A function declaration always starts with the **function** keyword, followed by:
Function Name: a meaningful name **describing the action**.
Parentheses ( ): used to **hold parameters**.
Curly Braces { }: where you write the code that **executes** when the function is called.

```
// Defining a function
function functionName(parameter1, parameter2) {
//code to execute
Return result;
}

// Calling the function
functionName(5, 10);
```

The return statement ends the function and sends back a value.
If you don't use return, the function returns undefined by default.
Function declarations are hoisted, meaning you can call them before they are defined.

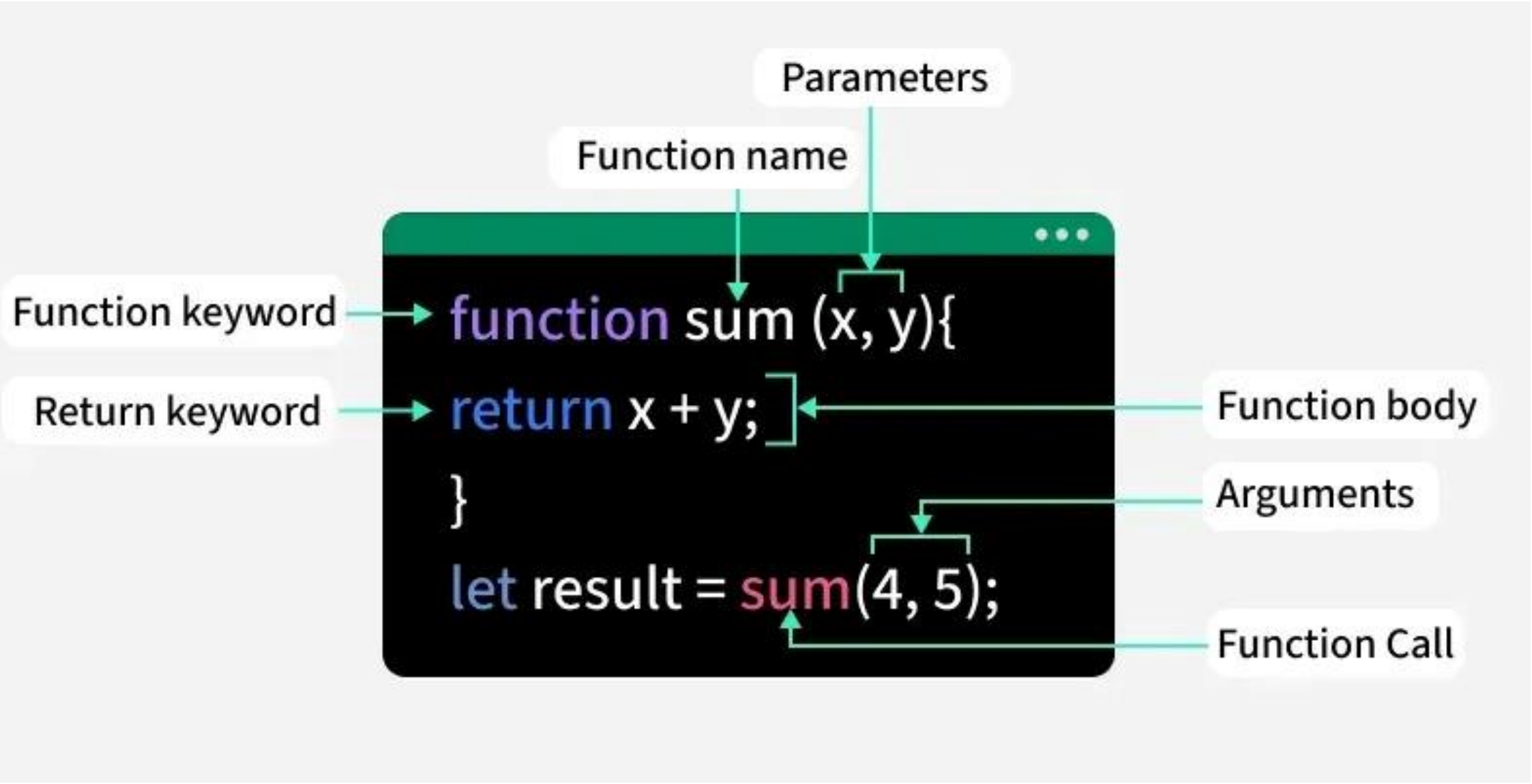
Function Parameters and Return Values

Concept:

- Functions often need **input values** to work with, these are called **parameters.**
- When you call the function and give actual values, those are called **arguments.**

A function can also send data back using the **return statement**.
This helps make the function reusable and dynamic. It can work with different inputs and produce different outputs.

```
// Defining a function
function functionName(parameter1, parameter2) {
//code to execute
Return result;        //sends value back
}
```


Parameters
Function name
Function keyword
function sum (x, y){
Return keyword
return x + y;
}
Function body
Arguments
let result = sum(4, 5);
Function Call

Function Expressions

Concept:

- A function expression is when a function is **stored inside a variable** instead of being declared normally.
- This allows you to treat functions like any other value, passing them around or assigning them **dynamically.**

Key Points:

- Can be **anonymous** (no name)
- **Cannot be hoisted** like function declarations
- Useful for **callbacks** and **modular code**

```
const variableName = function(parameters) {
  // code
  return result;
};
```

```
const sayHello = function(name) {
  return "Hello " + name;
};
console.log(sayHello("Asad"));  // Output: Hello Asad
```

Explanation:

- Function is assigned to the variable sayHello.
- We call the function using the variable name: sayHello("Asad").
- Since it's a function expression, it cannot be called before the line it is defined.

Arrow Functions (ES6)

Concept:

- Arrow functions are a shorter and cleaner way to write functions introduced in ES6 (ECMAScript 2015).
- They are especially useful for simple and one-line functions.
- Arrow functions use the => (fat arrow) syntax.They are also anonymous functions by nature (no name by default).

```
const greet = (name) => {
  return "Hello " + name;
};

console.log(greet("Asad")); // Output: Hello Asad
```

```
const functionName = (parameters) => {
  // code
  return result;
};
```

If your function has only one line of code, you can omit return and {}:

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

Simplified Version:

```
const greet = name => "Hello " + name;
console.log(greet("Khan")); // Output: Hello Khan
```

Scope (Global vs Local)

Concept:

- Scope determines **where a variable is accessible** in your code.

  There are **two main types of scope** in JavaScript:

1. **Global Scope:** Variables declared outside any function.

   Accessible anywhere in the code.

2. **Local Scope:** Variables declared inside a function.

   Accessible only inside that function.

Understanding scope is important to **avoid conflicts** and **unexpected behavior**.

Global Scope:

```
let globalVar = "I am global";

function showGlobal() {
  console.log(globalVar);        // Accessible here
}

showGlobal();
console.log(globalVar);          // Accessible here too
```

```
I am global
I am global
```

Local Scope:

```
function showLocal() {
  let localVar = "I am local";
  console.log(localVar); // Accessible here
}

showLocal();
console.log(localVar); // ❌ Error: localVar is not defined
```

Key Points

- Variables defined with **let** or **const** inside a block **{}** are block scoped.
- Variables defined without **let/const/var** inside a function become global (avoid this!).
- Scope helps protect variables from being accessed or modified accidentally.

Hoisting in Functions

Concept:

- **Hoisting** in JavaScript means that **function declarations** and **variable declarations** are moved ("hoisted") to the **top of their scope** before the code executes.
- This allows you to **call a function before it is defined** in the code.

Function Declaration Hoisting:

```
greet();    // Works due to hoisting

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

Explanation:

- Even though the greet() function is called before its definition, JavaScript moves (hoists) the entire function declaration to the top during execution.

Hoisting in Functions

Important Note:

- **Function declarations** are **hoisted completely**.
- **Function expressions** and **arrow functions** are not hoisted — you must define them before calling.

Example (Function Expression Not Hoisted)

```
sayHello();            // ❌ Error

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

Key Takeaway:

- ✅ Function Declarations → Hoisted
- ❌ Function Expressions / Arrow Functions → Not Hoisted
- Helps understand execution order in JavaScript.

Default Parameters

Concept:

- Default parameters allow you to assign default values to function parameters.If a value is not provided when the function is called, the default value is used automatically.

Syntax:

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

Why Use Default Parameters?:

- Prevents **undefined values**.
- Makes functions **more flexible** and **easier to call**.
- Improves **code readability** and **avoids errors**.

Anonymous Functions

Concept:

- An anonymous function is a function without a name.
- It's often used when you don't need to reuse the function — for example, when passing it as a callback or defining it inline.

Syntax:

```
function() {
  console.log("This is an anonymous function");
}
```

- ⚠ However, this by itself will not run unless it's assigned to a variable or used immediately.

```
let greet = function() {
  console.log("Hello from Anonymous Function!");
};

greet();
```

Anonymous Functions

Why Use Anonymous Functions?

- Useful for short, one-time tasks.
- Keeps code concise and readable.
- Common in callbacks, event handlers, and functional programming

💡 *Key Takeaways*

- Anonymous functions don't have names.
- They are commonly used inside other functions.
- Can be stored in variables or passed as arguments.
- Cannot be hoisted like normal functions.

Immediately Invoked Function Expressions (IIFE)

Concept:

- An IIFE (Immediately Invoked Function Expression) is a function that runs as soon as it's defined — no need to call it later.
- It's often used to avoid variable conflicts and create a private scope.

Syntax:

```
(function() {
  let message = "Hello from IIFE!";
  console.log(message);
})();
```

Example 2:

```
(function(name) {
  console.log("Welcome, " + name + "!");
})("Asad");
```

Immediately Invoked Function Expressions (IIFE)

Why Use IIFE?

- Avoids polluting the global scope.
- Keeps variables private.
- Useful for initializing scripts or running setup code instantly.

💡*Key Takeaways*

- IIFE = Function defined + invoked immediately.
- Common in module patterns and encapsulation.
- Helps maintain cleaner, safer code.

Callback Functions

Concept:

- A callback function is a function passed as an argument to another function and executed after the completion of that function.
- It allows asynchronous or step-by-step control over execution.

Syntax:

```
function greetUser(callback) {
  console.log("Hello!");
  callback();          // executing the passed function
}
```

Example:

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

function processUser(callback) {
  console.log("Processing user...");
  callback();
}

processUser(greet);
```

Callback Functions

Why Use Callbacks?

- To control execution order.
- To handle asynchronous tasks like:
  - Reading files
  - API requests
  - Timers or user actions

💡 *Key Takeaways*

- Callbacks are functions passed as parameters.
- They run after the main function finishes.
- Common in asynchronous operations (like setTimeout, fetch, etc.).

Nested Functions

Concept:

- A nested function is a function defined inside another function.
  - The inner function is only accessible within the outer function.
  - Useful for encapsulation and organizing code.

Syntax:

```
function greetUser(name) {
  function getMessage() {
    return "Hello, " + name + "!";
  }
  console.log(getMessage());
}

greetUser("Asad");
```

Why Use Nested Functions?

- Encapsulation: keeps helper functions private.
- Cleaner, modular code.
- Can access outer function variables (closure behavior).

Practice Challenge

Goal:

- Apply everything you learned about functions — including parameters, return values, nested functions, callbacks, and arrow functions — in one short coding exercise.

Challenge Task:

- Create a function-based mini program that simulates a shopping cart.

Requirements

1. Create a function named addToCart that takes two parameters:
   - itemName (string)
   - price (number)
2. Inside the function:
   - Keep track of total price using a variable outside the function.
   - Print a message showing which item was added and its price.
3. Create another function showTotal() that prints the final total.
4. Use arrow functions or nested functions where appropriate.

Solution

```
let total = 0;

function addToCart(itemName, price) {
  total += price;
  console.log(itemName + " added to cart: $" + price);
}

const showTotal = () => {
  console.log("Total price: $" + total);
};

// Adding items
addToCart("Shirt", 25);
addToCart("Jeans", 40);
addToCart("Shoes", 35);

// Display total
showTotal();
```

Any Question

Thank You