1 Basics of Functions

A function is like a **machine** that takes input (optional), processes it, and gives an output.

📌 Example:

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

greet(); // Output: Hello, welcome!
```

✅ Why?

- We **defined** a function `greet()`
- We **called** the function using `greet();`
- The function executed and printed the message.

🎯 Task 1:

Create a function `sayGoodbye()` that prints `"Goodbye, see you soon!"` and call it.

Basic function

2 Function with Parameters 🎯

Parameters are like **inputs** for the function.

📌 Example:

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

greetUser("John"); // Output: Hello, John!
```

✅ Why?

- The function **accepts a parameter** (`name`)
- We pass `"John"` when calling, so `"Hello, John!"` is printed.

🎯 Task 2:

Create a function `add(a, b)` that **takes two numbers** and prints their **sum**.

3 Return Statement 🎯

Sometimes, we need a function to **give back a value** instead of just printing it.

📌 Example:

```js
function add(a, b) {
    return a + b;
}

let result = add(5, 10);
console.log(result); // Output: 15
```

✅ Why?

- `return` **sends the result** back to where the function was called.
- We store it in `result` and print it.

🎯 Task 3:

Create a function `multiply(x, y)` that returns the **product** of two numbers.

We don’t need to call the function if we use return

4 Arrow Functions (Shorter Syntax) 🎯

Arrow functions are a **modern way** to write functions.

📌 Example:

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

console.log(add(3, 7)); // Output: 10
```

✅ Why?

- We **removed** `function` and replaced it with `=>`
- When the function has **only one return statement**, we can **skip** `{}`.

🎯 Task 4:

Convert your `multiply(x, y)` function into an **arrow function**.

A **callback function** is a function **passed inside another function**.

5 Callback Functions 🎯

A **callback function** is a function **passed inside another function**.

📌 Example:

js Copy Edit

```
function process(callback) {
    console.log("Processing...");
    callback();
}

process(() => console.log("Done!"));
// Output: Processing...
//         Done!
```

✅ Why?

- `callback` is a function we pass.
- The `process()` function runs it when needed.

6 Higher-Order Functions 🎯

A **higher-order function** is a function that **takes another function** as an argument **or returns a function**.

📌 Example:

js Copy Edit

```
function operation(action, a, b) {
    return action(a, b);
}

console.log(operation((x, y) => x + y, 4, 5)); // Output: 9
console.log(operation((x, y) => x * y, 4, 5)); // Output: 20
```

✅ Why?

- `operation()` takes a **function as an argument** (`action`).
- We pass `x + y` or `x * y` to decide the operation.

🎯 Task 6:

Create a higher-order function `applyFunction(func, a, b)` that **applies a given function** to `a` and `b`.

5️⃣ IIFE (Immediately Invoked Function Expressions)

💡 **A function that runs immediately after being defined.**

Why use IIFE?

- Avoids **polluting global scope**.
- Useful for **one-time execution** of code.

Example

js Copy Edit

```
(function() {
    console.log("This function runs immediately!");
})();
// Output: This function runs immediately!
```

Since the function **executes instantly**, it's useful for things like **module patterns** and **initialization scripts**.

4 Closures

💡 **A function that "remembers" variables from its outer scope, even after the outer function has executed.**

Why use closures?

- To **preserve data** across function calls.
- Used in **private variables, counters, and module patterns.**

Example: Counter

js Copy Edit

```js
function createCounter() {
    let count = 0; // Private variable

    return function() {
        count++;
        console.log(count);
    };
}

const counter = createCounter();
counter(); // Output: 1
counter(); // Output: 2
counter(); // Output: 3
```

Here, the `count` variable is **"remembered"** by the returned function **even after** `createCounter()` **is**