Functions in JavaScript

A **function** in JavaScript is a block of reusable code that performs a specific task. Functions make your code **modular**, **readable**, and **efficient**.

Types of Functions in JavaScript

1. Function Declaration (Named Function)

A function is defined using the function keyword and has a name.

```
function greet() {
  console.log("Hello, Suraj!");
}
greet(); // Output: Hello, Suraj!
```

2. Function Expression (Anonymous Function)

A function is assigned to a variable.

```
let add = function(a, b) {
  return a + b;
};
console.log(add(5, 3)); // Output: 8
```

3. Arrow Function (ES6)

A shorter way to write functions using =>.

```
const multiply = (a, b) => a * b;
console.log(multiply(4, 5)); // Output: 20
```

If there is only **one parameter**, parentheses () are optional

```
const square = x => x * x;
console.log(square(6)); // Output: 36
```

For multiple lines, use {}:

```
const sayHello = () => {
  console.log("Hello, JavaScript!");
};
sayHello(); // Output: Hello, JavaScript!
```

4. Immediately Invoked Function Expression (IIFE)

A function that runs immediately after being defined.

```
(function() {
  console.log("IIFE executed!");
})();
// Output: IIFE executed!
```

With parameters:

```
(function(name) {
  console.log("Hello, " + name);
})("Suraj");
// Output: Hello, Suraj
```

5. Function with Parameters and Return Value

A function can take **parameters** and **return** a value.

```
function subtract(a, b) {
  return a - b;
}
console.log(subtract(10, 3)); // Output: 7
```

```
. const example = (a, b) => {
  console.log("The value of a and b is:", a, b); // Logging the values of a and b
  return a * b; // Returning the product of a and b
};
```

```
console.log(example(5, 5)); // Calling the function with numeric values
```

6. Default Parameters (ES6)

If no value is passed, the default value is used.

```
function greet(name = "Guest") {
  console.log("Hello, " + name);
}
greet();         // Output: Hello, Guest
greet("Suraj");  // Output: Hello, Suraj
```

7. Rest Parameters (...)

Allows passing **multiple arguments** as an array.

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

8. Callback Functions

A function passed as an **argument** to another function.

```
function processData(data, callback) {
  console.log("Processing:", data);
  callback();
}


function finished() {
  console.log("Task Completed!");
}


processData("Data Loaded", finished);
// Output:
// Processing: Data Loaded
// Task Completed!
```

Function Hoisting in JavaScript

JavaScript has a concept called **hoisting**, which affects function behavior.

□ **Function declarations** are hoisted, meaning JavaScript moves them to the top before execution.
□ **Function expressions & arrow functions** are **not hoisted**, so calling them before defining will cause an error.

1. Calling a Function Before Declaration (Works ✅)

If you use a **function declaration**, you can call it before defining it.

```
main(); // ✅ Works because greet() and add() are function declarations


function main() {
  greet();
  console.log("Sum:", add(5, 3));
}


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


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

Why Does This Work?

Because **function declarations are hoisted**, JavaScript moves them to the top internally.

2. Calling a Function Expression Before Declaration (Error ❌)

If you use a **function expression** (assigned to a variable), calling it before definition causes an error.

```
main(); // ❌ ERROR: Cannot access 'main' before initialization


const main = function() {
  greet();
  console.log("Sum:", add(5, 3));
};


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


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

Why Does This Fail?

- Function expressions **are not hoisted**.
- const main = function() {} is like a variable, so JavaScript does not move it to the top.

3. Calling an Arrow Function Before Declaration (Error □)

Arrow functions also **behave like function expressions**, so calling them before definition causes an error.

```
main(); // □ ERROR: main is not defined


const main = () => {
  greet();
  console.log("Sum:", add(5, 3));
};


const greet = () => {
  console.log("Hello, Suraj!");
};


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

□ Best Practice: Define Functions Before Calling

Even though function declarations are hoisted, **it's a best practice to define functions first** before calling them.

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


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


function main() {
  greet();
  console.log("Sum:", add(5, 3));
}


main(); // □ Works safely!
```

□ Summary

Function Type	Hoisted?	Can Call Before Definition?
Function Declaration	□ Yes	□ Yes
Function Expression	□ No	□ No (Error)
Arrow Function	□ No	□ No (Error)

1. Function Declaration (Hoisted □)

□ If a function is defined using the function keyword **without assigning it to a variable**, it's a **function declaration**.

□ **It is hoisted**, so you **can call it before it's defined**.

□ **You can call it before the function is written.**
□ JavaScript **automatically moves function declarations to the top**.

---

2. Function Expression (Not Hoisted □)

□ If a function is assigned to a variable using const or let, it is a **function expression**.

□ **It is NOT hoisted**, so calling it before definition **causes an error**.

□ **How to Identify?** □ A function **assigned to a variable** is a **function expression**.
□ **You cannot call it before defining it** because it behaves like a variable.

---

3. Arrow Function (Not Hoisted □)

□ If a function is defined using the **arrow syntax (=>)**, it's an **arrow function**.

□ **It is NOT hoisted**, so calling it before definition **causes an error**.

□ **How to Identify? □ Uses the => (arrow function syntax)**.
□ Like function expressions, **it is not hoisted**.