Functions

- Functions in JavaScript are like the building blocks of your code, allowing you to break down complex tasks into smaller, manageable steps.
- A **function** is a self-contained, reusable block of code that performs a specific task when it's called or invoked.
- Functions are essential for maintaining clean, organized, and efficient code in JavaScript.
- They encapsulate logic, promoting code reusability and making it easier to understand and maintain.
- Functions can be compared to real-life actions or operations. For example, you can think of a function that calculates the average of a list of numbers as a mathematical operation.

Anatomy of a Function

- Every JavaScript function has a basic structure:

```
function functionName(parameters) {
  // Code to be executed
return result; // optional
}
```

- **functionName**: This is the name of your function, which you choose. It should be descriptive of what the function does.
- **parameters**: These are optional inputs that you can pass to the function. Parameters are like variables that store values passed to the function.
- **Code to be executed**: This is the block of JavaScript code enclosed in curly braces `{}`. It contains the instructions and logic that define what the function does.
- **return**: The `return` statement is optional but important. It specifies the value that the function will output. If omitted, the function returns `undefined`.

Function Invocation

- To use a function, you must invoke or call it:

```
functionName(arguments); // Arguments are the actual values you pass to the function
```

When a function is invoked, the code inside the function block is executed.

Example Function

```
// A simple function that adds two numbers
function addNumbers(a, b) {
  const result = a + b;
  return result;
}
// Calling the function and storing the result
const sum = addNumbers(5, 3); // Output: sum is now 8
```

Benefits of Functions

1. **Modularity**: Functions break down code into smaller, modular parts, making it easier to manage and test.
2. **Code Reusability**: You can use the same function multiple times throughout your program.
3. **Readability**: Functions make your code more readable and self-explanatory by encapsulating logic in named blocks.
4. **Maintenance**: Changes or updates can be made in a single function, reducing the risk of introducing bugs in other parts of your code.
5. **Collaboration**: Functions facilitate collaboration among developers, as they provide clear interfaces for working with different parts of a program.

Function Declaration

- In JavaScript, there are different ways to declare functions. We'll start by discussing the most common method: **Function Declaration**.
- Function Declarations are hoisted, which means they can be called before they are defined in the code.
- Here's the basic syntax of a Function Declaration:

```
function functionName(parameters) {
  // Code to be executed
  return result; // optional
}
```

- **functionName:** This is the name of your function, which you choose. It should be descriptive of what the function does.
- **parameters:** These are optional inputs that you can pass to the function. Parameters are like variables that store values passed to the function.
- The code inside the function block defines what the function does.
- The `return` statement is optional but specifies the value that the function will output.

Example of Function Declaration

```
function sayHello(name) {
  return `Hello, ${name}!`;
}
// Calling the function
const greeting = sayHello("Akbar");
// Output: greeting is now "Hello, Akbar!"
```

- In this example, we declared a function called `sayHello` that takes a `name` parameter and returns a greeting message.
- We then called the function with the argument `"Alice"` and stored the result in the `greeting` variable.

Hoisting

- Function Declarations are hoisted, which means they are moved to the top of their containing scope during compilation.
- This allows you to call a function before it's defined in the code.

```
sayHello(); // This works!
function sayHello() {
  console.log("Hello, World!");
}
```

Function Expressions

- In JavaScript, **Function Expressions** provide another way to define functions.
- Unlike Function Declarations, Function Expressions are not hoisted. They are defined as variables.
- Here's the basic syntax of a Function Expression

```
const functionName = function(parameters) {
  // Code to be executed
  return result; // optional
};
```

- **parameters:** These are optional inputs that you can pass to the function, just like in Function Declarations.
- The code inside the function block defines what the function does.
- The `return` statement is optional but specifies the value that the function will output.

Returning Values

You can return various types of values from a function, such as:

- Numbers
- Strings
- Booleans
- Objects
- Arrays
- Other functions

Example of Function Expression

```
// Function Expression
const sayHello = function(name) {
  return `Hello, ${name}!`;
};
// Calling the function
const greeting = sayHello("Rufat");  // Output: greeting is now "Hello, Rufat!"
```

- In this example, we defined a Function Expression named `sayHello` using a variable.
- This function takes a `name` parameter and returns a greeting message.
- We then called the function with the argument `"Rufat"` and stored the result in the `greeting` variable.

Function Expressions can be anonymous (without a name) or named.

```
// Anonymous Function Expression
const anonymousGreet = function(name) {
  return `Hello, ${name}!`;
};

// Named Function Expression
const namedGreet = function sayHello(name) {
  return `Hello, ${name}!`;
};
```

Function Declaration vs. Function Expression

- It's important to note that there is a difference between Function Declarations and Function Expressions.
- Function Declarations are defined with the `function` keyword and are hoisted.
- Function Expressions, on the other hand, are not hoisted and are typically defined using variables.

```
// Function Declaration
function sayHello() {
  console.log("Hello, World!");
}

// Function Expression
const sayHello = function(name) {
  console.log(`Hello, ${name}!`);
};
```

Arrow Functions

- Arrow functions are a concise way to write functions in JavaScript, introduced in ES6 (ECMAScript 2015).
- They provide a shorter syntax for defining functions, especially useful for small, single-expression functions.
- Here's the basic syntax of an Arrow Function:

```
const functionName = (parameters) => expression;
```

- **functionName**: This is the variable name that stores the function. It can be a descriptive name.
- **parameters**: These are optional inputs, similar to Function Expressions.
- The `=>` arrow signifies the function expression.
- **expression**: This is a single JavaScript expression that gets evaluated and returned.

Example of Arrow Function

```
// Arrow Function
const sayHello = (name) => `Hello, ${name}!`;
// Calling the function
const greeting = sayHello("Gullu");  // Output: greeting is now "Hello, Gullu!"
```

- In this example, we defined an Arrow Function named `sayHello`.
- It takes a `name` parameter and directly returns a greeting message using a template literal.

Differences from Function Expressions

- Arrow functions have a few key differences from traditional Function Expressions:
  - They have a more concise syntax.
  - Arrow functions do not have their own `this` context; they inherit the `this` value from their containing lexical scope.

Use Cases for Arrow Functions

- Arrow functions are well-suited for short, simple functions.
- They are often used as callback functions or when defining functions in functional programming-style code.
- Arrow functions are especially handy when working with higher-order functions like `map`, `filter`, and `reduce`.

Function Scope

- Function Scope refers to the concept that variables declared inside a function are only accessible within that function.
- Variables declared outside of any function have global scope and can be accessed from anywhere in your code.

Function Scope: Variables declared inside a function are only accessible within that function.

```
function myFunction() {
  const localVar = "I'm a local variable";
  console.log(localVar);
}
myFunction(); // Output: "I'm a local variable"
console.log(localVar); // ReferenceError: localVar is not defined
```

In this example, `localVar` is a local variable within the `myFunction` function. It cannot be accessed outside of it.

Global Scope

- Variables declared outside of any function have global scope and can be accessed from anywhere in your code.

```
const globalVar = "I'm a global variable";
function myFunction() {
  console.log(globalVar);
}
myFunction(); // Output: "I'm a global variable"
console.log(globalVar); // Output: "I'm a global variable"
```

`globalVar` is declared outside of any function and can be accessed both inside and outside functions.

Shadowing

- When a local variable in a function has the same name as a global variable, it **shadows** the global variable within that function.

```
const x = 10;
function myFunction() {
  const x = 5; // This local variable shadows the global x
  console.log(x); // Output: 5
}
myFunction();
console.log(x); // Output: 10
```

In this example, the local variable `x` inside `myFunction` shadows the global variable `x` when accessed within the function.

Block Scope (ES6)

- In ES6, the `let` and `const` keywords introduce **block scope**, allowing variables to be scoped to the nearest enclosing block, like an `if` statement or a loop.

```
if (true) {
  let blockScopedVar = 10;
}

console.log(blockScopedVar); // ReferenceError: blockScopedVar is not defined
```

`blockScopedVar` is scoped to the `if` block and cannot be accessed outside of it.

Module scope

Before modules, a variable declared outside any function was a global variable. In modules, a variable declared outside any function is hidden and not available to other modules unless it is explicitly exported.

Modules also rely on `type="module"` in the <script> tag.

```
<script type="module">
import message from "./message.js";
</script>
```

Exporting makes a function or object available to other modules. In the next example, I export a function from the `sequence.js` module file:

```
// in sequence.js
export { sequence, toList, take };
```

Importing makes a function or object, from other modules, available to the current module.

```
import { sequence, toList, toList } from "./sequence";
```

In a way, we can imagine a module as a self-executing function that takes the import data as inputs and returns the export data.

Closures

- Closures are a powerful and advanced concept in JavaScript that builds upon Lexical Scope.
- A **closure** is a function that "remembers" its lexical scope, even when it's executed outside that scope.
- Closures allow you to create and access variables from an outer function's scope within an inner function.
- Closures allow you to create private variables by encapsulating data within a function's scope.

```
function outer() {
  const outerVar = "I'm from outer";
  function inner() {
    console.log(outerVar); // Accesses outerVar from the outer function
  }
  return inner; // Return the inner function
}
const closureFunc = outer(); // Assign the inner function to closureFunc
closureFunc(); // Output: "I'm from outer"
```