

JavaScript

Functions

() {}



What are Functions?

A function is a reusable block of code designed to perform a specific task.

It can take inputs, process those inputs, and return a result.

Functions are essential for reducing redundancy in your code, allowing you to reuse code blocks instead of repeating them.

For example:

```
1 function greet(name) {  
2   return `Hello, ${name}!`;  
3 }  
4  
5 console.log(greet('Oluwakemi'));
```

Function Declarations vs. Function Expressions

Function Declaration: A function that is defined with the **function** keyword and can be called before it's defined due to hoisting.

Like this 📱

```
1 function numAddition(num1, num2) {  
2   return num1 + num2;  
3 }  
4  
5 numAddition(4, 10)
```

The **function keyword** is very important when working with Function Declaration.



Function Expression: A function assigned to a variable.

It is not hoisted, so it can only be called after it's defined.

Like this 📱

```
1 const addingNumbers = function (num1 + num2) {  
2   return num1 + num2  
3 }  
4  
5 addingNumbers();
```



Calling Functions

After declaring a function, you must call it for execution. The executing function should be in scope(scope determines the accessibility)when it is called.

It is not necessary to call a function after function declaration. you can call it before the function declaration. This is called hoisting in Javascript.

Functions declarations can be hoisted i.e they can be called before the function is being defined.

Functions expressions cannot be hoisted i.e they can only be called after they have been defined.



Examples of hoisted and non-hoisted functions



```
1 // Hoisted function
2
3 multiplyName() // function was called before definition
4
5 function multiplyNum (firstNumber, secondNumber) {
6     return firstNumber * secondNumber
7 }
```



```
1 // A function expression cannot be hoisted
2 const addingNumbers = function (num1 + num2) {
3     return num1 + num2
4 }
5
6 addingNumbers(); // function must be called after definition
```

Arrow Functions

Arrow functions are a concise way to write functions in JavaScript.

These functions are particularly useful when working with functions that are used as callbacks or when you want to simplify your code.

An arrow function uses the `=>` syntax, which is why it's called an "arrow" function. Here's the basic syntax:

```
1 // Traditional Function
2 function add(a, b) {
3     return a + b;
4 }
5
6 // Arrow Function
7 const addArrow = (a, b) => a + b;
```




Understanding Parameters and Arguments

Parameters are the variables listed as part of a function's definition. They act as placeholders for the values that will be passed into the function when it is invoked.


Parameters allow functions to be dynamic and reusable, as they enable the same function to operate on different values.

Parameter



```
1 function greet(name) {  
2     console.log("Hello, " + name + "!");  
3 }
```

In this example, `name` is a parameter. When the `greet` function is defined, it expects one parameter, `name`. However, at this point, `name` doesn't have a value yet; it will receive a value when the function is called.



Arguments are the actual values you pass to the function when you call it. These values are assigned to the function's parameters.

In other words, when you invoke a function, you provide arguments, and those arguments are mapped to the parameters defined in the function.

Argument



```
1 greet("Alice");
```

In this call to the **greet** function, "**Alice**" is the argument. This value is passed to the greet function, and within the function, the name parameter is assigned the value "Alice". The function then executes using this value.

The whole function:

Parameter



```
1 function greet(name) {  
2     console.log("Hello, " + name + "!");  
3 }  
4 greet("Alice") // Outputs "Hello Alice!"
```

Argument

A function can have multiple parameters, and you can pass multiple arguments to it:

Parameter 1

Parameter 2

```
1 function greet(name, message) {  
2     console.log("Hello, " + name + "! " + message );  
3 }  
4 greet("Alice", "Welcome home") // Outputs "Hello Alice! Welcome home"
```

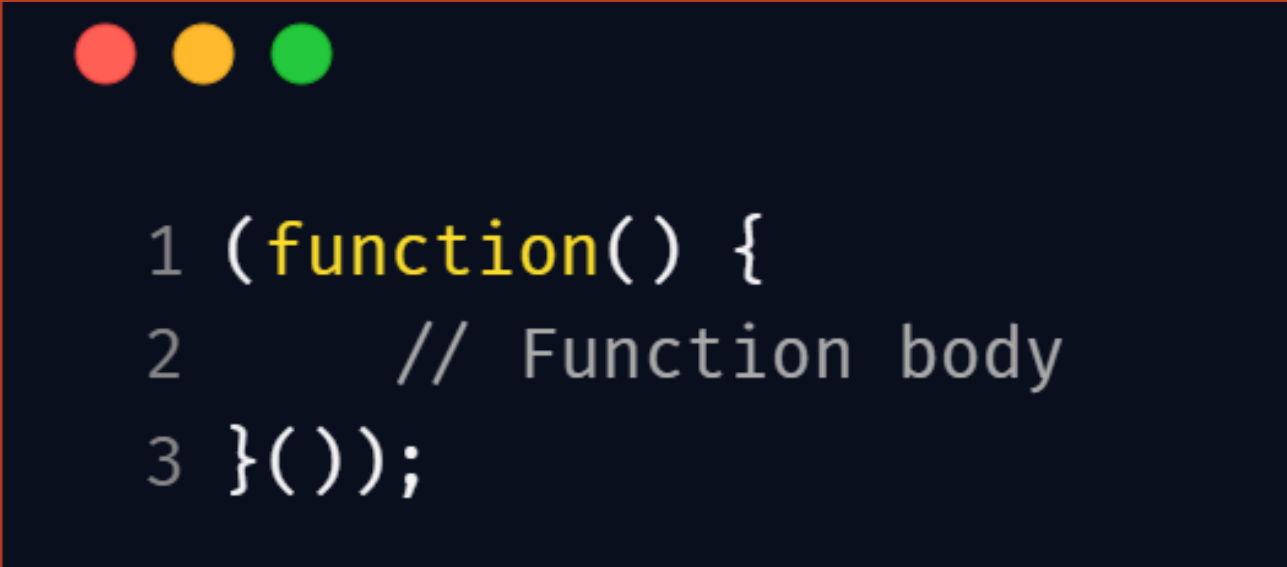
Argument 1

Argument 2

In this example, "**Alice**" is the first argument, which is assigned to the **name** parameter, and "**Welcome home**" is the second argument, assigned to the **message** parameter. The function then combines these into the greeting message.

Immediately Invoked Function Expressions (IIFE)


An Immediately Invoked Function Expression (**IIFE**) is a JavaScript function that is executed right after it is defined. The syntax typically looks like this:



```
1 (function() {  
2     // Function body  
3 }());
```


The key characteristic of an **IIFE** is that it runs immediately without needing to be called later.

One of the primary reasons to use an **IIFE** is to create a private scope. Variables declared inside an IIFE are not accessible from the outside, which helps in preventing them from polluting the global scope (**encapsulation**).



Since the function is executed right away, it's useful for initializing code that needs to run as soon as the script loads, without waiting for an explicit function call.

Example of an IIFE:



```
1 (function() {  
2     var message = "Hello, Oluwakemi!";  
3     console.log(message);  
4 })();  
5  
6 // Outputs: Hello, Oluwakemi!  
7 // The variable `message` is not accessible outside the IIFE.
```

In this example, the variable **message** is confined within the IIFE and cannot be accessed from the global scope.

This is a common pattern used in JavaScript to maintain clean, encapsulated code.



Higher Order Function

Higher-Order Functions are functions that either take other functions as arguments or return functions as their result.


This ability to pass functions around makes higher-order functions a key feature in functional programming.

They allow you to abstract common patterns in your code, making your code more flexible and easier to maintain.

By passing functions as arguments, you can reuse the same higher-order function with different behaviors. This promotes DRY (Don't Repeat Yourself) principles.

Higher-Order Functions can lead to more concise and readable code.

An Example of **HOF**



```
1 // First function
2 function multiplyByTwo(x) {
3     return x * 2;
4 }
5
6 // Second function (HOF)
7 function applyFunctionToArray(arr, func) {
8     return arr.map(func);
9 }
10
11 const numbers = [1, 2, 3, 4];
12 const doubledNumbers = applyFunctionToArray(numbers, multiplyByTwo);
13
14 console.log(doubledNumbers); // Outputs: [2, 4, 6, 8]
```

Function passed as an argument



In this example, **applyFunctionToArray** is a higher-order function because it takes another function (**multiplyByTwo**) as an argument and applies it to each element of the array.



Callback Function

A **callback** is a function that is passed as an argument to another function and is executed after the completion of that function.

Callbacks are a key aspect of handling asynchronous operations in JavaScript, such as API calls, timeouts, or event handling.

For instance, you don't want your code to proceed before receiving the data from an API call, so you pass a callback to handle the data once it arrives.

Since callbacks are just functions, they can be customized to perform any action you want after the completion of the initial function.

Example of a **callback function**



```
1 function fetchData(callback) {
2   // Simulating an API call with a timeout
3   setTimeout(function() {
4     const data = "Data fetched from API";
5     callback(data); // Calling the callback function with the fetched data
6   }, 2000);
7 }
8
9 function processData(data) {
10  console.log("Processing: " + data);
11 }
12
13 // Fetch data and then process it using a callback
14 fetchData(processData);
```

Explanation:

- **fetchData** is a function that simulates fetching data from an API after a delay.
- **callback** is the parameter that takes a function, which will be executed once the data is available.
- **processData** is passed as a callback to fetchData, so it will run once the data is "fetched."
- After 2 seconds, "**Processing:** Data fetched from API" will be logged to the console.



Asynchronous Function

Asynchronous functions are designed to handle operations that don't complete immediately, such as fetching data from a server, reading files, or waiting for a timer to finish.

Instead of blocking the execution of code while waiting for these operations, asynchronous functions allow the rest of the code to run and provide a way to handle the result of the asynchronous operation once it completes.

It is a promise-based function and can be defined in different ways like using the Promise keyword or `async/await`.

Using the Promise Keyword



A promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Here's an example:

```
1 function fetchData() {
2     return new Promise((resolve, reject) => {
3         setTimeout(() => {
4             const data = "Data fetched from API";
5             resolve(data); // Successfully fetched data
6         }, 2000);
7     });
8 }
9
10 fetchData()
11     .then(data => {
12         console.log("Processing: " + data);
13     })
14     .catch(error => {
15         console.error("Error: " + error);
16     });
```



Explanation:

- **fetchData** returns a promise that simulates an API call.
- **resolve(data)** is called if the operation is successful, passing the data to the next `.then()` block.
- **.catch()** handles any errors that might occur during the operation.

Using async/await



The **async** keyword is used to declare an asynchronous function, which allows you to use the **await** keyword inside it. **await** pauses the execution of the function until the promise is resolved, making asynchronous code look and behave more like synchronous code.

Here's an example:

```
1 async function fetchData() {
2     return new Promise((resolve, reject) => {
3         setTimeout(() => {
4             const data = "Data fetched from API";
5             resolve(data); // Successfully fetched data
6         }, 2000);
7     });
8 }
9
10 async function processData() {
11     try {
12         const data = await fetchData(); // Waits for fetchData to resolve
13         console.log("Processing: " + data);
14     } catch (error) {
15         console.error("Error: " + error);
16     }
17 }
18
19 processData();
```



Explanation:

processData is an async function that waits for `fetchData` to complete using `await`.

The code is cleaner and easier to read, resembling synchronous code while still handling asynchronous operations.

This provides a more readable and maintainable way to write asynchronous code compared to the traditional `.then()` and `.catch()` chaining of promises.



I hope you found this material
useful.

Remember to:

Like

Save for future reference

&

Share with your network, be
helpful to someone 🙌

Hi There!

Thank you for reading through

Did you enjoy this knowledge?



Follow my LinkedIn page for more work-life balancing and Coding tips.



LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app