JavaScript

Module 4 – Functions

Chapter 7 - Functions

Aitrich

What are Functions?

- Function is like a mini-program or a set of instructions that you can create and give a name.
- It's a way to group tasks, making your code more organized and easier to understand.

Benefits of using functions

1. Code organization
2. Reusability
3. Modularity

Function Syntax

- Basic structure: function functionName(parameters) { ... }

```
function functionName(parameter1, parameter2) {
    // function body
    // code to be executed
    return result; // optional
}
```

Parameters and Arguments

Passing data to functions

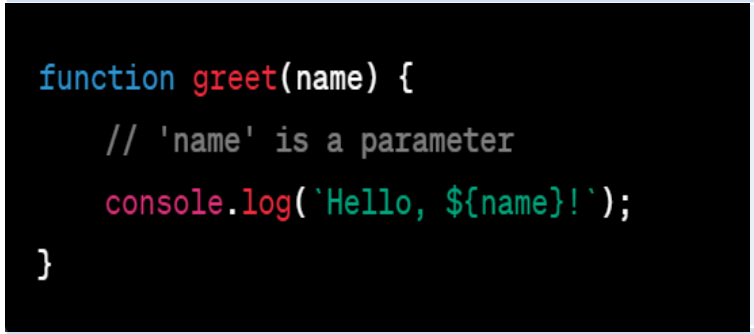
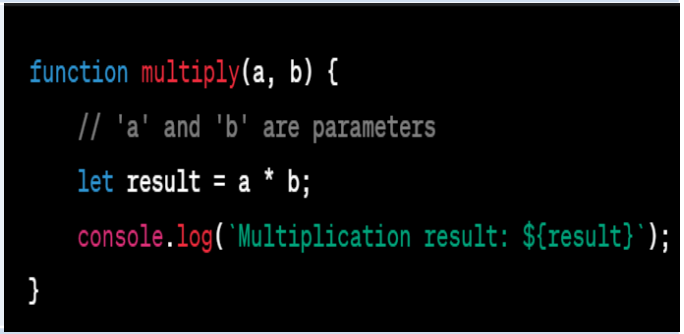
In JavaScript, you can pass data to functions through parameters. Parameters act as placeholders for values that will be supplied when the function is called.

```
// Function that takes two parameters
function addNumbers(num1, num2) {
    // Function body
    let sum = num1 + num2;
    console.log(`Sum of ${num1} and ${num2} is: ${sum}`);
}

// Calling the function with arguments
addNumbers(5, 8); // Output: Sum of 5 and 8 is: 13
```

- num1 and num2 are parameters, and when the function addNumbers is called with addNumbers(5, 8), the values 5 and 8 are passed as arguments, and the function performs the addition operation.

Difference between parameters and arguments

Parameters	Arguments
Definition: Parameters are variables listed in the function declaration.	**Definition:** Arguments are the actual values passed to a function when it is invoked.
	
Role: Parameters act as placeholders within the function, representing the values that will be passed during the function call.	**Role:** Arguments are the specific values that match the parameters of the function during the function call.

Returning Values

Returning a result from a function

In JavaScript, functions can produce and return values using the **return** statement.

Basic Return Statement

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

const sum = addNumbers(3, 5);
console.log(sum); // Output: 8
```

In this example, the **addNumbers** function returns the sum of its parameters, and the result is stored in the variable **sum**.

Returning Values

Returning Objects:

Functions can also return more complex data types like objects.

```
function createPerson(name, age) {
    return { name, age };
}

const person = createPerson("Alice", 25);
console.log(person); // Output: { name: 'Alice', age: 25 }
```

Returning Values

Returning Functions:

Functions can also return other functions, enabling the creation of higher-order functions.

Returning a result from a function allows for the creation of modular and reusable code. Returning a result from a function allows for the creation of modular and reusable code.

```
function multiplier(factor) {
    return function (number) {
        return number * factor;
    };
}

const multiplyByTwo = multiplier(2);
console.log(multiplyByTwo(5)); // Output: 10
```

Anonymous Functions

Anonymous functions, also known as function expressions, are functions without a named identifier. They are often used for short-lived operations or as arguments to higher-order functions.

Basic Anonymous Function

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

console.log(addNumbers(3, 5)); // Output: 8
```

addNumbers is an anonymous function assigned to a variable. It can be invoked using the variable name.

Arrow Functions

Arrow functions provide a concise syntax for anonymous functions. Arrow functions are especially useful for short, one-line functions.

```
const multiply = (a, b) => a * b;

console.log(multiply(2, 7)); // Output: 14
```

Arrow Functions

Use Cases:

- As arguments to higher-order functions.
- In-event listeners and callbacks.
- In situations where a function is used temporarily and doesn't need a name.
- Anonymous functions offer flexibility and are commonly employed in various JavaScript scenarios.

Function declaration
vs.
Function Expression

Function declaration

- **Syntax: function functionName() { /* code */ }**
- Declare a function using the **function** keyword.

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

Using a Declared Function

```
// Using the declared function
greet(); // Outputs: Hello, everyone!
```

To execute the function, call it by its name followed by parentheses.

Function Expression

Expression - Assigning a Function to a Variable

```
// Function Expression
var greetExpression = function() {
  console.log("Hello, students!");
};
```

- **Syntax: var variableName = function() { /* code */ };**
- Assign a function to a variable.

Using a Function Expression

Call the function using the variable name followed by parentheses.

```
// Using the function expression
greetExpression(); // Outputs: Hello, students!
```

Key Differences

Declaration: Created using the **function** keyword.

Expression: Function is assigned to a variable.

Hoisting

Declaration: This can be called before the declaration in the code.

Expression: This should be declared before usage.

When to Use Which?

Declaration: Use when you need the function to be available throughout your code.

Expression: Use when you want to assign a function to a variable or need a function locally within a block.

What is Function Invocation?

Function Invocation is like calling a function to perform its set of tasks.

How to Invoke a Function

After declaring a function, you can use it by invoking or calling it.

```
// Example function declaration
function greet() {
  console.log("Hello, students!");
}
```

Invoking The Function

To make the function run its code, you invoke it by using its name followed by parentheses.

```
// Invoking the function
greet(); // Outputs: Hello, students!
```

Parameters in Function Invocation

Functions can take inputs called parameters. When invoking, provide values for these parameters.

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

Passing Parameters

When you invoke a function with parameters, supply the values inside the parentheses.

```
greetWithName("Alice"); // Outputs: Hello, Alice!
```

Scope and closures in functions.

What is Scope?

Definition: Scope refers to the area where a variable is accessible.

Think of it like the visibility or reach of a variable.

Local Scope

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

```
function exampleFunction() {
  var localVar = "I am local!";
  // localVar is only visible inside exampleFunction
}
```

Global Scope:

Variables declared outside any function are accessible throughout the entire code.

```
var globalVar = "I am global!";
// globalVar is visible everywhere in the code
```

What are Closures?

Definition: A closure is a function that "closes over" its lexical scope, retaining access to variables even after the scope has finished executing.

```
function outerFunction() {
  var outerVar = "I am from outer!";

  function innerFunction() {
    console.log(outerVar); // Outputs: I am from outer!
  }

  return innerFunction;
}

var closureFunction = outerFunction();
closureFunction(); // Outputs: I am from outer!
```

- **innerFunction** "closes over" the scope of **outerFunction**.
- Even after **outerFunction** finishes, **innerFunction** still has access to **outerVar**.

Thank You