TASK

Functions, Scope, and Closures

Visit our website

Introduction

WELCOME TO THE FUNCTIONS, SCOPE, AND CLOSURES TASK!

A function is a unit/block of code that contains all the instructions needed to complete a specific task. Functions allow you to split a complex task into simpler tasks, which makes managing and maintaining scripts easier. Another benefit of using functions is that functions are the means by which we can restructure our code to minimise repetition and reduce potential errors. Functions are little blocks of code which we can call on repeatedly from the body of our code that enable us to avoid repeating the same lines of code needlessly.

Functions can either be **user-defined** or **built-in**. Built-in functions are built into the JavaScript language itself and are readily available for us to use.

BASIC JAVASCRIPT FUNCTIONS

Declaring a function in JavaScript involves providing a function name, followed by a list of parameters enclosed in parentheses `( )`, and the function body enclosed within curly braces `{ }`. Here's the basic syntax:

```
function functionName(parameter1, parameter2, ...parameterN) {
  // function body
  // statements defining what the function does
}
```

Here's a basic example of a function in JavaScript:

```
function greet() {
  console.log('Hello, World!');
}
```

In this case, `greet` is a function that prints out 'Hello, World!' to the console. You can **call** this function using its name followed by parentheses `()` like this: **`greet()`**.

JAVASCRIPT FUNCTION KEY COMPONENTS

A JavaScript function has three key components:

- **Parameters:** These are variables listed as a part of the function definition. They act as placeholders for the values on which the function operates, known as arguments.
- **Function Body:** Enclosed between curly braces `{ }`, the function body consists of statements that define what the function does.
- **Return Statement:** how a function sends the result of its operations back to the caller. Not all functions have to return a value; those that don't are often used for their side effects, such as modifying the global state or producing output.

```
function addNumbers(num1, num2) {  // num1 and num2 are parameters
  let sum = num1 + num2; // function body
  return sum; // return statement
}
```

In the above example, `num1` and `num2` are parameters, the lines of code within `{ }` constitute the function body, and the `return sum;` is the return statement.

CALLING A FUNCTION

After a function has been declared, it can be invoked or called anywhere in your code by using its name followed by parentheses `()`. If the function requires parameters, you'll include those within the parentheses. Each argument corresponds to the position of the parameter in the function declaration. To call the function `greet`, we would do this:

```
function greet(name) {  // Function Declaration
  console.log(`Hello, ${name}`); // To print the result in console
}
greet('Alice');  // Call the greet function // Output: Hello, Alice
```

In this case, the string 'Alice' is an argument that's passed into the `greet` function. When the function is called, it replaces the name parameter with 'Alice' and executes the function body.

Here's another example, a variant on the sum function above, where we again have a function with two parameters. (How does this function differ from the first addNumbers function you saw on the previous page?):

```
function addNumbers(num1, num2) {  // Function Declaration
  console.log(num1 + num2);
}

addNumbers(5, 10);  // Outputs: 15 // Function Calling
```

In this example, `addNumbers` is a function that takes two parameters, `num1` and `num2`. When we call `addNumbers(5, 10)`, the function takes those arguments, adds them together, and outputs the resulting value to the console..

Understanding function declarations and calls is a key part of mastering JavaScript, enabling you to write modular, reusable code that keeps your programs **DRY** (Don't Repeat Yourself) and efficient. Read more about the DRY principle **here**.

PARAMETERS VS ARGUMENTS: REVIEWING THE DIFFERENCE?

The primary difference between parameters and arguments lies in where they show up in the code:

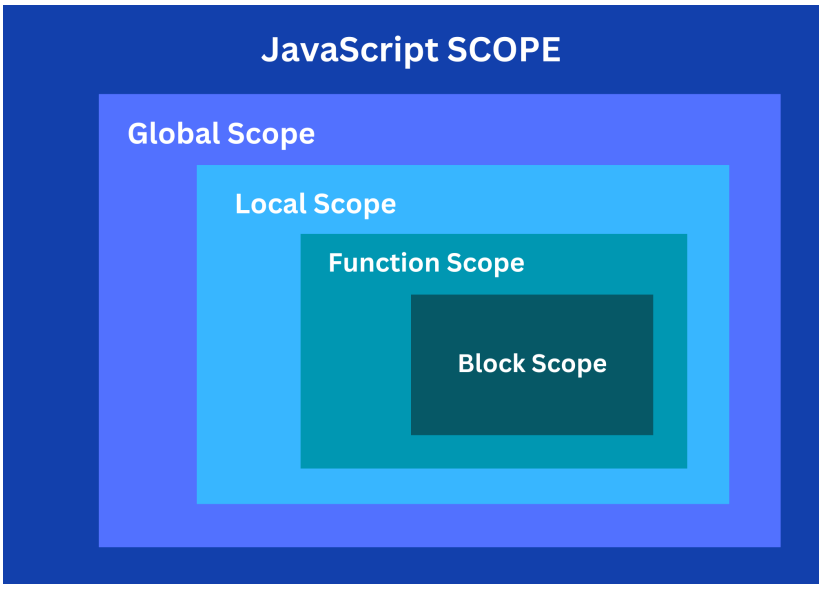
Parameters	Arguments
Parameters are used when defining a function. They represent the "input" the function needs to do its job, and they act as placeholders for actual data.	Arguments are used when calling a function. They represent the actual "input" that will be operated on by the function's code.

SCOPE

Scope is what we call a program's ability to find and use variables in a program. Another way to look at it is that the scope of a variable determines where in the code it can be seen from. The rule of thumb is that a function is covered in one-way glass: it can see out, but no one can see in. This means that a function can call variables that are *outside* the function, but the rest of the code cannot call variables that are defined *inside* the function.

TYPES OF SCOPE

JavaScript primarily has three types of scope: Global Scope, Function Scope, and Block Scope. Function and Block Scopes both fall into the category of Local Scope. Let's look at each in more detail:



Global scope

When a variable is declared outside all functions or block scopes, its scope is global. Global variables can be accessed from any part of the code, whether within a function or outside.

```
let globalVar = "I am global!"; // This is a global variable
```

```
function testScope() {
  console.log(globalVar); // Outputs: I am global!
}

testScope();
```

Function scope

Variables declared within a function are accessible only within the function body and are said to have function scope. They cannot be accessed outside of the function in which they are declared.

```
function testScope() {
  let localVar = "I am local!";
  console.log(localVar); // Outputs: I am local!
}

testScope();
console.log(localVar); // Uncaught ReferenceError: localVar is not
defined
```

Block scope

The introduction of the `let` and `const` keywords in ES6 introduced block scope into JavaScript. Variables declared with `let` or `const` are scoped to the block in which they are declared (unlike variables declared with `var`, which is scoped to the function).

```
if (true) {
      let blockVar = "I am block scoped";
    console.log(blockVar); // Outputs: I am block scoped
}

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

SPOT CHECK 1

Let's see what you can remember from this section.

1. What is the difference between parameter and argument?
2. What is the difference between global scope and local scope in JavaScript?

NESTED FUNCTIONS

A nested function, as the name suggests, is a function that is defined inside another function. For example, consider this code sample:

```
function multiplier(x) {
  function inside(y) {
    return x * y;
  }
  return inside;
}
```

Above, `inside()` is a nested function within the function `multiplier()`. You can learn more about nested functions **here**.

CLOSURES

Closures in JavaScript are a concept where an inner function has access to an outer (enclosing) function's variables and parameters, even after the outer function has finished executing. This is possible due to the nature of JavaScript's lexical scope. **A function, along with its lexical scope, forms a closure.** Let's look at an example. Read through the code carefully, even if you don't understand it at this point.

```
function multiplier(x) {
  function inside(y) {
    return x * y;
  }
  return inside;
}

const timesThree = multiplier(3); // Here we create a closure
console.log(timesThree(5));
```

In the above code example, when we call the `multiplier` function, it returns the `inside` function. Now, even though the execution of the `multiplier` function is complete, the returned `inside` function (now in the `timesThree` constant) still has access to the `multiplier`'s variable (`x`). This is a closure (the function along with its

lexical scope). By calling **timesThree** with an argument (in this case, 5), the **inside** function is run with the provided argument for its **y** parameter. The earlier call to **multiplier** provided an **x** value and the call to **inside** (via **timesThree**) has provided a **y** value, and so **x*y** from the **inside** function can be calculated and the result returned.

The example below is the same as the original example, with the addition of a step that outputs the **timesThree** const to help you see what is meant when we say that the **inside** function is returned and assigned to const. The code is also extensively commented on to help you identify exactly which parts are doing what.

```
function multiplier(x) {
  function inside(y) {
    return x * y;
  }
  return inside;
}

const timesThree = multiplier(3); // Here we create a closure

console.log(timesThree) /* Outputs the content of timesThree; as the
inside function has been returned by the call to multiplier on the
previous line, the content of timesThree is now identical to the code
for the inside function. You can thus call timesThree with a numeric
argument, like: timesThree(5); and the inside function will be run using
the 5 as its argument for y. The earlier call to multiplier in which we
created the closure provided an argument value for x, and so now x*y can
be calculated and returned by the 'inside' function. Let's try running
this and logging the output to the console*/

console.log(timesThree(5)); /* Even though multiplier has finished
executing, inside still has access to x, and so runs x*y, i.e. 3*5, and
logs 15 to the console */
```

In simple terms, you can think of a closure as a backpack that the nested **inside** function carries around wherever it goes. This backpack has all the variables (**x** in this case) that **inside** had access to in its original environment when it was created.

Let's review and summarise briefly. In the above example, the **multiplier** function takes an argument, **x**, and returns a new function, **inside**. This inner function takes another argument **y** and returns the product of **x** and **y**. Then you can use the

`multiplier` function to create new functions like `timesThree`, which multiplies its input by 3.

The part of the code that calls `multiplier` and then `inside`:

```
const timesThree = multiplier(3); // Think of it like: give me a
function that multiplies 3 with whatever you give it
console.log(timesThree(5)); // 15
```

could also be replaced with:

```
// We can also achieve the same result as following
console.log(multiplier(3)(5)); // 15
```

Closures can be really confusing for novices, so if you're still unsure of how they work, don't worry! A good code-along explanation that often helps learners gain clarity is provided in the first 10 minutes of **this video**. Another take on the subject provided comes from the popular JavaScript YouTuber **Akshay Saini**, and there are a number of other explanatory videos to be found online. If you're really battling with the concept of closures, try watching two or three different videos and working through the examples they provide first, and then returning to this section of the task and reading through, and running, the examples again.

QUICK INTRODUCTION TO ARROW FUNCTIONS

Arrow functions, a relatively recent addition to JavaScript, are a new way to write compact and clear functions that can simplify your code. First introduced with ES6, they provide a more concise syntax and a change in how **the keyword 'this'** is handled in the function scope.

What are arrow functions in JavaScript?

Arrow functions in JavaScript are a shorthand syntax for writing function expressions. They're called 'arrow' functions because of the `=>` symbol used, which resembles an arrow. They can make our code cleaner and easier to read when used

correctly.

Syntax of arrow functions

The basic syntax of an arrow function is as follows:

```
let functionName = (parameter1, parameter2, ...parameterN) => expression
```

This creates a function named **functionName** that accepts parameters **parameter1**, **parameter2**, etc., through to **parameterN**, and returns the result of the expression. You can read this in English as something like "let **functionName** take **parameter1, parameter2**, etc, and then calculate **expression** with those parameters and return the result".

For example, here's a very basic arrow function:

```
let add = (a, b) => a + b;
console.log(add(1, 2)); // Output: 3
```

This could be articulated in English as "let the **add** function take parameters **a** and **b**, and then add their values together and return the result." The **let** keyword is not strictly necessary - try copying and pasting the code in the above example and running it both with and without **let**, and see what happens.

An arrow function doesn't *have* to take any parameters. In such a case, you retain the parentheses, but without anything inside them, as shown below. Notice that if you output the **greeting** function without parentheses, you get the content (code) of the arrow function. To actually call the function and output the result, you need to include the parentheses in the function call.

```
let greeting = () => "Hello, World!";
console.log(greeting); // Output: () => "Hello, World!"
console.log(greeting()); // Output: Hello, World!
```

If you only have one parameter, you can even skip the parentheses! Here's an example of a one-parameter arrow function *with* parentheses:

```
let greeting = (name) => "Hello, " + name;
console.log(greeting("Bob")); // Output: Hello, Bob
```

And here's the same arrow function *without* the parentheses:

```
let greeting = name => "Hello, " + name;
console.log(greeting("Bob")); // Output: Hello, Bob
```

Try running them both and see what happens.

A note from the

HyperionDev Team

Here you can find resources that provide additional helpful information about **JS Functions**.

Instructions

Read and run the accompanying example files provided before doing the task to become more comfortable with the concepts covered in this task.

Compulsory Task 1

Follow these steps:

- Define a function **findSum** that takes an array of integers and returns their sum.
- Define a function **subtractNumbers** that subtracts the second number from the first number.
- Define a function **multiplyNumbers** that multiplies two numbers.
- Define a function **divideNumbers** that divides the first number by the second number, handling the case where the second number is 0.

Now, create an array with 3 integers.

- Use the array to call the **findSum** function and log its return value.
- Use the first and second number from the array to call **subtractNumbers** and log its return value.
- Use the third number and the first number from the array to call **multiplyNumbers** and log its return value.
- Call **divideNumbers** using the sum of all three numbers obtained from findSum and the third number from the array. Log its return value.

Remember, in this Compulsory Task, you're defining your own functions and using them to perform operations on an array of numbers. Be careful to handle edge cases, like division by zero, in your functions using **if** statements before division.

Compulsory Task 2

Follow these steps:

- Create a new JavaScript file in this folder called **digitalHideSeek.js**.
- Define a function **hide** that takes in a string as an argument, representing a hiding location. This function should store the location in a local variable **hideLocation**.
- Inside the **hide** function, define another function **seek** that returns the hidden location when called.
- The **hide** function should return the **seek** function, creating a closure around **hideLocation**.
- Now, call **hide** with a string argument describing your hiding spot and assign the return value (which is the **seek** function) to a new variable called **startGame**.
- Log the result of calling **startGame**. This should print your hiding location, demonstrating the concept of a closure.
- Try logging **hideLocation** directly from outside of the **hide** and **seek** functions. Observe the result and explain why you think this happens, demonstrating your understanding of scope.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

SPOT CHECK 1 ANSWERS

1. **Parameters** are used when defining a function. They represent the "input" the function needs to do its job and act as placeholders for actual data.

2. **Arguments** are used when calling a function. They represent the actual "input" that will be operated on by the function's code.