Task

Functions, Scope, and Closures

Visit our website

Introduction

WELCOME TO THE FUNCTIONS, SCOPE, AND CLOSURES TASK!

A function is a block of code designed to perform a specific task. Using functions allows you to split complex tasks into simpler ones, making scripts easier to manage and maintain. They help minimise repetition and reduce potential errors by enabling you to call the same block of code multiple times. 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.

This task will also cover scope and closures. Understanding scope is crucial, as it defines where variables can be accessed within your code. Functions control variable access by creating local scopes, which help prevent naming conflicts and unintended interactions between variables. Closures, on the other hand, allow functions to retain access to their scope even after they have finished executing, enabling powerful programming patterns such as data encapsulation and state management. By grasping functions, scope, and closures, you will write more efficient, maintainable, and error-free code.

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()`. We'll touch further on how to call this functionlater in this document.

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 an output.

```
// In the function below num1 and num2 are parameters
function addNumbers(num1, num2) {
  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) {

   console.log(`Hello, ${name}`); // Print greeting based on the name
parameter.

}

greet("Alice"); // Call the greet function to 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:

```
function addNumbers(num1, num2) {

   console.log(num1 + num2); // Log the sum of num1 and num2 to the
console.

}

addNumbers(5, 10); // Calling the addNumbers function with five and ten
as arguments
```

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.

Parameter versus arguments

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

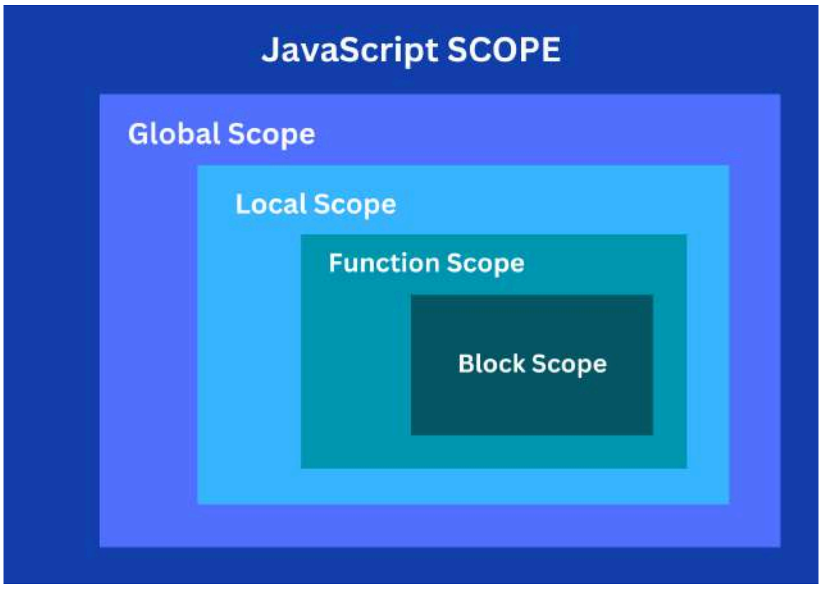
Parameters	**Arguments**
Parameters are used when defining a function. They represent the 'input' the function needs to do its job, and they act as placeholders for actual data.	Arguments are used when calling a function. They represent the actual 'input' that will be operated on by the function's code.

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.

```javascript
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 the function scope. They cannot be accessed outside of the function in which they are declared. Attempting to access a function-scoped variable from outside the function will result in a reference error.

```
function testScope() {
   // localVar is a local variable, so it is only available inside the
testScope function
  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 confined to the block in which they are declared (unlike variables declared with `var`, which are function-scoped). Attempting to access block-scoped variables outside their block results in a reference error, as they are only accessible within the block where they were defined.

```
if (true) {
  // blockVar is only accessible in this block
  let blockVar = "I am block scoped";
  console.log(blockVar); // Outputs: I am block scoped
}
console.log(blockVar);  //  Uncaught  ReferenceError:  blockVar  is  not
defined
```

NESTED FUNCTIONS

A nested function is a function defined inside another function. This allows the inner function to access variables and parameters of the outer function. Nested functions help organise code and keep related functionality together, making code more modular and maintainable:

```
// The outer function "multiplier" that takes one parameter "x".

function multiplier(x) {

  // The nested function "inside" which takes one parameter "y".

  function inside(y) {

    // Return the product of "x" and "y".

    return x * y;

  }

    // Return the "inside" function. It has access to "x" from "multiplier".

  return inside;

}

// Example usage:

const multiplyByTwo = multiplier(2);

console.log(multiplyByTwo(5));
```

In the provided code, **`inside`** is a nested function within the **`multiplier`** function. When **`multiplier`** is called, it doesn't immediately perform a calculation; instead, it returns the **`inside`** function. This returned nested function can then be used later, allowing you to apply further operations. However, the nested **`inside`** function is still able to access the parameter **`x`** and the argument which was passed to the **`multiplier`** function.

For instance, if you call **`multiplier(4)`**, the number **`4`** is passed as an argument and is stored in the parameter **`x`**. The multiplier function would then return the inside function, which can use that stored value of **`x`** whenever it is called later. This means that even when the multiplier function is no longer running, the inner function still has access to the value of x.

This ability for the inner function to still retain access to the outer function is known as a **closure**. Closures allow the inner function to continue using variables from its outer function even after the outer function has finished executing. We'll explore closures in more detail later, but for now, it's important to understand that nested functions can have this special ability. You can also learn more about nested functions **here**.

CLOSURES

Closures in JavaScript occur when an inner function has access to an outer function's variables and parameters, even after the outer function has finished executing. This is due to JavaScript's lexical scope, where a function retains access to its scope even after the function that created it has finished. Here's a step-by-step explanation of the closure example. Read through the code carefully:

1. Defining the `multiplier` function:

```
// The outer function "multiplier" that takes one parameter "x".

function multiplier(x) {

  // The nested function "inside" which takes one parameter "y".

  function inside(y) {

    // Return the product of "x" and "y".

    return x * y;

  }

    // Return the "inside" function. It has access to "x" from
"multiplier".

  return inside;

}

// Example usage:

const multiplyByThree = multiplier(3); // Creating a closure.
```

```
console.log(multiplyByThree(5)); // Outputs 15 which is 3 * 5.
```

- The **`multiplier`** function takes one parameter **x**.
- Inside **`multiplier`**, there is another function **`inside`** that takes on the parameter **y**.
- The **`inside`** function returns the product of **x** and **y**.
- The **`multiplier`** function returns the **`inside`** function.

2. Creating a closure:

```
const multiplyByThree = multiplier(3); // Creating a closure.
```

- We call **`multiplier(3)`**, passing **3** as the argument for **x**.
- The **`multiplier`** function returns the **`inside`** function, with **x** set to **3**.
- The returned **`inside`** function is stored in the constant **`multiplyByThree`**.

3. Calling the closure:

```
console.log(multiplyByThree(5)); // Outputs 15 which is 3 * 5.
```

- We call **`multiplyByThree(5)`**, which is the **`inside`** function with **x** set to **3**.
- Inside **`multiplyByThree`**, **y** is **5** (the argument passed).
- The function calculates **`x * y`**, which is **`3 * 5`**.
- The result, **15**, is returned and logged to the console.

Summary:

- **`multiplier(3)`**: Returns a function (**`inside`**) with access to **`x = 3`**.
- **`multiplyByThree(5)`**: Uses **`x = 3`** and **`y = 5`** to compute **`3 * 5`**, resulting in **15**.

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

```javascript
// The outer function "multiplier" that takes one parameter "x".

function multiplier(x) {

  // The nested function "inside" which takes one parameter "y".

  function inside(y) {

    // Return the product of "x" and "y".

    return x * y;

  }

    // Return the "inside" function. It has access to "x" from "multiplier".

  return inside;

}

// Example usage:

// Create a closure with x = 3

const multiplyByThree = multiplier(3);

/* logs the function itself. multiplyByThree is the inside function with x set to 3. */

console.log(multiplyByThree)

/* Calls the inside function with y = 5. Since x was set to 3, it calculates 3 * 5 and logs 15. */

console.log(multiplyByThree(5));
```

Think of a closure as a backpack that the inner function carries with it. This backpack holds all the variables (like x) that the function had access to when it was created.

In the example, `multiplier` takes an argument `x` and returns a new function, `inside`, which multiplies x by its argument y. You can then use `multiplier` to create functions like `multiplyByThree`, which multiplies its input by 3.

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

```
// Think of it like: give me a function that multiplies three with whatever you give it

const multiplyByThree = multiplier(3);

console.log(multiplyByThree(5)); // Output: 15 (3 * 5)
```

- When `multiplier(3)` is called, the `multiplier` function executes and returns the `inside` function.
- The returned `inside` function still retains access to the parameter `x` from the `multiplier` function. In this case, `x` is `3`.
- When the returned function is called with `multiplyByThree(5)`, the argument `5` is passed to the `inside` function as `y`.
- Within the `inside` function, the expression `x * y` would become `3 * 5`, which evaluates to `15`.

The above instance could also be replaced with:

```
// We can also achieve the same result as following

console.log(multiplier(3)(5)); // 15
```

- This single line works by directly calling the `multiplier` function with `3`, which returns the `inside` function.
- Immediately after, it calls the returned `inside` function with `5` as the argument.
- The `inside` function, then calculates the product of `3 * 5`, which is `15`.

Closures can be tricky at first, so if you're finding them confusing, that's perfectly okay! Keep practising and revisiting the concept, and it will become clearer over time.

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. You can learn more about arrow functions **here**.

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 instance, 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 when you output the `greeting` function without parentheses, the result depends on the environment. In most browsers, you typically see the code of the arrow function itself, while in Node.js, it usually displays the function type along with its name, such as `[Function: greeting]`. To actually execute the function and see its return value, you need to include parentheses when calling it, like this: `greeting()`.

```
let greeting = () => "Hello, World!";

console.log(greeting); // Output: [Function: greeting] or () => "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.

Instructions

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

Auto-graded task

Take note

The tasks below are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the tasks your best attempt and submit them when you're ready.

After you click "Request review" on your student dashboard, you will receive a 100% pass grade if you've submitted the tasks.

When you submit the tasks, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for these tasks, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

Auto-graded task 1

Follow these steps:

- Create a new JavaScript file in this folder called **arithmetic.js**.
- 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 zero.

Now, create an array with three 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, 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.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded 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.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

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.