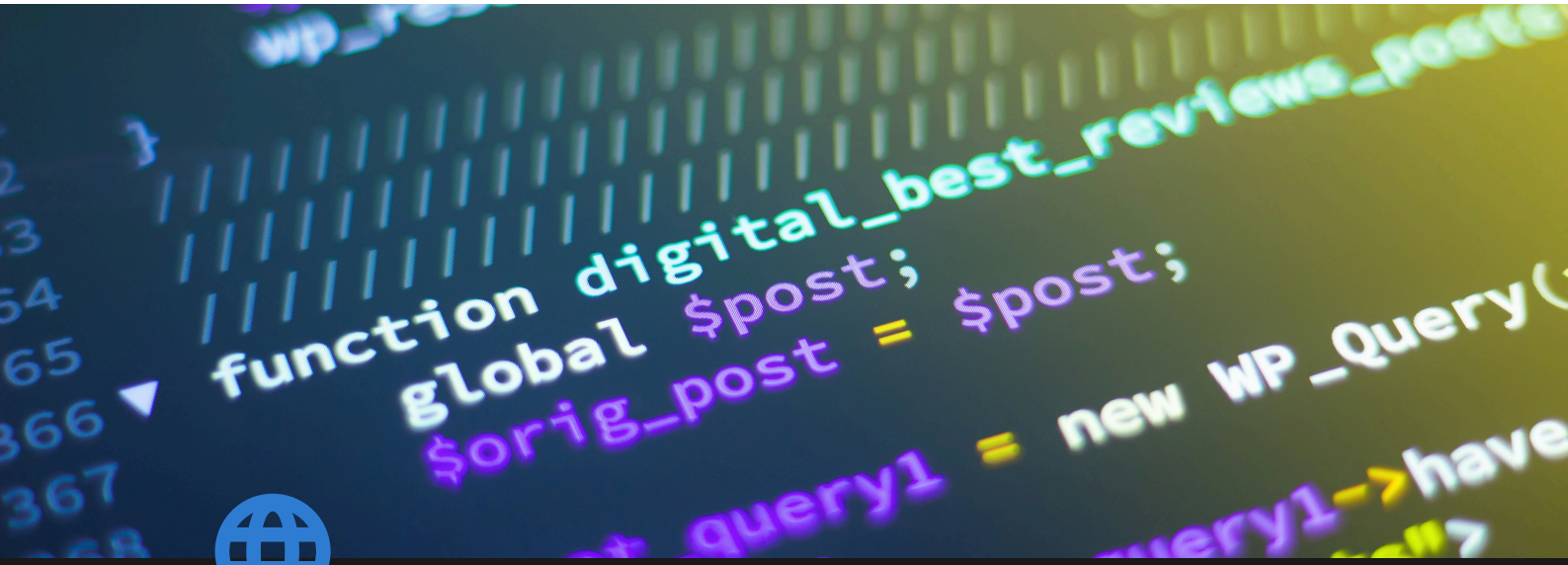


JS | Functions, Scope, Hoisting & Closures

Intended for Beginners

Agenda

1. Introduction to Functions in JavaScript (40 minutes)
2. Understanding Scope (30 minutes)
3. Hoisting in JavaScript (30 minutes)
4. Closures in JavaScript (30 minutes)

---

1. Introduction to Functions in JavaScript

What is a Function?

A function is a block of code designed to perform a specific task. You can define a function once and then call it multiple times, possibly with different inputs (called arguments). Functions help reduce code repetition and improve readability.

Function Definition

A function definition is how you create a function. In JavaScript, functions can be defined in several ways, but the two most common methods are **Function Declaration** and **Function Expression**.

Function Declaration

A **Function Declaration** is a way to define a function using the `function` keyword, followed by a name, parentheses `()` for parameters, and curly braces `{}` for the function body.

```
function greet(name) { return `Hello, ${name}!`; }
```

Here, `greet` is the function's name, `name` is the parameter, and the function returns a greeting string.

Example:

```
console.log(greet('Alice')); // Output: Hello, Alice!
```

Function Expression

A **Function Expression** involves defining a function inside an expression. This can be done by assigning a function to a variable.

```
const greet = function(name) { return `Hello, ${name}!`; };
```

In this example, the function is anonymous (has no name) and is assigned to the variable `greet`.

Example:

```
console.log(greet('Bob')); // Output: Hello, Bob!
```

Key Differences:

- **Hoisting:** Function declarations are hoisted to the top of their scope, meaning they can be called before they are defined in the code. Function expressions are not hoisted; you cannot use them before they are defined. Read more about Hoisting in the latter section.

Exercise 1:

Define a function using both a function declaration and a function expression that takes a number as input and returns its square.

```
// Function Declaration function square(num) { return num * num; } consol
e.log(square(4)); // Output: 16 // Function Expression const square = fun
ction(num) { return num * num; }; console.log(square(4)); // Output: 16
```

---

2. Scope in JavaScript

Scope in JavaScript refers to the accessibility of variables and functions in different parts of the code. Understanding scope is crucial for writing clean and bug-free code.

Global Scope

Variables declared outside of any function or block are in the **Global Scope**. They can be accessed from anywhere in the code.

```
let globalVar = "I'm global"; function checkGlobal() { console.log(global
Var); } checkGlobal(); // Output: I'm global
```

Function Scope

Variables declared inside a function are in the **Function Scope**. They can only be accessed within that function.

```
function checkFunctionScope() { let functionVar = "I'm inside a functio
n"; console.log(functionVar); } checkFunctionScope(); // Output: I'm insi
de a function // console.log(functionVar); // Error: functionVar is not d
efined
```

Block Scope

Variables declared inside a block (using `let` or `const`) are in the **Block Scope**. They are only accessible within that block.

```
if (true) { let blockVar = "I'm inside a block"; console.log(blockVar);
// Output: I'm inside a block } // console.log(blockVar); // Error: block
Var is not defined
```

Exercise 2:

Write a function that defines a variable inside a block (e.g., an `if` statement). Try accessing the variable outside the block and observe what happens.

```
function blockScopeTest() { if (true) { let blockVar = "Inside block"; co
nsole.log(blockVar); // Output: Inside block } // console.log(blockVar);
// Error: blockVar is not defined } blockScopeTest();
```

---

3. Hoisting in JavaScript

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that you can use functions and variables before they are declared in the code.

Function Hoisting

As mentioned earlier, function declarations are hoisted to the top of their scope.

```
greet('Charlie'); // Output: Hello, Charlie! function greet(name) { retur
n `Hello, ${name}!`; }
```

Variable Hoisting

Variable declarations (`var`) are hoisted, but their initialization is not.

```
console.log(myVar); // Output: undefined var myVar = 5; console.log(myVa
r); // Output: 5
```

In the code above, `myVar` is hoisted, but only the declaration, not the value `5`.

`let` and `const` Hoisting

Variables declared with `let` and `const` are also hoisted, but they are not initialized until the code execution reaches the line where they are defined. This leads to the **Temporal Dead Zone (TDZ)**, a period where the variable exists but is not yet initialized, and any attempt to access it will result in a `ReferenceError`.

```
// console.log(myLet); // Error: Cannot access 'myLet' before initializat
ion let myLet = 10; console.log(myLet); // Output: 10
```

The **Temporal Dead Zone (TDZ)** ensures that variables are not accessed before they are properly initialized, reducing potential errors and making the code more predictable.

Exercise 3:

Experiment with variable hoisting by declaring variables with `var`, `let`, and `const` and trying to access them before their declaration.

```
console.log(varVar); // Output: undefined var varVar = 5; console.log(let
Var); // Error: Cannot access 'letVar' before initialization let letVar =
10; console.log(constVar); // Error: Cannot access 'constVar' before init
ialization const constVar = 15;
```

Real-World Example

Imagine you're building a web application where you need to greet users based on the time of day. You can write a function to handle this logic and then use it throughout your app.

```
function greetUser(username) { const currentHour = new Date().getHours();
let greeting; if (currentHour < 12) { greeting = 'Good Morning'; } else i
f (currentHour < 18) { greeting = 'Good Afternoon'; } else { greeting =
'Good Evening'; } return `${greeting}, ${username}!`; } console.log(greet
User('David')); // Output depends on the time of day
```

This function can be called whenever a user logs in or interacts with the app, providing a personalized greeting.

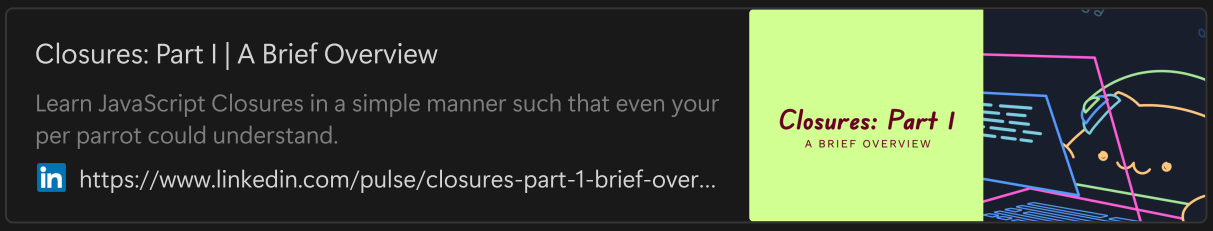
Exercise 4:

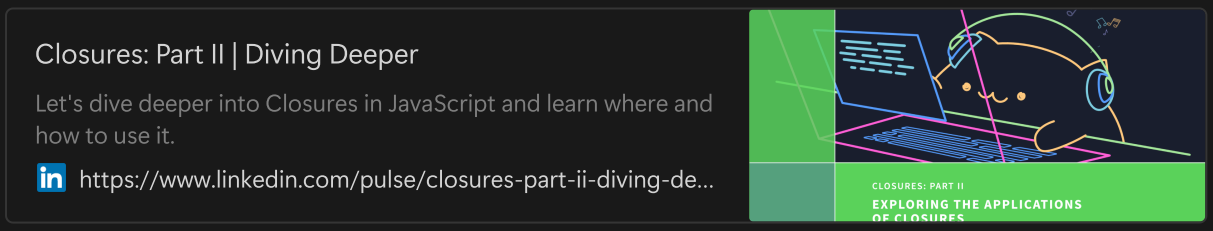
Extend the `greetUser` function to include a specific message for users who log in during the night (from 12 AM to 6 AM).

```
function greetUser(username) { const currentHour = new Date().getHours(); let greeting; if (currentHour < 6) { greeting = 'Why are you up so early?'; } else if (currentHour < 12) { greeting = 'Good Morning'; } else if (currentHour < 18) { greeting = 'Good Afternoon'; } else { greeting = 'Good Evening'; } return `${greeting}, ${username}!`; } console.log(greetUser('David')); // Output depends on the time of day
```

---

4. Closures in JavaScript





1. What is a Closure?

A closure is created when a function is defined inside another function, and the inner function has access to the outer function's variables. The inner function can access the outer function's variables even after the outer function has returned.

Example:

```
function outerFunction(outerVariable) { return function innerFunction(inn
erVariable) { console.log(`Outer variable: ${outerVariable}`); console.lo
g(`Inner variable: ${innerVariable}`); }; } const newFunction = outerFunc
tion('outside'); newFunction('inside'); // Output: // Outer variable: out
side // Inner variable: inside
```

In this example, `innerFunction` forms a closure, capturing the `outerVariable` from `outerFunction`.

---

2. Use Cases for Closures

Closures are useful in various scenarios, such as:

1. **Data Privacy**: Closures can be used to create private variables or methods.

```
function createCounter() { let count = 0; return { increment: function()
{ count++; return count; }, decrement: function() { count--; return coun
t; } }; } const counter = createCounter(); console.log(counter.increment
()); // Output: 1 console.log(counter.increment()); // Output: 2 console.
log(counter.decrement()); // Output: 1
```

Here, `count` is not directly accessible from outside the `createCounter` function. It is only accessible through the `increment` and `decrement` methods, effectively making `count` a private variable.

1. **Function Factories**: Closures can be used to create functions with preset configurations.

```
function createGreeting(greeting) { return function(name) { return `${gre
eting}, ${name}!`; }; } const greetHello = createGreeting('Hello'); const
greetHi = createGreeting('Hi'); console.log(greetHello('Alice')); // Outp
ut: Hello, Alice! console.log(greetHi('Bob')); // Output: Hi, Bob!
```

In this case, `createGreeting` is a function factory that creates greeting functions with a preset greeting message.

1. **Maintaining State**: Closures allow functions to maintain state between calls.

```
function makeMultiplier(multiplier) { return function(number) { return nu
mber * multiplier; }; } const double = makeMultiplier(2); const triple =
makeMultiplier(3); console.log(double(5)); // Output: 10 console.log(trip
le(5)); // Output: 15
```

The `makeMultiplier` function creates functions that remember the multiplier value, allowing the returned functions to use that value even after `makeMultiplier` has completed.

---

3. The `this` Keyword and Closures

When using closures, it's important to understand how the `this` keyword works, as it might not behave as expected. In a closure, the value of `this` is determined by the context in which the closure is invoked, not where it was created.

Example:

```
const obj = { name: 'Alice', getName: function() { return () => this.nam
e; } }; const getNameFunc = obj.getName(); console.log(getNameFunc()); //
Output: Alice
```

In this example, the arrow function inside `getName` forms a closure, capturing the value of `this` from `getName

Summary

We covered the basics of functions in JavaScript, including:

- **Function Definition:** How to define functions using declarations and expressions.
- **Scope:** Understanding global, block, and function scope.
- **Hoisting:** How JavaScript handles variable and function declarations before execution, and the concept of the **Temporal Dead Zone (TDZ)** to prevent accessing uninitialized variables.

Additional Exercises:

1. Write a function that calculates the factorial of a number using recursion.
2. Create a function that returns a random number between a specified range.

Practice these concepts with the exercises provided to reinforce your understanding. Functions are a critical part of JavaScript, and mastering them will set a strong foundation for more advanced topics.