the Master Course

{CODENATION}

Intermediate JavaScript

Scope and Higher Order Functions

{CODENATION}

{CODENATION}

Learning Objectives

To explore JavaScript variable scope

To be familiar with higher order functions

To use higher order functions

We've all experienced it...
Sometimes we have access to a
variable we've created and
sometimes we don't ..**why**?

Let's look at the three types of
variable scope

- **Global** Scope
- **Function** Scope
- **Block** Scope

Global Scope...


```
let age = 21;

console.log(`My age: ${age}`);


const callAge = () => {
    console.log(`My age: ${age}`);

};

callAge();
```

Variables declared **Globally** (outside any function) have **Global Scope**.

Global variables can be accessed from anywhere in a JavaScript program.

Function Scope...


```
let age = '21';

console.log(`My age: ${age}`);

const callAge = () => {
    let name = 'Karen';
    console.log(`My age is: ${age}`);
    console.log(`My name is: ${name}`);
};

callAge();

console.log(`Global Scope with local variable: ${name}`);
//ReferenceError: Name is not defined
```

Javascript has **function** scope, meaning each function creates a new scope.

Variables declared inside a function are not accessible from outside the function.

{CN}®

Block Scope...


```
function startLet() {
    for (let i = 0; i < 5; i++) {
        console.log(i); //Output: 0,1,2,3,4
    }
    console.log(i); //Error i is not defined
}
function startVar() {
    for (var i = 0; i < 5; i++) {
        console.log(i); //Output: 0,1,2,3,4
    }
    console.log(i); //Output: 5
}
console.log('Running with let:');
startLet();
console.log('Running with var:');
startVar();
```

Before ES6 (2015) Javascript only had **global** scope and **function** scope.

ES6 introduced the **Let** and **Const** keywords. These two keywords provide **block scope** in Javascript.

Variables declared inside a block cannot be accessed outside a block.

Let's compare and contrast the old keyword **Var** with **Let** in a **For Loop**.

{CN}®

Block Scope...


```
function startLet() {
    for (let i = 0; i < 5; i++) {
        if (true) {
            let colour = 'red';
            console.log(i, colour); //Output: 0,1,2,3,4 with red
        }
    }
    console.log(i, colour); //ReferenceError: i is not defined
}
function startVar() {
    for (var i = 0; i < 5; i++) {
        if (true) {
            var colour = 'blue';
            console.log(i, colour); //Output: 0,1,2,3,4 with blue
        }
    }
    console.log(i, colour); //Output: 5, blue
}
console.log('Running with let:');
startLet();
console.log('Running with var:');
startVar();
```

Now let's take a look at **Var** and **Let** in an **if/else statement**.

{CN}®

Scoping works outwards...

...JS looks for variables in the current **scope.** If it doesn't find it, it will then look outward to the previous scope until the **global scope** if needed.

This is called scope chain...

Scope Chain...


```
let globalVar = 'globalVar';
console.log(`Global Scope: ${globalVar}`);
const outerFun = () => {
    let outerVar = 'outerVar';
    console.log(`Outer function: ${globalVar}`);
    console.log(`Outer function: ${outerVar}`);
    console.log(`Outer function: ${innerVar}`); //ReferenceError: innerVar is not defined
    const innerFun = () => {
        let innerVar = 'innerVar';

        console.log(`Inner function: ${globalVar}`);
        console.log(`Inner function: ${outerVar}`);
        console.log(`Inner function: ${innerVar}`);
    };
    innerFun();
};
outerFun();
innerFun(); //ReferenceError: innerFun is not defined (as it's inside outerFun())
```

Higher Order Functions

Functions which accept a function as a parameter.

OR

Functions which return a function.

Higher Order Functions...

Example One

```
const whichGreeting = (timeOfDay) => {
    console.log(`Good ${timeOfDay}`);
};
const greet = (time, fn) => {
    if (time < 1200) {
        fn('Morning');
    } else if (time >= 1200 && time < 1800) {
        fn('Afternoon');
    } else {
        fn('Evening');
    }
};
greet(1400, whichGreeting);
```

Higher Order Functions...

Example Two

```
const add = () => {
    return 2 + 3;
};
add(); //logs 5
add; // logs the whole function
```

Higher Order Functions...

Example Three

```
const add = (num1) => {
    return (num2) => {
        return num1 + num2;
    };
};
console.log(add(2)); //returns the function in the main function
```

Higher Order Functions...

Example Four

```
const add = (num1) => {
    return (num2) => {
        return num1 + num2;
    };
};
console.log(add(2)(1)); //output: 3
```

{CODENATION}

Learning Objectives

To explore JavaScript variable scope

To be familiar with higher order functions

To use higher order functions

Activity 1...

Write a simple **function** that logs **'Hello Codenation"** to the **console**.

Then write a **higher-order function** which will run the simple function five times, even though you only call it once.

Hint: Pass the simple function as a parameter, which will involve a For loop.

Activity 2...

The array method **.map** is an example of a higher-order function.

Declare a variable with five numbers, then use **.map** to iterate through the array and multiply each array item by 3.

Activity 3...

Test this **function** to ensure it works by passing a number to the **doMaths** function.

Then passing a number and one of the four maths functions to the returned function.

```
const add = (a, b) => {
    return a + b;
};
const subtract = (a, b) => {
    return a - b;
};
const multiply = (a, b) => {
    return a * b;
};
const divide = (a, b) => {
    return a / b;
};
const doMaths = (num1) => {
    return (num2, fn) => {
        return fn(num1, num2);
    };
};
```