Scope, Hoisting, and Recursive Functions

Software Development Bootcamp

Scope and hoisting,recursion, and writing recursive functions

Topic

Scope And Hoisting

What Is Scope?

Scope in JavaScript refers to the current context of code, determining the accessibility of variables to JavaScript.

- **Global Scope:** Variables declared outside of any function or block
- **Local Scope:** Variables declared inside a function or block
  - **Block Scope:** Created with `let` and `const` keywords

- `globalVar` is accessible everywhere
- `localVar` is only accessible within `exampleFunction`

```
let globalVar = "I'm global";

function exampleFunction() {
 let localVar = "I'm local";
 console.log(globalVar); // Accessible
 console.log(localVar); // Accessible
}

console.log(globalVar); // Accessible
console.log(localVar); //
ReferenceError: localVar is not defined
```

What Is Hoisting?

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope before code execution.

- Only declarations are hoisted, not initializations
- Function declarations are hoisted completely
- `let` and `const` declarations are hoisted but **not** initialized

Hoisting Example

- `let` and `const` are hoisted but not initialized
- The function declaration `sayHello` is fully hoisted

```
console.log(x);
// ReferenceError: Cannot access 'x' before
initialization
const x = 5;

console.log(y);
// ReferenceError: Cannot access 'y' before
initialization
let y = 10;

sayHello();  // "Hello!"

function sayHello() {
   console.log("Hello!");
}
```

Why Understanding Scope and Hoisting Matters

1. **Avoid Bugs:** Proper understanding prevents unexpected behavior
2. **Code Organization:** Helps structure code for better readability and maintainability
3. **Debugging:** Makes it easier to track down issues related to variable access and initialization

Topic

Recursion

What Is Recursion?

- Recursion: When a function calls itself to solve a problem.
  - In programming recursion can be useful, but we need to make sure it stops at some point, or it will go on forever!

Why Use Recursion?

Recursion is a powerful programming technique with several benefits:

1. **Simplicity:** Recursive solutions can be more elegant and easier to understand for certain problems.
2. **Divide and Conquer:** Complex problems can be broken down into simpler sub-problems.
3. **Natural Fit:** Some problems are inherently recursive (e.g., tree structures, fractals).
4. **Memory Efficiency:** For some algorithms, recursive solutions can be more memory-efficient.
5. **Code Reusability:** Recursive functions often lead to more modular and reusable code.

Infinite Recursion

- Calling `go()` will continue indefinitely
- Stop the function by pressing CTRL + C
- We get the `RangeError` which means `go` has called itself too many times.

```
function go() {
    console.log('Go!')
    go() //Calling the go function again
}
go()
// Error: RangeError: Maximum call stack size exceeded
```

Recursion Requires Termination

- Recursion must eventually stop to be useful
- Use a guard clause, also called a base case or terminator

```
function countdown(seconds) {
    // base case
    if (seconds === 0) {
        console.log('Blastoff!')
    }
}
```

Countdown Example

- Checks if the number of seconds is 0.
  - If seconds is 0 it prints “Blastoff!”
- If seconds is not 0 it prints the number followed by ‘...’
- Then the function calls itself (this is the recursive aspect!) with the seconds - 1

```
function countdown(seconds) {
    if (seconds === 0) {
        console.log('Blastoff!')
    } else {
        console.log(seconds + '...')
        countdown(seconds - 1)
    }
}
countdown(10)
```

Recursion as Reduction

- Starts with a large problem and reduces it step by step to the base case.
- Metaphor: Like a Russian doll, each step nests within the next.
- Known solution at the base allows building up the solution.