Q&A Session

Presented by: @TA_Yaya

JavaScript Orientation

Definition of Data Types

Data types define the type of data that a variable can hold or represent.
JavaScript is a loosely typed or dynamic language, meaning that variables are not directly associated with any particular data type, and the same variable can hold different types of data over time

Primitive Types

Primitive types are immutable and represent basic data types.
These are the building blocks for more complex types

- Number
- Boolean
- String
- Undefined
- Null

Composite Types

These types are mutable and used for more complex data structures.

- Object
- Array

Presented by: @TA_Yaya

Defintion of Variable

A Variable is a container used to store data values. Variables allow you to assign, store, and manipulate data throughout your program.

How Can we declare a variable in JS?

First method (const): To declare a variable, start with the keyword "**const**" followed by the variable name, which must be alphanumeric (it cannot start with numbers or special characters).
By convention, for compound words, use **camelCase**.
Variable assignment: Assign a value of any data type in JavaScript using the equal sign (=), followed by the data type such as **Number, String, Boolean, Array, Object**, etc.

Second method (let): Follow the same procedure as for "const."

Understanding the difference between "const" and "let":
Reassigning a variable:
Note: When you create a variable with the keyword "**const**" it means that this variable cannot be reassigned. So, when you want to assign a new value or reassign the variable, use the keyword "**let**"
Third method (var): This method involves declaring a variable with the keyword "**var**" It is not commonly used today, as it's an older method used in the past.
It serves the same function as "**let**" and remember, a variable without a keyword is treated as if it were declared with "**var**"

Function

A Function is a reusable block of code designed to perform a specific task.
Functions allow you to encapsulate logic, making your code modular and easier to maintain.

Key Features of Functions in JavaScript

1.Function Declaration

- A named function can be declared using the function keyword.
- It can be called anywhere in the code after declaration.

```
Tabnine: Edit | Test | Explain | Document | Ask
function greet(name) {
    console.log("Hello, " + name);
  }

  greet("John");  //? Output: Hello, John
```

2. Function Expression

- A function can also be assigned to a variable.
- This is known as a function expression. Function expressions are not hoisted, meaning they cannot be called before they are defined.

```

const sayHello = function(name) {
    return "Hello, " + name;
  };

  console.log(sayHello("Jane"));  //? Output: Hello, Jane

```

Presented by: @TA_Yaya

>> NEFEL EDUCATION

Functions

Arrow Functions

Introduced in ES6, arrow functions provide a concise way to write functions. They do not have their own this context.

```
const add = (a, b) => a + b;

console.log(add(5, 3));  //? Output: 8
```

Parameters and Arguments

- Functions can take parameters, which are placeholders for the values (arguments) you pass when calling the function.

```
Tabnine: Edit | Test | Explain | Document | Ask
function multiply(x, y) {
    return x * y;
}

console.log(multiply(2, 3));  //? Output: 6
```

Presented by: @TA_Yaya

>> NEFEL EDUCATION

Functions

Return Statement:

The return statement is used to send a value back from a function. If no return is specified, the function returns undefined by default.

```
Tabnine: Edit | Test | Explain | Document | Ask
function square(num) {
    return num * num;
}

let result = square(4);
console.log(result);  // Output: 16
```

Anonymous Functions:

- Functions can be anonymous, meaning they don't have a name. These are commonly used as arguments in other functions.

```
Tabnine: Edit | Test | Explain | Document | Ask
setTimeout(function() {
    console.log("This is an anonymous function");
}, 1000);

//? Output: This is an anonymous function
```

Scope Concepts

In JavaScript, Scope refers to the accessibility or visibility of variables and functions in different parts of a program.

It determines where variables and functions can be accessed or used.

Global Scope

A variable declared outside any function or block has global scope. It can be accessed from anywhere in the code, including inside functions and blocks.

- Global Variables are available throughout the entire program.

```
let globalVar = "I am global";

Tabnine: Edit | Test | Explain | Document | Ask
function test() {
  console.log(globalVar);  //* Accessible inside the function
}

test(); //? Output: I am global
console.log(globalVar);  //* Accessible outside functions as well
```

Local Scope:

Variables declared within a function are in local scope. These variables can only be accessed inside that function and are not available outside of it.

- Local Variables are confined to the function they are declared in

```
Tabnine: Edit | Test | Explain | Document | Ask
function greet() {
    let name = "Alice";
    console.log(name);  //* Accessible within the function
  }

  greet();  //? Output: "Alice"
  console.log(name);  //* Error: name is not defined (outside the function)
```

Presented by: @TA_Yaya

>> NEFEL EDUCATION

Scope Concepts

Block Scope

Variables declared with let or const inside a block (code wrapped in curly braces {}) are in block scope. This means they are only accessible inside that specific block.

- Block-scoped variables (with let and const) are limited to the block where they are declared.

```
    var functionVar = "I'm a function-scoped variable";
    if (true) {
        var insideBlock = "Still inside the function";
    }
    console.log(insideBlock);  //* Works, because var is function-scoped
}

myFunction();
console.log(functionVar);  //? Output: Error: functionVar is not defined (outside the function)
```

Function Scope:

- Variables declared using var inside a function are function-scoped, meaning they are available throughout the entire function, but not outside it. Unlike let and const, var ignores block scope and can be accessed outside blocks within the same function.

```
Tabnine: Edit | Test | Explain | Document | Ask
function myFunction() {
    var functionVar = "I'm a function-scoped variable";
    if (true) {
        var insideBlock = "Still inside the function";
    }
    console.log(insideBlock);  //* Works, because var is function-scoped
}

myFunction();
console.log(functionVar);  //? Output: Error: functionVar is not defined (outside the function)
```

Hoisting in JS

Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their scope (the global scope or the function scope) before the code is executed.

This means that you can use variables and functions before they are actually declared in the code.

Here's how hoisting works:

1. **Function Hoisting**:Function declarations are fully hoisted. This means you can call a function before it is declared in your code.

```
1
2
3   greet(); //* This works because of hoisting
4
    Tabnine: Edit | Test | Explain | Document | Ask
5   function greet() {
6     console.log("Hello, MERN Ninjas 🥷 !");
7   }
8
9   //? Output: Hello, MERN Ninjas 🥷 !
```

2. **Variable Hoisting:**

With **variable hoisting**, only the declaration is hoisted, not the initialization.
Variables declared with var are hoisted and initialized with undefined. However, variables declared with let and const are hoisted but not initialized, leading to a Temporal Dead Zone until the variable's declaration is encountered.

Hoisting in JS

```
//* Example with (var): var

console.log(x); //? Output: undefined
var x = 5;
console.log(x); //? Output: 5
```

```
//* Example with (let and const):

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

Key Takeaways:

- Function declarations are fully hoisted.
- var declarations are hoisted but initialized as undefined.
- let and const declarations are hoisted but not initialized, leading to the Temporal Dead Zone before they are declared in the code.

Destructuring in JS

Destructuring in JavaScript is a convenient way to extract values from arrays or properties from objects into distinct variables.
This syntax allows you to write cleaner and more readable code when dealing with data structures.

1.Array Destructuring: Array destructuring allows you to unpack values from arrays into variables.

```
//! I) Array Destructuring:
const numbers = [1, 2, 3];

//* a)Destructuring assignment
const [first, second, third] = numbers;

console.log(first);  //? Output: 1
console.log(second); //? Output: 2
console.log(third);  //? Output: 3
```

2.Object Destructuring: Object destructuring allows you to unpack properties from objects into variables.

```
const person = {
    name: 'John',
    age: 30
  };

  //* Destructuring assignment
  const { name, age } = person;

  console.log(name); //? Output: John
  console.log(age);  //? Output: 30
```


THANK YOU!