Hema Coding School

YouTube Link: https://www.youtube.com/@HemaCodingSchool

Advanced JavaScript

Let Const

1. Scope

- **var**:
  - Function-scoped: The scope is confined to the function where it is declared.
  - If declared outside a function, it becomes globally scoped.
- **let** and **const**:
  - Block-scoped: The scope is limited to the block (enclosed by {}) where they are declared.

Function Scope

Function scope means that a variable declared within a function is only accessible within that function. Variables declared with var are function-scoped.

```
function Scope() {
   var a =10;
   if(true){
      console.log(a);
      var b = 20;
      console.log(b)
   }
   console.log(b)
}
Scope()
console.log(a);
// console.log(b);
```

Block Scope

Block scope means that a variable declared within a block {} is only accessible within that block. Variables declared with let and const are block-scoped.

```
function Scope() {
   if(true){
      let a =10;
      console.log(a);
      let b = 20;
      console.log(b)
   }
   console.log(b) // error
}
```

```
Scope()
console.log(a); //error
console.log(b);// error
```

Hoisting

- **var**:
  - Variables declared with var are hoisted to the top of their scope and initialized with undefined.
- **let** and **const**:
  - Variables declared with let and const are hoisted to the top of their block but are not initialized. Accessing them before their declaration results in a ReferenceError.

```
console.log(a)
var a = 10;

console.log(b);
let b = 15;

console.log(c)
const c = 12;
```

Function Hoisting

Function declarations are completely hoisted, meaning that both the declaration and the definition are moved to the top of their containing scope. This allows you to call a function before it is declared in the code.

```
hosting();
function hosting(){
   console.log('function is hosted')
}
```

Temporal Dead Zone

The TDZ starts from the beginning of the block where the variable is declared and ends when the variable is initialized.

```
{
   console.log(a)
   let a = 10;
   console.log(a)
}
```

Reassignment

- **var** and **let**:

- Variables declared with var and let can be reassigned.

- **const**:
  - Variables declared with const cannot be reassigned. However, if the variable is an object or array, the properties or elements can be modified.

```
var a = 10;
a= 20;
console.log(a);

let b = 10;
b = 30;
console.log(b);

const c = 15;
c = 63;
```

Redeclaration

- **var**:
  - Can be redeclared within the same scope without errors.
- **let** and **const**:
  - Cannot be redeclared within the same scope. Redeclaring them results in a SyntaxError..

```
var a = 10;
var a= 20;
console.log(a);

let b = 10;
let b = 30;
console.log(b);

const c = 15;
const c = 63;
```

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted (initialized as undefined)	Hoisted (not initialized)	Hoisted (not initialized)
Reassignment	Allowed	Allowed	Not allowed
Temporal Dead Zone	No	Yes	Yes
Redeclaration	Allowed	Not allowed	Not allowed

Hema Coding School

YouTube Link: https://www.youtube.com/@HemaCodingSchool

Template Literals:

Introduction

Template literals are a feature introduced in ES6 (ECMAScript 2015) that provide an easier and more readable way to work with strings. They allow for embedding expressions, creating multi-line strings, and adding string interpolation, which is more convenient than the traditional string concatenation methods.

Syntax

Template literals are enclosed by backticks (`) instead of single quotes (') or double quotes ("). Expressions can be embedded using the ${expression} syntax.

```
let greet1 = `Good evening to everyone. How is the day?`;
console.log(greet1);
```

Multi-line Strings:

- Supports multi-line strings without the need for escape characters.

```
let greet1 = `Good evening to
 everyone.
 How is the
  day?`;
console.log(greet1);
```

String Interpolation:

- Allows embedding expressions directly within the string.

```
let age = 22;
let name = "Hema";
let message = "My name is " + name + " and " + " age is " + age;
let message1 = `My name is ${name} and age is ${age}`;
console.log(message);
console.log(message1);
```

Tagged Template :

Tagged templates allow you to parse template literals with a function. This feature is more advanced and is used for custom parsing or formatting of template literals.

```
function taggedTemp(name, value){
    console.log(name)
console.log(value)
}
```

```
let name = "Mahesh";
taggedTemp`my name is ${name}`
```

Symbol Data Type in JavaScript

The Symbol is a primitive data type introduced in ECMAScript 6 (ES6). Symbols are unique and immutable values that can be used as keys for object properties. They provide a way to create private or unique identifiers for object properties.

Uniqueness: Every Symbol value is unique. Even if two symbols are created with the same description, they are different from each other.

Immutability: Once a Symbol is created, its value cannot be changed.

```
let name = Symbol();
let age = Symbol("Hello");

console.log(name)
console.log(age)
```

Non-Enumerable: Symbols are not included in for-in loops or Object.keys() methods, which makes them useful for defining non-enumerable properties.

```
let email = Symbol()
let employee = {}
employee.name = "Mahesh";
employee.age = 23;
// employee["email"] = "Mahesh@gmail.com";
employee[email] = "Mahesh@gmail.com";

console.log(employee)
console.log(employee[email])
console.log(Object.keys(employee))

for(let key in employee){
   console.log(key)
}
```

Set Data Type:

The Set data type in JavaScript is a collection of values where each value must be unique. Unlike arrays, sets do not allow duplicate values. This data structure was introduced in ECMAScript 2015 (ES6) and is useful for storing and managing a collection of unique elements.

Key Features of Set

1. **Unique Values**: A Set ensures that all its elements are unique. If you try to add a value that already exists in the set, it will be ignored.

2. **Any Type of Values**: Values in a Set can be of any type, including primitives and objects.
3. **Order of Elements**: Sets are ordered; they iterate over their elements in the insertion order.
4. **Size Property**: A Set has a size property that returns the number of elements in the set.
5. **Iterable**: A Set is iterable, allowing you to loop over its elements using methods like forEach or the for...of loop.

Methods of Set

- add(value): Adds a new element with the given value to the set.
- delete(value): Removes the specified element from the set.
- has(value): Returns true if the set contains the specified value, otherwise false.
- clear(): Removes all elements from the set.
- values(): Returns an iterator for the values in the set.

```
let set = new Set();
set.add(1)
set.add(2)
set.add(3)

set.delete(1)
console.log(set.clear())
console.log(set.has(1))
console.log(set)
```

```
var obj = new Set([1,2,3,4,4,5,2])
// console.log(obj)
for(let key of obj){
   console.log(key)
}
```

Map in Data Type:

The Map data type in JavaScript is a collection of key-value pairs where both the keys and values can be of any type. It is part of ECMAScript 2015 (ES6) and provides several benefits over the traditional object when it comes to managing key-value pairs.

Key Features of Map

1. **Any Type of Keys**: Unlike objects, where keys are typically strings or symbols, a Map allows keys to be of any type, including objects, functions, and primitive data types.
2. **Ordered Entries**: Entries in a Map are ordered. When you iterate over a Map, the entries are returned in the order in which they were added.
3. **Size Property**: A Map has a size property that returns the number of key-value pairs in the map.

4. **Iterable**: A Map is iterable, meaning you can loop over its entries, keys, or values using methods like forEach or the for...of loop.
5. **Methods for Manipulation**:
    - set(key, value): Adds or updates an entry in the map.
    - get(key): Retrieves the value associated with a key.
    - has(key): Checks if a key exists in the map.
    - delete(key): Removes an entry by key.
    - clear(): Removes all entries from the map.
    - entries(): Returns an iterator for the [key, value] pairs.
    - keys(): Returns an iterator for the keys.
    - values(): Returns an iterator for the values.

```
let map = new Map();

map.set("name", "Hema");
map.set(1, 'one');
map.set(true, 'boolean');

console.log(map.get("name"))
// console.log(map.has("name"))

map.delete('name');
console.log(map.has("name"))

map.clear()
console.log(map)
```

Interview Questions:

What is the difference between let and const?

- **Answer**: let allows you to declare variables that can be reassigned, while const declares variables that cannot be reassigned after their initial assignment. Both let and const are block-scoped, unlike var, which is function-scoped.

What is let and how does it differ from var?

- let declares a block-scoped local variable, optionally initializing it to a value. Unlike var, let variables are not hoisted to the top of their enclosing scope.

What is const and how does it differ from let?

- const declares a block-scoped variable that cannot be reassigned. However, if the variable is an object or an array, the contents can be changed.

Explain the concept of block scope with an example.

- **Answer**: Block scope means that a variable is only accessible within the block (denoted by {}) where it is declared. For example:

```
if (true) {
  let x = 10;
  console.log(x); // 10
}
console.log(x); // ReferenceError: x is not defined
```

What is the Temporal Dead Zone (TDZ)?

- **Answer**: The TDZ is the period between the start of a block and the point where a variable declared with let or const is initialized. Accessing the variable during this period results in a ReferenceError.

Can you reassign a const variable?

- **Answer**: No, a const variable cannot be reassigned after its initial assignment. However, if the const variable is an object or array, you can still modify its properties or elements.

What is a Symbol in JavaScript?

- **Answer**: A Symbol is a primitive data type introduced in ES6 that represents a unique and immutable value. Symbols are often used as keys for object properties to ensure uniqueness.

How do you create a Symbol?

- **Answer**: You create a Symbol using the Symbol() function, optionally passing a description.

What is a Map in JavaScript?

- **Answer**: A Map is a collection of key-value pairs where keys can be of any data type. Map objects remember the original insertion order of the keys.

How do you create a Map and add elements to it?

- **Answer**: You create a Map using the new Map() constructor and add elements using the set method.

```
const map = new Map();
map.set('key1', 'value1');
map.set('key2', 'value2');
```

What is a Map in JavaScript and how does it differ from a regular object?

- A Map is an ordered collection of key-value pairs where keys can be of any type. Unlike objects, Map keys are not limited to strings and symbols.

```
const map = new Map();
map.set("key1", "value1");
map.set(2, "value2");
console.log(map.get("key1")); // Output: "value1"
console.log(map.get(2)); // Output: "value2"
```

What is a Set in JavaScript?

- **Answer**: A Set is a collection of unique values. A value in a Set can only occur once.

What is a Set in JavaScript and how does it differ from an array?

- A Set is a collection of unique values. Unlike arrays, Set automatically removes duplicate values.

```
const set = new Set([1, 2, 2, 3, 4, 4]);
console.log(set); // Output: Set { 1, 2, 3, 4 }
```

How do you create a Set and add elements to it?

- **Answer**: You create a Set using the new Set() constructor and add elements using the add method.

```
const set = new Set();
set.add(1);
set.add(2);
set.add(2); // Duplicate, will not be added
```

How do you check if a Set contains a specific value?

```
const set = new Set();
set.add(1);
console.log(set.has(1)); // true
```

Explain how to iterate over the values in a Set.

- **Answer**: You can use the for...of loop to iterate over a Set

```
const set = new Set([1, 2, 3]);
for (const value of set) {
    console.log(value);
}
```