**Scopes in JavaScript**

1. Scope in JavaScript determines where variables and functions are accessible in your code.
2. JavaScript has different type of scope –
   1. **Global Scope -**
      1. Variables declared outside the function and block scope are called global scope.
      2. The variable declare in global scope, then we can access that variable anywhere in the project or code

**Example:**

let globalVar = "I'm global!";

function test() {

console.log(globalVar);

}

console.log(globalVar);

* 1. **Block Scope –**
     + 1. A block scope is anything inside the ***curly brackets { }***.
       2. The variable declared inside the block scope is only accessible in that scope.
       3. We can not access the variable outside the scope
       4. let and const support block scope

**Example:**

{

let a = 10;

const b = 20;

var c = 30;

}

console.log(c); // ✅ 30 (because var ignores block scope)

console.log(a); // ❌ Error

console.log(b); // ❌ Error

* 1. **Function Scope –**
     + 1. Variables declared inside the function are called function scope.

Ex. function fun\_nm() {

Fun\_body

  }

* + - 1. The variable declared inside the function scope is only accessible in that scope.
      2. We can not access the variable outside the scope
      3. var keyword support function scope

**Example:**

function myFunction() {

let localVar = "I'm local!";

console.log(localVar);

}

myFunction();

console.log(localVar);

* 1. **Lexical Scope –**
     + 1. Lexical scope means that a function can access variables defined outside the outer scope.

**Example:**

function outer() {

let outerVar = "I'm outer";

function inner() {

console.log(outerVar); // ✅ Accessible

}

inner();}

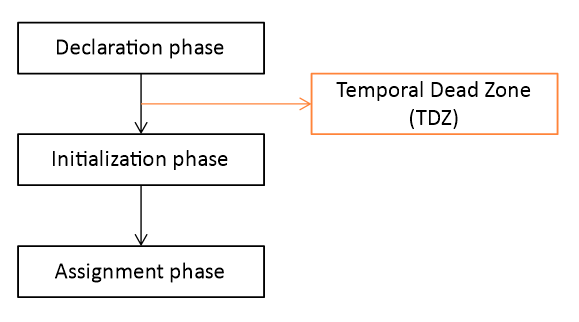
outer();

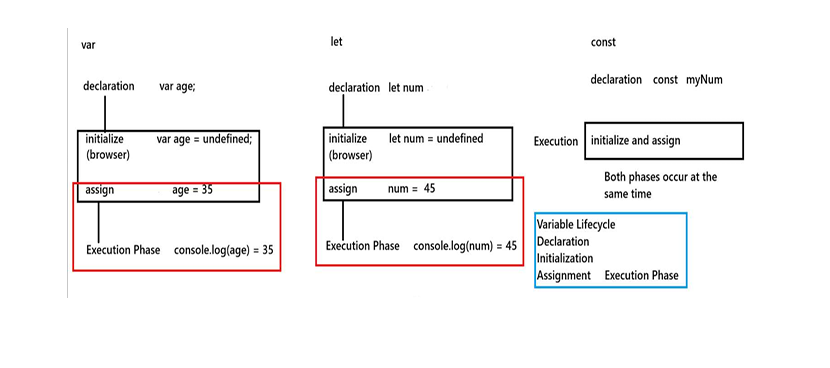
**Variables Hoisting**

1. Variable Hoisting in JavaScript is a behavior where variable declarations are moved (or “hoisted”) to the top of their containing scope during the compilation phase - before code is executed .
   1. var -   undefined (in compilate var is execute)
2. Variables declared with let and const are also hoisted but they are not initialized .
3. They stay in temporal dead zone (TDZ) from the start of the block until declaration is encountered.
4. While accessing the variable before declaration it throw referenceError
   1. let, const - referenceError (this variable first declared and then initialized)

**Temporal Dead Zone (TDZ) in JavaScript -**

* The Temporal Dead Zone (TDZ) is a critical behavior in JavaScript that affects variables declared with let and const.
* It represents the period between when a variable is hoisted (allocated in memory) and when it is initialized (assigned a value).
* During this time, accessing the variable throws a ReferenceError.



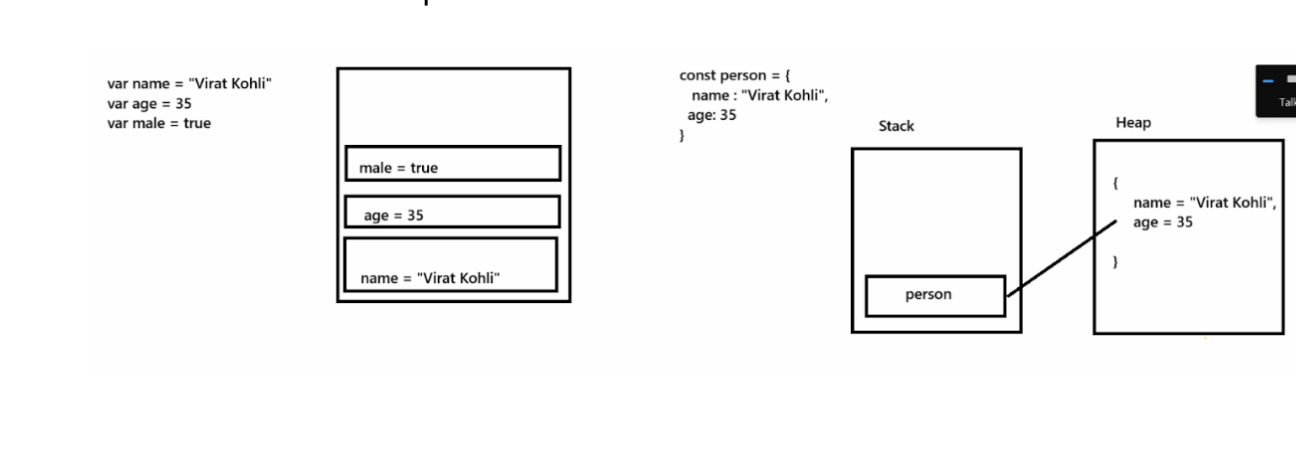


Comparison of Variable Keywords -

|  |  |  |  |
| --- | --- | --- | --- |
| **Aspect** | **var** | **let** | **const** |
| **Scope** | Global-scope and  Functional-scope | Block-scope and  Functional-scope | Block-scope and  Functional-scope |
| **Declaration** | Allows redeclaration | Redeclaration is not allowed | Redeclaration is not allowed |
| **Initialization** | Initialization is optional | Initialization is optional | Initialization is mandatory |
| **Reassignment** | Values can be reassigned | Values can be reassigned | Values cannot be reassigned |
| **Hoisting** | Hoisting is possible | Hoisting is not possible because of (TDZ) | Hoisting is not possible because of (TDZ) |
| **Best practices** | Not recommended due to confusion with scoping and hoisting | Best for variables that may change their value | Best for variables that remain constant |

**How Variable Allocate Memory**

* When we write a program in any language, we declare a lot of variables, arrays, objects in various sections of the program such as the main function, global section, classes, functions, methods, etc.
* These variables need some space in the Random Access Memory (RAM).
* So, where are they stored? These variables are stored in two types of memory, one called **Stack and another called Heap.**
* Both Stack and Heap are portions of main memory (RAM).
* The operating system allocates a stack and a heap for all programs.
* When any program starts, it is provided with a private working memory having a stack, heap
* JavaScript uses two types of memory to store variables: the **stack** and the **heap**. The type of memory used depends on the data type of the variable.
* **Stack Memory:**
  + The stack is a fast, rigid, last-in, first-out (LIFO) memory region. Think of it like a stack of plates—you can only add a plate to the top, and you can only remove the top plate. This makes memory allocation and deallocation very fast.
  + **How it works:**
    - The stack is used for **static data** with a fixed size, such as primitive values (numbers, booleans, null, undefined) and references to objects and arrays.
    - When you declare a variable with a primitive value, the value itself is stored directly on the stack.
    - When a function is called, a "stack frame" is pushed onto the stack. This frame contains the function's local variables. When the function returns, its frame is popped off, and the memory is automatically freed.
* **Heap Memory:**
  + The heap is a flexible, unstructured memory region. It's used for **non-primitives** (dynamic-size data like objects and arrays).
  + When you create an object or an array, the actual data is stored in the heap.
  + The variable you assigned it to, however, is a **reference** that is stored on the stack and points to the data's location in the heap.



**Datatypes in JavaScript**

1. Datatypes are a kind of data which we are stored in variable.
2. JavaScript is a dynamic typed language that means we are not mentioning a datatype before variable declaration.

Java/ c++/c - int age = 25,

Js - age = 15

1. We use ***typeof*** operator to print the datatype of variable

Datatype categorized in two part -

* 1. **Primitive Datatypes**
     1. **Primitive datatypes variables are stored in stack memory.**
     2. **Primitive datatypes are immutable that means we cannot modify the value of a variable.**
     3. **We can store single values and access fast**

**Primitive Datatype**

* + - 1. **Number -**
         1. Represent both integer and floating number

Ex. 25, 95.45, 7719090215

* + - * 1. Usage - Calculations, prices, scores, percentage etc
      1. **String**
         1. Represent the sequence of character enclosed in single (‘ ’) , double (“ ”) quotes and backticks (` `)
         2. Usage - fullName, address, email, pancard etc
      2. **Boolean**
         1. Represent logical true or false
         2. Usage - For condition, toggle value etc
      3. **Undefined**
         1. A variable is declared but not assign any value
         2. Usage - To check if variable was assigned
      4. **Null**
         1. Represent an Intentional absence of value
         2. Usage - when a variables should hold nothing intentionally
         3. null return Object as a value using typeof operator but actually null is not an object
         4. This mistake become impossible to fix
      5. **BigInt**
         1. Use for Large integer
         2. Usage - for financial data, cryptography, large dataset
      6. **Symbol**

It represent the unique value often used in object key

Usage - For creating unique identifiers in object especially in libraries or framework

* 1. **Non-primitive Datatypes**
     + 1. **Non-primitive variables are stored in heap memory.**
       2. **Non-primitive datatypes are mutable, which means we can modify the value of a variable.**

**Non-primitive datatypes -**

**1.Object**

* Collection of **key-value pairs**.
* Can hold different types of data.

**Example:**

let person = {

name: "John",

age: 25

};

console.log(person.name); // "John"

**2. Array**

* Special type of object.
* Stores values in an ordered list.
* Array is heterogeneous type of data.

let colors = ["red", "green", "blue"];

console.log(colors[1]); // "green"

1. **Function**

* Functions in JS are also **objects** (they can have properties).

function greet() {

return "Hello!";

}

console.log(greet()); // "Hello!"

**4. Date**

* Built-in object to handle dates and times.

let today = new Date();

console.log(today.toDateString()); // e.g. "Fri Sep 19 2025"

**5.Math**

* Math is a **built-in object** in JavaScript.
* It has **properties** (constants) and **methods** (functions).
* Used for mathematical calculations.

Rounding methods

console.log(Math.round(4.6)); // 5 (nearest integer)

console.log(Math.floor(4.9)); // 4 (always down)

console.log(Math.ceil(4.1)); // 5 (always up)

console.log(Math.trunc(4.9)); // 4 (remove decimals)