Lecture 02: Data Type

Variable in Javascript

const

const declares a block-scoped variable with a constant reference.

1. **Scope: Block Scope**. A const variable is only accessible within the block ({ ... }) in which it is defined.
2. **Reassignment: Not allowed**. A const variable cannot be reassigned a new value after its initial assignment. This will throw a TypeError.
3. **Initialization: Mandatory**. A const variable must be initialized at the time of its declaration. Failure to do so results in a SyntaxError.
4. **Hoisting:** const variables are hoisted to the top of their block but are not initialized. They are in a "Temporal Dead Zone" (TDZ) from the start of the block until the declaration is encountered. Accessing a const variable before its declaration results in a ReferenceError.
5. **Mutability:** const only ensures the variable's reference is immutable, not the value it points to. If a const variable holds an object or an array, the properties or elements of that object/array can be modified.

Example:

```
// Block Scope if (true) { const PI = 3.14159; } // console.log(PI); // Re
ferenceError: PI is not defined // Reassignment const GREETING = "Hello";
// GREETING = "Hi"; // TypeError: Assignment to constant variable. // Muta
bility const CONFIG = { port: 8080 }; CONFIG.port = 3000; // This is allow
ed. CONFIG still points to the same object. // Temporal Dead Zone // conso
le.log(MY_CONST); // ReferenceError: Cannot access 'MY_CONST' before initi
alization const MY_CONST = 100;
```

let

let declares a block-scoped variable that can be reassigned.

1. **Scope: Block Scope**. A let variable is only accessible within the block ({ ... }) in which it is defined.
2. **Reassignment: Allowed**. The value of a let variable can be updated or reassigned within its scope.
3. **Initialization: Optional**. A let variable can be declared without being initialized. If not initialized, it defaults to the value undefined.
4. **Hoisting:** let variables are hoisted to the top of their block but are not initialized. They are in the Temporal Dead Zone (TDZ) until their declaration is encountered. Accessing a let variable before its declaration results in a ReferenceError.

Example:

```
// Block Scope for (let i = 0; i < 3; i++) { // i is only visible here }
// console.log(i); // ReferenceError: i is not defined // Reassignment let
counter = 0; counter = 1; // This is allowed. // Initialization let name;
// Allowed. 'name' is now undefined. name = "Alice"; // Temporal Dead Zone
// console.log(myLetVar); // ReferenceError: Cannot access 'myLetVar' befo
re initialization let myLetVar = "test";
```

---

var

var declares a function-scoped or globally-scoped variable that can be reassigned and redeclared. Its use is discouraged in modern JavaScript (ES6+).

1. **Scope: Function Scope**. A var variable is accessible throughout the entire function in which it is defined, regardless of block boundaries. If defined outside any function, it has global scope.
2. **Reassignment & Redeclaration: Allowed**. A var variable can be reassigned and even redeclared within the same scope without error.
3. **Initialization: Optional**. If not initialized, it defaults to the value undefined.
4. **Hoisting:** var declarations are hoisted to the top of their function scope and are initialized with the value undefined. This means they can be accessed before their declaration without a ReferenceError.

Example:

```
// Function Scope (not Block Scope) if (true) { var leak = "I am visible o
utside the if-block"; } console.log(leak); // Outputs: "I am visible outsi
de the if-block" // Hoisting Behavior console.log(myVar); // Outputs: unde
fined (no error) var myVar = "Hello"; console.log(myVar); // Outputs: "Hel
lo" // Redeclaration var x = 10; var x = 20; // Allowed. x is now 20.
```

Summary of Key Differences

Feature	var	let	const
Scope	Function	Block	Block
Reassignable	Yes	Yes	No
Redeclarable	Yes	No	No
Hoisted Value	undefined	Uninitialized (TDZ)	Uninitialized (TDZ)
Global Object	Attaches to window	Does not attach	Does not attach
Modern Practice	Avoid	Use for reassignable variables	Use as default

Data Types in Javascript

1. Primitive Types

Primitives are fundamental, immutable data types. "Immutable" means that their value cannot be changed once created. Operations that appear to modify a primitive actually create a new one. A variable holding a primitive stores the primitive's value directly.

There are seven primitive types:

string

- **Purpose:** Represents textual data.
- **Syntax:** A sequence of characters enclosed in single quotes ('...'), double quotes ("..."), or backticks (`...`).

```
let name = "Alice"; let greeting = 'Hello, World!'; let template = `User: ${name}`; // Template literals can embed expressions
```

number

- **Purpose:** Represents both integer and floating-point numbers.
- **Syntax:** A numeric literal. There is no distinction between int and float.
- **Special Values:** Includes Infinity, -Infinity, and NaN (Not a Number).

```
let integerValue = 100; let floatValue = 3.14; let notANumber = NaN; // Result of an invalid math operation like 0 / 0 let infinity = Infinity;
```

boolean

- **Purpose:** Represents a logical entity with two possible values.
- **Syntax:** The keywords true or false.

```
let isActive = true; let isComplete = false;
```

undefined

- **Purpose:** Represents the unintentional absence of a value. A variable that has been declared but not assigned a value is automatically undefined.

```
let user; console.log(user); // Outputs: undefined
```

null

- **Purpose:** Represents the intentional absence of any object value. It is a primitive value that is explicitly assigned by a developer to indicate "no value."

```
let data = null; // Intentionally set to have no value
```

- **null vs. undefined:** undefined is the default when nothing is assigned. null is an explicit assignment of "nothing."

bigint

- **Purpose:** Represents whole numbers larger than the maximum safe integer value that the number type can represent.
- **Syntax:** An integer literal followed by the n suffix.

code JavaScript

```
const veryLargeNumber = 9007199254740991n; // The 'n' makes it a BigInt co
nst anotherBigInt = BigInt(9007199254740992);
```

symbol

- **Purpose:** Represents a unique, anonymous identifier. Symbols are primarily used as unique property keys for objects to avoid naming collisions.
- **Syntax:** Created using the Symbol() factory function.

```
const id1 = Symbol('id'); const id2 = Symbol('id'); console.log(id1 === id
2); // Outputs: false (every symbol is unique)
```

---

2. The Object Type (Non-Primitive)

An object is a mutable collection of key-value pairs (or properties). Unlike primitives, variables assigned to an object do not store the object itself, but rather a **reference** (or a pointer) to the object's location in memory.

- **Object Literals:** The most common way to create an object. code JavaScript

```
let person = { firstName: "John", lastName: "Doe", age: 30 };
```

- **Arrays:** A specialized type of object used for ordered collections. code JavaScript

```
let numbers = [10, 20, 30, 40];
```

- **Functions:** In JavaScript, functions are also a special type of object. code JavaScript

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

- Other built-in objects include Date, RegExp, Map, Set, etc.

Key Difference: Value vs. Reference

This is the most critical distinction between primitive and object types.

Primitives are Passed/Assigned by Value

When you assign a primitive from one variable to another, the value is **copied**.

```
let a = 10; let b = a; // The value 10 is copied into b b = 20; // This on
ly changes b console.log(a); // Outputs: 10 (a is unaffected) console.log
(b); // Outputs: 20
```

Objects are Passed/Assigned by Reference

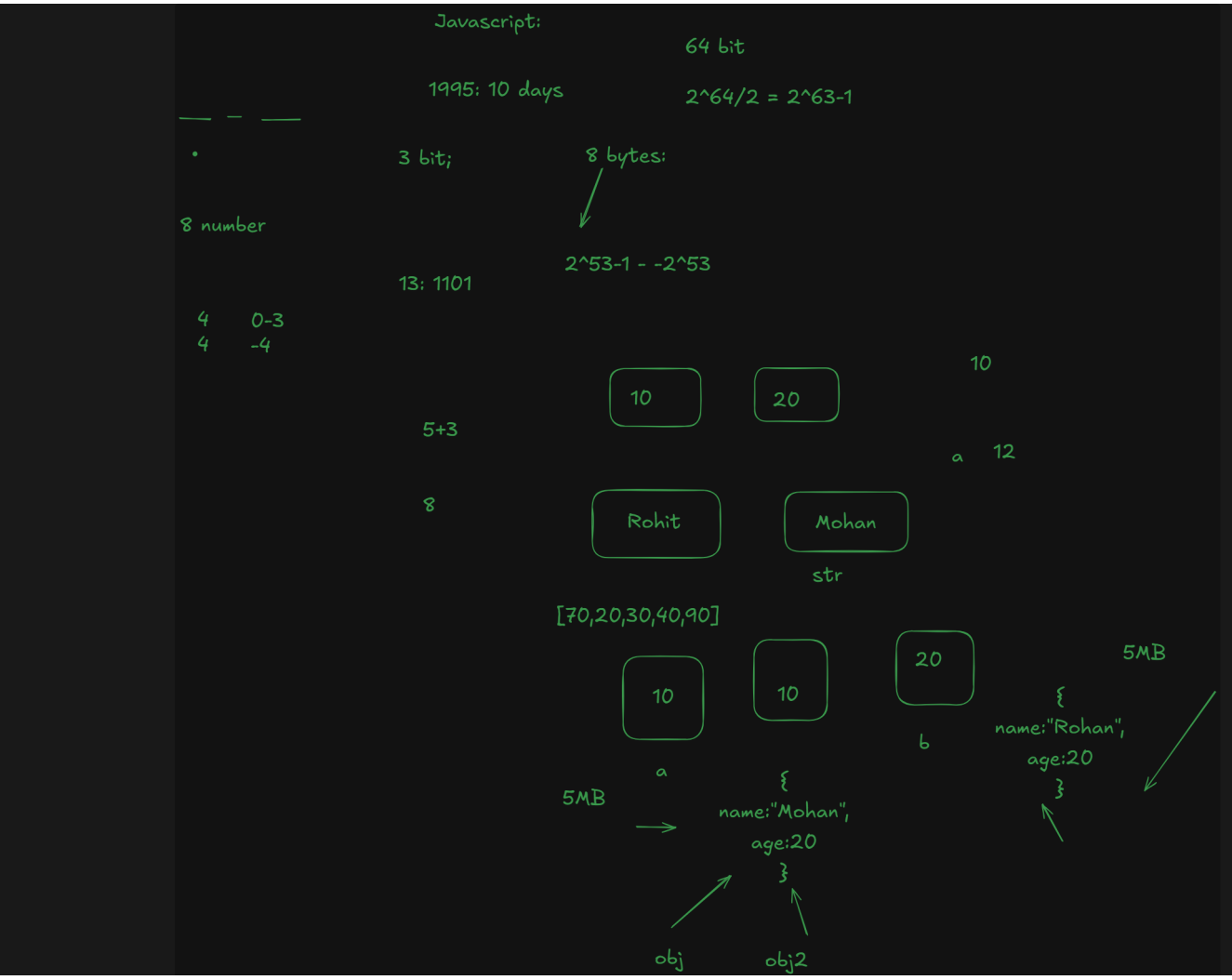
When you assign an object from one variable to another, the **reference (memory address)** is copied, not the object itself. Both variables point to the same object.

```
let obj1 = { value: 10 }; let obj2 = obj1; // The reference to the object is copied into obj2 // Both obj1 and obj2 now point to the exact same object in memory obj2.value = 20; // We are modifying the object through obj2 console.log(obj1.value); // Outputs: 20 (obj1 is affected because it points to the same object) console.log(obj2.value); // Outputs: 20
```

The typeof Operator

To determine the data type of a variable at runtime, you use the typeof operator.

```
typeof "Hello" // "string" typeof 42 // "number" typeof true // "boolean" typeof undefined // "undefined" typeof 10n // "bigint" typeof Symbol('id') // "symbol" typeof { a: 1 } // "object" typeof [1, 2, 3] // "object" typeof function(){} // "function" (a special case for functions) typeof null // "object" (this is a long-standing, well-known bug in JavaScript)
```


Javascript:
64 bit
1995: 10 days
2^64/2 = 2^63-1
3 bit;
8 bytes:
8 number
2^53-1 - -2^53
13: 1101
4 0-3
4 -4
10
10
20
5+3
a 12
8
Rohit
Mohan
str
[70,20,30,40,90]
10
10
20
5MB
{
name:"Rohan",
age:20
}
b
a
5MB
{
name:"Mohan",
age:20
}
obj
obj2