JavaScript Data Types and Concepts

Primitive Types

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `BigInt`
- `Symbol`

Non-Primitive Types

- `Object`
- `Array`
- `Function`

Variable Declaration

Variables can be declared using:

- `var` (function-scoped, hoisting applies)
- `let` (block-scoped, no hoisting issues)
- `const` (block-scoped, cannot be reassigned)

Primitive Type Examples

```
let age = 25; // Number
let name = "John Doe"; // String
let isLoggedIn = true; // Boolean
let user; console.log(user); // undefined
let data = null; // Null
let bigNumber = 123456789012345678901234567890n; // BigInt
let sym1 = Symbol("unique");
let sym2 = Symbol("unique");
console.log(sym1 === sym2); // false (symbols are unique)
```

Using Symbols in Objects

```
const sym = Symbol("key");
const obj = {
   name: "ABD",
   [sym]: "sss"
};
console.log(obj[sym]); // "sss"
console.log(Object.keys(obj)); // ['name'] (symbol keys are ignored)
```

Non-Primitive Type Examples

```
let person = { name: "Alice" }; // Object
let fruits = ["Apple", "Banana", "Cherry"]; // Array
function greet(name) { return `Hello, ${name}!`; } // Function
```

Type Checking

```
console.log(typeof 42); // "number"
console.log(typeof "Hello"); // "string"
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "object" (historical JavaScript bug)
console.log(typeof Symbol("id")); // "symbol"
console.log(typeof 123n); // "bigint"
console.log(Array.isArray([1, 2, 3])); // true
console.log({} instanceof Object); // true
console.log(new Date() instanceof Date); // true
```

Var vs Let vs Const

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted and initialized to `undefined`	Hoisted but not initialized (temporal dead zone)	Hoisted but not initialized (temporal dead zone)

Feature	`var`	`let`	`const`
Re-declaration	Allowed	Not allowed	Not allowed
Initialization	Optional	Optional	Required
Re-assignment	Allowed	Allowed	Not allowed
Use case	Older code, less strict scope needs	Variables that will change, block scope	Constants, variables that should not change

Null vs Undefined

- If we don't assign any value to a variable, `undefined` is assigned as a temporary value until a new value is initialized.
- If we want a variable to hold an empty or no value, we assign it `null`.

Lexical Scope

Scope determines where a variable can be accessed in our code.

```
var a = 10;
function square() {
  var result = 20;
  c();
  function c() {
    console.log(a);
  }
}
square();
```

Execution Context

1. Global Execution Context (GEC) is created, allocating memory for `a` and `square`.
2. When `square` is invoked, an execution context is created for it.
3. Memory for `result` and function `c` is allocated in the execution context.
4. When `c` is invoked, its execution context is created.

5. JavaScript looks for `a` in `c`'s memory, then `square`'s memory, then GEC.

Block Scope

A block is defined by `{}` and groups multiple statements.

```
if (true) {
  var x = 10;
  var y = 20; // Multiple statements
}
```

The block scope refers to variables and functions that can be accessed within `{}`.

Shadowing

Shadowing occurs when a variable in an inner scope has the same name as an outer variable, overriding it within its scope.

```
let x = 10; // Outer variable
function example() {
    let x = 20; // Inner variable shadows outer variable
    console.log(x); // Outputs 20
}
example();
console.log(x); // Outputs 10
```

Summary

- **Primitive vs. Non-Primitive Data Types**: JavaScript has seven primitive types and multiple non-primitive types.
- **Variable Declarations**: `var` (function-scoped), `let` and `const` (block-scoped).
- **Type Checking**: `typeof` operator helps determine data types.
- **Scope Types**:
  - Global Scope: Accessible everywhere.
  - Function Scope: Variables inside functions are limited to that function.
  - Block Scope: Variables inside `{}` are limited to the block.
  - Lexical Scope: Inner functions can access outer function variables.

- **Shadowing**: Inner variables can override outer variables within the same scope.
- **Null vs. Undefined**: `null` represents an empty or intentional absence of value, whereas `undefined` means a variable has been declared but not assigned a value.