Primitive Types

These are immutable and not objects

Type	Example	`typeof` result	Notes
`string`	`"Mehul"`	`"string"`	Immutable
`number`	`42`, `3.14`	`"number"`	`NaN` is also `"number"`
`bigint`	`123n`	`"bigint"`	For very large integers
`boolean`	`true`, `false`	`"boolean"`	—
`undefined`	`undefined`	`"undefined"`	Default value of uninitialized variables
`symbol`	`Symbol("id")`	`"symbol"`	Unique identifier
`null`	`null`	`"object"`	**Tricky!** `typeof null` → `"object"` (historical bug)

Non-Primitive Types (Objects)

All other things are objects:

Type	Example	`typeof` result	Notes
Object	`{a:1}`	`"object"`	Plain object
Array	`[1,2,3]`	`"object"`	Use `Array.isArray(arr)` → `true` to distinguish
Function	`function f(){}`	`"function"`	Callable object, has properties
Date	`new Date()`	`"object"`	Use `instanceof Date` to check
RegExp	`/abc/`	`"object"`	Use `instanceof RegExp`
Map/Set	`new Map()`, `new Set()`	`"object"`	Collection objects
Error	`new Error("msg")`	`"object"`	—

2. Tricky Points About JS Types

1. **`null` is `"object"`**

   ```
   typeof null; // "object" Historical bug, just remember null is a primitive but typeof reports "object".
   ```

2. **Arrays are objects**

   ```
   typeof []; // "object"

   Array.isArray([]); // true

   Arrays are a special kind of object, not a separate type in typeof.
   ```

3. **Functions are objects**

   ```
   typeof function(){}; // "function" Functions are callable objects, so they can have properties.
   ```

4. **`NaN` is a number**

   ```
   typeof NaN; // "number" Not a bug; JS defines NaN as a numeric type.
   ```

5. **Primitive wrapper objects**

   ```
   let str = "hello";
   console.log(str.length); // 5
   ```

   JS temporarily wraps primitives (string, number, boolean) as objects when you access properties.

6. **typeof vs instanceof**

   ```
   console.log([] instanceof Array); // true
   console.log([] instanceof Object); // true
   ```

   typeof only gives "object" or "function" for non-primitives.

   instanceof checks the **prototype chain**, more precise.

QUESTION

1> let a;

console.log(typeof a);

console.log(typeof b);

Answer:

- typeof a → "undefined"
- typeof b → "undefined"
  Why tricky: typeof does **not throw an error for undeclared variables**, unlike normal access.

2> let str1 = "hello";

let str2 = new String("hello");

console.log(typeof str1);

console.log(typeof str2);

console.log(str1 === str2);

Answer:

- typeof str1 → "string"
- typeof str2 → "object"
- str1 === str2 → false
  Why tricky: new String() creates an object wrapper, not a primitive.

3> console.log(typeof 0); // "number"

console.log(typeof false); // "boolean"

console.log(typeof ""); // "string"

They are primitive types but **falsy** values.

4> let sym = Symbol("id");

console.log(typeof sym);

Answer: "symbol"
Why tricky: Symbols are new in ES6 and are a primitive, not an object.

5> `console.log(Object.prototype.toString.call([]));`

`console.log(Object.prototype.toString.call({}));`

`console.log(Object.prototype.toString.call(null));`

Answer:

- `[]` → `"[object Array]"`
- `{}` → `"[object Object]"`
- `null` → `"[object Null]"`
  Why tricky: `Object.prototype.toString` gives a **more precise type** than `typeof`.

6> `console.log(typeof function(){} === "object");`

Answer: `false`
Why tricky: `typeof returns "function" for functions, even though functions are technically objects.`

Hoisting

Key points:

- Only **declarations** are hoisted, not initializations.
- Works for **variables (`var`)** and **functions**.
- `let` and `const` are hoisted **but are in a temporal dead zone** (TDZ).

-> var Hoisting

```
console.log(a); // undefined
var a = 5;
console.log(a); // 5
```

`var a; is hoisted to the top → undefined`

- Initialization `= 5` stays in place.

Tricky point: Accessing a `var` before declaration does **not throw error**, just gives `undefined`.

-> let and const Hoisting

```
console.log(b); // ReferenceError
let b = 10;
console.log(c); // ReferenceError
const c = 20;
```

`Declarations are hoisted, but variables are in` **`temporal dead zone`** `until initialized.`
Accessing them before declaration **throws ReferenceError**.

-> Function Declaration Hoisting

```
sayHello(); // "Hello!"
function sayHello() {
  console.log("Hello!");
}
```

`Function declarations are` **`fully hoisted`** `(both name and body).`
Can be called **before declaration**.

-> Function Expression Hoisting

```
sayHi(); // TypeError: sayHi is not a function
var sayHi = function() {
  console.log("Hi!");
}
var sayHi; is hoisted → undefined
```

Initialization is **not hoisted**, so calling it throws `TypeError`.

-> Arrow Function Hoisting

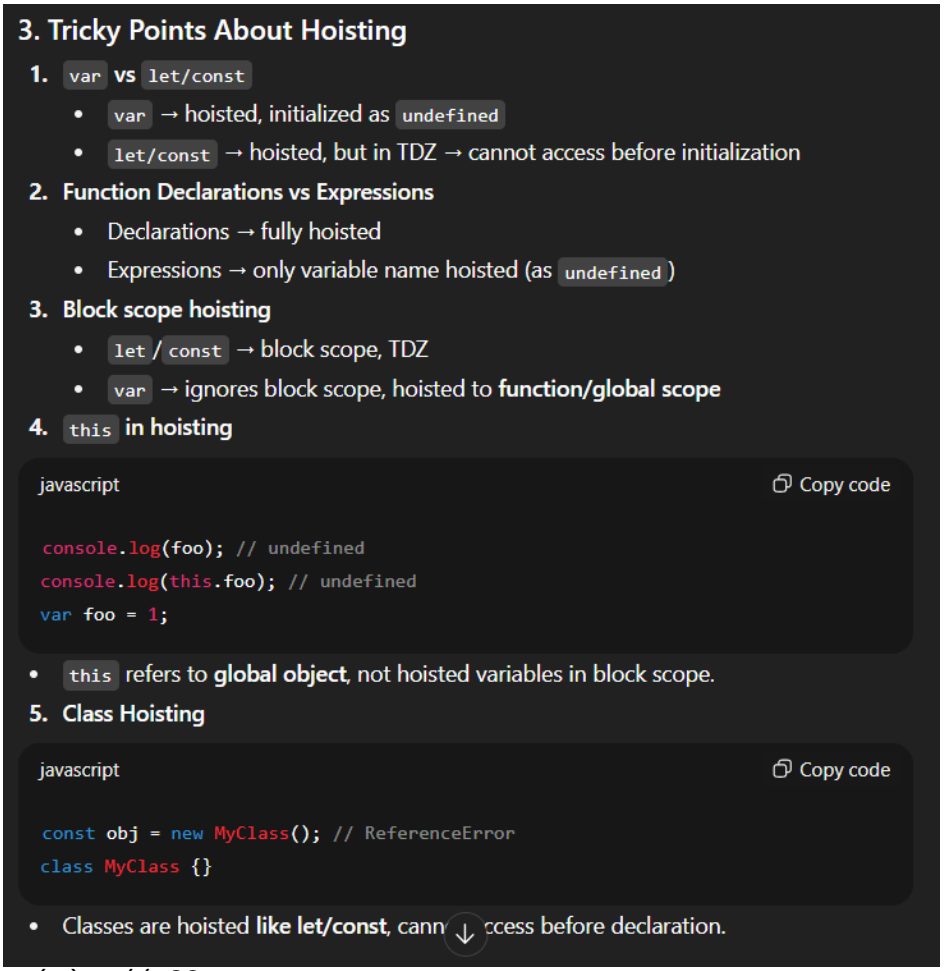
```
hello(); // ReferenceError
let hello = () => console.log("Hi");
Like let/const, arrow functions are not hoisted.
```

-> Hoisting in Blocks

```
{
  console.log(x); // ReferenceError
  let x = 10;
}
let/const are block-scoped → TDZ applies inside the block.
```

`var` ignores block scope:

```
{
    console.log(y); // undefined
    var y = 20;
}
```




```
console.log(y); // 20
```

Declaration Type	Hoisted?	Initialization?	Can Access Before?	Scope
`var`	Yes	No (undefined)	Yes (undefined)	Function/global
`let`	Yes	No	No (TDZ)	Block
`const`	Yes	No	No (TDZ)	Block
Function decl.	Yes	Yes	Yes	Function/global
Function expr.	Yes (var)	No	No	Function/global or block if let/const
Class	Yes	No	No	Block