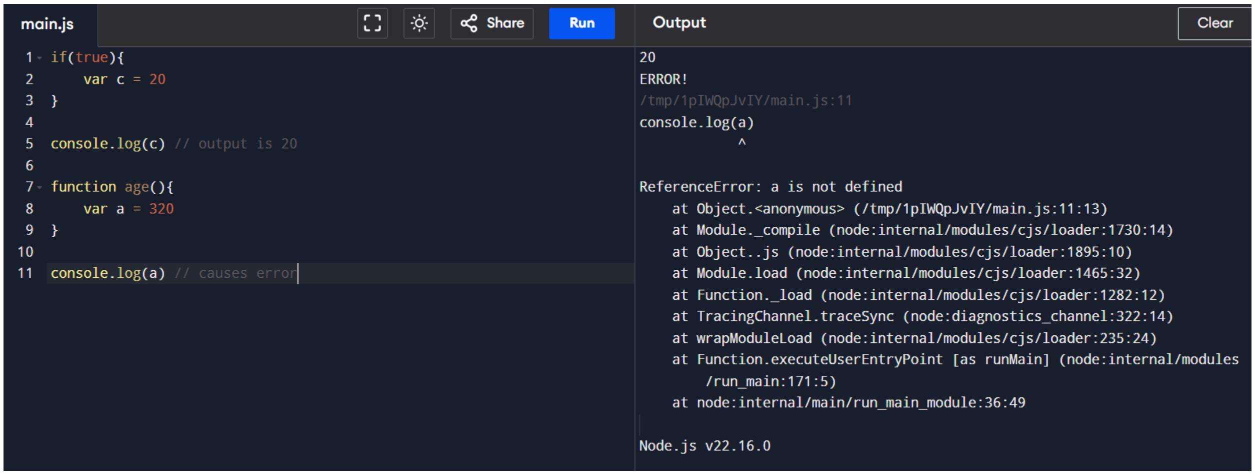
With “var” keyword, you could define a variable as many times as you want but there is one problem with it , the problem is that it doesn’t respect scoping

Means you declare variable inside the if block and you can log it outside the if block,

“Var” keyword only respects global scope and functional scope



```
if(true){
    var c = 20
}

console.log(c) // output is 20

function age(){
    var a = 320
}

console.log(a) // causes error
```

```
20
ERROR!
/tmp/1pIWQpJvIY/main.js:11
console.log(a)
            ^

ReferenceError: a is not defined
    at Object.<anonymous> (/tmp/1pIWQpJvIY/main.js:11:13)
    at Module._compile (node:internal/modules/cjs/loader:1730:14)
    at Object..js (node:internal/modules/cjs/loader:1895:10)
    at Module.load (node:internal/modules/cjs/loader:1465:32)
    at Function._load (node:internal/modules/cjs/loader:1282:12)
    at TracingChannel.traceSync (node:diagnostics_channel:322:14)
    at wrapModuleLoad (node:internal/modules/cjs/loader:235:24)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules
        /run_main:171:5)
    at node:internal/main/run_main_module:36:49

Node.js v22.16.0
```

✅ When you write:

let p;

What actually happens?

1. **A variable named `p` is created in memory.**
2. **A *slot* is reserved for it** — meaning memory is allocated.
3. **BUT you haven’t assigned any value**, so…
4. JavaScript automatically sets its value to `undefined`.

❗ Does it “hold a memory reference”?

Not really — because there is no value yet.

A reference only exists **when the value is an object**, like:

```
let p = {};
```

or

```
let p = [];
```

or

```
let p = someOtherObject;
```

But in your case:

```
let p;
```

There is **NO object**, so there is **NO reference**.

🧠 So why does it show `undefined`?

Because JavaScript has a simple rule:

Any declared variable without a value = `undefined`

Examples:

```
let x;
console.log(x); // undefined

var y;
console.log(y); // undefined

const z; // ❌ ERROR — const requires a value
```

📌 Important distinction

`undefined` means:

- The variable **exists**
- But **no value has been assigned**

`not defined` means:

- The variable itself **does NOT exist**

Example:

console.log(a); // ❌ ReferenceError: a is not defined

✔ Summary (easy version)

Code	Exists in Memory?	Has Value?	Result
`let p;`	✅ Yes	❌ No	`undefined`
`let p = 5;`	✅ Yes	5	`5`
`let p = {};`	✅ Yes	object reference	`{}`
`p` without declaring	❌ No	—	ReferenceError

🏁 Final simple explanation for you

`let p;` means:

- JS creates variable `p`
- Gives it a memory slot
- But since no value is given yet
- JavaScript automatically assigns `undefined`

There is **NO reference** because a reference only exists if `p = somethingObject`.

If you want, I can draw a memory diagram like this:

```
Variable   | Value
-----------|---------
p          | undefined
```

Primitive values → stored in the stack

`10` is a **primitive value** (number).

So when you write:

```
let p = 10;
```

- `p` (the variable) is stored in the **stack frame**
- The value `10` is also stored in the **stack**
- No heap allocation happens for primitive values

📌 In JavaScript, primitives include:

- number
- string
- boolean
- null
- undefined
- bigInt
- symbol

All of these go on the **stack**.

Number takes 8 bytes means 64 bits of memory, so 8 bytes not only stores integer but also it has to store floating number as well and the range is from 2^53 -1 to -2^53, so in this range you can store the negative as well as positive numbers and for floating points , it is different

And also bigint can't be stored in the 8 bytes of memory

Undefined and null , when to use

Null means temperature nahi bata sakta hu currently , dwarka ka

Undefined means bhaiyya aapne dwarka as a city database me daala hi nahi hai

Symbol is used to create unique stuffs, koi bhi value doge toh unique value create karke degaa,

If you pass same string inside Symbol() , it is still capable to create different value and unique

Objects are used to create meaning out of data

Function bhi object hi hota hai mere dost bhalehi typeof karke function ayega

Let str = “Rohit”

Str[0] = “M”

It rejected the change !! , jaa nahi karna kyuki me immutable hoon

---

1. Introduction to the Lecture and JavaScript's Philosophy

- **Overview**: This is the second lecture in a JavaScript series focusing on variables, data types, and internals. The goal is to transition from junior to senior-level understanding by exploring how JavaScript works behind the scenes (e.g., data storage). The instructor emphasizes asking the right questions and exploring answers deeply.
- **Key Insight**: JavaScript was designed for simplicity, targeting web developers familiar with HTML/CSS, not professional programmers. It avoids complex syntax to make it intuitive—your mental model matches the language's behavior.
- **Historical Context**: JavaScript was created in 1995 by Brendan Eich in just 10 days. This rapid development explains some quirks (e.g., bugs like typeof null). The series aims to explore Eich's thought process for better intuition.
- **In-Depth Explanation**: JavaScript's design philosophy prioritizes ease: no need for explicit types (dynamically typed), minimal boilerplate. This contrasts with languages like Java or C++, where you declare types upfront. Understanding this helps avoid overcomplicating code—think like a web designer, not a systems programmer.
- **Example**: No functions or classes needed for basic scripts; just write code directly.

2. Creating Variables with 'let'

- **Overview**: Variables store data. Use let to declare a variable, assign a name, and optionally a value.
- **Syntax**: let variableName = value;
- **Key Features**:
    - Mutable: Values can be changed later.
    - Block-scoped: Limited to the block (e.g., {}) where declared.
- **In-Depth Explanation**: let was introduced in ES6 (2015) for modern variable declaration. It prevents redeclaration in the same scope, reducing bugs. Variables are "hoisted" but in a Temporal Dead Zone (TDZ)—you can't access them before declaration.

Example from Transcript:

JavaScript

```javascript
let name = "Rohit";

console.log(name); // Outputs: Rohit

let age = 20;

console.log(name, age); // Outputs: Rohit 20

age = 30; // Valid, changes value
```

- `console.log(age); // Outputs: 30`
- **Running Code**: Use Node.js: node filename.js.
- **Why Mutable?**: Reflects real-world intuition—update values without restrictions, keeping the language simple.

3. Constants with 'const'

- **Overview**: Use const for values that shouldn't change after assignment.
- **Syntax**: const variableName = value;
- **Key Features**:
    - Immutable value: Cannot reassign.
    - Must assign value at declaration (no empty const x;).
    - Block-scoped like let.
- **In-Depth Explanation**: const ensures constants remain fixed, useful for configuration (e.g., API keys). For objects/arrays declared with const, the reference is immutable, but properties/elements can mutate (shallow immutability).

Example:

JavaScript

```javascript
const account = 1234;

console.log(account); // Outputs: 1234
```

- account = 23; // Error: Assignment to constant variable
- **Difference from 'let'**: let allows reassignment; const enforces finality, promoting safer code.

4. Old Variable Declaration with 'var'

- **Overview**: Legacy method from pre-ES6; avoid in modern code.
- **Syntax**: var variableName = value;
- **Issues**:
  - Allows redeclaration: var a = 10; var a = 20; (no error, overrides).
  - Function-scoped or global-scoped: Ignores block scope (e.g., if/loop blocks).
  - Hoisted fully: Declared at top of scope, initialized as undefined.
- **In-Depth Explanation**: var leads to bugs like accidental globals or scope leaks. ES6's let/const fix this with block scoping. Use var only for legacy support.

Example:
JavaScript

```
var a = 10;

console.log(a); // 10

var a = 20; // Redeclaration, no error

console.log(a); // 20

if (true) {

  var b = 30; // Not block-scoped

}
```

- console.log(b); // 30 (leaks outside if)
- **Why Avoid?**: No respect for block scope; allows multiple declarations, causing confusion.

5. Scope in JavaScript

- **Overview**: Scope determines variable accessibility.
- **Types**:
  - Global: Outside any function/block.
  - Function: Inside functions.
  - Block: Inside {} (for let/const).
- **'var' vs 'let/const'**: var is function/global; let/const respect blocks.
- **In-Depth Explanation**: Lexical scoping—child scopes access parent vars. var creates globals accidentally (e.g., in loops). This leads to memory leaks or conflicts.

Example:

JavaScript

```
if (true) {

  let c = 90; // Block-scoped

}

console.log(c); // Error: c is not defined

if (true) {

  var d = 100; // Function/global-scoped

}
```

- `console.log(d); // 100`
- **Best Practice**: Use let for mutables, const for immutables; avoid var.

6. Data Types in JavaScript

- **Overview**: Dynamically typed—no explicit types; inferred from value.
- **Categories**:
  - Primitive (simple, immutable): Number, String, Boolean, Undefined, Null, BigInt, Symbol (7 total).
  - Non-primitive (complex, mutable): Array, Object, Function.
- **In-Depth Explanation**: Primitives are stored by value; non-primitives by reference. This affects copying and mutation.

7. Primitive Data Types

7.1 Number

- **Overview**: Integers and floats (e.g., 10, 3.14).
- **Storage**: 8 bytes (64 bits) for double-precision floats (IEEE 754).
- **Range**: Safe integers: -2^53 + 1 to 2^53 - 1 (due to float sharing).

Example:

JavaScript

```
let a = 10;

let b = 2.36;
```

- `console.log(typeof a); // number`

7.2 String

- **Overview**: Text in single (") or double ("") quotes.
- **Immutable**: Can't change characters directly.

Example:

JavaScript

```
let str = "Rohit";

console.log(str[0]); // 'R' (access ok)

str[0] = 'M'; // No error, but no change (immutable)
```

- `console.log(str); // Still "Rohit"`

7.3 Boolean

- **Overview**: true or false.
- **Use**: Conditions, flags (e.g., isLoggedIn).

Example:

JavaScript

```
let isLoggedIn = true;
```

- `console.log(typeof isLoggedIn); // boolean`

7.4 Undefined

- **Overview**: Default for uninitialized variables.
- **Meaning**: Unintentional absence of value (system-assigned).

Example:

JavaScript

```
let user;

console.log(user); // undefined
```

- `console.log(typeof user); // undefined`

7.5 Null

- **Overview**: Intentional absence of value (developer-assigned).
- **Bug**: typeof null === 'object' (legacy issue, not fixed to avoid breaking code).
- **Difference from Undefined**: Null for "exists but no value now"; undefined for "not assigned/exists".
- **Real-World Example**: Weather API:

    - 25: Temperature value.
    - Null: City exists, but can't fetch temp now.
    - Undefined: City doesn't exist in database.

Example:

JavaScript

```
let weather = null;
```

- `console.log(typeof weather); // object (bug)`

7.6 BigInt

- **Overview**: For integers beyond Number's safe range.
- **Syntax**: Append 'n' (e.g., 9007199254740992n).
- **Use**: Cryptography, large IDs.

Example:

JavaScript

```
let bigNum = 9007199254740992n;
```

- `console.log(typeof bigNum); // bigint`

7.7 Symbol

- **Overview**: Unique, immutable primitives for keys/objects.
- **Use**: Unique identifiers (e.g., object properties).

Example:

JavaScript

```
const id1 = Symbol('id');

const id2 = Symbol('id');

console.log(id1 === id2); // false (unique even if same description)
```

- `console.log(typeof id1); // symbol`

8. Non-Primitive Data Types

8.1 Array

- **Overview**: Ordered collections; heterogeneous types.
- **Mutable**: Add/remove elements.
- **typeof**: 'object' (not a bug; arrays are objects).

Example:
JavaScript

```
let arr = [10, 20, "Rohit", true];

arr.push(90); // Mutates

console.log(arr); // [10, 20, 'Rohit', true, 90]
```

- `console.log(typeof arr); // object`

8.2 Object

- **Overview**: Key-value pairs for structured data.
- **Mutable**: Change properties.

Example:
JavaScript

```
let user = { name: "Rohit", age: 18, category: "General" };

user.name = "Rohan"; // Mutates

console.log(user); // { name: 'Rohan', age: 18, category: 'General' }
```

- `console.log(typeof user); // object`

8.3 Function

- **Overview**: Reusable code blocks; first-class (storable in variables).
- **typeof**: 'function' (but internally an object).

Example:
JavaScript

```
function add() { return "Hello"; }

let s = add; // Store in variable

console.log(s()); // Hello
```

- `console.log(typeof s); // function`

9. typeof Operator

- **Overview**: Checks data type.
- **Quirks**: null → 'object'; arrays/functions → 'object'/'function'.
- **In-Depth**: Proves non-primitives are objects under the hood.
- **Example**: See above.

10. Immutability vs Mutability

- **Primitives (Immutable)**: Can't change value once created; reassignment creates new value.
  - Proof: String char change fails.
- **Non-Primitives (Mutable)**: Change contents without new creation.
- **Reason**: Efficiency—primitives are small; non-primitives can be large.

11. Pass by Value vs Pass by Reference

- **Primitives**: Copy value (independent changes).
  - Example: let a = 10; let b = a; b = 20; → a=10, b=20.
- **Non-Primitives**: Share reference (changes affect all).
  - Example: Object/array copies mutate original.
- **Memory Reason**: Primitives (small): Copy ok. Non-primitives (large): Reference to save memory (e.g., avoid duplicating 5MB objects).

12. Internal Storage and Future Topics

- **Teased**: Stack (primitives) vs Heap (non-primitives); V8 engine details in next lecture.
- **In-Depth Insight**: Primitives on stack (fast, fixed-size); objects on heap (dynamic, pointers).

These notes cover the transcript comprehensively. For code execution or clarification, refer to tools if needed. If you have questions, ask!

(Optional Topic)

⭐ STARTING POINT: “Number takes 8 bytes (64 bits)”

Yes — JavaScript's `Number` type always uses **64 bits**.

But those 64 bits have to store:

✔ positive numbers
✔ negative numbers
✔ integers
✔ floating-point numbers (decimals)
✔ very small numbers
✔ very large numbers
✔ special values (NaN, Infinity)

So the 64 bits must be divided into sections.

⭐ HOW THE 64 BITS ARE USED

1 bit → sign (positive/negative)
11 bits → exponent (scale of the number)
52 bits → fraction/mantissa (precision digits)

This format is called **IEEE 754 Double Precision**.

⭐ IMPORTANT PART: "Only 53 bits store the actual number's digits"

There are:

- 52 fraction bits
- +1 hidden leading bit
- = **53 bits of precision**

So JavaScript can exactly store **53 bits worth of integers**.

Meaning:

⟹ The largest EXACT integer JS can store is

$2^{53} - 1$ (9007199254740991)

And the smallest EXACT negative integer is
$-(2^{53} - 1)$

This is the range you're talking about.

⭐ WHY ONLY $2^{53}-1$? WHY NOT USE ALL 64 BITS FOR INTEGER?

Because JavaScript does NOT separate integer and floating-point types.

Everything is floating-point.

So, JavaScript must reserve:

- **1 bit** for sign
- **11 bits** for exponent (to allow scientific notation like 1.23e18)
- Only **52 bits** left for fraction (precision)

Thus:

🔹 Only **precision bits** matter for integers
🔹 Precision = **53 bits**
🔹 Therefore maximum safe integer = **$2^{53} - 1$**

⭐ Now your statement, explained correctly:

> **“Number takes 8 bytes means 64 bits of memory”**

✔ Yes.

“So 8 bytes not only stores an integer but also floating numbers”

✔ Correct — **one single format** stores both.

“Range is from 2^53 − 1 to −2^53”

✔ That is the range where integers can be stored **exactly without losing precision**.

“So in this range you can store negative and positive numbers”

✔ Yes — integers are safe in that range.

“and for floating points, it is different”

✔ Exactly — floats follow a different rule:

- They can represent MUCH larger numbers (up to 10^{308})
- BUT they can't represent every integer exactly beyond 2^{53}

Because floats use scientific notation.

⭐ SUPER SIMPLE SUMMARY (easy to remember)

JavaScript Number = 64-bit floating-point format

Part	Bits	Purpose
Sign	1	+ or –
Exponent	11	How big/small the number is
Fraction	52	Digits of the number

✔ Only 53 bits of precision

→ Maximum exact integer: **±(2^{53} − 1)**
→ Floats can go huge, but integers beyond this lose accuracy.

⭐ ULTRA-SIMPLE EXPLANATION (one sentence)

JavaScript uses 64-bit floating-point numbers, and because only 53 bits are used to store the digits, integers are safely representable only from $-(2^{53}-1)$ to $+(2^{53}-1)$, while floating-point values have a totally different and much larger range.

Here is **the simplest, clearest example** you will ever see.
This will remove ALL your confusion.

Let's take a very simple number:

⭐ Example: How does 5 look inside a 64-bit JavaScript Number?

You wrote:

1 bit → sign

11 bits → exponent

52 bits → fraction (mantissa)

Let's put the number **5** inside this structure.

⭐ Step 1: Write 5 in binary

5 = 101 (in binary)

But a floating-point number must be written in **scientific binary form**:

101 → 1.01 × 2^2

So:

- **1.01** is the significant digits
- **2^2** is the exponent

⭐ Step 2: Fill the 64-bit IEEE754 slots

➢ SIGN bit (1 bit)

5 is positive →

sign = 0

➢ EXPONENT (11 bits)

The exponent used internally is:

exponent = actual_exponent + bias

bias = 1023

For 5, actual exponent = 2:

exponent = 2 + 1023 = 1025

Binary of 1025 (11 bits):

10000000001

➢ FRACTION / MANTISSA (52 bits)

From scientific form:

1.01 × 2^2

We drop the leading 1 (because it’s “hidden” — always assumed).

Mantissa stores only the fractional part after the dot:

.01

Pad it to 52 bits:

0100000000000000000000000000000000000000000000000000

⭐ Final Encoding of Number 5 in JS

Field	Bits
Sign	`0`
Exponent	`10000000001`
Fraction	`0100000000000000000000000000000000000000000000000000`

Full 64-bit number:

0 10000000001 0100000000000000000000000000000000000000000000000000

That is **how JavaScript stores the number 5** in memory.

⭐ Why integer precision is limited

Now count the precision bits:

- 52 bits of fraction
- 1 hidden leading bit
- = **53 bits of precision**

So you cannot represent integers beyond **2^{53}** exactly.

That's why JavaScript max safe integer = **$2^{53} - 1$**.

⭐ SUPER SIMPLE SUMMARY

A JavaScript number:

[sign (1 bit) | exponent (11 bits) | mantissa (52 bits)]

Example:

5 → 1.01×2^2

sign = 0

exponent = 2 + 1023

mantissa = 01 (padded)

Awesome — let’s break down **0.1** (the most famous floating-point example) in a **very simple, clear way**.

⭐ Why 0.1 is important

0.1 **cannot** be represented exactly in binary floating-point.
Just like **1/3 cannot be exactly written in decimal (0.3333…)**,
0.1 cannot be exactly written in binary — it becomes an infinite repeating fraction.

This is why:

0.1 + 0.2 !== 0.3 // true 😭

Let me show you **how 0.1 sits inside those 64 bits**.

⭐ Step 1: Convert 0.1 to binary (approx)

0.1 in decimal
= **0.000110011001100110011001100110011001100… (repeating forever)** in binary.

It never ends.

To fit inside JavaScript, it must be *cut off* to fit **53 bits** of precision.

⭐ Step 2: Convert binary to scientific form

Take the repeating binary and convert it to:

1.10011001100110011001100110011001100110011… × 2^{-4}

So:

- significand: `1.10011001100...`
- exponent: `-4`

⭐ Step 3: Fill IEEE-754 slots

➢ SIGN bit

0.1 is positive →
 `sign = 0`

➢ EXPONENT (11 bits)

IEEE format uses a **bias of 1023**.

Exponent we have is `-4`, so:

exponent = actual_exponent + bias

= -4 + 1023

= 1019

Binary of 1019 (11 bits):

01111111011

➢ FRACTION / MANTISSA (52 bits)

We remove the leading `1.` (hidden bit).
 Then store the next 52 bits of the repeating binary digits.

Mantissa (52 bits):

1001100110011001100110011001100110011001100110011010

These bits are the **BEST APPROXIMATION** JavaScript can store.

⭐ Final 64-bit representation of 0.1

0 01111111011 1001100110011001100110011001100110011001100110011010

(Sign | Exponent | Mantissa)

⭐ And now the magic / problem:

This 64-bit encoding is **not exactly 0.1**.
It's a tiny bit larger:

Actual stored value:

0.1000000000000000055511151231257827021181583404541015625

So when you add:

0.1 + 0.2 →

0.3000000000000000444...

Not exactly 0.3 → comparison fails.

⭐ SUPER SIMPLE SUMMARY for you

Part	Bits	Meaning
Sign	1	positive/negative
Exponent	11	how big/small number is
Mantissa	52	precision digits

0.1 cannot be represented exactly in binary
→ becomes repeating
→ must be rounded to 53-bit precision
→ small error
→ leads to weird behavior like `0.1 + 0.2 !== 0.3`.

⭐ Want me to show how 0.2 or 0.3 are represented?

Or why `0.1 * 3` also fails?

Great — let’s look at **0.2** and **0.3** the same way, and then understand why floating-point arithmetic becomes weird.

I’ll keep it **simple, clear, and visual**, just like before.

⭐ 1. Representation of 0.2 in IEEE-754 (64-bit)

Step 1: Convert 0.2 into binary

0.2 in decimal becomes:

0.001100110011001100110011001100110011001100… (repeating)

This is also an infinite repeating fraction in binary — just like 0.1.

Step 2: Scientific binary form

1.10011001100110011001100110011001100110011… × 2^{-3}

- significand = `1.10011001100...`
- exponent = `-3`

Step 3: IEEE-754 fields

➢ SIGN

`0` (positive)

➢ EXPONENT

Exponent = -3
Bias = 1023
Stored exponent = 1020

Binary (11 bits):

01111111100

➢ FRACTION (mantissa)

52 bits taken from the repeating digits:

1001100110011001100110011001100110011001100110011010

(You see the same repeating pattern as 0.1 because 0.1 and 0.2 share the same repeating cycle.)

⭐ Final 64-bit representation of 0.2

0 01111111100 1001100110011001100110011001100110011001100110011010

Stored value:

0.200000000000000011102230246251565404236316680908203125

Not exactly 0.2 — slightly larger.

⭐ 2. Representation of 0.3

0.3 = 0.1 + 0.2
So we expect the stored value to also be slightly “off”.

0.3 in binary:

0.01001100110011001100110011001100110011… (repeating)

Scientific:

1.001100110011001100110011001100110011… × 2^{-2}

➢ SIGN

`0`

➢ EXPONENT

Exponent = -2
Stored = 1021
Binary:

01111111101

➢ FRACTION

Mantissa (52 bits):

0011001100110011001100110011001100110011001100110011

⭐ Final 64-bit representation of 0.3

0 01111111101 0011001100110011001100110011001100110011001100110011

Stored value:

0.299999999999999988897769753748434595763683319091796875

Slightly smaller than 0.3.

⭐ 3. NOW the whole picture: Why 0.1 + 0.2 !== 0.3

Look at these:

	Actual stored value	Difference

0.1 = 0.10000000000000000555…	slightly bigger
0.2 = 0.20000000000000001110…	slightly bigger
0.3 = 0.29999999999999998889…	slightly smaller

Now add actual stored values of 0.1 + 0.2:

0.10000000000000000555…

+0.20000000000000001110…

=0.30000000000000001666…

And compare with stored 0.3:

0.29999999999999998889…

They are not equal ✔
So JavaScript says:

0.1 + 0.2 === 0.3 // false

⭐ 4. Extremely simple takeaway

✔ 0.1, 0.2, 0.3 cannot be stored exactly

They become infinite binary fractions.

✔ JavaScript stores approximations using 53 bits.

✔ 0.1 and 0.2 are stored slightly higher than real value.

✔ 0.3 is stored slightly lower than real value.

✔ So 0.1 + 0.2 overshoots the stored 0.3.

That’s why:

0.1 + 0.2 !== 0.3

⭐ If you want, I can also explain:

- why **integers DO NOT have this issue**
- why **BigInt solves the problem**
- why **decimal.js / big.js** libraries are used
- how **banking apps** avoid floating-point bugs

Perfect — let’s go through **ALL four topics**, step-by-step, clearly and cleanly.
(No confusion. No complicated math. Just pure understanding.)

⭐ 1. Why integers do NOT have floating-point issues (until 2^{53} − 1)

JavaScript can represent **53 bits of integer precision exactly**.

That means **every integer from:**

−(2^53 − 1) to +(2^53 − 1)

is stored **perfectly**.

Example:

5   → exact

1000   → exact

99999999999999 (14 digits) → exact

Because integers fit perfectly into the 53-bit mantissa.

❗ Why no issues?

Because integers don’t need repeating fractions like:

0.1   → 0.0001100110011...(repeat)

1/3   → 0.010101010101...(repeat)

Binary integers always end cleanly.

Example:

13 → 1101 (clean!)

7 → 111   (clean)

1024 → 10000000000 (clean)

No infinite fractional parts → no rounding → no error.

✔ Integers do break after $2^{53} - 1$

Example:

9007199254740992 === 9007199254740993

// true 😨 (precision lost)

Because beyond this range, 53 bits are not enough to store the integer EXACTLY.

⭐ 2. Why BigInt solves the problem

JavaScript normally uses **floating-point** for all numbers.

BigInt uses unlimited precision integer arithmetic.

Example:

let x = 9007199254740993n;

The n suffix tells JS:

> “This is a BigInt. Store it as an arbitrary-length integer.”

BigInt stores numbers like:

[digits][digits][digits][digits]....

There’s no exponent, no mantissa, no 53-bit limit.

✔ BigInt supports:

- Exact integers
- Unbounded size
- No rounding

- No floating-point conversion

Example:

```
2n ** 200n
```

Works perfectly.

⭐ 3. Why decimal.js / big.js are used

These libraries give you **base-10 exact decimal arithmetic**, not base-2 floating arithmetic.

Problem with JavaScript native:

```
0.1 + 0.2 = 0.30000000000000004
```

With decimal.js:

```
decimal(0.1).plus(0.2).toString();

// "0.3"
```

Because the library stores values like:

```
0.1 → "0.1"   (string or bignum decimal)
```

and does arithmetic in **decimal (base 10)** — same as humans.

✔ Accuracy

✔ No rounding surprises
✔ Suitable for money, percentages, interest, etc.

⭐ 4. How banking apps avoid floating-point bugs

Banks **NEVER** use floating-point numbers like 0.1 or 104.98.

They use **integer arithmetic**, like this:

✔ Rule 1: Store money in the smallest unit

Instead of:

₹12.34

banks store:

1234 paise

Instead of:

$45.67

banks store:

4567 cents

So all operations become **integer operations**:

1234 + 567 = 1801

No decimal issues. No rounding surprises.

⭐ Why is this done?

Because:

0.1 rupee ≠ exactly representable in binary.

But:

10 paise → integer → exact.

⭐ Rule 2: Use fixed-precision decimal libraries (like decimal.js)

Banks often use:

- **Decimal128** (MongoDB)
- **BigDecimal** (Java)
- **decimal.js** (JavaScript)
- **BigNumber** (Python)
- **NUMERIC(38, 18)** (PostgreSQL)

These store decimal numbers EXACTLY, not in floating-point.

⭐ Rule 3: Use string formatting for user display

When showing money to humans:

1801 paise → "₹18.01"

But inside the system it always stays **integer paise**.

⭐ Putting It All Together (Easy Summary)

✔ Integers don't break (until $2^{53} - 1$)

Because they fit perfectly into the 53-bit mantissa.

✔ BigInt solves integers beyond that.

Unlimited precision, no rounding.

✔ decimal.js / big.js solve decimal math.

Because they operate in **base-10**, not floating base-2.

✔ Banks avoid floating-point:

- Store money in smallest unit (paise/cents)
- Use integer math
- Use decimal libraries
- NEVER use JS Number for money

If you want, I can also explain:

- how **floating-point rounding works**
- what is **banker's rounding**
- why **0.1 * 3 !== 0.3**
- how JS internally stores **IEEE-754 special values (NaN, Infinity)**

Just tell me **which one**.

Alright — **SUPER SIMPLE, NO BRAINROT VERSION**.
Short, crisp, clean.
No equations unless needed.

✅ 1. What is a denormal/subnormal number?

A **subnormal number** is a tiny number that is *too small* to fit the normal IEEE-754 format.

When numbers get very close to **0**, they don't have enough bits left for their exponent, so IEEE switches to a special format to still represent them — but with **fewer precision bits**.

Think of it like:

> "I can't store this precisely, but I won't drop it to zero — I'll keep whatever bits I can."

Example:
`5e-324` is the smallest subnormal number in JS.

✅ 2. Why small numbers lose precision (underflow)?

Because when a number gets extremely small, the exponent cannot go lower than a limit.

So when the number is too tiny:

- it becomes **subnormal** → loses precision
- eventually becomes **0** → called **underflow**

So:

very_small_number ≈ 0

✅ 3. Why big numbers lose precision (overflow)?

Large numbers need a bigger exponent.

But exponent also has a max.

When you exceed it:

- the number jumps to **Infinity** → called **overflow**
- or precision becomes coarse → gaps between numbers grow

For large numbers, the gaps might be:

… 9,007,199,254,740,992
… 9,007,199,254,740,994
(notice: +2 jump)

You cannot represent every integer anymore.

✅ 4. What is ULP (Unit in Last Place)?

ULP = **the difference between two adjacent representable floating-point numbers.**

It is the smallest "step size" at that scale.

- For numbers close to 1 → ULP is very small
- For big numbers → ULP becomes very large

- For tiny subnormals → ULP becomes irregular

ULP determines how precise a number **can possibly be**.

✅ 5. Why floating-point addition is not associative?

Because each addition rounds the result differently.

So:

(a + b) + c

rounds once, then adds.

a + (b + c)

rounds in a *different order*.

If one number is very large and another very small:

1e16 + 1 ≈ 1e16   (1 gets lost)
(1e16 + 1) + 1 ≈ 1e16
1e16 + (1 + 1) = 1e16 + 2 (the 2 survives)

So:

(a + b) + c ≠ a + (b + c)

Floating-point math breaks standard algebra laws.

⭐ SIMPLE SUMMARY

Your brain can rest with this:

- **Subnormal numbers** = “I’m too small but don’t kill me yet.”
- **Underflow** = number so small it becomes 0.
- **Overflow** = number so big it becomes Infinity.
- **ULP** = the smallest jump between two floating numbers.
- **Addition isn’t associative** because rounding happens at different times.
