java script Random concepts

1. **Primitive Data Types**
   These are the basic building blocks. They are **immutable** and stored by **value**.
2. **Classic `for` loop**

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

2. **`for...of` loop**
   Used to iterate **over iterable objects** like arrays, strings, maps, sets, etc.

```
const arr = ['a', 'b', 'c'];
for (const item of arr) {
  console.log(item);
}
```

3. **`for...in` loop**
   Used to iterate **over enumerable properties** (keys) of an object.

```
const obj = { a: 1, b: 2 };
for (const key in obj) {
  console.log(key, obj[key]);
}
```

4. **`Array.prototype.forEach()`**
   Method to loop over arrays using a callback.

```
const nums = [1, 2, 3];
nums.forEach((num, index) => {
  console.log(index, num);
});
```

5. **`while` loop**

```
let i = 0;
while (i < 5) {
```

```
  console.log(i);
  i++;
}
```

6. **`do...while` loop**
   Executes the block **at least once**, then checks the condition.

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

7. **`map()`**
   Used for transforming arrays and returning a new one.

```
const nums = [1, 2, 3];
const squared = nums.map(num => num * num);
console.log(squared);
```

Primitive Data Types in JavaScript

Type	Example	Description
`String`	`'hello'`	Sequence of characters (text values)
`Number`	`42`, `3.14`	Integer and floating-point numbers
`Boolean`	`true`, `false`	Logical values (true or false)
`undefined`	`let x;`	Declared but not assigned a value
`null`	`let y = null;`	Intentional absence of any object value
`BigInt`	`123n`	For very large integers
`Symbol`	`Symbol('id')`	Unique and immutable identifiers

Non-Primitive (Reference) Data Types

Type	Example	Description
`Object`	`{name: "John"}`	Key-value pairs
`Array`	`[1, 2, 3]`	Ordered elements
`Function`	`function() {}`	Reusable code blocks

Type	Example	Description
`Date`, `RegExp`, etc.	Other built-in objects	

Key Points:

- Stored in the heap, reference stored in the stack.
- Compared by reference.
- Mutable (can be changed even if declared with `const`).

Primitive vs Non-Primitive (Reference) in JavaScript

Feature	Primitive	Non-Primitive (Reference)
Stored as	Value	Reference
Stored in	Stack	Heap (reference is in the stack)
Mutability	Immutable	Mutable
Comparison	Compared by value	Compared by reference
Examples	`string`, `number`, `boolean`, `undefined`, `null`, `bigint`, `symbol`	`object`, `array`, `function`
Memory efficiency	More efficient	Less efficient (can grow large)

> **Info**
>
> Even `null` is considered a primitive, but it returns `"object"` when checked with `typeof`. This is a long-standing bug in JavaScript.

```
typeof null // "object"
```

String Methods in JavaScript

JavaScript String Methods

Method	Description	Example
`.length`	Returns length of string	`'hello'.length // 5`
`.charAt(index)`	Returns character at specific index	`'hello'.charAt(1) // 'e'`

Method	Description	Example
`.at(index)`	Similar to charAt (supports negative index)	`'hello'.at(-1) // 'o'`
`.slice(start, end)`	Extracts part of string	`'hello'.slice(1, 4) // 'ell'`
`.substring(start, end)`	Similar to slice (no negative index)	`'hello'.substring(1, 4) // 'ell'`
`.substr(start, length)`	Extracts substring (deprecated)	`'hello'.substr(1, 3) // 'ell'`
`.toUpperCase()`	Converts to uppercase	`'hello'.toUpperCase() // 'HELLO'`
`.toLowerCase()`	Converts to lowercase	`'HELLO'.toLowerCase() // 'hello'`
`.trim()`	Removes whitespace from both ends	`' hi '.trim() // 'hi'`
`.trimStart()` / `.trimEnd()`	Trims start or end only	`' hi '.trimStart() // 'hi '`
`.includes(sub)`	Checks if string contains substring	`'hello'.includes('ell') // true`
`.startsWith(sub)`	Checks if string starts with given text	`'hello'.startsWith('he') // true`
`.endsWith(sub)`	Checks if string ends with given text	`'hello'.endsWith('lo') // true`
`.indexOf(sub)`	Returns first index of match or -1	`'hello'.indexOf('l') // 2`
`.lastIndexOf(sub)`	Returns last index of match or -1	`'hello'.lastIndexOf('l') // 3`
`.replace(old, new)`	Replaces first match	`'hello'.replace('l', 'L') // 'heLlo'`
`.replaceAll(old, new)`	Replaces all matches	`'lollipop'.replaceAll('l', 'L') // 'LoLipop'`
`.repeat(n)`	Repeats the string n times	`'ha'.repeat(3) // 'hahaha'`
`.split(separator)`	Splits string into array	`'a,b,c'.split(',') // ['a','b','c']`
`.concat(str)`	Joins two strings	`'Hello'.concat(' World') // 'Hello World'`

Method	Description	Example
`.match(regex)`	Matches string against regex	`'abc123'.match(/\\d+/) // ['123']`
`.padStart(length, pad)`	Pads at the beginning	`'5'.padStart(3, '0') // '005'`
`.padEnd(length, pad)`	Pads at the end	`'5'.padEnd(3, '0') // '500'`

`.length` is a property, not a method.

- Properties return a value directly.
- Methods are functions attached to an object — you call them with `()`.

Number Methods & Properties

🔢 JavaScript Number Methods & Properties

Method / Property	Description	Example
`Number()`	Converts a value to a number	`Number('42') // 42`
`parseInt(string)`	Parses and returns an integer	`parseInt('10.5') // 10`
`parseFloat(string)`	Parses and returns a floating-point number	`parseFloat('10.5') // 10.5`
`.toFixed(n)`	Returns string with fixed `n` decimal places	`(3.14159).toFixed(2) // '3.14'`
`.toPrecision(n)`	Formats number to `n` significant digits	`(3.14159).toPrecision(3) // '3.14'`
`.toString(base)`	Converts number to a string (optionally in base)	`(255).toString(16) // 'ff'`
`.valueOf()`	Returns the primitive value of a Number object	`(42).valueOf() // 42`
`isNaN(value)`	Checks if value is NaN	`isNaN('abc') // true`
`Number.isNaN(value)`	Strict check for NaN (doesn't coerce types)	`Number.isNaN(NaN) // true`

Method / Property	Description	Example
`Number.isInteger(value)`	Checks if value is an integer	`Number.isInteger(4) // true`
`Number.isFinite(value)`	Checks if value is finite	`Number.isFinite(100) // true`
`Math.round()`	Rounds to nearest integer	`Math.round(4.6) // 5`
`Math.floor()`	Rounds down	`Math.floor(4.9) // 4`
`Math.ceil()`	Rounds up	`Math.ceil(4.1) // 5`
`Math.trunc()`	Removes decimal part	`Math.trunc(4.9) // 4`

Few Number Constants in JavaScript

Property	Description	Example
`Number.MAX_VALUE`	Largest possible number	`1.7976931348623157e+308`
`Number.MIN_VALUE`	Smallest positive number	`5e-324`
`Number.POSITIVE_INFINITY`	Represents +Infinity	`Number.POSITIVE_INFINITY`
`Number.NEGATIVE_INFINITY`	Represents -Infinity	`Number.NEGATIVE_INFINITY`
`Number.NaN`	Not a Number	`Number.NaN`

JavaScript Array Methods

JavaScript Array Methods

Method	Description	Example
`.length`	Returns number of elements	`[1, 2, 3].length // 3`
`.push(item)`	Adds item to end	`[1, 2].push(3) // [1, 2, 3]`
`.pop()`	Removes last item	`[1, 2, 3].pop() // [1, 2]`
`.unshift(item)`	Adds item to beginning	`[2, 3].unshift(1) // [1, 2, 3]`
`.shift()`	Removes first item	`[1, 2, 3].shift() // [2, 3]`

Method	Description	Example
`.concat(arr)`	Merges arrays	`[1].concat([2, 3]) // [1, 2, 3]`
`.join(separator)`	Converts array to string	`[1, 2].join('-') // '1-2'`
`.slice(start, end)`	Extracts part of array	`[1, 2, 3].slice(1, 3) // [2, 3]`
`.splice(start, deleteCount, ...items)`	Removes/replaces/adds items	`[1, 2, 3].splice(1, 1, 4) // [1, 4, 3]`
`.indexOf(item)`	First index of item	`[1, 2, 3].indexOf(2) // 1`
`.lastIndexOf(item)`	Last index of item	`[1, 2, 2, 3].lastIndexOf(2) // 2`
`.includes(item)`	Checks if item exists	`[1, 2].includes(2) // true`
`.reverse()`	Reverses the array in place	`[1, 2].reverse() // [2, 1]`
`.sort()`	Sorts array (as strings by default)	`[3, 1, 2].sort() // [1, 2, 3]`
`.fill(value, start, end)`	Fills part of array	`[1, 2, 3].fill(0, 1, 2) // [1, 0, 3]`
`.flat(depth)`	Flattens nested arrays	`[1, [2, [3]]].flat(2) // [1, 2, 3]`
`.toString()`	Converts array to comma-separated string	`[1, 2].toString() // '1,2'`
`.find(callback)`	Returns first match	`[1, 2, 3].find(x => x > 1) // 2`
`.findIndex(callback)`	Returns index of first match	`[1, 2, 3].findIndex(x => x > 1) // 1`
`.filter(callback)`	Returns array of matches	`[1, 2, 3].filter(x => x > 1) // [2, 3]`
`.map(callback)`	Transforms array	`[1, 2].map(x => x * 2) // [2, 4]`
`.forEach(callback)`	Iterates over array	`[1, 2].forEach(x => console.log(x))`
`.reduce(callback, initial)`	Reduces to single value	`[1, 2, 3].reduce((a, b) => a + b, 0) // 6`
`.every(callback)`	Checks if all match condition	`[1, 2, 3].every(x => x > 0) // true`

Method	Description	Example
`.some(callback)`	Checks if any match condition	`[1, 2, 3].some(x => x > 2) // true`
`.at(index)`	Returns item at index (supports negatives)	`[1, 2, 3].at(-1) // 3`
`Array.isArray(arr)`	Checks if value is array	`Array.isArray([1, 2]) // true`

A callback is simply a function that you pass into another function as an argument, which is then called (or executed) at a later time.
A callback is like saying, "Hey, I'm giving you this function, call me back when you're done!"

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}
function processUserInput(callback) {
  const name = "Alice";
  callback(name); // Calls the passed-in function (greet) with 'name' as the argument
}
// Passing greet as a callback to processUserInput
processUserInput(greet);  // Output: Hello, Alice!
```

Object Methods

```
```

JavaScript Object Methods

Method	Description	Example
`Object.keys(obj)`	Returns an array of an object's own enumerable property names	`Object.keys({a: 1, b: 2 'b']`
`Object.values(obj)`	Returns an array of an object's own enumerable property values	`Object.values({a: 1, b: 2]`

Method	Description	Example
`Object.entries(obj)`	Returns an array of an object's own enumerable string-keyed property [key, value] pairs	`Object.entries({a: 1, b [['a', 1], ['b', 2]]`
`Object.assign(target, ...sources)`	Copies all enumerable properties from one or more source objects to a target object	`Object.assign({}, {a: 1 // {a: 1, b: 2}`
`Object.freeze(obj)`	Freezes an object (makes it immutable)	`Object.freeze({a: 1})`
`Object.is(obj1, obj2)`	Compares if two values are the same (same value and type)	`Object.is(42, 42) // tr`
`Object.create(proto)`	Creates a new object with the specified prototype object and properties	`Object.create({x: 1}) /`
`Object.hasOwn(obj, prop)`	Checks if an object has a property as its own property	`Object.hasOwn({a: 1}, '`
`Object.prototype.toString()`	Returns a string representation of the object	`({}).toString() // '[ob Object]'`
`Object.prototype.isPrototypeOf(obj)`	Checks if an object is in the prototype chain of another	`Object.prototype.isProt 1}) // true`
`Object.getPrototypeOf(obj)`	Returns the prototype of an object	`Object.getPrototypeOf({ Object.prototype`

Method	Description	Example
`Object.setPrototypeOf(obj, proto)`	Sets the prototype (parent) of an object	`Object.setPrototypeOf({ Array.prototype)`
`Object.defineProperty(obj, prop, descriptor)`	Defines a new property on an object with specific options	`Object.defineProperty({ {value: 1})`
`Object.defineProperties(obj, props)`	Defines multiple properties on an object	`Object.defineProperties {value: 1}})`

some concept impress you

How JavaScript Handles Hoisting Internally

When JavaScript executes a script, it does this in **two phases**:

1️⃣ Creation Phase (Memory Allocation)

- Functions & variables are stored in memory before execution starts.
- Function declarations are fully hoisted (available before execution).
- Variables (`var`) are hoisted but not initialized (set to `undefined`).

2️⃣ Execution Phase

- Code runs **line by line**, using the stored memory references.

🔹 Function Hoisting (Fully Hoisted ✅)

```
sayHello(); // ✅ Works due to hoisting

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

Behind the scenes:

```
// JavaScript engine processes like this internally:
function sayHello() {   // ✅ Function moved to the top (fully hoisted)
    console.log("Hello, World!");
}
```

```
sayHello();
```

1️⃣ `var` Hoisting (Partially Hoisted, `undefined`)

`var` is hoisted but not initialized.

- If accessed before declaration, it returns `undefined` (not an error).

```
console.log(a); // ❌ undefined (not an error)
var a = 5;
console.log(a); // ✅ **Behind the scenes:**
```

```
// JavaScript internally does this:
var a; // ✅ Hoisted to the top (default: undefined)
console.log(a); // ❌ undefined
a = 5; // ✅ Assigned value later
console.log(a); // ✅ 5
```

2️⃣ `let` & `const` Hoisting (Hoisted but in "Temporal Dead Zone")

- Unlike `var`, they are hoisted but not initialized.
- Accessing them before declaration causes an error.

```
console.log(b); // ❌ ReferenceError: Cannot access 'b' before
initialization
let b = 10;
```

```
// JavaScript internally does this:
let b; // ✅ Hoisted but NOT initialized (stays in Temporal Dead Zone)
console.log(b); // ❌ ReferenceError
b = 10; // ✅ Now it's initialized
```

🔹 Hoisting Summary Table

Feature	`var`	`let`	`const`	`function`
Hoisted?	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Initialized?	× `undefined`	× No (TDZ)	× No (TDZ)	✅ Fully hoisted
Can access before declaration?	✅ Yes (`undefined`)	× No (TDZ error)	× No (TDZ error)	✅ Yes

🔹 Key Takeaways

✅ Function declarations are fully hoisted.

✅ `var` is hoisted, but set to `undefined`.

✅ `let` and `const` are hoisted, but in Temporal Dead Zone (TDZ), causing errors if accessed too early.

✅ Hoisting helps JavaScript run scripts even if functions or variables are declared later.

Would you like to see hoisting in `class` or arrow functions? 🚀

🚀 Temporal Dead Zone (TDZ) in JavaScript

Temporal Dead Zone (TDZ) ***is the period between the** start of variable hoisting and its initialization** .

If you try to access the variable during this period, JavaScript throws a **ReferenceError**.

TDZ only affects `let` and `const` (not var)

Variables declared with `let` or `const` are hoisted but not initialized until the execution reaches the declaration.

```
console.log(a); // ❌ ReferenceError: Cannot access 'a' before
initialization
let a = 10;
console.log(a); // ✅ 10 (Accessing after declaration is fine)
//Behind the scenes:
// JavaScript internally does this:
let a;  // ✅ Hoisted but uninitialized (TDZ starts)
console.log(a); // ❌ ReferenceError (TDZ still active)
a = 10; // ✅ Now initialized, TDZ ends
console.log(a); // ✅ 10
```

🔹 Why Does TDZ Exist?

TDZ prevents using uninitialized variables and improves code safety by enforcing proper variable usage

Syntax of Arrow Functions

```
// Regular function
function add(a, b) {
    return a + b;
}
```

```
// Arrow function (shorter)
const add = (a, b) => a + b;

console.log(add(5, 3)); // ✅ 8
```

- No need for `function` keyword.
- `return` is implicit if the function body contains a single expression.
- Parentheses `( )` are optional if there's only one parameter.

🚀 Understanding Implicit `return` in Arrow Functions

In arrow functions, if the function body contains only a single expression, you can omit the `{}` (curly braces) and the `return` keyword—this is called implicit return.

✅ Example 1: Explicit vs. Implicit Return

🔹 Regular Function (Explicit `return`)

```
const add = (a, b) => {
    return a + b; // ✅ Explicit return (you must write `return`)
};
console.log(add(3, 4)); // ✅ 7
```

Arrow Function (Implicit `return`)

```
const add = (a, b) => a + b; // ✅ No `{}`, no `return` (implicit return)
console.log(add(3, 4)); // ✅ 7
```

✅ Example 2: Implicit Return with String

```
const greet = name => `Hello, ${name}!`; // ✅ No `{}`, no `return`
console.log(greet("Samay")); // ✅ "Hello, Samay!"
```

✅ Example 3: Implicit Return with an Object (❗ Extra `()` Needed)

If you return an object literal, you must wrap it in `()`. Otherwise, `{}` will be treated as a function block.

```
// ❌ Incorrect (doesn't work)
const getUser = (name, age) => { name: name, age: age };

// ✅ Correct (Wrap in `()`)
```

```
const getUser = (name, age) => ({ name: name, age: age });

console.log(getUser("Samay", 30));
// ✅ { name: "Samay", age: 30 }
📌 Why?
{} alone is treated as a function block, not an object.
Wrapping it in () makes JavaScript understand it's an object.
```

✅ When Do You Need `{}` and `return`?

```
const multiply = (a, b) => {
    console.log(`Multiplying ${a} and ${b}`); // Extra statement
    return a * b; // ✅ Must use `return`
};
console.log(multiply(3, 4)); // ✅ 12
```

🔹 What is a Promise in JavaScript?

A **Promise** in JavaScript is an object that represents the eventual **completion** (or **failure**) of an **asynchronous** operation. Instead of using **callbacks**, Promises help us write cleaner and more readable asynchronous code.

📌 Think of a Promise like a real-life promise:

- You order food online (a request is made).
- The restaurant promises to deliver it (a promise is created).
- The delivery can be successful ✅ (fulfilled) or fail ✕ (rejected).

🔹 Basic Syntax of a Promise

A Promise is created using the `new Promise()` constructor, which takes a function (called the **executor function**) with **two parameters**:

- **`resolve`** → Call this when the operation is successful.
- **`reject`** → Call this when the operation fails.

```
const myPromise = new Promise((resolve, reject) => {
    let success = true; // Change this to false to test rejection

    if (success) {
        resolve("✅ Operation successful!");
    } else {
        reject("❌ Operation failed!");
    }
});
```

```
console.log(myPromise); // Promise { <pending> }
 // why ? { <pending> }
// 1️⃣ .then() → Executes when the promise is resolved
 myPromise
    .then(result => {
        console.log(result); // ✅ "Operation successful!"
    });

//2️⃣ .catch() → Executes when the promise is rejected

myPromise
    .catch(error => {
        console.log(error); // ❌ "Operation failed!"
    });
3️⃣ .finally() → Always executes (whether resolved or rejected)

myPromise
    .finally(() => {
        console.log("🎯 Promise completed!");
    });
```

🔹 `Promise.all()`, `Promise.race()`, `Promise.allSettled()`, `Promise.any()`

1️⃣ `Promise.all()` → Waits for all promises to resolve or rejects if one fails

```
const p1 = new Promise(res => setTimeout(() => res("🚀 P1 Done!"), 1000));
const p2 = new Promise(res => setTimeout(() => res("🎯 P2 Done!"), 2000));
const p3 = new Promise((_, rej) => setTimeout(() => rej("❌ P3 Failed!"),
1500));

Promise.all([p1, p2, p3])
    .then(results => console.log(results))
    .catch(error => console.log("Error:", error)); // ❌ Stops if one
fails
```

2️⃣ `Promise.race()` → Returns the first promise that settles (resolved/rejected)

```
Promise.race([p1, p2, p3])
    .then(result => console.log("First done:", result))
    .catch(error => console.log("First failed:", error));
```

3️⃣ `Promise.allSettled()` → Returns results for all promises (whether resolved or rejected)

```
Promise.allSettled([p1, p2, p3])
    .then(results => console.log("All Settled:", results));
```

✅ It does not fail if one promise fails.

✅ It returns an array of objects, each containing:

- **`status`** → `"fulfilled"` if resolved, `"rejected"` if failed
- **`value`** → The resolved value (if fulfilled)
- **`reason`** → The error message (if rejected)

`Promise.allSettled()` is useful when you want to handle both success and failure cases gracefully.

4️⃣ `Promise.any()` → Resolves when the first successful promise resolves

```
Promise.any([p1, p2, p3])
    .then(result => console.log("First Success:", result))
    .catch(error => console.log("All failed:", error));
```

****Key Differences Between `Promise.all()`, `Promise.allSettled()`, and `Promise.any()`

Method	Resolves When	Rejects When	Returns
`Promise.all()`	All promises succeed ✅	If one fails ×	Array of values 📦
`Promise.allSettled()`	All promises settle (resolve or reject)	Never rejects 🚀	Array of `{status, value/reason}`
`Promise.any()`	First fulfilled promise ✅	If all fail ×	First successful value 📦

JavaScript Memory Management & Garbage Collection (Deep Dive)

JavaScript automatically manages memory allocation and cleanup using garbage collection. But how does it work?

🔹 1. Memory Lifecycle in JavaScript

Every variable or function you declare goes through three stages:

1. Allocation → Memory is allocated when a variable is declared or an object is created.
2. Usage → JavaScript uses the allocated memory during execution.

3. **Release (Garbage Collection)** → When a variable is no longer needed, memory is reclaimed.

```
function demo() {
  let name = "Samay"; // Memory allocated
  console.log(name);  // Memory is in use
} // Memory released after function execution (if no references exist)
demo();
```

After `demo()` runs, `"Samay"` is no longer needed, and JavaScript **automatically clears the memory**.

🔹 2. Stack vs Heap Memory

Memory Type	Used For	Lifecycle
Stack	Stores **primitives** (numbers, strings, booleans) and function call information	Cleared **automatically** when a function completes
Heap	Stores **objects, arrays, functions**	Cleared **only if no references exist** (garbage collection)

```
function stackExample() {
  let age = 30; // Stored in Stack (primitive)
  let person = { name: "Samay", age: 30 }; // Stored in Heap (object)
}
stackExample(); // `age` is removed from Stack, but `person` is in Heap
until garbage collected
```

✔️ `age` is automatically removed when `stackExample()` completes.

✖ `person` remains in memory **unless there are no references** to it.

🔹 3. How JavaScript Handles Garbage Collection

JavaScript's **Garbage Collector (GC)** follows an algorithm called **"Mark-and-Sweep"**, which works like this:

1. **Mark Phase** – The GC marks all objects that are still **reachable** (in use).
2. **Sweep Phase** – It removes all objects that are **unreachable** (not referenced anywhere).

```
function createUser() {
  let user = { name: "Samay" }; // user is stored in Heap
  return user;
}

let newUser = createUser(); // `user` is still reachable via `newUser`
newUser = null; // Now, `user` becomes unreachable and is garbage
collected
```

× Once `newUser = null`, the object is **marked for garbage collection**.

✔️ As long as `newUser` references the object, it stays in memory.

🔹 4. What If a Variable Is Returned from a Function?

If a function returns a reference to an object, **that object will not be garbage collected** until there are no references left.

```
function createPerson() {
  let person = { name: "Samay" }; // Allocated in Heap
  return person;
}
let user = createPerson(); // Now user points to person
```

✔️ Even after `createPerson()` finishes execution, the object is **still in memory** because `user` holds a reference to it.

🔹 5. How Memory Leaks Happen? (Monitor Them!)

A **memory leak** happens when unused objects are **never garbage collected**, leading to high memory usage.

🔥 Common Causes of Memory Leaks:

Cause	Example
Global Variables	Declaring variables without `let`, `const`, or `var` (`x = "leak";`)
Uncleared Timers & Intervals	`setInterval(() => console.log("Hello"), 1000);` (not cleared)
Event Listeners Not Removed	`element.addEventListener("click", () => {...})` (not removed)
Closures Holding References	Functions holding objects in memory even when not needed

```
let obj = {};
setInterval(() => {
  obj.leak = "This will never be garbage collected";
}, 1000); // This keeps adding data to `obj` forever
✅ Fix: Use clearInterval() when not needed.
```

Why in return Method not required declare

```
function createThing() {
  let secret = 42;
  return {
    getSecret: function () { // here get getSecret is not declared
      return secret;
    }
  };
}

// if i wrote like this

function createThing() {
  let secret = 42;
  getSecret: () => secret,              // ❌ Invalid syntax
  return {
    getSecret,
    setSecret
  };
}
// correct version
function createThing() {
  let secret = 42;

  const getSecret = () => secret;
  return {
    getSecret
      };
}
```

`secret` is not exposed directly — it's **encapsulated**.

1. method written in correct way because

Because you're **not declaring a variable** — you're directly defining a method as a key-value pair **inside an object literal**.

2. Why Method written wrong way

This is not a variable declaration. JavaScript sees `getSecret:` as a label (used with `break` / `continue` in loops), and the `() => secret` is just a dangling expression that doesn't get stored or executed.

- `getSecret: () => secret` is object-literal syntax — not valid outside of an object.
- You must declare the functions first using `const` or `function`, then return them.

What is a label in JavaScript?

A label is a name you can give to a block of code (like a loop), so you can use `break` or `continue` to control it from outside the loop.

```
outerLoop: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (j === 1) {
      break outerLoop; // breaks the outer loop, not the inner one
    }
    console.log(`i=${i}, j=${j}`);
  }
}
```

Array Accumulator

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let sum = arr.reduce((acc, item) => acc + item, 0);
console.log(sum); // 45
```

Linked list

```
function createNode(data, next = null) {
  return {
    data,
    next
  };
}
function createLinkedList() {
  let head = null;
  return {
    insertAtHead(data) {
      const newNode = createNode(data, head);
      head = newNode;
    },
```

What is a *shadowed variable*?

A shadowed variable happens when:

> *A variable declared **inside a function or block** has the **same name** as a variable from an **outer scope** (like a parameter). The inner one "shadows" or **hides** the outer one.*

```
function node(data, next = null) {
  let data = data; // ❌ shadowing
  let next = next; // ❌ shadowing
}
//JUST RETURN PARAMETER
function createNode(data, next = null) {
  return {
    data,
    next
  };
}
```

Class

Classes are in fact "special functions", and just as you can define function expressions and function declarations, a class can be defined in two ways: a class expression or a class declaration.

```
// Declaration
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

// Expression; the class is anonymous but assigned to a variable
const Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

// Expression; the class has its own name
const Rectangle = class Rectangle2 {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Objects