# C.1 HTML Interview Questions (35 Questions)

## Basic Level

### 1. What is HTML and what does it stand for?

➢ HTML stands for: HyperText Markup Language

➢ It is the standard markup language used to structure content on the web.

➢ HTML:

• Defines structure (headings, paragraphs, images, links)
• Is not a programming language
• Works with CSS (styling) and JavaScript (behavior)

### 2. What is the purpose of <!DOCTYPE html>?

➢ It tells the browser: "This document is HTML5."

➢ Purpose:

• Prevents browser from entering quirks mode
• Ensures standards-compliant rendering
• Defines the HTML version

### 3. What is the difference between HTML elements and HTML tags?

➢ HTML Tag:

The markup inside angle brackets. Example:  

➢ HTML Element:

The complete structure including opening tag + content + closing tag.
Example: Hello

### 4. What is the difference between  and ?

| Feature |  |  |
|---|---|---|
| Type | Block-level | Inline |
| Width | Full width | Content width |
| Use Case | Layout grouping | Styling small text parts |

### 5. What are semantic HTML elements? Give 5 examples.

Semantic elements clearly describe their meaning.
Examples:

• <header>

• <footer>
• <article>
• <section>
• <nav>
They improve SEO, accessibility, and readability.

## 6. What is the difference between <section> and <article>?

➢ <section> groups related content.
➢ <article> represents independent, self-contained content like blog posts.

## 7. When should you use <header>, <main>, and <footer>?

➢ <header>: Introductory content or navigation.
➢ <main>: Main content of the page (only one per page).
➢ <footer>: Closing content like copyright or links.

## 8. What is the purpose of the <nav> element?

➢ Used to define navigation links section.
➢ Improves accessibility and SEO.

## 9. What is the difference between block-level and inline elements?

➢ Block-level: Takes full width, starts on new line (e.g., , <p>).
➢ Inline: Takes only needed width, does not start new line (e.g., , <a>).

## 10. What are void elements or self-closing tags in HTML? Give examples.

➢ Elements that do not have closing tags.
Examples:
• <img>
• <br>
• <hr>
• <input>
• <meta>

# Intermediate Level

## 11. What is the difference between id and class attributes?

➢ id: Unique, used once per page.
➢ class: Can be used multiple times.
➢ id has higher CSS specificity.

## 12. What are data attributes (data-*) and when would you use them?

➢ Custom attributes to store extra information.
Example: data-id="101"
➢ Used in JavaScript for dynamic behavior.

## 13. What is the purpose of the alt attribute in images?

➢ Provides alternative text if image fails to load.
➢ Important for accessibility and screen readers.

## 14. What is the difference between  and <b>,  and <i>?

➢  and  have semantic meaning.
➢ <b> and <i> are purely visual.

## 15. What are meta tags and why are they important?

➢ Provide metadata about the page.
➢ Important for SEO, charset, and responsiveness.

## 16. What is the viewport meta tag and why is it crucial for responsive design?

➢ Controls layout on mobile browsers.
Example:
<meta name="viewport" content="width=device-width, initial-scale=1.0">

## 17. What is the difference between <link> and <script> tags?

➢ <link>: Used to link external resources like CSS.
➢ <script>: Used to include JavaScript.

## 18. What are the different input types in HTML5? List at least 8.

Examples:
• text
• password
• email
• number
• date
• file
• checkbox
• radio

## 19. What is the purpose of the name attribute in form inputs?

➢ Identifies form data when submitted.
➢ Required for backend processing.

➢ The name attribute defines the key used to send form data to the server. When a form is submitted, the input's value is sent as a key-value pair where the key comes from the name attribute. Without name, the input data is not submitted.

## 20. What is the difference between GET and POST methods in forms?

➢ GET: Data sent in URL, less secure, limited size.
➢ POST: Data sent in body, more secure, no size limit.

## 21. What is the purpose of <label> and how should it be associated with inputs?

➢The <label> element defines a **text description for an input field**.

➢ Improves accessibility.
➢ Use for attribute matching input id.

## 22. What are required, pattern, min, max attributes used for?

➢ Used for form validation.
required → field must be filled.
pattern → regular expression validation.
min/max → numeric limits.

## 23. What is the purpose of placeholder vs value attributes?

➢ placeholder: Hint text.

- Shows a **hint text** inside the input field.
- Disappears when the user starts typing.
- Does NOT get submitted with the form.

➢ value: Default actual value.

- Sets a default value inside the input.
- This value gets submitted with the form (unless changed).
- Stays visible even after typing unless replaced.

## 24. What is the difference between <button>, <input type='button'>, and <input type='submit'>?

➢ <button>: Flexible, can contain HTML.

- Can contain **HTML inside** (icons, images, spans, etc.)
- More flexible
- Default type inside a form = "submit"

➢ input type='button': Basic clickable button.

- Displays a clickable button
- Requires JavaScript to do anything
- Cannot contain HTML (only text via value)

➢ input type='submit': Submits form.

- Submits the form automatically
- Sends form data to server
- Default behavior inside <form>

## 25. What is the target attribute in anchor tags? What is target='_blank' and its security concern?

➢ target defines where to open link.

➢ _blank opens in new tab.

➢ Security issue: tabnabbing. ( **Tabnabbing** is a phishing attack where a malicious website changes the content of your original tab after you open a new one)
Solution: rel="noopener noreferrer".

# Advanced Level

## 26. What is the purpose of ARIA attributes in HTML?

➢ Accessible Rich Internet Applications

➢ Improve accessibility.
➢ Help screen readers understand UI components.

## 27. How do you make HTML forms accessible?

➢ Use labels.
➢ Provide alt text.
➢ Use semantic HTML.
➢ Proper error messages.

## 28. What is the difference between <script>, <script defer>, and <script async>?

➢ Normal: Blocks HTML parsing.
➢ defer: Executes after HTML parsing.
➢ async: Loads independently and executes immediately.

## 29. What are HTML entities and when do you need them? Give examples.

➢ **HTML entities** are special codes used to display:

- Reserved characters
- Special symbols
- Characters that are not easily typed on a keyboard

➢ They start with "&" and end with " ; "

Examples:
< → <
> → >
& → &

## 30. What is the purpose of <picture> element and how is it different from <img>?

➢ <picture> allows multiple sources for responsive images.
➢ <img> displays a single image.

➢ The <picture> element is used to provide **multiple image sources** and let the browser choose the best one based on:

- Screen size
- Device resolution
- Image format support
- Media conditions

➢ It gives you **more control** than <img>.

➢ **Basic Example**

```
<picture>
  <source media="(max-width: 600px)" srcset="mobile.jpg">
  <source media="(max-width: 1024px)" srcset="tablet.jpg">
  <img src="desktop.jpg" alt="Responsive image">
</picture>
```

Now:

- Mobile screen → loads mobile.jpg
- Tablet → loads tablet.jpg
- Desktop → loads desktop.jpg

## 31. What is the srcset attribute in images?

➢ Provides multiple image sizes for responsive design.

➢ The srcset attribute is used inside the <img> tag to provide **multiple image versions** for different screen sizes or resolutions.

➢ It allows the browser to automatically choose the most appropriate image.

➢ This improves:

- Performance
- Page load speed
- Mobile optimization
- Retina/High-DPI support

## 32. What is the difference between <audio> and <video> elements?

➢ <audio>: Embeds sound.

➢ <video>: Embeds video content.

## 33. How do you embed iframes and what are the security considerations?

➢ An <iframe> (Inline Frame) is used to **embed another webpage inside your current webpage**.

➢ It creates a window within a window.

➢ Use <iframe src="URL"></iframe>

➢ Security: sandbox attribute, avoid untrusted sources.

## 34. What is shadow DOM and how does it relate to web components?

➢ **Shadow DOM** is a browser feature that allows you to create a **separate, encapsulated DOM tree** inside an element.

➢ That means:

- The HTML inside it is **isolated**
- The CSS inside it does **not leak out**
- Outside CSS does **not affect it**
- JavaScript scope can also be isolated

➢ Think of it as a **mini private DOM inside an element**.

## 35. What is the purpose of contenteditable attribute?

➢ Makes an element editable by the user directly in the browser.

➢ Normally, users can only type inside form fields like <input> or <textarea>.
But when you add contenteditable="true" to an element, it turns that element into an editable area.

# C.2 CSS Interview Questions (40 Questions)

## Basic Level

### 1. What is CSS and what does it stand for?

➢ CSS stands for Cascading Style Sheets.

➢ It is used to style and design HTML elements.

➢ Controls layout, colors, fonts, spacing, and responsiveness.

Example:

p { color: blue; font-size: 16px; }

### 2. What are the different ways to apply CSS to HTML?

➢ Three ways:

• Inline CSS
• Internal CSS
• External CSS

### 3. What is the difference between inline, internal, and external CSS?

➢ Inline: style attribute inside element.

➢ Internal: <style> tag inside <head>.

➢ External: Separate .css file using <link>.

External CSS is best practice for maintainability.

### 4. What are CSS selectors? List different types.

➢ CSS selectors select HTML elements.

Types:

• Element (p {})
• Class (.box {})
• ID (#header {})
• Attribute ([type="text"])
• Descendant (div p)
• Universal (*)

### 5. What is the universal selector (*)?

➢ Universal selector (*) selects all elements.

Example:

* { margin: 0; padding: 0; }

## 6. What are class selectors and id selectors?

➢ Class selector:  .classname (reusable).

➢ ID selector: #idname (unique).

## 7. What is the difference between class and id in CSS?

➢ id is unique and has higher specificity.

➢ class can be reused multiple times.

## 8. What are pseudo-classes? Give 5 examples.

➢ Pseudo-classes define states.

Examples:

- :hover
- :active
- :focus
- :first-child
- :nth-child()

## 9. What are pseudo-elements? Give examples.

➢ Pseudo-elements style parts of elements.

Examples:

- ::before
- ::after
- ::first-letter
- ::first-line

## 10. What is the difference between :hover and :active?

➢ :hover → when mouse is over element.

➢ :active → when element is being clicked.

**Intermediate Level - Box Model & Layout**

## 11. What is the CSS box model? Explain all components.

➢The **CSS Box Model** describes how every HTML element is represented as a rectangular box and how its size is calculated.

➢ CSS Box Model consists of:

- Content

• Padding
• Border
• Margin
Total width = content + padding + border + margin.

## 12. What is the difference between margin and padding?

➢ Margin: Space outside border.
➢ Padding: Space between border and content.

## 13. What is margin collapse and when does it occur?

➢ **Margin collapse** happens when the vertical margins of two block-level elements overlap, and instead of adding together, the browser uses only the larger margin.

- It only affects **vertical margins** (margin-top and margin-bottom).
- It does NOT affect horizontal margins.

## 14. What is box-sizing: border-box and why is it useful?

➢ box-sizing: border-box includes padding and border in total width/height.
Example:
box-sizing: border-box;

➢ box-sizing: border-box makes the element's width and height include padding and border. Unlike the default content-box, it prevents layout issues by keeping the total size fixed. It simplifies layout calculations and is commonly used in modern CSS development.

## 15. What is the default value of box-sizing?

➢ Default value is content-box.

## 16. What is the difference between content-box and border-box?

➢ content-box: width excludes padding/border.
➢ border-box: width includes padding/border.

## 17. What are the different display property values? Explain each.

➢ display values:
block

- Takes full available width.
- Starts on a new line.

- Can set width & height.

Inline

- Takes only the required content width.
- Does NOT start on a new line.
- Cannot set width & height (ignored).
- Margin/padding works only horizontally properly.

inline-block

- Behaves like inline (does not break line).
- But allows width & height like block.

None

- Removes element completely from layout.
- Takes no space.
- Not visible.
- Not accessible by screen readers.

Flex

- Turns element into a flex container.
- Enables flexible layout system.
- Children become flex items.

Grid

- Turns element into a grid container.
- Allows 2D layout (rows + columns).

## 18. What is the difference between display: none and visibility: hidden?

➢ display:none removes element completely.
➢ visibility:hidden hides but keeps space.

## 19. What is display: inline-block used for?

➢ inline-block allows width/height while remaining inline.

## 20. What is the difference between block and inline-block elements?

➢ block: full width.
➢ inline-block: content width but allows dimensions(height and width).

# Intermediate Level - Positioning & Specificity

## 21. What is CSS specificity and how is it calculated?

➢ **CSS specificity** determines which CSS rule is applied when multiple rules target the same element.

When two or more selectors match the same element:

➢ The rule with **higher specificity wins**.

➢ Specificity determines which rule applies.
Order: Inline > ID > Class > Element.

## 22. What is the specificity value of inline styles, ids, classes, and elements?

➢ Inline: 1000
➢ ID: 100
➢ Class: 10
➢ Element: 1

## 23. What is the cascade in CSS?

➢ Cascade defines final style based on importance, specificity, and source order.

➢ The **cascade** is the process the browser uses to determine which CSS rule applies when multiple rules target the same element.

➢ CSS stands for:

**Cascading Style Sheets**

➢ “Cascading” means styles flow and override each other based on specific rules.

## 24. What does !important do and why should it be avoided?

➢ !important overrides normal rules.

➢ Avoid because it breaks maintainability.

## 25. What are the different position values in CSS?

➢ position values:

• static
• relative
• absolute
• fixed
• sticky

## 26. What is the difference between position: relative and position: absolute?

➢ relative: positioned relative to itself.

➢ absolute: positioned relative to nearest positioned ancestor.

## 27. What is position: fixed and when is it used?

➢ fixed: positioned relative to viewport.

## 28. What is position: sticky and how does it work?

➢ sticky: acts relative until scroll threshold, then fixed.

## 29. What is z-index and when does it work?

➢ z-index controls stacking order.
Works on positioned elements.

## 30. What is a stacking context in CSS?

➢ Stacking context is a 3D stacking order created by positioned elements with z-index.

**Advanced Level - Units & Responsive**

## 31. What are the different CSS units? Explain px, %, em, rem, vh, vw.

➢ CSS units:
px: pixels

- Fixed size.

- Does NOT depend on parent or viewport.
- Most commonly used unit.

%: percentage (Relative to the **parent element**.)
em: relative to parent font size
rem: relative to root
vh: viewport height
vw: viewport width

## 32. What is the difference between em and rem?

➢ em is relative to parent font-size.
➢ rem is relative to root font-size.

## 33. When should you use relative units vs absolute units?

➢ Relative units for responsive design.
➢ Absolute units for fixed layouts.

## 34. What are viewport units (vh, vw, vmin, vmax)?

➢ vh: 1% viewport height
➢ vw: 1% viewport width
➢ vmin: smaller of vh/vw
➢ vmax: larger of vh/vw

## 35. What are media queries and how do they work?

➢ Media queries apply styles based on conditions.

➢Help us to provide responsive designs.
Example:
@media (max-width:600px){ body{background:red;} }

## 36. What is mobile-first approach in responsive design?

➢ Mobile-first approach designs for small screens first using min-width.

➢ You design and write CSS for small screens first, then progressively enhance the layout for larger screens using min-width media queries.

## 37. What is the difference between min-width and max-width in media queries?

➢ min-width applies styles above value.
➢ max-width applies styles below value.

## 38. What are CSS variables (custom properties) and how do you use them?

➢ **CSS variables**, also called **custom properties**, are reusable values defined in CSS that can be used throughout a stylesheet.

➢ They help:

- Avoid repetition
- Improve maintainability
- Make themes easy
- Support dynamic styling

➢ CSS variables use --name.
Example:
:root { --main-color: blue; }
color: var(--main-color);

## 39. What is the calc() function in CSS?

➢ calc() performs calculations.
Example:
width: calc(100% - 20px);

## 40. What are CSS preprocessors (Sass, Less)? What advantages do they offer?

➢ CSS preprocessors like Sass and Less provide variables, nesting, mixins, and functions for better maintainability.

# C.3 – C.6 JavaScript Interview Questions

# C.3 JavaScript Introduction & V8 Engine (10 Questions)

## 1. What is JavaScript and what are its key features?

➢ JavaScript is a high-level, dynamic programming language used to build interactive web applications.
➢ Key Features:
• Dynamic typing
• First-class functions
• Prototype-based inheritance
• Event-driven & asynchronous
• Runs in Browser & Node.js

## 2. Difference between interpreted and compiled languages. Where does JS fit?

➢ Compiled: Entire code translated before execution (e.g., C, C++).
➢ Interpreted: Executed line-by-line (e.g., Python).
➢ JavaScript uses Just-In-Time (JIT) compilation (hybrid approach).

## 3. What is the V8 engine and which platforms use it?

➢ The **V8 engine** is an open-source JavaScript engine developed by Google.

➢ V8 is Google's JavaScript engine.
➢ Used in Chrome, Node.js, and Chromium-based browsers.

## 4. What is Just-In-Time (JIT) compilation?

➢ **Just-In-Time (JIT) compilation** is a technique where code is compiled into machine code **at runtime**, instead of before execution.

It combines:

- Interpretation
- Compilation

To improve performance.

## 5. Role of Call Stack in JavaScript execution.

➢ Call Stack manages function execution.
➢ Functions are pushed when called and popped(deleted) when completed.
➢ Follows LIFO principle.

## 6. Describe V8 architecture (Parser, AST, Ignition, TurboFan).

➢ Parser converts code into AST.

The **Parser** reads JavaScript source code and checks:

- Syntax correctness
- Structure
- Language rules

➢ AST is a tree-like structure that represents your code logically.
➢ Ignition generates bytecode.

What it does:

1. Converts AST → Bytecode
2. Executes bytecode

Bytecode is:

- Lower-level representation of code
- Faster than interpreting source directly

At this stage:
Your code runs — but not fully optimized.

➢ TurboFan optimizes frequently executed code into machine code.

It:

- Detects “hot” code (frequently executed)
- Converts bytecode → optimized machine code
- Applies performance optimizations

## 7. Difference between Memory Heap and Call Stack?

➢ Memory Heap stores objects & data.

It is:

- Larger memory space
- Unstructured
- Managed by Garbage Collector

➢ Call Stack stores execution contexts.

The **Call Stack** is where JavaScript keeps track of:

- Function calls
- Execution contexts
- Order of execution

## 8. How does V8 optimize JavaScript?

➢ JIT Compilation

V8 uses a hybrid approach:

1. Interprets code first (Ignition).
2. Detects frequently executed functions ("hot code").
3. Compiles hot code into optimized machine code using TurboFan.

So:

- Cold code → interpreted
- Hot code → optimized machine code

This makes repeated operations much faster.

➢ Hidden Classes

V8 creates **hidden classes** internally to:

- Track object structure
- Optimize property access

If multiple objects have the same structure:

let u1 = { name: "A", age: 20 };

let u2 = { name: "B", age: 22 };

V8 reuses the same hidden class → faster property lookup.

➢ Inline Caching

When accessing object properties repeatedly:

user.name;

user.name;

user.name;

V8 remembers:

- Where the property is stored
- Skips repeated lookup

This speeds up property access significantly.

➢ TurboFan optimization

When a function runs many times:

function add(a, b) {

return a + b;

}

V8 assumes types are consistent (e.g., numbers).

It generates specialized machine code for that type.

## 9. Explain compilation phases in V8.

➢ Parse → AST
➢ Compile → Bytecode
➢ Execute → Optimized machine code

## 10. Role of Hidden Classes in V8?

➢ Hidden classes optimize object property access performance.

JavaScript objects are dynamic.

You can do this:

let user = {};

user.name = "Snehasish";

user.age = 25;

Unlike languages like Java or C++, JS objects:

- Don’t have fixed structure
- Can change shape at runtime
- Properties can be added or removed anytime

That makes optimization difficult.

---

🔹 **What Are Hidden Classes?**

Hidden Classes are **internal structures used by V8** to optimize object property access.

They are NOT visible in JavaScript.

They:

- Track object structure (property layout)
- Allow fast property lookup
- Help V8 generate optimized machine code

# C.4 Variables (var, let, const) (15 Questions)

## 1. Different ways to declare variables?

➢ var
➢ let
➢ const

## 2. Difference between var, let, const?

➢ var: Function-scoped, hoisted, allows redeclaration.
➢ let: Block-scoped, no redeclaration, hoisted but remain in TDZ.
➢ const: Block-scoped, cannot reassign, hoisted but remain in TDZ.

## 3. Can you reassign const?

➢ No. But objects/arrays declared with const can be modified internally.

## 4. Scope of var, let, const?

➢ var → Function scope
➢ let & const → Block scope

## 5. Declare without var/let/const?

➢ Creates global variable (bad practice).

## 6. Temporal Dead Zone (TDZ)?

➢ Period between hoisting and initialization for let/const.
➢ Accessing before initialization throws ReferenceError.

## 7. Block scope vs Function scope?

➢ Block scope limited to {}.
➢ Function scope limited to function.

## 8. Why var not recommended?

➢ Causes scope leakage, variable pollution
➢ Allows redeclaration.
➢ Leads to unpredictable behavior.

## 9. Can const be declared without initialization?

➢ No. Must initialize at declaration.

## 10. Difference let vs const in reassignment?

➢ let → Reassign allowed.
➢ const → Reassign not allowed.

## 11. Output (var example)

```
var x = 10;
if(true){
 var x = 20;
 console.log(x);
}
console.log(x);
```

Output:
20
20

## 12. Output (let example)

```
let a = 10;
if(true){
 let a = 20;
 console.log(a);
}
console.log(a);
```

Output:
20
10

## 13. Output (const array)

```
const arr=[1,2,3];
arr.push(4);
```

```
console.log(arr);
arr=[5,6,7];
```

Output:
[1,2,3,4]
TypeError

### 14. Explain variable shadowing.

➢ Inner variable with same name hides outer variable within block.

➢ **Variable shadowing** happens when a variable declared inside an inner scope has the same name as a variable in an outer scope.

```
let x = 10;

{

 let x = 20;

 console.log(x);

}

console.log(x);
```

### 15. What are the best practices for using var, let, and const in production code?

➢ Use const by default.
➢ Use let when needed.
➢ Avoid var.

## C.5 Hoisting (20 Questions)

### 1. What is hoisting in javascript?

➢ JS moves declarations to top during creation phase.

➢Accessing function/variaables before its declaration

### 2. Which declarations are hoisted in js?

➢ var
➢ let/const (without initialization)
➢ Function declarations

## 3. Are variables declared with let and const hoisted?

➢ Yes, but remain in TDZ.

## 4. What value is assigned to hoisted var variables before initialization?

➢ undefined

## 5. What is the difference between hoisting of var and let/const?

➢ var → undefined
➢ let/const → TDZ

# Intermediate Level

## 6. Explain the creation phase and execution phase in JavaScript.

**Creation Phase (Memory Phase)**
Before any line executes:

- Memory is allocated
- Variables and functions are registered
- Scope chain is set up
- this is determined

What happens internally:

| Declaration | During Creation Phase |
|---|---|
| var | Allocated + initialized as undefined |
| let / const | Allocated but NOT initialized (TDZ) |
| Function declarations | Fully stored in memory |

**Execution Phase**
Now JavaScript runs code line by line.

- Assignments happen
- Functions execute
- Expressions evaluate

## 7. What is the Temporal Dead Zone in context of hoisting?

➢The **Temporal Dead Zone** is the time between:

- When a let or const variable is hoisted
- And when it is initialized

➢During this time: You cannot access the variable.

## 8. How are function declarations hoisted differently from function expressions?

Function Declaration:

Because:

- Fully hoisted
- Definition stored during creation phase

Function Expression

```
test();

var test = function() {
  console.log("Hello");
};
// TypeError: test is not a function
```

Why?

Only this is hoisted:

```
var test = undefined;
```

The function assignment happens later.

➢ Function declarations are fully hoisted with their definitions, while function expressions are hoisted as variables without initialization.

## 9. Are arrow functions hoisted?

➢ Arrow functions are not fully hoisted. They behave like function expressions and cannot be called before declaration.

## 10. What is the hoisting behavior of class declarations?

Classes behave like let.

Example:

const obj = new Person();

class Person {}

Output:

ReferenceError

Why?

- Class declarations are hoisted
- But remain in TDZ
- Cannot be accessed before declaration

## 11. Output example (var)

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

Output:
undefined
5

## 12. Output example (let)

```
console.log(b);
let b=10;
```

Output:
ReferenceError

## 13. Output-based: What will be the output?

```
test();
function test() {
console.log("Function Declaration");
}
```

Output:
Function Declaration

## 14. Output-based: What will be the output?

```
test();
var test = function() {
console.log("Function Expression");
}
```

Output:
TypeError: test is not a function

## 15. Output-based: What will be the output?

```
var x = 10;
function foo() {
console.log(x);
var x = 20;
console.log(x);
```

```
}
foo();
```

Output:
TypeError: test is not a function

## 16. Output-based: What will be the output?

```
function test() {
console.log(a);
console.log(b);
var a = 10;
let b = 20;
}
test();
```

Output:

Undefined

Reference Error

## 17. Output-based: What will be the output?

```
var a = 10;
{
console.log(a);
let a = 20;
}
```

Output:
ReferenceError

## 18. Output-based: What will be the output?

```
console.log(typeof myFunc);

var myFunc = function() {};

console.log(typeof myFunc);
```

Output:
undefined

function

## 19. Output-based: What will be the output?

```
foo();

var foo = function() {

console.log("First");

}

foo();

function foo() {

console.log("Second");

}

foo();
```

Output:
Second

First

First

**20. Explain the hoisting behavior in the following code and why it behaves that way:**

```
var x = 1;
function outer() {
console.log(x);
var x = 2;
function inner() {
console.log(x);
var x = 3;
console.log(x);
}
inner();
console.log(x);
}
outer();
```

Output:
undefined

undefined

3

2

# C.6 Functions (20 Questions)

## 1. What are the different ways to define a function in JavaScript?

➢ Function Declaration:
function greet() { console.log("Hello"); }

➢ Function Expression:
const greet = function() { console.log("Hello"); };

➢ Arrow Function:
const greet = () => console.log("Hello");

➢ Constructor Function:
function Person(name){ this.name = name; }

➢ Method inside object:
const obj = { greet() { console.log("Hi"); } };

## 2. What is the difference between function declaration and function expression?

➢ Function Declaration:
- Fully hoisted.
- Can be called before definition.

➢ Function Expression:
- Not fully hoisted.
- Stored in a variable.
- Cannot be called before initialization.

## 3. What are arrow functions and how are they different from regular functions?

➢ Arrow functions provide shorter syntax.
➢ Do not have their own 'this'.
➢ Do not have 'arguments' object.
➢ Cannot be used as constructors.

## 4. What is the difference between parameters and arguments?

➢ Parameters: Variables defined in function definition.
➢ Arguments: Actual values passed when calling the function.

Example:
function add(a, b) {} → a, b are parameters.
add(5, 10) → 5, 10 are arguments.

## 5. What are default parameters in JavaScript functions?

➢ Default parameters allow assigning default values.

Example:
```
function sum(a, b = 5) {
  return a + b;
}
```

## 6. What are rest parameters and how are they used?

➢ Rest parameters collect multiple arguments into an array.

Example:
```
function test(...args) {
  console.log(args);
}
```

## 7. What is the spread operator and how does it differ from rest parameters?

➢ Spread operator (...) expands elements.
➢ Rest collects elements. Collect multiple values into a single array.

Example:
```
const arr = [1,2,3];
console.log(...arr);
```

## 8. What is an IIFE (Immediately Invoked Function Expression) and why is it used?

➢ IIFE runs immediately after definition.

Example:

```
(function(){
  console.log("IIFE");
})();
```

## 9. What is a callback function?

➢ A function passed as argument to another function.

Example:

```
function greet(name, callback){
  console.log("Hi " + name);
  callback();
}
```

## 10. What is function hoisting and how does it differ from variable hoisting?

➢ Function declarations are fully hoisted.
➢ Variables declared with var are hoisted as undefined.
➢ let/const remain in TDZ.

## 11. Explain hoisting differences (declaration vs expression vs arrow).

➢ Declaration: Fully hoisted.
➢ Expression: Variable hoisted, function not initialized.
➢ Arrow: Same as expression.

## 12. What are higher-order functions?

➢ Functions that take another function as argument or return one.

Example:

```
[1,2,3].map(x => x*2);
```

## 13. How do arrow functions handle 'this' differently?

➢ Arrow functions don’t have their own this keyword value, it inherit 'this' from parent scope.
➢ Regular functions have their own 'this' depending on how they are called.

## 14. What is the arguments object? Does it exist in arrow functions?

➢ arguments is array-like object available in regular functions.

➢ Not available in arrow functions.

## 15. Explain function currying with example.

➢ A function with multiple parameters is transformed into a series of nested functions, each taking one argument.

Example:

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

## 16. Output-based: Default Parameter

```
function sum(a, b = 5) {
 return a + b;
}
console.log(sum(10));
console.log(sum(10, 20));
```

Output:

15

30

## 17. Output-based: Rest Parameter

```
function test(...args) {
 console.log(args);
 console.log(typeof args);
}
test(1,2,3,4,5);
```

Output:

[1,2,3,4,5]

object

## 18. Output-based: IIFE Scope

```
(function() {
 var a = 10;
 console.log(a);
})();
console.log(a);
```

Output:
10
ReferenceError

## 19. Output-based: Arrow Function

```
const multiply = (a, b) => a * b;
console.log(multiply(5, 3));
```

Output:
15

## 20. Output-based: Closure Example

```
function outer() {
 var x = 10;
 function inner() {
  console.log(x);
 }
 return inner;
}
var func = outer();
func();
```

Output:
10
Reason: Closure retains access to outer scope

# C.7 Higher-Order Functions (20 Questions)

## 1. What is a higher-order function in JavaScript?

➢ A higher-order function is a function that either:
• Takes another function as an argument, OR
• Returns a function as a result.

Example:
```
function greet(fn) {
  fn();
}
```

## 2. What does it mean that functions are first-class citizens in JavaScript?

➢ Functions can be:
• Assigned to variables
• Passed as arguments
• Returned from other functions
• Stored in objects/arrays

Example:
```
const sayHello = function() { console.log("Hello"); };
```

## 3. How does JavaScript enable higher-order functions?

➢ Because functions are first-class citizens.
➢ They can be passed and returned like normal values.
➢ This enables functional programming patterns.

## 4. Give 3 examples of built-in higher-order functions in JavaScript.

➢ map()
➢ filter()
➢ reduce()

## 5. Can a function return another function? Give an example.

Yes.

Example:

```
function outer() {
  return function inner() {
    console.log("Hello");
  };
}
```

## 6. Explain how map() is a higher-order function.

➢ map() takes a callback function as argument.

➢ It applies the function to each array element.

Example:

```
[1,2,3].map(n => n * 2);
```

## 7. Explain how filter() is a higher-order function.

➢ filter() takes a callback function.

➢ Returns elements that satisfy condition.

Example:

```
[1,2,3,4].filter(n => n % 2 === 0);
```

## 8. Explain how reduce() is a higher-order function.

➢ reduce() takes a callback function.

➢ Combines array values into single result.

Example:

```
[1,2,3].reduce((acc, n) => acc + n, 0);
```

## 9. Difference between higher-order and regular function?

➢ Regular function: Does not take or return a function.

➢ Higher-order function: Takes or returns a function.

## 10. Create a higher-order function that executes a function twice.

```
function executeTwice(fn) {
  fn();
  fn();
}
```

```
executeTwice(() => console.log("Hello"));
```

## 11. What is function composition?

➢ Function composition combines multiple functions into one.

Example:

```
const add = x => x + 2;
const multiply = x => x * 3;
const compose = x => multiply(add(x));
```

## 12. Benefits of using higher-order functions?

➢ Code reusability
➢ Cleaner syntax
➢ Functional programming support
➢ Improved abstraction

## 13. How do higher-order functions improve reusability?

➢ Common logic can be abstracted.
➢ Functions can be reused with different behaviors via callbacks.

## 14. Relationship between HOF and functional programming?

➢ Higher-order functions are core concept in functional programming.
➢ Enable immutability, pure functions, composition.

## 15. Create a higher-order function that returns a function (closure example).

```
function createMultiplier(factor) {
  return function(number) {
    return number * factor;
  };
}
```

## 16. Output-based

```
function higherOrder(fn) {
  fn();
```

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

Output:
Hello
Hello

## 17. Output-based

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}
const double = multiplier(2);
console.log(double(5));
```

Output:
10

## 18. Output-based

```
const numbers = [1,2,3];
const doubled = numbers.map(function(n){ return n*2; });
console.log(doubled);
```

Output:
[2,4,6]

**19. Output-based**

```
function operate(a,b,operation){
  return operation(a,b);
}
const result = operate(5,3,function(x,y){ return x+y; });
console.log(result);
```

Output:
8

**20. Output-based**

```
function createCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}
const counter = createCounter();
console.log(counter());
console.log(counter());
console.log(counter());
```

Output:
1
2
3

# C.8 Callback Functions (20 Questions)

## 1. What is a callback function in JavaScript?

➢ A callback function is a function that is passed as an argument to another function and is executed after some operation is completed.

Example:

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}
```

## 2. Why are callback functions used?

➢ To handle asynchronous operations.

➢ To execute code after another function completes.

➢ To improve flexibility and reusability.

➢ To customize behavior dynamically.

## 3. How do you pass a function as a callback?

➢ By passing the function reference as an argument.

Example:

```
function sayBye() {
  console.log("Goodbye");
}

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

greet(sayBye);
```

## 4. Difference between callback and regular function?

➢ Regular function executes directly.

➢ Callback is passed to another function and executed later.

## 5. 3 examples of functions that accept callbacks

➢ setTimeout()

➢ setInterval()

➢ map(), filter(), forEach()

## 6. What is the difference between synchronous and asynchronous callbacks?

➢ Synchronous callbacks run immediately.

➢ Asynchronous callbacks run later via event loop.

## 7. How do synchronous callbacks work? Give an example with map().const

```
arr = [1,2,3];
const doubled = arr.map(function(n) {
  return n * 2;
});
console.log(doubled);
```

Output: [2,4,6]

## How do asynchronous callbacks work? Give an example with setTimeout().

```
console.log("Start");
setTimeout(function() {
  console.log("Inside Timeout");
}, 1000);

console.log("End");
```

Output:
Start
End
Inside Timeout

## 9. How do you pass arguments to a callback function?

```
function process(data, callback) {
  callback(data);
}

process(5, function(num) {
  console.log(num * 2);
});
```

## 10. What is an anonymous callback vs a named callback?

➢ Anonymous: function() {}

➢ Named: function greet() {}

Example:

```
setTimeout(function(){ console.log("Hi"); }, 1000);
```

## 11. What are the advantages of using callback functions?

➢ Enables async programming.

➢ Improves modularity.

➢ Reusable logic.

## 12. How are callbacks used in event handling?

```
document.addEventListener("click", function() {
  console.log("Clicked");
});
```

## 13. How are callbacks used in array methods like forEach(), map(), filter()?

```
[1,2,3].forEach(function(n) {
  console.log(n);
});
```

## 14. What is callback hell and why is it a problem?

Deeply nested callbacks making code unreadable.

**Callback hell** refers to a situation where multiple asynchronous callback functions are nested inside one another, creating deeply indented and hard-to-read code.

It is also called:

“Pyramid of Doom”

## 15. How to avoid callback hell?

➢ Use Promises.

➢ Use async/await.

➢ Modularize code.

## 16. How callbacks enable async programming in js?

They allow non-blocking execution by deferring code execution until task completion.

## 17. What is the relationship between callbacks and the event loop?

Async callbacks go to callback queue.
Event loop moves them to call stack when empty.

## 18. How do error-first callbacks work in Node.js?

First parameter represents error.

Example:

```
fs.readFile("file.txt", function(err, data) {
```

```
  if (err) console.log(err);
});
```

### 19. What are the limitations of callbacks compared to Promises?

➢ Hard error handling

➢ Callback hell

➢ Difficult chaining

### 20. Output-based question

```
function processData(data, callback) {
  console.log("Processing:", data);
  callback(data * 2);
}

processData(5, function(result) {
  console.log("Result:", result);
});
```

Output:
Processing: 5
Result: 10

## C.9 Objects (15 Questions)

### 1. What are objects in JavaScript and how do you create them?

➢ Objects are collections of key-value pairs.

➢ Keys are strings (or symbols) and values can be any data type.

Creation Methods:
• Object literal:

```
const obj = { name: "John", age: 30 };
```

• Using new Object():

```
const obj = new Object();
```

• Constructor function:

```
function Person(name) { this.name = name; }
const p = new Person("John");
```

## 2. What is the difference between dot notation and bracket notation for accessing object properties?

➢ Dot notation: Used when property name is known and valid identifier.
obj.name

➢ Bracket notation: Used for dynamic keys or special characters.
obj["name"]

Bracket notation allows variables:
let key = "name";
obj[key];

## 3. How do you add, modify, and delete properties from an object?

➢ Add:
obj.city = "Delhi";

➢ Modify:
obj.age = 35;

➢ Delete:
delete obj.city;

## 4. What is the delete operator and how does it work?

➢ delete removes a property from an object.
➢ It does not affect variables declared with let/const/var.

Example:
const obj = { a: 1 };
delete obj.a;

## 5. What is the this keyword in object methods?

➢ 'this' refers to the object that calls the method.

Example:
```
const person = {
  name: "John",
  greet() {
    console.log(this.name);
  }
};
```

## 6. What are the different ways to create objects in JavaScript (literal, constructor, Object.create)?

➢ Literal:
const obj = {};

➢ Constructor:
function Person(name){ this.name = name; }

➢ Object.create():
const obj = Object.create({ greet(){ console.log("Hi"); }});

## 7. What is object destructuring and how does it work?

➢ Extract properties into variables.

Example:
const user = { name: "John", age: 30 };
const { name, age } = user;

## 8. Explain the difference between shallow copy and deep copy.

➢ Shallow copy copies top-level properties only.

➢ Nested objects share same reference.

Example:
const obj2 = { ...obj1 };

➢ Deep copy creates independent nested copies.
Example:
JSON.parse(JSON.stringify(obj1));

## 9. What are Object.keys(), Object.values(), and Object.entries()?

➢ Object.keys(obj) → returns array of keys.

➢ Object.values(obj) → returns array of values.

➢ Object.entries(obj) → returns array of [key, value] pairs.

## 10. How do you check if a property exists in an object?

➢ Using 'in' operator:
"key" in obj;

➢ Using hasOwnProperty():
obj.hasOwnProperty("key");

## 11. Output-based: What will be the output?

```
const obj = {
name: "John",
age: 30
};
delete obj.age;
console.log(obj);
```

Output:
{ name: "John" }

Reason:
delete removes the property 'age'.

## 12. Output-based: What will be the output?

```
const a = {};
const b = { key: "b" };
const c = { key: "c" };
a[b] = 123;
a[c] = 456;
console.log(a[b]);
```

Output:
456

Reason:
Object keys are converted to strings.
Both b and c become "[object Object]" so last assignment overwrites first.

## 13. Output-based: What will be the output?

```
const obj = { a: 1, b: 2, c: 3 };
const { a, ...rest } = obj;
console.log(a);
console.log(rest);
```

Output:
1
{ b: 2, c: 3 }

Reason:
Destructuring extracts 'a', rest operator collects remaining properties.

## 14. Output-based: What will be the output?

```
const person = {
name: "Alice",
greet: function() {
console.log(this.name);
}
};
const greet = person.greet;
greet();
```

Output:
undefined (or error in strict mode)

Reason:
Function is called without object context, so 'this' is not person.

### 15. Output-based: What will be the output?

```
const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { ...obj1 };
obj2.b.c = 3;
console.log(obj1.b.c);
console.log(obj2.b.c);
```

Output:
3
3

Reason:
Spread operator creates shallow copy.
Nested object 'b' shares same reference.

# C.10 Arrays & Array Methods (20 Questions)

### 1. What is an array and how do you create it in JavaScript?

➢ An array is an ordered collection of values stored in a single variable.
➢ Arrays can hold multiple data types.

Creation Methods:
• Literal syntax:

```
const arr = [1, 2, 3];
```

• Using Array constructor:

```
const arr = new Array(1, 2, 3);
```

### 2. What is the difference between map() and forEach()?

➢ map() returns a new array after transforming each element.
➢ forEach() executes a function for each element but does not return a new array.

map() is used for transformation.
forEach() is used for iteration.

## 3. What does the filter() method do?

➢ filter() creates a new array containing elements that satisfy a condition.

Example:
[1,2,3,4].filter(n => n % 2 === 0);
// Returns: [2,4]

## 4. What is the reduce() method and how does it work?

➢ reduce() executes a callback on each element and reduces the array to a single value.

Syntax:
array.reduce((accumulator, currentValue) => {}, initialValue);

Example:
[1,2,3].reduce((acc, curr) => acc + curr, 0);
// Returns: 6

## 5. What is the difference between push() and pop() methods?

➢ push() adds an element to the end of the array.
➢ pop() removes the last element from the array.

Both modify the original array.

## 6. Explain the map() method with an example. Does it modify the original array?

➢ map() applies a function to each element and returns a new array.
➢ It does NOT modify the original array.

Example:
const arr = [1,2,3];
const result = arr.map(x => x * 2);
Result: [2,4,6]
Original array remains unchanged.

## 7. Explain the filter() method with an example. What does it return?

➢ filter() returns a new array with elements that pass a condition.

Example:
const arr = [1,2,3,4];
const result = arr.filter(x => x > 2);
Returns: [3,4]

## 8. Explain the reduce() method in detail. What are accumulator and current value?

➢ accumulator: Stores accumulated result.

➢ current value: Current element being processed.

Example:
const arr = [1,2,3];
arr.reduce((acc, curr) => acc + curr, 0);

Step-by-step:
0+1=1
1+2=3
3+3=6

## 9. How do you chain map(), filter(), and reduce() methods?

Example:
const arr = [1,2,3,4,5];
const result = arr
.map(x => x * 2)
.filter(x => x > 5)
.reduce((acc, curr) => acc + curr, 0);

## 10. What is the difference between map() and reduce()?

➢ map() transforms each element and returns a new array.

➢ reduce() combines all elements into a single value.

## 11. Write a polyfill for the map() method.

```
Array.prototype.myMap = function(callback) {
  const result = [];
  for (let i = 0; i < this.length; i++) {
    result.push(callback(this[i], i, this));
  }
  return result;
};
```

## 12. Write a polyfill for the filter() method.

```
Array.prototype.myFilter = function(callback) {
  const result = [];
  for (let i = 0; i < this.length; i++) {
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }
  return result;
};
```

## 13. Write a polyfill for the reduce() method.

```
Array.prototype.myReduce = function(callback, initialValue) {
  let acc = initialValue;
  for (let i = 0; i < this.length; i++) {
    acc = callback(acc, this[i], i, this);
  }
  return acc;
};
```

## 14. What is the time complexity of map(), filter(), and reduce()?

➢ All three methods have O(n) time complexity.

➢ They iterate through the array once.

## 15. How do array methods handle empty elements?

➢ map(), filter(), and reduce() skip empty slots in sparse arrays.

Example:

```
const arr = [1,,3];
arr.map(x => x * 2); // Skips empty slot
```

## 16. Output-based: What will be the output?

```
const arr = [1, 2, 3, 4, 5];
const result = arr.map(x => x * 2);
console.log(result);
console.log(arr);
```

Output:

[2,4,6,8,10]

[1,2,3,4,5]

Reason:
map() does not modify original array.

## 17. Output-based: What will be the output?

```
const arr = [1, 2, 3, 4, 5];
const result = arr.filter(x => x % 2 === 0);
console.log(result);
```

Output:
[2,4]

## 18. Output-based: What will be the output?

```
const arr = [1, 2, 3, 4, 5];
const result = arr.reduce((acc, curr) => acc + curr, 0);
console.log(result);
```

Output:
15

## 19. Output-based: What will be the output?

```
const arr = [1, 2, 3, 4, 5];
const result = arr
.map(x => x * 2)
.filter(x => x > 5)
.reduce((acc, curr) => acc + curr, 0);
console.log(result);
```

Step-by-step:
map → [2,4,6,8,10]
filter (>5) → [6,8,10]
reduce sum → 24

Output:
24

### 20. Output-based: What will be the output?

```
const users = [
{ name: "John", age: 25 },
{ name: "Jane", age: 30 },
{ name: "Bob", age: 25 }
];
const result = users.reduce((acc, curr) => {
acc[curr.age] = (acc[curr.age] || 0) + 1;
return acc;
}, {});
console.log(result);
```

Output:
{ 25: 2, 30: 1 }

Reason:
reduce groups users by age and counts occurrences.

# C.11 Destructuring (10 Questions)

### 1. What is destructuring in JavaScript?

➢ Destructuring is a syntax in JavaScript that allows unpacking values from arrays or properties from objects into separate variables.

➢ It provides a cleaner and shorter way to extract data.

Example:
const arr = [1, 2];
const [a, b] = arr;

### 2. What is array destructuring and how does it work?

➢ Array destructuring extracts values based on position (index).

Example:
const numbers = [10, 20, 30];
const [x, y, z] = numbers;

Here:
x = 10
y = 20
z = 30

## 3. What is object destructuring and how does it work?

➢ Object destructuring extracts values based on property names.

Example:
const user = { name: "John", age: 25 };
const { name, age } = user;

Here:
name = "John"
age = 25

## 4. How do you assign default values in destructuring?

➢ Default values are assigned using = operator.

Example:
const [a = 5, b = 10] = [1];
Result:
a = 1
b = 10 (default value used)

## 5. What is the rest operator in destructuring?

➢ The rest operator (...) collects remaining elements into an array or object.

Example (Array):
const [a, ...rest] = [1,2,3,4];
rest = [2,3,4]

Example (Object):
const { name, ...others } = { name: "John", age: 25, city: "Delhi" };
others = { age: 25, city: "Delhi" }

## 6. How do you rename variables during object destructuring?

➢ Use colon syntax to rename variables.

Example:

```
const user = { name: "John", age: 25 };
const { name: userName, age: userAge } = user;

userName = "John"
userAge = 25
```

## 7. What is nested destructuring? Provide an example.

➢ Nested destructuring extracts values from nested objects or arrays.

Example:

```
const user = {
  name: "John",
  address: { city: "Delhi", zip: 12345 }
};

const { address: { city, zip } } = user;

city = "Delhi"
zip = 12345
```

## 8. How do you swap two variables using destructuring?

➢ Variables can be swapped without temporary variable.

Example:

```
let a = 10;
let b = 20;

[a, b] = [b, a];

After swap:
a = 20
b = 10
```

## 9. How do you use destructuring in function parameters?

➢ Destructuring can be used directly in function parameters.

Example:

```
function greet({ name, age }) {
  console.log(name, age);
}
```

greet({ name: "John", age: 25 });

### 10. Output-based: What will be the output?

**const [a, b, ...rest] = [1, 2, 3, 4, 5];**

**console.log(a);**

**console.log(b);**

**console.log(rest);**

Output:
1
2
[3, 4, 5]

Reason:
a takes first value.
b takes second value.
rest collects remaining values into array

# C.12 Data Types & Data Comparison (10 Questions)

### 1. What are the primitive data types in JavaScript?

➢ Primitive data types are immutable and stored by value.

JavaScript has 7 primitive types:
• String
• Number
• Boolean
• Undefined
• Null
• Symbol
• BigInt

Example:
let name = "John";  // String
let age = 25;  // Number
let isActive = true; // Boolean

## 2. What are the reference data types in JavaScript?

➢ Reference types store values by reference (memory address).

Reference types include:
• Object
• Array
• Function
• Date
• RegExp
• Map, Set

Example:
```
const obj = { name: "John" };
const arr = [1, 2, 3];
```

## 3. What is the typeof operator and how is it used?

➢ typeof returns the data type of a variable.

Example:
```
typeof "Hello"   // "string"
typeof 10        // "number"
typeof true      // "boolean"
typeof {}        // "object"
typeof []        // "object" (arrays are objects)
```

## 4. What is the difference between null and undefined?

➢ undefined means a variable has been declared but not assigned a value.

➢ null is an intentional assignment representing no value.

Example:
```
let a;
console.log(a); // undefined

let b = null;
console.log(b); // null
```

## 5. What is NaN and how do you check for it?

➢ NaN stands for Not-a-Number.

➢ It represents an invalid numeric operation.

Example:
let x = "abc" / 2; // NaN

Check using:
Number.isNaN(x);

## 6. What is type coercion in JavaScript? Explain implicit and explicit coercion.

➢ Type coercion is automatic or manual conversion of one data type into another.

Implicit coercion:
console.log(1 + "2"); // "12"

Explicit coercion:
Number("10"); // 10
String(123); // "123"

## 7. What is the difference between == and ===?

➢ == compares values after type coercion.

➢ === compares both value and type without coercion.

Example:
1 == "1" // true
1 === "1" // false

## 8. What are truthy and falsy values in JavaScript? List all falsy values.

➢ Truthy values evaluate to true in boolean context.

➢ Falsy values evaluate to false.

All falsy values:
• false
• 0
• -0
• 0n (BigInt zero)
• "" (empty string)
• null
• undefined
• NaN

## 9. Output-based: What will be the output?

```
console.log(typeof null);
console.log(typeof undefined);
console.log(typeof NaN);
console.log(typeof []);
console.log(typeof {});
```

Output:
object
undefined
number
object
object

Reason:
• typeof null is a known JavaScript bug → returns "object".
• NaN is considered a number type.

## 10. Output-based: What will be the output?

```
console.log(1 == "1");
console.log(1 === "1");
console.log(null == undefined);
console.log(null === undefined);
console.log(0 == false);
console.log(0 === false);
```

Output:
true
false
true
false
true
false

Reason:
• == performs type coercion.
• === compares both type and value strictly

# C.13 Bonus Section: Tricky Interview Questions (15 Questions)

### 1. Output-based: What will be the output?

```
for (var i = 0; i < 3; i++) {
setTimeout(() => console.log(i), 1000);
}
```

Output:
3
3
3

Reason:
'var' is function-scoped. The loop completes first and i becomes 3.
All callbacks share the same reference to i, so they print 3.

### 2. Output-based: What will be the output?

```
for (let i = 0; i < 3; i++) {
setTimeout(() => console.log(i), 1000);
}
```

Output:
0
1
2

Reason:
'let' is block-scoped. Each loop iteration gets a new binding of i,
so each callback captures its own value.

### 3. Output-based: What will be the output?

```
const arr = [1, 2, 3];
arr[10] = 10;
console.log(arr.length);
console.log(arr[5]);
```

Output:
11

undefined

Reason:
Setting arr[10] creates empty slots. Array length becomes highest index + 1.
Index 5 is empty, so it returns undefined.

## 4. Output-based: What will be the output?

```
console.log([] + []);
console.log([] + {});
console.log({} + []);
console.log({} + {});
```

Output:
""
"[object Object]"
0
"[object Object][object Object]"

Reason:
Arrays and objects are coerced to strings when using + operator.
{} + [] is interpreted differently due to parsing rules.

## 5. Output-based: What will be the output?

```
console.log(+"10");
console.log(+"abc");
console.log(!!"");
console.log(!!"hello");
```

Output:
10
NaN
false
true

Reason:
+ converts to number.
!! converts value to boolean.

## 6. Output-based: What will be the output?

```
const obj = {
a: 1,
b: 2,
a: 3
};
console.log(obj);
```

Output:
{ a: 3, b: 2 }

Reason:
Duplicate keys overwrite previous values.
Last assignment wins.

## 7. Output-based: What will be the output?

```
let a = [1, 2, 3];
let b = a;
b.push(4);
console.log(a);
console.log(b);
```

Output:
[1, 2, 3, 4]
[1, 2, 3, 4]

Reason:
Arrays are reference types.
Both variables point to same memory.

## 8. Output-based: What will be the output?

```
function test() {
console.log(arguments);
}
test(1, 2, 3);
```

Output:
Arguments(3) [1, 2, 3]

Reason:
arguments is an array-like object containing passed parameters.

## 9. Output-based: What will be the output?

```
const arr = [1, 2, 3];
const [a, b, c, d = 10] = arr;
console.log(d);
```

Output:
10

Reason:
Default value is used because arr has no fourth element.

## 10. Output-based: What will be the output?

```
const obj = { x: 1, y: 2 };
const { x: a, y: b } = obj;
console.log(a);
console.log(b);
console.log(x);
```

Output:
1
2
ReferenceError

Reason:
Variables are renamed to a and b.
'x' is not defined in current scope.

## 11. Output-based: What will be the output?

```
const nums = [1, 2, 3];
nums[10] = 10;
const doubled = nums.map(x => x * 2);
console.log(doubled);
```

Output:
[2, 4, 6, empty × 7, 20]

Reason:
map() skips empty slots but preserves array length.

## 12. Output-based: What will be the output?

```
function outer() {
let x = 10;
return function inner() {
console.log(x);
let x = 20;
};
}
outer()();
```

Output:
ReferenceError

Reason:
Inner function creates new 'x' in block scope.
Accessing it before initialization causes TDZ error.

## 13. Output-based: What will be the output?

```
console.log(typeof typeof 1);
```

Output:
string

Reason:
typeof 1 → "number"
typeof "number" → "string".

## 14. Output-based: What will be the output?

```
const arr = [1, 2, 3];
const [a, , c] = arr;
console.log(a, c);
```

Output:
1 3

Reason:
Second element is skipped using comma.

**15. Output-based: What will be the output?**

```
function test(a, b = a * 2) {
console.log(a, b);
}
test(5);
test(5, 10);
```

Output:
5 10
5 10

Reason:
Default parameter uses a * 2 when b not provided.
If b is provided, default is ignored.