C.1 HTML Interview Questions (35Questions)

---------Basic level---------
1. What is HTML and what does it stand for?- HTML: HyperText Markup Language
- A standard markup language used to create web pages

2. What is the purpose of <!DOCTYPE html>?- Declares the document type: Indicates that the document is written in HTML5
- Triggers standards mode: Ensures that the browser renders the page in standards mode

3. What is the difference between HTML elements and HTML tags?- HTML Element: A complete HTML structure, including the opening and closing tags, and the content in between
- HTML Tag: A part of an HTML element, used to define the structure and content of a web page

4. What is the difference between <div> and <span>?- <div>: A block-level element, used to group block-level elements
- <span>: An inline element, used to group inline elements

5. What are semantic HTML elements? Give 5 examples- Semantic HTML elements: Elements that provide meaning to the structure of a web page
- Examples:
    - <header>
    - <nav>
    - <main>
    - <section>
    - <footer>

6. What is the difference between <section> and <article>?- <section>: A thematic grouping of content, typically with a heading
- <article>: A self-contained piece of content, such as a blog post or news article

7. When should you use <header>, <main>, and <footer>?- <header>: Use for the top section of a web page, typically containing navigation and branding
- <main>: Use for the main content of a web page
- <footer>: Use for the bottom section of a web page, typically containing copyright information and contact details

8. What is the purpose of the <nav> element?- <nav>: Represents a section of a page that provides navigation links

9. What is the difference between block-level and inline elements?- Block-level elements: Elements that occupy the full width of their parent element, such as <div> and <p>
- Inline elements: Elements that occupy only the space needed to display their content, such as <span> and <a>

10. What are void elements or self-closing tags in HTML? Give examples.-
Void elements: Elements that do not have a closing tag, such as
<img>,
, and <input>
- Examples:
    - <img src="image.jpg" alt="An image">
    -
    - <input type="text" name="username">

----------Intermediate Level----------
---

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

id	class
->Must be unique in a page	Can be used on multiple elements
->Used to identify a single element	Used to group multiple elements
->Selected in CSS using #idName	Selected in CSS using .className
->Used for JavaScript targeting specific element	Used for styling or grouping elements

Example:

```
<div id="header"></div>
<p class="text"></p>
```

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

=>data-* attributes allow you to store custom data inside HTML elements.

Example:
```
<div data-user-id="101" data-role="admin"></div>
```
You use them:

To store extra information without affecting layout
To pass data to JavaScript
For dynamic UI behavior (like filtering, sorting)

Access in JS:
```
element.dataset.userId
```

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

The alt attribute:

Provides alternative text if the image fails to load
Improves accessibility (screen readers read it)
Helps with SEO

Example:

```
<img src="logo.png" alt="Company Logo">
```

14. Difference between and <b>, and <i>

Tag	Purpose
	Semantic importance (meaning)
<b>	Just bold text (visual only)
	Emphasized meaning
<i>	Italic text (visual only)

ðŸ‘‰ and are semantic (better for accessibility & SEO).

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

Meta tags provide metadata about the webpage.

Example:

```
<meta charset="UTF-8">
<meta name="description" content="Learn HTML basics">
```

They are important for:

Character encoding
SEO
Responsive design
Browser compatibility

16. What is the viewport meta tag and why is it crucial?

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

It:

Controls layout on mobile devices
Makes website responsive
Sets page width to device width
Without it, websites appear zoomed out on phones.

17. Difference between <link> and <script> tags

<link>	<script>
Connects external resources	Adds JavaScript code
Used for CSS, icons	Used for JS files
Self-closing	Has opening and closing tag

Example:

```
<link rel="stylesheet" href="style.css">
<script src="app.js"></script>
```

18. Different input types in HTML5 (at least 8)

text
password
email
number
tel

url
date
time
file
checkbox
radio
range
color
search

19. Purpose of the name attribute in form inputs

The name attribute:

Sends data to the server when form is submitted
Acts as the key in key-value pairs

Example:

<input type="text" name="username">
Without name, the value will not be submitted.

20. Difference between GET and POST methods

GET	POST
Data sent in URL	Data sent in request body
Visible in address bar	Not visible
Limited data size	Can send large data
Less secure	More secure
Used for retrieving data	Used for submitting data

21. Purpose of <label> and how to associate it

<label> improves:

Accessibility

Clickable area for inputs

Correct association:

Method 1 (for attribute)

```
<label for="email">Email</label>
<input type="email" id="email">
```

Method 2 (Wrap input)

```
<label>
  Email
  <input type="email">
</label>
```

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

They are form validation attributes.

required â†’ Field must be filled

pattern â†’ Defines regex format

min â†’ Minimum value

max â†’ Maximum value

Example:

```
<input type="number" min="1" max="10" required>
```

23. Difference between placeholder and value

placeholder	value
Hint text	Actual default value
Disappears when typing	Stays unless changed
Not submitted if unchanged	Submitted with form

Example:

```
<input type="text" placeholder="Enter name">
<input type="text" value="Raj">
```

24. Difference between <button>, <input type="button">, <input type="submit">

Element	Purpose
<button>	Can be button, submit, reset
<input type="button">	Just a clickable button
<input type="submit">	Submits form

Example:

```
<button type="submit">Submit</button>
<input type="button" value="Click">
<input type="submit" value="Send">
```

<button> is more flexible (can contain HTML inside).

25. What is the target attribute in anchor tags?

target specifies where to open the linked document.

```
<a href="https://example.com" target="_blank">Visit</a>
```

Common values:

_self (default)
_blank

_parent
_top

What is target="_blank"?

Opens the link in a new tab.
Security Concern
It can cause a vulnerability called reverse tabnabbing.

To fix:

<a href="https://example.com" target="_blank" rel="noopener noreferrer">
This prevents the new page from accessing the original page via window.opener.

----------Advance level----------

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

ARIA (Accessible Rich Internet Applications) improves accessibility when native HTML semantics are not enough.

ARIA:

Provides roles (role="button")
Defines states (aria-expanded="true")
Adds properties (aria-label="Close menu")

Example:

Toggle

Use ARIA when:

Building custom UI components (modals, dropdowns, tabs)
Native semantic elements cannot represent the UI behavior

Important Rule:
Use semantic HTML first. Use ARIA only when necessary.

27. How do you make HTML forms accessible?

Advanced accessibility practices include:
Proper <label> association
Use semantic inputs (type="email", type="number")
Use aria-describedby for hints/errors
Group related fields using <fieldset> and <legend>
Clear error messaging
Keyboard navigability
Sufficient color contrast

Example:

```
<fieldset>
  <legend>Contact Info</legend>
  <label for="email">Email</label>
  <input id="email" type="email" aria-describedby="emailHelp">
  <small id="emailHelp">We'll never share your email.</small>
</fieldset>
```

Accessibility ensures compatibility with:

Screen readers
Keyboard-only users
Assistive technologies

28. Difference between <script>, <script defer>, and <script async>

Type	Loading	Execution	Order Guarantee
<script>	Blocks HTML parsing	Immediately	Yes
defer	Downloads in parallel	After HTML parsed	Yes
async	Downloads in parallel	As soon as ready	No

<script>

Blocks DOM parsing until executed.

<script defer>

Best for DOM-dependent scripts
Maintains execution order
Executes after HTML is fully parsed

<script async>

Best for third-party scripts (analytics)
Does not guarantee order

Modern best practice:

```
<script src="app.js" defer></script>
```

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

HTML entities represent reserved or special characters.

Used when:

Character conflicts with HTML syntax
Special symbols are required

Non-breaking spaces are needed

Examples:

Character	Entity
<	<
>	>
&	&
"	"
Non-breaking space	
Â©	©

Example:

```
<p>5 &lt; 10</p>
```

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

<picture> provides art direction and responsive images.

It allows:

Different images for different screen sizes
Different formats (WebP, AVIF fallback)
Media condition-based selection

Example:

```
<picture>
  <source srcset="image.webp" type="image/webp">
  <source srcset="image.jpg" type="image/jpeg">
  <img src="image.jpg" alt="Sample Image">
</picture>
```

Difference:

<img>	<picture>
Single image source	Multiple image sources
Basic responsiveness	Advanced art direction
No format fallback	Format fallback support

31. What is the srcset attribute in images?

srcset allows browsers to choose the best image based on:

Screen resolution
Viewport size
Device pixel ratio

Example:

```
<img
  src="small.jpg"
```

```
  srcset="small.jpg 480w, medium.jpg 768w, large.jpg 1200w"
  sizes="(max-width: 600px) 480px, 800px"
  alt="Responsive image">
```

Benefits:

Better performance
Optimized loading
Reduced bandwidth usage

32. Difference between <audio> and <video>

Both are HTML5 media elements.

Feature	<audio>	<video>
Plays sound	yes	yes
Plays video	no	yes
Supports captions	Limited	Yes (via <track>)
Poster image	No	yes

Example:

```
<video controls>
  <source src="movie.mp4" type="video/mp4">
  <track src="subtitles.vtt" kind="subtitles">
</video>
```

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

Basic iframe:

```
<iframe src="https://example.com"></iframe>
```

Advanced secure iframe:

```
<iframe
  src="https://example.com"
  sandbox="allow-scripts"
  loading="lazy"
  referrerpolicy="no-referrer">
</iframe>
```

Security concerns:

Clickjacking
XSS attacks
Malicious third-party content

Best practices:

Use sandbox

Use allow restrictions
Use HTTPS
Avoid untrusted sources

34. What is Shadow DOM and how does it relate to Web Components?

Shadow DOM provides encapsulation of DOM and CSS.

It:

Isolates styles
Prevents global CSS conflicts
Creates component-based architecture

Used in:

Web Components
Custom Elements

Example (JavaScript):

```
const shadow = element.attachShadow({ mode: 'open' });
shadow.innerHTML = `<style>p {color:red}</style><p>Hello</p>`;
```

Shadow DOM ensures:

Scoped styling
DOM isolation
Reusable components

35. What is the purpose of contenteditable attribute?

contenteditable makes an element editable by the user.

Example:

```
<div contenteditable="true">
  You can edit this text.
</div>
```

Used in:

Rich text editors
CMS platforms
In-browser editing tools

Important considerations:

Does not provide built-in validation
Requires JS for saving content
Security risk if saving unsanitized HTML (XSS)

C.2 CSS Interview Questions (40 Questions)

---

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

CSS stands for Cascading Style Sheets.

It is used to:

Style HTML elements
Control layout, colors, fonts, spacing
Make websites visually attractive

Example:

```
p {
  color: blue;
}
```

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

There are 3 ways:

Inline CSS
Internal CSS
External CSS

3. Difference between Inline, Internal, and External CSS

Type	Where Written	Usage
Inline	Inside HTML element	For single element styling
Internal	Inside <style> tag in <head>	For single page styling
External	Separate .css file	For multiple pages

Inline Example

```
<p style="color:red;">Hello</p>
```

Internal Example

```
<style>
  p { color: blue; }
```

```
</style>
```

External Example

```
<link rel="stylesheet" href="style.css">
```

4. What are CSS selectors? List different types.

Selectors are used to select HTML elements to apply styles.

Common types:

```
Element selector (p)
Class selector (.box)
ID selector (#header)
Universal selector (*)
Group selector (h1, h2)
Attribute selector (input[type="text"])
Pseudo-class selector (:hover)
Pseudo-element selector (::before)
```

5. What is the universal selector (*)?

The universal selector * selects all elements on a page.

Example:

```
* {
  margin: 0;
  padding: 0;
}
```

It is commonly used for CSS reset.

6. What are class selectors and id selectors?

Class Selector

Starts with .

Used for multiple elements

```
.box {
  background: yellow;
}
```

ID Selector

Starts with #

Used for one unique element

```
#header {
  background: blue;
}
```

7. Difference between class and id in CSS

Class	ID
Used for multiple elements	Used for single element
Starts with .	Starts with #
Lower specificity	Higher specificity

Example:

```
<div class="box"></div>
<div id="main"></div>
```

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

Pseudo-classes define special states of elements.

Examples:

:hover

:active

:focus

:first-child

:nth-child()

Example:

```
a:hover {
  color: red;
}
```

9. What are pseudo-elements? Give examples.

Pseudo-elements style specific parts of an element.

Examples:

::before

::after

::first-letter

::first-line

::selection

Example:

```
p::first-letter {
  font-size: 30px;
}
```

10. Difference between :hover and :active

:hover	:active
When mouse is over element	When element is being clicked
Triggered on mouse enter	Triggered while pressing

Example:

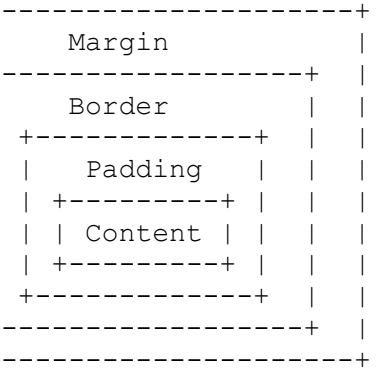
```
button:hover {
  background: green;
}

button:active {
  background: red;
}
```

----------intermediate level---------

11. What is the CSS Box Model? Explain all components.

The CSS Box Model describes how elements are structured and how their size is calculated in a webpage.

Every element is a rectangular box made of:



Components:

Content

Actual text or image inside the element.

Padding

Space between content and border.

Border

Surrounds padding and content.

Margin

Space outside the border (separates elements).

Example:

```
div {
  width: 200px;
  padding: 10px;
  border: 5px solid black;
  margin: 20px;
}
```

Total width (default box model) =
200 + 20 (padding) + 10 (border) = 230px
(Margin not included in element size)

12. Difference between margin and padding

Margin	Padding
Space outside border	Space inside border
Separates elements	Creates space inside element
Transparent	Background color applies
Can collapse	Does not collapse

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

Margin collapse happens when vertical margins of two block elements overlap.
Instead of adding, the larger margin is used.

Example:

```
div {
  margin-top: 20px;
}
```

If two elements have:

First: margin-bottom: 30px
Second: margin-top: 20px
Final space = 30px, not 50px.

Occurs:

Between vertical margins
Between parent and first/last child
Does NOT happen with padding or borders

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

By default, width/height apply only to content.

With:

box-sizing: border-box;

The width includes:

Content
Padding
Border

Example:

```
div {
  width: 200px;
  padding: 20px;
  border: 10px solid black;
  box-sizing: border-box;
}
```

Total width = 200px (fixed)

Why useful?

Easier layout calculations
Prevents overflow issues
Commonly used in modern CSS resets

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

Default value:

box-sizing: content-box;

16. Difference between content-box and border-box

content-box	border-box
Width = content only	Width = content + padding + border
Increases total size	Keeps total size fixed
Default value	Commonly used for layouts

17. Different display property values
1. block

Takes full width
Starts on new line

Example: , <p>

2. inline

Takes only required width
Does not start new line
Cannot set width/height

Example:

3. inline-block

Inline behavior

Allows width & height

Used for buttons, navigation items

4. none

Hides element completely

Removes from layout

5. flex

Enables flexbox layout

Aligns items horizontally/vertically

6. grid

Enables grid layout

Two-dimensional layout system

7. inline-flex

Inline version of flex

8. inline-grid

Inline version of grid

18. Difference between display: none and visibility: hidden
display: none visibility: hidden
Element removed from layout Element hidden but space remains
No space occupiedSpace still occupied
Cannot interact Cannot interact
19. What is display: inline-block used for?

Used when you want:

Elements to stay inline

But still set width and height

Example:

```
.box {
  display: inline-block;
  width: 100px;
  height: 100px;
}
```

Common use:

Navigation menus

Buttons in a row

Image galleries

20. Difference between block and inline-block

block	inline-block
Takes full width	Takes only needed width
Starts on new line	Stays in same line
Width/height can be set	Width/height can be set

Example:

```
div {
  display: block;
}

span {
  display: inline-block;
}
```

----------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.

It is calculated based on selector type.

Specificity hierarchy (highest to lowest):

Inline styles

IDs

Classes, attributes, pseudo-classes

Elements, pseudo-elements

Specificity is often written as:

Inline - ID - Class - Element

Example:

```
#header { color: red; }
.container p { color: blue; }
```

The #header rule wins because ID has higher specificity.

22. Specificity value of different selectors

Selector Type	Specificity Value
Inline style	1000
ID	100
Class / Attribute / Pseudo-class	10
Element / Pseudo-element	1
Universal selector *	0

Example:

```
#id        â†’ 100
.class     â†’ 10
p          â†’ 1
```

Combined example:

```
#nav .item p
```

Specificity =
100 (ID) + 10 (class) + 1 (element) = 111

23. What is the cascade in CSS?

Cascade means how CSS rules are applied when multiple rules affect the same element.

CSS decides based on:

Importance (!important)

Specificity

Source order (last rule wins)

Example:

```
p { color: red; }
p { color: blue; }
```

Result â†’ Blue (because it appears later).

24. What does !important do and why avoid it?

!important overrides normal specificity rules.

Example:

```
p {
  color: red !important;
}
```

It forces the rule to apply even if another rule has higher specificity.

Why avoid it?

Breaks natural cascade

Makes debugging difficult

Hard to maintain large projects

Best practice: Fix specificity instead of using !important.

25. Different position values in CSS

static (default)

relative

absolute

fixed

sticky

26. Difference between position: relative and position: absolute
relative

Positioned relative to its original position

Still remains in normal document flow

```
.box {
  position: relative;
```

```
  top: 10px;
}
```

absolute

Removed from normal flow

Positioned relative to nearest positioned ancestor

```
.box {
  position: absolute;
  top: 0;
  left: 0;
}
```

If no positioned parent â†’ relative to <body>.

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

Positioned relative to the viewport

Does not move while scrolling

Removed from normal flow

Example:

```
.header {
  position: fixed;
  top: 0;
}
```

Used for:

Navigation bars

Floating buttons

Chat widgets

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

sticky is a mix of relative and fixed.

Acts like relative until a scroll threshold

Then behaves like fixed

Example:

```
nav {
  position: sticky;
```

```
  top: 0;
}
```

Requirements:

Must define top, left, etc.

Works within parent container

Common use:

Sticky headers

Section titles

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

z-index controls stacking order of overlapping elements.

Higher value â†’ appears on top.

```
.box1 { position: absolute; z-index: 1; }
.box2 { position: absolute; z-index: 2; }
```

Important:

Works only on positioned elements (relative, absolute, fixed, sticky)

30. What is a stacking context in CSS?

A stacking context is a layer that controls how elements stack vertically.

Created when:

position with z-index

opacity < 1

transform

filter

position: fixed

position: sticky
Each stacking context works independently.

Example:

```
.parent {
  position: relative;
```

```
  z-index: 1;
}
```

Child elements cannot escape their parentâ€™s stacking context.

---

31. Different CSS Units

CSS units are broadly divided into absolute and relative units:

Absolute Units (fixed size, not influenced by other elements):

px â€” pixels, fixed size relative to the screen. 1px â‰ˆ 1/96th of an inch. Best for precise control.
pt â€” points, used mainly in print (1pt = 1/72 inch).
cm, mm, in â€” physical units, rarely used for screens.
Relative Units (scale based on other properties or viewport):
% â€” percentage of the parent element (width, height, padding, etc.).
em â€” relative to the font-size of the parent element. Can compound with nested elements.
rem â€” relative to the root element (<html>) font-size. Consistent across the page.
vh â€” 1% of viewport height.
vw â€” 1% of viewport width.
vmin / vmax â€” 1% of the smaller/larger dimension of viewport.

32. Difference Between em and rem

Feature	em	rem
Relative to	Parent elementâ€™s font-size	Root elementâ€™s font-size (<html>)
Nesting effect	Compounds in nested elements	Consistent, doesnâ€™t compound
Use case	Component-level scaling	Global typography or layout scaling

Example:

```
html { font-size: 16px; }
div { font-size: 2em; } /* 32px if parent is 16px */
p { font-size: 1.5rem; } /* 24px regardless of parent */
```

33. When to Use Relative vs Absolute Units

Absolute units (px): Precise control, design fidelity matters, non-scaling elements like borders/icons.
Relative units (em, rem, %): Responsive typography, fluid layouts, accessibility (user zoom or custom font sizes).
Viewport units (vh, vw): Fullscreen sections, responsive spacing based on screen size.
Rule of thumb: Use relative units for responsiveness and accessibility, absolute units for fixed layouts.

34. Viewport Units (vh, vw, vmin, vmax)

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

Example: Full-page hero section:

```
.hero {
  height: 100vh;
  width: 100vw;
}
```

35. Media Queries

Media queries allow conditional CSS based on device characteristics like width, height, resolution.

Syntax:

```
@media (max-width: 768px) {
  body { font-size: 14px; }
}
```

Evaluates the condition; if true, applies enclosed styles.
Can target width, height, orientation, resolution, and prefers-color-scheme.

36. Mobile-First Approach in Responsive Design

Start designing for small screens first (min-width media queries).
Gradually enhance for larger screens.
Advantages: Faster mobile load, progressive enhancement, simpler scaling.

```
/* Mobile first */
body { font-size: 14px; }

@media (min-width: 768px) {
  body { font-size: 16px; }
}
```

37. Difference Between min-width and max-width in Media Queries

Feature	min-width	max-width
Trigger	Styles apply when viewport â‰¥ given width	Styles apply when viewport â‰¤ given width
Use	Mobile-first	Desktop-first / legacy designs

38. CSS Variables (Custom Properties)

Defined with -- prefix in selectors (commonly :root).

Accessed using var().

Can be dynamic and change based on context (themes, media queries).

```
:root {
  --primary-color: #3498db;
}

button {
  background-color: var(--primary-color);
}
```

Advanced use: Combining variables with calc(), fallbacks:

```
color: var(--text-color, black);
width: calc(100% - var(--sidebar-width));
```

39. calc() Function in CSS

Performs mathematical operations in CSS values.

Supports +, -, *, /.
Useful for dynamic layouts where combining units is needed.

```
.container {
  width: calc(100% - 2rem);
  font-size: calc(1rem + 0.5vw);
}
```

40. CSS Preprocessors (Sass, Less)

CSS Preprocessors add programming capabilities to CSS: variables, nesting, mixins, functions, loops.
Compiled into standard CSS.

Advantages:

Variables â†’ centralize colors, spacing, fonts.
Nesting â†’ cleaner, hierarchical CSS.
Mixins & functions â†’ reusable logic.
Conditionals & loops â†’ dynamic styles, utility classes.

Example in Sass:

```
$primary-color: #3498db;

.button {
  color: $primary-color;
  &:hover {
    color: darken($primary-color, 10%);
  }
}
```

C.3 JavaScript Introduction & V8Engine (10 Questions)

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

JavaScript (JS) is a high-level, interpreted programming language primarily used to make websites interactive. It runs in browsers and on servers (via Node.js).

Key features:

Interpreted language â€“ runs line by line in the browser.
Dynamic typing â€“ variables can change types.
Event-driven â€“ responds to user actions like clicks.
Prototype-based â€“ uses prototypes for inheritance.
Lightweight and flexible â€“ can be used for front-end, back-end, and mobile apps.

2. Difference Between Interpreted and Compiled Languages

Feature	Compiled Language	Interpreted Language
Execution	Translated to machine code before running	Read and executed line by line at runtime
Speed	Usually faster	Usually slower
Errors	Detected at compile time	Detected at runtime
Examples	C, C++	JavaScript, Python

JavaScript is primarily interpreted, but modern engines use a combination of interpretation + JIT compilation for better performance.

3. What is the V8 Engine and Which Browsers/Platforms Use It?

V8 is Googleâ€™s JavaScript engine that converts JS into machine code for faster execution.

It powers:

Google Chrome
Microsoft Edge (Chromium-based)
Node.js (server-side JavaScript)

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

JIT compilation is a technique where the JS engine converts code to machine code at runtime (while the program is running).
This makes JS faster than purely interpreted code, because frequently executed code is optimized.

5. Role of the Call Stack in JavaScript Execution

The Call Stack is a data structure that keeps track of function calls in JavaScript.

How it works:

When a function is called, itâ€™s pushed onto the stack.

JavaScript executes the function.
When finished, itâ€™s popped off the stack.

Helps JavaScript track execution order and handle nested or recursive function calls.

```
function first() {
  second();
  console.log("first");
}

function second() {
  console.log("second");
}

first();
// Call stack execution order:
// 1. first()
// 2. second()
// 3. second() finishes â†’ pop
// 4. first() continues
```

----------Intermediate----------

6. V8 Engine Architecture (Parser, AST, Ignition, TurboFan)

V8 is Googleâ€™s high-performance JavaScript engine. Its architecture includes:

Parser

Reads the JavaScript code and checks syntax.
Converts code into an Abstract Syntax Tree (AST).

AST (Abstract Syntax Tree)

Tree representation of code structure (functions, variables, expressions).
Provides a way for the engine to understand what to execute and how.

Ignition (Interpreter)

Converts AST into bytecode (lower-level instructions).
Executes JS code quickly without full compilation, enabling fast startup.

TurboFan (Optimizing Compiler)

Monitors hot code paths (frequently executed functions).
Compiles hot code into highly optimized machine code for faster execution.
Works with feedback from Ignition to optimize performance.

Flow:

JavaScript code â†’ Parser â†’ AST â†’ Ignition â†’ bytecode execution
â†’ TurboFan â†’ optimized machine code

7. Difference Between Memory Heap and Call Stack

Feature	Memory	Heap	Call Stack
Purpose	Stores objects, variables, data dynamically		Manages function execution order
Structure	Unstructured memory		LIFO (Last In, First Out) stack
Allocation	Dynamic memory allocation		Automatic function call management
Example	let obj = {name: "John"} stored in heap		first() â†’ second() execution tracked in stack

8. How V8 Optimizes JavaScript Code

V8 uses multiple techniques:

JIT Compilation â€“ Converts hot code to machine code on the fly.
Inline Caching â€“ Speeds up repeated property access.
Hidden Classes â€“ Assigns internal â€œshapesâ€ to objects for faster property lookup.
Dead Code Elimination â€“ Removes unused code.
Code De-optimization â€“ If assumptions about types fail, V8 can revert and re-optimize.

9. Compilation Phases in V8: Parse, Compile, Execute

Parse

JS code â†’ AST

Syntax validation occurs

Compile

Ignition interprets AST â†’ bytecode
TurboFan optimizes hot code â†’ machine code

Execute

Bytecode runs in Ignition
Optimized machine code runs via TurboFan for frequently used functions

10. Role of Hidden Classes in V8 Optimization

Hidden classes are internal structures V8 creates for objects to track their properties.
They allow fast property access because objects with the same â€œshapeâ€ can share optimized code.

Example:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

let p1 = new Person("John", 30);
let p2 = new Person("Jane", 25);
```

Both p1 and p2 get the same hidden class because their properties are added in the same order.
Accessing p1.name or p2.name is faster than if properties were added in random order.
Key takeaway: Hidden classes enable predictable object layouts, which helps TurboFan generate efficient machine code

----------Advance level----------

11. Output-based Example with var

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

Output:

```
20
20
```

Explanation:

var is function-scoped (not block-scoped).
The x inside the if block overwrites the outer x.
Both console.log statements access the same variable.

12. Output-based Example with let

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

Output:

```
20
```

10

Explanation:

let is block-scoped, so the a inside the if block is a different variable.

Inner a shadows outer a temporarily.
After the block, outer a remains 10.

13. Output-based Example with const Array

```
const arr = [1, 2, 3];
arr.push(4);
console.log(arr);
arr = [5, 6, 7];
console.log(arr);
```

Output:

```
[1, 2, 3, 4]
Uncaught TypeError: Assignment to constant variable.
```

Explanation:

const prevents reassignment of the variable itself.
However, mutable objects like arrays or objects can still be modified.
arr.push(4) works, but arr = [5, 6, 7] throws an error.

14. Variable Shadowing

Variable shadowing occurs when a variable declared in a local scope has the same name as a variable in an outer scope, temporarily â€œhidingâ€ the outer variable.

```
let x = 100; // outer variable

function demo() {
  let x = 50; // shadows outer x
  console.log(x); // 50
}

demo();
console.log(x); // 100
```

Key points:

Shadowing is common with let and const, less problematic with var because it is function-scoped.

Can cause confusion or bugs if inner and outer variables are modified unintentionally.

15. Best Practices for var, let, and const

Keyword	Best Practice
var	Avoid in modern code; use only for legacy code maintenance. Its function-scoping can cause hoisting bugs.
let	Use for mutable variables. Provides block-scoping and prevents accidental overwrites.
const	Default choice for all variables unless reassignment is required. Ensures immutability of bindings. Use with objects/arrays carefully (object contents can still mutate).

Additional Advanced Tips:

Prefer const by default, then let if mutation is necessary.
Avoid var because of hoisting and global scope leaks.
Declare variables at the top of the block for clarity.
Use meaningful names to reduce accidental shadowing.
Combine const with Object.freeze for truly immutable objects if needed.

C.5 Hoisting (20 Questions)

Basic level

---

1. What is Hoisting in JavaScript?

Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their scope during the compilation phase, before code execution.
This allows you to use functions and variables before they are formally declared (with some caveats).

2. Which Declarations Are Hoisted in JavaScript?

Function declarations
Variables declared with var
Variables declared with let and const are also hoisted but behave differently (see below).

3. Are Variables Declared with let and const Hoisted?

Yes, they are hoisted, but:

They are not initialized until the code execution reaches their declaration.
Accessing them before declaration results in a ReferenceError.

4. What Value is Assigned to Hoisted var Variables Before Initialization?

var variables are hoisted and initialized with undefined.

Example:

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

This is why var can be accessed before its declaration, unlike let or const.

5. Difference Between Hoisting of var and let/const

Feature	var	let / const
Hoisted	Yes	Yes
Initialization	undefined	Not initialized
Access before declaration	Allowed (undefined)	ReferenceError
Scope	Function	Block
Temporal Dead Zone (TDZ)	No	Yes

----------Intermediate level----------

6. Creation Phase vs Execution Phase

JavaScript execution happens in two phases:

Creation Phase (Compile time)
JS engine scans code and hoists variables and functions.

Variables are allocated memory:

var â†’ undefined
let / const â†’ uninitialized (TDZ)

Function declarations are stored in memory as actual functions.
Execution Phase (Runtime)
Code runs line by line.
Variables are assigned values, and functions can be invoked.

Illustration:

```
console.log(a); // undefined
var a = 10;
```

During creation: var a is hoisted and set to undefined.
During execution: a = 10 is assigned.

7. Temporal Dead Zone (TDZ)

TDZ is the time between hoisting and actual declaration for let and const.
Accessing the variable in this zone throws ReferenceError.

```
console.log(b); // ReferenceError
let b = 5;
```

Helps prevent using variables before they are properly declared.

8. Hoisting of Function Declarations vs Function Expressions
Feature Function Declaration Function Expression
Hoisted Yes, fully (function body also hoisted) Only the variable name is hoisted (var â†’ undefined, let/const â†’ TDZ)
Example function foo(){} var foo = function(){}

```
foo(); // Works
function foo(){ console.log("Hi"); }

bar(); // Error (bar is undefined)
var bar = function(){ console.log("Hello"); }
```

9. Are Arrow Functions Hoisted?

Arrow functions are function expressions, so they are not hoisted.

Declared with var â†’ variable is hoisted but undefined
Declared with let/const â†’ TDZ applies

```
sayHi(); // ReferenceError
const sayHi = () => console.log("Hi");
```

10. Hoisting Behavior of Class Declarations

Class declarations are hoisted, but like let/const:
They cannot be accessed before initialization.
Accessing a class before declaration results in ReferenceError.

```
const obj = new Person(); // ReferenceError
class Person {
  constructor(name){ this.name = name; }
}
```

This ensures classes are initialized only when their declaration is reached, which avoids potential misuse.
----------Advance level----------
11.

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

âœ… Output:
undefined
5

âœ” Explanation:

During creation phase:

var a; // initialized as undefined

Execution phase:

First log â†’ undefined

Then a = 5

Second log â†’ 5

12.
```
console.log(b);
let b = 10;
```

â□Œ Output:
ReferenceError: Cannot access 'b' before initialization

âœ” Explanation:

let is hoisted but not initialized

It is inside the Temporal Dead Zone (TDZ) until its declaration line

13.
```
test();
function test() {
  console.log("Function Declaration");
}
```

âœ… Output:
Function Declaration

âœ” Explanation:

Function declarations are fully hoisted, including their body.

Equivalent during creation:

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

14.
```
test();
var test = function() {
  console.log("Function Expression");
}
```

â□Œ Output:
TypeError: test is not a function

âœ” Explanation:

Creation phase:

var test = undefined;

Execution:

test() â†’ calling undefined() â†’ TypeError

Function expression assignment hasnâ€™t happened yet

15.

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

âœ… Output:
undefined
20

âœ” Explanation:

Inside foo() creation phase:

var x = undefined;

So execution becomes:

```
console.log(undefined);
x = 20;
console.log(20);
```

Outer x = 10 is shadowed.

16.

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

âŒ Output:
undefined
ReferenceError

âœ” Explanation:

Inside function creation phase:

```
var a = undefined;
let b; // in TDZ
```

Execution:

console.log(a) â†’ undefined

console.log(b) â†’ ReferenceError (TDZ)

17.

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

â Output:
ReferenceError

âœ” Explanation:

Inside block:

let a creates a new block-scoped variable

From block start until declaration â†’ TDZ

So console.log(a) tries to access block a â†’ ReferenceError

Outer a = 10 is shadowed.

18.

```
console.log(typeof myFunc);
var myFunc = function() {};
console.log(typeof myFunc);
```

âœ… Output:
undefined
function

âœ” Explanation:

Creation phase:

var myFunc = undefined;

Execution:

First â†’ typeof undefined â†’ "undefined"

After assignment â†’ "function"

19.

```
foo();
var foo = function() {
  console.log("First");
}
foo();
function foo() {
  console.log("Second");
}
foo();
```

âœ… Output:

```
Second
First
First
```

âœ” Step-by-step Explanation:

Creation phase:

Function declaration is hoisted:

```
function foo() {
  console.log("Second");
}
```

var foo is hoisted but ignored (already declared)

Execution:

foo(); â†’ Calls function declaration â†’ "Second"

foo = function() { console.log("First"); } â†’ Overwrites it

foo(); â†’ "First"

foo(); â†’ "First"

20. Deep Hoisting Analysis

```
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();

ðŸ”□ Creation Phase Breakdown
Global:
var x = 1;
function outer() {...}

Inside outer() creation phase:
var x = undefined;
function inner() {...}

Inside inner() creation phase:
var x = undefined;

ðŸ”Ž Execution Flow
Step 1:
console.log(x);


Inside outer, local x exists â†’ undefined

Step 2:
var x = 2;

Step 3: Call inner()

Inside inner:

console.log(x); // undefined (local x)
var x = 3;
console.log(x); // 3

Step 4: Back to outer
console.log(x); // 2

âœ… Final Output:
undefined
undefined
3
2

--------------------C.6 Functions (20 Questions)-------------------------
1. Different Ways to Define a Function
1ï¸□âƒ£ Function Declaration
function greet() {
  console.log("Hello");
}

2ï¸□âƒ£ Function Expression
const greet = function() {
  console.log("Hello");
```

```
};

3ï¸âƒ£ Arrow Function
const greet = () => {
  console.log("Hello");
};

4ï¸âƒ£ Constructor Function
function Person(name) {
  this.name = name;
}

5ï¸âƒ£ Method inside Object
const obj = {
  greet() {
    console.log("Hello");
  }
};
```

2. Function Declaration vs Function Expression

Feature	Function Declaration	Function Expression
Hoisted	Fully hoisted	Variable hoisted (if var)
Can call before definition	Yes	No
Syntax	function foo(){}	const foo = function(){}

3. Arrow Functions vs Regular Functions

Feature	Regular Function	Arrow Function
this binding	Dynamic	Lexical (inherits from parent)
arguments object	Yes	No
Can be constructor	Yes	No
Syntax	Verbose	Short

Example:

```
const add = (a, b) => a + b;
```

4. Parameters vs Arguments

Parameters â†’ Variables in function definition.

Arguments â†’ Actual values passed during function call.

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

add(2, 3); // arguments
```

5. Default Parameters

Provide default values if argument is missing.

```
function greet(name = "Guest") {
  console.log(name);
}
```

```
greet(); // Guest
```

âœ… Intermediate Level

6. Rest Parameters (...)

Collect multiple arguments into an array.

```
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b);
}

sum(1, 2, 3, 4);
```

âœ” Must be the last parameter.

7. Spread Operator vs Rest Parameter

Both use ... but behave differently:

Spread	Rest
Expands array/object	Collects values into array
Used in calls	Used in function parameters

Spread:

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

Rest:

```
function test(...args) {}
```

8. IIFE (Immediately Invoked Function Expression)

A function that runs immediately after definition.

```
(function() {
  console.log("Runs immediately");
})();
```

Why used?

Avoid polluting global scope

Create private variables

Before ES6 modules, used for encapsulation

9. Callback Function

A function passed as an argument to another function.

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

greet("John", function() {
  console.log("Callback executed");
});
```

Used heavily in:

Events

Async programming

Array methods (map, filter, etc.)

10. Function Hoisting vs Variable Hoisting

Feature	Function Hoisting	Variable Hoisting
Behavior	Entire function hoisted	Only declaration hoisted
Initialization	With function body	undefined
Works before definition	Yes (declaration only)	No meaningful usage

---------- Advanced Level----------
---

11. Hoisting Comparison

Type	Hoisted?	How?
Function Declaration	Fully	Function body available
Function Expression (var)	Partial	Variable = undefined
Function Expression (let/const)	TDZ	ReferenceError
Arrow Function	Same as expression	Not usable before definition

12. Higher-Order Functions

A function that:

Takes another function as argument, OR

Returns a function

Example 1:

```
function operate(a, b, operation) {
  return operation(a, b);
}
```

Example 2:

```
function multiplier(x) {
  return function(y) {
    return x * y;
  };
}
```

Common built-in higher-order functions:

map

filter

reduce

forEach

13. Arrow Functions and this

Regular function:

```
const obj = {
  value: 10,
  show: function() {
    console.log(this.value);
  }
};
```

Arrow function:

```
const obj = {
  value: 10,
  show: () => {
    console.log(this.value);
  }
};
```

Arrow functions:

Do NOT have their own this

Inherit this from surrounding lexical scope

Cannot be used as constructors

14. arguments Object

Regular functions have access to:

```
function test() {
  console.log(arguments);
```

```
}
```

Arrow functions:

```
const test = () => {
  console.log(arguments); // â□Œ Error
};
```

Arrow functions do NOT have arguments.

Use rest parameter instead:

```
const test = (...args) => {
  console.log(args);
};
```

15. Function Currying

Currying transforms a function with multiple arguments into nested functions with single arguments.

Normal function:

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

Curried version:

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

add(2)(3)(4); // 9
```

Why use currying?

Function reusability

Partial application

Cleaner functional programming patterns

----------output based----------

16.

```
function sum(a, b = 5) {
```

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

```
 Output:
15
30
```

b = 5 is a default parameter.

If second argument is missing â†’ default value is used.

Execution:

sum(10) â†’ 10 + 5 = 15

sum(10, 20) â†’ 10 + 20 = 30 (default ignored)

17.

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

```
 Output:
[1, 2, 3, 4, 5]
object
```

...args is a rest parameter.

It collects all arguments into an array.

In JavaScript, arrays are of type "object".

18.

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

```
 Output:
10
ReferenceError: a is not defined
```

This is an IIFE (Immediately Invoked Function Expression).

var a is scoped inside the function.

Outside the function, a does not exist.

Prevents global scope pollution.

19.

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

Output:

15

Arrow function returning a * b.

Implicit return (no {} block).

5 * 3 = 15.

20.

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

Output:

10

C.7 Higher-Order Functions (20 Questions)

1. What is a Higher-Order Function in JavaScript?

A higher-order function (HOF) is a function that:

Takes another function as an argument, OR

Returns another function

Example:

```
function greet(name) {
  return "Hello " + name;
}

function processUser(callback) {
```

```
  return callback("John");
}

processUser(greet);
```

2. What Does It Mean That Functions Are First-Class Citizens?

In JavaScript, functions are first-class citizens, meaning:

They can be assigned to variables

Passed as arguments

Returned from other functions

Stored in objects/arrays

Example:

```
const sayHi = function() {
  console.log("Hi");
};
```

Because functions behave like regular values, higher-order functions are possible.

3. How Does JavaScript Enable Higher-Order Functions?

JavaScript enables HOFs because:

Functions are objects

Functions can be passed as arguments

Functions can return other functions

Closures preserve lexical scope

4. 3 Built-in Higher-Order Functions

These array methods take callback functions:

map()

filter()

reduce()

Example:

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

5. Can a Function Return Another Function?

Yes.

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

const fn = outer();
fn();
```

This is possible due to closures.

Intermediate Level
6. How is map() a Higher-Order Function?

map() takes a callback function and applies it to each element of an array.

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

const doubled = nums.map(function(num) {
  return num * 2;
});
```

It takes a function as input

Returns a new transformed array

7. How is filter() a Higher-Order Function?

filter() takes a callback that returns true or false.

```
const nums = [1,2,3,4];

const even = nums.filter(num => num % 2 === 0);
```

Executes callback for each element
Returns a new filtered array

8. How is reduce() a Higher-Order Function?

reduce() takes a callback and reduces the array to a single value.

```
const nums = [1,2,3,4];
const sum = nums.reduce((acc, curr) => acc + curr, 0);
```

Takes function as argument
Returns accumulated result

9. Higher-Order Function vs Regular Function

Regular Function	Higher-Order Function
Does not accept or return functions	Accepts or returns functions
Performs direct task	Operates on functions

Example Regular:

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

Example HOF:

```
function operate(a, b, operation) {
  return operation(a, b);
}
```

10. Create a HOF That Executes a Function Twice

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

executeTwice(function() {
  console.log("Running");
});
```

Output:

```
Running
Running
```

Advanced Level

11. What is Function Composition?

Function composition means combining multiple functions to produce a new function.

Example:

```
const add2 = x => x + 2;
const multiply3 = x => x * 3;

const compose = (f, g) => x => f(g(x));

const result = compose(add2, multiply3);
console.log(result(5)); // 17
```

Explanation:

multiply3(5) â†’ 15

add2(15) â†’ 17

Higher-order functions enable composition by returning and accepting functions.

12. Benefits of Higher-Order Functions

- Cleaner code
- Better abstraction
- Reusability
- Reduced duplication
- Encourages declarative programming
- Enables functional programming patterns

Example:
Instead of loops:

```
for (...) {}
```

Use:

```
arr.map(...)
```

13. How HOFs Improve Code Reusability

Example:

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

const double = createMultiplier(2);
const triple = createMultiplier(3);
```

We reuse the same logic to create different behaviors.

14. Relationship Between HOFs and Functional Programming

Functional programming relies on:

Pure functions
Immutability
Function composition
Higher-order functions

HOFs are foundational because:

They allow abstraction over behavior
They enable composition
They support declarative style

Example:

```
arr.filter(...).map(...).reduce(...)
```

15. HOF That Returns a Function (Closure Example)

```
function counter() {
  let count = 0;

  return function() {
    count++;
    return count;
  };
}

const increment = counter();

console.log(increment()); // 1
console.log(increment()); // 2
```

16.

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

Output:
Hello
Hello

higherOrder takes a function as an argument.

It executes fn() twice.

So "Hello" prints two times.

17.

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

Output:

10

multiplier(2) returns a function.

That returned function multiplies a number by factor.

double(5) â†’ 5 * 2 = 10

This works because of closure (inner function remembers factor).

18.

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

Output:

```
[2, 4, 6]
```

map() is a higher-order function.

It applies the callback to each element.

Returns a new array with transformed values.

19.

```
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
```

operate receives a function as argument.

Calls it with 5 and 3.

Returns 5 + 3 = 8.

20.

```
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?

A callback function is a function that is passed as an argument to another function and executed later.

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

2. Why Are Callback Functions Used?

Callbacks are used to:

Execute code after another function finishes

Handle asynchronous operations

Create flexible and reusable functions

Implement event-driven programming

3. How Do You Pass a Function as a Callback?

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

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

greet(sayBye);
```

You pass the function without parentheses.

4. Callback Function vs Regular Function

Regular Function	Callback Function
Called directly	Passed to another function

Executes immediately Executes later
Independent Dependent on another function
5. 3 Examples of Functions That Accept Callbacks

setTimeout()

setInterval()

Array methods like:

map()

filter()

forEach()

Example:

```
setTimeout(() => {
  console.log("Delayed");
}, 1000);
```

---------- Intermediate Level----------
---
6. Synchronous vs Asynchronous Callbacks
Synchronous Asynchronous
Executes immediately Executes later
Blocks execution Non-blocking
Runs in order Runs after delay/event
7. Synchronous Callback Example (map())

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

const doubled = nums.map(function(n) {
  return n * 2;
});

console.log(doubled);
```

Callback runs immediately for each element.

Output: [2, 4, 6]

8. Asynchronous Callback Example (setTimeout())

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

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

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

Output:
Start

```
End
Inside Timeout
```

setTimeout callback runs later via the event loop.

9. Passing Arguments to a Callback

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

process(10, function(num) {
  console.log(num * 2);
});
```

You can pass arguments normally when invoking the callback.

10. Anonymous vs Named Callback

Anonymous Callback

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

Named Callback

```
function greet() {
  console.log("Hi");
}

setTimeout(greet, 1000);
```

Anonymous:

Quick usage

Harder to debug

Named:

Better readability

Easier debugging

11. Advantages of Callbacks

Flexibility

Code reusability

Async control

Event-driven behavior

Custom logic injection

12. Callbacks in Event Handling

```
button.addEventListener("click", function() {
  console.log("Button clicked");
});
```

The function executes when the event occurs.

13. Callbacks in Array Methods

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

Each element triggers callback execution.

14. What is Callback Hell?

Nested callbacks inside callbacks:

```
getData(function(a) {
  getMoreData(a, function(b) {
    getEvenMoreData(b, function(c) {
      console.log(c);
    });
  });
});
```

Problems:

Hard to read

Difficult to maintain

Error handling becomes messy

15. How to Avoid Callback Hell

Use Promises

Use async/await

Modularize functions

Use named functions instead of deeply nested anonymous functions

---------- Advanced Level----------

16. How Callbacks Enable Asynchronous Programming

JavaScript is single-threaded.

Callbacks allow:

Non-blocking execution

Delegating async tasks (timers, APIs)

Execution after task completion

Example:

```
fetchData(function(data) {
  console.log(data);
});
```

The callback runs after data arrives.

17. Callbacks and the Event Loop

Flow:

Code runs in Call Stack

Async tasks go to Web APIs

After completion â†’ moved to Callback Queue

Event Loop pushes callback to Call Stack when empty

This is how asynchronous callbacks execute.

18. Error-First Callbacks (Node.js Pattern)

Pattern:

```
function callback(error, data) {}
```

Example:

```
fs.readFile("file.txt", function(err, data) {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

First argument â†’ error

Second argument â†’ result

19. Limitations of Callbacks Compared to Promises

Callback hell

Poor error handling

Hard to chain

No built-in state management

Inversion of control

Promises solve:

Chaining

Centralized error handling

Cleaner async flow

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)

Basic Level

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

Objects are collections of key-value pairs used to store related data.

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

Each key is called a property and values can be strings, numbers, arrays, functions, etc.

2. Difference between dot notation and bracket notation

Dot Notation
person.name

Simple and clean

Property name must be valid identifier

Cannot use variables directly

Bracket Notation
person["name"]

Can use variables

Required for special characters or spaces

```
const key = "age";
person[key]; // 30
```

3. Add, modify, delete properties
Add
```
person.city = "New York";
```

Modify
```
person.age = 35;
```

Delete
```
delete person.city;
```

4. What is the delete operator?

delete removes a property from an object.

```
delete person.age;
```

It:

Removes the property completely

Returns true if deletion succeeds

What is this in object methods?

this refers to the object that is calling the method.

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

```
person.greet(); // John
```

Intermediate Level

6. Different ways to create objects

1. Object Literal

```
const obj = { name: "John" };
```

2. Constructor Function

```
function Person(name) {
  this.name = name;
}
const p1 = new Person("John");
```

3. Object.create()

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

7. Object Destructuring

Extract properties into variables.

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

const { name, age } = person;

console.log(name); // John
```

8.Shallow Copy vs Deep Copy

Shallow Copy

Copies only top-level properties.

```
const obj2 = { ...obj1 };
```

Nested objects are still referenced.

Deep Copy

Creates full independent copy.

```
const deep = JSON.parse(JSON.stringify(obj1));
```

9. Object.keys(), Object.values(), Object.entries()

```
const obj = { a: 1, b: 2 };
```

Object.keys(obj) â†’ ["a", "b"]

Object.values(obj) â†’ [1, 2]

Object.entries(obj) â†’ [["a",1], ["b",2]]

10. How to check if property exists?
 Using in
"name" in person

 Using hasOwnProperty
person.hasOwnProperty("name")

 Advanced Level â€“ Output Based
```
const obj = {
  name: "John",
  age: 30
};
delete obj.age;
console.log(obj);
```

Output:
{ name: "John" }

age is removed.

```
const a = {};
const b = { key: "b" };
const c = { key: "c" };

a[b] = 123;
a[c] = 456;

console.log(a[b]);
```

 Output:
456

Objects used as keys are converted to string "[object Object]".

So:

```
a["[object Object]"] = 123
a["[object Object]"] = 456
```

Second assignment overwrites the first.

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

console.log(a);
console.log(rest);
```

Output:
1

```
{ b: 2, c: 3 }


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

const greet = person.greet;
greet();

Output:
undefined


this is lost because the function is called independently (not as
person.greet()).

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

--------------------------------------------------------------------------------
------------------------------
C.10 Arrays &amp;amp; Array Methods (20 Questions)

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

An array is an ordered collection of values.

const arr = [1, 2, 3, 4];


Or using constructor:

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

2. Difference between map() and forEach()
map() forEach()
Returns a new array    Returns undefined
Used for transformation      Used for side effects
Does not modify original     Does not modify original
arr.map(x => x * 2);      // returns new array
arr.forEach(x => console.log(x)); // just runs function
```

3. What does filter() do?

It creates a new array with elements that pass a condition.

```
arr.filter(x => x > 2);
```

4. What is reduce()?

It reduces an array to a single value.

```
arr.reduce((acc, curr) => acc + curr, 0);
```

acc â†’ accumulator

curr â†’ current value

0 â†’ initial value

5. Difference between push() and pop()

push()	pop()
Adds element at end	Removes last element
Returns new length	Returns removed element

```
arr.push(6);
arr.pop();
```

---------- Intermediate Level----------

6. map() explanation

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

const doubled = numbers.map(num => num * 2);
```

Output:

```
[2, 4, 6]
```

Does NOT modify original array.

7. filter() explanation

```
const numbers = [1, 2, 3, 4];

const even = numbers.filter(num => num % 2 === 0);
```

Output:

```
[2, 4]
```

Returns a new filtered array.

8. reduce() in detail

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

const sum = numbers.reduce((acc, curr) => {
  return acc + curr;
}, 0);
```

Accumulator (acc) â†’ stores result

Current value (curr) â†’ current element

Runs for each element

Flow:

```
0 + 1 = 1
1 + 2 = 3
3 + 3 = 6
```

9. Chaining methods

```
const result = arr
  .map(x => x * 2)
  .filter(x => x > 5)
  .reduce((acc, curr) => acc + curr, 0);
```

Each method returns a new array (except reduce), so chaining works.

10. Difference between map() and reduce()

map()	reduce()
Returns new array	Returns single value
Used for transformation	Used for aggregation
Same length as original	Any type (number, object, array, etc.)

---------- Advanced Level----------

11. Polyfill for map()

```
Array.prototype.myMap = function(callback) {
  const result = [];

  for (let i = 0; i < this.length; i++) {
    if (i in this) {
      result.push(callback(this[i], i, this));
    }
  }

  return result;
};
```

12. Polyfill for filter()

```
Array.prototype.myFilter = function(callback) {
  const result = [];
```

```
  for (let i = 0; i < this.length; i++) {
    if (i in this && callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }

  return result;
};
```

13. Polyfill for reduce()

```
Array.prototype.myReduce = function(callback, initialValue) {
  let acc = initialValue;
  let startIndex = 0;

  if (acc === undefined) {
    acc = this[0];
    startIndex = 1;
  }

  for (let i = startIndex; i < this.length; i++) {
    if (i in this) {
      acc = callback(acc, this[i], i, this);
    }
  }

  return acc;
};
```

14. Time Complexity

map() â†’ O(n)

filter() â†’ O(n)

reduce() â†’ O(n)

They iterate once through the array.

15. How array methods handle empty elements?

They skip empty slots in sparse arrays.

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

arr.map(x => x * 2);
// skips empty index
```

Output Based

16.

```
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]
```

Original array unchanged.

17.

```
const result = arr.filter(x => x % 2 === 0);
```

Output:

```
[2, 4]
```

18.

```
const result = arr.reduce((acc, curr) => acc + curr, 0);
```

Output:

```
15
```

19.

```
const result = arr
.map(x => x * 2)
.filter(x => x > 5)
.reduce((acc, curr) => acc + curr, 0);
```

Step-by-step:

```
map â†’ [2,4,6,8,10]
filter (>5) â†’ [6,8,10]
reduce â†’ 6+8+10 = 24
```

Output:

```
24
```

20.

```
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;
}, {});
```

Step-by-step:

25 â†’ 1
30 â†’ 1
25 â†’ 2

Output:

```
{ 25: 2, 30: 1 }
```

C.11 Destructuring (10 Questions)

Basic Level

1. What is destructuring in JavaScript?

Destructuring is a syntax that allows you to extract values from arrays or properties from objects into separate variables.

It makes code cleaner and more readable.

2. What is array destructuring?

It extracts values based on position.

```
const arr = [10, 20, 30];

const [a, b, c] = arr;

console.log(a); // 10
console.log(b); // 20
```

You can skip elements:

```
const [first, , third] = [1, 2, 3];
console.log(third); // 3
```

3. What is object destructuring?

It extracts properties based on property names.

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

```
const { name, age } = person;

console.log(name); // John
```

4. How to assign default values?

If value is undefined, default is used.

```
Array example:
const [a = 5, b = 10] = [1];

console.log(a); // 1
console.log(b); // 10

Object example:
const { name = "Guest" } = {};
console.log(name); // Guest
```

5. What is the rest operator in destructuring?

... collects remaining elements into an array or object.

```
Array:
const [a, ...rest] = [1, 2, 3, 4];

console.log(rest); // [2, 3, 4]

Object:
const { name, ...others } = {
  name: "John",
  age: 30,
  city: "NY"
};

console.log(others); // { age: 30, city: "NY" }
```

---------- Intermediate & Advanced Level-------
---

6. How to rename variables?

Use : syntax.

```
const person = { name: "Alice", age: 25 };

const { name: userName, age: userAge } = person;

console.log(userName); // Alice
```

7. What is nested destructuring?

Extracting values from nested objects/arrays.

```
const user = {
  name: "John",
  address: {
    city: "Mumbai",
    pin: 400001
  }
};

const {
  address: { city }
} = user;

console.log(city); // Mumbai
```

Array inside object:

```
const data = {
  scores: [10, 20, 30]
};

const {
  scores: [first]
} = data;

console.log(first); // 10
```

8. Swap two variables using destructuring

```
let a = 5;
let b = 10;

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

console.log(a); // 10
console.log(b); // 5
```

No temp variable needed âœ…

9. Destructuring in function parameters

Object example:

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

greet({ name: "John", age: 30 });
```

With default values:

```
function greet({ name = "Guest" } = {}) {
```

```
  console.log(name);
}
```

Array example:

```
function sum([a, b]) {
  return a + b;
}

sum([2, 3]); // 5
```

10. Output-Based Question

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

console.log(a);
console.log(b);
console.log(rest);
```

Output:

```
1
2
[3, 4, 5]
```

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

Basic Level

1. What are the primitive data types in JavaScript?

Primitive types are immutable and stored by value.

There are 7 primitive types:

string

number

bigint

boolean

undefined

null

symbol

Example:

```
let name = "John";      // string
let age = 25;           // number
let isActive = true;    // boolean
```

```
let x;                  // undefined
let y = null;           // null
```

2. What are reference data types?

Reference types store a reference (memory address).

Main reference types:

Object

Array

Function

Date

RegExp

Map / Set

Example:

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

3. What is the typeof operator?

typeof returns the type of a variable as a string.

```
typeof "Hello"   // "string"
typeof 10        // "number"
typeof true      // "boolean"
```

4. Difference between null and undefined

undefined	null
Variable declared but not assigned	Intentional empty value
Default value	Manually assigned
Type: "undefined"	Type: "object" (bug in JS)

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

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

5. What is NaN?

NaN = Not a Number.

Occurs when math operation fails.

```
console.log("abc" / 2); // NaN
```

To check for NaN:

```
Number.isNaN(value);
```

Better than:

```
isNaN(value); // performs coercion
```

---------- Intermediate & Advanced Level---
---

6. What is type coercion?

Type coercion = converting one data type into another.

Implicit Coercion (Automatic)

JavaScript converts types automatically.

```
"5" + 2    // "52"
"5" - 2    // 3
true + 1   // 2
```

Explicit Coercion (Manual)

You convert types yourself.

```
Number("5")   // 5
String(10)    // "10"
Boolean(1)    // true
```

7. Difference between == and ===

```
==    ===
Loose equality   Strict equality
Performs type coercion No type coercion
Compares value only    Compares value + type
1 == "1"   // true
1 === "1"  // false
```

Always prefer === â€¦

8. Truthy and Falsy Values

Falsy values (only 8 in JS):

false

0

-0

0n

"" (empty string)

null

undefined

NaN

Everything else is truthy.

Example:

```
if ("hello") {
  console.log("Truthy");
}
```

10. Output-Based Questions
9.

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

Output:

```
object
undefined
number
object
object
```

Explanation:

typeof null â†’ "object" (historical bug)

NaN is technically a number

Arrays are objects

10.

```
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

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

1. What is the output?

```
console.log([] == false);
```

Output:

true

0 == false â†’ true

2. What is the output?

```
console.log([] === false);
```

Output:

false

3. What is the output?

```
console.log(null == undefined);
console.log(null === undefined);
```

Output:

true
false

4. What is the output?

```
console.log(NaN === NaN);
```

Output:

false

Correct way:

```
Number.isNaN(NaN); // true
```

5. What is the output?
```
console.log(typeof null);
```

Output:

```
object
```

6. What is the output?
```
console.log([] + []);
```

Output:

```
""
```

7. What is the output?
```
console.log([] + {});
```

Output:

```
"[object Object]"
```

```
{} â†’ "[object Object]"
```

8.What is the output?
```
console.log({} + []);
```

Output (in console):

```
0
 {} treated as empty block
+[] â†’ +"" â†’ 0
```

(Behavior may vary depending on context.)

9. What is the output?
```
let a = 10;
(function() {
  console.log(a);
  let a = 20;
})();
```

Output:

ReferenceError

10. What is the output?

```
console.log(1 + "2" + 3);
```

Output:

"123"

Left to right:
1 + "2" â†’ "12"
"12" + 3 â†’ "123"

11. What is the output?

```
console.log("5" - 2);
```

Output:

3

12. What is the output?

```
console.log(true + false);
```

Output:

1

true â†’ 1
false â†’ 0

13. What is the output?

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

Output:

number

14. What is the output?

```
const obj = { a: 1 };
const obj2 = obj;
```

```
obj2.a = 5;
console.log(obj.a);
```

Output:

5

15. What is the output?

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

3
3
3

If let was used:

0
1
2