**HTML**

**Q1) What are the differences between HTML4 and HTML5?**

**Answer:** HTML4 and HTML5 are versions of the Hypertext Markup Language used to create web pages, but HTML5 introduced several improvements and new features to modernize web development.

HTML4 was the standard for many years and supported the basic structure of a webpage using elements like <table>, <div>, and <span>. It relied heavily on external technologies like Flash or JavaScript for multimedia content such as audio and video. It also lacked semantic tags, which made the code less readable and harder for search engines or screen readers to interpret.

HTML5, on the other hand, is a major update that introduced a cleaner and more efficient way to build modern websites. One of the key differences is the inclusion of semantic elements like <header>, <footer>, <article>, and <section>, which provide meaningful structure to the page content. HTML5 also introduced built-in support for multimedia using the <audio> and <video> tags, removing the need for third-party plugins. Additionally, HTML5 supports new input types in forms (like email, date, and range), better error handling, offline storage using localStorage and sessionStorage, and enhanced support for mobile and responsive design.

Overall, HTML5 is more powerful, flexible, and designed to support modern web development needs, while HTML4 is outdated and lacks many of the features that make today’s websites interactive and accessible.

**Q2) What are semantic tags in HTML? Give me some examples.**

**Answer:** Semantic tags in HTML are elements that clearly describe their meaning in a human- and machine-readable way. These tags help structure the content of a webpage and make it easier for developers, browsers, and search engines to understand the layout and purpose of different sections. Unlike non-semantic tags like <div> or <span>, which tell nothing about the content they contain, semantic tags provide meaningful context.

Examples of semantic tags include <header>, which defines the top section or header of a webpage; <nav>, which represents a section with navigation links; <main>, used to wrap the main content of the document; <article>, which defines self-contained content like a blog post or news article; <section>, which represents a thematic grouping of content; <aside>, typically used for sidebars or related content; and <footer>, which indicates the bottom section of the page, often containing contact info or copyright. These tags improve accessibility and search engine optimization by giving structure and meaning to the HTML code.

**Q3) What is the purpose of article, div, section, nav and aside?**

**Answer:** The purpose of article, div, section, nav, and aside tags in HTML is to organize content meaningfully and semantically, helping both developers and browsers understand the structure and intent of a webpage.

The <article> tag is used for self-contained, reusable content that can stand on its own, such as a blog post, news article, forum post, or user comment. It is ideal for content that could be independently distributed or syndicated.

The <div> tag is a non-semantic, generic container used to group elements purely for styling or scripting purposes. It doesn’t carry any inherent meaning about its contents, so it’s mainly used when no other semantic tag fits.

The <section> tag is used to group related content together within a page. It typically has a heading and represents a distinct section of a document, such as a chapter, tab, or feature area. While similar to <div>, it adds semantic meaning to the grouped content.

The <nav> tag defines a section that contains navigation links. This could include a site's main menu, sidebars, or any collection of links used for navigating through the website or application.

The <aside> tag is used for content that is tangentially related to the main content, like sidebars, call-out boxes, or additional information. It is often displayed alongside the main content and helps users access related resources without disrupting the main flow.

Using these tags appropriately enhances accessibility, SEO, and the maintainability of the HTML structure.

**Q4) Why will you use Meta tag?**

**Answer:** The <meta> tag in HTML is used to provide metadata about a web page—information that isn’t displayed on the page itself but is essential for browsers, search engines, and other web services. One primary reason to use a <meta> tag is to define the character encoding of a document, such as <meta charset="UTF-8">, which ensures that special characters render correctly in all browsers.

Another important use of the <meta> tag is for SEO (Search Engine Optimization). You can use it to specify a page’s description (<meta name="description" content="This is a portfolio website.">), which helps search engines understand what the page is about and can influence how it appears in search results. You can also define keywords, author, and viewport settings for responsive design.

For example, the tag <meta name="viewport" content="width=device-width, initial-scale=1.0"> ensures the website is properly scaled on different devices, enhancing mobile responsiveness. In addition, meta tags can be used for controlling page behavior, like refreshing or redirecting after a set time, and for social media sharing with Open Graph tags.

In short, the meta tag improves compatibility, accessibility, SEO, and the overall performance of your webpage on various platforms and devices.

**Q5) What are the differences between inline, inline-block and block elements?**

**Answer:** The main differences between inline, inline-block, and block elements in HTML and CSS relate to how they behave in terms of layout and spacing on a web page.

Inline elements do not start on a new line. They appear on the same line as adjacent elements and only take up as much width as necessary. You cannot set width or height properties on them. Common inline elements include <span>, <a>, and <strong>. For example, a <span> element inside a paragraph will not break the line.

Block elements, on the other hand, always start on a new line and take up the full width available by default. You can apply width, height, margin, and padding to block elements. Examples include <div>, <p>, <h1> to <h6>, and <section>. These elements create a visible structure and layout on the page.

Inline-block elements behave like inline elements in that they do not start on a new line, but unlike regular inline elements, you can set width and height on them. This makes inline-block a useful display option when you want elements to flow inline but still retain box model properties. For example, buttons styled as display: inline-block can sit side-by-side but still have customizable width and height.

In summary, use inline when elements should flow inline without layout control, block when full control and new line structure are needed, and inline-block when you need both inline flow and box-model styling control.

**Q6) What are properties and attributes in HTML?**

**Answer:** In HTML, attributes and properties are related concepts but are used in different contexts—attributes are part of the HTML markup, while properties are part of the DOM (Document Object Model) once the HTML is parsed by the browser.

Attributes are defined in the HTML code and provide additional information about an element. They are written inside the opening tag of an element and always consist of a name and a value (e.g., class="header", id="main", href="https://example.com"). Attributes set the initial state of an element when the page is first loaded.

Properties, on the other hand, are the representation of those attributes in JavaScript once the HTML has been loaded into the DOM. For example, when an element like <input type="text" value="Hello"> is parsed, the value attribute becomes a value property in JavaScript. You can access or change this using JavaScript code like inputElement.value = "New Text".

One key distinction is that changing an attribute using setAttribute() doesn’t always change the property, and vice versa. For example, if you change the value attribute of an input field using JavaScript, it won’t change the currently displayed value unless you also update the value property.

In short, attributes exist in the HTML markup, while properties exist on DOM elements in JavaScript and reflect the current state of those elements.

**Q7) What is a viewport ?**

**Answer:** The viewport is the visible area of a web page within the browser window that a user can see without scrolling. It changes depending on the device being used to view the web page—on a desktop computer, the viewport is usually wide and landscape-oriented, while on a mobile device, it is narrower and typically portrait-oriented.

In modern responsive web design, the viewport is extremely important because it affects how content is displayed on different devices. Developers use the viewport meta tag in HTML to control the layout and scaling of the page. For example, this line of code is commonly added in the **<head>** section of an HTML document:

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

This tag tells the browser to set the width of the page to the device’s screen width and to set the initial zoom level to 1. Without this meta tag, a mobile browser may render the page at a desktop width and scale it down, making the content look very small and hard to read.

In summary, the viewport is the portion of the web page that is currently visible to the user, and controlling it is crucial for creating responsive and user-friendly web designs across different screen sizes.

**Q8) What is hyperlink in html? What tag and attribute will you use for hyperlink?**

**Answer:** A **hyperlink** in HTML is a link that allows users to navigate from one page to another or from one part of a page to another. Hyperlinks are one of the fundamental features of the web, enabling users to move between documents, websites, or even different sections within the same page.

In HTML, the <a> tag (short for "anchor") is used to create a hyperlink. The href attribute (short for "hypertext reference") is used to specify the destination of the link. Here's an example:

<a href="https://www.example.com">Visit Example</a>

In this example, the text "Visit Example" becomes clickable. When a user clicks on it, they will be directed to https://www.example.com.

If you want the link to open in a new tab, you can also add the target="\_blank" attribute like this:

<a href="https://www.example.com" target="\_blank">Visit Example</a>

In summary, to create a hyperlink in HTML, you use the <a> tag along with the href attribute to define the destination. Optionally, the target attribute can be used to control how the link opens.

**Q9) What is the difference between HTML elements and tags?**

**Answer:**

In HTML, **tags** and **elements** are closely related but have different meanings.

An **HTML tag** is the markup code enclosed in angle brackets that tells the browser how to display content. Tags usually come in pairs: an opening tag and a closing tag. For example:

<p>This is a paragraph.</p>

Here, **<p>** is the opening tag, and **</p>** is the closing tag.

An **HTML element**, on the other hand, is the complete structure that includes the opening tag, the content inside it, and the closing tag. In the example above, the element is:

<p>This is a paragraph.</p>

So, the **tag** is just the part like **<p>** or **</p>**, while the **element** refers to the whole set that includes tags and the content.

In short, **tags are the building blocks**, and when combined with content, they form an **HTML element**.

**Q10) How to align text in HTML?**

**Answer:**

To align text in HTML, you typically use the text-align property in CSS, as HTML alone does not have direct attributes for modern text alignment. Here's how it works:

To align text to the **left**, **center**, or **right**, you can apply the text-align property to the container element (like a <div>, <p>, or <section>). For example:

<p style="text-align: left;">This text is left aligned.</p>

<p style="text-align: center;">This text is center aligned.</p>

<p style="text-align: right;">This text is right aligned.</p>

Alternatively, for better styling and maintainability, it’s recommended to use an external or internal CSS block:

<style>

.center-text {

text-align: center;

}

</style>

<p class="center-text">This text is center aligned using CSS class.</p>

While the deprecated align attribute was used in older HTML versions (e.g., <p align="center">), it is no longer recommended in HTML5. CSS is the standard way to control text alignment in modern web Development.

**Q11) Explain the use of an iframe tag.**

**Answer:** The <iframe> tag in HTML is used to embed another HTML document within the current document. It creates an inline frame, allowing you to load another webpage or content (such as videos, maps, or other resources) into the webpage. This is particularly useful for integrating third-party content or displaying external resources without navigating away from the main page.

Here’s how an <iframe> works:

<iframe src="https://www.example.com" width="600" height="400" title="Example Website"></iframe>

In the example above:

* src: Specifies the URL of the page you want to embed.
* width: Defines the width of the iframe.
* height: Specifies the height of the iframe.
* title: Provides a title for the iframe, which improves accessibility.

You can also use additional attributes like frameborder (to control the border appearance), allowfullscreen (to allow full-screen mode for embedded content like videos), and sandbox (for restricting the iframe's capabilities for security reasons).

For example, embedding a YouTube video:

<iframe width="560" height="315" src="https://www.youtube.com/embed/dQw4w9WgXcQ" frameborder="0" allowfullscreen></iframe>

This allows you to show external content while maintaining the integrity of your webpage’s design and layout. However, using iframes requires caution regarding security risks, such as clickjacking, and should be used with proper precautions, such as the sandbox attribute.

**Q12) Have you used Audio and Video tags? How do they work?**

**Answer:** Yes, the <audio> and <video> tags in HTML are used to embed audio and video files on a webpage, respectively. These tags allow web developers to provide multimedia content directly within the page without relying on third-party plugins like Flash. Below is an explanation of both tags and how they work:

<audio> **Tag:**

The <audio> tag is used to embed sound files such as MP3, Ogg, or WAV formats. It allows users to listen to audio directly within the webpage. You can provide playback controls like play, pause, volume, and seek bar using the controls attribute.

Example:

<audio controls>

<source src="audiofile.mp3" type="audio/mpeg">

Your browser does not support the audio element.

</audio>

Explanation:

* controls: This attribute adds the audio controls, allowing users to control playback (play, pause, volume).
* <source>: Specifies the source of the audio file. You can provide multiple <source> tags for different formats (MP3, Ogg, etc.) to ensure compatibility across different browsers.

**<video> Tag:**

The <video> tag is used to embed video files in the webpage. Similar to the <audio> tag, the <video> tag allows the inclusion of video content, and the controls attribute can be used to provide playback control.

Example:

<video controls width="600">

<source src="video.mp4" type="video/mp4">

<source src="video.ogg" type="video/ogg">

Your browser does not support the video tag.

</video>

Explanation:

* controls: Adds the video controls for the user (play, pause, volume, fullscreen).
* width: Specifies the width of the video player. You can also use height to define the height.
* <source>: Specifies the video source. You can add multiple <source> elements to support different formats for cross-browser compatibility.

Both <audio> and <video> tags also support additional attributes like:

* autoplay: Starts playing the media automatically when the page loads.
* loop: Loops the media playback indefinitely.
* muted: Mutes the audio or video by default.

**Example for video with autoplay and loop:**

<video controls autoplay loop>

<source src="movie.mp4" type="video/mp4">

<source src="movie.ogv" type="video/ogg">

Your browser does not support the video tag.

</video>

Using these HTML tags, you can provide multimedia content that plays directly within the webpage without relying on external plugins.

**Q13) What is the Difference between strong , b, em, i tags?**

**Answer:** In HTML, the <strong>, <b>, <em>, and <i> tags are used to style text and emphasize content, but they have different semantic meanings and uses.

<strong> **vs** <b>**:**

* <strong>: This tag is used to indicate that the text is of strong importance or emphasis. It has a semantic meaning, meaning it adds meaning to the content in the context of the webpage. Browsers typically render text inside a <strong> tag as bold, but the main purpose of this tag is to convey the significance of the content, not just its appearance.

Example:

<strong>This is important text</strong>

* <b>: The <b> tag is used to apply bold styling to text but does not add any semantic meaning to the content. It is purely for visual styling, making the text appear bold, but it doesn't imply that the content is important or carries any special significance.

Example:

<b>This text is bold</b>

<em> **vs** <i>**:**

* <em>: The <em> tag is used to emphasize a piece of text, indicating that it should be stressed in some way, typically with an italicized font. The semantic meaning of <em> is that the content inside it should be stressed or given special emphasis. Browsers usually render this as italics, but the key point is the meaning of the text, not just the style.

Example:

<em>This is emphasized text</em>

* <i>: The <i> tag is used for text that is typically displayed in italics but does not carry any special semantic meaning. It is generally used to style text in a way that distinguishes it, like for technical terms, foreign words, or names of books, etc., but it doesn't indicate emphasis or importance in the same way <em> does.

Example:

<i>This text is italicized</i>

**Summary of Differences:**

* **<strong>** is used to indicate important text, with semantic meaning, and is typically rendered as bold.
* **<b>** is used for bold styling but doesn't imply any importance or emphasis.
* **<em>** is used to emphasize text with semantic meaning, typically rendered as italicized text.
* **<i>** is used to apply italics for styling purposes, without conveying any emphasis or significance.

In conclusion, use **<strong>** and <**em**> for emphasizing text with meaning, and use <b> and <i> for purely visual styling.

**Q14) How will center a div?**

**Answer:** To center a <div> in HTML and CSS, you can use different methods depending on the situation. Here are a few common techniques:

**1. Using Flexbox:**

Flexbox is a modern and responsive method to center a <div> both vertically and horizontally.

<div class="container">

<div class="centered-div">I am centered</div>

</div>

<style>

.container {

display: flex;

justify-content: center; /\* Horizontal centering \*/

align-items: center; /\* Vertical centering \*/

height: 100vh; /\* Make the container take full viewport height \*/

}

.centered-div {

width: 300px;

height: 200px;

background-color: lightblue;

}

</style>

**2. Using Grid:**

CSS Grid is another powerful layout tool that allows you to center elements both horizontally and vertically.

<div class="container">

<div class="centered-div">I am centered</div>

</div>

<style>

.container {

display: grid;

place-items: center; /\* Centers both horizontally and vertically \*/

height: 100vh;

}

.centered-div {

width: 300px;

height: 200px;

background-color: lightgreen;

}

</style>

**3. Using Margin Auto (for horizontal centering):**

If you want to center the <div> horizontally but keep it fixed in size, you can use margin: auto.

<div class="centered-div">I am centered</div>

<style>

.centered-div {

width: 300px;

height: 200px;

background-color: lightcoral;

margin: 0 auto; /\* Horizontally centers the div \*/

}

</style>

To center the element both horizontally and vertically, you'll need a wrapper element with a fixed height.

**4. Using Absolute Positioning:**

You can also use absolute positioning to center a <div>.

<div class="container">

<div class="centered-div">I am centered</div>

</div>

<style>

.container {

position: relative;

height: 100vh;

}

.centered-div {

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%, -50%); /\* Adjusts the div to be exactly in the center \*/

width: 300px;

height: 200px;

background-color: lightgoldenrodyellow;

}

</style>

**Summary:**

* **Flexbox** and **Grid** are the most modern, flexible, and responsive methods for centering elements.
* **Margin: auto** works for horizontal centering when the element has a fixed width.
* **Absolute positioning** with transform can be used for precise centering.

**Q15) What is HTML?**

**Answer:** HTML (HyperText Markup Language) is the standard language used to create and design web pages. It provides the structure for a webpage by using a system of tags and attributes to define the content and its layout. HTML is the foundation of all web content and works alongside CSS (Cascading Style Sheets) and JavaScript to create fully functional and interactive websites.

HTML elements are represented by tags, such as <div>, <p>, <h1>, and <a>, which define different sections of the page. These elements can contain other elements or text and are used to structure content logically. For instance, headings, paragraphs, images, and links are all defined using specific HTML tags.

HTML documents are usually saved with the .html extension, and when rendered by a web browser, they display the content as part of the webpage. HTML is essential for creating web pages because it allows browsers to interpret and display information to users in an organized, readable format.

In short, HTML is the backbone of web development, providing the necessary structure to build websites and web applications.

**Q16) What is charset in html? Why will you use it?**

Answer: In HTML, charset refers to the character encoding used to represent text on a webpage. It specifies the set of characters that can be used in the document and determines how characters are stored and interpreted by the browser. The charset attribute is typically specified within the <meta> tag in the <head> section of an HTML document.

The most common character encoding used is UTF-8, which includes most of the characters from every language, symbols, and special characters. UTF-8 is the standard encoding for web content because it supports a wide range of characters and is backward-compatible with ASCII.

Here’s an example of how to define the charset in HTML:

<meta charset="UTF-8">

Why will you use it?

1. Support for Multiple Languages: Using UTF-8 allows your webpage to correctly display text in various languages, including special characters, accents, and symbols.
2. Prevents Encoding Issues: Without specifying the charset, the browser may misinterpret character encoding, leading to garbled text or strange symbols on the page.
3. Ensures Compatibility: UTF-8 is the most widely supported encoding across all modern browsers and devices, ensuring that your page will display correctly for all users.

In summary, specifying the charset in HTML is important to ensure the correct rendering of text and symbols across different languages and platforms, providing a smooth user experience.

Q17) Tell me some feature names of HTML-5.

Answer: HTML5 comes with many powerful features that enhance the capabilities of web applications. Some key features of HTML5 include:

1. **New Structural Elements**: HTML5 introduced semantic elements that improve the structure of webpages and make them more readable, such as <header>, <footer>, <section>, <article>, <nav>, and <aside>.
2. **Audio and Video Elements**: HTML5 added native support for embedding audio and video directly into webpages using the <audio> and <video> elements. This eliminates the need for external plugins like Flash.
3. **Canvas Element**: The <canvas> element allows you to draw graphics, animations, and interactive elements directly in the browser without the need for plugins or third-party libraries.
4. **Local Storage and Session Storage**: HTML5 provides mechanisms to store data in the browser through localStorage and sessionStorage. These allow you to store data persistently on the client-side and manage sessions for the user.
5. **Geolocation API**: HTML5 includes the Geolocation API, which allows web pages to access the user's geographical location (with their permission). This is commonly used in location-based services like maps and location tracking.
6. **Web Workers**: Web Workers allow for multi-threading in web applications, enabling background scripts to run concurrently with the main page thread. This helps to improve the performance of web applications, especially when dealing with heavy calculations.
7. **WebSockets**: WebSockets enable real-time, bidirectional communication between a server and a client over a single, persistent connection, facilitating faster and more efficient data exchange.
8. **Form Enhancements**: HTML5 introduced new input types like email, date, range, and url, along with new attributes like placeholder and required, to improve the user experience and simplify form validation.
9. **Offline Web Applications**: HTML5 allows the creation of offline-capable web applications by using the Application Cache and the Service Worker API, which enable web apps to run even when there’s no internet connection.
10. **New APIs**: HTML5 comes with several new APIs to extend functionality, including:

* **Drag-and-Drop API**: Allows users to drag and drop elements within the browser.
* **Notifications API**: Enables websites to show desktop notifications to users.
* **History API**: Allows manipulation of the browser's history without reloading the page.
* **File API**: Enables reading and writing files directly from the user's file system.

1. **Improved Syntax and Support for Mobile Devices**: HTML5 improves the structure of the code, with cleaner syntax and better support for mobile devices, making it more responsive and adaptive for various screen sizes.

These features provide significant improvements to web development by enabling better performance, user interactivity, and richer media experiences, as well as simplifying the development process.

4o mini

**Q18) What is a variable?**

**Answer:** A variable is a fundamental concept in programming that acts as a named container for storing data, allowing programmers to reference and manipulate information dynamically throughout their code. It consists of an identifier (name) and a value, which can change during execution (unless declared as a constant). Variables enable flexibility in coding by holding different types of data—such as numbers, strings, or objects—and are subject to scope rules that determine where they can be accessed. For example, in JavaScript, *let age = 25;* declares a variable *age* with the value *25*, which can later be updated or used in calculations.

**Q19) What are compilers and interpreters? And what is the difference between them?**

**Answer:** Both **compilers** and **interpreters** are tools that convert high-level programming languages (like JavaScript, C++, or Python) into machine code so that the computer can understand and execute them. However, they work in different ways.

**🔸 Compiler:**

A **compiler** translates the entire source code of a program **at once** into machine code or an intermediate code **before execution**.

* ✅ Fast execution after compilation.
* ❌ Errors are shown only after the entire program is compiled.
* 📦 Example languages: C, C++, Java (compiled to bytecode), Go.

**🔸 Interpreter:**

An **interpreter** reads and **executes the code line-by-line** or statement-by-statement at runtime.

* ✅ Easier to debug as it stops at the first error.
* ❌ Slower than compiled languages due to on-the-fly execution.
* 📦 Example languages: Python, JavaScript, Ruby.

🔹 Key Differences:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Feature | |  | | --- | |  |  |  | | --- | | **Compiler** | | |  | | --- | |  |  |  | | --- | | **Interpreter** | |
| Translation | Entire code at once | Line-by-line |
| Speed | Faster after compilation | Slower due to real-time execution |
| Error Reporting | |  | | --- | |  |  |  | | --- | | Shows all errors after compiling | | |  | | --- | |  |  |  | | --- | | Stops at the first error | |
| Output | Generates an executable file | |  | | --- | |  |  |  | | --- | | Does not produce a file | |
| Examples | C, C++, Java (partially) | JavaScript, Python, Ruby |

**Q19) What is scope? And what are block scope, global scope, and function scope?**

**Answer:** **Scope** refers to the context in which variables are accessible or visible in your code. It defines **where you can use or access a variable**.

JavaScript has three main types of scope:

**🔹 1. Global Scope**

* A variable declared **outside** of any function or block has **global scope**.
* It can be accessed **anywhere** in the code (even inside functions or blocks).

let globalVar = "I'm global";

function show() {

console.log(globalVar); // Accessible

}

show();

console.log(globalVar); // Also accessible here

**🔹 2. Function Scope**

* Variables declared **inside a function** using var, let, or const are **function scoped**.
* These variables are only accessible **within that function**.

function greet() {

let message = "Hello";

console.log(message); // Works

}

greet();

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

**🔹 3. Block Scope**

* Variables declared with let and const inside a **block {}** (like if, for, while) are **block scoped**.
* They are accessible **only inside that block**.

if (true) {

let blockVar = "I'm inside a block";

console.log(blockVar); // Works

}

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

🔸Note: var is not block scoped, it is function scoped. That’s why let and const are preferred in modern JavaScript for better scoping and cleaner code.

✅ Summary:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Scope Type | |  | | --- | |  |  |  | | --- | | **Defined In** | | |  | | --- | |  |  |  | | --- | | **Accessible In** | |
| Global | |  | | --- | |  |  |  | | --- | | Outside functions/blocks | | Line-by-line |
| Function | |  | | --- | |  |  |  | | --- | | Inside a function | | Slower due to real-time execution |
| Block | |  |  |  | | --- | --- | --- | | |  | | --- | |  |  |  | | --- | | Inside {} with let/const | |  |  | | --- | |  | | |  | | --- | |  |  |  | | --- | | Stops at the first error | |

**Q21) Explain find() vs filter().**

**Answer:** In JavaScript, both *find()* and *filter()* are array methods used to search through arrays, but they behave differently based on what you need.

* **find()**  
  The find() method searches an array and **returns the first element** that satisfies the given condition. If no matching element is found, it returns undefined. It stops searching as soon as it finds a match, making it more efficient when you only need a single result.
* **filter()**  
  The filter() method searches an array and **returns a new array** containing **all elements** that satisfy the condition. If no elements match, it returns an **empty array**. It goes through the **entire array** even after finding a match.

**Code Example:**

const numbers = [5, 12, 8, 130, 44];

// Using find()

const found = numbers.find(number => number > 10);

console.log(found); // Output: 12

// Using filter()

const filtered = numbers.filter(number => number > 10);

console.log(filtered); // Output: [12, 130, 44]

**Summary:**

* *find()* → returns **one** value (the first match).
* *filter()* → returns **an array** of **all matches**.

**Q22) Explain map() vs forEach().**

**Answer:** In JavaScript, both *map()* and *forEach()* are used to loop through arrays, but they have different purposes and outputs.

* **forEach()**  
  The *forEach()* method executes a provided function **once for each array element**. It is mainly used when you want to perform an action (like console logging or updating something) for every item. **It does not return anything** (returns undefined by default).
* **map()**  
  The *map()* method also goes through each element in an array, but it **returns a new array** containing the results of applying a function to every element. You use *map()* when you want to **transform** the original array into a new one.

**Code Example:**

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

// Using forEach()

numbers.forEach(num => {

console.log(num \* 2);

});

// Output:

// 2

// 4

// 6

// 8

// forEach() does not return a new array.

// Using map()

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

console.log(doubled);

// Output: [2, 4, 6, 8]

// map() returns a new array with transformed values.

**Summary:**

* *forEach()* → used for side effects (like printing, updating DOM), **no return value**.
* *map()* → used for transforming data, **returns a new array**.

**Q23) What is a function in Javascript ?**

**Answer:** In JavaScript, a **function** is a reusable block of code that performs a specific task or calculates a value. Functions help to organize code, avoid repetition, and make programs easier to understand and maintain. Instead of writing the same code multiple times, you can write it once inside a function and "call" the function whenever you need it.

A function can take inputs, called **parameters**, and can also return an output using the **return** statement. Functions can be built-in (like *alert(),* *prompt()*, etc.) or user-defined.

**Syntax of a Function:**

// Function declaration

function greet(name) {

return "Hello, " + name + "!";

}

// Calling the function

const message = greet("John");

console.log(message); // Output: Hello, John!

**Key Points:**

* Functions can accept parameters (inputs).
* Functions can return values using the *return* keyword.
* Functions can be declared or stored inside variables (function expressions).
* Functions improve code reusability and structure.

**Q24) Explain Function declaration vs function expression.**

**Answer:** In JavaScript, both **function declarations** and **function expressions** are used to create functions, but there are important differences between them.

**Function Declaration** is when you define a function using the function keyword, followed by the function name. One important feature is that function declarations are **hoisted** to the top of their scope, meaning you can call the function even before it's defined in the code.

**Example of Function Declaration:**

greet(); // Works because of hoisting

function greet() {

console.log("Hello from a function declaration!");

}

**Function Expression** is when you assign a function to a variable. Here, the function can be anonymous (no name) or named. Function expressions are **not hoisted**, which means you must define the function before calling it.

**Example of Function Expression:**

const greet = function() {

console.log("Hello from a function expression!");

};

greet(); // Must call after defining

**Key Differences:**

* **Hoisting:** Function declarations are hoisted; function expressions are not.
* **Usage:** Function declarations can be called earlier in the code, while function expressions must be defined first.
* **Flexibility:** Function expressions can be anonymous and are often used in callbacks or when passing functions as arguments.

**Q25) What are the break and continue statements?**

**Answer:** In JavaScript, the **break** and **continue** statements are used to control the flow of loops like for, while, and do-while.

* The **break** statement immediately stops the entire loop. When break is encountered, the loop ends, and the program continues with the next line of code after the loop.
* The **continue** statement skips the current iteration of the loop and moves directly to the next iteration, without exiting the loop completely.

**Example of break:**

for (let i = 1; i <= 5; i++) {

if (i === 3) {

break; // Stops the loop when i is 3

}

console.log(i);

}

// Output: 1, 2

**Example of continue:**

for (let i = 1; i <= 5; i++) {

if (i === 3) {

continue; // Skips when i is 3

}

console.log(i);

}

// Output: 1, 2, 4, 5

**Summary:**

* Use **break** when you want to exit the loop completely based on a condition.
* Use **continue** when you want to skip a particular iteration and continue looping.

**Q26) What is the global variable?**

**Answer:** A **global variable** in JavaScript is a variable that is declared outside of any function, block, or module. This means it can be accessed and modified from anywhere in the program, whether inside functions or outside them. Global variables are stored in the global execution context, which in browsers is the window object. While they can be convenient for sharing data across different parts of a program, overusing global variables can make code harder to manage, debug, and maintain, because any part of the code can change their value.

**Example of a global variable:**

let globalVar = "I am global!";

function display() {

console.log(globalVar); // Accessible inside the function

}

display();

console.log(globalVar); // Accessible outside as well

**Important Note:**  
If you accidentally assign a value to a variable without using let, const, or var, it automatically becomes a global variable, which can cause bugs.

**Example of accidental global:**

function createGlobal() {

accidentalGlobal = "Oops!"; // No 'let', 'const', or 'var'

}

createGlobal();

console.log(accidentalGlobal); // Still accessible globally

**Q27) How to handle an asynchronous program?**

**Answer:** In JavaScript, asynchronous programming is important for handling operations like network requests, file reading, or timers without blocking the main thread. There are several ways to manage asynchronous code effectively:

1. **Callbacks:**  
   A callback is a function passed into another function to be executed later. It is one of the oldest ways to handle asynchronous tasks. However, too many nested callbacks can create "callback hell," making code messy.

**Example:**

function fetchData(callback) {

setTimeout(() => {

callback("Data loaded");

}, 1000);

}

fetchData((data) => {

console.log(data);

});

1. **Promises:**  
   Promises provide a cleaner way to handle asynchronous operations. They represent a value that may be available now, later, or never. Promises have .then() for success and .catch() for error handling.

**Example:**

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded");

}, 1000);

});

};

fetchData()

.then((data) => console.log(data))

.catch((error) => console.log(error));

1. **Async/Await:**  
   async/await is a modern and cleaner way to work with Promises. It makes asynchronous code look more like synchronous code, improving readability.

**Example:**

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded");

}, 1000);

});

};

fetchData()

.then((data) => console.log(data))

.catch((error) => console.log(error));

**Summary: Callbacks vs Promises vs Async/Await**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Callbacks | Promises | Async/Await |
| Style | **Function passed as an argument** | **Object representing future completion** | **Cleaner syntax on top of Promises** |
| Readability | **Can become messy (callback hell)** | **Better than callbacks** | **Very clean and easy to read** |
| Error Handling | **Needs manual error handling** | **.catch() method for errors** | **try...catch block for error handling** |
| Chaining Operations | **Difficult** | **Easy with .then() chaining** | **Looks like normal synchronous code** |
| Introduced In | **Early JavaScript** | **ES6 (2015)** | **ES8 (2017)** |

**Q28) What is a variable?**

**Answer:** A variable

**Javascript 2nd Topic: ES6**

**Q1) Tell me about Es6. Or what ES6 features did you use?**

**Answer:** ES6, also known as ECMAScript 2015, introduced a lot of important features that made JavaScript more powerful, cleaner, and easier to write. It brought improvements for both frontend and backend development. Some important ES6 features I commonly use include:

* **let and const**:  
  Instead of using only var, now we use let for block-scoped variables and const for variables that should not be reassigned.
* **Arrow Functions**:  
  A shorter way to write functions. Example:

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

* **Template Literals**:  
  Easier string formatting using backticks ( `` ) and ${} syntax.  
  Example:

const name = "John";

console.log(`Hello, ${name}!`);

* **Destructuring**:  
  Easily extract values from arrays or objects.  
  Example:

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

const { name, age } = person;

* **Default Parameters**:  
  Set default values for function parameters.  
  Example:

function greet(name = "Guest") {

console.log(`Hello, ${name}!`);

}

* **Spread and Rest Operators**:  
  Spread (...) is used to expand arrays/objects, and Rest (...) is used to collect multiple elements.  
  Example (Spread):

const arr1 = [1, 2];

const arr2 = [...arr1, 3, 4];

* **Promises**:  
  Handle asynchronous operations more easily than callbacks.  
  Example:

const promise = new Promise((resolve, reject) => {

resolve("Success!");

});

promise.then(data => console.log(data));

* **Modules (import/export)**:  
  Break the code into reusable pieces across files.  
  Example:

// file.js

export const greet = () => console.log("Hello!");

// main.js

import { greet } from './file.js';

greet();

These ES6 features made my code more organized, readable, and efficient while building different full-stack projects.

**Q2) What are the differences between *var*, *let*, and *const*?**

**Answer:** In JavaScript, *var*, *let*, and *const* are used for declaring variables, but they have significant differences in terms of scope, hoisting, and mutability:

1. ***var****:*
   * **Scope**: *var* is function-scoped or globally scoped (if declared outside a function).
   * **Hoisting**: Variables declared with *var* are hoisted to the top of their scope, meaning they can be used before they are declared, but they are initialized as undefined.
   * **Re-declaration**: *var* allows re-declaration of the same variable in the same scope, which can lead to bugs and unexpected behavior.
   * **Mutability**: Variables declared with *var* can be re-assigned.

Example:

var name = "Alice";

console.log(name); // "Alice"

var name = "Bob"; // Re-declaration is allowed

console.log(name); // "Bob"

1. ***let:***
   * **Scope**: *let* is block-scoped, meaning it is only available within the block of code (such as inside a loop or if statement) where it was declared.
   * **Hoisting**: Variables declared with *let* are also hoisted, but they are not initialized. Accessing them before the declaration leads to a ReferenceError (due to the "temporal dead zone").
   * **Re-declaration**: *let* does not allow re-declaring the same variable within the same scope.
   * **Mutability**: Variables declared with *let* can be re-assigned.

Example:

let age = 25;

console.log(age); // 25

age = 30; // Re-assignment is allowed

console.log(age); // 30

1. **const**:
   * **Scope**: Like *let*, *const* is block-scoped.
   * **Hoisting**: *const* is also hoisted, but it is not initialized until its declaration is reached, so accessing it before declaration results in a ReferenceError.
   * **Re-declaration**: *const* does not allow re-declaration of the same variable within the same scope.
   * **Mutability**: Variables declared with *const* cannot be re-assigned, making them immutable. However, if a const variable holds an object or array, its properties/elements can still be modified (because the object/array itself is not immutable).

Example:

const pi = 3.14;

console.log(pi); // 3.14

// pi = 3.14159; // Error: Assignment to constant variable

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

person.age = 26; // Allowed: modifying properties

console.log(person.age); // 26

**Summary of Key Differences:**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | var | let | const |
| Scope | Function or global | Block | Block |
| Hoisting | Hoisted (initialized as undefined) | Hoisted (not initialized) | Hoisted (not initialized) |
| Re-declaration | Allowed | Not allowed | Not allowed |
| Re-assignment | Allowed | Allowed | Not allowed |

**Q3) Why would you use default parameters?**

**Answer:** In JavaScript, default parameters are used to set default values for function parameters when no value or *undefined* is passed. This feature allows you to write cleaner, more readable, and more maintainable code by providing default values that ensure the function behaves as expected even when arguments are missing.

1. **Avoiding *undefined* checks**: Without default parameters, you would need to check if a parameter is *undefined* and assign a default value manually, which can make the code more complex. With default parameters, this process becomes automatic.
2. **Simplifying Function Calls**: It allows you to call functions with fewer arguments, and the function will still work by using default values when necessary.
3. **Improves Code Readability**: Default parameters make it clear what the expected behavior is when no arguments are passed to a function.

**Example of Default Parameters:**

function greet(name = "Guest", age = 18) {

console.log(`Hello, ${name}! You are ${age} years old.`);

}

greet(); // "Hello, Guest! You are 18 years old."

greet("Alice"); // "Hello, Alice! You are 18 years old."

greet("Bob", 25); // "Hello, Bob! You are 25 years old."

**Key Points:**

* Default parameters are used when you want a parameter to have a fallback value if no argument is provided.
* Default parameters only apply if the argument passed is *undefined*, not if it's null or any other falsy value.
* You can use expressions or function calls as default values as well.

**Example with an expression as the default value:**

function calculatePrice(price, discount = price \* 0.1) {

console.log(`Final price: $${price - discount}`);

}

calculatePrice(100); // Final price: $90 (using default discount)

calculatePrice(100, 20); // Final price: $80 (using provided discount)

**Conclusion:**

Using default parameters enhances the function's flexibility, reduces the need for manual checks, and helps in keeping the code simple and clean.

**Q4) How does the Spread operator work?**

**Answer:** The **spread operator** (*...*) in JavaScript is a powerful tool that allows you to expand or spread elements of an iterable (such as an array or object) into individual elements. It is primarily used to copy or merge arrays and objects, making code more concise and easier to work with.

**Key Uses of the Spread Operator:**

1. **Copying Arrays**: The spread operator can be used to create a shallow copy of an array, which helps in avoiding the issues of direct reference copying.
   * **Example**:

const arr1 = [1, 2, 3];

const arr2 = [...arr1]; // arr2 is a copy of arr1

arr2.push(4);

console.log(arr1); // [1, 2, 3]

console.log(arr2); // [1, 2, 3, 4]

1. **Merging Arrays**: You can merge multiple arrays into one array with the spread operator.
   * **Example**:

const arr1 = [1, 2];

const arr2 = [3, 4];

const mergedArr = [...arr1, ...arr2]; // Merges arr1 and arr2

console.log(mergedArr); // [1, 2, 3, 4]

1. **Copying Objects**: The spread operator can also be used to copy properties from one object to another, creating a shallow copy.
   * **Example**:

const obj1 = { name: "Alice", age: 25 };

const obj2 = { ...obj1 }; // obj2 is a copy of obj1

obj2.age = 26;

console.log(obj1); // { name: "Alice", age: 25 }

console.log(obj2); // { name: "Alice", age: 26 }

1. **Merging Objects**: Similar to arrays, you can merge two or more objects into one. Properties from the objects on the right will overwrite those from the objects on the left.
   * **Example**:

const obj1 = { name: "Alice", age: 25 };

const obj2 = { age: 26, city: "New York" };

const mergedObj = { ...obj1, ...obj2 };

console.log(mergedObj); // { name: "Alice", age: 26, city: "New York" }

1. **Function Arguments**: The spread operator can be used to pass an array of values as individual arguments to a function.
   * **Example**:

function sum(a, b, c) {

return a + b + c;

}

const nums = [1, 2, 3];

console.log(sum(...nums)); // 6 (spreads the array into individual arguments)

**Important Notes:**

* The spread operator only performs a **shallow copy**, meaning it copies the top-level properties or elements. Nested objects or arrays inside an object will still reference the original.
* You can use the spread operator with **arrays**, **objects**, and **function arguments**, making it a versatile tool for dealing with iterable data.

**Example: Using the Spread Operator with Arrays and Objects:**

// Example 1: Using Spread in Arrays

const numbers = [1, 2, 3];

const moreNumbers = [4, 5, ...numbers]; // Spread the elements of 'numbers' into 'moreNumbers'

console.log(moreNumbers); // [4, 5, 1, 2, 3]

// Example 2: Using Spread in Objects

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

const updatedUser = { ...user, age: 31, city: "New York" }; // Modify and add new properties

console.log(updatedUser); // { name: "John", age: 31, city: "New York" }

**Conclusion:**

The spread operator is an extremely useful feature in JavaScript, enabling easy copying, merging, and passing of data structures like arrays and objects. It helps to write cleaner, more concise code, particularly when working with immutable patterns or combining multiple arrays/objects.

**Q5) What is a Prototype chain?**

**Answer:** In JavaScript, the **prototype chain** is a mechanism that allows an object to inherit properties and methods from another object. It is a fundamental concept in JavaScript’s inheritance model. When you try to access a property or method on an object, JavaScript first checks if that property exists on the object itself. If it doesn't, JavaScript looks for it in the object's prototype. This continues up the chain of prototypes until it either finds the property or reaches the end of the chain (i.e., *null*).

**How Does the Prototype Chain Work?**

Every JavaScript object has an internal property called *[[Prototype]]* (commonly accessed via *\_\_proto\_\_* or *Object.getPrototypeOf()).* The *[[Prototype]]* property points to another object, known as the object's prototype. This prototype can also have a *[[Prototype]]*, forming a chain that ultimately leads to *null*. The prototype chain is used to implement inheritance in JavaScript.

* **Example:**

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.greet = function() {

console.log(`Hello, my name is ${this.name}`);

};

const person1 = new Person('Alice', 30);

// Accessing the greet method from the prototype chain

person1.greet(); // "Hello, my name is Alice"

In this example, *person1* is an instance of *Person*, and it has access to the *greet()* method through the *Person.prototype* object. If the *greet()* method did not exist on *person1*, JavaScript would look for it in Person.prototype and find it there.

**Prototype Chain and Inheritance:**

JavaScript uses the prototype chain to implement inheritance. Objects can inherit properties and methods from other objects by setting the prototype of one object to another.

* **Example with Inheritance:**

function Animal(name) {

this.name = name;

}

Animal.prototype.speak = function() {

console.log(`${this.name} makes a noise.`);

};

function Dog(name) {

Animal.call(this, name); // Inherit properties from Animal

}

Dog.prototype = Object.create(Animal.prototype); // Set the prototype chain to inherit methods from Animal

Dog.prototype.constructor = Dog;

const dog = new Dog('Buddy');

dog.speak(); // "Buddy makes a noise."

In this example, *Dog* inherits the *speak()* method from *Animal* through the prototype chain.

**Understanding the Prototype Chain Lookup:**

1. When a property is accessed on an object, JavaScript checks if the property exists on the object itself.
2. If not found, JavaScript looks for the property in the object’s prototype.
3. This process continues up the prototype chain, checking each prototype, until it finds the property or reaches *null*.

**Prototype Chain and the *Object.prototype*:**

At the very end of the prototype chain is *Object.prototype*, the base prototype for all objects. If a property is not found on an object or any of its prototypes, JavaScript will look for it on *Object.prototype*. If the property is still not found, the lookup process ends at *null*.

* **Example:**

const obj = {};

console.log(obj.toString()); // Calls the toString method from Object.prototype

The *toString()* method is not defined on *obj*, but JavaScript finds it in *Object.prototype*.

**Prototype Chain Summary:**

* The **prototype chain** allows objects to inherit properties and methods from other objects.
* Every object in JavaScript has a *[[Prototype]]* property pointing to another object.
* The search for properties follows the prototype chain, starting from the object itself, up to *Object.prototype*.
* This mechanism is crucial for inheritance and method lookups in JavaScript.

**Q6) Difference between class and object.**

**Answer:** In JavaScript, **classes** and **objects** are two fundamental concepts used to create and organize data and functionality. However, they serve different purposes and have distinct characteristics.

**Class:**

A **class** is a blueprint or template for creating objects. It defines the properties and methods that the objects created from the class will have. A class itself does not hold any data but provides a structure and behavior for objects. In JavaScript, classes are syntactic sugar over the existing prototype-based inheritance.

* **Key Points:**
  + A class is a blueprint or a template.
  + It defines the properties and methods that the object will have.
  + Classes can have constructors, which are special methods used to initialize new objects.
  + You can define methods inside the class to perform actions.
* **Example of a Class:**

class Car {

// Constructor to initialize properties

constructor(brand, model) {

this.brand = brand;

this.model = model;

}

// Method to describe the car

describe() {

console.log(`This car is a ${this.brand} ${this.model}.`);

}

}

**Object:**

An **object** is an instance of a class. It is a real entity created based on the blueprint (class). When an object is created, it has its own copy of the properties and methods defined in the class. Objects hold data (in the form of properties) and behavior (in the form of methods).

* **Key Points:**
  + An object is an instance of a class.
  + It contains actual data and can invoke the methods defined in the class.
  + Objects are created using the *new* keyword followed by the class name.
  + Each object can have different values for its properties.
* **Example of an Object:**

const myCar = new Car('Toyota', 'Corolla'); // Creating an object from the Car class

myCar.describe(); // "This car is a Toyota Corolla."

**Summary of Differences:**

|  |  |  |
| --- | --- | --- |
| Aspect | Class | Object |
| Definition | A blueprint or template for creating objects. | An instance of a class with its own set of data. |
| Purpose | Defines properties and methods for objects. | Holds data and can invoke methods defined by the class. |
| Data | Does not hold data, it defines data structure. | Holds actual data in the form of properties. |
| Creation | Defined using the *class* keyword. | Created using the *new* keyword. |
| Example | *class Car { constructor(brand) { this.brand = brand; } }* | *const myCar = new Car('Toyota');* |

**In Conclusion:**

* **Class:** A class is a blueprint, a template, or a constructor for creating objects, and it defines the structure (properties) and behavior (methods) that objects of that class will have.
* **Object:** An object is an actual instance of a class that contains real data and can interact with its methods.

**Q7) Explain Call by value vs call by reference .**

**Answer:** In JavaScript, **call by value** and **call by reference** refer to how arguments are passed to functions. The primary difference lies in whether the function operates on the original variable or a copy of it. Let's explore both in detail.

**Call by Value:**

In **call by value**, a copy of the actual value is passed to the function. This means that changes made to the parameter inside the function do not affect the original variable. Call by value is used when passing primitive data types (like numbers, strings, booleans, null, and undefined).

* **Key Points:**
  + The function operates on a copy of the variable's value.
  + Changes inside the function do not affect the original variable.
  + This applies to **primitive types** in JavaScript.
* **Example of Call by Value:**

function modifyNumber(num) {

num = 10; // This change does not affect the original value

}

let x = 5;

modifyNumber(x);

console.log(x); // Output: 5 (original value remains unchanged)

In the above example, *num* inside the function is a copy of *x*. When *num* is changed, it does not affect *x*.

**Call by Reference:**

In **call by reference**, instead of passing a copy of the value, the reference (or memory address) of the variable is passed to the function. This means that changes made to the parameter inside the function will directly affect the original variable. Call by reference applies to **non-primitive types**, like objects and arrays.

* **Key Points:**
  + The function operates on the reference to the actual value (memory address).
  + Changes inside the function will modify the original variable.
  + This applies to **objects** and **arrays** in JavaScript.
* **Example of Call by Reference:**

function modifyArray(arr) {

arr.push(4); // This will modify the original array

}

let numbers = [1, 2, 3];

modifyArray(numbers);

console.log(numbers); // Output: [1, 2, 3, 4] (original array is modified)

In this example, the array *numbers* is passed by reference to the *modifyArray* function. As a result, the function modifies the original array.

**Summary of Differences:**

|  |  |  |
| --- | --- | --- |
| Aspect | Call by Value | Call by Reference |
| Data Type | Applies to **primitive types** (number, string, boolean, etc.). | Applies to **objects** and **arrays**. |
| Behavior | A copy of the value is passed to the function. | A reference (memory address) to the value is passed. |
| Effect on Original | Changes inside the function do not affect the original value. | Changes inside the function affect the original object or array. |
| Example | Numbers, strings, booleans, etc. | Arrays, objects, and functions. |

**In Conclusion:**

* **Call by Value**: When you pass a primitive value to a function, the function gets a copy of that value. Any changes made inside the function do not affect the original variable.
* **Call by Reference**: When you pass an object or array to a function, the function gets a reference to the original data. Any changes made inside the function will affect the original data.

**Q8) What is a Higher-order Function?**

**Answer**: In JavaScript, a **Higher-order Function** is a function that either **takes another function as an argument, returns a function**, or **both**. Simply put, higher-order functions work with other functions, treating them as first-class citizens. This is possible because in JavaScript, functions are objects and can be passed around just like any other value. Higher-order functions are extremely useful for writing cleaner, more modular, and reusable code. Common examples of higher-order functions are *map()*, *filter()*, and *forEach()*, where each of them takes a callback function to process array elements.

Here’s a simple example:

// Higher-order function that takes another function as an argument

function greetUser(greetingFunction) {

greetingFunction();

}

// A simple function to pass

function sayHello() {

console.log("Hello!");

}

// Passing sayHello into greetUser

greetUser(sayHello);

In this example, *greetUser* is a higher-order function because it accepts another function (*sayHello*) as its argument. Higher-order functions are key to functional programming in JavaScript and are widely used in modern web development.

4o

**Q9) What is the API? Difference between Get vs post?**

**Answer**: An **API (Application Programming Interface)** is a set of rules and protocols that allows different software applications to communicate with each other. It acts like a messenger that takes a request, processes it, and returns the appropriate response. In web development, APIs are often used to allow the frontend (client-side) to interact with the backend (server-side) and databases. APIs simplify complex programming tasks and enable developers to use predefined functions without having to build everything from scratch. Common examples include fetching user data, submitting forms, or posting content through RESTful APIs.

For example, when you use a weather app, it fetches real-time weather data from a remote server using an API. Similarly, when a frontend React application interacts with a backend Node.js server, it does so through API requests, often using HTTP methods like GET, POST, PUT, and DELETE.

APIs can be **RESTful** (working with URLs and HTTP) or **GraphQL-based** (fetching only required data). They can be **public** (e.g., OpenWeather API) or **private** (used within a company’s internal systems). In web development, APIs play a crucial role in enabling **frontend-backend communication, third-party integrations, and microservices architecture**. 🚀

**Difference Between GET and POST:**

**GET** and **POST** are two of the most commonly used HTTP methods when communicating with APIs:

* **GET** is used to **request data** from a server. It is **read-only**, meaning it does not change or modify data on the server. Data sent through GET is appended to the URL in the form of query parameters (visible in the URL).  
  Example:  
  GET *https://example.com/api/users?id=123*
* **POST** is used to **send data** to the server to **create** or **modify** a resource. It sends data inside the body of the HTTP request, which is **not visible** in the URL. POST is typically used for actions like submitting a form, uploading a file, or creating a new database entry.  
  Example:

POST https://example.com/api/users

Body:

{

"name": "John Doe",

"email": "john@example.com"

}

In short:

* **GET** = retrieve data
* **POST** = send data to be processed

Both are important depending on what action you want the application to perform!

**Q10) Explain Difference between local storage and Session storage.**

**Answer**: **Local Storage** and **Session Storage** are both part of the Web Storage API that allows you to store data in the browser, but they differ mainly in how long the data persists and how they are scoped.

* **Local Storage** stores data with **no expiration time**. This means that even if the user closes the browser and comes back later, the data will still be available until it is explicitly deleted by the user or through code. Local storage is useful for saving things like user preferences, login sessions, or cart items that you want to persist across multiple visits.
* **Session Storage**, on the other hand, is **temporary** and lasts only for the **duration of the page session**. Once the user closes the tab or browser, the session storage is cleared automatically. It is useful for storing data that you only need while the user is actively visiting a page, such as form inputs or temporary navigation states.

**Quick comparison table:**

|  |  |  |
| --- | --- | --- |
| Feature | Local Storage | Session Storage |
| Lifespan | Until manually cleared | Until tab/browser closes |
| Storage limit | About 5MB | About 5MB |
| Accessible across tabs | Yes | No |

**Code Example:**

// Local Storage

localStorage.setItem("username", "Shahbaz");

console.log(localStorage.getItem("username")); // Output: Shahbaz

// Session Storage

sessionStorage.setItem("sessionName", "ShahbazSession");

console.log(sessionStorage.getItem("sessionName")); // Output: ShahbazSession

In short, use **Local Storage** when you want to store something more permanent, and **Session Storage** when you want it only for that browser session!

**Q11) What are cookies? And why will you use it?**

**Answer**: **Cookies** are small pieces of data stored directly on the user's browser by a website. They are mainly used to remember information about the user between different pages or visits. Each time the user makes a request to the server (like loading a new page), cookies are automatically sent along with the request, allowing the server to identify the user and maintain information like login status, preferences, or shopping cart items.

You would use cookies for several reasons:

* To **maintain user sessions**, such as keeping a user logged in across multiple pages.
* To **personalize user experience**, like remembering language settings or theme choices.
* For **tracking and analytics**, helping website owners understand user behavior.
* For **storing small amounts of data** that need to persist even after the user leaves or reloads the page.

**Example of Setting and Getting Cookies in JavaScript:**

// Set a cookie

document.cookie = "username=Shahbaz; expires=Fri, 1 May 2025 12:00:00 UTC; path=/";

// Get cookies

console.log(document.cookie);

// Output might be: "username=Shahbaz"

In short, **cookies are essential** when you need to **store small but important information** and **maintain continuity** between user interactions on a website!

**Q12) What is object-oriented programming?**

**Answer**: **Object-Oriented Programming (OOP)** is a programming paradigm that organizes software design around **objects**, rather than functions and logic. Objects are instances of **classes**, which are blueprints defining the properties (also called attributes) and behaviors (also called methods) of the object. The main idea behind OOP is to bundle related data and functionality together, making the code more modular, reusable, and easier to maintain.

OOP is based on four key principles:

* **Encapsulation**: Hiding the internal state and requiring all interaction to be performed through an object's methods.
* **Inheritance**: Allowing one class to inherit properties and methods from another class.
* **Polymorphism**: Allowing different classes to be treated as instances of the same parent class, mainly through method overriding.
* **Abstraction**: Hiding complex realities while exposing only the necessary parts.

**Example of Object-Oriented Programming in JavaScript:**

// Defining a class

class Car {

constructor(brand, model) {

this.brand = brand;

this.model = model;

}

displayInfo() {

console.log(`This car is a ${this.brand} ${this.model}.`);

}

}

// Creating an object

const myCar = new Car('Toyota', 'Corolla');

// Using a method

myCar.displayInfo();

// Output: This car is a Toyota Corolla.

In simple words, **Object-Oriented Programming** helps us **think in terms of real-world objects** (like cars, users, products) and **model them efficiently** in code! 🚀

**Q13) When will you return something from a function?**

**Answer**: You will return something from a function when you want the function to **produce a result** that can be **used elsewhere** in your program. Returning a value allows you to take the output from a function and either store it in a variable, pass it into another function, or display it. Without a return statement, a function will complete its task but won't give anything back to the part of the code that called it. Typically, you return something when the function performs a calculation, processes some input, or determines a result that needs to be reused later.

**Example:**

function addNumbers(a, b) {

return a + b; // returning the sum

}

const result = addNumbers(5, 3);

console.log(result); // Output: 8

In the example above, the *addNumbers* function returns the sum of two numbers. The returned value is stored in the *result* variable and then printed.  
👉 Without *return*, the function would perform the addition but not give us any value to work with outside the function.

**Q14) Explain Difference between Array vs LinkedList.**

**Answer**: An **Array** and a **LinkedList** are both used to store collections of elements, but they are structured and managed differently. An **Array** stores elements in **contiguous memory locations** and allows for **fast access** to any element using its index (O(1) time complexity). However, inserting or deleting elements in the middle of an array can be expensive, because it requires shifting other elements. On the other hand, a **LinkedList** is made up of **nodes**, where each node contains the data and a reference (or pointer) to the next node. This structure allows **easy insertion and deletion** of elements (especially in the middle or beginning) without shifting other elements, but accessing a specific element requires **traversing** the list from the beginning, resulting in **O(n) time complexity**. In short, arrays are better for **fast access** and linked lists are better for **fast insertions and deletions**.

**Small Code Example:**

**Array Example:**

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

console.log(arr[2]); // Output: 3 (Direct access by index)

**LinkedList Example (Basic Structure in JS):**

class Node {

constructor(data) {

this.data = data;

this.next = null;

}

}

let first = new Node(1);

let second = new Node(2);

first.next = second; // Linking nodes

console.log(first.next.data); // Output: 2 (Access through next pointer)

Here’s a **small and clean table** :

|  |  |  |
| --- | --- | --- |
| Feature | Array | LinkedList |
| Memory Allocation | Contiguous memory blocks | Non-contiguous memory (nodes linked together) |
| Access Time | Fast (O(1) by index) | Slow (O(n), needs traversal) |
| Insertion/Deletion | Slow (needs shifting elements) | Fast (just change pointers) |
| Memory Efficiency | Fixed size (static arrays) or resizing needed | Flexible size (dynamically grows) |
| Best Use Case | When frequent access by index is needed | When frequent insertions/deletions are needed |

**Q15) How will you debug a JavaScript application?**

**Answer**: To debug a JavaScript application effectively, I would follow these steps:

1. **Use Browser Developer Tools**: Every modern browser (Chrome, Firefox, Edge) provides built-in developer tools. I would open the developer tools by pressing *F12* or *Ctrl+Shift+I* and navigate to the **Console** and **Sources** tabs. In the **Console**, I can view any errors or logs that occur in the application, and in **Sources**, I can set breakpoints to pause execution and inspect variables or step through the code line by line.
2. **Console Logging**: Adding *console.log()* statements at various points in the code can help track the flow of the application and the values of variables. It's useful for inspecting the state of the application and ensuring that it behaves as expected.
3. **Breakpoints and Debugging**: I can set breakpoints in the code (especially inside the **Sources** tab) to pause execution at certain lines, allowing me to inspect variables, call stacks, and step through the code line by line to understand what's happening.
4. **Use Debugger Keyword**: I can insert the *debugger;* statement directly in the code. This will automatically pause the execution at that point when developer tools are open, allowing me to inspect the program’s state.
5. **Network Tab for API Calls**: If my JavaScript application involves fetching data from APIs, I can use the **Network** tab in the developer tools to inspect network requests and responses. This helps to check if the API calls are successful, if the server responds with the correct data, or if there are any issues with the request (status code, headers, etc.).
6. **Error Handling**: I would also add proper error handling (try-catch blocks) to handle any unexpected issues gracefully. This provides more descriptive error messages, which are helpful during debugging.
7. **Linting and Static Analysis**: Tools like ESLint can help catch potential issues in the code early by analyzing it for syntax or logical errors, making it easier to identify mistakes before they become bugs.
8. **Unit Testing**: Writing unit tests with frameworks like Jest or Mocha can help catch errors early by verifying that each function or component behaves as expected. This is especially useful for debugging after making changes or adding new features.

By following these methods, I can systematically track down and fix issues in a JavaScript application.

**Q16) What is template literal in ES6?**

**Answer**: In ES6, a **template literal** (also known as a **template string**) is a way to work with strings in JavaScript that provides a more readable and flexible syntax for string interpolation and multi-line strings. Template literals are enclosed by backticks (*`*) rather than single or double quotes.

**Key features of template literals:**

1. **String Interpolation**: You can embed expressions inside a string, which will be evaluated and interpolated. Expressions are enclosed in *${}*.

**Example**:

const name = "Shahbaz";

const age = 25;

const greeting = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(greeting);

// Output: Hello, my name is Shahbaz and I am 25 years old.

1. **Multi-line Strings**: Template literals allow you to create multi-line strings easily without needing escape characters like *\n*.

**Example**:

const message = `This is a

multi-line

string.`;

console.log(message);

// Output:

// This is a

// multi-line

// string.

1. **Expression Evaluation**: Any valid JavaScript expression can be embedded inside *${}*. This could be variables, mathematical operations, function calls, or even object properties.

**Example**:

const a = 10;

const b = 20;

const sum = `The sum of a and b is ${a + b}.`;

console.log(sum);

// Output: The sum of a and b is 30.

1. **Tagged Templates**: Template literals can also be used with a function (tag) that processes the template literal before it's returned. This is called **tagged template literals**.

**Example**:

function tagExample(strings, ...expressions) {

console.log(strings); // Array of string literals

console.log(expressions); // Array of expressions

}

const name = "Shahbaz";

tagExample`Hello, ${name}! How are you?`;

// Output:

// [ 'Hello, ', '!', ' How are you?' ]

// [ 'Shahbaz' ]

**Benefits:**

* **Improved readability** when working with complex strings.
* **Easy string interpolation** and dynamic content insertion.
* **Cleaner multi-line strings** without needing special syntax for newlines.

Template literals make string handling much more powerful and flexible in JavaScript.

**Q17) Difference between Spread Operator and rest operator.**

**Answer**: The **spread operator** (*...*) and **rest operator** (*...*) in JavaScript have the same syntax, but they are used in different contexts and for different purposes. Below is a detailed explanation of their differences:

**1. Spread Operator (***...***):**

The spread operator is used to **spread** or **expand** the elements of an array or object into individual elements. It is used when you want to **unpack** values from an array or object into another array or object.

* **Use case for Arrays**: It can be used to spread elements of an array into a new array.

**Example**:

const arr1 = [1, 2, 3];

const arr2 = [...arr1, 4, 5]; // Spread elements of arr1 into arr2

console.log(arr2);

// Output: [1, 2, 3, 4, 5]

* **Use case for Objects**: It can also be used to clone an object or merge objects.

**Example**:

const obj1 = { name: "John", age: 30 };

const obj2 = { ...obj1, city: "New York" }; // Spread obj1 properties into obj2

console.log(obj2);

// Output: { name: "John", age: 30, city: "New York" }

**2. Rest Operator (***...***):**

The rest operator is used to **collect** multiple elements into a single array or object. It is commonly used in function parameters to collect remaining arguments into a single array or to gather remaining properties when destructuring objects.

* **Use case in Function Parameters**: It allows a function to accept an arbitrary number of arguments and collect them into an array.

**Example**:

function sum(...numbers) {

return numbers.reduce((acc, curr) => acc + curr, 0);

}

console.log(sum(1, 2, 3, 4)); // Output: 10

* **Use case in Destructuring Objects/Arrays**: It can collect the remaining properties or elements that aren't specifically picked up during destructuring.

**Example (Array Destructuring)**:

const arr = [1, 2, 3, 4, 5];

const [first, second, ...rest] = arr;

console.log(rest);

// Output: [3, 4, 5]

**Example (Object Destructuring)**:

const obj = { name: "Alice", age: 25, country: "USA" };

const { name, ...rest } = obj;

console.log(rest);

// Output: { age: 25, country: "USA" }

**Key Differences:**

1. **Purpose**:
   * The **spread operator** is used to **expand** or **spread** elements from an array or object.
   * The **rest operator** is used to **collect** remaining elements into a single array or object.
2. **Context**:
   * **Spread** is used for array or object manipulation (copying, merging).
   * **Rest** is used in function arguments or destructuring assignments to collect remaining values.
3. **Example Use**:
   * **Spread**: *const newArr = [...oldArr, 4, 5];*
   * **Rest**: *function sum(...args) { return args.reduce((a, b) => a + b); }*

**Summary:**

* **Spread operator (***...***)**: Expands or spreads elements (array or object).
* **Rest operator (***...***)**: Collects elements into an array or object (used in function arguments or

**Q18) What is destructuring?**

**Answer**: **Destructuring** is a feature in JavaScript that allows you to unpack values from arrays or properties from objects into distinct variables. It provides a concise and readable way to extract data from arrays and objects, making your code more efficient and less verbose.

**Array Destructuring**

Array destructuring allows you to assign values from an array to variables in a specific order. The variables are assigned values based on the position of the elements in the array.

**Example:**

const arr = [1, 2, 3];

const [a, b, c] = arr;

console.log(a); // Output: 1

console.log(b); // Output: 2

console.log(c); // Output: 3

In this example, the values of *arr* are destructured and assigned to the variables *a*, *b*, and *c* in the order of their positions in the array.

**Skipping values in an array:** You can also skip values in the array by leaving a blank space.

**Example:**

const arr = [1, 2, 3];

const [a, , c] = arr;

console.log(a); // Output: 1

console.log(c); // Output: 3

**Object Destructuring**

Object destructuring allows you to extract values from objects and assign them to variables based on the property names of the object.

**Example:**

const person = { name: "John", age: 30, city: "New York" };

const { name, age, city } = person;

console.log(name); // Output: John

console.log(age); // Output: 30

console.log(city); // Output: New York

In this example, the properties *name*, *age*, and *city* from the *person* object are destructured and assigned to variables with the same names.

**Renaming variables while destructuring:** You can also rename the variables while destructuring by using a colon.

**Example:**

const person = { name: "John", age: 30, city: "New York" };

const { name: fullName, age: yearsOld } = person;

console.log(fullName); // Output: John

console.log(yearsOld); // Output: 30

**Default Values in Destructuring**

You can provide default values for variables in case the value from the array or object is *undefined*.

**Example (Array with Default Value):**

const arr = [1];

const [a, b = 2] = arr;

console.log(a); // Output: 1

console.log(b); // Output: 2 (default value)

**Example (Object with Default Value):**

const person = { name: "John" };

const { name, age = 25 } = person;

console.log(name); // Output: John

console.log(age); // Output: 25 (default value)

**Nested Destructuring**

You can also destructure objects and arrays that are nested inside each other.

**Example (Nested Array Destructuring):**

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

const [a, [b, c], d] = arr;

console.log(a); // Output: 1

console.log(b); // Output: 2

console.log(c); // Output: 3

console.log(d); // Output: 4

**Example (Nested Object Destructuring):**

const person = {

name: "John",

address: { city: "New York", country: "USA" }

};

const { name, address: { city, country } } = person;

console.log(name); // Output: John

console.log(city); // Output: New York

console.log(country); // Output: USA

**Why Use Destructuring?**

* **Cleaner code:** Destructuring allows you to extract values without needing to write repetitive code.
* **Default values:** You can easily provide default values for variables.
* **Nested objects/arrays:** It simplifies the process of working with nested data structures.
* **Readability:** It makes your code more readable and expressive.

**In Summary:**

Destructuring is a feature that helps you unpack values from arrays or properties from objects into variables in a more concise and readable way. It reduces the need for repetitive code and makes working with complex data structures easier.

**Q19) Tell me about Ternary Operators.**

**Answer**: **Ternary Operator** is a shorthand way of writing an *if-else* statement in JavaScript. It is a conditional operator that evaluates an expression and returns one of two values based on whether the condition is true or false. It is called the "ternary operator" because it involves three operands: the condition, the value if true, and the value if false.

**Syntax:**

condition ? value\_if\_true : value\_if\_false;

* **condition**: The expression that is evaluated to either *true* or *false*.
* **value\_if\_true**: The value returned if the condition is *true*.
* **value\_if\_false**: The value returned if the condition is *false*.

**Example 1: Basic Ternary Operator**

const age = 18;

const canVote = age >= 18 ? "Yes, you can vote" : "No, you can't vote";

console.log(canVote); // Output: Yes, you can vote

In this example, the condition checks if *age* is greater than or equal to 18. If *true*, it returns *"Yes, you can vote"*, otherwise it returns *"No, you can't vote"*.

**Example 2: Using Ternary Operator for Assignments**

const temperature = 30;

const weather = temperature > 25 ? "It's hot" : "It's cool";

console.log(weather); // Output: It's hot

Here, if the temperature is greater than 25, the string *"It's hot"* is assigned to the variable *weather*, otherwise *"It's cool"* is assigned.

**Example 3: Nested Ternary Operator**

You can also nest ternary operators, but it's important to avoid making the code too complex or difficult to read.

const age = 15;

const canDrive = age >= 18 ? "Yes, you can drive" : age >= 16 ? "You can drive with a permit" : "No, you can't drive";

console.log(canDrive); // Output: You can drive with a permit

In this example, the ternary operator checks multiple conditions:

* If *age* is 18 or greater, it returns *"Yes, you can drive"*.
* If not, it checks if the *age* is 16 or greater and returns *"You can drive with a permit"*.
* If neither condition is *true*, it returns *"No, you can't drive"*.

**Example 4: Ternary Operator with Functions**

function getStatus(age) {

return age >= 18 ? "Adult" : "Minor";

}

console.log(getStatus(20)); // Output: Adult

console.log(getStatus(15)); // Output: Minor

In this function, the ternary operator is used to check if the person is an adult or a minor based on their age.

**Advantages of Ternary Operator:**

* **Concise**: It reduces the amount of code, making it shorter and easier to read in simple cases.
* **Inline Use**: It is useful when you want to return values or make assignments directly in expressions.

**Disadvantages of Ternary Operator:**

* **Readability**: For more complex conditions or nested ternaries, it can become harder to read and understand.
* **Misuse**: It can be overused in situations where an *if-else* statement would be clearer.

**Summary:**

The ternary operator is a concise way of performing conditional checks in JavaScript. It’s often used for simple conditions, assignments, and quick decisions, but should be used carefully when conditions become complex, as readability can suffer.

**Q20) Difference between regular function and array function.**

**Answer**: The difference between a **regular function** and an **arrow function** in JavaScript primarily lies in their syntax, behavior of the *this* keyword, and how they handle arguments.

**1. Syntax:**

* **Regular Function**: A regular function uses the *function* keyword to define a function.

function add(a, b) {

return a + b;

}

* **Arrow Function**: Arrow functions have a shorter syntax, using *=>* and don't require the *function* keyword.

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

**2. Handling of** *this***:**

* **Regular Function**: In a regular function, the value of *this* is dynamically bound to the context in which the function is called. For example, inside an object method, *this* refers to the object.

const person = {

name: 'John',

greet: function() {

console.log(this.name);

}

};

person.greet(); // Output: John

* **Arrow Function**: Arrow functions **do not have their own** *this* **context**. Instead, they inherit *this* from the surrounding lexical context (where the function was created). This means that *this* in an arrow function refers to the *this* value of the enclosing function or scope.

const person = {

name: 'John',

greet: () => {

console.log(this.name); // Here `this` refers to the surrounding scope, not the object

}

};

person.greet(); // Output: undefined (because `this` doesn't point to the object)

**In short**: Arrow functions are **lexically bound** to *this*, whereas regular functions are **dynamically bound**.

**3. *arguments* Object:**

* **Regular Function**: Regular functions have access to the *arguments* object, which is an array-like object containing all the arguments passed to the function.

function sum() {

let total = 0;

for (let i = 0; i < arguments.length; i++) {

total += arguments[i];

}

return total;

}

console.log(sum(1, 2, 3)); // Output: 6

* **Arrow Function**: Arrow functions do not have their own *arguments* object. If you need to access arguments in an arrow function, you would need to use rest parameters (*...args*) or refer to the *arguments* of an enclosing function.

const sum = (...args) => {

return args.reduce((total, num) => total + num, 0);

};

console.log(sum(1, 2, 3)); // Output: 6

**4. Use in Constructors:**

* **Regular Function**: Regular functions can be used as constructors to create instances of objects. You can use the *new* keyword to create a new instance.

function Person(name, age) {

this.name = name;

this.age = age;

}

const person1 = new Person('Alice', 25);

console.log(person1.name); // Output: Alice

* **Arrow Function**: Arrow functions cannot be used as constructors and will throw an error if used with *new*.

const Person = (name, age) => {

this.name = name;

this.age = age;

};

const person1 = new Person('Alice', 25); // Error: Person is not a constructor

**5. Implicit Return:**

* **Regular Function**: Regular functions require a *return* statement to return a value. Without it, the function returns *undefined*.

function greet() {

return 'Hello, World!';

}

console.log(greet()); // Output: Hello, World!

* **Arrow Function**: Arrow functions with a single expression can implicitly return the value, without using the *return* keyword. However, if you use curly braces *{}* for the function body, you will need to explicitly use *return*.

const greet = () => 'Hello, World!';

console.log(greet()); // Output: Hello, World!

For a multi-line expression, you must use return:

javascript

CopyEdit

const add = (a, b) => {

return a + b;

};

**6. *this* in Event Handlers:**

* **Regular Function**: In event handlers, regular functions will use the context in which they are called (the element that triggered the event).

const button = document.querySelector('button');

button.addEventListener('click', function() {

console.log(this); // `this` refers to the button element

});

* **Arrow Function**: In event handlers, arrow functions will inherit *this* from the surrounding context, which may not always be the desired behavior.

const button = document.querySelector('button');

button.addEventListener('click', () => {

console.log(this); // `this` refers to the surrounding lexical scope, not the button

});

**Summary:**

* **Syntax**: Arrow functions are more concise than regular functions.
* ***this* behavior**: Regular functions have their own *this*, while arrow functions lexically inherit *this*.
* ***arguments* object**: Regular functions have access to *arguments*; arrow functions do not.
* **Constructors**: Regular functions can be used as constructors; arrow functions cannot.
* **Implicit return**: Arrow functions can return a value implicitly, but regular functions require an explicit *return*.

Both types of functions are useful in different contexts. Arrow functions are great for shorter, cleaner code, especially for callback functions and methods where you want to preserve the context of this. Regular functions, on the other hand, are more flexible and are suitable for use cases like constructors or methods where the this context might change based on how the function is called.

**Q21) What do you mean by IIFE?**

**Answer**: **IIFE (Immediately Invoked Function Expression)**

An **IIFE** is a **function that runs immediately after it is defined**.

In JavaScript, **IIFE** stands for **Immediately Invoked Function Expression**.  
It is a design pattern where you **define a function and immediately call it**, without explicitly invoking it later.

**Syntax:**

(function() {

// code inside the function

console.log('This runs immediately!');

})();

Or with an arrow function:

(() => {

console.log('Arrow function IIFE!');

})();

* The function is wrapped inside parentheses *()* to **turn it into an expression**.
* Then another () is added immediately to **execute** it.

**Why use IIFE?**

* **Avoid polluting the global scope**: Variables inside an IIFE are not accessible outside the function, keeping them private.
* **Encapsulation**: It creates a private scope for variables and functions.
* **Execute code immediately**: Useful when you want some setup code to run once.

**Example:**

(function() {

let message = 'Hello from IIFE';

console.log(message);

})();

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

* *message* is private inside the IIFE and cannot be accessed outside.

**Simple way to remember:**

**IIFE = Function + Immediate Execution**

**Ignore parts from here**

**Q1) What is Interpreted & compiled programming languages?**

Answer:

**Interpreted Programming Languages:** Interpreted languages are executed line by line by an interpreter at runtime, rather than being compiled into machine code beforehand. This allows for greater flexibility and ease of debugging, as developers can run and test code without a separate compilation step. However, interpreted languages can be slower in execution compared to compiled languages since each line must be parsed and executed every time the program runs. Examples of interpreted languages include **JavaScript**, **Python**, and **Ruby**.

**Compiled Programming Languages:** Compiled languages require a compiler to translate the entire source code into machine code (binary) before execution. This process optimizes the code for performance, resulting in faster execution times. Once compiled, the program can run independently of the original source code. However, this can make debugging more challenging, as errors must be fixed and the code recompiled before testing again. Examples of compiled languages include C, C++, and Go.

In summary, interpreted languages prioritize ease of use and flexibility, while compiled languages focus on performance and efficiency.

**Q37) What is single-threaded & multi-threaded programming languages?**

Answer:

**Single-Threaded Programming:** Single-threaded programming refers to the execution model in which a program runs on a single thread of execution. In this model, only one operation can be processed at a time, meaning that tasks are executed sequentially. While this simplifies programming and avoids issues related to concurrent access to shared resources (like race conditions), it can lead to inefficiencies in resource utilization, especially in I/O-bound tasks. JavaScript is a common example of a single-threaded language, using an event loop to manage asynchronous operations without blocking the main execution thread.

**Multi-Threaded Programming:** Multi-threaded programming allows a program to create multiple threads of execution that can run concurrently. Each thread can perform different tasks simultaneously, which can significantly improve performance and responsiveness, especially in CPU-bound tasks or when dealing with multiple I/O operations. This model can take advantage of multi-core processors, where threads can run on separate cores, making better use of system resources. Languages like C, C++, and Java support multi-threading through libraries or built-in features, allowing developers to manage threads, synchronization, and concurrency.

**Q37) What is a strongly typed and weakly typed programming language?**

**Answer:**

A **strongly typed programming language** enforces strict rules about how variables and data types interact. It does not allow implicit type conversions that might lead to unexpected behavior. For example, in Python, attempting to add a string and a number ("10" + 5) results in an error. Examples of strongly typed languages include Python, Java, and C#.

A **weakly typed programming language** is more flexible in handling data types and often allows implicit type conversions (also known as type coercion). This can lead to unexpected results but also allows for more flexibility in coding. For example, in JavaScript, "10" + 5 results in "105" due to automatic type conversion. Examples of weakly typed languages include JavaScript, PHP, and Perl.

**Q38) Why is JavaScript considered a weakly typed language from an architectural point of view?**

**Answer**: JavaScript is considered a weakly typed language from an architectural point of view because it allows implicit type conversion (type coercion), meaning values of different types can interact without explicit conversion by the programmer. This behavior is built into the JavaScript engine to enhance flexibility and ease of use but can sometimes lead to unexpected results.

✔ Dynamic Typing: JavaScript variables do not have a fixed type. A variable can hold different types of values at different times. For example, let x = 10; x = "hello"; is valid.

✔ Type Coercion: JavaScript automatically converts types when performing operations. For instance, "10" + 5 results in "105" instead of throwing an error.

✔ Loosely Enforced Type Rules: Unlike strongly typed languages, JavaScript does not strictly enforce type constraints during operations. This can sometimes cause unexpected behavior due to implicit conversions.

✔ Interpreted Nature: JavaScript is designed to run in browsers and execute dynamically, favoring flexibility and quick execution over strict type safety.

**Q39) What is a closure in JavaScript?**

Answer:

A closure in JavaScript is a function that retains access to its outer function’s variables even after the outer function has finished executing. This happens because JavaScript functions remember the scope in which they were created.

✔ Example:

function outerFunction() {

let count = 0; // Variable in outer scope

return function innerFunction() {

count++;

console.log(count); // Inner function has access to count

};

}

const counter = outerFunction();

counter(); // Output: 1

counter(); // Output: 2

Here, innerFunction still has access to count, even though outerFunction has already executed.

✔ **Why are closures useful?**

* They help in **data encapsulation** by keeping variables private.
* They enable functions like **memoization, event handling, and maintaining state** in JavaScript applications.

**Q40) What is a Promise and async/await in JavaScript? What is the difference between them?**

**Answer:**

**Promise:**

A **Promise** is a JavaScript object that represents the eventual completion or failure of an **asynchronous operation**. It acts as a placeholder for a value that will be available in the future.

When a Promise is created, it is initially in the **pending** state. It can either:

1. **Resolve (fulfilled)** – The operation was successful, and the result is returned.
2. **Reject (failed)** – The operation failed, and an error is returned.

A Promise provides two main methods to handle the result:  
✔ .then() – Executes when the Promise is resolved (successful).  
✔ .catch() – Executes when the Promise is rejected (error occurred).

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

let success = true;

if (success) {

resolve("✅ Data fetched successfully!");

} else {

reject("❌ Error fetching data!");

}

}, 2000); // Simulating a delay

});

};

fetchData()

.then(data => console.log(data)) // Runs if resolved

.catch(error => console.log(error)); // Runs if rejected

👉 Here, fetchData returns a Promise that resolves after 2 seconds.  
👉 If success is true, the Promise resolves, and .then() runs.  
👉 If success is false, the Promise rejects, and .catch() runs.

**async/await:**

The async/await syntax is a modern and cleaner way to handle Promises. It allows us to write asynchronous code in a synchronous style, making it more readable and easier to debug.

✔ **async** – Declares a function as asynchronous, meaning it always returns a Promise.  
✔ **await** – Pauses the function execution until the Promise is resolved (or rejected). It can only be used inside an async function.

const fetchData = async () => {

try {

let data = await new Promise(resolve => setTimeout(() => resolve("✅ Data fetched successfully!"), 2000));

console.log(data);

} catch (error) {

console.log("❌ Error fetching data!", error);

}

};

fetchData();

👉 Here, the function fetchData() is marked async, so it can use await to pause execution until the Promise resolves.  
👉 If an error occurs, try...catch handles it.

**Difference Between Promise and async/await:**

✔ **Syntax Style:**

* **Promises** use .then() and .catch(), which can lead to nested code.
* **async/await** makes asynchronous code look synchronous, improving readability.

✔ **Error Handling:**

* **Promises** use .catch() to handle errors.
* **async/await** uses try...catch, making error handling more structured.

✔ **Code Readability:**

* **Promises** can become hard to read if multiple .then() are chained (callback hell).
* **async/await** makes code look cleaner and sequential, reducing complexity.

✔ **Debugging:**

* **Promises** require debugging through .then() chains.
* **async/await** allows debugging with standard try/catch blocks, making it easier to trace errors.

**Q41) What is a synchronous and asynchronous operation/function in JavaScript? Answer:**

✔ **Synchronous Operation**: A **synchronous function** executes **line by line**, meaning each operation **must complete before the next one starts**. The JavaScript engine **waits** for a task to finish before moving to the next one.

console.log("Start");

console.log("Processing...");

console.log("End");

//output

Start

Processing...

End

✔ **Asynchronous Operation:** An asynchronous function allows JavaScript to execute other code while waiting for a task to complete. JavaScript doesn't block execution; instead, it continues running and processes the async operation once it's done (using callbacks, Promises, or async/await).

console.log("Start");

setTimeout(() => {

console.log("Delayed Execution");

}, 2000);

console.log("End");

//output

Start

End

Delayed Execution

**Q42) What is a prototype in JavaScript?**

**Answer:** In JavaScript, a prototype is an object that is associated with every function and object by default. It is a fundamental concept of JavaScript's inheritance model and plays a crucial role in the language's object-oriented capabilities.

**🔹 Key Concepts of Prototypes:**

1. **Prototype Object:**
   * Every JavaScript function has a property called prototype. When you create an object using a constructor function, that object inherits properties and methods from the prototype object.
2. **Inheritance:**
   * JavaScript uses prototype-based inheritance, meaning that objects can inherit properties and methods from other objects. This allows for the creation of reusable and extensible code.
3. **Adding Properties and Methods:**
   * You can add properties and methods to an object's prototype, which then become available to all instances of that object.

**🔹 Example of Prototypes:**

// Constructor function

function Person(name, age) {

this.name = name;

this.age = age;

}

// Adding a method to the Person prototype

Person.prototype.introduce = function() {

console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);

};

// Creating instances

const person1 = new Person("Alice", 30);

const person2 = new Person("Bob", 25);

// Using the prototype method

person1.introduce(); // Output: Hello, my name is Alice and I'm 30 years old.

person2.introduce(); // Output: Hello, my name is Bob and I'm 25 years old.

**🔹 How Prototypes Work:**

* When you call a method on an object, JavaScript first checks if the method exists on the object itself. If it doesn’t find it, it looks up the prototype chain until it finds the method or reaches the end of the chain (null).

**🔹 Prototype Chain:**

* The prototype chain is a series of linked objects where each object has a reference to its prototype. This chain allows for property and method lookups, enabling inheritance.

**🔹 Conclusion:**

Prototypes are a powerful feature in JavaScript that facilitate inheritance and code reuse. By adding properties and methods to an object's prototype, developers can create shared behaviors across multiple instances, promoting efficient coding practices and enhancing the language's object-oriented capabilities.

**Q43) What is event Delegation in js?**

**Answer:** Event delegation is a powerfull technique in javascript for handling events efficiently, ecpecially when dealing with large number of child elements or dynamically added elements. Instead of addinng individual event listener to each child element one can add a single event listener to the parent element. This parent elements then checks the events target (the element where the event originated) to determine if it should handle the event.

**🔹 Example Without Event Delegation (Inefficient)**

document.getElementById("item1").addEventListener("click", () => {

console.log("Item 1 clicked");

});

document.getElementById("item2").addEventListener("click", () => {

console.log("Item 2 clicked");

});

**👉** If there are many items, this approach becomes inefficient because it requires multiple event listeners.

**🔹 Example With Event Delegation (Efficient)**

document.getElementById("list").addEventListener("click", (event) => {

if (event.target.tagName === "LI") {

console.log(`${event.target.textContent} clicked`);

}

});

**👉** Here, we attach the event listener to the parent <ul>, and when any <li> is clicked, it is detected using event.target.

**🔹 Why Use Event Delegation?**

✔ **Better Performance:** Instead of adding multiple event listeners, we use just one, reducing memory usage.  
✔ **Handles Dynamically Added Elements:** Works for elements added to the DOM later.  
✔ **Simplifies Code:** Reduces redundancy and makes event handling cleaner.

🔹 Conclusion:

Event Delegation leverages event bubbling to manage multiple child elements efficiently with a single event listener on a parent element. It improves performance, keeps the code clean, and is especially useful for handling dynamically created elements.

**Q44) What is NaN in JavaScript?**

**Answer:** NaN stands for **"Not-a-Number"** in JavaScript. It is a special value that represents an invalid or undefined mathematical operation. NaN is a property of the global Number object (Number.NaN) and is of type **number**, even though it means "not a number."

**🔹 When does NaN occur?**

NaN is returned when an operation that is supposed to produce a number fails. Some common cases include:

console.log(0 / 0); // NaN (undefined division)

console.log(Math.sqrt(-1)); // NaN (square root of a negative number)

console.log(parseInt("abc")); // NaN (invalid number conversion)

console.log("hello" \* 5); // NaN (invalid mathematical operation)

🔹 Checking for NaN

Since NaN is not equal to itself (NaN !== NaN), you cannot check it using equality (== or ===). Instead, use:

console.log(isNaN(NaN)); // true

console.log(Number.isNaN(NaN)); // true (recommended)

🔹 Key Characteristics of NaN

1. NaN is of type number:

console.log(typeof NaN); // "number"

1. NaN is not equal to any value, including itself:

console.log(NaN === NaN); // false

1. Any arithmetic operation involving NaN results in NaN:

console.log(NaN + 5); // NaN

🔹 Conclusion

NaN is a special numeric value in JavaScript that represents an invalid or undefined mathematical operation. Since NaN !== NaN, always use Number.isNaN() to check for it. Understanding NaN helps in debugging unexpected results in calculations and conversions.

**Q45) What is an API?**

**Answer:**

API (**Application Programming Interface**) is a **set of rules** that allows different software applications to communicate with each other. It defines how requests and responses should be structured so that systems can exchange data efficiently.

For example, when you use a weather app, it fetches real-time weather data from a remote server using an API. Similarly, when a frontend React application interacts with a backend Node.js server, it does so through API requests, often using HTTP methods like GET, POST, PUT, and DELETE.

APIs can be **RESTful** (working with URLs and HTTP) or **GraphQL-based** (fetching only required data). They can be **public** (e.g., OpenWeather API) or **private** (used within a company’s internal systems). In web development, APIs play a crucial role in enabling **frontend-backend communication, third-party integrations, and microservices architecture**. 🚀

**Q46) What is the File Extension of JSON?**

**Answer:** .json

**Q47) What is the File Extension of JSON?**

**Answer:** JSON supports the following **six** data types:

✅ **String** – A sequence of characters enclosed in double quotes.  
**Example:** "name": "John Doe"

✅ **Number** – Includes integers and floating-point numbers.  
**Example:** "age": 30, "height": 5.9

✅ **Boolean** – Represents true or false values.  
**Example:** "isDeveloper": true

✅ **Null** – Represents an empty or unknown value.  
**Example:** "address": null

✅ **Object** – A collection of key-value pairs (like a dictionary or JavaScript object).  
**Example:**

**Q48) What are the HTTP method supported by REST?**

**Answer:** RESTful APIs use various HTTP methods to perform operations on resources. The most commonly used HTTP methods in REST are:

✅ **GET** – Used to retrieve data from a server. It does not modify any data.  
**Example:** Fetching a list of users from /users.

✅ **POST** – Used to send data to the server and create a new resource.  
**Example:** Submitting a new user to /users.

✅ **PUT** – Used to update an existing resource or replace it entirely.  
**Example:** Updating a user's information at /users/1.

✅ **PATCH** – Similar to PUT, but used for **partial updates** instead of replacing the entire resource.  
**Example:** Updating only the email of a user at /users/1.

✅ **DELETE** – Used to remove a resource from the server.  
**Example:** Deleting a user from /users/1.

These methods follow **CRUD (Create, Read, Update, Delete)** principles, making REST APIs efficient and standardized for web development. 🚀

**Q49) Can you use GET request instead of PUT to create a resourse?**

**Answer:**

🔹 **GET is meant for retrieving data, not modifying it** – It is considered a **safe** and **idempotent** operation, meaning it should not change the server state.

🔹 **PUT (or POST) is designed for resource creation or modification** – **PUT** is used when updating or creating a resource at a specific URL, and **POST** is commonly used for creating new resources.

🔹 **GET requests can be cached** – If you use **GET** for creating a resource, some systems might cache the response, causing unintended behavior (e.g., not actually creating the resource on the server).

**Correct Approach**

✔ Use **POST** to create a new resource.  
✔ Use **PUT** to update an existing resource (or create it if the URL is known and follows an "upsert" behavior).  
✔ Use **GET** only to retrieve data, not to modify it.

So, using **GET** to create a resource is not recommended and violates REST principles. 🚀

**Q50) What is JSON?**

**Answer:**

**JSON (JavaScript Object Notation)** is a lightweight data format used for storing and exchanging data between a client and a server. It is **easy to read, write, and parse**, making it a popular choice for APIs and configuration files.

JSON represents data in **key-value pairs**, similar to JavaScript objects, but it is language-independent and widely supported across different programming languages.

{

"name": "John Doe",

"age": 30,

"email": "john@example.com",

"skills": ["JavaScript", "React", "Node.js"],

"isDeveloper": true

}

**Key Features of JSON:**

✅ Lightweight – Uses minimal syntax, making it efficient for data exchange.  
✅ Human-readable – Easy to understand and edit.  
✅ Structured – Supports nested objects and arrays.  
✅ Used in APIs – Commonly used in RESTful APIs to send and receive data.

JSON is a preferred format in modern web development for seamless data communication between frontend and backend. 🚀

**Q51) What are CRUD Operations?**

**Answer:** CRUD stands for **Create, Read, Update, and Delete**, which are the four fundamental operations performed on a database or any data storage system. These operations are essential in web applications for managing data efficiently.

**CRUD Operations and HTTP Methods in REST APIs:**

✅ **Create** – Adds new data (Uses POST in REST).  
✅ **Read** – Retrieves existing data (Uses GET in REST).  
✅ **Update** – Modifies existing data (Uses PUT or PATCH in REST).  
✅ **Delete** – Removes data (Uses DELETE in REST).

**Example in a User Management System:**

* **Create:** Add a new user → POST /users
* **Read:** Get user details → GET /users/1
* **Update:** Modify user info → PUT /users/1 or PATCH /users/1
* **Delete:** Remove a user → DELETE /users/1

CRUD operations form the backbone of any full-stack application, allowing users to interact with databases efficiently. 🚀

**Q51) What is the Role of JSON.stringify()?**

**Answer:** JSON.stringify() is a JavaScript method that **converts a JavaScript object or array into a JSON-formatted string**. This is useful when sending data to a server, storing it in local storage, or logging it for debugging.

const user = {

name: "John Doe",

age: 30,

isDeveloper: true

};

const jsonString = JSON.stringify(user);

console.log(jsonString);

//output

{"name":"John Doe","age":30,"isDeveloper":true}

**Key Roles of JSON.stringify():**

✅ Converts JavaScript objects/arrays into JSON format.  
✅ Helps in sending structured data via APIs (fetch() or XMLHttpRequest).  
✅ Allows data storage in **localStorage** or **sessionStorage** as strings.

Without JSON.stringify(), JavaScript objects cannot be sent as raw data in APIs, as they must be in JSON format. 🚀

**Q52) What is the Role of JSON.parse()?**

**Answer:** JSON.parse() is a JavaScript method that **converts a JSON-formatted string back into a JavaScript object or array**. This is useful when receiving JSON data from an API, retrieving data from local storage, or working with JSON files.

const jsonString = '{"name": "John Doe", "age": 30, "isDeveloper": true}';

const userObject = JSON.parse(jsonString);

console.log(userObject.name); // Output: John Doe

console.log(userObject.age); // Output: 30

Key Roles of JSON.parse():

✅ Converts JSON strings into usable JavaScript objects or arrays.  
✅ Helps process API responses received as JSON-formatted strings.  
✅ Allows retrieval of structured data from localStorage or sessionStorage.

Without JSON.parse(), JSON data would remain as a string and could not be accessed or manipulated like a regular JavaScript object. 🚀

**Q52) If JavaScript is Single-Threaded, How Does It Handle Asynchronous Operations?**

**Answer:** JavaScript is **single-threaded**, meaning it has **one Call Stack** that executes code sequentially. However, JavaScript can handle **asynchronous operations** efficiently using the **Event Loop, Web APIs, and Callback Queue**. These components work together to enable non-blocking execution.

**How Asynchronous Operations Work in JavaScript?**

🟢 **1. Call Stack (Main Thread Execution)**

* JavaScript executes synchronous code **line by line** in the **Call Stack**.
* If an operation takes time (like fetching data), it must be handled **asynchronously** to prevent blocking.

🟢 **2. Web APIs / Background Tasks**

* Asynchronous tasks like setTimeout(), fetch(), Promises, and event listeners are **delegated to Web APIs** (provided by the browser or Node.js runtime).
* These tasks run **in the background** without blocking the main thread.

🟢 **3. Callback Queue / Microtask Queue**

* Once an async task is completed, its callback function is sent to the **Callback Queue** (for setTimeout & event listeners) or the **Microtask Queue** (for Promises & async/await).
* The **Microtask Queue** has higher priority and executes before the Callback Queue.

🟢 **4. Event Loop (Brings Tasks Back to the Call Stack)**

* The **Event Loop** continuously checks if the **Call Stack is empty**.
* If empty, it moves tasks from the **Microtask Queue** (first) and then the **Callback Queue** to the **Call Stack** for execution.

**Example: Asynchronous Behavior in JavaScript**

✅ setTimeout() goes to the Web API and executes last (via Callback Queue).  
✅ Promise.resolve().then() executes before setTimeout() (via Microtask Queue).

Conclusion

Even though JavaScript is single-threaded, it achieves asynchronous behavior using the Event Loop, Web APIs, and Queues, allowing smooth execution without blocking the main thread. 🚀

**Q37) How can you select an element from the DOM using JavaScript?**

Answer: To select an element from the DOM in JavaScript, you can use various methods. The most common ones are getElementById(), getElementsByClassName(), getElementsByTagName(), querySelector(), and querySelectorAll().

For example:

let element = document.getElementById('myElement');

getElementById() selects an element with a specific ID, returning a single element. On the other hand, querySelector() allows selecting an element using CSS selectors. If you need to select multiple elements, you can use querySelectorAll() or getElementsByClassName().

**Q38) What are the different ways to manipulate DOM elements using JavaScript?**

Answer: There are multiple ways to manipulate DOM elements in JavaScript. You can change the content, modify styles, add or remove classes, and even change attributes. Here are a few examples:

1. Changing Content: you can change the inner content of an element using innerHTML or textContent.

document.getElementById('myElement').innerHTML = 'New Content';

1. Changing Styles: You can modify an element's style using the style property.

document.getElementById('myElement').style.color = 'blue';

1. Adding or Removing Classes: You can add or remove classes from an element using classList..

document.getElementById('myElement').classList.add('active');

document.getElementById('myElement').classList.remove('inactive');

1. Manipulating Attributes: You can change an element's attributes using setAttribute() and getAttribute().

document.getElementById('myElement').setAttribute('src', 'image.jpg');

**Q39) What is the difference between innerHTML and textContent?**

Answer: innerHTML and textContent are both used to get or set the content of an element, but there are key differences:

* innerHTML: This property returns or sets the HTML content of an element. If used to set the content, it can inject HTML elements, including scripts and other tags, into the element. It can be a security risk if you're working with untrusted content (e.g., XSS attacks).

document.getElementById('myElement').innerHTML = '<b>Bold Text</b>';

* textContent: This property returns or sets the text content of an element, ignoring any HTML tags within the element. It is generally safer when you're only working with text and don’t want to risk injecting malicious code.

document.getElementById('myElement').textContent = 'This is plain text';

**Q40) How can you add a new element to the DOM?**

**Answer:** You can add a new element to the DOM by creating a new element and appending it to an existing element. First, use document.createElement() to create the new element, and then use methods like appendChild() or insertBefore() to add it to the document.

Examples:

let newElement = document.createElement('div');

newElement.innerHTML = 'New Element Content';

document.body.appendChild(newElement);

Here, a new <div> element is created and its content is set. The appendChild() method then adds it as the last child of the <body> element.

📝 Common Interview Follow-Ups(will be added later):

* How can you handle events using JavaScript in the DOM?
* Can you explain event delegation in JavaScript?
* What is the purpose of createElement() and how is it used?
* What is the classList property in JavaScript, and how can it be used for DOM manipulation?
* How would you remove an element from the DOM?
* What is the difference between parentNode and parentElement in DOM manipulation?
* Can you explain how to use querySelector() and querySelectorAll() to select multiple elements?
* What are the performance considerations when manipulating large DOM elements?

**Javascript 2nd Topic: Control Flow (Q41-Q45=>5 questions)**

**Q41) What is the role of if-else statements in JavaScript?**

Answer: In JavaScript, if-else statements are used to control the flow of execution based on specific conditions. The condition within the if block is evaluated, and if it is true, the code within the if block runs. If the condition is false, the else block, if present, gets executed. It is commonly used to make decisions in the code based on different conditions. Example:

let age = 18;

if (age >= 18) {

console.log('You are an adult.');

} else {

console.log('You are a minor.');

}

In this example, the condition checks if age is greater than or equal to 18. If it is, it logs "You are an adult." If not, it logs "You are a minor."

**Q42) What is a switch statement in JavaScript and how does it work?**

Answer: A switch statement in JavaScript provides a way to perform multiple comparisons based on the value of a variable. Unlike multiple if-else conditions, it checks a variable against several possible values and executes the corresponding block of code when a match is found. If no match is found, the default case, if provided, will run.

Example:

let fruit = 'apple';

switch (fruit) {

case 'apple':

console.log('Apple selected');

break;

case 'banana':

console.log('Banana selected');

break;

default:

console.log('Unknown fruit');

}

Here, the variable fruit is checked against the values in each case. If it matches 'apple', the code logs "Apple selected." If no case matches, the default block is executed.

**Q43) What are the differences between for, while, and do-while loops in Js?**

Answer: JavaScript provides three types of loops for repeating a block of code: for, while, and do-while.

1. for Loop : A for loop is typically used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement.

for (let i = 0; i < 5; i++) {

console.log(i);

}

1. while Loop : A while loop runs as long as the condition is true. It checks the condition before executing the code block, so if the condition is false initially, the code inside will not run.

let i = 0;

while (i < 5) {

console.log(i);

i++;

}

1. do-while Loop : A do-while loop executes the code block at least once, even if the condition is false, because it checks the condition after running the code.

let i = 0;

do {

console.log(i);

i++;

} while (i < 5);

**Q44) How do break and continue work in JavaScript loops?**

Answer: break and continue are used to control the flow of loops.

1. break : The break statement immediately exits the loop, regardless of the condition. This is useful when you want to terminate a loop early.

for (let i = 0; i < 10; i++) {

if (i === 5) break; // Exits the loop when i equals 5

console.log(i);

}

1. continue : The continue statement skips the current iteration and moves to the next one. It’s useful when you want to skip specific iterations based on a condition.

for (let i = 0; i < 10; i++) {

if (i === 5) continue; // Skips the iteration when i equals 5

console.log(i);

}

**Q45) What is the purpose of the return statement in JavaScript functions?**

Answer: The return statement in JavaScript is used to exit from a function and optionally send a value back to the caller. When a function reaches the return statement, it stops execution and returns the specified value. If no value is provided, it returns undefined.

Example:

function add(a, b) {

return a + b; // Returns the sum of a and b

}

let result = add(2, 3);

console.log(result); // Logs 5

In this example, the add function returns the sum of a and b. The value is then assigned to the variable result.

📝 Common Interview Follow-Ups(will be added later):

* Can you explain the concept of "fall-through" in a switch statement?
* What happens if there’s no break statement in a switch case?
* How do you avoid infinite loops in JavaScript?
* Can you use return in an arrow function, and how does it behave differently from a regular function?
* How would you handle multiple conditions with a switch statement?
* What is a labeled loop in JavaScript, and when might it be used?
* How do forEach loops differ from traditional for loops in JavaScript?
* How can you handle errors within loops in JavaScript?

**Javascript 3rd Topic: ES6 Features (Q46-Q53=>8 questions)**

**Q46) What are arrow functions in JavaScript?**

**Answer:** Arrow functions, introduced in ES6, provide a more concise syntax for writing functions. The syntax removes the need for the function keyword and uses the => syntax instead. They are often used for short functions or callbacks. One of the key differences from regular functions is that arrow functions do not have their own this value; instead, they inherit this from the surrounding context. **Example:**

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

console.log(add(2, 3)); // Outputs 5

In this example, the add function takes two parameters and returns their sum. The syntax is more compact than the traditional function declaration.

**Q47) What is the let keyword in JavaScript?**

**Answer** :The let keyword, introduced in ES6, is used to declare block-scoped variables, as opposed to var, which declares variables globally or function-scoped. This makes let more predictable when working with loops or conditional blocks, as its scope is limited to the block in which it is defined. **Example:**

let x = 10;

if (true) {

let x = 20;

console.log(x); // Outputs 20

}

console.log(x); // Outputs 10

Here, the value of x inside the if block is different from the value outside, demonstrating the block scope of let.

**Q48) What are template literals in JavaScript?**

**Answer:** Template literals, introduced in ES6, allow for easier string interpolation. They use backticks (`) instead of quotes and enable the inclusion of expressions inside strings using ${expression}. This feature simplifies the creation of strings that involve dynamic values. **Example:**

const name = 'John';

const greeting = `Hello, ${name}!`;

console.log(greeting); // Outputs "Hello, John!"

In this example, the value of name is dynamically inserted into the string using template literals.

**Q49) What are default parameters in JavaScript?**

**Answer:** Default parameters in ES6 allow you to assign default values to function parameters if they are not provided when the function is called. This makes it easier to handle optional parameters without needing to check if they are undefined. **Example:**

function greet(name = 'Guest') {

console.log(`Hello, ${name}`);

}

greet(); // Outputs "Hello, Guest"

greet('Alice'); // Outputs "Hello, Alice"

In this example, the name parameter has a default value of 'Guest', which is used if no argument is passed to the function.

**Q50) What is destructuring in JavaScript?**

**Answer:** Destructuring, introduced in ES6, allows you to unpack values from arrays or objects into variables in a more convenient and concise way. It makes it easier to extract multiple properties from an object or elements from an array in a single statement. **Examples:**

* **Object Destructuring:**

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

const { name, age } = person;

console.log(name, age); // Outputs "John 30"

* **Array Destructuring:**

const numbers = [1, 2, 3];

const [a, b] = numbers;

console.log(a, b); // Outputs "1 2"

In these examples, the values of name and age are extracted from the person object, and the values 1 and 2 are extracted from the numbers array.

**Q51) What are Promise objects in JavaScript?**

**Answer:** A Promise in JavaScript represents an asynchronous operation that can be either resolved or rejected. Promises are used to handle asynchronous operations, allowing you to handle successful or failed results with then() and catch() methods. They simplify asynchronous code and make it more readable. **Example:**

const myPromise = new Promise((resolve, reject) => {

let success = true;

if (success) {

resolve('Operation successful');

} else {

reject('Operation failed');

}

});

myPromise

.then(result => console.log(result)) // Outputs "Operation successful"

.catch(error => console.log(error)); // Outputs if rejected

In this example, the Promise resolves with a success message and handles it using then(). If it fails, it is caught by catch().

**Q52) What are classes in JavaScript?**

**Answer:** ES6 introduced classes to JavaScript, providing a more structured way of creating objects and handling inheritance. Classes are syntactical sugar over the existing prototype-based inheritance and allow for a more object-oriented approach to defining objects and methods.**Examples:**

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

greet() {

console.log(`Hello, my name is ${this.name}`);

}

}

const john = new Person('John', 30);

john.greet(); // Outputs "Hello, my name is John"

In this example, a Person class is defined with a constructor to initialize name and age, and a method greet() to print a greeting.

**Q53) What is the spread operator in JavaScript?**

**Answer:** The spread operator (...) in JavaScript allows you to expand elements from an iterable (like an array or object) into individual elements. It simplifies copying, merging, or combining arrays and objects..**Example:**

* **Array Spread:**

const arr1 = [1, 2, 3];

const arr2 = [...arr1, 4, 5];

console.log(arr2); // Outputs [1, 2, 3, 4, 5]

* **Object Spread:**

const obj1 = { name: 'John', age: 30 };

const obj2 = { ...obj1, city: 'New York' };

console.log(obj2); // Outputs { name: 'John', age: 30, city: 'New York' }

In these examples, the spread operator is used to copy and merge arrays and objects.

📝 Common Interview Follow-Ups(will be added later):

* How does the const keyword differ from let in JavaScript?
* Can you explain the difference between the call(), apply(), and bind() methods in JavaScript?
* What are the benefits of using arrow functions over regular functions?
* How do you handle asynchronous operations without using Promises in JavaScript?
* What are getter and setter methods in JavaScript classes?
* Can you explain the concept of "hoisting" in JavaScript?
* How do async and await work in JavaScript? What’s their relation to Promises?
* Can you explain the concept of "modules" in JavaScript ES6?

**Javascript 4th Topic: APIs (Q54-Q60=>7 questions)**

**Q54) What is the Fetch API in JavaScript?**

**Answer:** The Fetch API provides a modern way to make asynchronous HTTP requests in JavaScript. It returns a Promise that resolves to the Response object representing the response to the request. The Fetch API is a more flexible and powerful replacement for the older XMLHttpRequest. You can use fetch() to send GET, POST, and other HTTP requests to retrieve or send data from a server. **Example:**

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.log('Error:', error));

In this example, fetch() is used to send a GET request. The response is converted to JSON format, and the data is logged to the console. Any errors are caught and logged as well.

**Q55) What is the difference between fetch() and XMLHttpRequest?**

**Answer:** The main difference between fetch() and XMLHttpRequest is that fetch() is promise-based, making it easier to work with asynchronous code, while XMLHttpRequest uses callbacks. This makes fetch() more modern, cleaner, and easier to read and maintain.

* **Promises**: With fetch(), you can use .then() and .catch() for handling success and failure, while XMLHttpRequest uses callback functions.
* **Streams**: fetch() supports streams, allowing you to handle large responses as they are received, whereas XMLHttpRequest doesn’t have this feature.
* **No Built-In Support for JSON Parsing**: fetch() requires you to manually parse JSON responses, while XMLHttpRequest parses responses based on their content type.

**Example with fetch():**

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.log('Error:', error));

**Q56) How does the localStorage API work in JavaScript?**

**Answer** The localStorage API allows you to store key-value pairs in the browser's storage, which persists even after the page is reloaded or the browser is closed. It is synchronous and can store data as strings. You can use localStorage to save user preferences, settings, or other information that should be retained across sessions.**Example:**

localStorage.setItem('username', 'JohnDoe');

let username = localStorage.getItem('username');

console.log(username); // Outputs "JohnDoe"

In this example, the setItem() method stores a value, and the getItem() method retrieves the value. Data stored in localStorage will persist until explicitly removed with removeItem() or cleared using clear().

**Q57) What is the sessionStorage API in JavaScript?**

**Answer:** sessionStorage is similar to localStorage, but with one key difference: it only persists data for the duration of the page session. A session ends when the browser or tab is closed. It is typically used for storing temporary data that only needs to be available during a single session.**Example:**

sessionStorage.setItem('theme', 'dark');

let theme = sessionStorage.getItem('theme');

console.log(theme); // Outputs "dark"

In this example, the setItem() method stores a value in the session storage, and the getItem() method retrieves it. The data will be cleared when the session ends (i.e., when the tab is closed).

**Q58) What is the Geolocation API in JavaScript?**

**Answer:** The Geolocation API allows websites to access the geographical location of a user's device. It is commonly used for applications like maps or location-based services. The API provides methods to get the user's current position or watch for changes in their position..**Example:**

navigator.geolocation.getCurrentPosition(function(position) {

console.log('Latitude: ' + position.coords.latitude);

console.log('Longitude: ' + position.coords.longitude);

});

In this example, getCurrentPosition() is used to retrieve the user's current geographical coordinates. The position object contains the coords property with latitude and longitude.

**Q59) How does the Web Storage API differ from cookies in JavaScript?**

Answer: The Web Storage API (which includes localStorage and sessionStorage) differs from cookies in several ways:

* Storage Size: Web Storage can store larger amounts of data (up to 5-10MB per domain) compared to cookies, which are limited to around 4KB.
* Lifetime: localStorage persists data across sessions, while cookies can have expiration dates set by the server.
* Data Handling: Cookies are sent with every HTTP request, while data in Web Storage is stored on the client side and not transmitted with requests, improving performance.
* Simplicity: Web Storage is easier to use and more efficient for client-side storage compared to cookies.

**Q60) What is the Notification API in JavaScript?**

**Answer:** The Notification API allows web pages to display notifications to the user, even if the page is not in the foreground. This API is often used in conjunction with service workers to send push notifications for real-time updates, such as messages or alerts...**Example:**

if (Notification.permission === 'granted') {

new Notification('Hello, you have a new message!');

} else {

Notification.requestPermission().then(permission => {

if (permission === 'granted') {

new Notification('Hello, you have a new message!');

}

});

}

In this example, the Notification object is used to display a notification. Before sending a notification, the browser must request permission to show notifications.

📝 Common Interview Follow-Ups(will be added later):

* How would you handle errors in the Fetch API?
* What are some common use cases for the localStorage and sessionStorage APIs?
* How do you ensure compatibility for the Geolocation API across different browsers?
* How do you handle JSON responses from APIs using the Fetch API?
* What are some security concerns when using localStorage and sessionStorage?
* How can you set expiration for localStorage data?
* How would you handle sending data through the Fetch API using POST requests?
* How can you use the Notification API to send push notifications with service workers?