HTML QUESTIONS

Q1. is it a tag of html? If not, what is it and why do we use it?

Answer 1 html - >

Yes, **<!DOCTYPE html>** is indeed a tag in HTML, specifically called the Document Type Declaration (DTD). It is used to specify the version of HTML or XHTML that the web page is written in.

By using the **<!DOCTYPE html>** declaration, you ensure that your HTML code is interpreted correctly by modern web browsers, which helps ensure consistency and compatibility across different devices and platforms. It's considered good practice to include this declaration at the beginning of your HTML documents.

Q2 Explain Semantic tags in html? And why do we need it?

Answer 2 html - >

Semantic tags in HTML are a set of elements that provide meaning and structure to the content within a web page. These tags convey the purpose and significance of different parts of the document, making it more accessible to both humans and search engines.

Here are some commonly used semantic tags in HTML5:

1. **<header>**: Represents the introductory content or a container for a group of introductory content at the top of a page or section.

Q3 Differentiate between HTML Tags and Elements?

Answer 3 html - >

In HTML, there is a distinction between tags and elements. Here's the difference between the two:

HTML Tags:

* Tags are the building blocks of HTML markup.
* They are used to define the structure and appearance of elements within an HTML document.
* Tags are represented by angle brackets (< and >) and usually come in pairs: an opening tag and a closing tag.
* Opening tags contain the name of the element and any attributes, while closing tags are identical except for a forward slash (/) placed before the element name.
* Example: **<p>** is an opening tag, and **</p>** is a closing tag for a paragraph element.

HTML Elements:

* Elements are made up of tags, their content, and any attributes.
* An HTML element consists of an opening tag, the content (text or other nested elements), and a closing tag.
* Elements represent the actual components or parts of a webpage, such as headings, paragraphs, images, links, lists, etc.
* Example: **<p>This is a paragraph.</p>** represents a paragraph element with the opening tag **<p>**, the content "This is a paragraph.", and the closing tag **</p>**.

In summary, tags are the markup symbols that define the structure and behavior of HTML elements, while elements consist of tags, their content, and any associated attributes. Tags are used to enclose and define elements within an HTML document

Q4. Build Your Resume using HTML only.

Answer 4 html - ><https://github.com/jagtapnimisha2/Resume_build>

Q6. What are some of the advantages of HsTML5 over its previous version?

Answer 6 html - >

**Advantages**

Multimedia support.

Short and simple syntax.

Improved security features.

Inlcude semantic tags.

Cross-platform support.

Q7. Create a simple Music player using html only?

Answer 7 html - ><https://github.com/jagtapnimisha2/music_player>

Q8 What is the difference between tag and **Error! Filename not specified.**tag?

Answer 8 html - >

**<img>** tag: The **<img>** tag is specifically used to insert an image into an HTML document. It is a self-closing tag and does not require a closing tag. The **<img>** tag requires the **src** attribute, which specifies the source URL or path of the image file. Additionally, the **alt** attribute is used to provide alternative text for the image, which is displayed if the image cannot be loaded or for accessibility purposes.

Example:

htmlCopy code

<img src="image.jpg" alt="Description of the image">

In summary, the **<figure>** tag is a container element used to group an image or media element along with its caption using the **<figcaption>** tag, while the **<img>** tag is solely used for embedding images into an HTML document.

1. Top of Form

Q9 What’s the difference between html tag and attribute and give example of some global attributes?

Answer 9 html - >

In HTML, there is a distinction between tags and attributes. Let me explain the difference:

HTML Tags: HTML tags are used to define elements and structure within an HTML document. They are represented by the opening and closing angle brackets (**<** and **>**). Tags enclose content and provide meaning to the browser about how the content should be interpreted and displayed.

Example: **<p>** is a tag used to define a paragraph element.

HTML Attributes: HTML attributes provide additional information or properties to HTML elements. They are specified within the opening tag of an element using name-value pairs. Attributes modify the behavior or appearance of elements and are used to provide extra information about the elements.

Example: In **<img src="image.jpg" alt="Description">**, the **src** and **alt** attributes provide information about the image source and alternate text.

Global Attributes: Global attributes are attributes that can be used on any HTML element, regardless of the element's specific purpose or type. These attributes provide common functionalities that apply to various elements.

Some examples of global attributes are:

1. **class**: Specifies one or more class names for an element, allowing the element to be styled with CSS or targeted by JavaScript.
2. **id**: Provides a unique identifier for an element, which can be used for styling or scripting purposes.
3. **style**: Defines inline CSS styles to be applied to an element.
4. **title**: Specifies additional information about an element, typically displayed as a tooltip when the user hovers over the element.
5. **lang**: Specifies the language of the content within an element.
6. **data-\***: Allows custom data attributes to be added to elements, which can be accessed via JavaScript for various purposes.

These global attributes can be used on different HTML elements to enhance their functionality or provide additional information, irrespective of the specific element type.

Q10 build Table which looks like the given image

Answer 10 html - > <https://github.com/jagtapnimisha2/Resume_build>

← CSS QUESTIONS →

Q1 Whats Box Model in CSS & Which CSS Properties are part of it ?

Answer 1 Css - >

The box model is a fundamental concept in CSS that describes how elements are structured and displayed on a web page. It consists of several components that define the total space occupied by an element, including its content, padding, border, and margin.

The components of the box model are:

1. Content: This is the actual content of the element, such as text, images, or other HTML elements.
2. Padding: The padding is a transparent area surrounding the content inside the element. It creates space between the content and the element's border.
3. Border: The border is a line or outline that surrounds the padding and content of the element. It can have a specific width, style, and color.
4. Margin: The margin is the transparent area outside the element, providing spacing between the element and adjacent elements.

The following CSS properties are directly related to the box model:

1. **width** and **height**: These properties define the dimensions of the content box.
2. **padding**: Sets the padding space around the content.
3. **border**: Specifies the properties of the element's border, including width, style, and color.
4. **margin**: Sets the margin space around the element.
5. **box-sizing**: Determines how the total width and height of an element are calculated. The default value is **content-box**, but you can also use **border-box**, which includes padding and border within the defined width and height.

These properties allow you to manipulate and control the size, spacing, and appearance of elements on a web page based on the box model concept. Understanding the box model is crucial for precise layout and positioning of elements in CSS.

Q2 What are the Different Types of Selectors in CSS & what are the advantages of them?

Answer 2 Css - >

CSS selectors are used to target and select specific HTML elements for styling. There are various types of CSS selectors available, each with its own advantages and use cases. Here are some commonly used types of selectors:

1. Type Selectors: Type selectors target elements based on their HTML tag names. For example, **p** selects all **<p>** elements. Type selectors are simple and straightforward to use.

Advantages: Type selectors are easy to understand and can be used to apply styles to all elements of a specific type.

1. Class Selectors: Class selectors target elements based on their assigned class attribute. They are denoted with a dot (**.**) followed by the class name. For example, **.highlight** selects all elements with the class "highlight".

Advantages: Class selectors offer flexibility and reusability. Multiple elements can share the same class, allowing you to apply consistent styles to those elements.

1. ID Selectors: ID selectors target elements based on their unique ID attribute. They are denoted with a hash (**#**) followed by the ID name. For example, **#header** selects the element with the ID "header".

Advantages: ID selectors provide a specific and unique way to target individual elements. They are useful when you need to apply styles or perform specific actions on a particular element.

1. Attribute Selectors: Attribute selectors target elements based on their attribute values. They can select elements with specific attributes, attribute values, or attribute value patterns. For example, **[type="submit"]** selects all elements with the attribute **type** set to "submit".

Advantages: Attribute selectors allow you to target elements based on their attributes, which can be handy when working with dynamic or specific attribute values.

1. Pseudo-classes and Pseudo-elements: Pseudo-classes and pseudo-elements target elements based on their state or position within the document. Examples include **:hover**, **:first-child**, **::before**, and **::after**. They are denoted with a colon (**:**) or double colon (**::**) followed by the pseudo-class or pseudo-element name.

Advantages: Pseudo-classes and pseudo-elements enable you to style elements based on various states or positions, such as hovering over an element or adding content before or after an element.

The advantages of different CSS selectors include flexibility, specificity, reusability, and the ability to target elements based on various criteria. Using a combination of these selectors allows you to apply styles precisely to desired elements, create consistent styles, and enhance the overall appearance and functionality of your web pages.

Q3 What is VW/VH & How its different from PX?

Answer 3 Css - >

VW (Viewport Width) and VH (Viewport Height) are relative units of measurement in CSS that allow you to specify sizes based on the dimensions of the viewport, which is the visible area of the web page.

Here's how VW and VH differ from PX (pixels):

1. VW (Viewport Width): VW is a unit of measurement that represents a percentage of the viewport width. For example, 1vw is equal to 1% of the viewport width. If the viewport width is 1000 pixels, 1vw would be equivalent to 10 pixels.

Example:

width: 50vw; /\* Set the width to be 50% of the viewport width \*/

1. VH (Viewport Height): VH is a unit of measurement that represents a percentage of the viewport height. Similar to VW, 1vh is equal to 1% of the viewport height. For instance, if the viewport height is 800 pixels, 1vh would be equal to 8 pixels.

Example:

height: 75vh; /\* Set the height to be 75% of the viewport height \*/

1. 3. PX (Pixels): PX is an absolute unit of measurement that represents a fixed number of pixels. It is not relative to the viewport size. 1px corresponds to a single pixel on the screen, regardless of the viewport dimensions.

Example:

font-size: 16px; /\* Set the font size to be 16 pixels \*/

Q4 Whats difference between Inline, Inline Block and block ?

Answer 4 Css - >

The differences between **inline**, **inline-block**, and **block** are related to how HTML elements are displayed and how they interact with other elements. Here's a breakdown of each display property:

1. Inline:

* Inline elements do not start on a new line and only occupy the space necessary for their content.
* They cannot have width and height properties applied to them.
* Examples of inline elements include **<span>**, **<a>**, **<strong>**, and **<em>**.
* Inline elements can be placed next to each other horizontally.

1. Inline-block:

* Inline-block elements are similar to inline elements in that they do not start on a new line.
* However, they can have width, height, padding, and margin properties applied to them.
* They respect the box model and allow for block-level properties while still being inline.
* Examples of inline-block elements include **<input>**, **<button>**, and **<img>**.

1. Block:

* Block elements start on a new line and occupy the full width available.
* They can have width, height, padding, and margin properties applied to them.
* Block elements create a block-level formatting context, meaning they stack vertically.
* Examples of block elements include **<div>**, **<p>**, **<h1>** to **<h6>**, and **<ul>**.

In summary:

* Inline elements do not start on a new line and only occupy the necessary space for their content.
* Inline-block elements are similar to inline elements but can have width, height, and other block-level properties.
* Block elements start on a new line, occupy the full width available, and stack vertically.

The choice of display property depends on the desired layout and behavior you want to achieve for your HTML elements.

Q5 How is Border-box different from Content Box?

Answer 5 Css - >

In CSS, the **box-sizing** property allows you to define how the total width and height of an element are calculated. It determines how the element's content, padding, border, and margin are taken into account when sizing and positioning the element. There are two main values for the **box-sizing** property: **content-box** and **border-box**.

1. Content Box:

* The default value of the **box-sizing** property is **content-box**.
* In the content box model, the specified width and height of an element only include the content area.
* Any padding, border, or margin added to the element will increase the overall width and height of the element.

.box {

width: 200px; /\* Only includes the content area \*/

padding: 20px;

border: 2px solid #000;

margin: 10px;

/\* The total width will be: 200px (content) + 40px (padding) + 4px (border) + 20px (margin) = 264px \*/

/\* The total height will be: 20px (content) + 40px (padding) + 4px (border) + 20px (margin) = 84px \*/

}

1. Border Box:

* When the **box-sizing** property is set to **border-box**, the specified width and height of an element include the content, padding, and border.
* The margin is still added separately to the total width and height of the element.

Example:

.box {

box-sizing: border-box;

width: 200px; /\* Includes content, padding, and border \*/

padding: 20px;

border: 2px solid #000;

margin: 10px;

/\* The total width will be: 200px (content + padding + border) + 20px (margin) = 220px \*/

/\* The total height will be: 20px (content + padding + border) + 20px (margin) = 60px \*/

}

Q6 What’s z-index and How does it Function ?

Answer 6 Css - >

The **z-index** property in CSS controls the stacking order or layering of positioned elements on a web page along the z-axis, which is the axis that represents depth or distance from the viewer. It specifies the relative z-order of elements and determines which elements appear on top of others when they overlap.

Here's how **z-index** functions:

1. Stacking Context: The **z-index** property works within a stacking context. A stacking context is an area in which elements are stacked relative to each other based on their **z-index** values. Each stacking context can have its own stacking order independent of other stacking contexts.
2. Stacking Order: The stacking order determines which elements are visually in front or behind other elements. Elements with a higher **z-index** value are stacked on top of elements with lower **z-index** values within the same stacking context.
3. Default Stacking Order: By default, elements have a **z-index** value of **auto**, which means they follow the stacking order based on their position in the HTML document. Elements that come later in the HTML markup are placed on top of elements that come earlier, assuming no other **z-index** values are specified.
4. Specifying z-index: To explicitly control the stacking order, you can assign a **z-index** value to an element. The **z-index** value can be positive, negative, or zero. Elements with higher **z-index** values will be stacked on top of elements with lower **z-index** values within the same stacking context.

Example:

.element1 {

z-index: 1;

}

.element2 {

z-index: 2;

}

In this example, **element2** will be stacked on top of **element1** because it has a higher **z-index** value.

1. Stacking Context Hierarchy: Elements within a stacking context are stacked above elements in lower-level stacking contexts, regardless of their **z-index** values. Therefore, even if an element within a lower-level stacking context has a higher **z-index** value than an element in a higher-level stacking context, the higher-level element will still be stacked above the lower-level element.

Understanding the **z-index** property allows you to control the stacking order of elements and create visually layered layouts on your web page. It's important to consider the stacking context hierarchy and properly assign **z-index** values to achieve the desired stacking order.

Q6 What’s Grid & Flex and difference between them?

Answer 6 Css - >

Grid and Flexbox are two powerful CSS layout systems that offer different approaches to building and arranging elements within a container. Here's a comparison of Grid and Flexbox:

1. Grid:

* Grid is a two-dimensional layout system that allows you to create complex grid structures with rows and columns.
* It enables you to define both the layout and alignment of elements in both horizontal and vertical directions.
* Grid is best suited for creating grid-like structures, such as website layouts with multiple sections or grids of images.
* Grid provides precise control over the placement and sizing of elements within the grid.
* It uses the properties **display: grid** on the container and **grid-column** and **grid-row** on the items to define the grid structure.

Advantages of Grid:

* Provides a powerful and flexible grid system.
* Allows you to define both row and column tracks, and their sizing.
* Offers fine-grained control over element placement and alignment within the grid.
* Supports responsive design with media queries for different grid configurations.

1. Flexbox:

* Flexbox is a one-dimensional layout system that focuses on arranging elements along a single direction (either horizontally or vertically).
* It is ideal for creating layouts with flexible and dynamic content, such as navigation menus, flexible item arrangements, and centering elements.
* Flexbox automatically adjusts the size of items to fit the container and distribute remaining space.
* It uses the properties **display: flex** on the container and various flex-related properties (**flex-direction**, **justify-content**, **align-items**, etc.) on the items to control the layout.

Advantages of Flexbox:

* Provides a simpler approach to layout compared to Grid.
* Offers automatic distribution of space and alignment of items.
* Supports responsive design and dynamic layouts.
* Enables easy centering of items vertically and horizontally.

In summary, Grid is best suited for creating complex two-dimensional layouts with precise control over rows, columns, and item placement. Flexbox, on the other hand, is more suitable for arranging elements along a single direction and providing flexible and dynamic layouts. Both Grid and Flexbox have their strengths and can be used together or separately depending on the specific layout requirements of your project.

Q7 Difference between absolute and relative and sticky and fixed position explain with example

Answer 7 Css - >

1. Absolute Positioning:

* When an element is positioned **absolute**, it is removed from the normal document flow, and its position is relative to its nearest positioned ancestor or the initial containing block if no positioned ancestor is found.
* Absolute positioning allows you to precisely position an element using the **top**, **right**, **bottom**, and **left** properties.
* Other elements on the page will ignore the space occupied by the absolutely positioned element.

Example:

htmlCopy code

<div class="container"> <div class="absolute-box"></div> </div>

cssCopy code

.container { position: relative; /\* Positioned ancestor \*/ } .absolute-box { position: absolute; top: 20px; left: 20px; }

n this example, the **.absolute-box** element is positioned absolutely within the **.container** element. It will be positioned 20 pixels from the top and 20 pixels from the left of its nearest positioned ancestor (**.container**).

1. Relative Positioning:

* When an element is positioned **relative**, it remains in the normal document flow, but you can adjust its position relative to its normal position.
* Relative positioning allows you to offset an element using the **top**, **right**, **bottom**, and **left** properties without affecting the position of other elements.

Example:

Sure! Let's explore the differences between **absolute** and **relative** positioning, as well as **sticky** and **fixed** positioning, along with examples:

1. Absolute Positioning:

* When an element is positioned **absolute**, it is removed from the normal document flow, and its position is relative to its nearest positioned ancestor or the initial containing block if no positioned ancestor is found.
* Absolute positioning allows you to precisely position an element using the **top**, **right**, **bottom**, and **left** properties.
* Other elements on the page will ignore the space occupied by the absolutely positioned element.

Example:

htmlCopy code

<div class="container"> <div class="absolute-box"></div> </div>

cssCopy code

.container { position: relative; /\* Positioned ancestor \*/ } .absolute-box { position: absolute; top: 20px; left: 20px; }

In this example, the **.absolute-box** element is positioned absolutely within the **.container** element. It will be positioned 20 pixels from the top and 20 pixels from the left of its nearest positioned ancestor (**.container**).

1. Relative Positioning:

* When an element is positioned **relative**, it remains in the normal document flow, but you can adjust its position relative to its normal position.
* Relative positioning allows you to offset an element using the **top**, **right**, **bottom**, and **left** properties without affecting the position of other elements.

Example:

htmlCopy code

<div class="relative-box"></div>

cssCopy code

.relative-box { position: relative; top: 10px; left: 10px; }

In this example, the **.relative-box** element is positioned relatively. It will be shifted 10 pixels from the top and 10 pixels from the left of its normal position, but other elements on the page will still respect its space.

1. Sticky Positioning:

* When an element is positioned **sticky**, it behaves like **relative** positioning until it reaches a specified threshold, at which point it becomes "stuck" and remains fixed relative to the viewport.
* Sticky positioning is commonly used for creating sticky headers or sidebars that remain visible as the user scrolls.

Example:

htmlCopy code

<div class="sticky-box"></div>

cssCopy code

.sticky-box { position: sticky; top: 20px; }

In this example, the **.sticky-box** element is positioned sticky. It will behave like **relative** positioning until the user scrolls to a point where its top position is 20 pixels from the viewport's top. At that point, it will become "stuck" and maintain its position relative to the viewport.

1. Fixed Positioning:

* When an element is positioned **fixed**, it is removed from the normal document flow and remains fixed relative to the viewport.
* Fixed positioning is commonly used for creating elements such as fixed navigation bars or overlay elements that stay in a fixed position even when the page is scrolled.

Example:

htmlCopy code

<div class="fixed-box"></div>

cssCopy code

.fixed-box { position: fixed; top: 20px; right: 20px; }

In this example, the **.fixed-box** element is positioned fixed. It will be fixed 20 pixels from the top and 20 pixels from the right of the viewport, regardless of scrolling.

In summary, **absolute** and **relative** positioning allow you to adjust the position of elements within the normal document flow, while **sticky** and **fixed** positioning enable elements to be positioned relative to the viewport or their nearest positioned ancestor. Each type of positioning has its own use cases and can be used to achieve different layout effects on a web page.

Top of Form

Q9. Build given layout using grid or flex see below image for reference?

Answer 9 CSS - ><https://github.com/jagtapnimisha2/Layout>

Q.10 Build Responsive Layout both desktop and mobile and Tablet, see below image for

reference ?

Answer 10 CSS - ><https://github.com/jagtapnimisha2/Responsive_layout>

Q12 What are Pseudo class in CSS & How its different From Pseudo Elements?

Answer 12 CSS - >

Pseudo-classes and pseudo-elements are CSS selectors that allow you to target specific parts or states of elements in a document. While they might sound similar, they have different purposes and syntax.

Pseudo-classes:

* Pseudo-classes select and style elements based on their state or relationship to the document or user interaction.
* They are preceded by a colon (**:**) and added to a selector to target a specific state or condition of an element.
* Pseudo-classes are used to style elements such as links, form inputs, and elements based on user interaction (e.g., **:hover**, **:focus**, **:active**).
* Examples of pseudo-classes: **:hover**, **:focus**, **:first-child**, **:nth-child(n)**, **:not(selector)**.

Example:

cssCopy code

a:hover { color: red; } input:focus { outline: 2px solid blue; }

In this example, the **:hover** pseudo-class targets links (**<a>**) when the mouse hovers over them and changes their color to red. The **:focus** pseudo-class targets input elements when they receive focus and adds a blue outline.

Pseudo-elements:

* Pseudo-elements select and style a specific part of an element or generate additional content in the document.
* They are preceded by a double colon (**::**) and added to a selector to target a specific part of an element or generate content.
* Pseudo-elements are used to style and modify specific parts of elements, such as the first line of a paragraph (**::first-line**) or to generate content before or after an element (**::before**, **::after**).
* Examples of pseudo-elements: **::before**, **::after**, **::first-line**, **::first-letter**.

Example:

cssCopy code

p::first-line { font-weight: bold; } h1::before { content: "Chapter: "; }

In this example, the **::first-line** pseudo-element targets the first line of a paragraph (**<p>**) and applies a bold font weight. The **::before** pseudo-element is used with an **h1** element and generates content before the element, adding the text "Chapter: ".

To summarize, pseudo-classes are used to style elements based on their state or relationship to the document, while pseudo-elements target specific parts of elements or generate additional content. They have different syntaxes, with pseudo-classes using a colon (**:**) and pseudo-elements using a double colon (**::**), although for compatibility reasons, some pseudo-elements also accept a single colon.

← JavaScript Questions →

Q1 What is Hoisting in JavaScript?

Answer 1 JavaScript- >

In JavaScript, hoisting is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed. This means that regardless of where variables and functions are declared in the code, they are treated as if they were declared at the beginning of their scope.

Q2 What are different higher order functions in JS? What is the difference between .map()

and .forEach() ?

Answer 2 JavaScript- >

Key differences between **map** and **forEach**:

* Return value: **map** returns a new array with the transformed values, while **forEach** does not return any value.
* Use case: **map** is commonly used when you want to transform each element of an array and create a new array. **forEach** is used when you want to perform a certain action for each element of an array without creating a new array.
* Side effects: **map** does not modify the original array, whereas **forEach** operates directly on the elements of the original array.
* Chaining: Since **map** returns a new array, it can be easily chained with other array methods. **forEach** does not return anything, so it cannot be chained directly.

It's important to choose the appropriate method based on your requirements. If you need a new transformed array, use **map**. If you simply want to iterate over the elements and perform an action, use **forEach**.

Q3 What is the difference between .call() .apply() and .bind()? explain with an example

Answer 3 JavaScript- >

In JavaScript, the methods **.call()**, **.apply()**, and **.bind()** are used to manipulate the context (the value of **this**) of a function. They are often used to borrow methods from other objects, set the value of **this** explicitly, or create new functions with a specific context. Here's an explanation of each method along with an example:

1. **.call()**: The **.call()** method is used to invoke a function immediately, providing a specific **this** value and passing arguments individually (comma-separated).

Syntax:

functionName.call(thisValue, arg1, arg2, ...);

Example:

const person = {

name: "John",

greet: function (message) {

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

},

};

const friend = {

name: "Jane",

};

person.greet.call(friend, "Hello");

// Output: "Hello, Jane!"

1. **apply()**: Similar to **.call()**, the **.apply()** method invokes a function immediately and sets the value of **this**, but it accepts arguments as an array or an array-like object.

Syntax:

javascriptCopy code

functionName.apply(thisValue, [arg1, arg2, ...]);

Example:

javascriptCopy code

const numbers = [1, 2, 3, 4, 5]; const maxNumber = Math.max.apply(null, numbers); console.log(maxNumber); // Output: 5

In this example, **Math.max()** is a built-in function that returns the maximum value from a set of numbers. Since **Math.max()** expects individual arguments, we use **.apply()** to pass the **numbers** array as arguments.

1. **.bind()**: The **.bind()** method creates a new function with a specific **this** value, but unlike **.call()** and **.apply()**, it does not immediately invoke the function. Instead, it returns a new function that can be invoked later.

Syntax:

javascriptCopy code

const newFunction = functionName.bind(thisValue, arg1, arg2, ...);

Example:

javascriptCopy code

const person = { name: "John", greet: function () { console.log(`Hello, ${this.name}!`); }, }; const friend = { name: "Jane", }; const greetFriend = person.greet.bind(friend); greetFriend(); // Output: "Hello, Jane!"

In this example, **bind()** creates a new function **greetFriend** that has the **this** value set to **friend**. When **greetFriend()** is called, it invokes the original **greet** function with the desired context.

To summarize:

* **.call()** and **.apply()** immediately invoke the function and set the **this** value, with the difference being the way arguments are passed.
* **.bind()** creates a new function with the desired **this** value and returns it, allowing for deferred invocation.

These methods provide flexibility in manipulating the context of functions and are commonly used in scenarios where you want to explicitly set the value of **this** or borrow methods from other objects.

Top of Form

Q4 Explain Event bubbling and Event Capturing in JavaScript with suitable examples

Answer 4 JavaScript- >

1. Event Bubbling: Event bubbling is the default behavior in which an event is first handled at the deepest (innermost) element where it occurred and then bubbles up through its parent elements in the DOM hierarchy. This means that after the innermost element handles the event, the event is propagated to its parent, then the parent's parent, and so on, until it reaches the outermost (top) element.

<div class="outer">

<div class="inner">

<button id="btn">Click me</button>

</div>

</div>

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

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

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

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

console.log('Button clicked');

});

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

console.log('Inner div clicked');

});

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

console.log('Outer div clicked');

});

1. Event Capturing: Event capturing is the opposite of event bubbling. In this phase, the event is first captured at the outermost element and then moves inward to the innermost element. This is less commonly used than event bubbling, but it can be leveraged by setting the **capture** option to **true** when adding an event listener.

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

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

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

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

console.log('Button clicked');

}, true);

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

console.log('Inner div clicked');

}, true);

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

console.log('Outer div clicked');

}, true);

Q5 What is function currying with example?

Answer 5 JavaScript- >

Function currying is a technique in JavaScript where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument. It allows you to create specialized versions of a function by fixing some arguments in advance. Curried functions can be useful for creating reusable function compositions or for partial function application.

function add(x) {

return function(y) {

return x + y;

}

}

const addFive = add(5);

console.log(addFive(3)); // Output: 8

Q7 What are promises? What are the different states of a promise? Support your answer

with an example where you need to create your own promise.

Answer 7 JavaScript- >

Promises are objects in JavaScript that represent the eventual completion (or failure) of an asynchronous operation and allow handling the result asynchronously. They provide a cleaner and more structured way to work with asynchronous operations compared to traditional callback-based approaches.

A Promise has three different states:

1. Pending: The initial state when a Promise is created. It means the asynchronous operation is still in progress, and the Promise is neither fulfilled nor rejected.
2. Fulfilled: The state when a Promise is successfully resolved with a value. It means the asynchronous operation completed successfully, and the Promise holds the resolved value. Once fulfilled, a Promise is considered settled, and its value is available.
3. Rejected: The state when a Promise encounters an error or failure. It means the asynchronous operation failed to complete, and the Promise holds the reason for rejection. Like fulfillment, rejection also settles the Promise.

Q8 What is ‘this’ keyword in JavaScript? explain with an example & create?

Answer 8 JavaScript- >

In JavaScript, the **this** keyword refers to the context in which a function is executed. It represents the object on which a method is being called or the object that is currently being constructed. The value of **this** can change based on how a function is invoked.

Let's explore the different scenarios where **this** can be used:

1. Global Context: When **this** is used in the global scope (outside of any function or object), it refers to the global object. In a browser environment, the global object is **window**.

console.log(this); // Output: Window

1. Function Context: When **this** is used within a regular function (not an arrow function), its value depends on how the function is called. It can vary based on different factors:

* As a Method: When a function is called as a method of an object, **this** refers to the object itself.

const person = {

name: "John",

sayHello: function() {

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

}

};

person.sayHello(); // Output: Hello, John!

1. Constructor Context: When a function is used as a constructor function with the **new** keyword, **this** refers to the newly created object.

function Person(name) {

this.name = name;

}

const john = new Person("John");

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

1. Event Handlers: When a function is used as an event handler, **this** usually refers to the DOM element that triggered the event.

<button id="myButton">Click me</button>

const button = document.querySelector("#myButton");

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

console.log(this); // Output: <button id="myButton">Click me</button>

});

Q9 Explain event loop Call Stack Callback queue and Micro Task queue in Your Words?

Answer 9 JavaScript- >

Sure! Let me explain the event loop, call stack, callback queue, and microtask queue in simpler terms:

1. Event Loop: The event loop is a mechanism in JavaScript that allows the runtime environment (such as a web browser) to handle asynchronous operations while also maintaining a smooth user experience. It continuously monitors the call stack and queues to execute tasks in a timely manner.
2. Call Stack: The call stack is a data structure that keeps track of function calls in JavaScript. When a function is called, it is added to the top of the call stack. The currently executing function is always at the top. Each function is executed in a last-in-first-out (LIFO) order. When a function finishes executing, it is removed from the stack.
3. Callback Queue: The callback queue is a queue that holds callback functions (also known as event handlers) waiting to be executed. Callback functions are added to the queue when asynchronous events, such as user interactions or timer events, occur. These callbacks are processed by the event loop when the call stack is empty.
4. Microtask Queue: The microtask queue (also known as the job queue or microtask checkpoint) is a queue that holds microtasks. Microtasks are functions that need to be executed after the current task but before rendering. They have higher priority than regular callback functions in the callback queue. Common examples of microtasks include promises and mutation observers.

The order of execution is as follows:

* When the call stack is empty, the event loop checks the microtask queue.
* If there are any microtasks in the queue, they are executed one by one until the queue is empty.
* After processing all microtasks, the event loop checks the callback queue.
* If there are any callback functions in the queue, the event loop takes one and pushes it onto the call stack for execution.
* The function is executed, and if it contains asynchronous operations, they are registered to run later.
* The process continues, with the event loop monitoring the call stack, microtask queue, and callback queue for new tasks.

This cycle of checking the call stack, microtask queue, and callback queue continues indefinitely as long as there are tasks to be processed.

Overall, the event loop, call stack, callback queue, and microtask queue work together to manage the execution of JavaScript code, handle asynchronous operations, and maintain the responsiveness of the application.

Q10 Explain Debouncing and Create a project where you are using Debouncing?

Answer 10 JavaScript- >

Debouncing is a technique used in web development to optimize performance by reducing the frequency of repetitive or redundant function calls, particularly in response to user input events like scrolling, resizing, or typing. It ensures that a function is only executed after a certain period of inactivity has occurred.

When an event occurs, such as scrolling or typing, the associated event handler function is typically triggered immediately. However, in certain cases, this can lead to performance issues, especially if the event handler function involves resource-intensive operations or if the event fires rapidly in succession. Debouncing helps mitigate these issues by postponing the execution of the function until a specific delay has passed without any new events occurring.

The basic idea behind debouncing is as follows:

Set up a timer variable or flag.

When the event occurs, clear the existing timer (if any) to start fresh.

Start a new timer to delay the execution of the function.

If the event continues to fire within the delay period, repeat steps 2 and 3.

Once the delay period elapses without any new events, execute the function.

<https://github.com/jagtapnimisha2/Debuging>

Q11 Explain Closures and Use cases of Closures?

Answer 11 JavaScript- >

A closure in JavaScript is a combination of a function and the lexical environment within which that function was declared. It allows a function to retain access to variables and parameters from its parent scope even after the parent function has finished executing. In other words, a closure "closes over" the variables it references, preserving their values and allowing them to be accessed later.

Here are some use cases of closures in JavaScript:

1. Encapsulation and Data Privacy: Closures can be used to create private variables and encapsulate functionality. By defining variables within a function scope and returning an inner function that has access to those variables, you can control the visibility and accessibility of the variables.
2. Function Factories: Closures can be used to create functions with pre-configured settings or behavior. By accepting parameters in the outer function and returning an inner function that uses those parameters, you can create specialized functions.
3. Asynchronous Operations and Callbacks: Closures can be useful in handling asynchronous operations and managing callbacks. By capturing variables within a closure, you can maintain their state and use them in the callback function when the asynchronous operation completes.

Q12 Create a Blog web app using JavaScript **(10 Marks)**

- Fetch data from https://jsonplaceholder.typicode.com/posts and show it to ui

- User can also add new blog

- Add Delete functionality also

Answer 12 JavaScript- ><https://github.com/jagtapnimisha2/blog_web_appp>

← React Questions →

Q1What’s React and What are the advantages of it?

Answer 1 React- >

React is a popular JavaScript library for building user interfaces. It was developed by Facebook and has gained widespread adoption in the web development community. React follows a component-based architecture, where UI elements are broken down into reusable components. It allows developers to create dynamic and interactive web applications efficiently.

Advantages of React include:

1. Component-Based Architecture: React promotes a modular approach to building UIs by breaking them down into reusable components. Components encapsulate their own logic, styles, and state, making them easier to manage, test, and maintain. This reusability helps in writing clean and organized code.
2. Virtual DOM: React uses a virtual representation of the DOM (Document Object Model) called the Virtual DOM. It keeps a lightweight copy of the actual DOM in memory and efficiently updates and renders only the necessary components when the underlying data changes. This approach improves performance and enhances the user experience.
3. Declarative Syntax: React uses a declarative syntax, allowing developers to describe how the UI should look based on the application's state. Instead of manually manipulating the DOM, you define the desired UI state, and React takes care of updating the actual DOM to reflect that state. This makes the code more predictable, easier to read, and less prone to bugs.
4. One-Way Data Flow: React follows a unidirectional data flow, where data flows from parent components to child components. This makes it easier to understand how data changes and enables better control over data flow and state management. It helps in building scalable and maintainable applications.
5. Rich Ecosystem and Community Support: React has a vast ecosystem with a wide range of libraries, tools, and extensions that complement its functionality. It has a strong and active community that contributes to its development, provides support, and shares knowledge. This ecosystem enables developers to leverage existing solutions and accelerates development.

Q2 What's Virtual Dom in React & What are the advantages of it?

Answer 2 React- >

The Virtual DOM (Document Object Model) is a key concept in React. It is a lightweight copy or representation of the actual DOM, stored in memory. React uses the Virtual DOM to efficiently update and render components based on changes in the application's state.

Advantages of the Virtual DOM in React:

1. Performance Optimization: The Virtual DOM helps improve performance by reducing the number of direct manipulations to the actual DOM. Updating the entire DOM can be costly in terms of performance, but React's Virtual DOM calculates and applies only the necessary updates, resulting in optimized rendering and better user experience.
2. Efficient Reconciliation: React's diffing algorithm efficiently reconciles the previous and updated Virtual DOM representations, identifying and updating only the components that have changed. This selective rendering approach minimizes the number of DOM operations required and reduces unnecessary re-renders, making React faster and more efficient.
3. Abstraction of Complex Operations: The Virtual DOM provides an abstraction layer that simplifies complex operations involved in updating the UI. Developers can focus on specifying the desired state of the UI, and React takes care of efficiently updating the actual DOM to reflect that state. This abstraction reduces the cognitive load and makes development more manageable.

Q3 Explain LifeCycle of React Components?

Answer 3 React- >

In React, components have a lifecycle that consists of various phases and methods. These methods allow you to perform specific actions at different stages of a component's existence. The React component lifecycle can be divided into three main phases: Mounting, Updating, and Unmounting.

1. Mounting Phase:
   * **constructor()**: The constructor is called when a component is being initialized. It is used to set the initial state and bind event handlers.
   * **static getDerivedStateFromProps()**: This method is called before rendering and allows you to update the state based on changes in props.
   * **render()**: The render method is responsible for returning the JSX representation of the component.
   * **componentDidMount()**: This method is called after the component is rendered for the first time. It is commonly used to initiate API calls, add event listeners, or perform other side effects.
2. Updating Phase:
   * **static getDerivedStateFromProps()**: This method is also called during the updating phase, similar to the mounting phase, and allows you to update the state based on changes in props.
   * **shouldComponentUpdate()**: This method determines whether the component should re-render or not. By default, it returns true, but you can implement custom logic to optimize performance by preventing unnecessary re-renders.
   * **render()**: The render method is called again to update the component's UI.
   * **componentDidUpdate()**: This method is called after the component is updated and re-rendered. It is commonly used to perform side effects, such as making additional API calls or updating the DOM based on the new state or props.
3. Unmounting Phase:
   * **componentWillUnmount()**: This method is called when the component is about to be removed from the DOM. It allows you to perform cleanup tasks, such as removing event listeners or canceling pending requests.

Q4 Whats the difference between between Functional Components and Class

Components?

Answer 4 React- >

Functional Components and Class Components are two ways of creating components in React, and they have some differences in terms of syntax and capabilities.

Functional Components:

* Syntax: Functional components are defined as JavaScript functions that return JSX (a syntax extension for JavaScript) to describe the component's UI.
* State Management: Functional components do not have their own state. Previously, with React versions before 16.8, functional components were stateless and relied on external state management solutions (like Redux) or passed props from parent components.
* Lifecycle Methods: In earlier versions of React, functional components did not have access to lifecycle methods. However, with the introduction of React Hooks in React 16.8, functional components gained the ability to use lifecycle-related hooks such as **useEffect()** to handle side effects and perform actions at specific times.
* Simplicity: Functional components are generally simpler and more concise, as they are just JavaScript functions. They focus on rendering UI based on the provided props.

Q5 What are the hooks in React & Can we use Hooks in Class Components?

Answer 5 React- >

Hooks are a feature introduced in React 16.8 to allow developers to use state and other React features in functional components. Hooks provide a way to use state and lifecycle methods without the need for writing class components. They enhance the capabilities of functional components and promote code reuse and better component organization.

Some commonly used hooks in React include:

1. **useState**: This hook allows functional components to have their own state. It returns a state variable and a function to update that state, enabling components to manage and update their state.
2. **useEffect**: This hook enables components to handle side effects and perform actions based on certain conditions or events. It replaces lifecycle methods like **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**.
3. **useContext**: This hook allows functional components to consume a Context in React. It provides access to the value stored in the nearest matching Context Provider.
4. **useReducer**: This hook is an alternative to **useState** and provides more control over complex state management. It allows components to define a reducer function similar to the ones used in Redux.
5. **useCallback** and **useMemo**: These hooks optimize performance by memoizing functions and values, respectively. They prevent unnecessary re-renders when dependencies don't change.

Q6 What are the LifeCycle method and the advantages of it?

Answer 6 React- >

In React, lifecycle methods are special methods that are called at different stages of a component's lifecycle. These methods allow you to perform specific actions or implement certain behaviors during various phases of a component's existence.

Here are some commonly used lifecycle methods in React class components:

1. **constructor()**: The constructor is called when a component is being initialized. It is used to set the initial state and bind event handlers. It is also where you would typically initialize any external dependencies or perform other setup tasks.
2. **componentDidMount()**: This method is called after the component has been rendered to the DOM for the first time. It is commonly used to initiate API calls, fetch data from a server, or set up subscriptions to external data sources. It is a good place to perform any initializations that require access to the actual DOM.
3. **componentDidUpdate(prevProps, prevState)**: This method is called after the component has been updated, i.e., when new props or state are received. It is often used to perform side effects based on changes in props or state, such as updating the DOM, making additional API calls, or updating the component's internal state.
4. **componentWillUnmount()**: This method is called when the component is about to be removed from the DOM. It allows you to perform cleanup tasks, such as removing event listeners, canceling pending requests, or unsubscribing from external data sources. It helps prevent memory leaks and ensures that resources are properly released.

Q7 What’s useState Hook & Advantages of it?

Answer 7 React- >

The **useState** hook is a built-in hook in React that allows functional components to have their own local state. It provides a way to declare and manage state within functional components, which were traditionally stateless before the introduction of hooks.

Advantages of **useState** hook:

1. Simplicity: The **useState** hook simplifies state management in functional components by allowing them to have their own state. It eliminates the need to convert functional components into class components just for the sake of state management.
2. Cleaner Code: With **useState**, you can declare and update state within the component's body, making the code more readable and concise. The state logic is encapsulated within the component, improving code organization.
3. Multiple State Variables: The **useState** hook supports multiple instances of state variables within a single component. You can use **useState** multiple times to declare and manage different state variables independently.
4. Functional Updates: The **setState** function returned by **useState** allows you to update state based on the previous state. By passing a function to **setState**, you can ensure that state updates are based on the latest state value, preventing potential race conditions or stale data.
5. Performance Optimization: React uses a shallow comparison to determine if a component needs to re-render. By breaking down state into smaller pieces with multiple **useState** calls, you can optimize rendering by updating only the necessary parts of the component when a specific state changes.
6. Hooks Compatibility: The **useState** hook is fully compatible with other React hooks, such as **useEffect**, **useContext**, and custom hooks. This allows for easy integration and composition of different hooks to handle complex component behaviors.

Q8 Explain useEffect & Advantages of it?

Answer 8 React- >

The **useEffect** hook is a built-in hook in React that allows functional components to perform side effects. Side effects include actions such as fetching data, subscribing to events, or manipulating the DOM. The **useEffect** hook provides a way to introduce side effects into functional components without the need for class components and lifecycle methods.

Advantages of **useEffect** hook:

1. Side Effect Handling: The **useEffect** hook provides a centralized and intuitive way to handle side effects within functional components. It separates the side effect logic from the component's rendering logic, promoting code clarity and maintainability.
2. Replacing Lifecycle Methods: The **useEffect** hook can replace various lifecycle methods, such as **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**. By specifying dependencies and cleanup logic, you can achieve similar functionality without the need for class components.
3. Asynchronous Operations: The **useEffect** hook supports asynchronous operations inside the side effect function. You can use **async/await** or return a Promise to handle asynchronous tasks, such as fetching data from an API.
4. Dependency Tracking: By specifying dependencies in the second argument, you can control when the side effect function should be executed. This enables fine-grained control over when the side effect should occur, optimizing performance and preventing unnecessary re-executions.

Q10 Explain useReducer and Its advantages

Answer 10 React- >

The **useReducer** hook is a built-in hook in React that provides an alternative approach to managing complex state and state transitions within functional components. It is typically used when the state logic becomes more involved and requires more control than what **useState** alone can offer.

Advantages of **useReducer** hook:

1. Centralized State Management: **useReducer** allows you to centralize the management of complex state within a single reducer function. This promotes a more organized and predictable approach to state updates, especially when the state logic involves multiple variables or complex interactions.
2. Readability and Maintainability: By using a reducer function, you separate the state transition logic from the component's rendering logic. This results in more readable and maintainable code, as the state transitions are concentrated in one place rather than scattered throughout the component.
3. Complex State Transitions: **useReducer** is well-suited for managing state transitions that involve complex business logic or interdependent state variables. The reducer function provides a structured way to handle these transitions, making the code more manageable and reducing the potential for bugs.
4. Time Travel and Undo/Redo: With **useReducer**, you can easily implement features like time travel (allowing you to jump back and forth between different states) and undo/redo functionality. By keeping track of the state history within the reducer, you can restore previous states or navigate through state changes.
5. Performance Optimization: When combined with memoization techniques like **React.memo** or **useMemo**, **useReducer** can optimize rendering performance by preventing unnecessary re-renders. By returning the same state object reference when the state doesn't change, you can avoid unnecessary re-renders in child components.
6. Integration with Context: **useReducer** integrates well with the **useContext** hook and can be used to manage the state of a context provider. It allows for more complex state management and provides a structured way to handle state updates within the context.

Q13 Build Calculator Using React Only ?

Answer 13 React- >

<https://github.com/jagtapnimisha2/calculator1>

Q12 Build A simple counter app using React?

Answer 12 React- > <https://github.com/jagtapnimisha2/simple_counter_app>

Q14 Build a Tic Tac Toe Game using Class Component of React?

Answer 14 React- ><https://github.com/jagtapnimisha2/tic_tac_toe>

Q15 Explain Prop Drilling & How can we avoid it?

Answer 15 React- >

Prop drilling can become a significant pain point as your React application grows in complexity. By using the Context API and Hooks, you can avoid prop drilling and create a more maintainable and flexible application. With these tools, you can efficiently manage state and ensure a cleaner, more readable codebase

Q16 Create a task manager where user can create tasks and see his task **(10 Marks)**

- Redirect him to task dashboard section after login

- Use https://reqres.in/ api to authenticate user and redirect him to task manager

dashboard where he can see his task and create

Answer 16 React- ><https://github.com/jagtapnimisha2/task_manager>