UI Developer Interview Questions

HTML Questions

1. What is Element, Tags & Attribute in HTML?

CSS Questions

2. What is the Box Model in CSS?
3. Components of the Box Model?
4. What is an Inline-Block Element?
5. What is Positioning?
6. Difference Between CSS and SCSS?
7. What are Mixin & Placeholder in SCSS?

JavaScript Questions

8. What is Synchronous and Asynchronous in JS?
9. What is DOM & How to Manipulate the DOM?
10. What is a Promise in JS?
11. What is a Callback Function in JavaScript?
12. How to Get Rid of Callback Hell?
13. What is Function Definition & Function Expression?

Answer:

1. What is Element, Tags & Attribute in HTML?

Element: A combination of a start tag, its attributes, content, and an optional end tag. It represents a component of the webpage.

Tag: Keywords enclosed in angle brackets that define an HTML element. Example: `<p>` and `</p>`.

Attribute: Provides additional information about an element, specified in the opening tag. Example: `href`, `src`, `class`.

Example:

1. Visit Example

- **Element:** `<a href="https://example.com" target="_blank">Visit Example</a>`
- **Tags:** `<a>` and `</a>`
- **Attributes:** `href="https://example.com"`, `target="_blank"`

2. What is the Box Model in CSS?

The Box Model in CSS is a fundamental concept that describes how every HTML element is represented as a rectangular box.

3. Components of the Box Model?

The components of the CSS Box Model are:

1. Content:
   The area where the text, images, or other content is displayed.

2. Padding:
   The space between the content and the border. It increases the size of the box without affecting the border.

3. Border:
   A line that surrounds the padding and content. Its thickness and style can be customized (e.g., solid, dashed, etc.).

4. Margin:
   The space outside the border, used to separate the element from other elements. Margins do not have a background and are transparent.

4. What is an Inline-Block Element?

An inline-block element is an HTML element that combines features of both inline and block elements.

Example:

```html
<style>
  .inline-block {
      display: inline-block;
      width: 100px;
      height: 50px;
      background-color: lightblue;
      margin: 5px;
  }
</style>

<div class="inline-block">Box 1</div>
<div class="inline-block">Box 2</div>
<div class="inline-block">Box 3</div>
```

5. What is Positioning?

Positioning in CSS refers to the way elements are placed in the document relative to other elements, the document itself, or the browser viewport. The position property controls this behavior.

6. Difference Between CSS and SCSS?

Here's a concise explanation of the difference between CSS and SCSS:

CSS (Cascading Style Sheets)
- Definition: A stylesheet language used to style HTML elements.
- Syntax: Basic and straightforward, with limited features for managing complex styling.
- Features:
  - Rules are written plainly.
  - No support for variables, nesting, or functions.
  - Requires repetitive code for common patterns.

SCSS (Sassy Cascading Style Sheets)
- Definition: A preprocessor scripting language that extends CSS by adding more powerful features.
- Syntax: Superset of CSS, so valid CSS is valid SCSS.
- Features:
  - Variables: Store reusable values (e.g., colors, font-sizes).
  - Nesting: Write selectors inside each other to mimic the HTML structure.

- Mixins: Reusable blocks of code with optional parameters.
- Inheritance: Share properties between selectors using `@extend`.
- Functions & Logic: Perform calculations or apply conditional styling.

Advantages of SCSS:
1. Reusability: Mixins, functions, and variables save time and reduce code duplication.
2. Readability: Nesting makes code more structured and easier to follow.
3. Maintainability: Centralized variables and reusable styles simplify updates.

SCSS is compiled into CSS, so browsers can interpret it.

7. What are Mixin and Placeholder in SCSS?

Mixin

- A reusable block of CSS code that can accept parameters and be included wherever needed using the `@include` directive.
- Useful for reducing code repetition and managing dynamic styles.

Example:

```
@mixin button($color) {
  background-color: $color;
  padding: 10px;
  border-radius: 5px;
}

.primary-btn {
  @include button(blue);
}

.secondary-btn {
  @include button(green);
}
```

-

Placeholder

- Defines a style block that can be extended using the `@extend` directive.
- Placeholders don't output CSS on their own unless extended.

Example:

```
%shared-style {
  font-size: 16px;
```

```
  font-weight: bold;
}

.header {
  @extend %shared-style;
}

.footer {
  @extend %shared-style;
}
```

-

Difference:

- Mixins allow parameterization and can be included multiple times with dynamic values.
- Placeholders cannot take parameters and are useful for sharing static styles among elements.

8. What is Synchronous and Asynchronous in JavaScript?

Synchronous:

- Operations are executed one after another in a sequential order.
- Each task waits for the previous one to complete before starting.
- If one task takes a long time, it blocks the execution of subsequent tasks.

Example:

```
console.log("Task 1");
console.log("Task 2");
console.log("Task 3");
// Output: Task 1, Task 2, Task 3 (in order)
```

Asynchronous:

- Operations are executed independently of the main program flow.
- Tasks can run in the background, and their results are handled once they're completed.
- It allows other tasks to execute without waiting for a long-running task to finish.

Example:

```
console.log("Task 1");
```

```
setTimeout(() => console.log("Task 2"), 2000); // Executes after 2 seconds
console.log("Task 3");
// Output: Task 1, Task 3, Task 2
```

Key Difference:

- **Synchronous** tasks are blocking, whereas **asynchronous** tasks are non-blocking.

9. What is the DOM, and How Can You Manipulate It?

What is the DOM?
The **Document Object Model (DOM)** is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree of objects, where each element, attribute, and text is represented as a node. This allows scripts like JavaScript to dynamically interact with and manipulate the content, structure, and styling of a web page.

How to Manipulate the DOM?
JavaScript provides methods to manipulate the DOM. Here are common ways to interact with the DOM:

1. **Accessing Elements**:
    - `document.getElementById('id')`
    - `document.querySelector('.class')`
    - `document.getElementsByClassName('class')`
    - `document.getElementsByTagName('tag')`

Example:

```
const header = document.getElementById('header');
console.log(header.textContent);
```

2. **Changing Content**:
    - `element.textContent = 'New Content';`
    - `element.innerHTML = '<strong>Updated HTML</strong>';`

Example:

```
const para = document.querySelector('p');
para.textContent = 'This is updated text!';
```

3. **Modifying Attributes**:

- `element.setAttribute('attribute', 'value');`
- `element.getAttribute('attribute');`
- `element.removeAttribute('attribute');`

Example:

```
const img = document.querySelector('img');
img.setAttribute('src', 'new-image.jpg');
```

4. **Styling Elements**:

   - `element.style.property = 'value';`

Example:

```
const div = document.querySelector('div');
div.style.backgroundColor = 'blue';
```

5. **Adding/Removing Classes**:

   - `element.classList.add('class-name');`
   - `element.classList.remove('class-name');`
   - `element.classList.toggle('class-name');`

Example:

```
const box = document.querySelector('.box');
box.classList.add('highlight');
```

6. **Creating and Appending Elements**:

   - `document.createElement('tag')`
   - `parentElement.appendChild(newElement)`
   - `parentElement.insertBefore(newElement, referenceElement)`

Example:

```
const newItem = document.createElement('li');
newItem.textContent = 'New List Item';
document.querySelector('ul').appendChild(newItem);
```

7. **Event Handling**:

   - Add event listeners to trigger actions.

- `element.addEventListener('event', callback);`

Example:

```
const button = document.querySelector('button');
button.addEventListener('click', () => alert('Button Clicked!'));
```

By using these methods, you can dynamically change the structure, content, and style of a webpage in response to user actions or other events.

10. What is a Promise in JavaScript?

A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It is a way to handle asynchronous tasks more efficiently, avoiding callback-based approaches.

States of a Promise

1. **Pending**: The initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation was successful, and the promise has a resolved value.
3. **Rejected**: The operation failed, and the promise has a reason (error).

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;

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

Using a Promise

You can handle the result of a promise using `.then()`, `.catch()`, and `.finally()`.

Example:

```
myPromise
  .then((result) => {
    console.log(result); // Logs: Operation successful!
  })
```

```
  .catch((error) => {
    console.log(error); // Logs: Operation failed!
  })
  .finally(() => {
    console.log("Promise completed.");
  });
```

Key Features of Promises

Chaining: Promises can be chained for sequential execution.

```
 myPromise
  .then((result) => {
    console.log(result);
    return "Another task";
  })
  .then((result) => {
    console.log(result);
  });
```

1.

Error Handling: Errors can be caught using `.catch()`.

```
 myPromise
  .then(() => {
    throw new Error("Something went wrong!");
  })
  .catch((error) => {
    console.error(error.message);
  });
```

2.

Combining Promises: Use `Promise.all()` or `Promise.race()` for multiple promises.

```
 Promise.all([promise1, promise2]).then((results) => console.log(results));
```

3.

Why Use Promises?

- Provides cleaner and more readable code for handling asynchronous tasks.
- Helps avoid **callback hell** by allowing chaining and better error management.

11. What is a Callback Function in JavaScript?

A **callback function** is a function that is passed into another function as an argument and is executed (called back) after the completion of a certain task or event. Callback functions are commonly used for handling asynchronous operations, such as handling API responses, file reading, or event handling.

How Callback Functions Work

When a function is executed and completes its task, the callback function is called to process the result, perform additional actions, or continue the flow.

Example of a Callback Function:

```
function fetchData(url, callback) {

  // Simulate an asynchronous task, like fetching data

  setTimeout(() => {

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

    callback(data); // Calling the callback function with the data

  }, 2000);

}



function handleData(data) {

  console.log("Data received:", data);

}



// Calling fetchData and passing handleData as a callback

fetchData("https://api.example.com", handleData);
```

In the example above, the `handleData` function is the callback passed into `fetchData`. Once the data is "fetched" after 2 seconds, the callback is executed, and the data is logged.

Common Use Cases of Callback Functions:

1. **Asynchronous tasks**: Callbacks are often used in functions like `setTimeout`, `setInterval`, or in API requests like `fetch()`.

Event handling: Callbacks are used to handle events like button clicks, form submissions, etc.

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

  alert("Button clicked!");

});
```

2.

Callback Hell

Callback functions can lead to nested, difficult-to-read code, especially when multiple callbacks are chained together (often referred to as "callback hell"). This is one reason why **Promises** and **async/await** syntax were introduced to handle asynchronous code in a cleaner, more readable manner.

12. How Do You Get Rid of Callback Hell?

Callback hell refers to the situation where callbacks are nested within other callbacks, making the code difficult to read, maintain, and debug. It usually happens when dealing with multiple asynchronous operations that depend on each other. To avoid callback hell and improve code readability, there are several approaches you can take:

1. Use Promises

Promises allow you to chain asynchronous operations and avoid deep nesting of callbacks. Each `.then()` block handles a subsequent operation, making the flow more linear and easier to follow.

Example:

```
fetchData(url)
  .then(response => processData(response))
  .then(data => displayData(data))
  .catch(error => console.error('Error:', error));
```

This makes the code cleaner, with each asynchronous operation handled in its own `.then()` block.

2. Use `async/await`

`async/await` is a modern syntax that makes asynchronous code look and behave like synchronous code, reducing the need for callbacks altogether. It allows for a cleaner structure without deeply nested callbacks or promise chaining.

Example:

```
async function fetchDataAndDisplay() {
  try {
    const response = await fetchData(url);
    const data = await processData(response);
    displayData(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

3. Modularize the Code

Break down large, complex functions into smaller, more manageable functions. Each function can handle a specific task, improving readability and reusability.

Example:

```
function fetchData(url, callback) {
  // Fetch data from API
  setTimeout(() => {
    callback({ success: true, data: "Some data" });
  }, 1000);
}

function processData(response, callback) {
  // Process data
  callback(response.data.toUpperCase());
}

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

```
fetchData("url", (response) => {
  processData(response, (processedData) => {
    displayData(processedData);
  });
});
```

You can break this down into individual, easier-to-read functions, helping prevent nested callbacks.

4. Use Libraries

Some libraries (like **Async.js**) help manage async operations and avoid callback hell by providing methods for parallel execution, sequencing, and error handling.

Example with Async.js:

```
async.series([
  function(callback) {
    fetchData(url, callback);
  },
  function(callback) {
    processData(response, callback);
  }
], function(error, results) {
  if (error) console.error(error);
  else displayData(results);
});
```

5. Use Event Emitters

In situations where you have multiple asynchronous operations that need to be executed independently, consider using **event emitters** (in Node.js or other environments that support them) to handle events instead of callbacks.

Summary of Techniques:

- Use **Promises** for cleaner code with chaining.
- Use `async/await` for asynchronous code that looks synchronous.
- **Modularize** your code to break it down into smaller, manageable functions.
- Use **libraries** like Async.js to simplify asynchronous control flow.

- Consider **event-driven programming** when handling independent asynchronous tasks.

By adopting these techniques, you can avoid callback hell and make your asynchronous code much more maintainable and easier to understand.

13. What is the Difference Between Function Definition and Function Expression in JavaScript?

Function Definition (Function Declaration):
A function definition, or function declaration, is a way to define a named function using the `function` keyword. It is hoisted, meaning you can call the function before its definition appears in the code.

Syntax:

```
function functionName(parameters) {
  // Function body
}
```

Example:

```
sayHello();

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

Key Features of Function Definitions:

1. Must have a name.
2. Can be called before its definition due to hoisting.

Function Expression:
A function expression involves assigning a function (either named or anonymous) to a variable. It is not hoisted, so it cannot be called before its definition.

Syntax:

```
const variableName = function(parameters) {
  // Function body
};
```

Example:

```
const sayHello = function() {
  console.log("Hello, world!");
};

sayHello();
```

Key Features of Function Expressions:

1. The function can be anonymous or have a name.
2. Cannot be called before its assignment as it is not hoisted.

Key Differences:

1. **Hoisting**: Function definitions are hoisted, meaning you can use them before they are defined in the code. Function expressions are not hoisted, so you must define them before calling them.
2. **Naming**: Function definitions always have a name, whereas function expressions can be anonymous.
3. **Usage**: Function expressions are often used when passing functions as arguments or when defining functions dynamically, while function definitions are commonly used for reusable, named functions in the code.