**1. What is HTML? Differences between HTML & HTML5? Advantages of HTML5?**

* **HTML (HyperText Markup Language)** is the standard language for creating web pages.
* **Differences**: HTML5 supports multimedia elements (audio, video), improved parsing, new semantic elements, and APIs (e.g., local storage).
* **Advantages of HTML5**: Better multimedia support, semantic structure, improved web applications, offline capabilities, and reduced need for third-party plugins.

**2. What is the difference between HTML and XHTML?**

* **HTML** is less strict with syntax, allowing for some errors.
* **XHTML** is a stricter and cleaner version of HTML, adhering to XML syntax rules. All tags must be closed, and elements must be properly nested.

**3. What is the role of DOCTYPE in HTML?**

* The <!DOCTYPE> declaration specifies the HTML version being used and helps browsers render the page correctly. It must be the first line in the HTML document.

**4. Difference between Head & Body in HTML? Where to place JS link reference?**

* **Head**: Contains meta-information, links to stylesheets, and scripts (not rendered on the page).
* **Body**: Contains the content that is displayed to the user.
* Place JavaScript references at the end of the body or in the head (with defer attribute) to optimize loading.

**5. What is Title Tag in HTML?**

* The <title> tag defines the title of the document, displayed in the browser's title bar or tab. It should be placed within the <head> section.

**6. What are different HTML heading tags?**

* There are six heading tags: <h1>, <h2>, <h3>, <h4>, <h5>, <h6>. <h1> is the highest level, and <h6> is the lowest.

**7. What are Meta Tags?**

* Meta tags provide metadata about the HTML document, such as description, keywords, author, and character set. They are placed in the <head> section.

**8. What are HTML Elements? What is the difference between Element & Tag?**

* **HTML Element**: A complete set of tags and content (e.g., <p>Hello</p>).
* **Tag**: The markup that defines an element (e.g., <p> and </p>).

**9. What are the roles and uses of the <div> element in HTML?**

* The <div> element is a block-level container used for grouping content and applying CSS styles. It's often used for layout purposes.

**10. What is the difference between <div> and <span> element?**

* <div> is a block-level element, while <span> is an inline element. Use <div> for larger sections and <span> for styling small portions of text.

**11. What is the use of <label> Tag?**

* The <label> tag associates a text label with a form element, enhancing accessibility and usability.

**12. Role of various elements:**

* **<a>**: Creates hyperlinks.
* **<br>**: Inserts a line break.
* **<hr>**: Inserts a horizontal rule (divider).
* **<em>**: Emphasizes text (usually italic).
* **<img>**: Embeds images.
* **<input>**: Creates interactive controls for forms.
* **<button>**: Creates clickable buttons.

**13. Role of semantic elements:**

* **<header>**: Represents introductory content.
* **<main>**: Represents the main content of the document.
* **<section>**: Defines sections of content.
* **<footer>**: Represents footer content.
* **<address>**: Provides contact information.

**14. What are Root, Parent, Child, & Nested elements?**

* **Root**: The top element of the document (e.g., <html>).
* **Parent**: An element that contains other elements.
* **Child**: An element contained within another element.
* **Nested**: Elements placed inside other elements.

**15. What are Empty Elements?**

* Empty elements (or self-closing tags) have no content and do not require a closing tag (e.g., <img>, <br>).

**16. What are Semantic Elements in HTML?**

* Semantic elements clearly describe their meaning in a human- and machine-readable way (e.g., <article>, <footer>).

**17. Can HTML tags be written in Uppercase?**

* Yes, HTML tags can be written in uppercase, but lowercase is the standard convention. XHTML requires lowercase.

**18. 3 differences between Block-Level & Inline Elements:**

1. **Display**: Block-level elements start on a new line, while inline elements do not.
2. **Width**: Block-level elements take the full width available; inline elements only take as much width as necessary.
3. **Margin & Padding**: Block elements respect vertical margins and padding, whereas inline elements respect only horizontal margins and padding.

**19. List of Block-Level & Inline Elements in HTML:**

* **Block-Level**: <div>, <p>, <h1> to <h6>, <section>, <header>, <footer>, <article>, <table>.
* **Inline**: <span>, <a>, <img>, <strong>, <em>, <label>, <br>.

**20. What are HTML Attributes? Types of HTML attributes?**

* HTML attributes provide additional information about elements. Types include:
  + **Global attributes** (e.g., class, id, style).
  + **Specific attributes** for elements (e.g., src for <img>, href for <a>).

**21. What are the id, style, & class attributes of an element? When to use what?**

* **id**: Unique identifier for an element (one per page).
* **class**: Used to apply styles to multiple elements.
* **style**: Inline CSS for specific styling on an element. Use it sparingly for maintainability.

**22. What will happen if two elements have the same ids?**

* The id attribute must be unique within a page. If two elements share the same id, it can lead to unpredictable behavior, particularly in JavaScript.

**23. What are Data Attributes in HTML?**

* Data attributes (e.g., data-\*) allow you to store custom data on elements, accessible via JavaScript. Example: <div data-user-id="123">.

**24. What are the 5 Types of Links in HTML?**

1. **Internal Links**: Link to another section on the same page.
2. **External Links**: Link to a different website.
3. **Anchor Links**: Link to a specific location within a page.
4. **Mailto Links**: Opens the user's email client.
5. **Tel Links**: Links to a phone number.

**25. What is the difference between Absolute & Relative URLs?**

* **Absolute URL**: Full web address (e.g., https://example.com/page).
* **Relative URL**: Address relative to the current page (e.g., /page or page.html).

**26. What is the purpose of the <nav> element in HTML?**

* The <nav> element is used to define a section of navigation links.

**27. How do you add an external stylesheet in your HTML?**

* Use the <link> tag in the <head> section:

<link rel="stylesheet" href="styles.css">

**28. How do you open a link in a new tab?**

* Use the target attribute with value \_blank in the <a> tag:

<a href="https://example.com" target="\_blank">Open Link</a>

**29. How do you create an Email Link?**

* Use the mailto: protocol in the <a> tag:

<a href="mailto:example@example.com">Email Us</a>

**30. What are the different Types of Lists in HTML?**

* **Ordered List**: <ol> for numbered items.
* **Unordered List**: <ul> for bullet points.
* **Definition List**: <dl> for terms and definitions.

**31. What is a Nested List in HTML?**

* A list that contains other lists as items. For example, an <ul> inside an <li> of another <ul>.

**32. What are <table>, <tr>, <th>, <td> elements?**

* **<table>**: Defines a table.
* **<tr>**: Defines a row in the table.
* **<th>**: Defines a header cell in a table (bold and centered).
* **<td>**: Defines a standard cell in a table.

**33. What is the colspan attribute in HTML?**

* The colspan attribute allows a cell to span multiple columns in a table. For example: <td colspan="2">.

**34. What is the best way to add a border to a table, column, and cell?**

* Use CSS for styling:

table {

border: 1px solid black;

}

th, td {

**35. What is CSS? What are the 3 ways to Implement CSS in HTML?**

**CSS (Cascading Style Sheets)** is a stylesheet language used to describe the presentation of a document written in HTML or XML. It controls the layout, colors, fonts, and overall visual appearance of web pages.

**Three ways to implement CSS in HTML:**

1. **Inline CSS**: Styles are applied directly within an HTML element using the style attribute.

<h1 style="color: blue;">Hello World</h1>

1. **Internal Stylesheet**: CSS is included within a <style> tag in the <head> section of an HTML document.

<head>

<style>

body { background-color: lightblue; }

</style>

</head>

1. **External Stylesheet**: CSS is written in a separate file (e.g., styles.css) and linked to the HTML document using a <link> tag in the <head>.

<head>

<link rel="stylesheet" href="styles.css">

</head>

**36. What is Inline Style in CSS? When to use it in real applications?**

**Inline Style** refers to applying CSS directly within an HTML element via the style attribute. This method is useful for applying unique styles to a single element without affecting others.

**When to use:**

* For quick testing or debugging.
* For one-off styles that are unlikely to be reused.
* In situations where external or internal stylesheets are impractical (e.g., within dynamically generated HTML).

**37. What is Internal Stylesheet in CSS? When to use it in real applications?**

**Internal Stylesheet** is defined within a <style> tag inside the <head> section of an HTML document. It applies styles to that specific document only.

**When to use:**

* When styles are specific to a single page.
* For small projects or prototypes where external files might be overkill.
* When you need to override external styles without creating a separate stylesheet.

**38. What is External Stylesheet in CSS? When to use it in real applications?**

**External Stylesheet** is a separate CSS file linked to an HTML document. This method allows you to keep CSS separate from HTML, promoting cleaner code and easier maintenance.

**When to use:**

* In larger projects with multiple pages, where consistent styling is needed.
* When styles are reused across different pages.
* For better organization, making it easier to manage and update styles without editing individual HTML files.

**39. What are CSS Selectors and what are their types?**

CSS Selectors are patterns used to select the elements you want to style in an HTML document. They can be categorized into several types:

* **Universal Selector (\*)**: Selects all elements.

\* {

margin: 0;

padding: 0;

}

* **Type Selector**: Selects all elements of a specific type (e.g., all <p> elements).

p {

color: blue;

}

* **Class Selector (.)**: Selects all elements with a specific class.

.highlight {

background-color: yellow;

}

* **ID Selector (#)**: Selects a single element with a specific ID.

#header {

font-size: 24px;

}

* **Attribute Selector**: Selects elements with a specific attribute.

input[type="text"] {

border: 1px solid black;

}

* **Pseudo-class Selector**: Styles elements based on their state (e.g., :hover).

css

Copy code

a:hover {

text-decoration: underline;

}

* **Pseudo-element Selector**: Styles a specific part of an element (e.g., ::first-line).

p::first-line {

font-weight: bold;

}

**40. How do you Include CSS in a webpage or HTML?**

CSS can be included in a webpage in three primary ways:

* **Inline CSS**: Directly within an HTML tag using the style attribute.

<h1 style="color: red;">Hello, World!</h1>

* **Internal CSS**: Within a <style> tag inside the <head> section of your HTML document.

<head>

<style>

body {

background-color: lightblue;

}

</style>

</head>

* **External CSS**: By linking to an external stylesheet using a <link> tag in the <head>.

<head>

<link rel="stylesheet" href="styles.css">

</head>

Using external stylesheets is generally preferred for maintainability and reusability.

**41. Explain Box Model in CSS.**

The CSS Box Model describes how elements are structured and interact on a webpage. Each element is represented as a rectangular box comprising several layers:

* **Content**: The innermost area where text and images appear.
* **Padding**: Space between the content and the border, used for spacing within the element.
* **Border**: Surrounds the padding (if any) and the content, which can be styled (e.g., solid, dashed).
* **Margin**: The outermost layer, providing space between the element and others around it.

Understanding the box model is essential for layout design.

**42. Explain Padding, Margin, and Border.**

* **Padding**: The space between the content and the border of an element. It increases the area around the content.

.box {

padding: 20px; /\* Adds 20px space inside the border \*/

}

* **Border**: The line that surrounds the padding and content. You can control its style, width, and color.

.box {

border: 2px solid black; /\* Creates a solid black border \*/

}

* **Margin**: The space outside the border, providing distance between the element and other elements.

.box {

margin: 10px; /\* Adds 10px space outside the border \*/

}

**43. What are the different data types available in JavaScript? Provide examples of each.**

JavaScript has several data types:

* **Number**: Numeric values.

let age = 30; // integer

let price = 19.99; // float

* **String**: Text values, enclosed in quotes.

let name = "Alice"; // single or double quotes

* **Boolean**: True or false values.

let isActive = true;

* **Undefined**: A variable declared but not assigned a value.

let result; // undefined

* **Null**: Represents an intentional absence of any object value.

let value = null; // explicitly set to null

* **Symbol**: A unique and immutable primitive value, often used as object property keys.

const uniqueId = Symbol('id');

* **Object**: A collection of properties, which can hold various data types.

let person = { name: "Bob", age: 25 }; // object

**44. Explain the difference between var, let, and const. When would you use each of these declarations?**

* **var**: The oldest way to declare variables, function-scoped or globally scoped. It can be re-declared and updated.

var x = 10;

var x = 20; // valid

* **let**: Introduced in ES6, block-scoped. You can update but not re-declare in the same scope.

let y = 10;

// let y = 20; // invalid, will throw an error

* **const**: Also introduced in ES6, block-scoped. Used for variables that should not be re-assigned. Must be initialized at declaration.

const z = 30;

// z = 40; // invalid, will throw an error

**45. What is hoisting in JavaScript? How does it affect variables declared with var, let, and const?**

Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their containing scope during compilation.

* **var**: Variables declared with var are hoisted and initialized to undefined.

console.log(a); // undefined

var a = 5;

* **let and const**: These variables are also hoisted but are not initialized, leading to a "temporal dead zone" until the declaration is encountered.

// console.log(b); // ReferenceError: Cannot access 'b' before initialization

let b = 10;

**46. What is a callback function in JavaScript? Give an example of how and when to use a callback function.**

A callback function is a function that is passed as an argument to another function and is executed after some operation is completed. This is commonly used in asynchronous programming.

Example:

function fetchData(callback) {

setTimeout(() => {

const data = "Data received!";

callback(data);

}, 2000);

}

fetchData(function(response) {

console.log(response); // Outputs: Data received!

});

In this example, fetchData simulates an asynchronous operation and calls the provided callback with the result after 2 seconds.

**47. What are arrow functions? How do they differ from traditional functions in JavaScript?**

Arrow functions provide a concise way to write functions in JavaScript. They do not have their own this, which makes them ideal for use in methods that require a specific context.

Example:

const add = (a, b) => a + b; // Arrow function

console.log(add(5, 3)); // Outputs: 8

**Differences**:

* **Syntax**: Arrow functions are more concise.
* **this Binding**: They inherit this from their surrounding context.
* **No arguments object**: Arrow functions do not have their own arguments object.

**48. Explain the concept of 'callback hell' in JavaScript. Why does it occur, and what problems does it cause?**

Callback hell refers to a situation in asynchronous programming where multiple nested callbacks create complex and hard-to-read code. It often occurs when performing multiple asynchronous operations that depend on each other.

Example:

fetchData((data) => {

processData(data, (result) => {

saveData(result, (status) => {

console.log(status);

});

});

});

**Problems**:

* Decreased readability and maintainability.
* Increased complexity makes debugging difficult.
* Error handling can become cumbersome.

**49. What are some common solutions to avoid callback hell in JavaScript? Provide code examples for one solution.**

Solutions to avoid callback hell include:

* **Using Promises**: Promises allow for cleaner chaining of asynchronous operations.
* **Using async/await**: This provides a more readable way to handle asynchronous code.

Example using async/await:

async function fetchData() {

const data = await getData();

const result = await processData(data);

const status = await saveData(result);

console.log(status);

}

This flattens the structure of the code, making it easier to read and maintain.

**50. Describe the scope of variables in JavaScript. How does the scope differ between variables declared with var, let, and const?**

**Scope** refers to the accessibility or visibility of variables in certain parts of your code. In JavaScript, there are three main types of scope:

1. **Global Scope**: Variables declared outside any function or block are globally scoped and accessible from anywhere in the code.
2. **Function Scope**: Variables declared within a function using var are function-scoped, meaning they are only accessible within that function.
3. **Block Scope**: Variables declared with let or const are block-scoped, which means they are only accessible within the nearest enclosing {} block (like loops or conditionals).

**Differences between var, let, and const:**

* **var**:
  + Function-scoped.
  + Can be redeclared and updated within its scope.
  + Hoisted (initialized to undefined).

function testVar() {

if (true) {

var x = 10; // function-scoped

}

console.log(x); // Outputs: 10

}

testVar();

* **let**:
  + Block-scoped.
  + Cannot be redeclared in the same block but can be updated.

function testLet() {

if (true) {

let y = 20; // block-scoped

}

// console.log(y); // ReferenceError: y is not defined

}

testLet();

* **const**:
  + Block-scoped, like let.
  + Cannot be redeclared or updated once assigned.

function testConst() {

if (true) {

const z = 30; // block-scoped

// z = 40; // TypeError: Assignment to constant variable.

}

// console.log(z); // ReferenceError: z is not defined

}

testConst();

**51. How does JavaScript handle asynchronous operations? Briefly explain the role of the event loop in this process.**

JavaScript handles asynchronous operations using callbacks, Promises, and async/await, allowing it to execute long-running tasks (like fetching data) without blocking the main execution thread.

**The Event Loop** plays a crucial role in managing asynchronous operations. Here's how it works:

1. **Call Stack**: Where function calls are executed.
2. **Web APIs**: Functions that can handle asynchronous tasks (like setTimeout, fetch).
3. **Callback Queue**: A queue for functions that are ready to be executed after the main thread is free.

When an asynchronous task completes, its callback is pushed to the callback queue. The event loop continuously checks the call stack. If it's empty, it takes the first callback from the queue and pushes it onto the stack for execution.

This process allows JavaScript to perform non-blocking operations, keeping the application responsive.

**52. What is the difference between synchronous and asynchronous code in JavaScript? Provide examples of each.**

* **Synchronous Code**: Executes in sequence, blocking further execution until the current operation is complete. This can lead to delays if a task takes a long time.

**Example of Synchronous Code:**

console.log("Start");

console.log("Doing something...");

// Simulating a blocking operation

for (let i = 0; i < 1e9; i++) {} // Blocking loop

console.log("Done");

Output:

sql

Copy code

Start

Doing something...

Done

* **Asynchronous Code**: Executes independently of the main execution flow, allowing other operations to continue while waiting for a task to complete.

**Example of Asynchronous Code:**

console.log("Start");

setTimeout(() => {

console.log("Doing something after 2 seconds...");

}, 2000); // Non-blocking

console.log("Done");

Output:

sql

Copy code

Start

Done

Doing something after 2 seconds...

In this example, the setTimeout function allows the program to continue executing while it waits, demonstrating the non-blocking nature of asynchronous code.