1. Why do we use DOCTYPE in HTML?

The DOCTYPE tells the browser which HTML standard to use and ensures the page is rendered in standards mode instead of quirks mode.
In modern HTML, we use '<!DOCTYPE html>' to indicate HTML5. It is a declaration, not an HTML tag.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Example</title>
  </head>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

2. What is the use of the <head> tag?

The <head> element contains metadata about the document that is not directly displayed on the page. It typically includes:
• <title> – tab title and used by search engines.
• <meta> tags – charset, viewport, SEO data.
• Links to external CSS and fonts – <link rel="stylesheet" ...>.
• Scripts that should load before the body – usually with defer.

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="My portfolio">
  <title>Portfolio</title>
  <link rel="stylesheet" href="styles.css">
  <script src="app.js" defer></script>
</head>
```

3. Where is it better to place a script tag: in <head> or before </body>?

If we are not using defer or async, it is usually better to place the <script> tag just before the closing </body>. This ensures that the HTML and DOM are already parsed before the script runs, which is important when the script accesses DOM elements.

```
<body>
  <!-- page content -->
  <script src="app.js"></script>
</body>
```

4. If the script must be in <head> and accesses DOM elements, how do you handle it?

Use the defer attribute. It tells the browser to download the script while parsing HTML but execute it only after the DOM has been fully parsed, so DOM elements are available when the script runs.

```
<head>
  <script src="app.js" defer></script>
</head>
<body>
  <button id="btn">Click</button>
</body>

// app.js
document.getElementById("btn").addEventListener("click", () => {
  console.log("Button clicked");
});
```

5. Difference between script at bottom of body vs defer?

Script at bottom (no defer):
• HTML is parsed.
• When parser reaches script, it stops, downloads, and executes script.
• DOM is ready at that time.

Script in head with defer:
• HTML parses and script downloads in parallel.
• Script executes after HTML parsing is complete, before DOMContentLoaded.

defer often has better performance characteristics.

6. Is defer generally a better way? Why?

Yes. defer allows parallel download of scripts while HTML is parsed. Then scripts execute after DOM is ready, which makes behavior predictable and improves perceived performance.

7. How is async different from defer?

Both async and defer download scripts in parallel with HTML parsing, but execution differs:
• defer: scripts execute after HTML parsing finishes, in document order.
• async: script executes as soon as it is downloaded, which can pause HTML parsing. Order between multiple async scripts is not guaranteed.
So async is best for independent scripts (e.g., analytics) that do not depend on DOM or other scripts.

8. If a script with async is in <head>, what happens while parsing HTML?

The browser starts downloading the async script and parsing HTML at the same time. When the script finishes downloading, HTML parsing is paused, the script is executed, and then HTML parsing resumes. At that moment only part of the DOM may exist, so async scripts should not rely on DOM elements being available.

9. Can we create custom element names like <ainash>?

Yes. The browser does not throw an error for unknown tags; they are treated as unknown HTML elements. In standard HTML they behave like inline-level elements unless styled otherwise. For official Web Components, the custom element name must contain a dash (for example: <my-card>).

```
<ainash>This is a custom element.</ainash>
```

10. What is the default display value of a custom unknown element?

By default, unknown/custom elements behave similarly to inline elements in most browsers, so they act like display: inline unless CSS overrides it.

11. Difference between inline and block elements?

Inline elements:
• Do not start on a new line.
• Occupy only as much width as their content.
• For non-replaced inline elements you typically cannot set explicit width/height.
• Left/right margin works; top/bottom margin is usually ignored for layout.

Block elements:
• Start on a new line.

• Take full available width by default.
• Can set width, height, margin, and padding in all directions.

12. For inline elements, what about margin and padding?

For non-replaced inline elements such as <span>:
• Left and right margins affect horizontal space.
• Top and bottom margins do not typically affect line layout.
• Padding on all sides is allowed. Left/right padding widens the inline box, and top/bottom padding increases the visual height but does not behave like block margin.

13. What is the default display of images?

<img> elements are inline elements by default, but they are replaced inline elements, so they still accept width and height.

```
img {
  display: inline;  // default
}
```

14. How can images have width/height if they are inline?

Images (and elements like <video> and <iframe>) are replaced inline elements. Their content comes from external resources. Browsers allow explicit width and height for replaced elements, even though they are inline by default.

```
<img src="photo.jpg" alt="photo" width="200" height="150">
```

15. Two categories of inline elements?

The two broad categories are:
• Replaced inline elements: <img>, <video>, <iframe> – allow width/height and come from external sources.
• Non-replaced inline elements: <span>, <a>, <em> – content is text/HTML and do not behave like blocks.

16. Explain the Box Model.

The CSS box model describes how the browser calculates the space an element takes. Each element has:
• Content box – actual content (text, children).
• Padding – space between content and border.
• Border – line surrounding padding and content.
• Margin – space outside the border.

With box-sizing: content-box (default) width/height apply to content only; border-box width/height include content, padding, and border.

```
.box {
  box-sizing: border-box;
  width: 200px;
  padding: 10px;
  border: 2px solid black;
  margin: 20px;
}
```

17. Are all elements rectangular because of the Box Model?

For layout calculations, the browser treats every element as a rectangular box (or multiple rectangular fragments for inline content). Even if we visually create circles or other shapes using border-radius or clipping, the underlying box used for layout and hit-testing is still rectangular.

18. What are the different data types in JavaScript?

Primitives:
• number
• string
• boolean
• null
• undefined
• symbol
• bigint

Non-primitives:
• objects (including arrays, functions, dates, etc.).

19. Have you used classes in JavaScript?

Yes. ES6 classes are syntactic sugar over constructor functions and prototypes. They are used to create objects with shared methods, though in React we mostly use functional components now.

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
}
```

20. If we create a class (User) and check typeof, what is it?

typeof User will be 'function', because classes are built on top of constructor functions.

```
class User {}
console.log(typeof User); // "function"
```

21. What is typeof for a function?

typeof any function expression or declaration is 'function'.

```
function f() {}
console.log(typeof f); // "function"
```

22. What is typeof for an array?

typeof for an array is 'object'. To check arrays specifically, use Array.isArray(value).

```
const arr = [1, 2, 3];
console.log(typeof arr);         // "object"
console.log(Array.isArray(arr)); // true
```

23. What are Promises?

A Promise is a special object that represents the eventual completion or failure of an asynchronous operation. It has three states: pending, fulfilled (resolved), and rejected. We attach handlers using .then, .catch, and .finally, or we use async/await syntax.

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => resolve("done"), 1000);
});

p.then(value => console.log(value))
 .catch(err => console.error(err));
```

24. Create a Promise that resolves when a button is clicked.

We can create a Promise and call resolve inside a click handler. Until the button is clicked the Promise stays pending.

```
const btn = document.getElementById("btn");

const clickPromise = new Promise((resolve, reject) => {
  btn.addEventListener("click", () => {
    resolve("Button clicked");
  });
});

clickPromise.then(msg => console.log(msg));
```

25. Create a Promise without using new Promise.

Any async function returns a Promise automatically. Its return value becomes the resolved value; thrown errors cause rejection.

```
async function getData() {
  return "Hello"; // resolves with "Hello"
}

getData().then(console.log);
```

26. Does await block the main thread? How does it resume?

await does not block the main thread. It only pauses the current async function. When the engine hits await somePromise:
• The async function is suspended and control returns to the event loop.
• Other code can run.
• When the Promise settles, the continuation of the async function is queued as a microtask and resumes from the line after the await.

27. In modules with imports/exports and console logs, what is the order of output?

In ES modules, imported modules are loaded and evaluated before the main module body runs. So any top-level console.log in imported files runs before console.log lines after the import statement in the main file.

```
// util.js
console.log("From util");
export const x = 10;

// main.js
import { x } from "./util.js";
console.log("From main", x);
// Output:
// From util
// From main 10
```

28. If we declare var anything = 10 in a module file, what scope is it in?

Inside an ES module, top-level declarations, including var, are scoped to the module, not to the global object (window). They cannot be accessed globally or from other files unless exported. Modules run in strict mode by default.

```
// myModule.js
var anything = 10;
export { anything };
```

29. Machine coding: click to create circles and check intersection.

Requirements:
• On clicking anywhere, create a circle centered at the click position with random radius between 10 and 100px.
• Circle has border only, no fill.
• Maintain a maximum of 2 circles. On the 3rd click, remove the old two and start again.
• When two circles exist, check intersection using distance between centers and sum of radii: if distance < r1 + r2 they intersect.

```
const circles = [];

document.addEventListener("click", (e) => {
  const radius = Math.floor(Math.random() * 91) + 10; // 10-100

  const circle = document.createElement("div");
  circle.className = "circle";
  circle.style.width = circle.style.height = radius * 2 + "px";
  circle.style.position = "fixed";
  circle.style.left = e.clientX - radius + "px";
  circle.style.top = e.clientY - radius + "px";
  circle.style.borderRadius = "50%";
  circle.style.border = "2px solid black";
  circle.style.background = "transparent";

  document.body.appendChild(circle);

  circles.push({ el: circle, x: e.clientX, y: e.clientY, r: radius });

  if (circles.length > 2) {
    circles[0].el.remove();
    circles[1].el.remove();
    circles.splice(0, 2);
  }

  if (circles.length === 2) {
    const [c1, c2] = circles;
    const dx = c2.x - c1.x;
    const dy = c2.y - c1.y;
    const dist = Math.hypot(dx, dy);

    if (dist < c1.r + c2.r) {
      console.log("intersecting");
    } else {
      console.log("not intersecting");
    }
  }
});
```