1. What is JSX?

JSX stands for **JavaScript XML**. It allows developers to write HTML-like syntax directly within JavaScript code, making it easier and more intuitive to create React elements. JSX provides a more readable and concise way to define UI components compared to using React.createElement(), which can become verbose and hard to manage in complex UIs. Although it looks like HTML, **JSX is not HTML** — it's syntactic sugar that gets transpiled into React.createElement() by transpilers like Babel. Ultimately, JSX returns JavaScript objects, just like React.createElement() does.

2. **Superpowers of JSX:**

- **Transpiles to** React.createElement()
  JSX is syntactic sugar for React.createElement(). Under the hood, JSX is converted into JavaScript objects with a special $$typeof property set to React.element, and the type corresponds to the HTML tag or React component being rendered.
- **Simplified Syntax**
  JSX provides an HTML-like syntax that is easier and more intuitive to write compared to raw JavaScript function calls.
- **Improved Readability and Maintainability**
  The structure and formatting of JSX closely resemble actual HTML, making it easier for developers to understand, debug, and maintain the UI code.
- **Component Composition**
  JSX supports composing UI by combining smaller components into larger ones, encouraging modular and reusable code.
- **Supports Nesting**
  Components and elements can be easily nested inside one another, which is essential for building hierarchical UI structures.
- **Scales Well for Complex UIs**
  JSX's expressive nature allows developers to build and manage complex user interfaces efficiently.

3. **What is the role of the type attribute in the <script> tag? What options can I use there?**
   The type attribute in the <script> tag specifies the MIME type (media type) of the script, telling the browser how to interpret the code inside the tag (or the file referenced by the src attribute).

   **Common values for the type attribute:**

- **text/javascript** *(default)*
  This is the default value and can be omitted. It indicates that the script is standard JavaScript.

- **module**
  This specifies that the script is a JavaScript **module**, allowing you to use import and export statements. Modules are automatically deferred and scoped.
  <script type="module" src="app.js"></script>

- **application/json**
  Used when embedding JSON data within a <script> tag. The contents won't be executed as code — it's usually accessed by JavaScript for configuration or templating purposes.
  <script type="application/json" id="config-data">
  { "theme": "dark", "lang": "en" }
  </script>

- **Custom types**
  You can use custom type values for things like templating engines (e.g., Handlebars, JSX, or GraphQL). The browser will ignore the content, and JavaScript can read and process it manually.
  <script type="text/x-handlebars-template" id="template"></script>

- **Summary:**
  The type attribute helps the browser know how to handle the content of the <script> tag. It’s essential when using **modules**, embedding **JSON**, or working with **template languages** or **custom data**.

4. **What is a MIME type?**

MIME stands for **Multipurpose Internet Mail Extensions**. Despite the name, MIME types are widely used on the web to indicate the **type of content** being handled, not just in email.

A **MIME type** (also called a **media type**) tells the browser or any client what kind of data is being sent or received, so it knows how to process or display it.

5. {TitleComponent} vs {< TitleComponent />} vs {< TitleComponent ></ TitleComponent >} in JSX

| Syntax | Meaning | Use Case |
|---|---|---|
| {TitleComponent} | Reference to the component (function/class) | Pass component as a prop, or dynamically render |
| {<TitleComponent />} | Render the component | Standard rendering of components |
| {<TitleComponent></TitleComponent>} | Same as above, but with long-form tag | Needed when passing children |

**Q: Why is defer used in the <script> tag in HTML?**

**A:**
The defer attribute is used in the <script> tag to ensure that the script is **downloaded in parallel** with the HTML parsing but **executed only after the entire HTML document has been parsed**. This improves page load performance and ensures that scripts don't block the rendering of the page.

**Key points about defer:**

- Scripts with defer are **executed in the order they appear** in the HTML.
- Only works with **external scripts** (<script src="...">, not inline).
- Scripts run **after the DOM is fully parsed**, but **before the DOMContentLoaded event**.
- It avoids the traditional issue where scripts block HTML parsing.

**Q: What does it mean when a script has type="module"? Is it automatically deferred and scoped?**

**A:**
Yes — when you use type="module" in a <script> tag, the script behaves differently:

✅ **Automatically Deferred:**

- **Modules are deferred by default**, even if you don't explicitly add the defer attribute.

- This means they do **not block HTML parsing** and will execute after the document has been parsed.

🔒 **Scoped Execution (Module Scope):**

- Code inside a module runs in **strict mode** by default.
- Variables declared in a module are **scoped to the module** — they are not added to the global window object.
- You can use **import and export** statements within modules, enabling modular, maintainable code.

**Q: Does traditional browser behavior delay script loading until HTML is parsed?**

**A:**
Not exactly — **by default**, when the browser encounters a <script> tag **without defer or async**, it:

1. **Immediately stops parsing the HTML**.
2. **Starts downloading the script**.
3. **Waits for it to finish downloading and executing**.
4. **Then resumes parsing the rest of the HTML**.

This blocking behavior **does slow down page rendering**, especially if the script is large or served from a slow network.

✅ **This is why defer or type="module" is preferred:**

- They **don't block** HTML parsing.
- Scripts are **downloaded in parallel** while the browser continues parsing HTML.
- Execution happens **only after the DOM is fully parsed**, improving load performance.

---

🚦 **In Summary:**

| Script Type | Blocks HTML Parsing | Download Timing | Execution Timing |
|---|---|---|---|
| <script src="..."> | ✅ Yes | Immediately | Immediately (when loaded) |
| <script src="..." defer> | ❌ No | In parallel | After HTML is parsed |

| Script Type | Blocks HTML Parsing | Download Timing | Execution Timing |
|---|---|---|---|
| <script type="module"> | ❌ No | In parallel (auto-deferred) | After HTML is parsed |

✅ **If you do NOT use defer, here's what happens:**

When the browser encounters a <script> tag **without** defer (or async) while parsing the HTML:

1. **It immediately pauses HTML parsing**.
2. **Starts downloading the script**.
3. **Waits for the script to fully download and execute**.
4. **Only then resumes parsing the rest of the HTML**.

❗ **This means:**

- The script **executes as soon as it is encountered** (after download), not after the whole HTML is parsed.
- This can **block the rendering** of the page and slow down load time.
- If the script tries to access HTML elements that haven't been parsed yet (like with document.getElementById), it can **fail**.