1. What is JSX?
   1. const element = <h1> Hello, World! </h1>

This tag is neither a string nor HTML. It’s called JSX, and it is a syntax extension of JavaScript. It is recommended to use this syntax with React to describe the UI. JSX produces React ‘elements.’ If HTML is copy pasted into react components it will give us an error because JSX is stricter and has a few more rules than HTML.

There are certain rules for JSX, and they are as follows:

* **Return a single root element**. When you want to return multiple elements, you wrap them with a single parent tag.
* A screenshot of a computer program

  Description automatically generated
* A screenshot of a computer program

  Description automatically generated
* JSX looks like HTML, but under the hood it is transformed into plain JavaScript objects. You can’t return objects from a function without wrapping them into an array. This explain why you also can’t return two JSX tags without wrapping them into another tag or a Fragment.
* **Close all the tags**: JSX requires tags to explicitly closed: self-closing tags like <img> must become <img />, and wrapping tags like <li> oranges must be written as <li>oranges</li>.
* **camelCase most of the things**: JSX turns into javascript and attributes in JSX becomes keys of JavaScript Objects. What it means that React converts this JSX into plain JavaScript using a function called React.createElement.

const element = <h1>Hello World!</h1>;

gets converted into:

const element = React.createElement(“h1”, null, “Hello, World!”);

When you add attributes to JSX tags these attributes are turned into a javascript object.

For example:

const button = <button className="btn" disabled={true} />;

becomes:

const button = React.createElement("button", { className: "btn", disabled: true });

Here, the className and disabled are keys in the JavaScript object passed as the second argument to React.createElement.

In our own components, we often want to read these attributes into variables. But JS has limitations on variable names. For example: their names can’t contain dashes or be reserved words like class.

That is why, in React, many HTML and SVG attributes are written in camelCase. For example, instead of stroke-width we use strokeWidth. Since class is a reserved word, in React you write className instead.

1. What are the superpowers of JSX?

Answer:

**1. HTML + JavaScript Together**

* **What It Means:**  
  JSX allows you to combine HTML-like syntax with JavaScript logic, making it easier to create UI components directly in your code.
* **Why It’s Powerful:**  
  You don’t need to switch between files or languages; everything you need to build a UI is in one place.
* **Example:**
* const user = { firstName: "John", lastName: "Doe" };
* const greeting = <h1>Hello, {user.firstName} {user.lastName}!</h1>;
  + Here, the {user.firstName} and {user.lastName} allow you to insert JavaScript variables directly into the HTML-like JSX.

**2. Dynamic Rendering**

* **What It Means:**  
  JSX allows you to use JavaScript logic to conditionally display or repeat elements based on your data or state.
* **Why It’s Powerful:**  
  Instead of manually updating the UI when data changes, JSX handles it dynamically based on logic.
* **Example:**
* const isLoggedIn = true;
* const button = (
* <div>
* {isLoggedIn ? <button>Logout</button> : <button>Login</button>}
* </div>
* );
  + If isLoggedIn is true, the "Logout" button is shown; otherwise, the "Login" button is shown.

**3. Readable and Intuitive**

* **What It Means:**  
  JSX looks similar to HTML, making it easy to understand and write, even for beginners.
* **Why It’s Powerful:**  
  It reduces the learning curve for developers transitioning from HTML to React.
* **Example:**
* const app = (
* <div>
* <h1>Welcome to My App</h1>
* <p>This is a paragraph explaining the app.</p>
* </div>
* );
  + This looks just like HTML but is embedded inside JavaScript, which React turns into UI elements.

**4. React Elements Made Easy**

* **What It Means:**  
  JSX simplifies the process of creating React elements by hiding the complex React.createElement syntax.
* **Why It’s Powerful:**  
  You can focus on the structure and design without worrying about the underlying code.
* **Example:**
* const heading = <h1>Hello!</h1>;
  + This is equivalent to writing:
  + const heading = React.createElement("h1", null, "Hello!");

**5. Rich Attribute Support**

* **What It Means:**  
  JSX allows you to use attributes like in HTML, but you can also pass JavaScript expressions or objects as values.
* **Why It’s Powerful:**  
  You can dynamically set attributes based on your data or logic.
* **Example:**
* const imgSrc = "logo.png";
* const altText = "Company Logo";
* const logo = <img src={imgSrc} alt={altText} width={100} height={100} />;
  + The src, alt, width, and height attributes can be set dynamically.

**6. Automatic Protection Against Injection Attacks**

* **What It Means:**  
  JSX automatically escapes any JavaScript code or values you insert, ensuring malicious code can’t harm your app.
* **Why It’s Powerful:**  
  It keeps your app secure without requiring extra effort from you.
* **Example:**
  + A screen shot of a computer

    Description automatically generated
  + JSX renders <script>alert('Hacked!')</script> as plain text, not as a script, preventing harm.

**7. Component Integration**

* **What It Means:**  
  JSX makes it simple to reuse and nest components inside other components, like building blocks for your app.
* **Why It’s Powerful:**  
  It encourages modular design, which keeps your code clean and manageable.
* **Example:**
* A screenshot of a computer

  Description automatically generated
  + The Header component is reused within the App.

**8. Full JavaScript Support**

* **What It Means:**  
  You can use JavaScript features like loops, functions, and conditionals directly in your JSX.
* **Why It’s Powerful:**  
  It gives you full control over how your UI is generated.
* **Example:**
* A screenshot of a computer program

  Description automatically generated
  + The map() function generates a list of <li> elements dynamically.

**9. Optimized Compilation**

* **What It Means:**  
  JSX gets compiled into optimized JavaScript, which React can process efficiently.
* **Why It’s Powerful:**  
  You get the benefit of writing readable code without sacrificing performance.
* **Example:**
* const element = <h1>Hello</h1>;
  + Compiled into: const element = React.createElement("h1", null, "Hello");

1. The type attribute in the <script> tag specifies the type of script being used. By default, if the type attribute is omitted, the browser assumes the script is JavaScript (text/javascript).

**Common Values for *type***:

1. **text/javascript**:
   * Specifies that the script is written in JavaScript.
   * This is the default value and doesn't need to be explicitly declared in modern browsers.
2. **module**:
   * Indicates that the script is an ES6 module.
   * Enables usage of import and export statements.
   * Example:

html

Copy code

<script type="module" src="module.js"></script>

1. **application/json**:
   * Used when the script content is JSON data.
   * Example:

html

Copy code

<script type="application/json">

{ "name": "example", "value": 42 }

</script>

1. **application/ld+json**:
   * Used for structured data in JSON-LD format, often for SEO purposes.
   * Example:

html

Copy code

<script type="application/ld+json">

{

"@context": "https://schema.org",

"@type": "Person",

"name": "John Doe"

}

</script>

1. **Other MIME types**:
   * You can use other MIME types for custom or non-standard scripts, but they won’t be executed by browsers directly. Instead, they can be processed by custom scripts or frameworks.

**Deprecated Values**

* **text/ecmascript**, **application/javascript**, and **application/ecmascript**: These were used in the past but are now rarely used since text/javascript is widely supported.

**Why Use the type Attribute?**

* To inform the browser about the type of content in the <script> tag.
* Essential when working with ES modules or non-JavaScript content like JSON or custom scripts.

1. {TitleComponent}: This refers to the reference to the component function/class itself. It does not render anything instead, it is the actual component definition.

Use case: Typically used when you want to pass the component as a prop, dynamically determine which component to render, or refer to the component itself.

For example:

function Wrapper({Component}) {

return <Component />;

}

<Wrapper Component={TitleComponent} />;

Here, TitleComponent is passed as a prop and rendered dynamically.

{<TitleComponent />}: This is a self-closing tag syntax used to render a functional or class based React component. It means the component is being instantiated and rendered immediately, and it does not have any children.

Use Case: Use this when the component does not need child elements or any closing tag.

Example:

function App() {

return <div>{<TitleComponent />}</div>;

}

Here, TitleComponent is rendered, and its output appears in the DOM.

Notes:

If the component accepts props, you can pass them here:

{<TitleComponent title = “Hello” />}

{<TitleComponent></TitleComponent>}:

This is the explicit opening and closing tag syntax used to render a component. IT allows you to nest children elements inside the component.

Use: use this when you need to pass children ( other elements or content) into the component.

Example:

function TitleComponent({ children }){

return <h1>{children}</h1>;

}

function App(){

return <div>{TitleComponent>Hello World</TitleComponent>}</div>;

}

Notes:

The children prop will contain everything inside the opening and closing tags of the component.

This syntax is more verbose than the self-closing syntax but necessary when children need to be passed.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated