1. What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript that looks like HTML but compiles to React's `createElement` calls.

Why JSX is useful:

- Lets you write UI structure directly in JS.
- Makes components more readable and expressive.
- Allows embedding JS expressions inside markup.
- Compiles to efficient React element creation.

Example:

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

Compiles to:

```
React.createElement("h1", null, "Hello, World!");
```

Short version:

JSX is a syntax that looks like HTML but compiles to React function calls, making UI building easier and more declarative.

2. Superpowers of JSX?

⭐ Superpowers of JSX

1. HTML-like syntax inside JavaScript

JSX lets you write UI structure and JavaScript logic in the same place.
It feels like HTML but compiles to JS, making components easy to read and maintain.

2. JavaScript expressions inside markup

You can embed any JS expression using `{}`:

<div>{user.name.toUpperCase()}</div>

This makes UI dynamic and expressive.

3. Better developer experience & readability

JSX is more intuitive than raw `createElement` calls.
Large component trees are easier to visualize, debug, and maintain.

4. Type safety & compile-time validation

With TypeScript + JSX, you get:

- prop type checking
- autocomplete
- compile-time errors
  This reduces runtime bugs significantly.

5. Prevents XSS by default

JSX escapes values automatically, so injecting user data is safe:

<div>{userInput}</div>

This protects your app from cross-site scripting unless you explicitly use `dangerouslySetInnerHTML`.

Short version:

JSX lets you write HTML-like syntax in JS, embed expressions, improves readability, offers compile-time safety, and automatically prevents XSS.

3. Role of type attribute in script tag? What options can I use there?

The `type` attribute in a `<script>` tag tells the browser **what kind of script** it is and **how to handle it**.

✅ Role of the `type` attribute

- It specifies the **MIME type** of the script.
- Helps the browser know whether to **execute**, **ignore**, or **treat it differently** (e.g., as a module).
- Modern browsers default to `text/javascript` if no type is provided.

✅ Common `type` options you can use

1. `type="text/javascript"` (default)

The standard JavaScript type.
Today, you don't usually need to write this explicitly.

2. `type="module"`

Tells the browser the script is an **ES module**.

- Supports `import` / `export`
- Loaded with `defer` behavior by default
- Executes in strict mode

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

3. `type="text/babel"` (used with Babel standalone)

Allows Babel in the browser to compile JSX/ES6 → JS.

4. `type="application/json"`

Used when embedding JSON data inside a script tag (not executed as JS).

5. `type="importmap"`

Defines module import paths for browser-native ES modules.

```
<script type="importmap">
{
  "imports": {
    "lodash": "/libs/lodash.js"
  }
}
</script>
```

6. Non-JS types for templating

These are not executed; the browser ignores them:

- `type="text/template"`
- `type="text/x-handlebars-template"`
- `type="text/html"`

Used by frameworks or template engines.

Short version:

The `type` attribute specifies the script format. Common values are `module`, `text/javascript`, `application/json`, and template types. The most important modern one is `type="module"` for ES modules.

4. {TitleComponent} vs {<TitleComponent/>} vs {<TitleComponent><TitleComponent/>} in JSX ?

✅ 1. `{TitleComponent}`

This is **a reference to the component function itself**, not an element.

Example:

console.log(TitleComponent);

Useful when:

- Passing the component as a prop
- Storing it in a variable
- Conditionally selecting a component

But **it does NOT render anything**.

✅ 2. `{<TitleComponent />}`

This is **incorrect JSX**.
You cannot wrap JSX elements inside `{ }` *unless* you're using them inside a JavaScript expression.
Correct usage is simply:

<TitleComponent />

So `{<TitleComponent />}` is invalid syntax.

✅ 3. `<TitleComponent><TitleComponent/>`

This is **valid only if the component accepts children**.

Example:

<TitleComponent>
  <SubtitleComponent />
</TitleComponent>

Here:

- The outer `<TitleComponent>` is a component instance.
- The inner `<TitleComponent />` (or any component) is passed as `props.children`.

This is used for layout-style or wrapper components:

<Card>
  <UserInfo />
</Card>

Short version for interviews:

- **`{TitleComponent}`** → reference to the component function (not rendered).
- **`<TitleComponent />`** → actual JSX element, renders the component.
- **`<TitleComponent><TitleComponent/></TitleComponent>`** → nesting; inner component becomes `children` of the outer one.
- **`{<TitleComponent/>}`** → invalid syntax.

5. React.createElement vs JSX?

Here’s a crisp, interview-ready comparison:

✅ React.createElement

- Low-level API for creating React elements.
- Takes **type, props, children** as arguments.
- Hard to read for complex UIs.

Example:

React.createElement("h1", { className: "title" }, "Hello");

Used internally by React.

✅ JSX

- A **syntactic sugar** on top of `React.createElement`.
- Looks like HTML, easier to read and write.
- Compiled by Babel into `React.createElement` calls.

Example:

<h1 className="title">Hello</h1>

Compiles to:

React.createElement("h1", { className: "title" }, "Hello");

Key Differences

1. Readability

JSX is far more expressive and maintains UI structure visually.

2. Developer Experience

JSX supports:

- inline JS expressions
- TypeScript types
- linting and editor tooling

3. Compilation

JSX must be transpiled (Babel/Vite), while `createElement` does not.

4. Low-level vs High-level

`createElement` is what React uses internally;
JSX is what developers actually use.

Short version (for interviews):

JSX is syntactic sugar for `React.createElement`. JSX is easier to read and write, while `createElement` is the low-level function React uses under the hood.

6. Behind the Scenes of JSX?

JSX compiles to React.createElement calls, which generate Virtual DOM objects. React uses these objects to efficiently update the real DOM through its diffing algorithm. JSX is just a developer-friendly syntax — the browser never sees it.

7. Babel & parcel role in JSX?

⭐ Role of Babel & Parcel in JSX

JSX is **not valid JavaScript**, so browsers cannot understand it.
That's where **Babel** and **Parcel** come in — they work together to transform JSX into browser-safe JavaScript.

✅ Role of Babel

Babel is the **transpiler** that converts JSX → JavaScript.

Babel does:

- Converts JSX into `React.createElement()` (or JSX runtime functions).
- Converts modern JS (ES6+) into older JS for browser compatibility.
- Applies plugins/presets like `@babel/preset-react`.

Example:

<h1>Hello</h1>

Babel converts into:

React.createElement("h1", null, "Hello");

So, **Babel's job = Syntax transformation.**

✅ Role of Parcel

Parcel is the **bundler** that handles:

- Reading JSX files
- Sending them through Babel automatically (built-in support)
- Bundling them into one optimized output file
- Running dev server + HMR
- Optimizing/minifying code for production

Parcel configures Babel **under the hood** so you don't have to manually set it up.

So, **Parcel's job = Processing, bundling, optimizing.**

Short Version:

Babel transpiles JSX into plain JavaScript. Parcel detects JSX files and passes them through Babel, then bundles and optimizes the final output. Babel handles the syntax; Parcel handles the build pipeline.

8. Components , Functional Components , Composing Components?

Here is a clear, concise, interview-ready explanation of all three:

⭐ 1. Components

Components are the **building blocks** of a React application.
They allow you to break the UI into small, reusable, isolated pieces.

Key points:

- A component is just a function or class that returns UI.
- Accepts **props** as inputs.
- Must be **pure** (same input → same output) in terms of rendering.

⭐ 2. Functional Components

Functional components are **JavaScript functions** that return JSX.

Example:

```
function Title() {
  return <h1>Hello</h1>;
}
```

Why they're preferred today:

- Simple and easy to read
- Support **Hooks** (state, effects, etc.)
- Lightweight compared to class components
- Encouraged by React since v16.8

Functional components = the modern standard.

⭐ 3. Composing Components

Composing components means **combining smaller components to build bigger ones**.

Example:

```
function Header() {
  return (
    <div>
      <Logo />
      <NavMenu />
    </div>
  );
}
```

This is React's biggest strength:

- Reusability
- Separation of concerns
- Easy maintenance
- Scalable UI structures

React apps are essentially trees of composed components.

Short Version:

- **Components:** The reusable building blocks of a React UI.
- **Functional Components:** Functions that return JSX and can use Hooks.
- **Composing Components:** Combining smaller components to form larger, more complex UIs.