In React, **components** are the building blocks of your application's user interface. They let you split the UI into independent, reusable pieces that can be managed and composed together.

There are two main types:

- **Functional Components**

These are JavaScript functions that return JSX (a syntax extension that looks like HTML). They're simpler and more common in modern React, especially with the introduction of Hooks.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- Class Components

These use ES6 classes and include more features like lifecycle methods and internal state (though Hooks now allow similar functionality in functional components).

```
class Welcome extends React.Component {
  render() {
      return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Components can also:

- Accept props (inputs) to make them dynamic.

- Maintain state to track internal data.

- Be nested inside other components to build complex UIs.

JSX (JavaScript XML) is a **syntax extension for JavaScript** that lets you write HTML-like code directly within your React components. It's not required to use React, but it makes your code more readable and expressive.

Here's what makes JSX special:

- **HTML + JavaScript**: You can embed JavaScript expressions inside curly braces {} within your HTML-like code.

  const name = "Saumyajit"; const greeting =

Hello, {name}!

;

- **Behind the scenes**: JSX gets compiled by tools like Babel into React.createElement() calls, which build the virtual DOM.

```
const greeting = React.createElement('h1', null, 'Hello, Saumyajit!');
```

In React, **state** refers to data that a component maintains and can change over time—triggering a re-render when it does. It's what makes components dynamic and interactive.

In Functional Components

You use the useState Hook:

```
import { useState } from 'react';

function Example() {
  const [count, setCount] = useState(0); // count is the state variable
  return (
    <div>
      <p>Clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click</button>
    </div>
  );
}
```

- useState(0) initializes the state with 0.
- setCount updates the state and causes the component to re-render.

In Class Components

State is managed with this.state and updated using this.setState():

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return (
      <div>
        <p>Clicked {this.state.count} times</p>
```

```
      <button onClick={() => this.setState({ count: this.state.count + 1 })}>
        Click
      </button>
    </div>
  );
 }
}
```

In React, **props** (short for "properties") are how components talk to each other. They let you pass data from a **parent component** to a **child component**, making your UI dynamic and reusable.

How Props Work

Props are passed like HTML attributes:

```
function Welcome(props) {
 return <h1>Hello, {props.name}!</h1>;
}


function App() {
 return <Welcome name="Saumyajit" />;
}
```

Here, App passes the name prop to Welcome, which then uses it in the JSX.

Key Features

- **Read-only**: Props can't be changed by the receiving component.
- **Reusable**: You can use the same component with different props.
- **Flexible**: Props can be strings, numbers, arrays, objects, or even functions.

Destructuring for Clean Code

Instead of props.name, you can destructure:

```
function Welcome({ name }) {
 return <h1>Hello, {name}!</h1>;
}
```

In a React **class component**, props are accessed using this.props. They work the same way as in functional components—allowing you to pass data from a parent to a child—but the syntax is a bit different.

Basic Example

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

function App() {
  return <Welcome name="Saumyajit" />;
}
```

Here, the App component passes the name prop to Welcome, and Welcome accesses it using this.props.name.

Destructuring Props

To clean up your code, you can destructure props inside the render() method:

```
class Welcome extends React.Component {
  render() {
    const { name } = this.props;
    return <h1>Hello, {name}!</h1>;
  }
}
```

Key Points

- Props are **read-only**—you can't modify them inside the child component.
- They help make components **reusable** and **customizable**.
- You can pass **any data type**: strings, numbers, arrays, objects, or even functions.

Conditional rendering in React means showing different UI elements based on certain conditions—just like using `if` statements in JavaScript, but inside your component's render logic.

Here are the most common ways to do it:

1. If/Else Statements

Useful for more complex logic:

```
function Greeting({ isLoggedIn }) {
```

```
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please sign in.</h1>;
}
```

2. Ternary Operator

Great for inline conditions:

```
function Greeting({ isLoggedIn }) {
  return <h1>{isLoggedIn ? 'Welcome back!' : 'Please sign in.'}</h1>;
}
```

3. Logical AND (`&&`)

Perfect when you want to render something only if a condition is true:

```
function Notification({ hasMessages }) {
  return (
    <div>
      {hasMessages && <p>You have new messages!</p>}
    </div>
  );
}
```

4. Switch Statements

Handy when you have multiple conditions:

```
function StatusMessage({ status }) {
  switch (status) {
    case 'loading':
      return <p>Loading...</p>;
    case 'success':
      return <p>Data loaded!</p>;
    case 'error':
      return <p>Error occurred.</p>;
    default:
      return null;
  }
}
```

5. Returning `null`

If you don’t want to render anything:

```
function HiddenComponent({ isVisible }) {
  if (!isVisible) return null;
  return <div>This is visible</div>;
}
```