React Components Explained

In React, a **component** is a self-contained, reusable piece of UI (User Interface). Think of them as custom, isolated building blocks for your application. They encapsulate their own logic, state, and rendering, making it easier to manage and update complex UIs.

Components allow you to:

- **Divide UI into independent, reusable pieces:** Instead of building a large, monolithic UI, you can break it down into smaller, manageable components (e.g., a Button component, a Header component, a UserProfile component).
- **Manage state:** Components can have their own internal data (state) that changes over time, allowing for dynamic and interactive UIs.
- **Receive data (props):** Components can receive data from their parent components through properties (props), enabling communication and data flow.
- **Improve maintainability and scalability:** By promoting reusability and isolation, components make your codebase easier to understand, maintain, and scale as your application grows.

Differences Between React Components and JavaScript Functions

While React components (especially functional components) often look like JavaScript functions, there are key distinctions:

Feature	JavaScript Function	React Component
Purpose	General-purpose code execution, calculations, data manipulation.	Specifically designed to return UI elements (JSX).
Return Value	Can return any data type (string, number, object, array, etc.).	Must return JSX (or `null` or a string/number) that describes what to render.
Props	Takes arguments.	Takes a single `props` object as an argument, which contains data from its parent.
State	No inherent "state" mechanism.	Can have internal "state" (using `useState` hook in functional components or `this.state` in class components) that causes re-renders.
Lifecycle	No inherent lifecycle. Executes when called.	Has a lifecycle (mounting, updating, unmounting) with hooks/methods to tap into.
Reusability	Code reusability through function calls.	UI reusability through component instances.
Syntax	Standard JavaScript function syntax.	Standard JavaScript function or class syntax, but specific to React's API (e.g., `useState`, `useEffect`, `render()`).

Export to Sheets

Types of React Components

Historically, React had two primary types of components:

1. **Class Components**
2. **Function Components**

With the introduction of Hooks in React 16.8, Function Components gained the ability to manage state and lifecycle, making them the preferred choice for new development due to their simplicity and readability. However, understanding Class Components is still valuable, especially when working with older codebases.

Class Component Explained

A **class component** is a JavaScript ES6 class that extends `React.Component` and has a `render()` method.

Key Characteristics:

- **ES6 Class:** It's defined using the `class` keyword.
- **Extends `React.Component`:** This inheritance gives it access to React's component features, including `state`, lifecycle methods, and the `render()` method.
- **`render()` method:** This is the only mandatory method in a class component. It must return JSX that describes the component's UI.
- **`this.state`:** Class components manage their internal state using `this.state` and update it with `this.setState()`.
- **Lifecycle methods:** They have specific methods (e.g., `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`) that get called at different stages of the component's lifecycle.
- **Props:** Props are accessed via `this.props`.

Example:

```
import React from 'react';


class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: 'Hello'
    };
  }


  render() {
    return (
```

```
    <div>
      <h1>{this.state.message}, {this.props.name}!</h1>
      <button onClick={() => this.setState({ message: 'Hi there' })}>Change Greeting</button>
    </div>
  );
 }
}

export default Greeting;
```

Function Component Explained

A **function component** (also known as a "stateless functional component" before Hooks, and now often just "functional component") is a plain JavaScript function that accepts `props` as an argument and returns JSX.

Key Characteristics:

- **Plain JavaScript Function:** Defined using `function` or arrow function syntax.
- **`props` as argument:** Receives props directly as an argument.
- **Returns JSX:** The function's return value must be JSX (or `null`, string, number).
- **Hooks (since React 16.8):** With Hooks (`useState`, `useEffect`, `useContext`, etc.), function components can now manage state, perform side effects, and access lifecycle features without needing to be a class.
- **Simpler and more concise:** Generally less boilerplate code compared to class components.
- **Preferred for new development:** Due to Hooks, functional components are the modern and recommended way to write React components.

Example:

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

function Greeting(props) { // or const Greeting = (props) => { ... }
  const [message, setMessage] = useState('Hello');

  return (
    <div>
      <h1>{message}, {props.name}!</h1>
      <button onClick={() => setMessage('Hi there')}>Change Greeting</button>
```

```
    </div>
  );
}


export default Greeting;
```

Component Constructor (in Class Components)

The `constructor()` is a special method in a JavaScript class. In a React **class component**, the constructor is the first method called when an instance of the component is created.

Purpose:

- **Initialize `state`:** This is the primary place to set up the initial `this.state` for the component.
- **Bind event handlers:** If you have custom methods that need to access `this` (like `this.setState`), you'll typically bind them in the constructor.
- **Call `super(props)`:** It is **essential** to call `super(props)` as the first statement in the constructor of a React component. This calls the constructor of the parent `React.Component` class, ensuring that `this.props` is correctly initialized before your component's constructor runs.

Example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props); // Must be called first!
    this.state = {
      count: 0,
      text: props.initialText // Initializing state using props
    };
    // Binding a method to 'this'
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }
```

```
  render() {
    // ...
  }
}
```

Note: With functional components and Hooks, the `constructor` and `this` binding are no longer necessary, as `useState` handles state initialization, and event handlers automatically retain the correct `this` context.

`render()` Function (in Class Components)

The `render()` method is the **only required method** in a React **class component**.

Purpose:

- **Describe UI:** Its primary job is to return the React elements (JSX) that describe what should be rendered to the DOM.
- **Pure function:** The `render()` method should ideally be a pure function. This means it should not modify component state, interact with the browser directly (e.g., manipulate the DOM), or perform any other side effects. It should only return JSX based on `this.props` and `this.state`.
- **Returns:** It must return one of the following:
    - **React elements:** Usually created via JSX.
    - **Arrays or Fragments:** To render multiple elements without adding an extra DOM node.
    - **Portals:** To render children into a different DOM subtree.
    - **Strings or numbers:** These are rendered as text nodes.
    - **Booleans or `null`:** These render nothing.

Example:

```
class MyOtherComponent extends React.Component {
  render() {
    // Accessing props and state
    const { name } = this.props;
    const { isLoggedIn } = this.state;

    return (
      <div>
        {isLoggedIn ? (
```

```
          <p>Welcome back, {name}!</p>
        ) : (
          <p>Please log in.</p>
        )}
      </div>
    );
  }
}
```