React.Component

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. `React.Component` is provided by `React`.

Overview

`React.Component` is an abstract base class, so it rarely makes sense to refer to `React.Component` directly. Instead, you will typically subclass it, and define at least a `render()` method.

Normally you would define a React component as a plain JavaScript class:

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

If you don't use ES6 yet, you may use the `create-react-class` module instead. Take a look at Using React without ES6to learn more.

The Component Lifecycle

Each component has several "lifecycle methods" that you can override to run code at particular times in the process. Methods prefixed with **`will`** are called right before something happens, and methods prefixed with **`did`** are called right after something happens.

Mounting

These methods are called when an instance of a component is being created and inserted into the DOM:

- `constructor()`
- `componentWillMount()`
- `render()`
- `componentDidMount()`

Updating

An update can be caused by changes to props or state. These methods are called when a component is being re-rendered:

- `componentWillReceiveProps()`
- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

Unmounting

This method is called when a component is being removed from the DOM:

- `componentWillUnmount()`

Error Handling

This method is called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- `componentDidCatch()`

Other APIs

Each component also provides some other APIs:

- `setState()`

- `forceUpdate()`

Class Properties

- `defaultProps`
- `displayName`

Instance Properties

- `props`
- `state`

Reference

`render()`

```
render()
```

The `render()` method is required.

When called, it should examine `this.props` and `this.state` and return one of the following types:

- **React elements.** Typically created via JSX. An element can either be a representation of a native DOM component (`<div />`), or a user-defined composite component (`<MyComponent />`).
- **String and numbers.** These are rendered as text nodes in the DOM.
- **Portals**. Created with `ReactDOM.createPortal`.
- `null`. Renders nothing.
- **Booleans**. Render nothing. (Mostly exists to support `return test && <Child />` pattern, where `test` is boolean.)

When returning `null` or `false`, `ReactDOM.findDOMNode(this)` will return `null`.

The `render()` function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser. If you need to interact with the browser, perform your work in `componentDidMount()` or the other lifecycle methods instead. Keeping `render()` pure makes components easier to think about.

Note

`render()` will not be invoked if `shouldComponentUpdate()` returns false.

Fragments

You can also return multiple items from `render()` using an array:

```
render() {
  return [
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

Note:

Don't forget to add keys to elements in a fragment to avoid the key warning.

`constructor()`

```
constructor(props)
```

The constructor for a React component is called before it is mounted. When implementing the constructor for a `React.Component` subclass, you should call `super(props)` before any other statement. Otherwise, `this.props` will be undefined in the constructor, which can lead to bugs.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use `componentDidMount()` instead.

The constructor is the right place to initialize state. To do so, just assign an object to `this.state`; don't try to call `setState()` from the constructor. The constructor is also often used to bind event handlers to the class instance.

If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

In rare cases, it's okay to initialize state based on props. This effectively "forks" the props and sets the state with the initial props. Here's an example of a valid `React.Component` subclass constructor:

```
constructor(props) {
  super(props);
  this.state = {
    color: props.initialColor
  };
}
```

Beware of this pattern, as state won't be up-to-date with any props update. Instead of syncing props to state, you often want to lift the state up instead.

If you "fork" props by using them for state, you might also want to implement `componentWillReceiveProps(nextProps)` to keep the state up-to-date with them. But lifting state up is often easier and less bug-prone.

`componentWillMount()`

```
componentWillMount()
```

`componentWillMount()` is invoked immediately before mounting occurs. It is called before `render()`, therefore calling `setState()` synchronously in this method will not trigger an extra rendering. Generally, we recommend using the `constructor()` instead.

Avoid introducing any side-effects or subscriptions in this method. For those use cases, use `componentDidMount()` instead.

This is the only lifecycle hook called on server rendering.

---

componentDidMount()

```
componentDidMount()
```

`componentDidMount()` is invoked immediately after a component is mounted. Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in `componentWillUnmount()`.

Calling `setState()` in this method will trigger an extra rendering, but it is guaranteed to flush during the same tick. This guarantees that even though the `render()` will be called twice in this case, the user won't see the intermediate state. Use this pattern with caution because it often causes performance issues. It can, however, be necessary for cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

---

componentWillReceiveProps()

```
componentWillReceiveProps(nextProps)
```

`componentWillReceiveProps()` is invoked before a mounted component receives new props. If you need to update the state in response to prop changes (for example, to reset it), you may compare `this.props` and `nextProps` and perform state transitions using `this.setState()` in this method.

Note that React may call this method even if the props have not changed, so make sure to compare the current and next values if you only want to handle changes. This may occur when the parent component causes your component to re-render.

React doesn't call `componentWillReceiveProps()` with initial props during mounting. It only calls this method if some of component's props may update.
Calling `this.setState()` generally doesn't trigger `componentWillReceiveProps()`.

`shouldComponentUpdate()`

```
shouldComponentUpdate(nextProps, nextState)
```

Use `shouldComponentUpdate()` to let React know if a component's output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

`shouldComponentUpdate()` is invoked before rendering when new props or state are being received. Defaults to `true`. This method is not called for the initial render or when `forceUpdate()` is used.

Returning `false` does not prevent child components from re-rendering when *their* state changes.

Currently, if `shouldComponentUpdate()` returns `false`, then `componentWillUpdate()`, `render()`, and `componentDidUpdate()` will not be invoked. Note that in the future React may treat `shouldComponentUpdate()` as a hint rather than a strict directive, and returning `false` may still result in a re-rendering of the component.

If you determine a specific component is slow after profiling, you may change it to inherit from `React.PureComponent`which implements `shouldComponentUpdate()` with a shallow prop and state comparison. If you are confident you want to write it by hand, you may compare `this.props` with `nextProps` and `this.state` with `nextState` and return `false` to tell React the update can be skipped.

We do not recommend doing deep equality checks or using `JSON.stringify()` in `shouldComponentUpdate()`. It is very inefficient and will harm performance.

`componentWillUpdate()`

```
componentWillUpdate(nextProps, nextState)
```

`componentWillUpdate()` is invoked immediately before rendering when new props or state are being received. Use this as an opportunity to perform preparation before an update occurs. This method is not called for the initial render.

Note that you cannot call `this.setState()` here; nor should you do anything else (e.g. dispatch a Redux action) that would trigger an update to a React component before `componentWillUpdate()` returns.

If you need to update `state` in response to `props` changes, use `componentWillReceiveProps()` instead.