

React Cheat Sheet v16

☐ Component☐ Children☐ Context☐ react-dom☐ test-utils☐ misc

render

react

[docs](#)

```
render() {  
  return <div />;  
}
```

constructor

react

[docs](#)

```
constructor(props) {  
  super(props);  
  this.state = {  
    list: props.initialList  
  };  
}  
  
// where props aren't used in constructor  
  
constructor() {  
  super();  
  this.state = {  
    list: []  
  };  
}
```

componentWillMount

react

[docs](#)

```
componentWillMount() {  
  // invoked once.  
  // fires before initial 'render'  
}
```

componentDidMount

react

[docs](#)

```
componentDidMount() {  
  // good for AJAX: fetch, ajax, or subscriptions.  
  
  // invoked once (client-side only).  
  // fires before initial 'render'  
}
```

componentWillReceiveProps

react

[docs](#)

```
componentWillReceiveProps(nextProps) {  
  // invoked every time component receives new props.  
  // does not fire before initial 'render'  
}
```

shouldComponentUpdate

react

[docs](#)

```
shouldComponentUpdate(nextProps, nextState) {  
  // invoked before every update (new props or state).  
  // does not fire before initial 'render'.  
}
```

componentWillUpdate

react

[docs](#)

```
componentWillUpdate(nextProps, nextState) {  
  // invoked immediately before update (new props or state).  
  // does not fire before initial 'render'.  
  
  // (see componentWillReceiveProps if you need to call setState)  
}
```

✗ **this.setState**

componentDidUpdate

react

[docs](#)

```
componentDidUpdate(prevProps, prevState) {  
  // invoked immediately after DOM updates  
  // does not fire after initial 'render'  
}
```

componentWillUnmount

react

[docs](#)

```
componentWillUnmount() {  
  // invoked immediately before a component is unmounted.  
}
```

setState (function)

react

[docs](#)

```
// good for state transitions  
  
this.setState((prevState, props) => {  
  return {count: prevState.count + props.step};  
});
```

setState (object)

react

[docs](#)

```
// good for static values

this.setState({mykey: 'my new value'});
```

setState (optional callback)

react

[docs](#)

```
// fires after setState
// prefer componentDidUpdate

this.setState(
  (prevState, props) => ({ count: prevState.count + props.step }),
  () => console.log(this.state.count)
);
```

forceUpdate

react

[docs](#)

```
// forces a re-render; AVOID if possible

this.forceUpdate();
```

displayName

react

[docs](#)

```
displayName: "MyComponent"
```

defaultProps

react

[docs](#)

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

CustomButton.defaultProps = {
  name: 'guest'
};
```

Children.map

react

[docs](#)

```
React.Children.map(this.props.children, (child, i) => {
  return child;
})
```

Children.forEach

react

[docs](#)

```
React.Children.forEach(this.props.children, (child, i) => {
  console.log(child + ' at index: ' + i);
})
```

Children.count

react

[docs](#)

```
React.Children.count(this.props.children);
```

Children.only

react

[docs](#)

```
React.Children.only(this.props.children);
```

Children.toArray

react

[docs](#)

```
React.Children.toArray(this.props.children)
```

Context (example)

react

[docs](#)

```
// requires 'prop-types' library

import { string } from "prop-types";

class Cowboy extends React.Component {
  childContextTypes: {
    salutation: string
  }

  getChildContext() {
    return { salutation: "Howdy" };
  }

  render() {
    return React.Children.only(this.props.children);
  }
}

const Greeting = (props, context) =>
  <div>{this.context.salutation} {this.props.name}</div>

Greeting.contextTypes = {
  salutation: PropTypes.string
}

// <Greeting name="Michael" />
// => Michael.

// <Cowboy><Greeting name="Michael" /></Cowboy>
// => Howdy Michael.
```

contextTypes

react

[docs](#)

```
// add to the context-aware component
// requires 'prop-types' library

contextTypes: {
  color: PropTypes.string
},
```

childContextTypes

react

[docs](#)

```
// add to the context provider
// requires 'prop-types' library

childContextTypes: {
  color: PropTypes.string
},
```

getChildContext

react

[docs](#)

```
// add to the context provider

getChildContext() {
  return {color: "purple"};
}
```

render

react-dom

[docs](#)

```
import { render } from "react-dom";
```



```
render(  
  <MyComponent />,  
  document.getElementById("component-root"),  
  () => console.log("MyComponent mounted.")  
);
```

hydrate

react-dom

[docs](#)

```
import { hydrate } from "react-dom";  
  
hydrate(  
  <MyComponent />,  
  document.getElementById("component-root"),  
  () => console.log("MyComponent hydrated.")  
);
```

unmountComponentAtNode

react-dom

[docs](#)

```
import { unmountComponentAtNode } from "react-dom";  
  
unmountComponentAtNode(document.getElementById('MyComponent'))
```

findDOMNode

react-dom

[docs](#)

```
import { findDOMNode } from "react-dom";  
  
findDOMNode(componentRef);
```

createPortal

react-dom

[docs](#)

```
import { createPortal } from "react-dom";

class MyPortalComponent extends React.Component {
  render() {

    return createPortal(
      this.props.children,
      document.getElementById("portal-element"),
    );
  }
}
```

renderToString

react-dom/server

[docs](#)

```
import { renderToString } from "react-dom/server";
ReactDOMServer.renderToString(<MyComponent />);
```

renderToStaticMarkup

react-dom/server

[docs](#)

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<MyComponent />);
```

renderToNodeStream

react-dom/server

[docs](#)

```
import { renderToNodeStream } from "react-dom/server";
renderToNodeStream(<MyComponent />);
```

renderToStaticNodeStream

react-dom/server

[docs](#)

```
import { renderToStaticNodeStream } from "react-dom/server";
renderToStaticNodeStream(<MyComponent />);
```

Simulate (basic)

react-dom/test-utils

[example](#) [docs](#)

```
var subject = TestUtils.renderIntoDocument(
  <div onClick={handleClick} />
);

TestUtils.Simulate.click(subject);
```

Simulate (with data)

react-dom/test-utils

[example](#) [docs](#)

```
function handleChange (event) {
  console.log('A change was simulated with key: ' + event.key);
}

var subject = TestUtils.renderIntoDocument(
  <input type="text" onChange={handleChange} />
);

TestUtils.Simulate.change(subject, { key: "Enter" });
```

renderIntoDocument

react-dom/test-utils

[example](#) [docs](#)

```
var componentTree = TestUtils.renderIntoDocument(<div><span /></div>);

console.log('You mounted a component tree with a ' + componentTree.tag
```

mockComponent

react-dom/test-utils

[docs](#)

```
// no example
```

isElement

react-dom/test-utils

[example](#)[docs](#)

```
expect(TestUtils.isElement(<div />)).toBe(true);
```

isElementOfType

react-dom/test-utils

[example](#)[docs](#)

```
var MyComponent = React.createClass({
  render () {
    return <div />;
  }
});

expect(
  TestUtils.isElementOfType(<MyComponent />, MyComponent)
).toBe(true);
```

isDOMComponent

react-dom/test-utils

[example](#)[docs](#)

```
var subject = TestUtils.renderIntoDocument(<div />);

expect(
  TestUtils.isDOMComponent(subject)
).toBe(true);
```

isCompositeComponent

react-dom/test-utils

[example](#)[docs](#)

```
var subject = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);  
  
expect(  
  TestUtils.isCompositeComponent(subject)  
) .toBe(true);
```

isCompositeComponentWithType

react-dom/test-utils

[example](#) [docs](#)

```
var CompositeComponent = React.createClass({  
  render () {  
    return <div />;  
  }  
});  
  
var subject = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);  
  
expect(  
  TestUtils.isCompositeComponentWithType(  
    subject,  
    CompositeComponent  
  )  
) .toBe(true);
```

findAllInRenderedTree

react-dom/test-utils

[example](#) [docs](#)

```
var CompositeComponent = React.createClass({  
  render () {  
    return <div><div /></div>;  
  }  
});  
  
var componentTree = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);
```

```
var allDivs = TestUtils.findAllInRenderedTree(  
  componentTree,  
  (c) => c.tagName === 'DIV'  
)  
  
expect(allDivs).toBeAn('array');  
expect(allDivs.length).toBe(2);
```

scryRenderedDOMComponentsWithClass

react-dom/test-utils

[example](#) [docs](#)

```
var CompositeComponent = React.createClass({  
  render () {  
    return (  
      <div className="target">  
        <div className="not-target">  
          <div className="target" />  
        </div>  
      </div>  
    );  
  }  
});  
  
var componentTree = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);  
  
var allDOMComponentsWithMatchingClass = TestUtils.scryRenderedDOMCompo  
  componentTree,  
  'target'  
);  
  
expect(allDOMComponentsWithMatchingClass).toBeAn('array');  
expect(allDOMComponentsWithMatchingClass.length).toBe(2);
```

findRenderedDOMComponentWithClass

react-dom/test-utils

[example](#) [docs](#)

```
var MyCompositeComponent = React.createClass({  
  render () {  
    return <MyNestedComponent />;  
  }  
})
```

```
});

var MyNestedComponent = React.createClass({
  render () {
    return <div className="nested"/>;
  }
});

var componentTree = TestUtils.renderIntoDocument(<MyCompositeComponent
  <MyNestedComponent />
/>);

var singleComponentWithMatchedClass = TestUtils.findRenderedDOMComponentWithClass(
  componentTree,
  'nested'
);

expect(singleComponentWithMatchedClass).toBeAn('object');
expect(singleComponentWithMatchedClass).toNotBeAn('array');
expect(singleComponentWithMatchedClass.className).toBe('nested');
```

scryRenderedDOMComponentsWithTag

react-dom/test-utils

[example](#) [docs](#)

```
var CompositeComponent = React.createClass({
  render () {
    return <div><div /></div>;
  }
});

var componentTree = TestUtils.renderIntoDocument(
  <CompositeComponent />
);

var allDivs = TestUtils.scryRenderedDOMComponentsWithTag(
  componentTree,
  'DIV'
);

expect(allDivs).toBeAn('array');
expect(allDivs.length).toBe(2);
```

findRenderedDOMComponentWithTag

react-dom/test-utils

[example](#) [docs](#)

```

var MyCompositeComponent = React.createClass({
  render () {
    return <MyNestedComponent />;
  }
});

var MyNestedComponent = React.createClass({
  render () {
    return <div />;
  }
});

var componentTree = TestUtils.renderIntoDocument(<MyCompositeComponent />);

var onlyDiv = TestUtils.findRenderedDOMComponentWithTag(
  componentTree,
  'div'
);

expect(onlyDiv).toBeAn('object');
expect(onlyDiv).toNotBeAn('array');
expect(onlyDiv.tagName).toBe('DIV');

```

scryRenderedComponentsWithType

react-dom/test-utils

[example](#) [docs](#)

```

var MyCompositeComponent = React.createClass({
  render () {
    return (
      <div>
        <Target />
        <br />
        <Target />
      </div>
    )
  }
});

var Target = React.createClass({
  render () {
    return <div />;
  }
});

var componentTree = TestUtils.renderIntoDocument(

```



```
    <MyCompositeComponent />
  );

var allTargetComponents = TestUtils.scryRenderedComponentsWithType(
  componentTree,
  Target
);

expect(allTargetComponents).toBeAn('array');
expect(allTargetComponents.length).toBe(2);
```

findRenderedComponentWithType

react-dom/test-utils

[example](#) [docs](#)

```
var MyCompositeComponent = React.createClass({
  render () { return <TargetComponent /> }
});

var TargetComponent = React.createClass({
  render () { return <div /> }
});

var componentTree = TestUtils.renderIntoDocument(
  <MyCompositeComponent />
);

var onlyTargetComponent = TestUtils.findRenderedComponentWithType(
  componentTree,
  TargetComponent
);

expect(onlyTargetComponent).toBeAn('object');
expect(onlyTargetComponent).toNotBeAn('array');
expect(TestUtils.isCompositeComponentWithType(
  onlyTargetComponent,
  TargetComponent
)).toBe(true);
```

Shallow rendering (basics)

react-dom/test-utils

[example](#) [docs](#)

```
// 1. create a renderer
```

```
var renderer = TestUtils.createRenderer();

// 2. render component into renderer
renderer.render(<MyComponent />);

// 3. capture renderer output
var subject = renderer.getRenderOutput();

// 4. make assertions
expect(subject.type).toBe('div');
```

Shallow rendering (type example)

react-dom/test-utils

[example](#) [docs](#)

```
var renderer = TestUtils.createRenderer();

renderer.render(<MyComponent />);

var subject = renderer.getRenderOutput();

expect(subject.type).toBe('div'); // => true
```

Shallow rendering (props example)

react-dom/test-utils

[example](#) [docs](#)

```
var renderer = TestUtils.createRenderer();

renderer.render(<MyComponent className="my-component" />);

var subject = renderer.getRenderOutput();

expect(subject.props.className).toBe('my-component'); // => true
```

Shallow rendering (child-count example)

react-dom/test-utils

[example](#) [docs](#)

```
var renderer = TestUtils.createRenderer();
```

```
renderer.render(  
  <MyList items={[1, 2, 3]} />  
);  
  
var subject = renderer.getRenderOutput();  
  
var childCount = React.Children.count(subject.props.children);  
  
expect(childCount).toBe(3); // => true
```

Shallow rendering (child-equality example)

react-dom/test-utils

[example](#) [docs](#)

```
var renderer = TestUtils.createRenderer();  
  
renderer.render(  
  <MyComponent>  
    <div>Thing 1</div>  
    <div>Thing 2</div>  
  </MyComponent>  
);  
  
var subject = renderer.getRenderOutput();  
  
expect(subject.props.children).toEqual([  
  <div>Thing 1</div>,  
  <div>Thing 2</div>  
]); // => true
```

Shallow rendering (events example)

react-dom/test-utils

[example](#) [docs](#)

```
var renderer = TestUtils.createRenderer();

var spy = expect.createSpy();

renderer.render(<MyComponent onClick={spy} />);

var subject = renderer.getRenderOutput();

expect(spy.call.length).toEqual(1); // => true
```

Shallow rendering (state changes example)

react-dom/test-utils

[example](#) [docs](#)

```
var renderer = TestUtils.createRenderer();

renderer.render(<ClickCounter />);

// test initial rendering
var result = renderer.getRenderOutput();

expect(result.props.children).toEqual(0);

// test post-click rendering
result.props.onClick();

var clickedResult = renderer.getRenderOutput();

expect(clickedResult.props.children).toEqual(1);
```

Lifecycle methods

NAME	CALLED FOR	RECEIVES CONTEXT	SETSTATE() TRIGG
componentWillMount	initial render()	no	no

NAME	CALLED FOR	RECEIVES CONTEXT	SETSTATE() TRIGG
componentDidMount	initial render()	no	yes
componentWillReceiveProps	new props	yes	yes
shouldComponentUpdate	new props/state	yes	yes
componentWillUpdate	new props/state	yes	n/a
componentDidUpdate	new props/state	no	yes
componentWillUnmount	unmounting	no	n/a

ref (class component)

[docs](#)

```
class AutoFocusTextInput extends React.Component {
  componentDidMount() {
    this.textInput.focus();
  }

  render() {
    return (
      <CustomTextInput
        ref={(input) => { this.textInput = input; }} />
    );
  }
}
```

ref (functional component)

[docs](#)

```
function CustomTextInput(props) {
  let textInput = null;

  function handleClick() {
    textInput.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={ref} />
    </div>
  );
}
```

```
      ref={(input) => { textInput = input; }} />
    <input
      type="button"
      value="Focus the text input"
      onClick={handleClick}
    />
  </div>
);
}
```

functional component

react

[docs](#)

```
const Greeting = props => <div>Hello {props.name}</div>;
```

functional component (with context)

react

[docs](#)

```
import { string } from "prop-types";

const Greeting = (props, context) => (
  <div>{context.salutation} {props.name}</div>
);

Greeting.contextTypes = { salutation: string };
```

Learn React



Learn all about **functional components** in a new course by Learn React.