

Contents

JSX

React Elements

React Element Tags

React Element Attributes

React Elements Embedded Javascript

React Element Inline Styles

React Fragments

React Components

Functional Components

Component Props

The Children Prop

Conditional Rendering

Lists in Components

Memo

Context

Hooks

useState

useEffect

useLayoutEffect

useRef

useCallback

[useMemo](#)

[useContext](#)

Class Component

[Constructor](#)

[State](#)

Lifecycle Methods

[componentDidMount](#)

[componentWillUnmount](#)

[componentDidUpdate](#)

Error Boundaries

[getDerivedStateFromError](#)

[componentDidCatch](#)

Useful Links

JSX

Almost all React code is written in JSX. JSX is a syntax extension of Javascript that allows us to write HTML-like syntax in Javascript.

React Elements

React Element Tags

React elements look just like HTML, in fact they render the same equivalent HTML elements.

```
<h1>My Header</h1>
<button>My Button</button>
<ul>
  <li>list item 1</li>
  <li>list item 2</li>
  <li>list item 3</li>
</ul>
```

Single tag elements like the `img` element and the `br` element must be closed like so

```

<br />
```

React Element Attributes

React elements have different attributes compares to their HTML counterparts. Since JSX is still javascript, we use camelcase. Also, `class` is a protected keyword in javascript (creating classes) so the HTML `class` attribute in JSX is `className`.

```
<div className="my-container">
  
</div>
```

React Elements Embedded Javascript

The power of JSX is that it's javascript and HTML. This means you can write javascript to render different attributes directly in your javascript using curly braces `{}` .

```
const divClass = "my-div-class"

<div className={divClass}></div>
```

React Element Inline Styles

React elements just like HTML elements can use the `style` attribute, but you pass them as javascript objects instead of double quote strings.

In HTML

```
<h1 style="color:blue;text-align:center">This is a header</h1>
```

In JSX

```
<h1 style={{ color: 'blue', textAlign: 'center' }}>This is a header</h1>
```

React Fragments

React has a special element called a *fragment*. It's a special element that doesn't actually render into the DOM, but can act as a parent to a group of elements.

```
import { Fragment } from 'react'

<Fragment>
  <h1> My H1 </h1>
  <p> My Paragraph </p>
</Fragment>
```

If you don't want to import `Fragment` from the React library, you can also use `<>` .

```
<>
  <h1> My H1 </h1>
  <p> My Paragraph </p>
</>
```

Fragments are useful in components since components require us to return one parent level element, and if we don't want to needlessly add HTML elements to our website, we can just use fragments.

React Components

Components are the building blocks of your web application. We use them to organize groups of React elements together so they're reusable. There are two kinds of components, class components and functional components but functional components are the de facto standard today. They both follow the same two rules:

1. Component names must be capitalized i.e. MyComponent instead of myComponent
2. They must return JSX, more specifically one parent level JSX element (more on this later).

Functional Components

Functional components are just javascript functions that return JSX.

Here's how you create a functional component using function declaration:

```
function MyComponent() {  
  return <h1>My Component</h1>  
}
```

You can also use an arrow function:

```
const MyComponent = () => {  
  return <h1>My Component</h1>  
}
```

The component can then be used like any React element.

```
const MyComponent = () => {  
  return <h1>My Component</h1>  
}  
  
const MyOtherComponent = () => {  
  return (  
    <div>  
      <MyComponent />  
      <p> Sample Text </p>  
    </div>  
  )  
}
```

```
    </div>
  )
}
```

Component Props

We can pass data to our components through custom attributes on the component element. We can choose any name for the attribute as long they don't overlap with the existing general element attributes (i.e. `className`, `styles`, `onClick` etc.). These properties are then grouped into an object where the attribute name is the key, and the value is the value. Here we are passing a prop `title` from the `App` component to our component `MyComponent`.

```
const MyComponent = (props) => {
  return <h1>{props.title}</h1>
}

const App = () => {
  return (
    <MyComponent title="Hello World" /> // Props == { title: "Hello Wor
  )
}
```



Reminder: you can embed any of the values in your props object in JSX since it's just javascript, just remember to use curly braces (`{}`).

Since the props are just an object, it's common to destructure the values for cleaner, simpler code.

```
const MyComponent = ({title}) => {
  return <h1>{title}</h1>
}

const App = () => {
  return (
    <MyComponent title="Hello World" />
  )
}
```

Any JavaScript value can be passed as a prop such as arrays, objects, other elements and components!

The Children Prop

All component have a special prop called `children`. Any data (usually components and react elements) sitting between the opening and closing tags of the component get passed in as `children`.

```
const Greeting = ({ children }) => {
  return children //<h1> Hello World! </h1>
}

const App = () => {
  return (
    <Greeting>
      <h1>Hello World!</h1>
    </Greeting>
  )
}
```

We can render it anywhere in our component's JSX! Just remember that it's a javascript variable so make sure it's wrapped in curly braces `{}`.

```
const GreetingCard = ({ children }) => {
  return (
    <div>
      <h1> Greetings! </h1>
      {children}
    </div>
  )
}

const App = () => {
  return (
    <GreetingCard>
      <p>Example Children Paragraph</p>
      <button> Example Children Button</button>
    </GreetingCard>
  )
}
```

Conditional Rendering

Since our components are written in JSX which is just javascript, we can conditionally render different things with javascript. A basic example is to use an `if` statement in our functional component.

```
const Greeting = ({large}) => {
  if(large) {
    return <h1> Hello World! </h1>
  }
  return <p> Hello World! </p>
}

const App = () => {
  return (
    <div>
      <Greeting large={true}/> // <h1> Hello World! </h1>
      <Greeting large={false}/> // <p> Hello World! </p>
    </div>
  )
}
```

We can also use a ternary operator!

```
const Greeting = ({large}) => {
  return (large ? <h1> Hello World! </h1> : <p> Hello World! </p>)
}

const App = () => {
  return (
    <div>
      <Greeting large={true}/> // <h1> Hello World! </h1>
      <Greeting large={false}/> // <p> Hello World! </p>
    </div>
  )
}
```

In a component, if we return `null` nothing will render to the DOM.

```
const Greeting = ({display, message}) => {
  return ( display ? <h1> {message} </h1> : null )
}
```



```
const App = () => {
  return (
    <div>
      <Greeting message="rendered!" display={true}/> // <h1> rendered
      <Greeting message="not rendered!" display={false}/> // nothing
    </div>
  )
}
```

Lists in Components

If we want to duplicate elements/components, we can do so by looping through an array with the `.map()` method as long as we return JSX from the callback.

Remember, this is javascript so wrap your `map` call in `{}`. We must remember to add the attribute `key` to the top level JSX element we return from `map`; the value must also be a unique value for each iteration.

```
const ShoppingList = ({items}) => {
  return (
    <ul>
      {items.map((item) => <li key={item}> {item} </li>)}
    </ul>
  )
}

const App = () => {
  const groceries = ['broccoli', 'carrots', 'chicken', 'garlic'];
  return <ShoppingList items={groceries} />
}
```

IMPORTANT REMINDER Remember, when mapping over an array into JSX, you **must** include a `key` attribute with a unique value. It's tempting to use the index from `map` but this may cause issues down the road.

```
const ShoppingList = ({items}) => {
  return (
    <ul>
      // DON'T DO THIS
      {items.map((item, idx) => <li key={idx}> {item} </li>)}
    </ul>
  )
}
```

```

        </ul>
      )
    }

    const App = () => {
      const groceries = ['broccoli', 'carrots', 'chicken', 'garlic'];
      return <ShoppingList items={groceries} />
    }

```

The reason is because React uses the `key` to determine which components to re-render, with the `keys` themselves being unique identifiers hence why the values need to be unique. Indexes are just numbers and in lists where there are multiple maps, if you always use the index as the `key` React may get confused.

```

const ShoppingList = () => {
  const vegetables = ['broccoli', 'carrots', 'garlic'];
  const meats = ['chicken', 'beef']

  return (
    <ul>
      {vegetables.map((veg, idx) => <li key={idx}> {veg} </li>)}
      {meats.map((meat, idx) => <li key={idx}> {meat} </li>)}
    </ul>
  )
}

```

As you can see in the above example, the `map` of vegetables has indexes 0, 1 and 2 since there are 3 items. The `map` for meats will be have index 0 and 1 since there are 2 items. This is a problem since React can't differentiate the vegetable item with `key 0` and the meat item with `key 0`, or the vegetable item with `key 1` and the meat item with `key 1`. This is why we need to use unique `keys`!

Below we can fix this using the `name` attribute, which we know are unique.

```

const ShoppingList = () => {
  const vegetables = ['broccoli', 'carrots', 'garlic'];
  const meats = ['chicken', 'beef']

  return (
    <ul>
      {vegetables.map((veg) => {<li key={veg}> {veg} </li>})}

```

```

        {meats.map((meat) => {<li key={meat}> {meat} </li>})}
      </ul>
    )
  }
}

```

Memo

Use `react.memo` allows a component to skip re-rendering if it's props haven't changed when it's parent re-renders.

When a component re-renders, all its child components will also re-render. Wrapping a component in `react.memo` will prevent that component re-rendering caused by upstream re-renders if the props have not changed.

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
```

`react.memo` take two arguments:

1. The functional component we want to memoize.
2. An **optional** function that determines if the component should re-render. It receives two arguments, the component's previous props and its new props.

```

import { memo } from 'react'

const MemoizedComponent = memo(
  function SomeComponent(props) {},
  (prevProps, nextProps) => { //return true or false }
)

```

The second argument is a custom comparison function that for the majority of circumstances are not needed. When not passed a second argument, `react` defaults to shallow comparison of the old props and new props with `Object.is`. If you need custom comparison, the function should return `true` if the props are equal and therefore **not** re-render. Or it should return `false` if the props are not equal and **should** re-render.

Context

If we had some data values we wanted to share across multiple sibling or nested child components, we would have to lift that data up to a commonly shared parent and needlessly drill it down through multiple components and their props. Take the following example

```
const GrandChildA = ({ text }) => { // I need text
  return (
    <div>
      <h1> Grand Child A </h1>
      <p> {text} </p>
    </div>
  )
}

const ChildA = ({ text }) => { // I don't need text
  return (
    <div>
      <h1> Child A </h1>
      <GrandChildA text={text} />
    </div>
  )
}

const ParentA = ({ text }) => { // I don't need text
  return (
    <div>
      <h1> Parent A </h1>
      <ChildA text={text} />
    </div>
  )
}

const ChildB = ({ text }) => { // I need the text
  return (
    <div>
      <h1> Child B </h1>
      <p> {text} </p>
    </div>
  )
}

const ParentB = ({ text }) => { // I don't need text
  return (
    <div>
      <h1> Parent B </h1>
      <ChildB text={text} />
    </div>
  )
}
```

```

    )
  }

  const App = () => {
    const text = "Hello World";

    return (
      <div>
        <h1> App </h1>
        <ParentA text={text} />
        <ParentB text={text} />
      </div>
    )
  }

```

Here, we have to pass the `text` value from the `App` component all the way through `Parent A` and `ChildA` just so `GrandChildA` can receive it, even though `ParentA` and `ChildA` don't need the `text` other than to pass it down. The same is true for `ParentB` in order to get the `text` value to `ChildB`. This is called *prop drilling*.

We can't move the `text` value down the tree since both `GrandChildA` and `ChildB` need it, and `App` is the lowest common parent between them. This makes our code extremely brittle since moving and the components serving to pass the `text` prop needlessly complex.

We can solve this with `React Context` which allows us to lift the data into a special component called `Context` that allows any of it's children no matter where they are to access it's values without the need for prop drilling.

We need the `createContext` function from `React` and pass it the initial value we want to share. It returns us back an object that contains two components:

1. the `Provider` component which we wrap around the portion of the component tree we want to access the value.
2. The `Consumer` component which has access to the values from the created context which we place in any component that needs the value.

```

import { createContext } from 'react';

const TextContext = createContext('');

```

```
const GrandChildA = () => { // I need text
  return (
    <div>
      <h1> Grand Child A </h1>
      <TextContext.Consumer>
        {text => <p> {text} </p>}
      </TextContext.Consumer>
    </div>
  )
}

const ChildA = () => { // I don't need text
  return (
    <div>
      <h1> Child A </h1>
      <GrandChildA />
    </div>
  )
}

const ParentA = () => { // I don't need text
  return (
    <div>
      <h1> Parent A </h1>
      <ChildA />
    </div>
  )
}

const ChildB = () => { // I need the text
  return (
    <div>
      <h1> Child B </h1>
      <TextContext.Consumer>
        {text => <p> {text} </p>}
      </TextContext.Consumer>
    </div>
  )
}

const ParentB = () => { // I don't need text
  return (
    <div>
      <h1> Parent B </h1>
      <ChildB />
    </div>
  )
}
```

```
const App = () => {  
  return (  
    <TextContext.Provider value="Hello World">  
      <ParentA />  
      <ParentB />  
    </TextContext.Provider>  
  )  
}
```

Hooks

Hooks were introduced in React version 16.8 as a way to extend additional functionality into functional components. Previously this functionality was only available to class components, but through hooks we can super charge our functional components!

To better understand hooks, we need to understand the React component lifecycle. There are three main phases of any React component:

1. The mounting phase when a component is created and inserted into the DOM. This is the initial render and only happens once in a components lifecycle.
2. The updating phase is when a component re-renders due to updates. This happens either due to *prop* changes or *state* changes (more below).
3. The final phase is the un-mounting phase, when a component is removed from the DOM.

Hooks are normally called at the *top* of our components.

useState

useState hook allows us to store values scoped to a component. Any changes to those values will cause the component and any of it's child components to re-render.

As mentioned above, components re-render in the updating phase (2) due to prop changes and state changes. State is data stored inside of a component that can be updated/changed. When this state data changes, this will trigger a re-render of the component. While we can store and change data in a variable, those changes will not

trigger a re-render. With the `useState` hook, it does allow us to trigger re-renders on changes to that data.

`useState` is a function that we can pass an optional argument representing the initial value we want to store. The `useState` hook returns back an array containing two values, the first is the current state value, the second is a setter function to update this state value.

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

const MyComponent = () => {
  const [value, setValue] = useState(initialValue);
}
```

The `setValue` function we de-structured is called the **setter** function. When we call this setter function, we pass it the new value we want to set the state to.

Let's look at a basic counter example.

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

const Counter = () => {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      {count}
      <button onClick={increment}> increment </button>
      <button onClick={decrement}> decrement </button>
    </div>
  )
}
```

Whenever `setCount` is called, the `Counter` component will re-render, which is the behaviour we want since `count` is has updated and we want our DOM to display the new value.

It's important to note that the setter function from `useState` is asynchronous. This means that if you try to log the state immediately after setting it, you might not see the updated state.

useEffect

useEffect is a hook that allows us to create side effects in our functional components.

useEffect takes two arguments:

1. The first argument is a callback function called the *effect function* that contains the side effect code we want to run.
2. The second argument is an array called the *dependency array* which contains values from outside the scope of the *effect function*. Whenever one of these values changes, useEffect will run the *effect function*.

```
import { useEffect } from 'react'

const MyComponent = () => {
  useEffect(() => {
    // side effect code here
  }, [// dependencies go here]);
}
```

The *effect function* will run:

1. Once when the component mounts.
2. Whenever any value in the dependency array changes.

A common use case for useEffect is to fetch some data and store it in a state variable.

```
import { useState, useEffect } from 'react'

const UserList = () => {
  const [userList, setUserList] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => response.json())
      .then(users => setUserList(users))
  }, []);
}
```

```

    return (
      <div>
        { userList.map(user => <h2 key={user.id}> {user.name} </h2>) }
      </div>
    )
  }
}

```

Passing an empty *dependency array* will only call our *effect function* once during the mounting phase since there are no dependencies to react to.

The *effect function* will run every time a value in the dependency array changes. Values the *effect function* relies on but comes from outside of its scope are added to the *dependency array*. These include props:

```

import { useState, useEffect } from 'react'

const UserList = ({sourceURL}) => {
  const [userList, setUserList] = useState([]);

  useEffect(() => {
    fetch(sourceURL)
      .then(response => response.json())
      .then(users => setUserList(users))
  }, [sourceURL]);

  return (
    <div>
      { userList.map(user => <h2 key={user.id}> {user.name} </h2>) }
    </div>
  )
}

```

As well as other state variables:

```

import { useState, useEffect } from 'react';

import User from '../components/user';
import { userAPI } from '../api/userAPI';

const UserList = () => {
  const [userName, setUserName] = useState('');
  const [user, setUser] = useState(null);

  const handleTextChange = (event) => {

```

```

        setUsername(event.target.value);
    }

    useEffect(() => {
        userAPI.getByUsername(userName)
            .then(user => setUser(user))
    }, [userName]);

    return (
        <div>
            <h2> Search by username </h2>
            <input type='text' onChange={handleTextChange} />
            <User user={user} />
        </div>
    )
}

```

If we want to run a callback when the component unmounts, we can do so by returning that callback from the *effect function*. This is useful for cleanup functions that need to undo effects like subscriptions.

```

useEffect(() => {
    ChatAPI.subscribeToUser(userID);

    return () => { // This function runs on unmount
        ChatAPI.unsubscribeFromUser(userID);
    };
}, []);

```

It's important to note, the *effect function* runs *after* React renders/re-renders the component to ensure our effect callback does not block browser painting.

useLayoutEffect

The `useLayoutEffect` hook is almost identical to the `useEffect` hook except it runs the effect callback immediately after DOM changes.

The one key difference between the `useLayoutEffect` hook and `useEffect` hook: `useLayoutEffect` runs the effect callback synchronously immediately after React has performed all DOM mutations, but before the browser has a chance to paint. This is useful if you need to make DOM mutations and don't want the screen to flicker between renders.

```
import { useEffect } from 'react'

const MyComponent = () => {
  useEffect(() => {
    // side effect code here
  }, [// dependencies go here]);
}
```

useRef

useRef is a hook that stores a value in a component like **useState** except changes to that value won't cause the component to re-render.

It accepts one argument as the initial value and returns a *reference* object.

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

const MyComponent = () => {
  const ref = useRef(initialValue);

  // ...remaining component code
}
```

The value of `ref` is:

```
{
  current: initialValue
}
```

We can access and mutate the current value of the ref through the `ref.current` property. This value will persist across re-renders.

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

export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
  }
}
```

```

    }

    return (
      <>
        <h1>count: {ref.current} </h1>
        <button onClick={handleClick}>
          Click me!
        </button>
      </>
    );
  }
}

```

Every time we click the button and trigger `handleClick`, we are incrementing the `ref.current` value. However, because this mutation does not trigger the component to re-render, so the count does not update in the DOM even though the stored value is updating.

It is common to store DOM node references in a `ref`.

```

import { useRef, useEffect } from 'react';

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, []);

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

```

When the `InputFocus` component mounts, we will call on the DOM node for the input and automatically focus it.

useCallback

The `useCallback` hook is a performance improvement hook that prevents functions from being needlessly recreated between re-renders.

Whenever a component renders or re-renders, any functions that are created are recreated. In the component below, we create a `hideUser` function that we use in the button.

The `useCallback` hook signature takes two arguments:

1. The function we want to persist between re-renders.
2. A dependency array containing values that tells `useCallback` when to recreate the function when any of them change.

```
import { useMemo } from 'react'

const MyComponent = () => {
  const computedValue = useMemo(
    () => { //...computationally expensive function },
    [//dependencies]
  )
}
```

Looking at an example

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

const UserList = ({ users = [] }) => {
  const [shownUsers, setShownUsers] = useState(users);

  const hideUser = (userID) => {
    const newUsers = users.filter(user => user.id !== userID);
    setShownUsers(newUsers);
  };

  return (
    <div>
      {userList.map(user => (
        <div key={user.id}>
          <h2> {user.name} </h2>
          <button onClick={() => hideUser(user.id)}> hide user </
        </div>
      ))}
    </div>
  )
}
```

```

const App = () => {
  const users = [
    {id: 1, name: 'Mike'},
    {id: 2, name: 'Steve'},
    {id: 3, name: 'Andrew'},
    {id: 4, name: 'Pierre'}
  ];

  return <UserList users={users} />
}

```

In the `UserList` component, we receive a `users` array as a prop. We create a `hideUser` function that filters out the selected user. However, every time we set `shownUsers` and re-render the component, we recreate the `hideUser` function even though we don't need to. This recreation costs us performance.

We can wrap it in the `useCallback` hook which prevents our functions from being recreated during re-renders.

```

import { useState, useCallback } from 'react';

const UserList = ({ users = [] }) => {
  const [shownUsers, setShownUsers] = useState(users);

  const hideUser = useCallback((userID) => {
    const newUsers = users.filter(user => user.id !== userID);
    setShownUsers(newUsers);
  }, [users]);

  return (
    <div>
      {userList.map(user => (
        <div key={user.id}>
          <h2> {user.name} </h2>
          <button onClick={() => hideUser(user.id)}> hide user </button>
        </div>
      ))}
    </div>
  )
}

const App = () => {
  const users = [
    {id: 1, name: 'Mike'},

```

```

    {id: 2, name: 'Steve'},
    {id: 3, name: 'Andrew'},
    {id: 4, name: 'Pierre'}
  ];

  return <UserList users={users} />
}

```

Now the `hideUser` function will only recreate if the `users` prop changes.

useMemo

The `useMemo` is a performance improvement hook that *memoizes* the return value of a function.

Memoization is the caching of computed results from a expensive function call and returning the cached result when the function is called with the same inputs.

It follows the same signature as `useCallback` taking two arguments:

1. The function we want memoize.
2. A dependency array containing values that the function uses. When any of these values change it tells `useMemo` to rerun the function (with the new values) and memoize the new return value.

```

import { useMemo } from 'react'

const MyComponent = () => {
  const computedValue = useMemo(
    () => { //...computationally expensive function },
    [//dependencies]
  )
}

```

Lets look at an example

```

import { useState } from 'react';

const factorialOf = (num) => {

```



```

    if (num < 0)
      return -1;
    else if (num === 0)
      return 1;
    else {
      return (num * factorialOf(num - 1));
    }
  }
}

const Factorial = () => {
  const [number, setNumber] = useState(0);
  const [bool, setBool] = useState(true);

  const clickHandler = () => setBool(!bool);

  const factorialNumber = factorialOf(number); // Computationally expensi

  return (
    <div>
      <input type='text' onChange={(event) => setNumber(event.target.
      <h2>{ factorialNumber }</h2>
      <button onClick={clickHandler}> re-render </button>
    </div>
  )
}

```

In this example, the `factorialOf` function is an expensive function to calculate. The `clickHandler` function on the button click only serves to re-render our component by calling a state setter function. Every time we render/re-render the component, `factorialNumber` is recalculated, even when `number` hasn't changed. It's expensive to recalculate `factorialNumber` needlessly whenever the component re-renders (it's parent re-renders or state changes unrelated to `number` i.e. `setBool` is called). We can fix this with `useMemo`

```

import { useState, useMemo } from 'react';

const factorialOf = (num) => {
  if (num < 0)
    return -1;
  else if (num === 0)
    return 1;
  else {
    return (num * factorialOf(num - 1));
  }
}

```

```

}

const Factorial = () => {
  const [number, setNumber] = useState(0);
  const [bool, setBool] = useState(true);

  const clickHandler = () => setBool(!bool);

  const factorialNumber = useMemo(factorialOf(number), [number]); // Comp

  return (
    <div>
      <input type='text' onChange={(event) => setNumber(event.target.
      <h2>{ factorialNumber }</h2>
      <button onClick={clickHandler}> re-render </button>
    </div>
  )
}

```

Here, `factorialNumber` will compute the value from `factorialOf` once per input of number and cache it. Any future calls of `factorialOf` with the same number will return the cached value saving us expensive and needless re-computations.

useContext

`useContext` allows us to access Context without need to use it's Consumer component.

Without `useContext`

```

import { createContext } from 'react';

const TextContext = createContext('');

const ChildA = () => {
  return (
    <div>
      <TextContext.Consumer>
        {text => <p> {text} </p>}
      </TextContext.Consumer>
    </div>
  )
}

```

```

const ParentA = () => {
  return (
    <div>
      <h1> Parent A </h1>
      <ChildA />
    </div>
  )
}

const App = () => {
  return (
    <TextContext.Provider value="Hello World">
      <ParentA />
    </TextContext.Provider>
  )
}

```

To use the useContext hook, we pass it the context value we want to access.

```

import { createContext, useContext } from 'react';

const TextContext = createContext('');

const ChildA = () => {
  const text = useContext(TextContext);

  return (
    <div>
      <p> {text} </p>
    </div>
  )
}

const ParentA = () => {
  return (
    <div>
      <h1> Parent A </h1>
      <ChildA />
    </div>
  )
}

const App = () => {
  return (
    <TextContext.Provider value="Hello World">

```

```
        <ParentA />
      </TextContext.Provider>
    )
  }
```

Class Component

Class components are the other type of component we can write. The common practice is to use functional components but there are some existing use cases that still call for class components. The JSX for the component is returned from the `render` method.

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

class MyClassComponent extends Component {
  render() {
    return (
      <div>
        <h1> My Class Component </h1>
      </div>
    )
  }
}
```

Constructor

The **constructor** method runs before the class component mounts.

Typically used to declare the initial state and bind custom class methods to the class instance.

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

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    this.increment = this.increment.bind(this);
  }
}
```

```
    increment() {  
      // increment code  
    }  
    ...  
  }
```

However, modern javascript syntax does not require a constructor. We can just use public class fields.

```
class Counter extends Component {  
  state = { counter: 0 };  
  
  increment() {  
    // increment code  
  }  
  ...  
}
```

You access all methods, props and state of the class instance with the `this` keyword.

State

After initializing state in the component, the class instance has a `setState` method that you must use if you wish to update the state. **Do not mutate the state object!**

```
setState(nextState, callback?)
```

You can pass `setState` two arguments, with the second being optional.

1. The next state, either an object or a function.

- a) With an object, just with the new fields you want updated. React will perform a shallow merge of the new state object against the previous state object.

```
class Form extends Component {  
  state = {  
    name: '',  
  },
```

```

        address: ''
    };

    handleNameChange = (e) => {
        const newName = e.target.value;
        this.setState({
            name: newName
        });
    }

    handleAddressChange = (e) => {
        const newAddress = e.target.value;
        this.setState({
            address: newAddress
        });
    }

    render() {
        return (
            <>
                <input value={this.state.name} onChange={this.handleNameChange} />
                <input value={this.state.address} onChange={this.handleAddressChange} />
                <p>Hello, {this.state.name}. You live at {this.state.address} </p>
            </>
        );
    }
}

```

b) With a pure function that receives the `prevState` and `props`. It should return the state object to be shallowly merged with the previous state object.

```

class Counter extends Component {
    state = {
        count: 0,
    };

    increment = () => {
        this.setState((prevState) => ({count: prevState.count + 1}))
    }

    decrement = () => {
        this.setState((prevState) => ({count: prevState.count - 1}))
    }

    render() {

```

```

    return (
      <>
        <p> { this.state.count } </p>
        <button onClick={this.increment}> Increment </button>
        <button onClick={this.decrement}> Decrement </button>
      </>
    );
  }
}

```

1. An optional callback that runs *after* the state is updated.

```

class Form extends Component {
  state = {
    name: '',
  };

  handleNameChange = (e) => {
    const newName = e.target.value;
    this.setState({
      name: newName
    }, () => { console.log('state updated!') });
  }

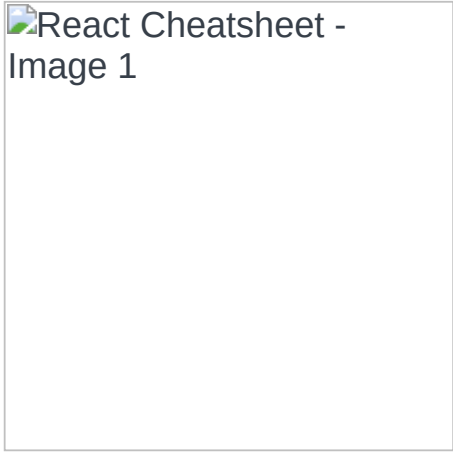
  render() {
    return (
      <>
        <input value={this.state.name} onChange={this.handleNameChange} />
        <p>Hello, {this.state.name} </p>
      </>
    );
  }
}

```

Lifecycle Methods

Class components also have methods that hook into React's rendering and re-rendering cycles called *lifecycle methods*. Many methods are now deprecated or considered UNSAFE as the React team is pushing forward with functional components + hooks. This cheatsheet will only reference the commonly used ones.

[The full list can be found here](#)

React Cheatsheet -
Image 1

componentDidMount

The `componentDidMount` method calls *after* react mounts the component to the DOM.

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

class UserList extends Component {
  state = { users: [] }

  componentDidMount() {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => response.json())
      .then(users => this.setState({users: users}))
  }

  render() {
    return (
      <div>
        <h1> My Class Component </h1>
      </div>
    )
  }
}
```

componentWillUnmount

The `componentWillUnmount` method is called immediately *before* the component unmounts. It is commonly used for cleanup.

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

import { ChatAPI } from '../api/chatAPI';

class ChatWindow extends Component {
  componentDidMount() {
    ChatAPI.subscribeToUser(userID);
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromUser(userID);
  }
  render() {...}
}
```

componentDidUpdate

The `componentDidUpdate` method is called immediately after the component has re-rendered from state or prop changes (ignoring the initial render).

A method that receives previous props and previous state which can be compared with new state and props. This is often used to make network requests if needed. Take the following example

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

import { chatAPI } from '../api/chatAPI';

class ChatWindow extends Component {
  state = {
    serverURL: 'https://www.SomeChatServer.com'
  };

  componentDidMount() {
    // open new subscription
    chatAPI.subscribe(this.serverUrl, this.props.roomID);
  }

  componentDidUpdate(prevProps, prevState) {
    // Check if we've changed rooms or server
  }
}
```

```

    if (
      this.props.roomID !== prevProps.roomID ||
      this.state.serverUrl !== prevState.serverURL
    ) {
      // unsubscribe from previous room and server
      chatAPI.unsubscribe(prevState.serverURL, prevProps.roomID);
      // subscribe to new room and server
      chatAPI.subscribe(this.serverUrl, this.props.roomID);
    }
  }

  componentWillUnmount() {
    // close all subscriptions
    chatAPI.unsubscribeAll();
  }

  // ...
}

```

Error Boundaries

getDerivedStateFromError

Error boundaries are components that catch errors that occur anywhere in their child component tree. It allows us to display some fallback UI instead of the component that crashed.

It is a class component we create that has a `static getDerivedStateFromError` method which can update state in response to an error.

We wrap this error boundary component around the portion of our component tree we want our fallback UI for. To set this up, we create state that tracks whether or not there is an error. From the `static getDerivedStateFromError` method, we return the new state object with the updated error state.

```

import React, { Component } from 'react';

class ErrorBoundary extends Component {
  state = { hasError: false }

```

```

    static getDerivedStateFromError(error) {
      return { hasError: true };
    }

    render() {
      if (this.state.hasError) {
        return <h1>Something went wrong.</h1>; // Fallback UI
      }
      return this.props.children;
    }
  }

  const ChildComponent = () => {
    throw new Error('Oops!'); // Error thrown
    return <h1>Hello from child component</h1>;
  }

  const App = () => {
    return (
      <ErrorBoundary>
        <ChildComponent />
      </ErrorBoundary>
    );
  }

```

componentDidCatch

The `componentDidCatch` lifecycle method runs when a child component in its component tree throws an error during rendering.

The method receives two parameters, the `error` that was thrown as well as `info` about the rendering issue. It is commonly used in an Error Boundary component to log the error to some error reporting service.

```

import React, { Component } from 'react';

class ErrorBoundary extends Component {
  state = { hasError: false, errorInfo: null };

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI
    return { hasError: true };
  }

```

```

    }

    componentDidCatch(error, errorInfo) {
      // You can also log the error to an error reporting service
      console.log(error, errorInfo);
    }

    render() {
      if (this.state.hasError) {
        // You can render any custom fallback UI
        return <h1>Something went wrong.</h1>;
      }
      return this.props.children;
    }
  }

  const ChildComponent = () => {
    // In real-world applications, errors might be thrown under certain conditions
    throw new Error('Oops!'); // This is just for illustrative purposes.
    return <h1>Hello from child component</h1>;
  }

  const App = () => {
    return (
      <ErrorBoundary>
        <ChildComponent />
      </ErrorBoundary>
    );
  }

```

Remember! `getDerivedStateFromError` is called during the **render** phase, so it does not permit side effects such as logging errors. However, `componentDidCatch` is called during the commit phase, where side effects are allowed such as logging our errors.

