Credits: https://www.w3schools.com/REACT/react_class.asp

React Class Components

Before React 16.8, Class components were the only way to track state and lifecycle on a React component. Function components were considered "state-less".

With the addition of Hooks, Function components are now almost equivalent to Class components. The differences are so minor that you will probably never need to use a Class component in React.

Even though Function components are preferred, there are no current plans on removing Class components from React.

This section will give you an overview of how to use Class components in React.

Feel free to skip this section, and use Function Components instead.

React Components

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render() function.

Components come in two types, Class components and Function components, in this chapter you will learn about Class components.

Create a Class Component

When creating a React component, the component's name must start with an upper case letter.

The component has to include the `extends React.Component` statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example

Create a Class component called `Car`

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

Now your React application has a component called Car, which returns a `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

Example

Display the `Car` component in the "root" element:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Component Constructor

If there is a `constructor()` function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state` later in this tutorial.

The constructor function is also where you honor the inheritance of the parent component by including the `super()` statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (`React.Component`).

Example

Create a constructor function in the Car component, and add a color property:

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a Car!</h2>;
  }
}
```

Use the color property in the render() function:

Example

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}
```

Props

Another way of handling component properties is by using `props`.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about `props` in the next chapter.

Example

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
class Car extends React.Component {
  render() {

    return <h2>I am a {this.props.color} Car!</h2>;

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car color="red"/>);
```

Props in the Constructor

If your component has a constructor function, the props should always be passed to the constructor and also to the React.Component via the `super()` method.

Example

```
class Car extends React.Component {

  constructor(props) {

    super(props);

  }

  render() {

    return <h2>I am a {this.props.model}!</h2>;

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car model="Mustang"/>);
```

Components in Components

We can refer to components inside other components:

Example

Use the Car component inside the Garage component:

```
class Car extends React.Component {

  render() {

    return <h2>I am a Car!</h2>;

  }

}


class Garage extends React.Component {

  render() {

    return (

      <div>

      <h1>Who lives in my Garage?</h1>

      <Car />

      </div>

    );

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage />);
```

Components in Files

React is all about re-using code, and it can be smart to insert some of your components in separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Note that the file must start by importing React (as before), and it has to end with the statement `export default Car;`.

Example

This is the new file, we named it `Car.js`:

```
import React from 'react';


class Car extends React.Component {

  render() {

    return <h2>Hi, I am a Car!</h2>;

  }

}


export default Car;
```

To be able to use the `Car` component, you have to import the file in your application.

Example

Now we import the `Car.js` file in the application, and we can use the `Car` component as if it was created here.

```
import React from 'react';

import ReactDOM from 'react-dom/client';

import Car from './Car.js';


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

React Class Component State

React Class components have a built-in `state` object.

You might have noticed that we used `state` earlier in the component constructor section.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

Creating the state Object

The state object is initialized in the constructor:

Example

Specify the `state` object in the constructor method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
  this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

The state object can contain as many properties as you like:

Example

Specify all the properties your component need:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

Using the `state` Object

Refer to the `state` object anywhere in the component by using the `this.state.propertyname` syntax:

Example:

Refer to the `state` object in the `render()` method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Changing the `state` Object

To change a value in the state object, use the `this.setState()` method.

When a value in the `state` object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an `onClick` event that will change the color property:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>

      </div>

    );

  }

}
```

Always use the `setState()` method to change the state object, it will ensure that the component knows its been updated and calls the render() method (and all the other lifecycle methods).

Lifecycle of Components

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. `constructor()`
2. `getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

constructor

The `constructor()` method is called before anything else, when the component is initiated, and it is the natural place to set up the initial `state` and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:

The constructor method is called, by React, every time you make a component:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  render() {

    return (

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

    );

  }

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

getDerivedStateFromProps

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Example:

The `getDerivedStateFromProps` method is called right before the render method:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  static getDerivedStateFromProps(props, state) {

    return {favoritecolor: props.favcol };

  }

  render() {

    return (

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

    );

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header favcol="yellow"/>);
```

render

The `render()` method is required, and is the method that actually outputs the HTML to the DOM.

Example:

A simple component with a simple `render()` method:

```
class Header extends React.Component {

  render() {
    return (

      <h1>This is the content of the Header component</h1>

    );

  }

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

componentDidMount

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }

  render() {

    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's `state` or `props`.

React has five built-in methods that gets called, in this order, when a component is updated:

1. `getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`
5. `componentDidUpdate()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

Also at *updates* the `getDerivedStateFromProps` method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the `state` object based on the initial props.

The example below has a button that changes the favorite color to blue, but since the `getDerivedStateFromProps()` method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

Example:

If the component gets updated, the `getDerivedStateFromProps()` method is called:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow" />);
```

shouldComponentUpdate

In the `shouldComponentUpdate()` method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is `true`.

The example below shows what happens when the `shouldComponentUpdate()` method returns `false`:

Example:

Stop the component from rendering at any update:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  shouldComponentUpdate() {

    return false;

  }

  changeColor = () => {

    this.setState({favoritecolor: "blue"});

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <button type="button" onClick={this.changeColor}>Change color</button>

      </div>

    );

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

Example:

Same example as above, but this time the `shouldComponentUpdate()` method returns `true` instead:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  shouldComponentUpdate() {

    return true;

  }

  changeColor = () => {

    this.setState({favoritecolor: "blue"});

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <button type="button" onClick={this.changeColor}>Change color</button>

      </div>

    );

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

render

The `render()` method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

Example:

Click the button to make a change in the component's state:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  changeColor = () => {

    this.setState({favoritecolor: "blue"});

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <button type="button" onClick={this.changeColor}>Change color</button>

      </div>

    );

  }

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

getSnapshotBeforeUpdate

In the `getSnapshotBeforeUpdate()` method you have access to the `props` and `state` *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the `getSnapshotBeforeUpdate()` method is present, you should also include the `componentDidUpdate()` method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a `getSnapshotBeforeUpdate()` method, this method is executed, and writes a message to the empty DIV1 element.

Then the `componentDidUpdate()` method is executed and writes a message in the empty DIV2 element:

Example:

Use the `getSnapshotBeforeUpdate()` method to find out what the `state` object looked like before the update:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <div id="div1"></div>
        <div id="div2"></div>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

componentDidUpdate

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

Example:

The `componentDidUpdate` method is called after the update has been rendered in the DOM:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }

  componentDidUpdate() {

    document.getElementById("mydiv").innerHTML =

    "The updated favorite is " + this.state.favoritecolor;

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <div id="mydiv"></div>

      </div>

    );

  }
}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`

componentWillUnmount

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

Example:

Click the button to delete the header:

```
class Container extends React.Component {

  constructor(props) {

    super(props);

    this.state = {show: true};

  }

  delHeader = () => {

    this.setState({show: false});

  }

  render() {

    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
      {myheader}
      <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}

class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Container />);
```