JSX

Nested JSX elements

In order for the code to compile, a JSX expression must have exactly one outermost element. In the below block of code the `<a>` tag is the outermost element.

```
const myClasses = (
  <a href="https://www.codecademy.com">
    <h1>
      Sign Up!
    </h1>
  </a>
);
```

JSX Syntax and JavaScript

JSX is a syntax extension of JavaScript. It's used to create DOM elements which are then rendered in the React DOM.
A JavaScript file containing JSX will have to be compiled before it reaches a web browser. The code block shows some example JavaScript code that will need to be compiled.

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<h1>Render me!</h1>,
document.getElementById('app'));
```

Multiline JSX Expression

A JSX expression that spans multiple lines must be wrapped in parentheses: `(` and `)`. In the example code, we see the opening parentheses on the same line as the constant declaration, before the JSX expression begins. We see the closing parentheses on the line following the end of the JSX expression.

```
const myList = (
  <ul>
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
  </ul>
);
```

JSX syntax and HTML

In the block of code we see the similarities between JSX syntax and HTML: they both use the angle bracket opening and closing tags (`<h1>` and `</h1>`).
When used in a React component, JSX will be rendered as HTML in the browser.

```
const title = <h1>Welcome all!</h1>
```

JSX attributes

The syntax of JSX attributes closely resembles that of HTML attributes. In the block of code, inside of the opening tag of the `<h1>` JSX element, we see an `id` attribute with the value `"example"` .

```
const example = <h1 id="example">JSX Attributes</h1>;
```

ReactDOM JavaScript library

The JavaScript library `react-dom` , sometimes called `ReactDOM` , renders JSX elements to the DOM by taking a JSX expression, creating a corresponding tree of DOM nodes, and adding that tree to the DOM.

The code example begins with `ReactDOM.render()` . The first argument is the JSX expression to be compiled and rendered and the second argument is the HTML element we want to append it to.

```
ReactDOM.render(
  <h1>This is an example.</h1>,
  document.getElementById('app')
);
```

Embedding JavaScript in JSX

JavaScript expressions may be embedded within JSX expressions. The embedded JavaScript expression must be wrapped in curly braces.

In the provided example, we are embedding the JavaScript expression `10 * 10` within the `<h1>` tag. When this JSX expression is rendered to the DOM, the embedded JavaScript expression is evaluated and rendered as `100` as the content of the `<h1>` tag.

```
let expr = <h1>{10 * 10}</h1>;
// above will be rendered as <h1>100</h1>
```

The Virtual Dom

React uses Virtual DOM, which can be thought of as a blueprint of the DOM. When any changes are made to React elements, the Virtual DOM is updated. The Virtual DOM finds the differences between it and the DOM and re-renders only the elements in the DOM that changed. This makes the Virtual DOM faster and more efficient than updating the entire DOM.

JSX className

In JSX, you can't use the word `class`! You have to use `className` instead. This is because JSX gets translated into JavaScript, and `class` is a reserved word in JavaScript. When JSX is rendered, JSX `className` attributes are automatically rendered as `class` attributes.

```
// When rendered, this JSX expression...
const heading = <h1 className="large-
heading">Codecademy</h1>;

// ...will be rendered as this HTML
<h1 class="large-heading">Codecademy</h1>
```

JSX and conditional

In JSX, `&&` is commonly used to render an element based on a boolean condition. `&&` works best in conditionals that will sometimes do an action, but other times do nothing at all. If the expression on the left of the `&&` evaluates as true, then the JSX on the right of the `&&` will be rendered. If the first expression is false, however, then the JSX to the right of the `&&` will be ignored and not rendered.

```
// All of the list items will display if
// baby is false and age is above 25
const tasty = (
  <ul>
    <li>Applesauce</li>
    { !baby && <li>Pizza</li> }
    { age > 15 && <li>Brussels Sprouts</li> }
    { age > 20 && <li>Oysters</li> }
    { age > 25 && <li>Grappa</li> }
  </ul>
);
```

JSX conditionals

JSX does not support if/else syntax in embedded JavaScript. There are three ways to express conditionals for use with JSX elements:

1. a ternary within curly braces in JSX
2. an `if` statement outside a JSX element, or
3. the `&&` operator.

```
// Using ternary operator
const headline = (
  <h1>
    { age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff' }
  </h1>
);

// Using if/else outside of JSX
let text;

if (age >= drinkingAge) { text = 'Buy Drink' }
else { text = 'Do Teen Stuff' }

const headline = <h1>{ text }</h1>

// Using && operator. Renders as empty div if length is 0
const unreadMessages = [ 'hello?', 'remember me!'];

const update = (
  <div>
    {unreadMessages.length > 0 &&
      <h1>
        You have {unreadMessages.length} unread messages.
      </h1>
    }
  </div>
);
```

Embedding JavaScript code in JSX

Any text between JSX tags will be read as text content, not as JavaScript. In order for the text to be read as JavaScript, the code must be embedded between curly braces `{ }`.

```
<p>{ Math.random() }</p>

// Above JSX will be rendered something like this:
<p>0.88</p>
```

JSX element event listeners

In JSX, event listeners are specified as attributes on elements. An event listener attribute's *name* should be written in camelCase, such as `onClick` for an `onclick` event, and `onMouseOver` for an `onmouseover` event.

An event listener attribute's *value* should be a function. Event listener functions can be declared inline or as variables and they can optionally take one argument representing the event.

```
// Basic example
const handleClick = () => alert("Hello world!");

const button = <button onClick={handleClick}>Click
here</button>;

// Example with event parameter
const handleMouseOver = (event) => event.target.style.color
= 'purple';

const button2 = <div onMouseOver={handleMouseOver}>Drag here
to change color</div>;
```

Setting JSX attribute values with embedded JavaScript

When writing JSX, it's common to set attributes using embedded JavaScript variables.

```
const introClass = "introduction";
const introParagraph = <p className={introClass}>Hello
world</p>;
```

JSX .map() method

The array method `map()` comes up often in React. It's good to get in the habit of using it alongside JSX.

If you want to create a list of JSX elements from a given array, then `map()` over each element in the array, returning a list item for each one.

```
const strings = ['Home', 'Shop', 'About Me'];

const listItems = strings.map(string => <li>{string}</li>);

<ul>{listItems}</ul>
```

JSX empty elements syntax

In JSX, empty elements must explicitly be closed using a closing slash at the end of their tag: `<tagName />`.

A couple examples of empty element tags that must explicitly be closed include `<br>` and `<img>`.

```
<br />
<img src="example_url" />
```

`React.createElement()` Creates Virtual DOM Elements

The `React.createElement()` function is used by React to actually create virtual DOM elements from JSX. When the JSX is compiled, it is replaced by calls to `React.createElement()`.

You usually won't write this function yourself, but it's useful to know about.

```
// The following JSX...
const h1 = <h1 className="header">Hello world</h1>;

// ...will be compiled to the following:
const h1 = React.createElement(
  'h1',
  {
    className: 'header',
  },
  'Hello world'
);
```

JSX `key` attribute

In JSX elements in a list, the `key` attribute is used to uniquely identify individual elements. It is declared like any other attribute.

Keys can help performance because they allow React to keep track of whether individual list items should be rendered, or if the order of individual items is important.

```
<ul>
  <li key="key1">One</li>
  <li key="key2">Two</li>
  <li key="key3">Three</li>
  <li key="key4">Four</li>
</ul>
```

React Components

`render()` Method

React class components must have a `render()` method. This method should return some React elements created with JSX.

```
class MyComponent extends React.Component {
  render() {
    return <h1>Hello from the render method!</h1>;
  }
}
```

React Component Base Class

React class components need to inherit from the `React.Component` base class and have a `render()` method. Other than that, they follow regular JavaScript class syntax. This example shows a simple React class component.

```
class MyComponent extends React.Component {
  render() {
    return <h1>Hello world!</h1>;
  }
}
```

Importing React

In order to use React, we must first import the React library. When we import the library, it creates an object that contains properties needed to make React work, including JSX and creating custom components.

```
import React from 'react';
```

React Components

A React component is a reusable piece of code used to define the appearance, behavior, and state of a portion of a web app's interface. Components are defined as functions or as classes. Using the component as a factory, an infinite number of component instances can be created.

```
import React from 'react';

function MyFunctionComponent() {
  return <h1>Hello from a function component!</h1>;
}

class MyClassComponent extends React.Component {
  render() {
    return <h1>Hello from a class component!</h1>;
  }
}
```

JSX Capitalization

React requires that the first letter of components be capitalized. JSX will use this capitalization to tell the difference between an HTML tag and a component instance. If the first letter of a name is capitalized, then JSX knows it's a component instance; if not, then it's an HTML element.

```
// This is considered a component by React.
<ThisComponent />

// This is considered a JSX HTML tag.
<div>
```

`ReactDOM.render()`

`ReactDOM.render()` 's first argument is a component instance. It will render that component instance.

In this example, we will render an instance of `MyComponent` .

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyComponent extends React.Component {
  render() {
    return <h1>Hello world!</h1>;
  }
}

ReactDOM.render(<MyComponent />,
document.getElementById('app'));
```

Multi-line JSX Expressions

Parentheses are used when writing a multi-line JSX expression. In the example, we see that the component's `render()` method is split over multiple lines. Therefore it is wrapped in parentheses.

```
render() {
  return (
    <blockquote>
      <p>Be the change you wish to see in the world.</p>
      <cite>
        <a
          target="_blank"

href="https://en.wikipedia.org/wiki/Mahatma_Gandhi"
        >
          Mahatma Gandhi
        </a>
      </cite>
    </blockquote>
  );
}
```

Code in `render()`

A React component can contain JavaScript before any JSX is returned. The JavaScript before the `return` statement informs any logic necessary to render the component. In the example code, we see JavaScript prior to the `return` statement which rounds the `value` to an integer.

```
class Integer extends React.Component {
  render() {
    const value = 3.14;
    const asInteger = Math.round(value);
    return <p>{asInteger}</p>;
  }
}
```

Object Properties As Attribute Values

In React, JSX attribute values can be set through data stored in regular JavaScript objects. We see this in the example block of code.

In our code example we first see our JavaScript object `seaAnemones` and the values stored with this image. We then see how these stored values are used to set the `<img>` attributes in our JSX expression for the `SeaAnemones` component.

```
const seaAnemones = {
  src:

'https://commons.wikimedia.org/wiki/Category:Images#/media/Fil

  alt: 'Sea Anemones',
  width: '300px',
};

class SeaAnemones extends React.Component {
  render() {
    return (
      <div>
        <h1>Colorful Sea Anemones</h1>
        <img
          src={seaAnemones.src}
          alt={seaAnemones.alt}
          width={seaAnemones.width}
        />
      </div>
    );
  }
}
```

Components Interacting

Returning HTML Elements and Components

A class component's `render()` method can return any JSX, including a mix of HTML elements and custom React components.

In the example, we return a `<Logo />` component and a "vanilla" HTML title.

This assumes that `<Logo />` is defined elsewhere.

```
class Header extends React.Component {
  render() {
    return (
      <div>
        <Logo />
        <h1>Codecademy</h1>
      </div>
    );
  }
}
```

React Component File Organization

It is common to keep each React component in its own file, `export` it, and `import` it wherever else it is needed. This file organization helps make components reusable. You don't need to do this, but it's a useful convention.

In the example, we might have two files: **App.js**, which is the top-level component for our app, and **Clock.js**, a sub-component.

```
// Clock.js
import React from 'react';

export class Clock extends React.Component {
  render() {
    // ...
  }
}

// App.js
import React from 'react';
import { Clock } from './Clock';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>What time is it?</h1>
        <Clock />
      </div>
    );
  }
}
```

`this.props`

React class components can access their props with the `this.props` object.

In the example code below, we see the `<Hello>` component being rendered with a `firstName` prop. It is accessed in the component's `render()` method with `this.props.firstName`.

This should render the text "Hi there, Kim!"

```
class Hello extends React.Component {
  render() {
    return <h1>Hi there, {this.props.firstName}!</h1>;
  }
}

ReactDOM.render(<Hello firstName="Kim" />,
document.getElementById('app'));
```

`defaultProps`

A React component's `defaultProps` object contains default values to be used in case props are not passed. If a prop is not passed to a component, then it will be replaced with the value in the `defaultProps` object.

In the example code, `defaultProps` is set so that profiles have a fallback profile picture if none is set. The `<MyFriends>` component should render two profiles: one with a set profile picture and one with the fallback profile picture.

```
class Profile extends React.Component {
  render() {
    return (
      <div>
        <img src={this.props.profilePictureSrc} alt="" />
        <h2>{this.props.name}</h2>
      </div>
    );
  }
}

Profile.defaultProps = {
  profilePictureSrc: 'https://example.com/no-profile-
picture.jpg',
};

class MyFriends extends React.Component {
  render() {
    return (
      <div>
        <h1>My friends</h1>
        <Profile
          name="Jane Doe"
          profilePictureSrc="https://example.com/jane-
doe.jpg"
        />
        <Profile name="John Smith" />
      </div>
    );
  }
}
```

`props`

Components can pass information to other components. When one component passes information to another, it is passed as *props* through one or more *attributes*.

The example code demonstrates the use of attributes in props. `SpaceShip` is the component and `ride` is the attribute. The `SpaceShip` component will receive `ride` in its props.

```
<SpaceShip ride="Millennium Falcon" />
```

this.props.children

Every component's `props` object has a property named `children`. Using `this.props.children` will return everything in between a component's opening and closing JSX tags.

```
<List>  // opening tag
  <li></li> // child 1
  <li></li> // child 2
  <li></li> // child 3
</List> // closing tag
```

Binding `this` keyword

In React class components, it is common to pass event handler functions to elements in the `render()` method. If those methods update the component state, `this` must be bound so that those methods correctly update the overall component state.

In the example code, we bind `this.changeName()` so that our event handler works.

```
class MyName extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'Jane Doe' };
    this.changeName = this.changeName.bind(this);
  }

  changeName(newName) {
    this.setState({ name: newName });
  }

  render() {
    return (
      <h1>My name is {this.state.name}</h1>
      <NameChanger handleChange={this.changeName} />
    )
  }
}
```

Call `super()` in the Constructor

React class components should call `super(props)` in their constructors in order to properly set up their `this.props` object.

```
// WRONG!
class BadComponent extends React.Component {
  constructor() {
    this.state = { favoriteColor: 'green' };
  }
  // ...
}

// RIGHT!
class GoodComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoriteColor: 'green' };
  }
  // ...
}
```

`this.setState()`

React class components can change their state with `this.setState()`.

`this.setState()` should always be used instead of directly modifying the `this.state` object.

`this.setState()` takes an object which it merges with the component's current state. If there are properties in the current state that aren't part of that object, then those properties are unchanged.

In the example code, we see `this.setState()` used to update the `Flavor` component's state from `'chocolate'` to `'vanilla'`.

```
class Flavor extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      favorite: 'chocolate',
    };
  }

  render() {
    return (
      <button
        onClick={(event) => {
          event.preventDefault();
          this.setState({ favorite: 'vanilla' });
        }}
      >
        No, my favorite is vanilla
      </button>
    );
  }
}
```

Dynamic Data in Components

React components can receive dynamic information from *props*, or set their own dynamic data with *state*. Props are passed down by parent components, whereas state is created and maintained by the component itself.

In the example, you can see `this.state` set up in the constructor, used in `render()`, and updated with `this.setState()`. `this.props` refers to the props, which you can see in the `render()` method.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { showPassword: false };
  }

  render() {
    let text;
    if (this.state.showPassword) {
      text = `The password is ${this.props.password}`;
    } else {
      text = 'The password is a secret';
    }

    return (
      <div>
        <p>{text}</p>
        <button
          onClick={(event) => {
            event.preventDefault();
            this.setState((oldState) => ({
              showPassword: !oldState.showPassword,
            }));
          }}
        >
          Toggle password
        </button>
      </div>
    );
  }
}
```

Component State in Constructor

React class components store their state as a JavaScript object. This object is initialized in the component's `constructor()`.

In the example, the component stores its state in `this.state`.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      favoriteColor: 'green',
      favoriteMusic: 'Bluegrass',
    };
  }

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

Don't Change State While Rendering

When you update a React component's state, it will automatically re-render. That means you should never update the state in a render function because it will cause an infinite loop.

In the example, we show some bad code that calls `this.setState()` inside of its `render()` method.

```
class BadComponent extends React.Component {
  constructor(props) {
    super(props);
    this.count = 0;
  }
  render() {
    // Don't do this! This is bad!
    this.setState({ count: this.state.count + 1 });
    return <div>The count is {this.state.count}</div>;
  }
}
```

Lifecycle Methods

Component Mount

A React component *mounts* when it renders to the DOM for the first time. If it's already mounted, a component can be rendered again if it needs to change its appearance or content.

Unmounting Lifecycle Method

React supports one unmounting lifecycle method, `componentWillUnmount`, which will be called right before a component is removed from the DOM.

`componentWillUnmount()` is used to do any necessary cleanup (canceling any timers or intervals, for example) before the component disappears.

Note that the `this.setState()` method should not be called inside `componentWillUnmount()` because the component will not be re-rendered.

```
componentWillUnmount(prevProps, prevState) {
   clearInterval(this.interval);
}
```

Component Mounting Phase

A component "mounts" when it renders for the first time. When a component mounts, it automatically calls these three methods, in the order of:

1. `constructor()`
2. `render()`
3. `componentDidUpdate()`

Hooks

Function Components

In React, you can use a function as a component instead of a class. Function components receive `props` as a parameter.
In the example code, we show two equivalent components: one as a class and one as a function.

```
// The two components below are equivalent.
class GreeterAsClass extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

function GreeterAsFunction(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Why Hooks?

Hooks are functions that let us “hook into” state and lifecycle functionality in function components.
Hooks allow us to:

- reuse stateful logic between components
- simplify and organize our code to separate concerns, rather allowing unrelated data to get tangled up together
- avoid confusion around the behavior of the `this` keyword
- avoid class constructors, binding methods, and related advanced JavaScript techniques

Rules for Using Hooks

There are two main rules to keep in mind when using Hooks:

1. Only call Hooks from React functions
2. Only call Hooks at the top level, to be sure that Hooks are called in the same order each time a component renders.

Common mistakes to avoid are calling Hooks inside of loops, conditions, or nested functions.

```
// Instead of confusing React with code like this:
if (userName !== '') {
 useEffect(() => {
   localStorage.setItem('savedUserName', userName);
 });
}

// We can accomplish the same goal, while consistently
calling our Hook every time:
useEffect(() => {
 if (userName !== '') {
   localStorage.setItem('savedUserName', userName);
 }
});
```

The State Hook

The `useState()` Hook lets you add React state to function components. It should be called at the top level of a React function definition to manage its state.

`initialState` is an optional value that can be used to set the value of `currentState` for the first render. The `stateSetter` function is used to update the value of `currentState` and rerender our component with the next state value.

```
const [currentState, stateSetter] = useState(initialState);
```

State Setter Callback Function

When the previous state value is used to calculate the next state value, pass a function to the state setter. This function accepts the previous value as an argument and returns an updated value.
If the previous state is not used to compute the next state, just pass the next state value as the argument for the state setter.

```
function Counter({ initialCount }) {
 const [count, setCount] = useState(initialCount);
 return (
   <div>
     Count: {count}
     <button onClick={() =>
setCount(initialCount)}>Reset</button>
     <button onClick={() => setCount((prevCount) =>
prevCount - 1)}>-</button>
     <button onClick={() => setCount((prevCount) =>
prevCount + 1)}>+</button>
   </div>
 );
}
```

Multiple State Hooks

`useState()` may be called more than once in a component. This gives us the freedom to separate concerns, simplify our state setter logic, and organize our code in whatever way makes the most sense to us!

```
function App() {
 const [sport, setSport] = useState('basketball');
 const [points, setPoints] = useState(31);
 const [hobbies, setHobbies] = useState([]);
}
```

Side Effects

The primary purpose of a React component is to return some JSX to be rendered. Often, it is helpful for a component to execute some code that performs side effects in addition to rendering JSX.
In class components, side effects are managed with lifecycle methods. In function components, we manage side effects with the Effect Hook. Some common side effects include: fetching data from a server, subscribing to a data stream, logging values to the console, interval timers, and directly interacting with the DOM.

The Effect Hook

After importing `useEffect()` from the `'react'` library, we call this Hook at the top level of a React function definition to perform a side effect. The callback function that we pass as the first argument of `useEffect()` is where we write whatever JavaScript code that we'd like React to call after each render.

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

function TitleCount() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return <button onClick={(prev) => setCount(prev + 1)}>+
</button>;
}
```

Effect Cleanup Functions

The cleanup function is optionally returned by the first argument of the Effect Hook. If the effect does anything that needs to be cleaned up to prevent memory leaks, then the effect returns a cleanup function. The Effect Hook will call this cleanup function before calling the effect again as well as when the component is being unmounted from the DOM.

```
useEffect(() => {
  document.addEventListener('keydown', handleKeydown);
  return () => document.removeEventListener('keydown',
handleKeydown);
});
```

Multiple Effect Hooks

`useEffect()` may be called more than once in a component. This gives us the freedom to individually configure our dependency arrays, separate concerns, and organize our code in whatever way makes the most sense to us!

```
function App(props) {
 const [title, setTitle] = useState('');
 useEffect(() => {
   document.title = title;
 }, [title]);

 const [time, setTime] = useState(0);
 useEffect(() => {
   const intervalId = setInterval(() => setTime((prev) =>
prev + 1), 1000);
   return () => clearInterval(intervalId);
 }, []);

 // ...
}
```

Effect Dependency Array

The dependency array is used to tell the `useEffect()` method when to call our effect. By default, with no dependency array provided, our effect is called after every render. An empty dependency array signals that our effect never needs to be re-run. A non-empty dependency array signals to the Effect Hook that it only needs to call our effect again when the value of one of the listed dependencies has changed.

```
useEffect(() => {
 alert('called after every render');
});

useEffect(() => {
 alert('called after first render');
}, []);

useEffect(() => {
 alert('called when value of `endpoint` or `id` changes');
}, [endpoint, id]);
```

Lifecycle Phases

There are three categories of lifecycle methods: mounting, updating, and unmounting.
A component "mounts" when it renders for the first time. This is when mounting lifecycle methods get called.
The first time that a component instance renders, it does not update. Starting with the second render, a component updates every time that it renders.
A component's unmounting period occurs when the component is removed from the DOM. This could happen if the DOM is rerendered without the component, or if the user navigates to a different website or closes their web browser.

Mounting Lifecycle Methods

React supports three mounting lifecycle methods for component classes: `componentWillMount()`, `render()`, and `componentDidMount()`. `componentWillMount()` will be called first followed by the `render()` method and finally the `componentDidMount()` method.

Updating Lifecycle Method

When a component updates, `shouldComponentUpdate()` gets called after `componentWillReceiveProps()`, but still before the rendering begins. It automatically receives two arguments: `nextProps` and `nextState`. `shouldComponentUpdate()` should return either true or false. The best way to use this method is to have it return false *only under certain conditions*. If those conditions are met, then your component will not update.

```
shouldComponentUpdate(nextProps, nextState) {
  if ((this.props.text == nextProps.text) &&
    (this.state.subtext == nextState.subtext)) {
    return false;
  } else {
    return true;
  }
}
```

Stateless Components From Stateful Components

Stateful and Stateless Components

In React, a *stateful* component is a component that holds some state. *Stateless* components, by contrast, have no state. Note that both types of components can use props.

In the example, there are two React components. The `Store` component is stateful and the `Week` component is stateless.

```
class Store extends React.Component {
  constructor(props) {
    super(props);
    this.state = { sell: 'anything' };
  }
  render() {
    return <h1>I'm selling {this.state.sell}.</h1>;
  }
}

class Week extends React.Component {
  render() {
    return <h1>Today is {this.props.day}!</h1>;
  }
}
```

React Programming Pattern

One of the most common programming patterns in React is to use stateful parent components to maintain their own state and pass it down to one or more stateless child components as props. The example code shows a basic example.

```
// This is a stateless child component.
class BabyYoda extends React.Component {
  render() {
    return <h2>I am {this.props.name}!</h2>;
  }
}

// This is a stateful Parent element.
class Yoda extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'Toyoda' };
  }

  // The child component will render information passed down
from the parent component.
  render() {
    return <BabyYoda name={this.state.name} />;
  }
}
```

Changing Props and State

In React, a component should never change its own props directly. A parent component should change them.

State, on the other hand, is the opposite of props: a component keeps track of its own state and can change it at any time.

The example code shows a component that accepts a prop, `subtitle`, which never changes. It also has a state object which does change.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { now: new Date() };
    this.updateTime = this.updateTime.bind(this);
  }

  updateTime() {
    this.setState({ now: new Date() });
  }

  render() {
    return (
      <div>
        <h1>It is currently
{this.state.now.toLocaleTimeString()}</h1>
        <h2>{this.props.subtitle}</h2>
        <button onClick={this.updateTime}>Update the
clock</button>
      </div>
    );
  }
}
```

Passing State Change Functions as Props

If a React parent component defines a function that changes its state, that function can be passed to a child component and called within the component to updating the parent component's state.

In this example, because `this.setState()` causes the `Name` component to re-render, any change to the `<input>` will update the `Name` component's state, causing a new render and displaying the new state value to the `<p>` tag content.

```
class Name extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: '' };
    this.handleNameChange =
this.handleNameChange.bind(this);
  }

  handleNameChange(e) {
    this.setState({
      name: e.target.value,
    });
  }

  render() {
    return (
      <div>
        <input onChange={this.handleNameChange} />
        <p>{this.state.name}</p>
      </div>
    );
  }
}
```

Event Handlers and State in React

Event handler functions in React are often used to update state. These handler functions often receive an event as an argument, which is used to update state values correctly.

In the example code, we use `event.target.value` to get the input's value.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { text: '' };
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    this.setState({
      name: event.target.value,
    });
  }

  render() {
    return (
      <div>
        <input onChange={this.handleChange} value=
{this.state.text} />
        <p>You typed {this.state.text}</p>
      </div>
    );
  }
}
```

Using Stateless Updaters and Presenters

A common React programming pattern is to use a parent stateful component to manage state and define state-updating methods. Then, it will render stateless child components. One or more of those child components will be responsible for updating the parent state (via methods passed as props). One or more of those child components will be responsible for displaying that state.

In the example code, `StatefulParent` renders `<InputComponent>` to change its state and uses `<DisplayComponent>` to display it.

```
class StatefulParent extends React.Component {
  constructor(props) {
    super(props);
    // Set up initial state here
    // Bind handler functions here
  }

  handlerMethod(event) {
    // Update state here
  }

  render() {
    return (
      <div>
        <InputComponent onChange={handler} />
        <DisplayComponent valueToDisplay=
{this.state.valueToDisplay} />
      </div>
    );
  }
}
```

Advanced React

React CSS Styles

React supports inline CSS styles for elements. Styles are supplied as a `style` prop with a JavaScript object.

```
// Passing the styles as an object
const color = {
  color: 'blue',
  background: 'sky'
};
<h1 style={color}>Hello</h1>

// Passing the styles with an inline object, as a shorthand
<h1 style={{ color: 'red' }}>I am red!</h1>
```

Style Names And Values

In React, style names are written in “camelCase”, unlike in CSS where they are hyphenated. In most cases, style values are written as strings. When entering numeric values, you don’t have to enter `px` because React automatically interprets them as pixel values.

```
// Styles in CSS:
// font-size: 20px;
// color: blue;

// Would look like this style object in React:
const style = {
  fontSize: 20,
  color: 'blue',
};
```

Presentational and Container Components

A common programming pattern in React is to have presentational and container components. Container components contain business logic (methods) and handle state. Presentational components render that behavior and state to the user.

In the example code, `CounterContainer` is a container component and `Counter` is a presentational component.

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

  increment() {
    this.setState((oldState) => {
      return { count: oldState.count + 1 };
    });
  }

  render() {
    return <Counter count={this.state.count} increment=
{this.increment} />;
  }
}

class Counter extends React.Component {
  render() {
    return (
      <div>
        <p>The count is {this.props.count}.</p>
        <button onClick={this.props.increment}>Add
1</button>
      </div>
    );
  }
}
```

Static Property

In React, prop types are set as a static property (`.propTypes`) on a component class or a function component. `.propTypes` is an object with property names matching the expected props and values matching the expected value of that prop type. The code snippet above demonstrates how `.propTypes` can be applied.

```
class Birth extends React.Component {
  render() {
    return <h1>{this.props.age}</h1>
  }
}

Birth.propTypes = {
  age: PropTypes.number
}
```

`.isRequired`

To indicate that a prop is required by the component, the property `.isRequired` can be chained to prop types. Doing this will display a warning in the console if the prop is not passed. The code snippet above demonstrates the use of `.isRequired` .

```
MyComponent.propTypes = {
  year: PropTypes.number.isRequired
};
```

Type Checking

In React, the `.propTypes` property can be used to perform type-checking on props. This gives developers the ability to set rules on what data/variable type each component prop should be and to display warnings when a component receives invalid type props.
In order to use `.propTypes` , you'll first need to import the `prop-types` library.

```
import PropTypes from 'prop-types';
```

Controlled vs. Uncontrolled Form Fields

In React, form fields are considered either *uncontrolled,* meaning they maintain their own state, or *controlled,* meaning that some parent maintains their state and passes it to them to display. Usually, the form fields will be controlled.
The example code shows an uncontrolled and controlled input.

```
const uncontrolledInput = <input />;

const controlledInput = (
  <input value={this.state.value} onChange=
{this.handleInputChange} />
);
```

Controlled Components

A controlled form element in React is built with a change handler function and a `value` attribute.

```
const controlledInput = (
  <input value={this.state.value} onChange=
{this.handleInputChange} />
);
```