Single page application:

The term “single-page application” (or SPA) is usually used to describe applications that were built for the web. These applications are accessed via a web browser like other websites, but offer more dynamic interactions resembling native mobile and desktop apps.

The most notable difference between a regular website and a SPA **is the reduced amount of page refreshes**. SPAs have a heavier usage of **AJAX — a way to communicate with back-end servers without doing a full page refresh — to get data loaded into our application**. As a result, the process of rendering pages happens mostly on the client-side.

Single-page App Cons

While building SPAs is trendy and considered a modern development practice, it’s important to be aware of its cons, including:

- The browser does most of the heavy lifting, which means performance can be a problem — especially on less capable mobile devices.
- Careful thought must be put into search engine optimization (SEO) so your content can be discoverable by search engines and social media websites that provide a link preview.

AJAX:

AJAX is a technique for creating fast and dynamic web pages. AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page without reloading the whole page.

What is React

ReactJS basically is an open-source **JavaScript library** which is used for building user interfaces specifically for single page applications. It’s used for **handling view layer for web and mobile apps**. React also allows us to create reusable UI components.

React allows developers to create large web applications which can change data, without reloading the page. The main purpose of React is to be fast, scalable, and simple. It works only on user interfaces in application. This corresponds to view in the MVC template. It can be used with a combination of other JavaScript libraries or frameworks, such as Angular JS in MVC.

What are the ReactJS Features?

Let us take a closer look at some important features of React.

- **JSX**

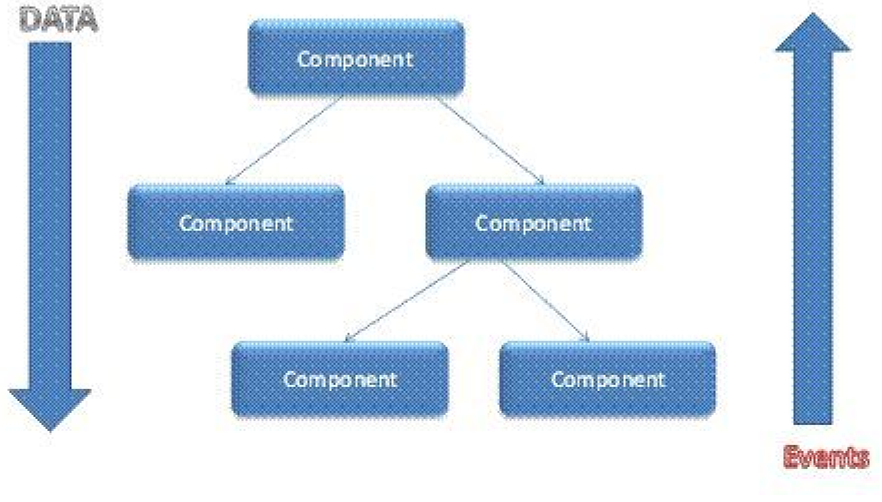
  In React, instead of using regular JavaScript for templating, it uses JSX. JSX is simple JavaScript which allows HTML quoting and uses these HTML tag syntax to render subcomponents. HTML syntax is processed into JavaScript calls of React Framework. We can also write in pure old JavaScript.

- **React Native**

  React has native libraries which were announced by Facebook in 2015, which provides the react architecture to native applications like IOS, Android and UPD.

- **Single-Way data flow**

  In React, a set of immutable values are passed to the components renderer as properties in its HTML tags. Component cannot directly modify any properties but can pass a callback function with the help of which we can do modifications. This complete process is known as “properties flow down; actions flow up”.



- **Virtual Document Object Model**

  React creates an **in-memory data structure cache** which computes the changes made and then updates the browser. This allows a special feature which enable programmer to code as if whole page is render on each change where as react library only render components which actually change.

Why React Js?

Now, the main question arises in front of us is why one should use ReactJS. There are so many open-source platforms for making the front-end web application development easier, like Angular. Let us take a quick look on the benefits of React over other competitive technologies or frameworks. With the front-end world changing on a daily basis, it’s hard to devote time to learning a new framework – especially when that framework could ultimately become a dead end. So, if you're looking for the next best thing but you're feeling a little bit lost in the framework jungle, I suggest checking out React.

- **Simplicity**

  ReactJS is just simpler to grasp right away. The component-based approach, well-defined lifecycle, and use of just plain JavaScript make React very simple to learn, build a professional web (and mobile applications), and support it. React uses a special syntax called JSX which allows you to mix HTML with JavaScript. This is not a requirement; Developer can still write in plain JavaScript but JSX is much easier to use.

- **Easy to learn**

  Anyone with a basic previous knowledge in programming can easily understand React while Angular and Ember are referred to as ‘Domain specific Language’, implying that it is difficult to learn them. For react you just need basic knowledge of CSS and HTML.

- **Native Approach**

  React can be used to create mobile applications (React Native). And React is a diehard fan of reusability, meaning extensive code reusability is supported. So at the same time we can make IOS, Android and Web application.

- **Data Binding**

  React uses one-way data binding and an application architecture called Flux controls the flow of data to components through one control point – the dispatcher. It's easier to debug self-contained components of large ReactJS apps.

React set up:

Installing node & npm

Getting React up and running is not as simple as downloading one large piece of software. You will need to install many, smaller software packages.

The first thing to install is a good installer! You need a way to download and install software packages easily, without having to worry about dependencies.

In other words, you need a good package manager. We'll be using a popular package manager named npm. npm is a great way to download, install, and keep track of JavaScript software.

You can install npm by installing Node.js. Node.js is an environment for developing server-side applications. When you install Node.js, npm will install automatically.

How npm is different

First, npm makes installation extremely easy. Installing software over and over sounds like a headache, but it really isn't. We'll walk through how soon!

Second, npm modules ("modules" are another name for software that you download via npm) are usually small. There are countless modules for different specific purposes. Instead of starting your app with a giant framework that includes tons of code you don't need, you can install only modules that you will actually use! This helps keep your code quick, easy to navigate, and not vulnerable to dependencies that you don't understand.

npm init

Alright, let's make a React app on your home computer! Where do you start?

To begin, decide where you want to save your app, and what you want to name it. In the terminal, `cd` to wherever you want to save your app. Use `mkdir` to make a new directory with your app's name. `cd` into your new directory.

Once you've done all that, type this command into your terminal:

```
npm init
```

You will get a lot of prompts! You can answer them, but it's also safe to just keep hitting `return` and not worry about it.

The command `npm init` automatically creates a new file named `package.json`. `package.json` contains metadata about your new project. `package.json` keeps track of the modules that you install. Other developers can look at your `package.json` file, easily install the same modules that you've installed, and run their own local versions of your project! This is fantastic for collaborating.

Install React

Alright! You've made a project folder, navigated into it, and used `npm init` to create a `package.json` file. Now you're ready to install some modules!

To install the `react` module, type this command in the terminal:

```
npm install --save react
```

Install ReactDOM

If you look at `package.json`, you can see that there's an object named `dependencies` that now has `react` listed as a dependency. This indicates that your project is "dependent" on having react installed. If someone tries to run your project, it probably won't work unless they install react first.

You can also see something else new in your directory: a folder named `node_modules`.

`node_modules` is where `npm` modules are saved. If you open `node_modules`, you should see a folder named `react`, which contains the code that makes React run.

To install `react-dom`, type one of these two commands in the terminal:

```
npm install --save react-dom
```

Once you install `react-dom`, you will be able to access it in your files with the code `var ReactDOM = require('react-dom').`

Install Babel

Before React code can run in the browser, it must be changed in certain ways. One necessary transformation is compiling JSX into vanilla JavaScript.

Babel is a JavaScript compiler that includes the ability **to compile JSX into regular JavaScript**. Babel can also do many other powerful things. It's worth exploring outside of the context of this course!

Babel's npm module's name is babel-core. You're going to install `babel-core` slightly differently than you installed `react` and `react-dom`. Instead of `npm install --save babel-core`, you will use the command `npm install --save-dev babel-core`.

This is because you will only be using `Babel` in *development mode*. When a React app is shipped into production, it no longer needs to make transformations: the transformations will be hard-coded in place. The `--save-dev` flag saves an npm module for development version only.

Just as `--save` can be shortened to `-S`, `--save-dev` can be shortened to `-D`.

You're also going to install two other babel-related modules, named `babel-loader` and `babel-preset-react`, respectively. We'll explain those soon!

Use one of these terminal commands to install babel-core, babel-loader, and babel-preset-react:

```
npm install --save-dev babel-core babel-loader babel-preset-react
```

Configure Babel

Babel can be configured in many ways.

1. .babelrc
2. .babelrc.js
3. babel.config.js
4. package.json

Ref : https://babeljs.io/docs/en/configuration

In order to make Babel work, you need to write a babel *configuration file*.

In your root directory, create a new file named **.babelrc**. If you get prompted about starting a filename with a period, go ahead and say that it's okay.

Save the following code inside of **.babelrc**:

```
{ presets : [ ‘react’ ] }
```

You've installed Babel, but you haven't "plugged it in" to your React app yet. You need to set up a system in which your React app will automatically run through Babel and compile your JSX, before reaching the browser.

Also, JSX to JavaScript is just one of many transformations that will need to happen to your React code. You need to set up a "transformation manager" that will take your code and run it through all of the transformations that you need, in the right order. How do you make that happen?

There are a lot of different software packages that can make this happen. The most popular as of this writing is a program called `webpack`.

Install webpack

webpack is a module that can be installed with npm, just like react and react-dom. You'll also be installing two webpack-related modules named webpack-dev-server and html-webpack-plugin, respectively. We'll explain these a little more soon.

webpack should be saved in development mode, just like babel.

Install webpack, webpack-dev-server, and html-webpack-plugin with one of these two terminal commands:

```
npm install --save-dev webpack-dev-server html-webpack-plugin
```

Webpack.config.js

Alright! Webpack has been installed!

Webpack's job is to run your React code through various *transformations*. Webpack needs to know exactly what transformations it should use!

You can set that information by making a special webpack configuration file. This file must be located in the outermost layer of your root directory, and must be named webpack.config.js. It is where you will put all of the details required to make webpack operate.

In your root directory, create a new file named webpack.config.js.

Configure webpack

https://webpack.js.org/concepts/

Ref : https://www.codecademy.com/articles/react-setup-i

“Basic Component”— Stateful component with lifecycle

This is your traditional React Component. Specifically, you need a component that has a state and/or lifecycle.

```
import React, { Component } from 'react'

class StatefulComponent extends Component {
  state = {
    visible: true,
  }

  componentDidMount() {
    this.showComponent()
  }

  showComponent() {
    setTimeout(() => {
      this.hideComponent()
    }, 4000)
  }
  hideComponent() {
    this.setState({
      visible: false,
    })
  }

  render() {
    const styles = { display: this.state.visible ? 'block' : 'none' }
    return <div style={styles}> I will hide in 4 seconds. </div>
  }
}

export default StatefulComponent
```

Stateless / Functional component

A stateless component has no state, it means that you can’t reach `this.state` inside it. It also has no lifecycle.

```
import React from 'react';

function App() {

  const greeting = 'Hello functional component!';

  return <Headline value={greeting} />;

}

function Headline(props) {

  return <h1>{props.value}</h1>;

}

export default App;
```

Since props are always coming as object, and most often you need to extract the information from the props anyway, JavaScript object destructuring comes in handy. You can directly use it in the function signature for the props object

```
import React from 'react';

function App() {

  const greeting = 'Hello functional component!';

  return <Headline value={greeting} />;

}

function Headline({ value }) {

  return <h1>{value}</h1>;

}

export default App;
```

Functional component (Using ES6 Arrow function)

```
import React from 'react';

const App = () => {

  const greeting = 'Hello Functional Component!';

  return <Headline value={greeting} />;

};

const Headline = ({ value }) => <h1>{value}</h1>;

export default App;
```

Functional vs State component

A functional component has no state, no lifecycle methods and it’s easy to write (plain function), a class component has state, lifecycle methods and React creates an instance of a class component every time React renders it.

Inline styling & commenting

React recommends using inline styles. When we want to set inline styles, we need to use camelCase syntax. React will also automatically append px after the number value on specific elements. The following example shows how to add myStyle inline to h1 element.

```
import React from 'react';

class App extends React.Component {
  render() {
    var myStyle = {
      fontSize: 100,
      color: '#FF0000'
    }
    return (
      <div>
        <h1 style = {myStyle}> Header </h1>

        { //End of the line Comment...}
        { /*Multi line comment...*/ }

      </div>
    );
  }
}
export default App;
```


React State without a Constructor

```
import React, { Component } from 'react';
const list = ['a', 'b', 'c'];

class App extends Component {
  state = {
    toggle: true,
  };

  onToggleList = () => {
    this.setState(prevState => ({
      toggle: !prevState.toggle,
    }));
  }

  render() {
    return (
      <div>
        <Toggle
          toggle={this.state.toggle}
          onToggleList={this.onToggleList}
        />
        {this.state.toggle && <List list={list} />}
      </div>
    );
  }
}
```

```
const Toggle = ({ toggle, onToggleList }) => (
  <button type="button" onClick={onToggleList}>
    {toggle ? 'Hide' : 'Show'}
  </button>
);

const List = ({ list }) => (
  <ul>
    {list.map((item) => (
      <Item key={item} item={item} />
    ))}
  </ul>
);
const Item = ({ item }) => <li>{item}</li>;
export default App;
```

Using State correctly

There are three things you should know about setState().

- **Do Not Modify State Directly**

  For example, this will not re-render a component:

  ```
  // Wrong
  this.state.comment = 'Hello';
  ```

  Instead, use setState():

  ```
  // Correct
  this.setState({comment: 'Hello'});
  ```

- **State Updates are Merged**

  When you call setState(), React merges the object you provide into the current state**.** For example, your state may contain several independent variables:

```
constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

  Then you can update them independently with separate setState() calls:

```
componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
    });
    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

  The merging is shallow, so this.setState({comments}) leaves this.state.posts, but completely replaces this.state.comments.

- **State Updates may be asynchronous**

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

We used an arrow function above, but it also works with regular functions:

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

How State Updates Are Merged in React

setState( ) is a method that is available to all React components. Invoking it lets React know that the component state has been changed. We can use it by passing in a new version of the state via an object literal.

```
class App extends React.Component {
    constructor () {
    super()
    this.state = {
      name: ‘Bob’,
      isLoggedIn: ‘false’
    }
}
handleLogIn = () => {
    this.setState({isLoggedIn : true })
}
```

It is important to note here that React merges the object you provide within the setState( ) method into the current state object you are working with. In other words, only the “isLoggedIn” is updated to be “true”, while the remaining variable “name” (with value ‘Bob’) is left intact.

We just have to supply the setState( ) method with the key/value pair(s) for which we would like to update the value. All thanks to shallow merging.

At this point the newly updated state object will look something like this:

```
state = {
   name: ‘Bob’,
   isLoggedIn: true
}
```

Shallow merging only merges things on the first level though (hence the term shallow), which means that we have to be careful when we use setState( ) on state objects with nested structures.

```
 this.state = {
        name: ‘Bob’,
        isLoggedIn: ‘false’,
        address : {
              street: ‘123 Flatiron way’,
              city: null
         }
     }
```

Following the logic of shallow merging, you might try to update the value of “address.city” for the state object illustrated above with the method setState( ) like this…

```
this.setState({
      address: {
           city: ‘New York City’
      }
})
```

However, this will change the state object’s structure to look something like:

```
{
        name: ‘Bob’,
        isLoggedIn: ‘false’,
        address : {
              city: ‘New York City’
         }
 }
```

As you can see, we lost “address.street” within the state object because the nested object (value of the key “address”) got overwritten by a new object with a single key of “city”. The logic of shallow merging does not apply for nested objects.

How do we update the value for “address.city” without totally overwriting the nested object (and thus saving the “address.street” information)? In other words, is there a way to deep merge? There are multiple ways to tackle this but the spread operator provides us a feasible solution.

```
this.setState({
    address: {
        ...this.state.address,
        city: ‘New York City’
    }
})
```

With this implementation of setState( ) the updated state object will look like (with the “address.street” information still intact):

```
{
      name: ‘Bob’,
      isLoggedIn: ‘false’,
      address : {
           Street : ‘123 Flatiron way’,
           city: ‘New York City’
       }
 }
```

To recap, React.Component’s setState() uses shallow merging to update state, ultimately saving us from listing out all of the keys and values from the initial state object. We can’t, however, apply the same logic of shallow merging on nested state objects. For these kinds of object structures, we can make use of the handy spread operator.

Component life cycle

Clock example

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


class Clock extends Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }


  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }


  tick() {
    this.setState({
      date: new Date()
    });
  }


  render() {
    return (
      <div>
      <h1>Hello, world!</h1>
      <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
    </div>
  );
 }
}

export default Clock;
```

Now the clock ticks every second. Let’s quickly recap what’s going on and the order in which the methods are called:

1. When <Clock /> is passed to ReactDOM.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.statewith an object including the current time. We will later update this state.
2. React then calls the Clock component’s render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock’s render output.
3. When the Clock output is inserted in the DOM, React calls the componentDidMount()lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component’s tick() method once a second.
4. Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls the render()method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.

Passing data from parent to child component:

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated from the child component...'})
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp = {this.updateState}></Content>
      </div>
    );
  }
}
class Content extends React.Component {
  render() {
    return (
      <div>
        <button onClick = {this.props.updateStateProp}>CLICK</button>
        <h3>{this.props.myDataProp}</h3>
      </div>
    );
  }
}
export default App;
```

Props:

The main difference between state and props is that **props are immutable**. This is why the container component should define the state that can be updated and changed, while the child components should only pass data from the state using props.

Default props:

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.headerProp}</h1>
        <h2>{this.props.contentProp}</h2>
      </div>
    );
  }
}
App.defaultProps = {
  headerProp: "Header from props...",
  contentProp:"Content from props..."
}
export default App;
```

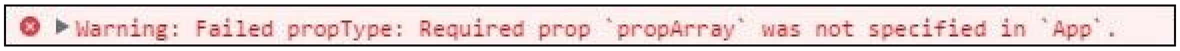
Props validation:

Properties validation is a useful way to force the correct usage of the components. This will help during development to avoid future bugs and problems, once the app becomes larger. It also makes the code more readable, since we can see how each component should be used.

In this example, we are creating App component with all the props that we need. App.propTypes is used for props validation. If some of the props aren't using the correct type that we assigned, we will get a console warning.

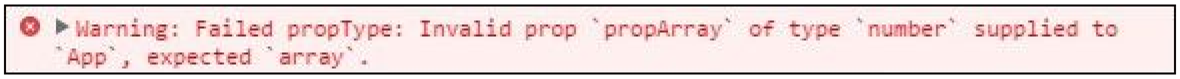
```
import React from 'react';
import ReactDOM from 'react-dom';
import PropTypes from 'prop-types';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1> Hello, {this.props.name} </h1>
            <h3>Array: {this.props.propArray}</h3>
            <h3>Bool: {this.props.propBool ? "True..." : "False..."}</h3>
            <h3>Func: {this.props.propFunc(3)}</h3>
            <h3>Number: {this.props.propNumber}</h3>
            <h3>String: {this.props.propString}</h3>
         </div>
      );
   }
}

App.defaultProps = {
   name: 'Tutorialspoint.com',
   propArray: [1, 2, 3, 4, 5],
   propBool: true,
   propFunc: function(e) {
      return e
   },
   propNumber: 1,
   propString: "String value..."
}

App.propTypes = {
   name: PropTypes.string,
   propArray: PropTypes.array.isRequired,
   propBool: PropTypes.bool.isRequired,
   propFunc: PropTypes.func,
   propNumber: PropTypes.number,
   propString: PropTypes.string,
};

export default App;
```

As can be noticed, we have use isRequired when validating propArray and propBool. This will give us an error, if one of those two don't exist. If we delete propArray:[1,2,3,4,5] from the App.defaultProps object, the console will log a warning.



```
Warning: Failed propType: Required prop `propArray` was not specified in `App`.
```

If we set the value of propArray: 1, React will warn us that the propType validation has failed, since we need an array and we got a number.



```
Warning: Failed propType: Invalid prop `propArray` of type `number` supplied to
`App`, expected `array`.
```

For more ,

https://docs.google.com/document/d/1IsAidVcGrwa9mC03BM3Yz5WfTeSxnO_G0kEpZiohrfw/edit

--

ReactDOM.findDOMNode

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

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: ''
    }
    this.updateState = this.updateState.bind(this);
    this.clearInput = this.clearInput.bind(this);
  };
  updateState(e) {
    this.setState({data: e.target.value});
  }
  clearInput() {
    this.setState({data: ''});
    ReactDOM.findDOMNode(this.refs.myInput).focus();
  }
  render() {
    return (
      <div>
        <input value = {this.state.data} onChange = {this.updateState}
          ref = "myInput"></input>
        <button onClick = {this.clearInput}>CLEAR</button>
        <h4>{this.state.data}</h4>
      </div>
    );
  }
}
export default App;
```

-

Keys:

React keys are useful when working with dynamically created components. Setting the key value will keep your components uniquely identified after the change.

Rendering multiple components

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((numbers) =>
  <li>{numbers}</li>
);

ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

We can refactor the previous example into a component that accepts an array of numbers and outputs an unordered list of elements.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

When you run this code, you’ll be given a warning that a key should be provided for list items. A “key” is a special string attribute you need to include when creating lists of elements. We’ll discuss why it’s important in the next section.

Let’s assign a key to our list items inside numbers.map() and fix the missing key issue.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Keys:

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

When you don’t have stable IDs for rendered items, you may use the item index as a key as a last resort:

We don’t recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state. Check out Robin Pokorny’s article for an in-depth explanation on the negative impacts of using an index as a key. If you choose not to assign an explicit key to list items then React will default to using indexes as keys.

Extracting Components with Keys

Keys only make sense in the context of the surrounding array.

For example, if you extract a ListItem component, you should keep the key on the <ListItem /> elements in the array rather than on the <li> element in the ListItem itself.

```
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Keys Must Only Be Unique Among Siblings

Keys used within arrays should be unique among their siblings. However they don't need to be globally unique. We can use the same keys when we produce two different arrays:

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
  {id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
  {id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
ReactDOM.render(
  <Blog posts={posts} />,
  document.getElementById('root')
);
```

Keys serve as a hint to React but they don’t get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name:

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

With the example above, the Post component can read props.id, but not props.key.

Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences:

React events are named using camelCase, rather than lowercase.
For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly. For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Look at this

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

If calling bind annoys you, there are two ways you can get around this. If you are using the experimental public class fields syntax, you can use class fields to correctly bind callbacks:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

This syntax is enabled by default in Create React App. If you aren’t using class fields syntax, you can use an arrow function in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Click me
      </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the LoggingButton renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

Passing Arguments to Event Handlers

Inside a loop it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use arrow functions and Function.prototype.bind respectively

In both cases, the e argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with bindany further arguments are automatically forwarded.

Conditional rendering:

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Consider these two components:

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

We’ll create a Greeting component that displays either of these components depending on whether a user is logged in:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

Element Variables

You can use variables to store elements. This can help you conditionally render a part of the component while the rest of the output doesn’t change.

Consider these two new components representing Logout and Login buttons:

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

It will render either <LoginButton /> or <LogoutButton /> depending on its current state. It will also render a <Greeting /> from the previous example:

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isLoggedIn: false};
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}
```

```
ReactDOM.render(
  <LoginControl />,
  document.getElementById('root')
);
```

Inline If with Logical && Operator

You may embed any expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical && operator. It can be handy for conditionally including an element:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

It works because in JavaScript, true && expression always evaluates to expression, and false && expression always evaluates to false.

Therefore, if the condition is true, the element right after && will appear in the output. If it is false, React will ignore and skip it.

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator condition ? true : false.

In the example below, we use it to conditionally render a small block of text.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

It can also be used for larger expressions although it is less obvious what’s going on:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Preventing Component from Rendering

In rare cases you might want a component to hide itself even though it was rendered by another component. To do this `return null` instead of its render output.

Form :

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Controlled Components

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

The textarea Tag

In HTML, a <textarea> element defines its text by its children:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

In React, a <textarea> uses a value attribute instead. This way, a form using a <textarea> can be written very similarly to a form that uses a single-line input:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Notice that this.state.value is initialized in the constructor, so that the text area starts off with some text in it.

The select Tag

In HTML, <select> creates a drop-down list. For example, this HTML creates a drop-down list of flavors:

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option value="coconut" selected >Coconut</option>
  <option value="mango">Mango</option>
</select>
```

Note that the Coconut option is initially selected, because of the selected attribute. React, instead of using this selected attribute, uses a value attribute on the root select tag. This is more convenient in a controlled component because you only need to update it in one place. For example:

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Overall, this makes it so that <input type="text">, <textarea>, and <select> all work very similarly - they all accept a value attribute that you can use to implement a controlled component.

Handling Multiple Inputs

When you need to handle multiple controlled input elements, you can add a ‘name’ attribute to each element and let the handler function choose what to do based on the value of event.target.name

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const value = target.type === 'checkbox' ? event.target.checked : event.target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            type="number"
            name="numberOfGuests"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Uncontrolled Components

In most cases, we recommend using controlled components to implement forms. **In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.**

To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

-

Refs and the DOM:

Refs provide a way to access DOM nodes or React elements created in the render method.

Creating Refs

Refs are created using React.createRef() and attached to React elements via the ref attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

Accessing Refs

When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

```
const node = this.myRef.current;
```

Adding a Ref to a DOM Element

This code uses a ref to store a reference to a DOM node:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    return (
      <div>
        <input type="text" ref={this.textInput} />
        <input type="button" value="Focus the text input" onClick={this.focusTextInput} />
      </div>
    );
  }
}
```

React will assign the current property with the DOM element when the component mounts, and assign it back to null when it unmounts. ref updates happen before componentDidMount or componentDidUpdate lifecycle hooks.

Adding a Ref to a Class Component

If we wanted to wrap the CustomTextInput above to simulate it being clicked immediately after mounting, we could use a ref to get access to the custom input and call its focusTextInput method manually:

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

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

Note that this only works if CustomTextInput is declared as a class:

```
class CustomTextInput extends React.Component {
  // ...
}
```

Refs and Functional Components

You may not use the ref attribute on functional components because they don’t have instances:

```
function MyFunctionalComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // This will *not* work!
    return (
      <MyFunctionalComponent ref={this.textInput} />
    );
  }
}
```

You should convert the component to a class if you need a ref to it, just like you do when you need lifecycle methods or state.

You can, however, use the ref attribute inside a functional component as long as you refer to a DOM element or a class component:

```
function CustomTextInput(props) {
  // textInput must be declared here so the ref can refer to it
  let textInput = React.createRef();

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

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

Passing ref :

```
import React from 'react'

/* parent component */

class RefOnChildComponent extends React.Component {
 constructor (props) {
   super(props)
   this.myInput = React.createRef()
   this.formSubmit = this.formSubmit.bind(this)
 }

 formSubmit () {
   alert(`${this.myInput.current.value} from parent`)
 }

 render () {
   return (
     <div>
       <form onSubmit={this.formSubmit} >
         <span> Parent :</span>
         <input type='text' ref={this.myInput} />
         <input type='submit' value='Submit' />
       </form>
       <br/> <br/>
       <FuncCustomComp />
     </div>
   )
 }
}


/* child component */

const FuncCustomComp = () => {
 let inputRef = null
 let innerMostRef = null
  const onClick = () => {
   inputRef.focus()
   alert(`${inputRef.value} from child component and ${innerMostRef.value} from Inner Child
component`  )
 }
  return (
   <div>
     <div>
       <span>Child : </span>
       <input type='text' ref={(input) => {inputRef = input}} />

       <InnerChild innerMostRef = { (input) => {innerMostRef = input}} />
     <input type="submit"  value="submit"  onClick={onClick}/>
    </div>
  </div>
 )
}

/* Inner child component */

const InnerChild = (props) => {
 return (
  <React.Fragment>
    <span> Inner Child :</span>
    <input type='text' ref={props.innerMostRef}
    />
   </React.Fragment>
 )
}

export default RefOnChildComponent
```

-

Fragments:

A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

```
render() {
  return (
   <React.Fragment>
     <ChildA />
     <ChildB />
     <ChildC />
   </React.Fragment>
  );
}
```

A common pattern is for a component to return a list of children. Take this example React snippet:

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

<Columns /> would need to return multiple <td> elements in order for the rendered HTML to be valid. If a parent div was used inside the render() of <Columns />, then the resulting HTML will be invalid.

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

results in a <Table /> output of:

```
<table>
  <tr>
    <div>
      <td>Hello</td>
      <td>World</td>
    </div>
  </tr>
</table>
```

So, we introduce Fragments.

Usage

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

which results in a correct <Table /> output of:

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
```

Short Syntax

There is a new, shorter syntax you can use for declaring fragments. It looks like empty tags:

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```

You can use <></> the same way you'd use any other element except that it doesn't support keys or attributes.

Error boundaries:

In the past, JavaScript errors inside components used to corrupt React's internal state and cause it to emit cryptic errors on next renders.

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Error boundaries do not catch errors for:

- Event handlers (learn more)
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

A class component becomes an error boundary if it defines a new lifecycle method called componentDidCatch(error, info):

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

  componentDidCatch(error, info) {
    // Display fallback UI
    this.setState({ hasError: true });
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info);
  }

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

Then you can use it as a regular component:

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

Note that error boundaries only catch errors in the components below them in the tree. An error boundary can't catch an error within itself. If an error boundary fails trying to render the error message, the error will propagate to the closest error boundary above it. This, too, is similar to how catch {} block works in JavaScript.

componentDidCatch Parameters

error is an error that has been thrown..
info is an object with componentStack key. The property has information about component stack during thrown error.

```
//...
componentDidCatch(error, info) {

  /* Example stack information:
     in ComponentThatThrows (created by App)
     in ErrorBoundary (created by App)
     in div (created by App)
     in App
  */
  logComponentStackToMyService(info.componentStack);
}
```

How About Event Handlers?

Error boundaries do not catch errors inside event handlers.React doesn't need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle hooks, the event handlers don't happen during rendering. So if they throw, React still knows what to display on the screen.

If you need to catch an error inside event handler, use the regular JavaScript try / catch statement:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
  }
  handleClick = () => {
    try {
      // Do something that could throw
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <div onClick={this.handleClick}>Click Me</div>

}
```

Context:

Context provides a way to pass data through the component tree without having to pass props down manually at every level. In other words, Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

When to Use Context

Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language. For example, in the code below we manually thread through a “theme” prop in order to style the Button component:

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // The Toolbar component must take an extra "theme" prop
  // and pass it to the ThemedButton. This can become painful
  // if every single button in the app needs to know the theme
  // because it would have to be passed through all components.
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

function ThemedButton(props) {
  return <Button theme={props.theme} />;
}
```

Using context, we can avoid passing props through intermediate elements:

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// A component in the middle doesn't have to
// pass the theme down explicitly anymore.
```

```
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton(props) {
  // Use a Consumer to read the current theme context.
  // React will find the closest theme Provider above and use its value.
  // In this example, the current theme is "dark".
  return (
    <ThemeContext.Consumer>
      {theme => <Button {...props} theme={theme} />}
    </ThemeContext.Consumer>
  );
}
```

Before you use Context:

If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.

For example, consider a Page component that passes a user and avatarSize prop several levels down so that deeply nested Link and Avatar components can read it:

```
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... which renders ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... which renders ...
<Link href={user.permalink}>
```

```
  <Avatar user={user} size={avatarSize} />
</Link>
```

It might feel redundant to pass down the user and avatarSize props through many levels if in the end only the Avatar component really needs it. It’s also annoying that whenever the Avatar component needs more props from the top, you have to add them at all the intermediate levels too.

One way to solve this issue without context is to pass down the Avatar component itself.

```
function Page(props) {
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={props.user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// Now, we have:
<Page user={user} />
// ... which renders ...
<PageLayout userLink={...} />
// ... which renders ...
<NavigationBar userLink={...} />
// ... which renders ...
{props.userLink}
```

This *inversion of control* can make your code cleaner in many cases by reducing the amount of props you need to pass through your application and giving more control to the root components.

React.createContext

```
const {Provider, Consumer} = React.createContext(defaultValue);
```

Creates a { Provider, Consumer } pair. When React renders a context Consumer, it will read the current context value from the closest matching Provider above it in the tree.

Provider

```
<Provider value={/* some value */}>
```

Consumer

```
<Consumer>
  {value => /* render something based on the context value */}
</Consumer>
```

Consuming multiple contexts:

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Signed-in user context
const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // App component that provides initial context values
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
          <Layout />
        </UserContext.Provider>
      </ThemeContext.Provider>
    );
  }
}

function Layout() {
  return (
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

// A component may consume multiple contexts
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

Lifecycle Events

React class components can have hooks for several lifecycle events.

Hooks allow function components to access them too, in a different way.

During the lifetime of a component, there's a series of events that gets called, and to each event you can hook and provide custom functionality.

What hook is best for what functionality is something we're going to see here.

First, there are 3 phases in a React component lifecycle:

- Mounting
- Updating
- Unmounting

Let's see those 3 phases in detail and the methods that get called for each.

Mounting
When mounting you have 4 lifecycle methods before the component is mounted in the DOM: the constructor, getDerivedStateFromProps, renderand componentDidMount.

Constructor
The constructor is the first method that is called when mounting a component.

You usually use the constructor to set up the initial state using this.state = ...

getDerivedStateFromProps()
When the state depends on props, getDerivedStateFromProps can be used to update the state based on the props value.

It was added in React 16.3, aiming to replace the componentWillReceiveProps deprecated method.

In this method you haven’t access to this as it's a static method.

It’s a pure method, so it should not cause side effects and should return the same output when called multiple times with the same input.

Returns an object with the updated elements of the state (or null if the state does not change)

render()
From the render() method you return the JSX that builds the component interface.

It’s a pure method, so it should not cause side effects and should return the same output when called multiple times with the same input.

componentDidMount()
This method is the one that you will use to perform API calls, or process operations on the DOM.

Updating

When updating you have 5 lifecycle methods before the component is mounted in the DOM: the getDerivedStateFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate and componentDidUpdate.

getDerivedStateFromProps()
See the above description for this method.

shouldComponentUpdate()
This method returns a boolean, true or false. You use this method to tell React if it should go on with the rerendering, and defaults to true. You will return false when rerendering is expensive and you want to have more control on when this happens.

render()

See the above description for this method.

getSnapshotBeforeUpdate()

In this method you have access to the props and state of the previous render, and of the current render.

Its use cases are very niche, and it’s probably the one that you will use less.

componentDidUpdate()

This method is called when the component has been updated in the DOM. Use this to run any 3rd party DOM API or call APIs that must be updated when the DOM changes.

It corresponds to the componentDidMount() method from the mounting phase.

Unmounting

In this phase we only have one method, componentWillUnmount.

componentWillUnmount()

The method is called when the component is removed from the DOM. Use this to do any sort of cleanup you need to perform.

Note :

If you are working on an app that uses componentWillMount, componentWillReceiveProps or componentWillUpdate, those were deprecated in React 16.3 and you should migrate to other lifecycle methods.

React router:

```
./index.js
import React from 'react'
import ReactDOM from 'react-dom'

ReactDOM.render(
<AppContainer />
 , document.getElementById('root')
)

./app.js

import { Switch, Route, BrowserRouter } from 'react-router-dom'
render () {
   return (
     <BrowserRouter>
       <Switch>
         <Route exact path='/' component={SignInContainer} />
         <Route exact path='/account/set-password' component={SetPasswordContainer} />
         <PrivateRoute exact path='/home' component={HomeContainer} />
         <PrivateRoute exact path='/account/update' component={UpdateProfileContainer} />
         <Route exact path='/call/video' component={VideoCallContainer} />
         <Route exact path='/call/audio' component={AudioCallContainer} />
       </Switch>
     </BrowserRouter>

   )

 }
```

Why Immutability Is Important

In the previous code example, we suggested that you use the .slice() operator to create a copy of the squares array to modify instead of modifying the existing array. We'll now discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data. The first approach is to *mutate* the data by directly changing the data's values. The second approach is to replace the data with a new copy which has the desired changes.

Data change with Mutation:

```
var player = {score: 1, name: 'Jeff'};
player.score = 2;
// Now player is {score: 2, name: 'Jeff'}
```

Data change without Mutation:

```
var player = {score: 1, name: 'Jeff'};

var newPlayer = Object.assign({}, player, {score: 2});
// Now player is unchanged, but newPlayer is {score: 2, name: 'Jeff'}

// Or if you are using object spread syntax proposal, you can write:
// var newPlayer = {...player, score: 2};
```

The end result is the same but by not mutating gain several benefits described below.

- **Detecting Changes**

Detecting changes in mutable objects is difficult because they are modified directly. This detection requires the mutable object to be compared to previous copies of itself and the entire object tree to be traversed.

Detecting changes in immutable objects is considerably easier. If the immutable object that is being referenced is different than the previous one, then the object has changed.

- **Determining When to Re-render in React**

The main benefit of immutability is that it helps you build *pure components* in React. Immutable data can easily determine if changes have been made which helps to determine when a component requires re-rendering.

You can learn more about shouldComponentUpdate() and how you can build *pure components* by reading Optimizing Performance.
