# React Reference Document

Michael Long

Last Updated: 04/Aug/2020

**Preface** 4

**Section I -- Introductory Topics** 6

**0 Introduction to React** 6
- 0.0 Why React? 6

**1 React Concepts** 6
- 1.0 ReactDOM & JSX 6
- 1.1 Hello World of React 6
  - 1.1a Example 6

**2 Components** 7
- 2.0 Introduction to Components 7
- 2.1 Functional Components 7
  - 2.1a Example 7
- 2.2 Component Conventions 7
- 2.3 Parent / Child Components 8
  - 2.3a Example 9
- 2.4 Adding JavaScript into JSX 9
  - 2.4a Convert JSX to JavaScript 9
- 2.5 Events in React / JSX 10
  - 2.5a Example 10

**3 Styling in React** 11
- 3.0 Assigning Classes in JSX 11
  - 3.0a Example 11
- 3.1 Styling React with CSS 11
- 3.2 Inline Styling with JSX 11

**4 Properties (Props)** 12
- 4.0 Concept of Props 12
- 4.1 Applying Props to Components with Parameters 12
  - 4.1a Example 12
  - 4.1b Example 12
  - 4.1c Example 13

**5 Data in Components** 14
- 5.0 Useful JavaScript Functions 14
  - 5.0a Entire Array Functions 14

array.map() 14
array.filter() 14
array.sort() 14
5.0b Entire Array to One Value Functions 14
array.find() 14
array.reduce() 14
array.reduceRight() 14
array.findIndex() 14
5.0c Boolean Check on Entire Array 14
array.every() 14
array.some() 14
5.1 Introduction 15
5.2 “map()” with Components 15
5.2a Example 16
“App.js” 16
“myNameData.js” 16

**6 Components as Classes 17**
6.0 Converting Function to Class 17
6.0a Function to Class 17
“Function version” 17
“Class version” 17
6.1 Constructors 18
6.1a Example 18
“constructor()” method 18
6.2 Methods in Classes 18
6.2a Binding Methods 19

**7 State With Components 20**
7.0 “State” Concept 20
7.1 Initial State 20
7.1a Example 20
7.2 Changing State 21
7.2a Simple Example 21
7.2b Binding Example 22
7.2c Using Previous State Example 23
7.2d Alternative to Binding Method 24
7.3 Lifecycle Methods 24

7.3a Understanding Lifecycle Methods 24
“render()” method 24
“componentDidMount()” method 24
“shouldComponentUpdate()” method 24
“componentWillUnmount()” method 24
7.3b Deprecated Lifecycle Methods 25
7.4a Conditional Rendering 25
Example using Condition ? True : False 25
Example using Condition && Condition ? True : False 25

**8 Working with Data 26**
8.0 Fetch and APIs 26
8.0a Example API Call 26
8.1 Form Data 26
8.1a General Approach to Input 26
Simple Input Example 27
8.1b Form Elements in JSX 27
“textarea” Example 27
“checkbox” Example 28
“radio” Example 28
“select” Example 29
8.1c Submit a Form 29
Simple Submit Example 29

**9 Introductory Conclusion / Best Practices 31**
9.0 Container / Component Construct 31

**Section II -- Advanced Topics 32**

**10 Introduction to Advanced Topics 32**

**11 Hooks 32**
11.1 State in Functional Components 32
11.1a Example: “useState()” 32
11.2 Multiple Hooks 32
11.2a Example: Setting Multiple Hooks 33
11.2b Example: Object in Hook 33
11.3 Lifecycle Methods with Hooks 33
11.3a “componentDidMount()” in “useEffect()” 34
Syntax for “componentDidMount()” 34

11.3b “componentDidUpdate()” in “useEffect()” 34
Syntax for “componentDidUpdate()” 34
11.3c “componentWillUnmount()” in “useEffect()” 34
Syntax for “componentWillUnmount()” in “useEffect()” 34
11.4 Conclusion on Hooks 34

## Preface

This document explores some of the basic uses of the React framework and how to get started. Although there is sufficient information to get started, this document is intended to be used as a reference when working with React, as some concepts should be studied and practiced beyond the material provided.

The material is a collection of notes written from the following course:

Introductory Material: Section I: [0 ~ 9]
Ziroll, Bob. freeCodeCamp.org, “Learn React JS - Full Course for Beginners - Tutorial 2019”, Dec., 2018. Accessed Aug. 4, 2020
https://www.youtube.com/watch?v=DLX62G4lc44&t=13155s

Advanced Material: Section II: [11]
Sevilleja, Chris. scotch.io, “Getting Started with React Hooks”, Oct. 27, 2018. Accessed Aug. 4, 2020
https://scotch.io/tutorials/getting-started-with-react-hooks

I hope this document is useful, and is available to be shared.

## Section I -- Introductory Topics

### 0 Introduction to React

#### 0.0 Why React?

React Framework is fast through the usage of the Virtual DOM, the abstraction of reusable “components”. Maintained by Facebook, and currently desirable in the job market.

### 1 React Concepts

#### 1.0 ReactDOM & JSX

React uses a pseudo-JavaScript language called JSX. React interprets the JSX and “compiles” it down into JavaScript.

#### 1.1 Hello World of React

```
import React from “react”
import ReactDOM from “react-dom”

//ReactDOM.render(<what to render>,<where to render it>)
ReactDOM.render(
<h1>Hello World!</h1>,
document.getElementById(“root”)
)
```

Note: a “render” call can only contain one element in it’s render, if multiple elements need to rendered, consider wrapping them in something like a <div>

#### 1.1a Example

```
ReactDOM.render(
 <div>
       <h1>Hello World!</h1>
       <p>Paragraph</p>
 </div>,
 document.getElementById(“root”)
)
```

# 2 Components

## 2.0 Introduction to Components

“Components” are the main building blocks, this is one of the main constructs that makes React a powerful tool for readability, reusability, and organization.

## 2.1 Functional Components

To make organizing and composing the different components to render easier, we will be wrapping them in a function.

### 2.1a Example

```
function Home() {
return (
            <div>
                  <h1>Homepage</h1>
                  <p>This is my page</p>
            </div>
      )
}

ReactDOM.render(<Home />, document.getElementByID(“root”))
```

## 2.2 Component Conventions

An individual component should be in it’s own file named the same way the function is, i.e “Home()” should be called “Home.js”. Note that each new component will need to have React imported, and then export it with the JavaScript ES6 convention. The new component will then need to be reimported into the main JavaScript file, i.e “index.js”. Refer to 2.2a for an example.

## 2.2a Example

“components/App.js”
```
import React from “react”

function App() {
return (
            <div>
                  <h1>Homepage</h1>
                  <p>This is my page</p>
            </div>
      )
}

export default App
```

“index.js”
```
import React from “react”
import ReactDOM from “react-dom”
import Home from “./components/Home” // or “./components/Home.js”

ReactDOM.render(<App />, document.getElementById(‘root’))
```

## 2.3 Parent / Child Components

Hierarchy of components allows for more complex interactions and further modularity. A component is able to render another component through nesting, thus allowing the hierarchical structure to exist. Refer to “2.3a Example”.

### 2.3a Example

"components/Nested.js"

```
import React from “react”

function Nested() {
return (
            <div>
                  <p>I am nested</p>
            </div>
      )
}

export default Nested
```

"components/App.js"

```
import React from “react”
import Nested from “./Nested”

function App() {
return (
            <div>
                  <Nested />
            </div>
      )
}

export default App
```

"index.js"

```
import React from “react”
import ReactDOM from “react-dom”
import App from “./components/App”

ReactDOM.render(<App />, document.getElementById(‘root’))
```

## 2.4 Adding JavaScript into JSX

### 2.4a Convert JSX to JavaScript

When the interpreter looks at the code with “<” in JSX, it is interpreted as if it were HTML, whereas using brackets to transition to JavaScript. Thus, to add JavaScript into JSX, encase the JavaScript in brackets, see “2.4b Example” for how this should look in code.

### 2.4b Example

```
firstname = ‘Michael’
lastname = ‘Long’
return (
<h1>Hello {firstname + ‘ ‘ + lastname} World</h1>
<h1>Hello {`${firstname} ${lastname}`} World</h1>
)
```

## 2.5 Events in React / JSX

Events are handled similarly to HTML, in which the event handler is referenced as the JavaScript version (where it is camel cased, e.g “onclick” in HTML would be “onClick” in JSX. Modification of data with Events will be explored in section “7. State with Components”.

List of available events can be found here:
https://reactjs.org/docs/events.html#supported-events

### 2.5a Example

```
import React from “react”

function App() {
return (
<button onClick={() => console.log(“Clicked”)}>
            Clickable
      </button>
      )
}

export default App
```

# 3 Styling in React

## 3.0 Assigning Classes in JSX

Some of the characters cannot be used in JSX, thus they need to be referred to in their camel case equivalent. In addition, properties such as “class” need to be referred to as “className” in which it can be styled from there. Likewise class list will be referred to as “classList”

### 3.0a Example

```
| return (
| <h1 className='to_style'>Hello World</h1>
| )
```

## 3.1 Styling React with CSS

*works exactly like conventional styling with CSS once the class has been assigned*
Documentation: https://developer.mozilla.org/en-US/docs/Web/CSS

## 3.2 Inline Styling with JSX

*less relevant, can be used for dynamic styling with JavaScript, but for the time being this is a distraction from the core concepts of React. May expand on this section later when relevant*

# 4 Properties (Props)

## 4.0 Concept of Props

Properties, referred to as “Props”, are a method for assigning attributes to a component to make components modular. Effectively, with this concept a component becomes reusable.

## 4.1 Applying Props to Components with Parameters

Similar to a function, you can pass components parameters.

### 4.1a Example

```
<MyComponent
firstname=”Michael”
      lastname=”Long”
/>
```

To actually assign it, access it through an object passed in the arguments, conventionally called “props”

### 4.1b Example

```
function MyComponent(props) {
return (
            <div className=”myclass”>
                  <h1>First Name: {props.firstname}</h1>
                  <h1>Last Name: {props.lastname}</h1>
            </div>
      )
}
```

Oftentimes it is more useful to pass an object instead of hardcoded values, in the event that we use something like JSON.

4.1c Example

```
function MyComponent(props) {
return (
            <div className=”myclass”>
                  <h1>First Name: {props.nameobj.firstname}</h1>
                  <h1>Last Name: {props.nameobj.lastname}</h1>
            </div>
      )
}
```

Assuming the object is

```
{nameobj: {firstname: “Michael”, lastname: “Long”} }
```

# 5 Data in Components

## 5.0 Useful JavaScript Functions

### 5.0a Entire Array Functions

#### array.map()

```
array.map(function()) // applies function to each value, returns array
```

#### array.filter()

```
array.filter(function()) // finds all values that match, returns array
```

#### array.sort()

```
array.sort() // Returns sorted array
```

### 5.0b Entire Array to One Value Functions

#### array.find()

```
array.find(function()) // finds first instance matching condition,
// returns one value
```

#### array.reduce()

```
array.reduce(function()) // reduces array to single value from
// left to right, returns one value
```

#### array.reduceRight()

```
array.reduceRight(function()) // reduces array from right to left,
// returns one value
```

#### array.findIndex()

```
array.findIndex(function()) // finds first instance matching condition,
returns one value
```

### 5.0c Boolean Check on Entire Array

#### array.every()

```
array.every(function()) // Checks if all values match condition, returns
boolean
```

#### array.some()

```
array.some(function()) // Checks if some values match
condition, returns true if one or more match
```

Note: Reference documentation for these methods can be found here: https://www.w3schools.com/jsref/jsref_obj_array.asp

## 5.1 Introduction

Data is typically going to be acquired through an API or some other endpoint. Insertion and manipulation of data is easier when we apply higher order array functions to manipulate or filter the data we are getting. This can then be applied to creating components.

## 5.2 “map()” with Components

With the array.map() method, a new component can be instantiated and easily placed into a ReactDOM.render() method. See “6.2a Example” for what this looks like.

## 5.2a Example

“App.js”

```
 1. | import myNameData from “./myNameData”
 2. |
 3. | function App() {
 4. | const myNewComponents = myData.map(
 5. |             dataToAdd => <aNameComponent
 6. |                   key={myData.akey}
 7. |                   first={myData.first}
 8. |             last={myData.last}
 9. |             />
10. | )
11. |
12. | return (
13. |       <div>
14. |             {myNewComponents}
15. |       </div>
16. | )
17. | }
18. | export default App
```

“myNameData.js”

```
 1. | const myNameData = [
 2. | {
 3. |       akey: 1,
 4. |             first: “Michael”
 5. |             last: “Long”
 6. |       }
 7. | ]
```

Note: When using a Higher-Order array function into a component, there needs to be a unique “key” value, otherwise the JSX compiler will throw a warning. See line 6 in “App.js” in “6.2a Example” above. Most APIs should have some id value to use in their data.

Note: When assigning property data, remember that it needs to be expressed as JavaScript within the JSX on lines 5 ~ 9 in “App.js” in “6.2a Example” above.

## 6 Components as Classes

### 6.0 Converting Function to Class

While a component can be represented as a function, some additional features of components can be utilized when the component is represented as a class which inherits from the “React.Component” class.

#### 6.0a Function to Class

“Function version”

```
import React from “react”

function App() {
return (
       <h1>Hello World</h1>
       )
}

export default App
```

“Class version”

```
import React from “react”

class App extends React.Component {
render() {
       <h1>Hello World</h1>
       }
}

export default App
```

Note: Since a class construction is being used instead of a function: props, and methods need to be referred to with “{this.prop.value}”, or “this.method()”. This is a common error when transitioning from functional components to class-based components.

## 6.1 Constructors

Constructor is needed for enabling dynamic data in the next section “7. State With Components”. This construction allows inheritance within components and provides more functionality such as state.

### 6.1a Example

“constructor()” method

```
import React from “react”

class App extends React.Component {

      constructor() {
            super() // Inherit from the super class
            this.state = { // See “Note” below
                  first: “Michael”
                  last: “Long”
            }
      }

render() {
      <h1>
            My Name: {this.state.first} {this.state.last}
            </h1>
      }

}

export default App
```

Note: “this.state” is an object which will contain all of the dynamic values tied to the component. This concept will be explored more in the next section (“7. State With Components”)

## 6.2 Methods in Classes

As with most class abstractions, a method can be declared within a class. See this reference for using classes in JavaScript https://www.w3schools.com/js/js_object_classes.asp

## 6.2a Binding Methods

Binding with “Function.prototype.bind()” allows a class to create a new method from an existing method, effectively “borrowing” it. Refer to this reference for binding a method: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind

This is important to note for later sections in which some of the “React.Component” methods will be required to be binded in order to function.

## 7 State With Components

### 7.0 “State” Concept

The concept of “state” is going to enable dynamic memory within a React application. State is simply referring to a snapshot of data. State is initialized in the “constructor()” method as an object with an initial set of values. Once state is created, React will abstract the modification of state in the JSX through several different control methods which will be explored below.

### 7.1 Initial State

State is initialized in the “constructor()” method, “8.1a Example” demonstrates this, and is the same example as “7.1a Example”.

7.1a Example

```
import React from “react”

class App extends React.Component {

      constructor() {
            super() // Inherit from the super class
            this.state = {
                  first: “Michael”
                  last: “Long”
            }
      }

render() {
      <h1>
            My Name: {this.state.first} {this.state.last}
            </h1>
      }

}

export default App
```

State is initialized in the “constructor()” method as an object with entries to describe the initial value.

When data in the state object needs to be used, it can be referenced with “this.state.myData” whereas “myData” is the data you would like to reference.

## 7.2 Changing State

The power of state is the ability to have dynamic data, thus it should be modifiable. State cannot be modified directly, as state’s are technically immutable snapshots, in which everytime state changes: the entire state object is replaced with a new state. State is updated with the method “setState()”, see “7.2a Simple Example” for how this would look in code.

### 7.2a Simple Example

```
setValue() {
this.setState( {value: 1} )
}
```

However, for this code to work, the method “setValue()” needs to be bound in the constructor of the class. See “6.2a Binding Methods” for reference. A complete example can be seen in “7.2b Example”, refer to line 10 for the example.

## 7.2b Binding Example

```
import React from “react”

class App extends React.Component {

      constructor() {
            super()
            this.state = {
                  value: 0
            }
            this.setValue = this.setValue.bind(this)
      }

setValue() {
      this.setState( {value: 1} )
}

render() {
      <h1>
            My Value: {this.state.value}
            </h1>
      }

}

export default App
```

With this the function to update the value will now work, however note that:
1. The function isn’t being called yet
2. It isn’t utilizing previous state.
To use the previous state, we’ll refer to “7.3c Using Previous State Example”.

## 7.2c Using Previous State Example

```
import React from “react”

class App extends React.Component {

      constructor() {
            super()
            this.state = {
                  value: 0
            }
            this.setValue = this.setValue.bind(this)
      }

setValue() {
      this.setState(prev => {
            return {
                  value: prev.value + 1
            }
      })
}

render() {
      <h1>
            My Value: {this.state.value}
            </h1>
      }

}

export default App
```

Now by holding onto the previous state then updating the current state: the “value” can be set to be the previous state’s “value” plus one. Note that you cannot use the increment modifier (i.e “++”) since you would be modifying the state. State is immutable, but a new state can be set over the current state.

### 7.2d Alternative to Binding Method
A method that uses the “setState()” method can alternatively be declared as an arrow function instead of being bound. This functionality is supported in later versions of React, and is a less bug prone way of writing React components.

## 7.3 Lifecycle Methods

### 7.3a Understanding Lifecycle Methods

Lifecycle methods are functions that are invoked when an event happens, such as creation, deletion, render, etc. This section will list out some of the methods

“render()” method

Method responsible for displaying a component to the screen, this method will be invoked when React determines a change needs to be rendered. Is invoked many times.

“componentDidMount()” method

Method that is invoked when the component is first created, useful for setting up initialization values, obtaining API data, displaying a loading bar, etc. Is only invoked once.

“shouldComponentUpdate()” method

Method that will trigger to rerender a component, this is a check that will enable logic to determine whether it should render. Is invoked many times.

“componentWillUnmount()” method

Method that is invoked when a component is being removed from the application. Is useful for cleaning up data. Is invoked once.

## 7.3b Deprecated Lifecycle Methods

Deprecated methods can be found here:
https://reactjs.org/blog/2018/03/29/react-v-16-3.html#component-lifecycle-changes

## 7.4a Conditional Rendering

Ternary operators can be applied to determine which component should be rendered. When applying these operations, when the component is mounted with the lifecycle method “componentDidMount()” you can update a conditional to determine whether or not one component should be present or another.

Example using Condition ? True : False

```
<div>
 {props.loading ? <LoadingComp /> : <LoadedComp>}
</div>
```

Example using Condition && Condition ? True : False

```
<div>
 {props.cond && props.cond2 ? <True /> : <False />
</div>
```

## 8 Working with Data

### 8.0 Fetch and APIs

Similar to how vanilla JavaScript works, React supports the "fetch()" method with promise callbacks. For reference on using "fetch()", see: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

#### 8.0a Example API Call

```
componentDidMount() {
// consider setting initial states (e.g loading)
fetch("https://myapi.com/api/1")
      .then(response => response.json())
.then(data => {
      this.setState({
            name: data
      })
})
}
```

### 8.1 Form Data

When gathering input, a user will provide data that needs to be stored and updated within state. State will be updated from user input, then the state will be rendered back to the user. The method "onChange={}" will fire each time the element changes state. The value in the element can be obtained by accessing "event.target.value".

#### 8.1a General Approach to Input

Input field in JSX should be given the following properties: "name" (as a means to access the correct value when updating state), "onChange={}" event (as a means to execute a function to change the state", a value to represent what's in state, and a function to update the state.

The value that is displayed on screen should match state, as "state should be the single source of truth" (Bob Ziroll).

## Simple Input Example

```
// assume class component

inputUpdate(event) {
const {name, value} = event.target
this.setState({ [name]: value })
}

render() {
return (
      <form>
            <input
                  type=”text”
                  value={this.state.myVal}
                  name=”myVal”
                  onChange={this.inputUpdate}
            />
      </form>
)
}
```

## 8.1b Form Elements in JSX

### “textarea” Example

```
// assume class component

inputUpdate(event) {
const {name, value} = event.target
this.setState({ [name]: value })
}

render() {
return (
      <form>
            <textarea
                  // Note, textarea is self closing
                  value={this.state.myVal}
                  name=”myVal”
                  onChange={this.inputUpdate}
            />
      </form>
)
}
```

“checkbox” Example

```
// assume class component

inputUpdate(event) {
const {name, value, type, checked} = event.target
type !== “checked” ? this.setState({ [name]: value }) :
this.setState({ [name]: checked })
}

render() {
return (
      <form>
            <input
                  type=”checkbox”
                  checked={this.state.myCheckVal}
                  name=”myCheckBox”
                  onChange={this.inputUpdate}
            />
      </form>
)
}
```

“radio” Example

```
// assume class component

inputUpdate(event) {
const {name, value, type, checked} = event.target
type !== “checked” ? this.setState({ [name]: value }) :
this.setState({ [name]: checked })
}

render() {
return (
      <form>
            <input
                  type=”radio”
                  value=”Option A”
                  name=”mySharedRadio”
                  onChange={this.inputUpdate}
                  checked={this.state.isPicked === “Option A”}
            />
      </form>
)
}
```

“select” Example

```
// assume class component

inputUpdate(event) {
const {name, value, type, checked} = event.target
type !== “checked” ? this.setState({ [name]: value }) :
this.setState({ [name]: checked })
}

render() {
return (
      <form>
            <select
                  value={this.state.optionSelected}
                  name=”mySelect”
                  onChange={this.inputUpdate}
            >
                  <option value=”OptionA”>Option A</option>
                  <option value=”OptionB”>Option B</option>
                  <option value=”OptionC”>Option C</option>
            </select>
      </form>
)
}
```

## 8.1c Submit a Form

Best practice is to have a function which handles a submit within the form element.

Simple Submit Example

```
// assume class component

inputSubmit() {
// Submit data to endpoint, or other action
}

render() {
return (
      <form onSubmit={this.inputSubmit}>
            <button>Submit</button>
      </form>
)
}
```

Note: In HTML5, when a button is found within a form, it’s default input type is changed to be “input type=”submit”, thus we can just have the button without modifying any value.

# 9 Introductory Conclusion / Best Practices

## 9.0 Container / Component Construct

To make code more readable, it is best practice to separate rendering logic and state logic. The “Container” will have a majority of the JSX and handle what is displayed on screen, the “Component” will thus contain the state management. The Component in the “render()” method will then simply pass in the Container as if it was another component.

## Section II -- Advanced Topics

## 10 Introduction to Advanced Topics

This section of the document will break away from the introductory topics and look at some of the changes to React as the framework evolves.

## 11 Hooks

Previously, components with state were defined as classes. The direction of React however is looking to take the abstraction of a component into strictly functional. A “Hook” is a feature that allows React class features to be used without a class.

### 11.1 State in Functional Components

With a hook, state is created and modified with the “useState(value, function())” method, in which “value” is the name of the value, and the “function()” is a method invoked to update the state. This can be thought of as “this.state” and “this.setState()” respectively.

11.1a Example: “useState()”

```
function App() {
const [myVal, updateFunction] = useState(“”)

return (
      <h1>Initial State: {myVal}</h1>
)
}
```

## 11.2 Multiple Hooks

When multiple states are needed, a hook needs to be defined for each.

### 11.2a Example: Setting Multiple Hooks

```
function App() {
const [myValOne, updateFunctionOne] = useState(“One”)
const [myValTwo, updateFunctionTwo] = useState(“Two”)

return (
      <h1>Initial State: {myValOne} {myValTwo}</h1>
)
}
```

Alternatively, an object can be passed within the “useState()” method

### 11.2b Example: Object in Hook

```
function App() {
const [myVal, updateFunction] = useState([{
      firstName: “Michael”,
      lastName: “Long”,
      age: 25
}])

return (
      <h1>Initial State:
            {myVal.firstName},
            {myVal.lastName},
            {myVal.ageName},
      </h1>
)
}
```

## 11.3 Lifecycle Methods with Hooks

Lifecycle Methods are one of the primary reasons to use a class based approach to components. With hooks, a similar method called “useEffect(function(), array)” in which “function()” is the method applied when a render occurs and “array” which will only run “function” if a value changes.

## 11.3a “componentDidMount()” in “useEffect()”

If an empty array is passed to this method, it will only run once on mount. This is equivalent to “componentDidMount()”.

Syntax for “componentDidMount()”

```
useEffect( () => myFunction(), [])
```

## 11.3b “componentDidUpdate()” in “useEffect()”

If a non-empty array is passed to this method, it will run if that array changes.

Syntax for “componentDidUpdate()”

```
useEffect( () => myFunction(), [value-to-check])
```

## 11.3c “componentWillUnmount()” in “useEffect()”

To run a method before the component unmounts, the “useEffect()” method can execute a function on return with no secondary argument.

Syntax for “componentWillUnmount()” in “useEffect()”

```
useEffect( () => { return () => { //insert exit function here } })
```

## 11.4 Conclusion on Hooks

Hooks are a useful abstraction to remove the need for class based components altogether. With the method “useEffect()”, the equivalent lifecycle methods can be implemented and used.