# React Js Core Concepts

*By* **Sunil Kumar**

# Preface

This **React Js Core Concepts** handbook contains the core concepts of ReactJs along with React-Router.

The book does not try to cover every concept of React & React Router, instead it focuses on the minimum amount of knowledge required to work as an individual contributor.

This book is written by Sunil Kumar. My passion is to learn and deep dive in frontend development. I also write on topics like JavaScript, TypeScript, Angular, SCSS, NodeJs and various other frontend technologies and frameworks. I also publish UI related work on my website https://www.ui-geeks.in.

*You can reach me at:*
Email: skumar.mca2010@gmail.com
Twitter: @skumar_mca2010

Would appreciate your constructive feedback.

**Enjoy reading the book!**

# Table of Content

*CHAPTER 1*

# Introduction

# Preface to React

## What is React

*Declarative, efficient JS Library for building User Interfaces*

## Single Page Application

*Application that loads a single HTML page, updates views without reloading*

## Key Features

*Lightweight, Component-based, Virtual DOM, Uni-directional*

## Virtual DOM

*Concept where virtual representation of UI is kept in memory*

## Shadow vs Virtual DOM

*Browser own Shadow DOM, Libraries own Virtual DOM*

## React Fiber

*Reconciliation engine of React 16*

## Important Milestones

*Journey from v0.3 on May, 13 -  v16.8 on Feb 17 with Hooks and running*

## Create React App

*Setup your first React application*

## Hello World

*Your first React App*

## Why React

*Comparing React with Angular, Vue*

## React

React is a declarative, efficient, and flexible JavaScript library for building interactive user interfaces. It is an open-source JavaScript framework developed by Facebook (now Meta). React helps to build encapsulated *components* that manage their own *state*, then compose them to make complex UIs.

***DOM*** manipulation is an expensive task, therefore, React uses the concept of **"Virtual DOM"** for efficient *DOM* manipulation, where an *in memory JavaScript representation* is maintained and only necessary updates are made to the actual DOM.

React is used to build Single Page Application (SPA).

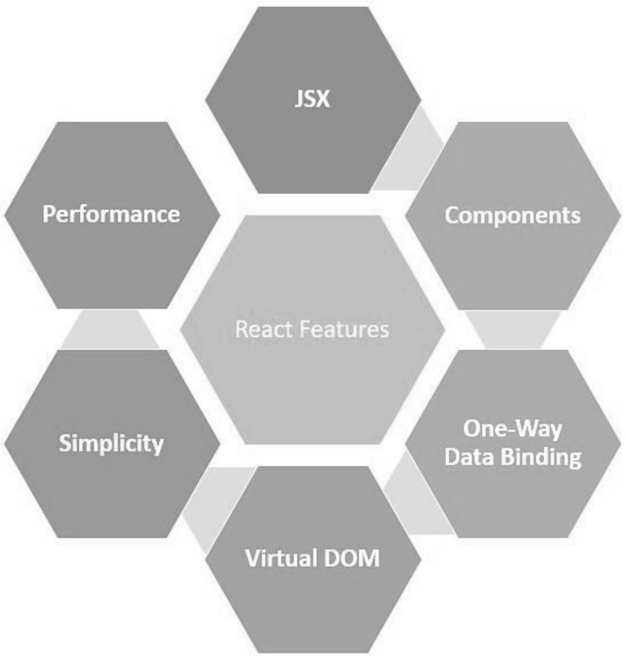
### Single Page Application

A Single-Page Application (SPA) is an application that **loads a single HTML** page and all the necessary assets (such as JavaScript and CSS) required for the application to run. It uses HTML5 and Ajax to facilitate quick and smooth responses to user events.

Any interactions with the page or subsequent pages **do not require a round trip** to the server which means the page is **not reloaded**. All the views of the app are loaded and unloaded into the same page itself.

SPAs are fast to render but it comes with some tradeoff disadvantages such as *SEO* (Search Engine Optimization), additional effort required to main *application state*, implement page *navigation*, etc.

## Key Features



- **Lightweight, fast** and modern way of creating **User Interfaces**.
- **Reusable component-based** approach.
- Quick Rendering With **Virtual DOM**.
- **Unidirectional data** flow, provides stable code.
- An open-source Facebook library: **Huge ecosystem** and flexibility.
- Backward compatibility.
- Efficient Design And **Developer Tools**.
- Ease Of Use.

## What is Virtual DOM

The **virtual DOM (VDOM)** is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called ***reconciliation***.

The term "virtual DOM" is usually associated with ***React elements*** since they are the objects representing the user interface. React, however, also uses internal objects called ***"fibers"*** to hold additional information about the component tree.

## Is the Shadow DOM the same as the Virtual DOM

**No**, they are different. The *Shadow DOM* is a browser technology designed primarily for scoping variables and CSS in ***web components***.

The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

## What is React Fiber

**Fiber** is the new ***reconciliation engine*** in **React 16**. It is an ongoing reimplementation of React's code algorithm. It is the culmination of over two years of research by the React team.

Its main goal is to enable **incremental rendering** of the virtual DOM along with the ability to **pause, abort, or reuse work** as new updates come in, ability to **assign priority** to different types of updates and **new concurrency** primitives.

## Create React App

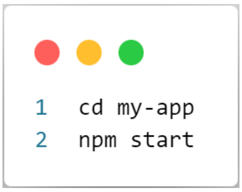
It is the best way to create a new single page application in React. It uses ***Babel*** and ***Webpack*** under the hood.

**Pre-requisites:**

1. NodeJs >= 14.0.0
2. Npm >= 5.6

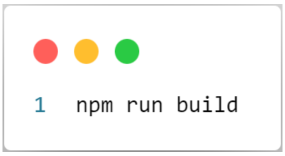
*Syntax*:
**npx *create-react-app*** *<app-name>*
***npx*** *is a package runner tool that comes with the npm v**5.2+**.*

*Example*: npm ***create-react-app*** *my-app*
Then navigate to the application and run command **npm start**.



```
cd my-app
npm start
```

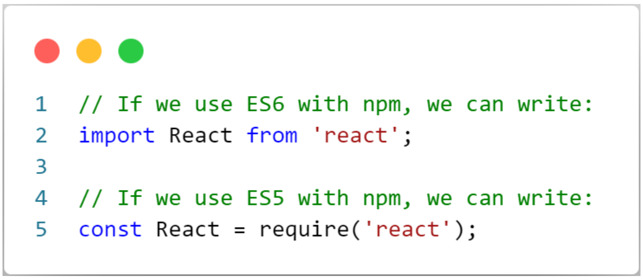
**For creating a production build**, run the build command. It will create an optimized build of your app in the ***build*** *folder*.



```
npm run build
```

**Note**: There are various other ways of creating a react application. More details here: *https://react.dev/learn/start-a-new-react-project*

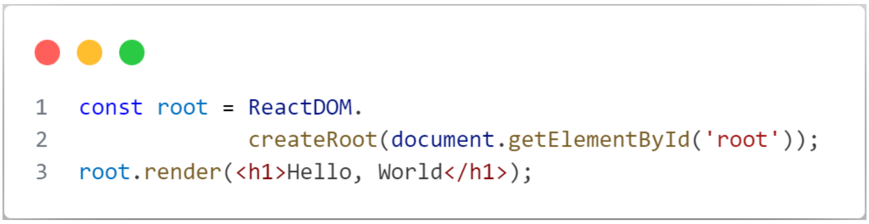
***React*** is the entry point to the React library.



```
// If we use ES6 with npm, we can write:
import React from 'react';

// If we use ES5 with npm, we can write:
const React = require('react');
```

If we load *React* from a **<script>** tag, the top-level APIs are available on the ***React*** global.

## Hello World Program



```
const root = ReactDOM.
             createRoot(document.getElementById('root'));
root.render(<h1>Hello, World</h1>);
```

***ReactDOM.createRoot()***

This method creates a ***root*** *node inside the browser DOM element*, on which the react app will be mounted/hosted. Everything inside the *root node* is managed by React DOM.

***root.render()***

This method renders/displays the app content on the **root node**. This method returns *undefined*. Above code will display a ***H1*** heading saying "**Hello, World**".

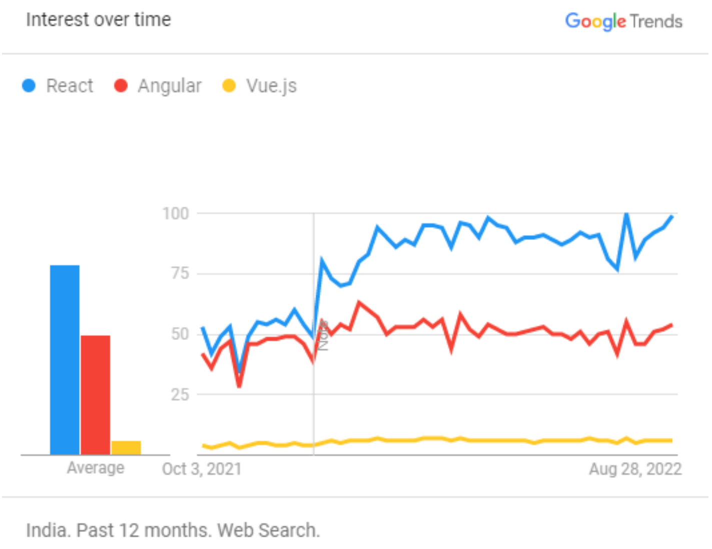
The **first time** we call ***root.render()***, React will clear all the existing HTML content inside the root node before rendering the React component into it.

If we call the ***root.render()*** method **multiple times on the same root node**, React will update the DOM as necessary to reflect the latest JSX.

## Why React

The key features of React makes it quite popular among other frontend libraries and frameworks. Below trend from **Google** shows why companies are choosing React for developing the UI.

### Google Trend



## Important Milestones

| **Version** | **Date** | **Description** |
|---|---|---|
| *v0.3* | May, 2013 | First public release. |
| *v0.14* | Oct, 2015 | **Splitted** main React package into **react and react-dom**. Added stateless function component syntax. |
| *v15.3* | Jul, 2016 | Added **React.PureComponent.** |
| *v15.4* | Nov, 2016 | **Separated React-DOM** and distributed as a separate package. |
| *v15.6* | Jun, 2017 | Added support for **CSS variables** in style attribute. |
| *v16.0* | Sep, 2017 | Added **Error Boundary.** |
| *v16.6* | Oct, 2018 | Added ***React.memo()*** as an alternative to *PureComponent* and ***React.lazy()*** for code splitting. |
| *v16.8* | Feb, 2019 | Added **Hooks** to use *state* and other React features without writing a *class component*. |
| *v16.9* | Aug, 2019 | Added **React.Profile** API for gathering performance measurements. |
| *v16.13* | Feb, 2020 | Added React **Concurrent Mode**. |
| *v18.0* | Mar, 2022 | Added ***useId*** Hook for generating unique IDs and **enabled automatic batching** of state updates. |

## Summary

1. React is a declarative, efficient library for building **user interfaces**. It is used to build ***Single Page Application (SPA)***.
2. SPA loads a **single HTML page**, interactions on page do not require round trip to server.
3. *Lightweight*, *Virtual DOM*, *Uni-directional data* flow are few of the key features of React.
4. **Virtual DOM** is the **in-memory** representation of the UI, and uses a ***reconciliation*** process to update **DOM**.
5. **Shadow DOM** is different from the *Virtual DOM*.
6. **Fiber** is the new reconciliation engine in React 16.
7. ***npx create-react-app*** can be used for creating react applications, using *Babel* and *Webpack* under the hood.

*CHAPTER 2*

# JSX

# Preface to JSX

## What is JSX

*Syntax extension to JavaScript, produces React elements*

## Why JSX

*Because rendering logic is inherently coupled with UI*

## Embedding Expressions

*Embedding JS, attributes, childrens*

## JSX Represents Objects

*Babel compiles the JSX to return object*

## Rendering Elements

*Add elements to Virtual DOM, update only what's necessary*

# JSX

It is a syntax extension to JavaScript. JSX may look like a template language, but it comes with the full power of JavaScript. JSX produces React ***"elements"***.

*const element = <h1>Hello, World<h1>*

## Why JSX

UI is primarily built on *HTML*, *JavaScript* and *CSS*. From the initial days, we kept content in HTML and logic in JavaScript, usually in separate files.

But React embraces the fact that rendering logic is inherently coupled with other UI logic like handling events, state changes, etc. Therefore, instead of artificially separating *technologies* by putting markup and logic in separate files, React ***separates concerns*** with loosely coupled units called **"components"** that contain both.

React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code.

## Rendering Elements

Elements are the smallest building blocks of React apps. An element describes what we want to see on the UI.

*const element = <h1>Hello from React!</h1>*

Unlike browser DOM elements, React elements are plain *objects*, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

React elements are immutable, which means, once we create an element, we can't change its children or attributes.

React only updates what's necessary. React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the updated state.

## Rules of JSX

### 1. Return only a single root element

To return multiple elements from a component, wrap them with a single parent element. We can use either ***<div>*** or **React Fragments** (**<></>**).

Read more about Fragments in **Chapter-9 (Fragments)**.

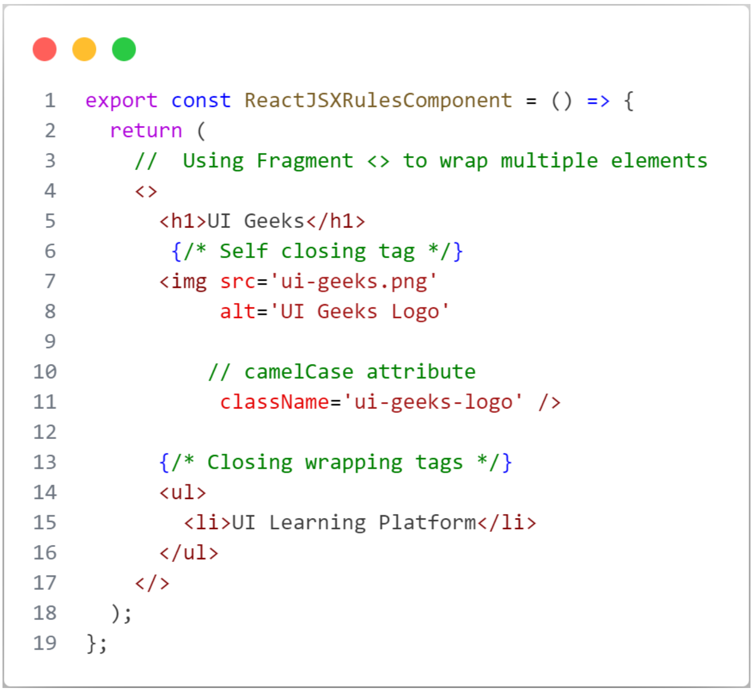
### 2. Close all the tags

JSX requires tags to be explicitly closed. Self closing tags like ***<img>*** must become ***<img />***, and wrapping tags like ***<li>*** must be written as ***<li></li>***.

### 3. camelCase all most all the attributes

After compilation, JSX is converted to JavaScript and attributes of JSX become the keys of the JavaScript *objects*. In our components, we may need to read those attributes, but JavaScript has limitations on variable names. For example, their names **can't contain dashes** or be reserved words like ***class***.

This is why, in React, many HTML and SVG attributes are written in **camelCase**. For example, instead of ***font-size*** we use ***fontSize*** and since class is a reserved word, we write ***className*** instead of ***class***.

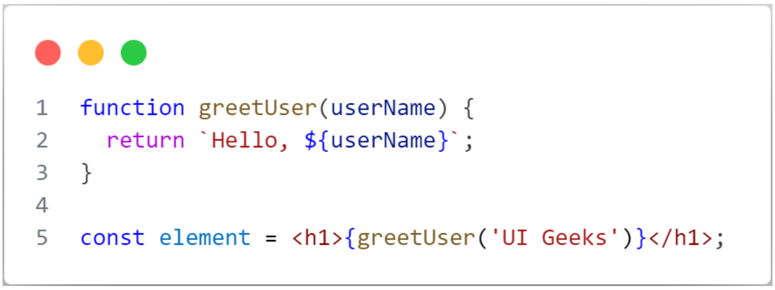


```
export const ReactJSXRulesComponent = () => {
  return (
    //  Using Fragment <> to wrap multiple elements
    <>
      <h1>UI Geeks</h1>
       {/* Self closing tag */}
      <img src='ui-geeks.png'
           alt='UI Geeks Logo'

          // camelCase attribute
           className='ui-geeks-logo' />

      {/* Closing wrapping tags */}
      <ul>
        <li>UI Learning Platform</li>
      </ul>
    </>
  );
};
```

## Embedding Expressions in JSX

Any valid JavaScript expression can be embedded inside the curly braces ({}) in JSX.

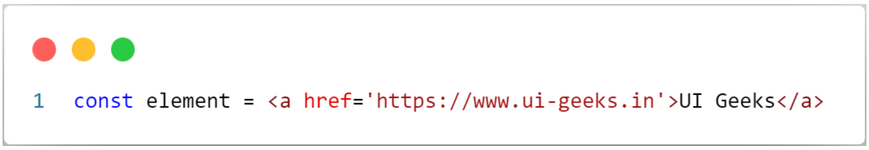
After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects. This means JSX can be assigned to variables, passed as an argument of a function or can be called inside loops, etc.



```
function greetUser(userName) {
  return `Hello, ${userName}`;
}

const element = <h1>{greetUser('UI Geeks')}</h1>;
```

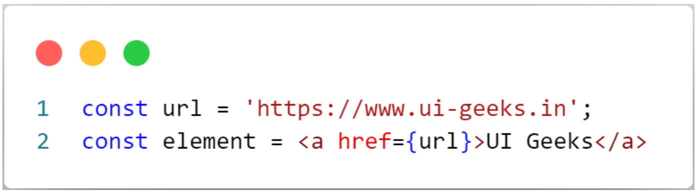
## Specifying Attributes with JSX

Double/Single quotes can be used to specify ***string literals*** as attributes.



```
const element = <a href='https://www.ui-geeks.in'>UI Geeks</a>
```

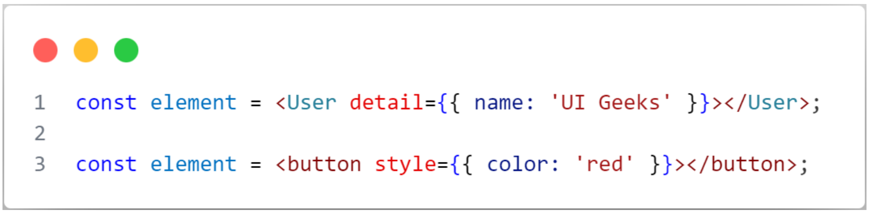
Also, ***curly braces*** can be used to embed *JavaScript expressions* as an attribute value.



```
const url = 'https://www.ui-geeks.in';
const element = <a href={url}>UI Geeks</a>
```

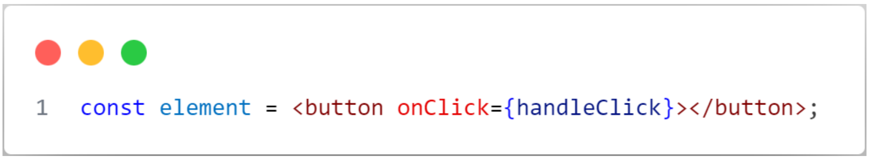
In addition to *strings*, *numbers*, and other *JavaScript types* and *expressions*, we can also pass ***objects*** as attributes value. We can use **double curly brackets {{ }}** to pass objects as a prop.

The common use cases of passing objects as props are to pass *API responses* to child components and to add *inline styles*.



```
const element = <User detail={{ name: 'UI Geeks' }}></User>;

const element = <button style={{ color: 'red' }}></button>;
```

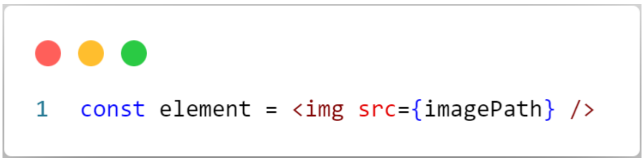
Since JSX is closer to JavaScript than HTML, React DOM uses ***cameCase*** property naming convention instead of HTML attribute names. For example, ***class*** becomes ***className*** in JSX and ***onclick*** becomes ***onClick***.



```
const element = <button onClick={handleClick}></button>;
```

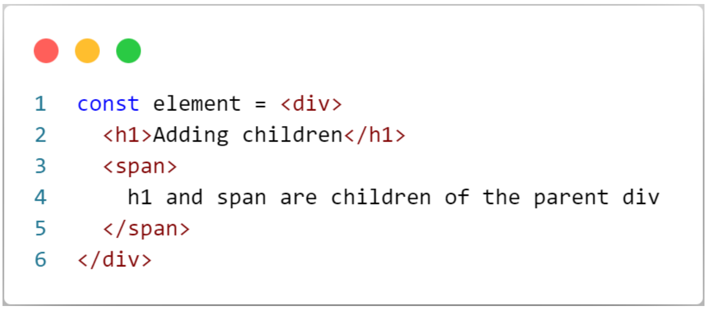
## Specifying Children with JSX

If a tag is empty, we may close it immediately with **/>**.



```
const element = <img src={imagePath} />
```

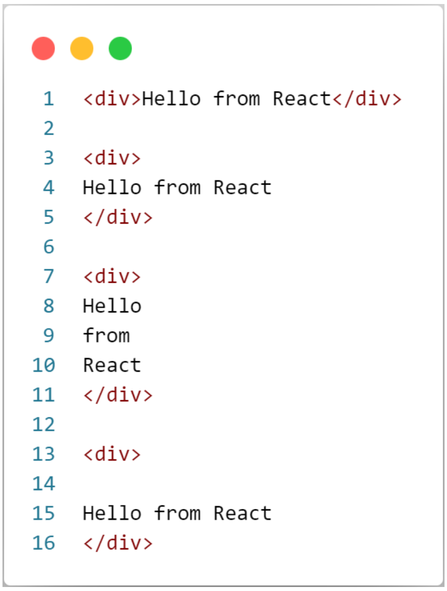
But JSX may contain children too, everything between opening and closing tag is considered as the children of the element.



```
const element = <div>
  <h1>Adding children</h1>
  <span>
    h1 and span are children of the parent div
  </span>
</div>
```

JSX removes *whitespaces* at the beginning and end of the line. It also removes *blank lines*. New lines adjacent to tags are removed. New lines that occur in the middle of *string literals* are condensed into a single space.

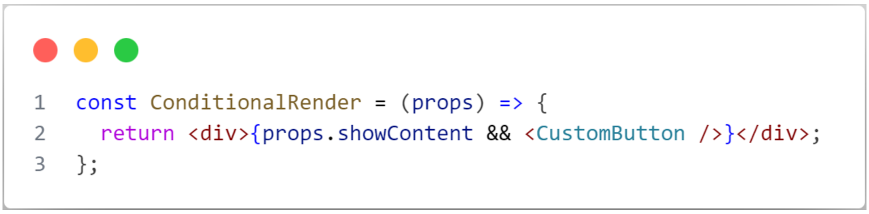
*Example*: All of the below <div> render the same output.



```
<div>Hello from React</div>

<div>
Hello from React
</div>

<div>
Hello
from
React
</div>

<div>

Hello from React
</div>
```

## Booleans, null and undefined are Ignored

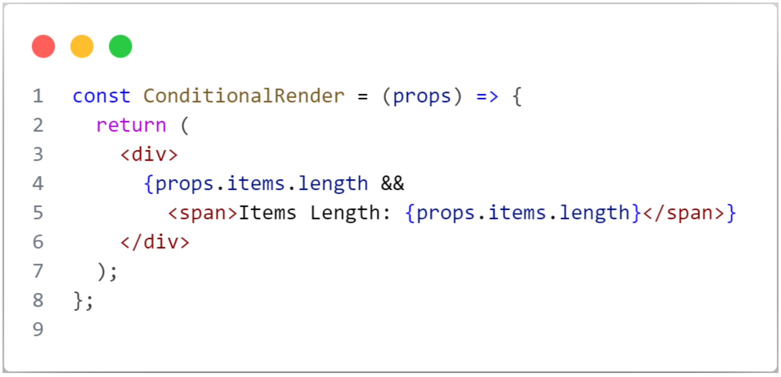
*false*, *null*, *undefined* and *true* are valid children but are **not rendered**. Although these values can be used for **conditionally rendering** React elements.



```
const ConditionalRender = (props) => {
  return <div>{props.showContent && <CustomButton />}</div>;
};
```

With the usage of JavaScript ***logical AND operator (&&)***, above code will render **CustomButton** component, only if ***showContent*** variable has ***truthy value***.

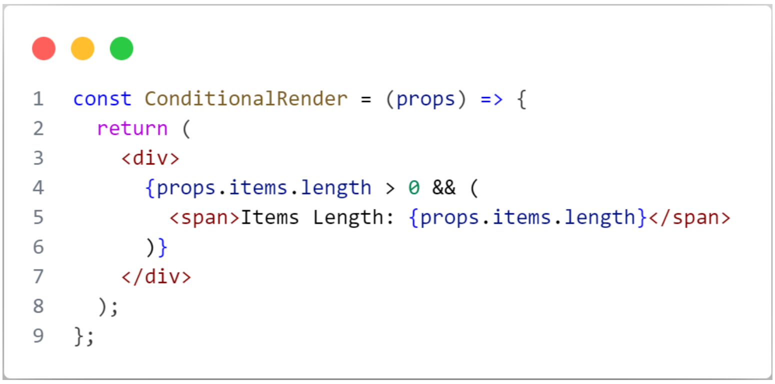
One caveat in conditional rendering is that some ***"falsy" values***, such as **zero (0)**, are still rendered by React.



```
const ConditionalRender = (props) => {
  return (
    <div>
      {props.items.length &&
        <span>Items Length: {props.items.length}</span>}
    </div>
  );
};

```

Above code will render ***<span>Item Length: 0</span>***.

To fix the above problem, make sure that the expression before ***&&*** is **always *boolean***.



```
const ConditionalRender = (props) => {
  return (
    <div>
      {props.items.length > 0 && (
        <span>Items Length: {props.items.length}</span>
      )}
    </div>
  );
};
```

Above code will not render anything, if ***props.items.length === 0***.

## JSX Prevents Injection Attacks

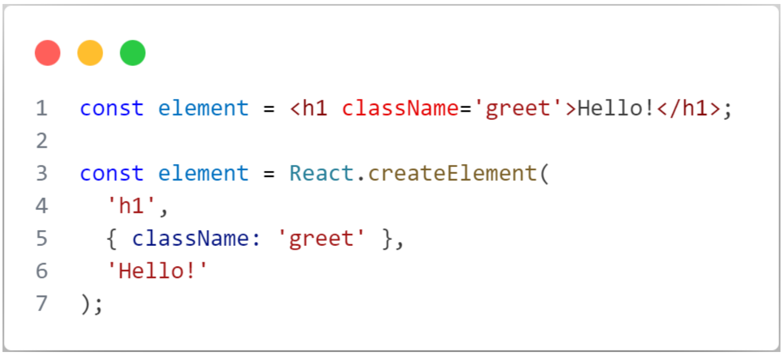
By default, React DOM *escapes* any values embedded in JSX before rendering them. Thus it ensures that we can **never inject** anything that's not explicitly written in the application.

Everything is converted to a *string* before being rendered. This helps prevent **XSS (Cross-Site-Scripting)** attacks.

## JSX Represents Objects

Babel compiles JSX down to ***React.createElement()*** calls and returns an *object*. These objects are called ***"React elements"***.

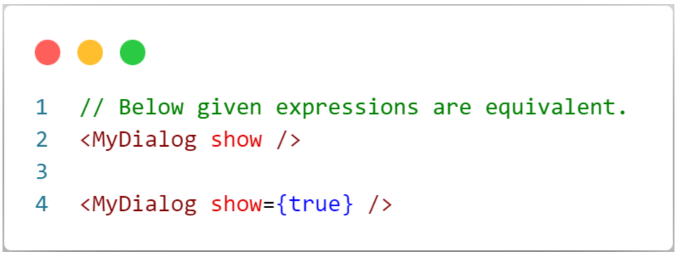
Below given example shows two representations of React Element. First is the JSX representation while later is the compiled representation of the ***element***.



```
const element = <h1 className='greet'>Hello!</h1>;

const element = React.createElement(
  'h1',
  { className: 'greet' },
  'Hello!'
);
```

## Props Default to "True"

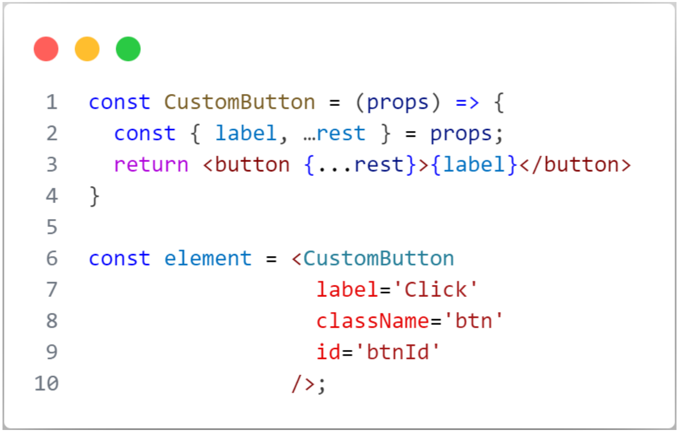
If we do not provide a value for a prop, it defaults to ***true***.



```
// Below given expressions are equivalent.
<MyDialog show />

<MyDialog show={true} />
```

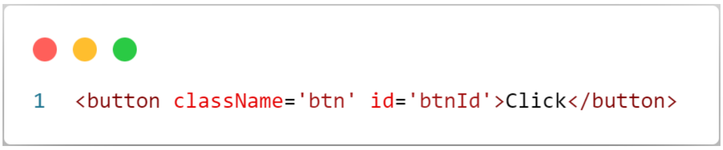
## Using Spread Attributes

If we have props as an *object*, and we need to pass all prop attributes to JSX, we can use JavaScript's spread operator (**...**).



```
const CustomButton = (props) => {
  const { label, …rest } = props;
  return <button {...rest}>{label}</button>
}

const element = <CustomButton
                  label='Click'
                  className='btn'
                  id='btnId'
                />;
```

Above code will render a button with given attributes.



```
<button className='btn' id='btnId'>Click</button>
```

## Summary

1. **JSX** is a syntax extension to JavaScript, which produces **React elements**.
2. React separates concerns with loosely coupled units called **components**.
3. JavaScript expressions can be embedded inside **curly braces ({})**.
4. **Double quotes** and **curly braces** can be used to add *attributes* to JSX.
5. ***Boolean***, ***null*** and ***undefined*** are ignored while rendering on UI.
6. JSX prevents **Injection Attacks**, because React DOM *escapes* embedded values.
7. **Babel** compiles JSX to *React.createElement()*, which returns an *object*.
8. **Props** default to ***true*** and *spread operators (...)* can be used to add multiple attributes.
9. **React elements** are *plain objects* and are cheap to create. React only updates the required changes to real DOM.

CHAPTER 3

# Components

# Preface to Components

## What is a Component

*Isolated, reusable building blocks of UI*

## Function Component

*JavaScript function that takes 'props' as input, returns react element*

## Class Components

*ES6 class extending React.Component to define a component, render()*

## Props

*Input values passed to the component, read-only*

## State

*Local to component, preserves values between renders*

## Lifecycle Phases

*Mounting, Updating and Unmounting phase*

## Render Props

*Sharing code using a prop whose value is a function*

## Components

Components let us split the UI into independent, reusable pieces, and think about each piece in isolation. Conceptually, components are like JavaScript *functions*. They accept arbitrary **inputs** (called ***"props"***) and return ***React elements***.

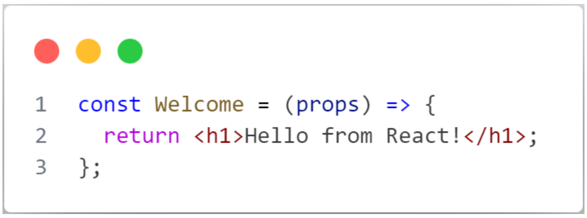
React provides two types of components:

1. **Function** Component
2. **Class** Component

### Function Component

This is the simplest way to define a component. A Function component is a JavaScript *function* that we can sprinkle with markup.

They are regular JavaScript functions, but their names **must start with a capital letter or they won't work**.



```
const Welcome = (props) => {
  return <h1>Hello from React!</h1>;
};
```

Above function is a valid React *component* because it accepts a single *input* ***"props"*** (which stands for **properties**) *object argument* and it returns a React element. Such components are called ***"function components"*** because they are literally JavaScript *functions*.

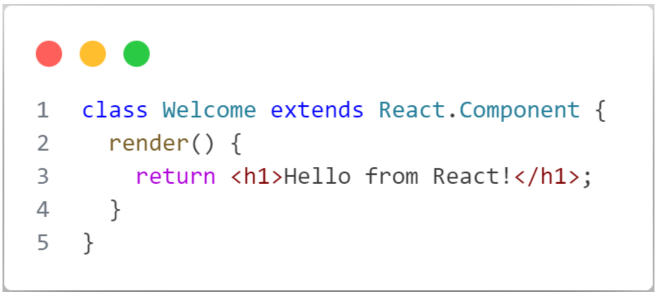
*Read more about Function Components in* **Chapter 5.**

## Class Component

We can also use ***ES6 class*** to define a component. The defined *class* should **extend *React.Component*** and should contain only one ***render()*** method. Class components are a bit slower as compared to *function components*, because we may need to write more lines of code to implement class construct.

### render()

This function returns the React element. It will be called on **component creation** (*Mounting phase*), along with **each update** to the component. Basically, it will be called whenever any change is made to the component.



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

If a component contains ***constructor()***, it will be called when the component gets initiated. We declare component properties in the *constructor()* function.

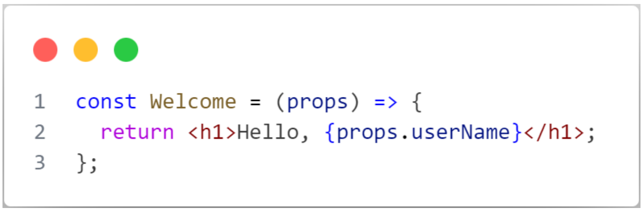
**Note**: Before **version 16.8**, only *class components* can be **stateful** components. *Function components* were used as **stateless** components. But with the introduction of ***Hooks*** in **v16.8**, *function components* can also be **stateful**. React prefers *function components* over *class components* for **performance benefits**.

**Note**: Always start component names with a ***Capital letter***. React treats components starting with ***lowercase letters*** as **DOM** tags. For example, ***<div />*** represents an HTML tag, but ***<Welcome />*** represents a component and requires **Welcome** to be in scope.

*Read more about Class Components in* **Chapter 4.**

## Props

Props (*stands for* **properties**) are the read-only input values passed into React components. It is similar to *arguments* of a function. It holds all the input values passed to the components. React components use *props* to communicate with each other. Let's add ***"prop"*** to our **Welcome** component.



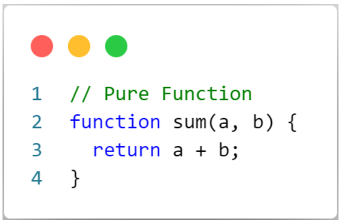
```
const Welcome = (props) => {
  return <h1>Hello, {props.userName}</h1>;
};
```

### Props are Read-Only

Whether we declare a component as *class* or *function*, *props* are **immutable**, it must never modify its own props. All React components must act like ***pure functions*** with respect to their props.

### What are pure functions?

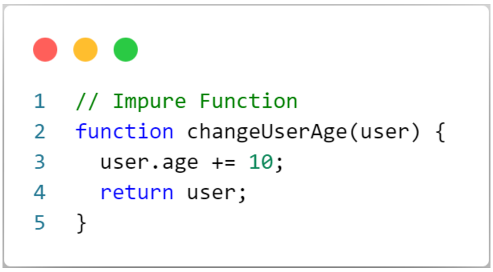
Functions which do not modify the value of their inputs are called ***"pure"*** functions.



```
// Pure Function
function sum(a, b) {
  return a + b;
}
```

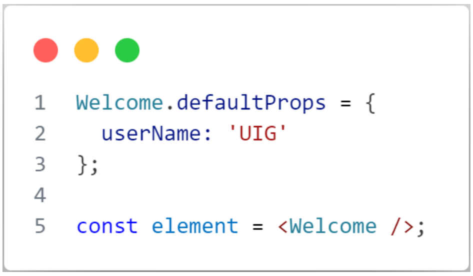
Above function **"sum"** is a pure function, because it did not modify the input values, which assures that this function will always return the same result for the same inputs.

In contrast to this, ***"impure" functions*** are ones which modify the values of their inputs. Below function ***"changeUserAge"*** is an impure function, since it modifies the input value **"user"**.



```
// Impure Function
function changeUserAge(user) {
  user.age += 10;
  return user;
}
```

A *class component* can have ***defaultProps*** defined on the *class*, to set the default values of the *props,* in case *prop* values are *undefined*.



```
Welcome.defaultProps = {
  userName: 'UIG'
};

const element = <Welcome />;
```

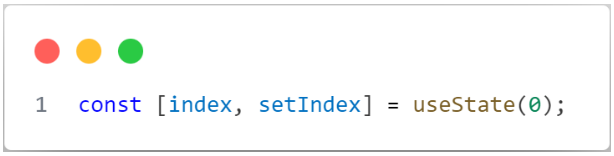
*Read more on **defaultProps** in* **Chapter 4 (Class Component)**

## State

State is a way to **“preserve”** values between multiple renders. *State* is ***local*** to the component, which means the *state* of a component can’t be accessed by another component, unless *state* value is passed as **“prop”** to another component.

To update a component with new data, we would need to retain the values between renders and inform React whenever data gets changed. The ***state*** plays the role of retaining the values, while the ***setter method*** informs the React to re-render.

For example, in the below *function component* ***useState()*** hook is used to create a *state* variable. Variable **index** retains the value while the setter method **setIndex()** updates value and informs React to re-render.



```
const [index, setIndex] = useState(0);
```

## Unidirectional or top-down data flow

Neither parent nor child components can know if a certain component is stateful or stateless. This is why the *state* is often called local or encapsulated. It is not accessible to any other component other than the one that owns and sets it. A component may choose to pass its state as props to its child components.

This is commonly called a **“top-down”** or **“unidirectional”** data flow. Any *state* is always owned by some specific component, and any data or UI derived from that *state* can only affect components **“below” them in the tree**.

## Component Lifecycle

Each component has several “lifecycle methods” that we can override to run code at particular times in the process.

### Phases of a component

Every component goes through three phases:

1. Mounting
2. Updating
3. Unmounting

#### Mounting Phase

Mounting is the phase where the component is created and inserted into the DOM.

#### Updating Phase

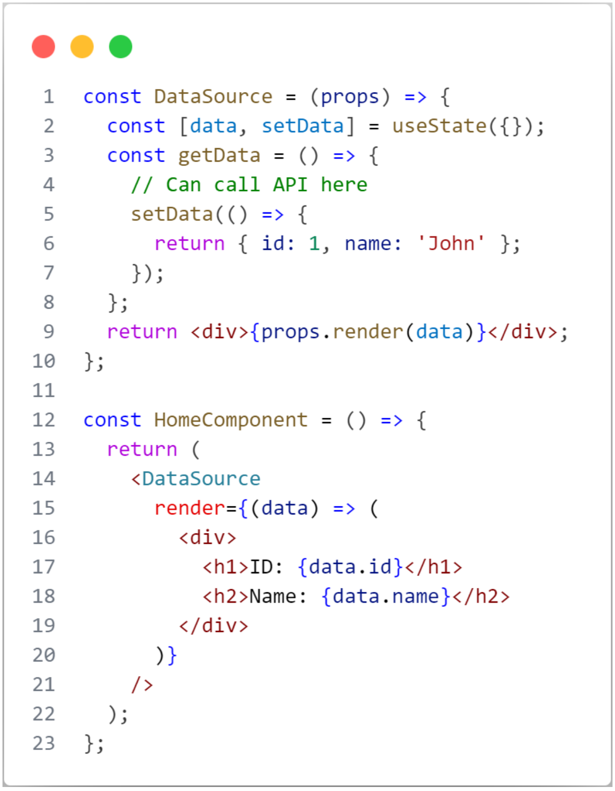
An update can be caused by changes to ***props*** or ***state***. Whenever the value of *props* or *state* is changed, React **re-renders** the component to reflect updated values on the UI. The lifecycle methods of the update phase are called on every update.

#### Unmounting Phase

Unmounting is the phase where a component is removed from the DOM. Components go out of scope and the lifecycle method of unmounting phase can be used to perform **cleanup activities**.

## Render Props

The term **"render prop"** refers to a technique for sharing code between components using a *prop* whose value is a *function*. A component with a render prop takes a *function* that returns a React element and calls it instead of implementing its own logic. It is called "render prop" because it's a *prop* that specifies how to render something.

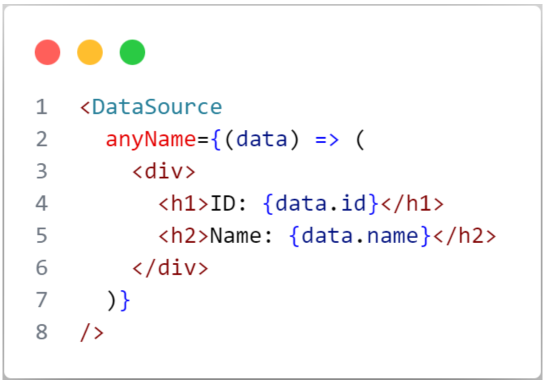


```
const DataSource = (props) => {
  const [data, setData] = useState({});
  const getData = () => {
    // Can call API here
    setData(() => {
      return { id: 1, name: 'John' };
    });
  };
  return <div>{props.render(data)}</div>;
};

const HomeComponent = () => {
  return (
    <DataSource
      render={(data) => (
        <div>
          <h1>ID: {data.id}</h1>
          <h2>Name: {data.name}</h2>
        </div>
      )}
    />
  );
};
```

In the above example, the **DataSource** component will do the data manipulation and provide the ***data*** to the passed *prop function* **"render"**. In this way, the state of the **DataSource** component can be reused in any component.

More concretely, a *render prop* is a *function prop* that a component uses to know what to render.

It's important to remember that just because the pattern is called **"render props"** we don't have to use a prop named **"render"**. In fact, **any name can be used** for defining the *render props*.



```
<DataSource
  anyName={(data) => (
    <div>
      <h1>ID: {data.id}</h1>
      <h2>Name: {data.name}</h2>
    </div>
  )}
/>
```

**Note**: Using a *render prop* can negate the advantage that comes from using **React.PureComponent** if we create the function inside a *render* method. Because the shallow prop comparison will always return *false* for new props, and each *render* in this case will generate a new value for the render prop.

## Summary

1. **Components** split the UI into *independent*, *reusable pieces*. Components accept input ***props*** and return *React elements*.
2. React provides **Function Components** and **Class Components**.
3. Component **names** should start with a **Capital** letter.
4. **Class component** uses *ES6 class*, which is **extended** by ***React.Component***. It should contain a ***render()*** function. The ***"this.props.children"*** is a special prop holding the component's children.
5. ***Props*** is the input object, these are **read-only** properties.
6. *Pure functions* **do not** modify input values while *impure functions* do.
7. ***State*** is **local** to components, which **preserve** values between multiple renders.
8. React follows **uni-directional** data flow, where ***state*** is passed from *parent* to *child* component.
9. Every component goes through the *Mounting*, *Updating* and *Unmounting* Phase.
10. **Render props** is a technique to *share code* between components using a *prop* which is a *function*.
11. It is not required to name the prop as **"render"**, instead any valid names could be used.

*CHAPTER 4*

# Class Component

## Preface to Class Component

### Define Class Component

*Define Class component by extending ES6 class with React.Component*

### Props to Class Component

*Input values passed to the component, read-only*

### State

*Local to component, preserves values between renders*

### Component Class Properties

*defaultProps, displayName properties*

### Component Lifecycle Methods

*constructor, componentDidMount, render, etc*

### Error Handling Methods

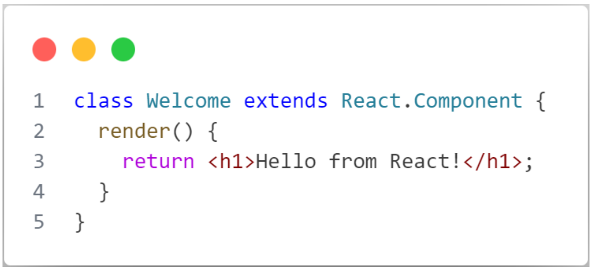
*componentDidCatch, getDerivedStateFromError()*

### Error Boundary

*React Class Component that catch javascript errors*

## Class Component

We can also use ***ES6 class*** to define a component. The defined *class* should **extend *React.Component*** and should contain only one ***render()*** method. Class components are a bit slower as compared to *function components*, because we may need to write more lines of code to implement class construct.



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

### render()

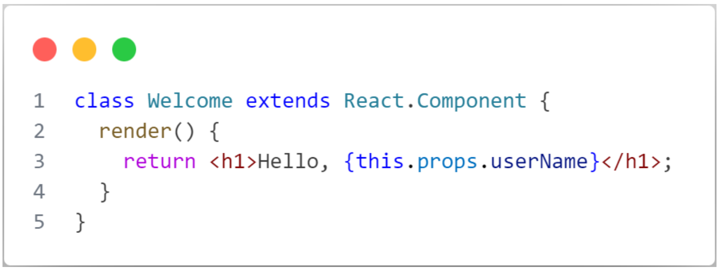
This function returns the React element. It will be called on **component creation** (*Mounting phase*), along with **each update** to the component. Basically, it will be called whenever any change is made to the component.

If a component contains ***constructor()***, it will be called when the component gets initiated. We declare component properties in the *constructor()* function.

### Props

Props are the input values passed to the component. It holds all the input values passed to the components.

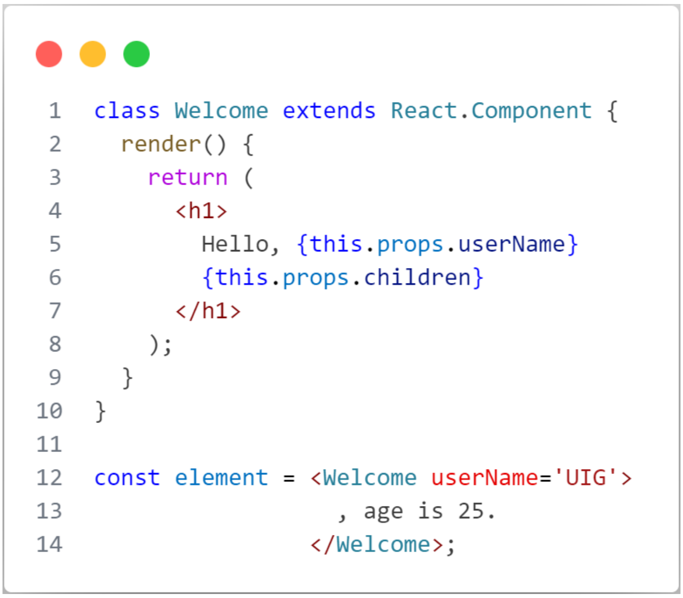
Let's add ***"prop"*** to our **Welcome** component. In the case of a ***class component***, **"this"** refers to the *class context*, therefore, ***"this.props"*** points to the input values of the *class component*.



```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.userName}</h1>;
  }
}
```

Passing input ***props*** to the component.
*const element = <Welcome **userName**="John" />;*
*const element1 = <Welcome **userName**="Jack" />;*

In particular, *this.props.children* is a special prop, typically defined by the child tags in the JSX expression rather than the tag itself.



```
class Welcome extends React.Component {
  render() {
    return (
      <h1>
        Hello, {this.props.userName}
        {this.props.children}
      </h1>
    );
  }
}

const element = <Welcome userName='UIG'>
                  , age is 25.
                </Welcome>;
```

Above will render **"Hello, UIG, age is 25."**, because everything between the opening and closing tags becomes the value of ***this.props.children***.

### Props are Read-Only

Whether we declare a component as class or function, it must never modify its own props. All React components must act like ***pure functions*** with respect to their props.

## State

State is a way to **"preserve"** values between multiple renders. *State* is ***local*** to the component, which means the *state* of a component can't be accessed by another component, unless *state* value is passed as **"prop"** to another component.

Neither parent nor child components can know if a certain component is stateful or stateless. This is why the *state* is often called local or encapsulated. It is not accessible to any other component other than the one that owns and sets it. A component may choose to pass its *state* as *props* to its child components.

This is commonly called a **"top-down"** or **"unidirectional"** data flow. Any *state* is always owned by some specific component, and any data or UI derived from that *state* can only affect components **"below" them in the tree**.

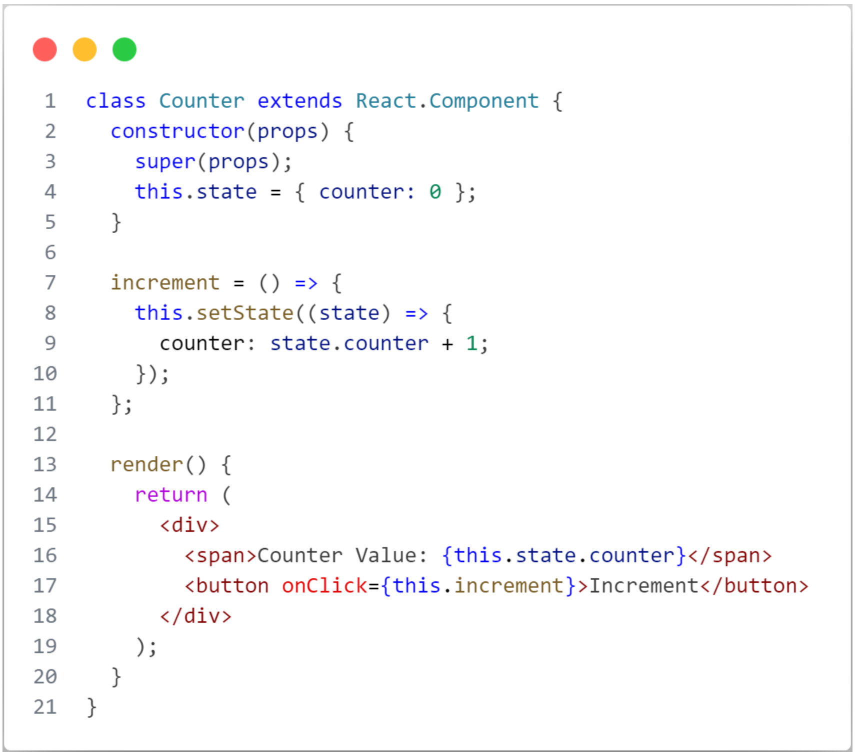
### Adding State To Class Component

To update a component with new data, we would need to retain the values between renders and inform React whenever data gets changed. The ***state*** plays the role of retaining the values, while the ***setter method*** updates the state and informs the React to re-render.

The ***this.state*** variable holds the *state values*. Let's try to understand the *state* with an example.

Below **Counter** component is a *class* component which uses the *state variable* to **“preserve”** values between multiple renders. The ***this.state*** should always be an object.

For example, in the below *component* ***this.state*** used to create a *state* variable. Variable **counter** retains the value while the setter method **this.setState()** updates value and informs React to re-render.



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

  increment = () => {
    this.setState((state) => {
      counter: state.counter + 1;
    });
  };

  render() {
    return (
      <div>
        <span>Counter Value: {this.state.counter}</span>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

***onClick={this.incrementCounter}***

This adds the ***“click”*** event to the button here. The method "***this.increment()***" will be called once the user clicks the button.

**super()**: It is required to pass ***"props"*** to the base class constructor (i.e, constructor of React.Component). The ***super()*** call initializes the ***"this"*** of our component.

***this.state = { }***

Above statement initializes the *state* of the component. **Constructor** is the only place where we can ***initialize*** the *state*, all other places should only update the *state* values.

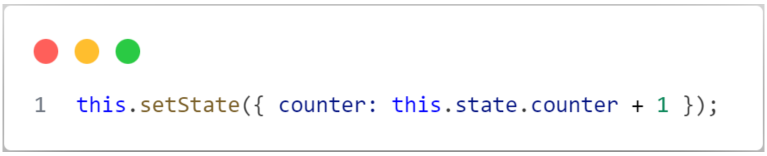
***this.setState()***

Any **update** to the state value should be made by *this.setState()* method. This method updates the value of state and ***triggers*** a ***re-render*** for the component.

The state updates may be **batched** by *React* for **performance benefits**, which means it is not guaranteed the next state update will get the updated state values. But often there are use-cases where we need updated value before calling setState().

To support this, ***this.setState()*** has **multiple** syntaxes. Let's learn about multiple syntaxes of this.setState() method.

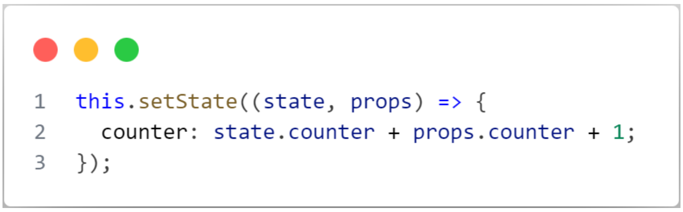
*Syntax 1*:



```
this.setState({ counter: this.state.counter + 1 });
```

This is a simpler syntax for updating the state value. But ***do not guarantee*** the updated state value.

Syntax 2:

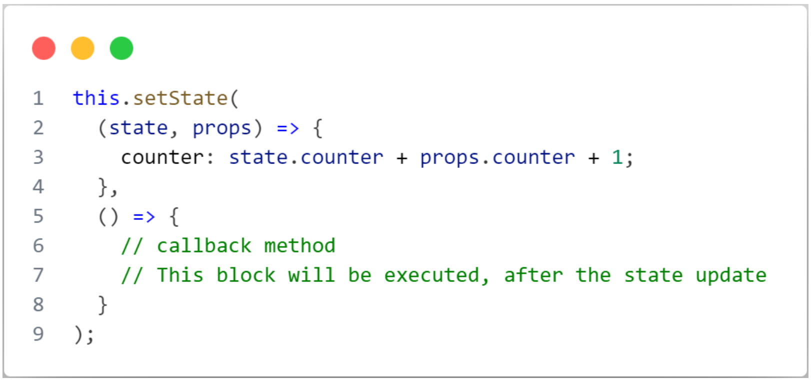


```
this.setState((state, props) => {
  counter: state.counter + props.counter + 1;
});
```

This uses a *function syntax* and provides the last updated value of ***state*** and ***props*** as input to the *setState()* call.

Syntax 3:

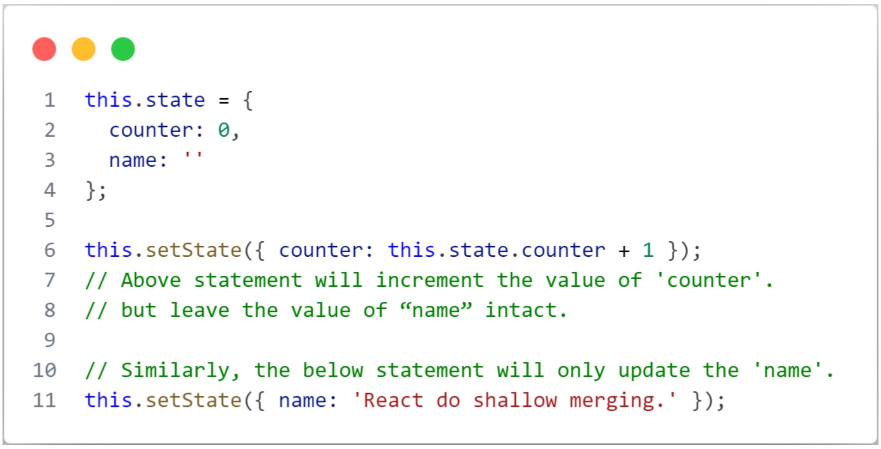
There could be use-case, where we need to call other methods/statements after the state has been updated. For such use-cases, *setState()* has an ***optional second argument***, which is a **"callback"** method. This *"callback"* is called once the state update is completed.



```
this.setState(
  (state, props) => {
    counter: state.counter + props.counter + 1;
  },
  () => {
    // callback method
    // This block will be executed, after the state update
  }
);
```

When we call *setState()*, React merges the object we provide into the current state. The merging is shallow, which means it will only update the given value, leaving other state values intact.

Let's say our component has **2 state variables**, and when we update one variable, it will not affect the other variable and vice-versa.

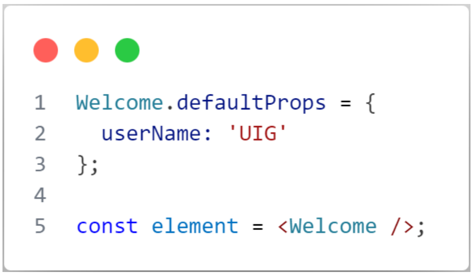


```
this.state = {
  counter: 0,
  name: ''
};

this.setState({ counter: this.state.counter + 1 });
// Above statement will increment the value of 'counter'.
// but leave the value of “name” intact.

// Similarly, the below statement will only update the 'name'.
this.setState({ name: 'React do shallow merging.' });
```

## Component Class Properties

### defaultProps

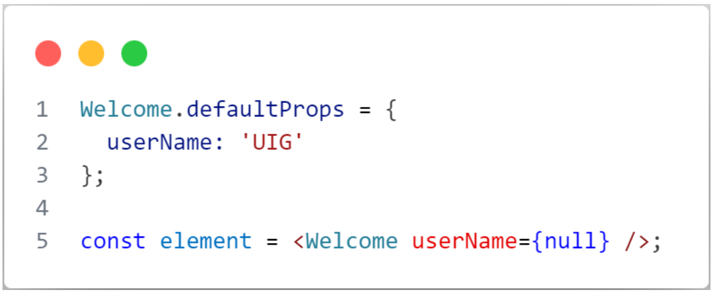
It can be defined as a ***static*** property on the component class itself, to set the *default props for the class*. This is used for ***undefined*** and missing *props*, but not for ***null*** *props*.



```
Welcome.defaultProps = {
  userName: 'UIG'
};

const element = <Welcome />;
```

Above will render **“Hello, UIG”**, because we didn’t pass the value of ***userName*** at the time of calling component, therefore it will read the value from ***defaultProps***.

The *defaultProps* do not work for ***null*** values.



```
Welcome.defaultProps = {
  userName: 'UIG'
};

const element = <Welcome userName={null} />;
```

Above will render **"Hello,"**, because we passed the value of ***userName*** as ***null*** at the time of calling component, therefore it will ignore the value of ***defaultProps***.

**Note**: Defining ***defaultProps*** in *class* components is similar to using **default values** in *function* components.
*const Welcome = ({ userName = 'UIG' }) => { }*

## displayName

The ***displayName*** string is used in debugging messages. Usually, we don't need to set it explicitly because it's inferred from the name of the *function* or *class* component.

We may want to set it explicitly if we want to display a different name for debugging purposes or when we create a higher-order component.

*Example*:
*Welcome.displayName = "Welcome Component";*

## Component Lifecycle

Each component has several “lifecycle methods” or phases that we can override to run code at particular times in the process. Let’s learn about the phases and respective lifecycle methods of a class component.

### Phases of a component

Every component goes through three phases:

4. Mounting
5. Updating
6. Unmounting

#### Mounting Phase

Mounting is the phase where the component is created and inserted into the DOM.

#### Updating Phase

An update can be caused by changes to ***props*** or ***state***. Whenever the value of *props* or *state* is changed, React **re-renders** the component to reflect updated values on the UI. The lifecycle methods of the update phase are called on every update.

#### Unmounting Phase

Unmounting is the phase where a component is removed from the DOM. Components go out of scope and the lifecycle method of unmounting phase can be used to perform **cleanup activities**.

## Lifecycle Methods

Lifecycle methods can be grouped by the phases. Methods marked in **bold** are commonly used methods.

- Mounting
  - **constructor()**
  - static getDerivedStateFromProps()
  - **render()**
  - **componentDidMount()**

- Update
  - static getDerivedStateFromProps()
  - shouldComponentUpdate()
  - **render()**
  - getSnapshotBeforeUpdate()
  - **componentDidUpdate()**

- Unmounting
  - **componentWillUnmount()**

- Error Handling
  - static getDerivedStateFromError()
  - componentDidCatch()

Lets learn about each lifecycle method in detail.

## Mount LifeCycle Methods

### constructor()

The *constructor* for a React component is called before it is mounted. When implementing the constructor, we should call ***super(props)*** before any other statement. Otherwise, ***"this.props"*** will be ***undefined*** in the *constructor*.

Avoid introducing any *side-effects* or *subscriptions* in the constructor.

In React, *constructors* are only used for two purposes:

- Initializing **local state** by assigning an object to this.state.
- Binding **event handler** methods to an instance.

The *constructor* is the only place where we should assign ***this.state*** directly. In all other places, we should use *this.setState()*.

**Note**: *constructor* is ***optional***. If we neither initialize state nor bind methods, then we don't need to implement a constructor.

### static getDerivedStateFromProps()

It is invoked right before calling the *render()* method, both on the **initial mount** and on **subsequent updates**. It should ***return an object*** to update the *state*, or ***null*** to update nothing.

*static getDerivedStateFromProps(props, state)*

This method exists for *rare use cases*, where the ***state*** depends on changes in ***props***. This method doesn't have access to the component instance. It is fired on every render, regardless of the cause.

### render()

The render() method is the only **required** method in a *class* component. When called, it should examine ***this.props*** and ***this.state*** and return one of the following types:

- **React elements**
  Any valid JSX element.

- **Array and fragments**
  Let us **return multiple elements** from the render. By default *render()* method should return only **one element**, but ***fragments*** can be used to **combine/wrap** multiple elements that can be returned from the *render()* method. Read more about Fragments in **Chapter 9 (Fragments)**.

- **Portal**
  Portals help render children into a ***different DOM subtree***.

- **String and numbers**
  These are rendered as **text nodes** in the DOM.

- **Booleans or null**
  Renders **nothing**. Most used to implement *conditional rendering*, for example: isTrue && <Welcome />.

The *render()* method should be pure, meaning that it does **not modify state**, it returns the same result each time it's invoked, and it does not directly interact with the browser.

**Note**: ***render()*** will not be invoked if ***shouldComponentUpdate()*** returns *false*.

### componentDidMount()

It is invoked immediately after a component is mounted. **Initialization** that **requires DOM nodes** should be written here, for example, setting tooltips where we need DOM nodes before rendering to get its placement . It is also a good place to **make API calls** or **set up any subscriptions**.

**Note**: We may call setState() immediately in componentDidMount(). It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the render() will be called twice, the user won't see the intermediate state. Use this pattern with caution because it often causes performance issues.

## Update LifeCycle Methods

Below are the lifecycle methods which are called in the update phase of the component.

### shouldComponentUpdate()

This method is used to **skip the re-rendering** of the component. This method should ***return a boolean value***. If it returns ***false***, then **UNSAFE_componentWillUpdate()**, **render()** and **componentDidUpdate()** methods are not invoked.

*shouldComponentUpdate(nextProps, nextState)*

The default behavior is to re-render on every change in *state* or *props*. We can use this  method to let React know if the output is not affected by the current change in *state* or *props*. React prefers to rely on default behavior in the majority of the cases.

It is invoked **before rendering** when **new** *props* or *state* are being received. Defaults to **true**. This method is not called for the *initial render* or when *forceUpdate()* is used.

**Note**: Returning ***false*** does not prevent *child components* from re-rendering when ***their*** *state* changes.

### getSnapshotBeforeUpdate()

It is invoked right before the most recently rendered output is committed to the DOM. It enables components to capture some information from the DOM (like scroll position) before it is potentially changed.

*getSnapshotBeforeUpdate(prevProps, prevState)*

Any value returned from this method will be passed as a parameter to the ***componentDidUpdate()***. It should return either a ***snapshot value*** or ***null***.

### componentDidUpdate()

It is invoked immediately after updating occurs. This method is not called for the initial render.

*componentDidUpdate(prevProps, prevState, snapshot)*

**snapshot**: If the component implements the ***getSnapshotBeforeUpdate()*** lifecycle, the value it returns will be the value of the third parameter **"snapshot"**, otherwise this parameter will be ***undefined***. This method can be used to operate on the DOM when the component has been updated or calling API on change of props.

**Note**: We may call *setState()* immediately in *componentDidUpdate()* but **it must be wrapped in a condition** or it will cause an **infinite loop**.

**Note**: *componentDidUpdate()* will not be invoked if **shouldComponentUpdate()** returns **false**.

## Unmounting LifeCycle Method

Below is the lifecycle method which is called in the unmount phase of the component.

### componentWillUnmount()

It is invoked immediately before a component is ***unmounted*** and ***destroyed***. This method is used to perform **cleanup tasks** like, invalidating timers, canceling network calls or for unsubscribing from the subscriptions.

Once a component is unmounted, it will never be mounted again. We should not call *setState()* in *componentWillUnmount()* because the component will never be re-rendered.

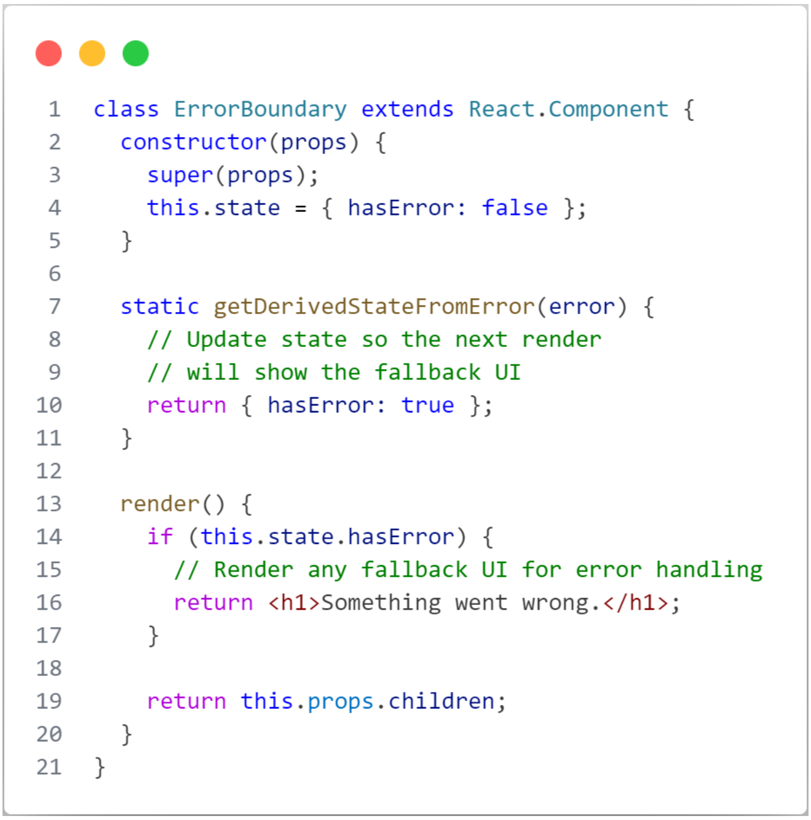
## Error Handling Methods

These methods are called when there is an error during rendering, in a lifecycle method, or in the *constructor* of any child component.

## static getDerivedStateFromError()

It is invoked after an error has been thrown by a **descendant component**. It receives the error that was thrown as a parameter and should return a value to update state.

*static getDerivedStateFromError(error)*



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

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

  render() {
    if (this.state.hasError) {
      // Render any fallback UI for error handling
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

**Note**: *getDerivedStateFromError()* is called during the **"render"** phase, so *side-effects* are not permitted. For those use cases, use **componentDidCatch()** instead.

### componentDidCatch()

It is invoked after an error has been thrown by a child/descendant component. Typically, this is used together with *static getDerivedStateFromError()* to update state on error and display a fallback UI to the user. It is called during the "*commit*" phase, so side-effects are permitted. It should be used for things like logging the errors.

*componentDidCatch(**error**, **info**)*

**error**: Error thrown from the component.

**info**: An *object* with a ***componentStack*** key containing information about which component threw the error.

Production and development builds of React slightly differ in the way *componentDidCatch()* handles errors.

On **development**, the errors will ***bubble up*** to the ***window***, this means that any ***window.onerror*** or ***window.addEventListener('error', callback)*** will intercept the errors that have been caught by *componentDidCatch()*.

On **production**, the errors will ***not bubble up***, which means any ancestor error handler will only receive errors ***not explicitly caught*** *by componentDidCatch()*.

## Error Boundary

By default, if the application throws an error during rendering, it corrupts React's internal state and causes it to *emit cryptic errors* on the next render, in response to which, React removes its UI from the screen.

React did not provide a way to handle them gracefully in components and could not recover from them. But for better user experience, a JavaScript error in a part of the UI shouldn't break the whole application.

To solve this problem, **React 16** introduced the concept of an **"error boundary"**. Error boundaries are React *class* components that **catch JavaScript errors** anywhere in their child component tree, **log those errors**, and **display a fallback UI** instead of the component tree that crashed.

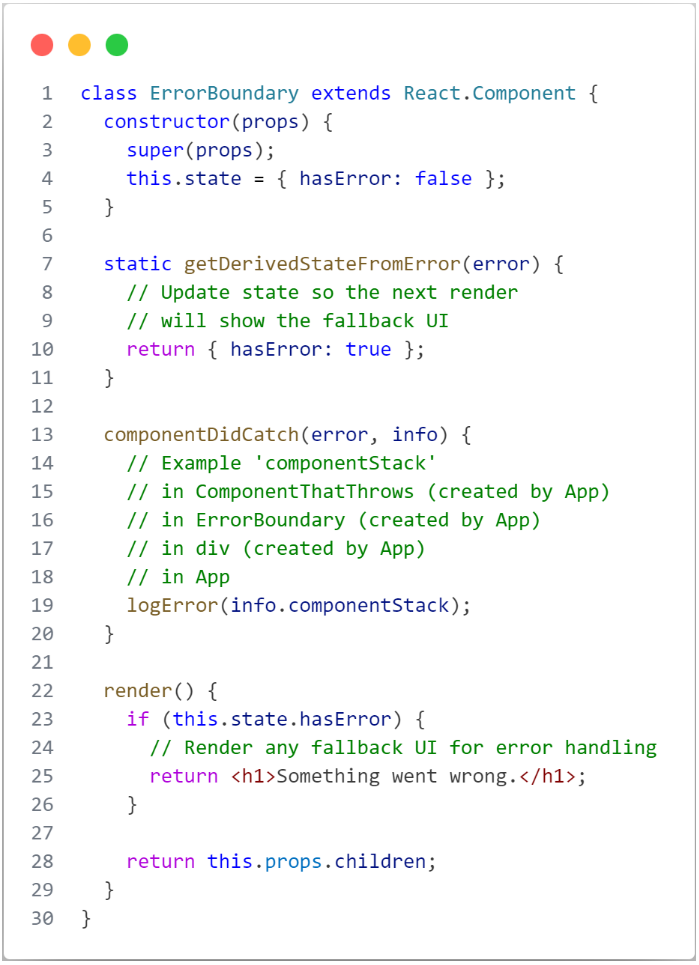
Error boundaries catch errors during rendering, in lifecycle methods, and in *constructors* of the whole tree below them.

Error boundaries **do NOT** catch errors for:
- Event Handlers
- Asynchronous code (like *setTimeout*)
- Server side rendering
- Error thrown in the error boundary itself

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods:
***static getDerivedStateFromError()*** or ***componentDidCatch()***.

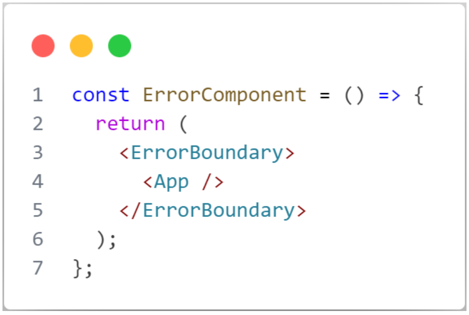
Use *static getDerivedStateFromError()* to render fallback UI after an error and *componentDidCatch()* to log error information.

**Note**: Only **class components** can be used as Error Boundary, since **function components** do not have any hook for catching errors.



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

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

  componentDidCatch(error, info) {
    // Example 'componentStack'
    // in ComponentThatThrows (created by App)
    // in ErrorBoundary (created by App)
    // in div (created by App)
    // in App
    logError(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // Render any fallback UI for error handling
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Error Boundary can be used as a regular component wrapping the child components.



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

## Summary

1. **Class component** uses *ES6 class*, which is **extended** by ***React.Component***. It should contain a ***render()*** function.
2. The ***"this.props.children"*** is a special *prop* holding the component's children.
3. The ***defaultProps*** is set on the *class* itself, to set the **default values** for ***undefined*** *props* but not for ***null*** values.
4. The ***displayName*** is used to give a descriptive name to components for debugging purposes.
5. The **lifecycle methods** are used to run code at particular times in the rendering process.
6. The ***constructor()*** called before *mount*, should call ***super(props)*** to initialize **"this.props"**.
7. ***static getDerivedStateFromProps()*** called before *render* on **initial** and **subsequent updates**.
8. The ***render()*** is the only **required** method, should return *React elements*, *Arrays/fragments*, *Portal*, *string/number*, *boolean* or *null*.
9. The ***componentDidMount()*** is called after **mount**. DOM manipulation, API calls should be made here.
10. The ***shouldComponentUpdate()*** **skips** the rendering if this method returns ***false***.
11. The ***componentDidUpdate()*** is invoked after every update.
12. The ***componentWillUnmount()*** is called when the component is **unmounted** or destroyed. This is used for **clean up** activities.

13. The ***getDerivedStateFromError()*** and ***componentDidCatch()*** are used for handling errors.

14. **Error boundaries** are *class components* that **catch JavaScript errors**. Errors can be logged and a **fallback UI** can be displayed.

15. Error boundaries do not catch errors for: *Event handlers, asynchronous code, server side rendering* and in error boundary itself.

16. A class component becomes an error boundary if it defines ***getDerivedStateFromError()*** or ***componentDidCatch()*** or both.

*CHAPTER 5*

# Function Component

# Preface to Function Component

## Define Function Components

*Define Function component using JavaScript functions*

## Props to Function Component

*Input values passed to the component, read-only*

## Adding State to Component

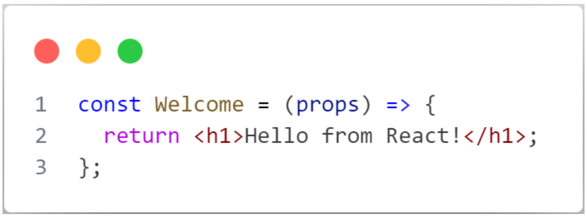
*Local state object of the component, preserves values between renders, useState()*

## Lifecycle Hooks

*useEffect()*

## Function Component

This is the simplest way to define a component.



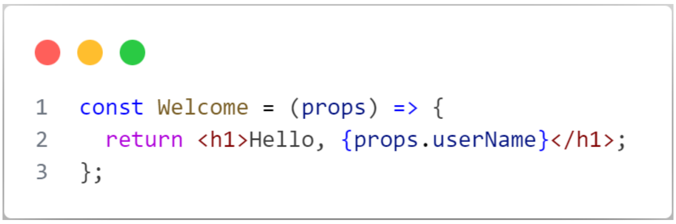
```
const Welcome = (props) => {
  return <h1>Hello from React!</h1>;
};
```

Above function is a valid React component because it accepts a single *input* ***"props"*** (which stands for **properties**) *object argument* and it returns a React element. Such components are called ***"function components"*** because they are literally JavaScript *functions*.

### Props

Props are the input object to the component. It holds all the input values passed to the components.

Let's add ***"prop"*** to our **Welcome** component.



```
const Welcome = (props) => {
  return <h1>Hello, {props.userName}</h1>;
};
```

### Props are Read-Only

Whether we declare a component as class or function, it must never modify its own props. All React components must act like ***pure functions*** with respect to their props.

## State

State is a way to **“preserve”** values between multiple renders. *State* is ***local*** to the component, which means the *state* of a component can’t be accessed by another component, unless *state* value is passed as **“prop”** to another component.

Neither parent nor child components can know if a certain component is stateful or stateless. This is why the *state* is often called local or encapsulated. It is not accessible to any other component other than the one that owns and sets it. A component may choose to pass its *state* as *props* to its child components.

### Adding State To Function Component

With the introduction of ***Hooks***, *function components* can also have *state*. The ***useState*** is the *hook* used to add *state* to the *function* component.

> **Note**: A hook is a special *function* that lets us “**hook into**” React features. For example: ***useState()*** is a Hook that adds *state* to function components. Read more about hooks in **Chapter 10 (Hooks)**

*Syntax*:
*const [**stateValue**, **updateFunction**] = **useState**(**initialValue**);*

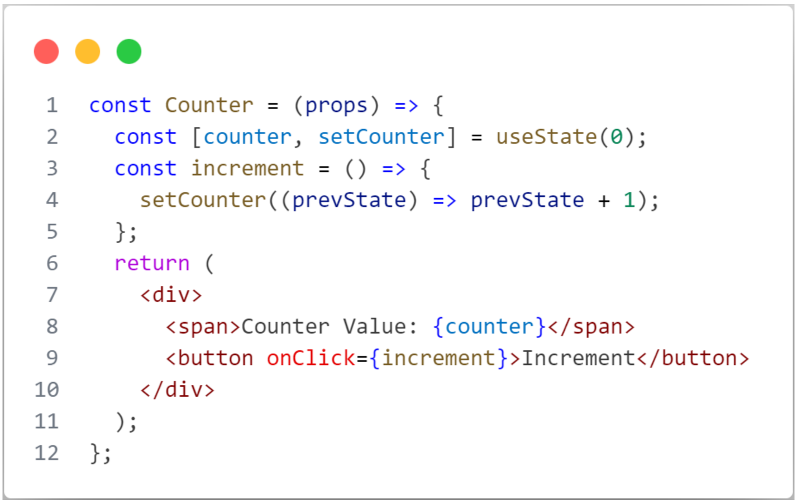
**useState:** This is the hook used to add *state* to a function component. It returns a **pair** - an *array* with two items. The first item is the current value and second is a function that updates it.

**stateValue:** It is the first value of the pair of values returned from the *useState()* function. This is the name of the state variable. In contrast to the class component's *state* being an *object*, the *state* of a function component can be any valid *type*. For example, it can be *numbers*, *strings*, *arrays*, etc.

**updateFunction:** This is the second value of the pair of values returned from the *useState()* function. This function is equivalent to *setState()* of the class component and used to update the respective *state* variable.

**initialValue:** It is the initial value of the *state* variable. The initial value is only assigned once, on the mount of the component and is ignored on further update events.

Let's build our Counter component using the **function component** and **useState()** *hook*.



```
const Counter = (props) => {
  const [counter, setCounter] = useState(0);
  const increment = () => {
    setCounter((prevState) => prevState + 1);
  };
  return (
    <div>
      <span>Counter Value: {counter}</span>
      <button onClick={increment}>Increment</button>
    </div>
  );
};
```

Above code is much simpler as compared to the *class component* equivalent.

## Component Lifecycle

Each component has several “lifecycle methods” or phases that we can override to run code at particular times in the process. Let’s learn about the phases and respective lifecycle methods of a class component.

### Phases of a component

Every component goes through three phases:

7. Mounting
8. Updating
9. Unmounting

#### Mounting Phase

Mounting is the phase where the component is created and inserted into the DOM.

#### Updating Phase

An update can be caused by changes to ***props*** or ***state***. Whenever the value of *props* or *state* is changed, React **re-renders** the component to reflect updated values on the UI. The lifecycle methods of the update phase are called on every update.

#### Unmounting Phase

Unmounting is the phase where a component is removed from the DOM. Components go out of scope and the lifecycle method of unmounting phase can be used to perform **cleanup activities**.

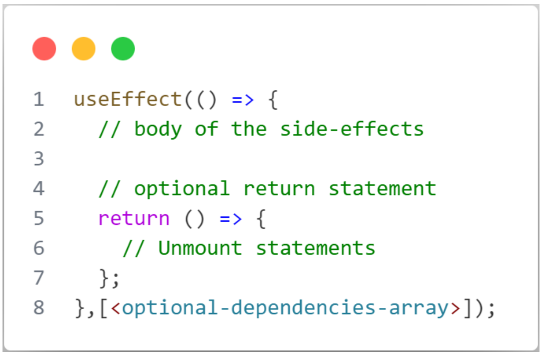
## Lifecycle Method

### useEffect()

The ***useEffect*** is a React Hook that synchronizes a component with an external system. It is used to perform *side effects* in *function components*. Side effects can be calling APIs, setting up the subscription, setting timers, etc.

By using this hook, we tell React that our component needs to do something after render. React will remember the function and call it later after performing the DOM updates. React guarantees the DOM has been updated by the time it runs the effects.

**Note**: Hooks embrace JavaScript closures to access the function state.

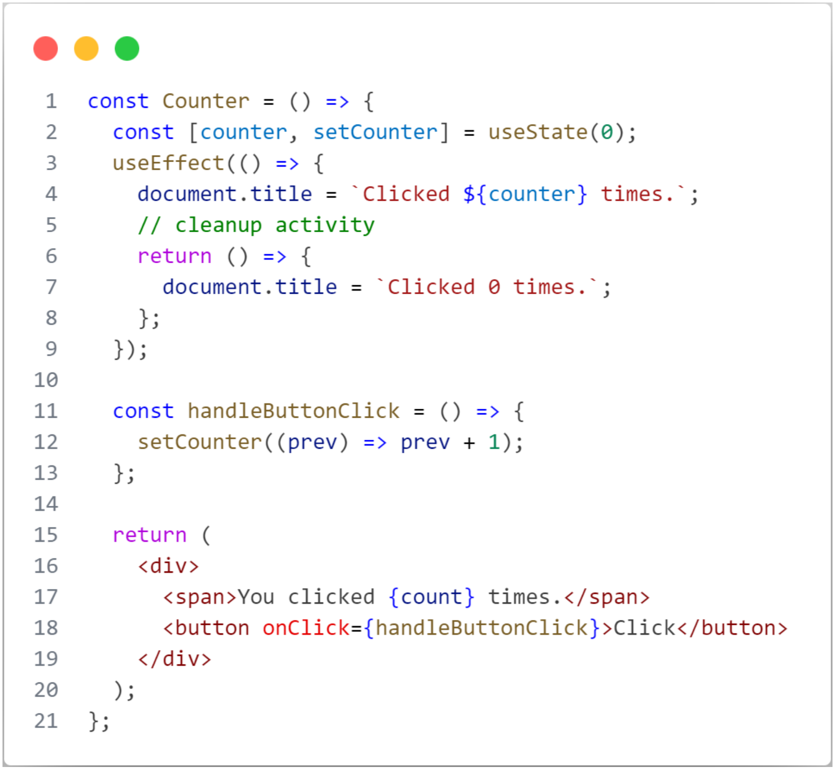


```
useEffect(() => {
  // body of the side-effects

  // optional return statement
  return () => {
    // Unmount statements
  };
},[<optional-dependencies-array>]);
```

**<optional-dependencies-array>**: This is the optional *Array* of dependencies, which instructs React, when to call this side-effect. React observes the changes in values of the dependency array items, and will call the side-effect statements, on change of any dependency.

**return () => { }**

***useEffect()*** can optionally return a function, which will be called on component unmount or before running the next effect.



```
const Counter = () => {
  const [counter, setCounter] = useState(0);
  useEffect(() => {
    document.title = `Clicked ${counter} times.`;
    // cleanup activity
    return () => {
      document.title = `Clicked 0 times.`;
    };
  });

  const handleButtonClick = () => {
    setCounter((prev) => prev + 1);
  };

  return (
    <div>
      <span>You clicked {count} times.</span>
      <button onClick={handleButtonClick}>Click</button>
    </div>
  );
};
```

Compared to class components' lifecycle methods, this one hook is equivalent to ***componentDidMount()***, ***componentDidUpdate()*** and ***componentWillUnmount()*** combined together.

Refer to **Chapter 10 (Hooks)**, to learn how ***useEffect()*** can be used as *componentDidMount()*, *componentDidUpdate()* and *componentWillUnmount()*.

**Note**: If *useEffect()* returns a function with some dependency or without any dependency, in such cases the returned function (cleanup method) will be invoked on every update. This may lead to performance issues, if not used wisely.

**Note**: Unlike *componentDidMount()* or *componentDidUpdate()*, effects scheduled with **useEffect() don't block the browser from updating the screen**. This makes the app **more responsive**.

The majority of effects don't need to happen synchronously. In uncommon cases where they do (such as measuring the layout), there is a separate *useLayoutEffect()* hook with an API identical to *useEffect()*.

## Summary

1. **Function components** are literally *JavaScript functions*.
2. A **hook** is a special function used to hook into React features, like *state* and *lifecycle methods*.
3. The ***useState()*** hook is used to add *state* to the function component.
4. The ***useEffect()*** is used to perform *side-effects/lifecycle events*. It is equivalent to *componentDidMount()*, *componentDidUpdate()* and *componentWillUnmount()*.

*CHAPTER 6*

# Controlled & UnControlled Component

## Preface to Controlled & Uncontrolled Component

### Controlled Components

*Value is controlled by React, accepts input props, update via handlers*

### Uncontrolled Component

*Value is controlled by DOM, refs used to access DOM*

### Default Values

*Initial value of element in uncontrolled component*

### When to use Uncontrolled Component

*Managing DOM nodes, integrating with third-party libraries*

### Handling Events

*Synthetic camelCase event handler functions*

## Controlled Components

A component is **“controlled”** when the information in it is driven by the *props* rather than its own local *state*. This lets the parent component fully specify the child component’s behavior. Controlled components are flexible to use, but require parent components to fully configure them via *props*.

For every *state value*, we will choose the component that **owns** it. This principle is also known as having a **“single source of truth”**. It doesn’t mean that all *state values* live in one component, but that for each *state*, there is a component that *owns* it.

Instead of duplicating the shared *state* between components, ***lift it up*** to their common shared parent, and *pass it down* to the children that need it.

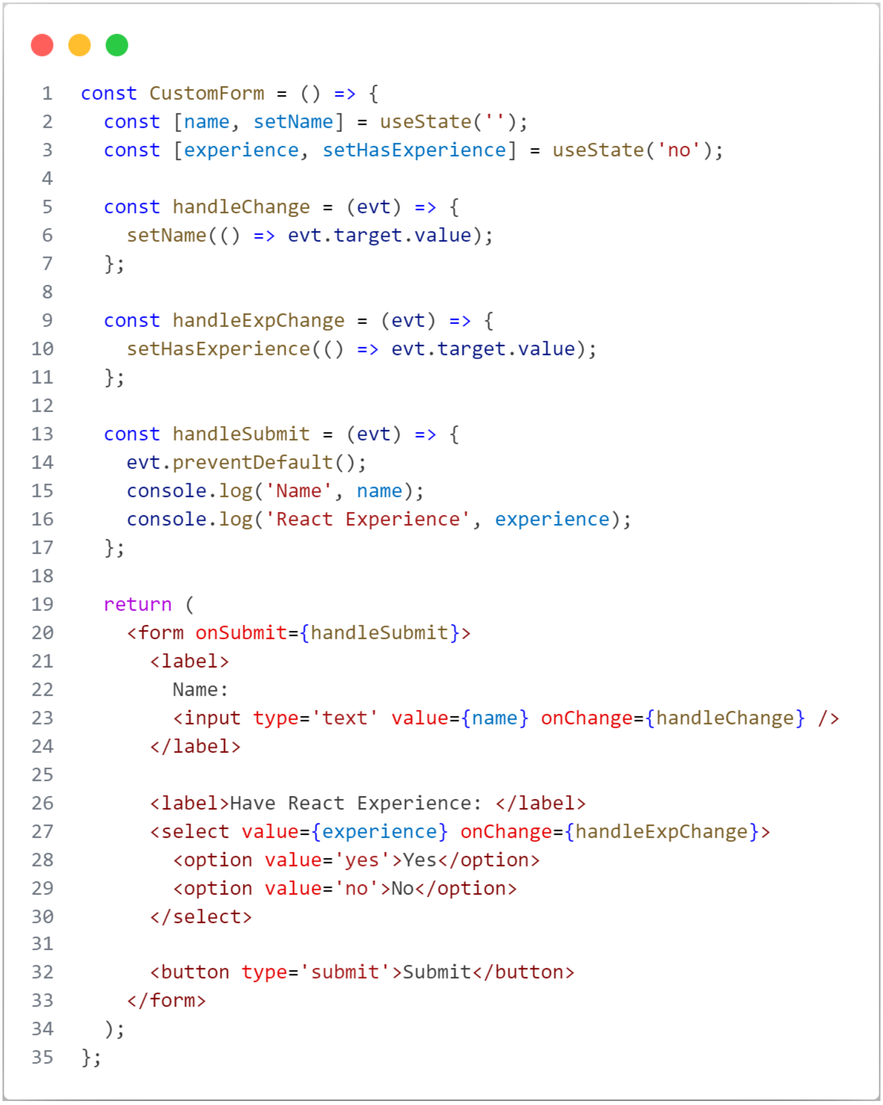
Controlled components are a bit faster, because of less *state* management and are easier to debug because the *state* is controlled by the parent component.

### Creating Controlled Form Element

HTML form elements work a bit differently from other DOM elements in React, because form elements naturally keep some internal state. In React, **mutable state** is typically kept in the *state* property of the components and updated by React. In such a case, React *state* is the **“single source of truth”**.

An *input form element* whose value is controlled by React can be called a **“controlled component”**. With a controlled component, the input’s value is always driven by the React *state*. Controlled components accept input props and update them by calling the change handlers.

In HTML, form elements such as *<input>*, *<select>*, *<textarea>* typically maintain their own state and update it based on user input.



```jsx
const CustomForm = () => {
  const [name, setName] = useState('');
  const [experience, setHasExperience] = useState('no');

  const handleChange = (evt) => {
    setName(() => evt.target.value);
  };

  const handleExpChange = (evt) => {
    setHasExperience(() => evt.target.value);
  };

  const handleSubmit = (evt) => {
    evt.preventDefault();
    console.log('Name', name);
    console.log('React Experience', experience);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type='text' value={name} onChange={handleChange} />
      </label>

      <label>Have React Experience: </label>
      <select value={experience} onChange={handleExpChange}>
        <option value='yes'>Yes</option>
        <option value='no'>No</option>
      </select>

      <button type='submit'>Submit</button>
    </form>
  );
};
```

**Note**: We can pass an ***array* into the *value*** attribute, allowing us to select ***multiple options*** in a **<select>** tag.

```
<select multiple={true} value={[“a”, ”c”] }>
        <option value="a">A</option>
        <option value="b">B</option>
        <option value="c">C</option>
        <option value="d">D</option>
</select>
```

## Uncontrolled Components

A component with the local *state* is called an **uncontrolled component** because its parent component can’t influence its behavior.

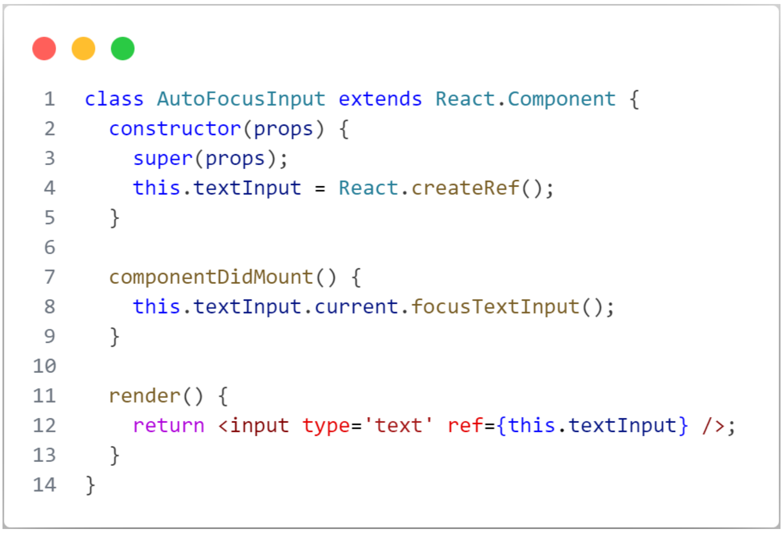
### Creating Uncontrolled Form Element

Components whose values are not controlled by React, instead are controlled by DOM itself can be called **“uncontrolled components”**. For example, an *<input type=”file” />* is always an uncontrolled element because its value can only be set by a user, and not programmatically.

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

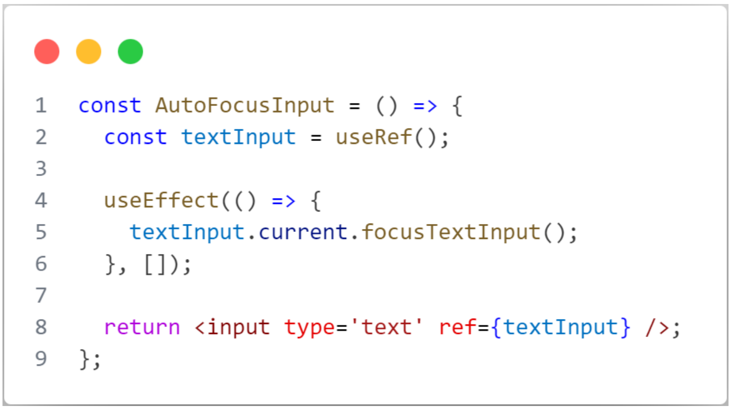
***Refs*** are used to access DOM nodes or React elements in uncontrolled components. React prefers to avoid using ***refs*** for anything that can be done declaratively.

*Example 1*: Using *ref* in **class components**. Below code focuses the *<input />* element on mount.



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

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

  render() {
    return <input type='text' ref={this.textInput} />;
  }
}
```

*Example 2*: Using *ref* in **function components**.



```
const AutoFocusInput = () => {
  const textInput = useRef();

  useEffect(() => {
    textInput.current.focusTextInput();
  }, []);

  return <input type='text' ref={textInput} />;
};
```

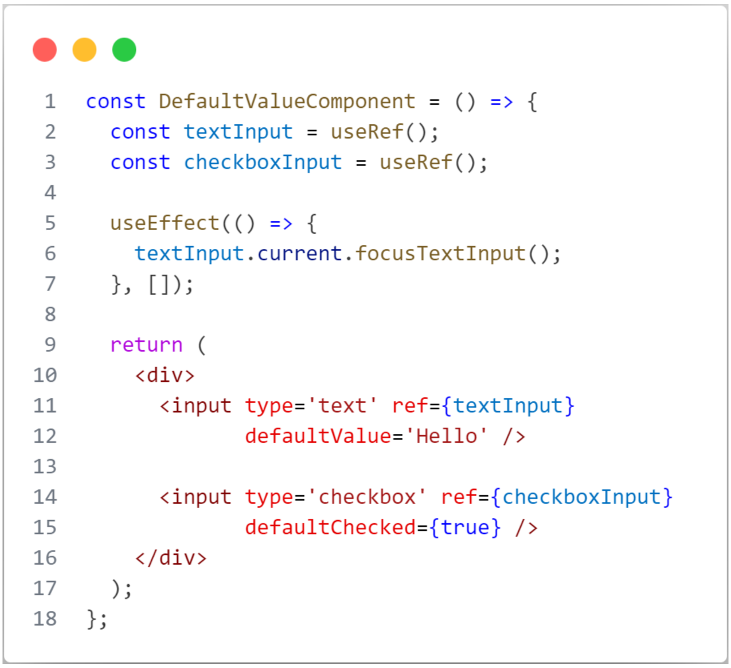
React will assign the ***"current"*** property with the DOM element when the component *mounts*, and assign it back to ***null*** when it *unmounts*. For *class components*, ***ref*** updates happen before *componentDidMount()* or *componentDidUpdate()* lifecycle methods.

## Default Values

In the React rendering lifecycle, the ***value*** attribute on form elements will override the *value* in the DOM. With an *uncontrolled component*, we often want React to specify the **initial value**, but leave subsequent updates *uncontrolled*.

In such cases, we can add ***defaultValue*** attribute instead of ***value***. Changing the value of ***defaultValue*** after a component has mounted will not cause any update of the value in the DOM.

Similarly, **checkbox**, **radio** elements have ***defaultChecked*** attributes.



```
const DefaultValueComponent = () => {
  const textInput = useRef();
  const checkboxInput = useRef();

  useEffect(() => {
    textInput.current.focusTextInput();
  }, []);

  return (
    <div>
      <input type='text' ref={textInput}
             defaultValue='Hello' />

      <input type='checkbox' ref={checkboxInput}
             defaultChecked={true} />
    </div>
  );
};
```

## When to use Uncontrolled Components

Since an uncontrolled component keeps the source of truth in the DOM, it is sometimes easier to integrate React and non-React code when using uncontrolled components.

It can be slightly less code if we want to be quick and dirty. Otherwise, we should always use controlled components.

Few of the use-cases for using uncontrolled components:

- Managing focus, text selection, or media playback.
- Integrating with third-party DOM libraries.

## Handling Events

We can add event handlers and pass them as a *prop* to the JSX. Event handlers are the *functions* that will be triggered in response to interactions like button click, hover, etc.

Handling events with React elements is very similar to handling events on DOM elements, with some syntax differences:

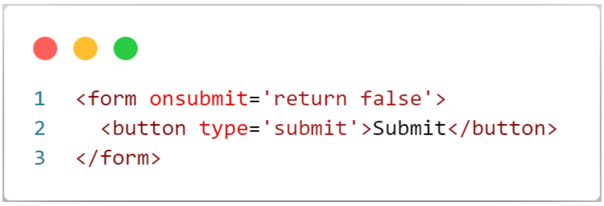
- React events are named using **camelCase**, rather than **lowercase**.
- With JSX, we pass a function as the **event handler**, rather than **string**.

**For Example, in the HTML:**
*<button **onclick**="handleClick()">Click</button>*

**In React, same is written as:**
*<button **onClick**={**handleClick**}>Click</button>*

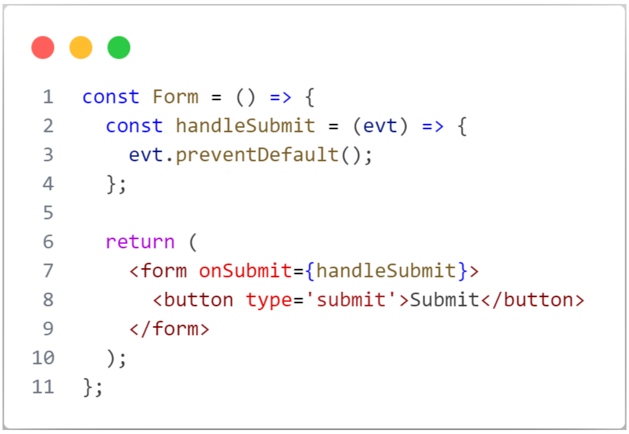
Another difference is that we cannot return *false* to prevent default behavior in React, we must call *preventDefault()* explicitly.

**For Example, in the HTML:**



```html
<form onsubmit='return false'>
  <button type='submit'>Submit</button>
</form>
```

**In React, same is written as:**



```jsx
const Form = () => {
  const handleSubmit = (evt) => {
    evt.preventDefault();
  };

  return (
    <form onSubmit={handleSubmit}>
      <button type='submit'>Submit</button>
    </form>
  );
};
```

## Synthetic Event

React defines synthetic events according to the *W3C* spec, so we don't need to worry about cross-browser compatibility.

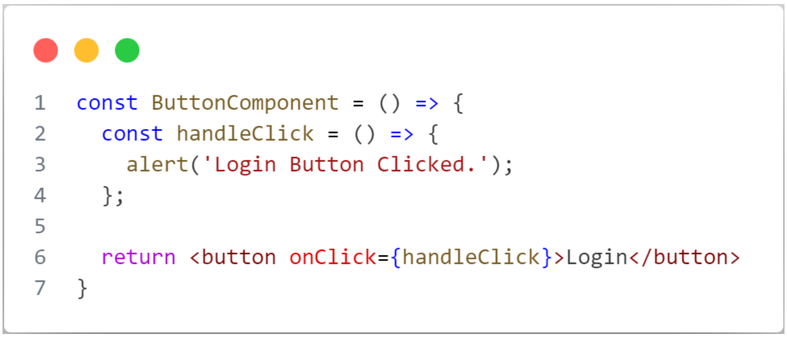
React events do not work exactly the same as native events. Some React events do not map directly to the browser's native event. For example, in *onMouseLeave*, *e.nativeEvent* will point to a *mouseout* event.

## Adding event handlers

To add an event handler, we will define a *function* and then **pass it as a prop** to the JSX element.

By convention, it is common to name event handlers as “**handle”** followed by the *event name, like handleClick, handleMouseEnter, etc* and corresponding event props should start with **“on”**, followed by a capital letter like *onClick*, *onMouseEnter*, etc.

In the below example, we defined an event handler **“handleClick”**, which is passed to the *prop* **“onClick”** of the *button* JSX.



```
const ButtonComponent = () => {
  const handleClick = () => {
    alert('Login Button Clicked.');
  };

  return <button onClick={handleClick}>Login</button>
}
```

**Note**: Make sure to use appropriate HTML tags for the event handlers. For example, to handle clicks, use ***<button onClick={handleClick}></button>*** instead of *<span onClick={handleClick}></span>*.

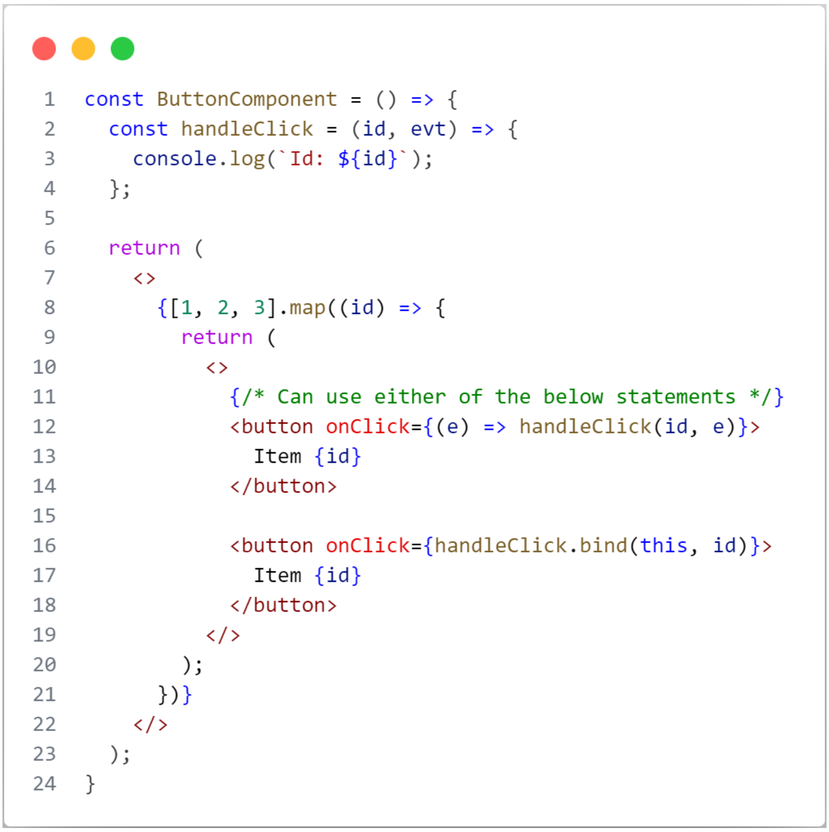
Using a real browser *<button>* enables built-in browser behaviors like keyboard navigation and other Web Accessibility requirements.

## Passing Arguments to Event Handlers

By default the event handler receives the ***synthetic event** object* as the argument.

But there could be scenarios where we need to pass additional data to the event handler. For example, while displaying a list of components, we may need data of the current clicked item.

We may use the JavaScript ***arrow function*** or ***bind function*** to pass additional data.



```
const ButtonComponent = () => {
  const handleClick = (id, evt) => {
    console.log(`Id: ${id}`);
  };

  return (
    <>
      {[1, 2, 3].map((id) => {
        return (
          <>
            {/* Can use either of the below statements */}
            <button onClick={(e) => handleClick(id, e)}>
              Item {id}
            </button>

            <button onClick={handleClick.bind(this, id)}>
              Item {id}
            </button>
          </>
        );
      })}
    </>
  );
}
```

## Summary

1. An input element whose value is controlled by React is called a **Controlled Component**.
2. It accepts *input props* and updates them by calling the *change handlers*.
3. In Uncontrolled components, value is controlled by **DOM** and not by *React*.
4. **Refs** are used to access DOM nodes or React elements.
5. React will assign the ***"current"*** property with the **DOM element** on *mount* and set it back to ***null*** on *unmount*.
6. Should be used for managing focus, text selection, integration with third-party libraries, etc.
7. Form elements can be *controlled* or *uncontrolled*.
8. In a *controlled* element, value is controlled by React while in an *uncontrolled* element, value is controlled by DOM.
9. The ***defaultValue*** attribute is used to assign initial values to the form elements. Similarly, *checkbox*, *radio* elements have ***defaultChecked*** attributes.
10. **React events** are named using *camelCase*, and are **functions** rather than *strings*.
11. React defines **Synthetic events**, following *W3C spec*, to make them **cross-browser** compatible.

*CHAPTER 7*

# Lists & Keys

# Preface to List & Keys

### Define Lists

*Use JavaScript loops, map(), etc to iterate react elements*

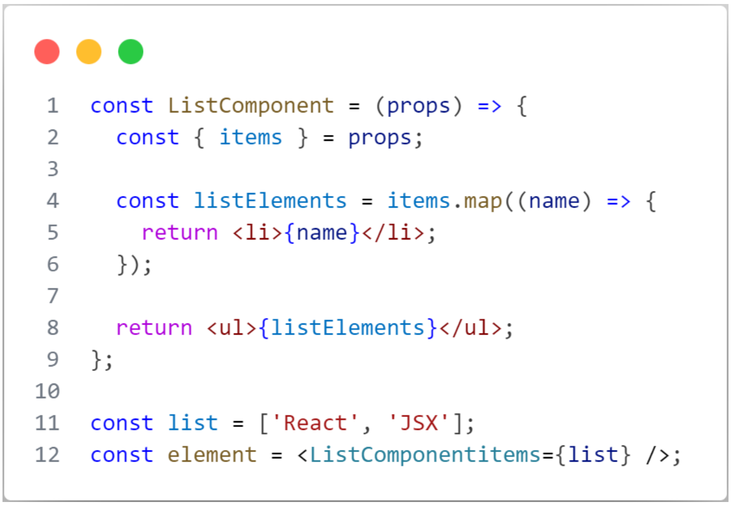
### Keys

*A string or number that helps react identify item in list, unique among siblings*

## Lists

We often need to show several instances of the same component/element using different data. For example, displaying list of users, table rows of student records, etc. In such use cases, we can store data in JavaScript *Arrays/Objects*.

Since JSX embraces JavaScript expressions, we can use JavaScript constructs like *for* loop, *map()*, *forEach()* or any other iterable entity of JavaScript to render a list of React elements/components using the data stored in *Arrays/Objects*.



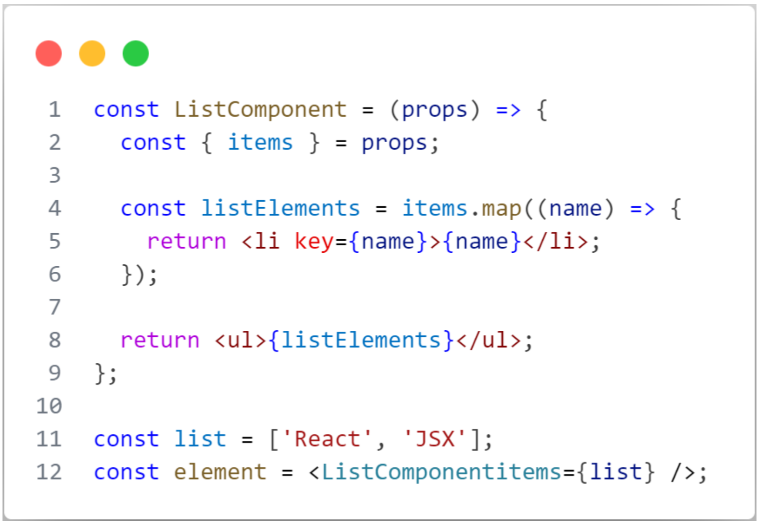
```
const ListComponent = (props) => {
  const { items } = props;

  const listElements = items.map((name) => {
    return <li>{name}</li>;
  });

  return <ul>{listElements}</ul>;
};

const list = ['React', 'JSX'];
const element = <ListComponentitems={list} />;
```

In the above example, ***map()*** method is used to build the list of ***<li>*** elements. Above code will render below output:

- React
- JSX

## 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. Let's add a ***key*** to elements of our last example.

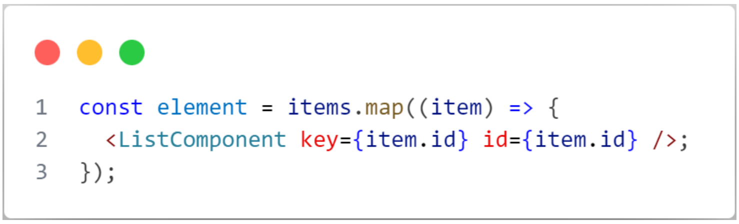


```
const ListComponent = (props) => {
  const { items } = props;

  const listElements = items.map((name) => {
    return <li key={name}>{name}</li>;
  });

  return <ul>{listElements}</ul>;
};

const list = ['React', 'JSX'];
const element = <ListComponentitems={list} />;
```

If we do not add an **explicit key** to list items then React will use the array **indexes** as the keys.

**Note**: React **doesn'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.

Keys serve as a hint to React but they don't get passed to our components.



```
const element = items.map((item) => {
  <ListComponent key={item.id} id={item.id} />;
});
```

In the above example, **ListComponent** can read ***props.id***, but not ***props.key***.

## 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 with two different arrays.

## Summary

1. **Iterables** entities like *map(), forEach(),* etc. are used to render lists of react elements.
2. **Keys** are added to give items a stable entity. It helps React to identify the changed elements.
3. React doesn’t recommend using **indexes** for keys.
4. Keys must only be **unique among the siblings** and not globally.

*CHAPTER 8*

# Context

# Preface to Context

### What is Context

*A way to pass props to any child component in tree, without adding at each level*

### Prop Drilling

*Add prop to every child component in the tree*

### Context API

*APIs to create and consume Context, createContext(), useContext(), etc*

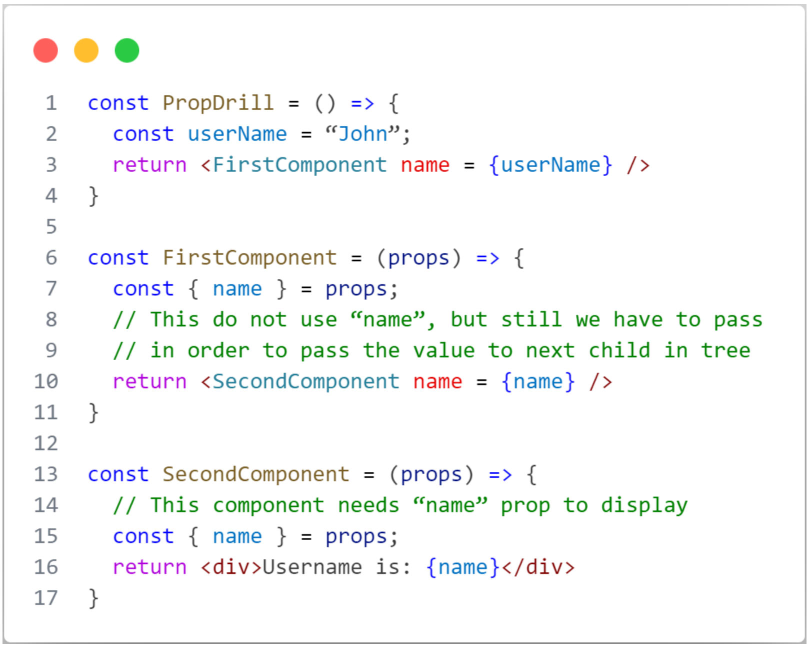
### Issues using Context

*Re-renders child below current component, ignoring shouldComponentUpdate()*

## Context

Context provides a way to pass data through the component tree without having to pass *props* down manually at every level.

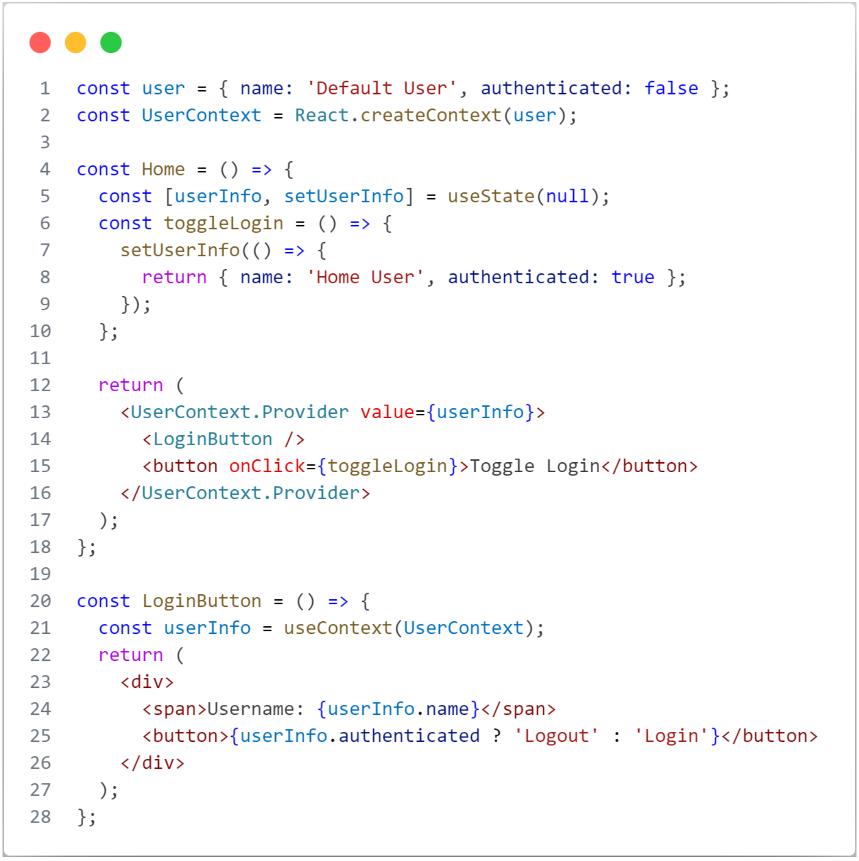
In React application, data is passed **top-down** (parent to child component) via *props*, but this approach can lead to a problem called **“prop drilling”**, where we have to add *prop* to every child component in the tree, whether or not that child component needs that *prop*. This continuous drilling is required to send the *props* to the desired component in the tree.



```
const PropDrill = () => {
  const userName = “John”;
  return <FirstComponent name = {userName} />
}

const FirstComponent = (props) => {
  const { name } = props;
  // This do not use “name”, but still we have to pass
  // in order to pass the value to next child in tree
  return <SecondComponent name = {name} />
}

const SecondComponent = (props) => {
  // This component needs “name” prop to display
  const { name } = props;
  return <div>Username is: {name}</div>
}
```

### 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 user, theme, or preferred language. By using *context*, we can avoid passing *props* to intermediate elements of the child tree.



```
const user = { name: 'Default User', authenticated: false };
const UserContext = React.createContext(user);

const Home = () => {
  const [userInfo, setUserInfo] = useState(null);
  const toggleLogin = () => {
    setUserInfo(() => {
      return { name: 'Home User', authenticated: true };
    });
  };

  return (
    <UserContext.Provider value={userInfo}>
      <LoginButton />
      <button onClick={toggleLogin}>Toggle Login</button>
    </UserContext.Provider>
  );
};

const LoginButton = () => {
  const userInfo = useContext(UserContext);
  return (
    <div>
      <span>Username: {userInfo.name}</span>
      <button>{userInfo.authenticated ? 'Logout' : 'Login'}</button>
    </div>
  );
};
```

In above code, the **LoginButton** component will render below elements on the mount of the component.

*<span>Username: Default User</span>*
*<button>Login</button>*

But once a user clicks on toggleButton in the **Home** component, the *Home* component will be re-rendered, because the value of the context will change. Now the *LoginButton* component will render the below elements.

*<span>Username: Home User</span>*
*<button>Logout</button>*

## Context API

### React.createContext

This method creates a ***Context*** object. When React renders a component that *subscribes* to this ***Context*** object it will read the *current context value* form the closest matching ***Provider*** above it in the tree.

*Syntax*:
*const **ReactContext** = **React.createContext**(**defaultValue**)*

**defaultValue**: The defaultValue argument is only used when a component does not have a matching Provider above it in the tree.

**Note**: Passing ***undefined*** as a **Provider value** does not cause components to use ***defaultValue***.

### Context.Provider

Every Context *object* comes with a Provider React Component that allows consuming components to subscribe to context changes.

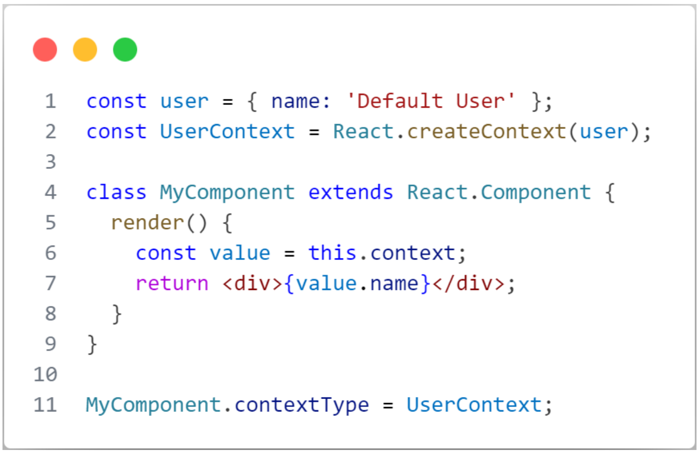
*Syntax*:
*<**ReactContext**.Provider value={/* some value */} >*

The ***Provider*** component accepts a ***value*** *prop* to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

All consumers that are descendants of a Provider will re-render whenever Provider's ***value*** *prop* changes. The propagation from Provider to consumer is not subject to the *shouldComponentUpdate()* method, so the consumer is updated even when an ancestor component skips an update.

## Class.contextType

The ***contextType*** property can be used to assign a **Context object** created by *React.createContext()*. Using this property lets us consume the nearest current value using ***"this.context"***.



```
const user = { name: 'Default User' };
const UserContext = React.createContext(user);

class MyComponent extends React.Component {
  render() {
    const value = this.context;
    return <div>{value.name}</div>;
  }
}

MyComponent.contextType = UserContext;
```

## Context.Consumer

It is a **React component** that ***subscribes*** to *context changes*. This component lets us subscribe to a context within a **function component**.

*Syntax*:

*<ReactContext.**Consumer**>*

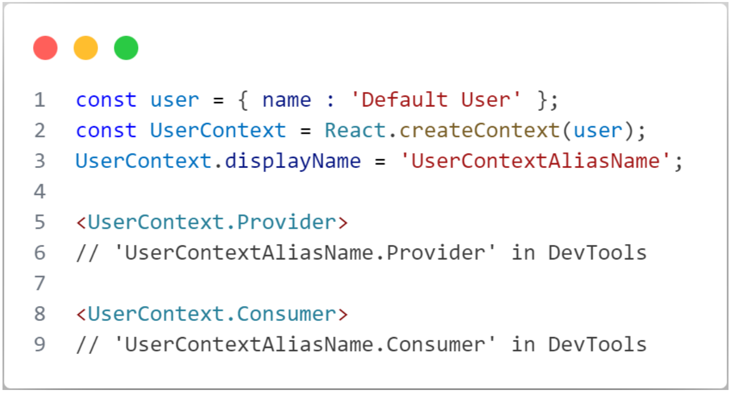
*{ **value =>** { /* Render elements based on context value */ }*

*</ReactContext.**Consumer**>*

This component requires a ***function* as a child**. The function receives the current context value and returns a React node. The ***value*** argument will be equal to the ***value prop*** of the closest ***Provider*** for this context in the tree.

## Context.displayName

Context object accepts a displayName *string* property. React DevTools uses this name to determine what to display for the context. This is used in rare-cases.



```
const user = { name : 'Default User' };
const UserContext = React.createContext(user);
UserContext.displayName = 'UserContextAliasName';

<UserContext.Provider>
// 'UserContextAliasName.Provider' in DevTools

<UserContext.Consumer>
// 'UserContextAliasName.Consumer' in DevTools
```

## Issues using Context

Although Context is a good fit for sharing data between parent and child components in a deeply nested component tree. Still, there are a couple of issues with the re-rendering mechanism of Context.

Let's learn few of the issues:

- When **Provider *value prop*** changes, it will **re-render** all the child components below the current component, irrespective of the implementation of *shouldComponentUpdate()* lifecycle method. This may cause performance issues.

- **Provider *value prop*** update has to be made with caution, because it will only re-render the child components **below** the current component and not the components **above** the current component in the component tree.

## Summary

1. **Context** is used to pass data through the component tree, without passing manually at each level. Fixes the issue of **“prop drilling”**.
2. Use context to share data which is global for a given component tree.
3. The ***React.createContext()*** method created a **Context** object. Multiple components can subscribe to it and will be notified on change.
4. The ***Context.Provider*** component is used to **subscribe** to the *Context*.
5. The ***Class.contextType*** property is used in the *class component* to consume the *nearest current value*.
6. The ***Context.Consumer*** is the component that **subscribes** to the *Context* object. This requires **function as a child** which gets the ***value*** attribute.
7. Issues with Context: On change of value, it will re-render all child components below the current component and not the components above current component, irrespective of whether they need it or not.

*CHAPTER 9*

# Fragments

## Preface to Fragments

### Fragments

*Used to return multiple elements, avoiding wrapper hell*

### Syntax

*<React.Fragment> or <>*

## Fragments

**Fragments** are used to **return multiple elements** from a component. By definition React components can only return just one React Element.

One way to group a list of children could be to wrap all childrens inside ***<div>*** element, but this will unnecessarily **add extra** *<div>* to the DOM as well , which may result in a problem known as **"wrapper hell"**. Along with the "wrapper hell" problem, it will also result in change of structure of DOM nodes, which may result in making the HTML invalid.

Fragments solves the above problems by grouping a list of children **without adding extra** nodes to DOM.

*Syntax*:

```
<React.Fragment>
        <ChildComponent1 />
        <ChildComponent2 />
        ......................................
        <ChildComponentN />
</React.Fragment>
```

*There is a shorter syntax too*:

```
<>
        <ChildComponent1 />
        <ChildComponent2 />
        ......................................
        <ChildComponentN />
</>
```

## Summary

1. **Fragments** are used to **return multiple elements** from a component.
2. Groups elements without adding extra DOM nodes.
3. Fixes the **“wrapper hell”** and invalid re-structuring of code.

*CHAPTER 10*

# Hooks

# Preface to Hooks

## What are hooks

*Added in v16.8, use state and lifecycle feature in function component*

## Rules of hooks

*Used only in function component, at top level, not inside loop or condition*

## Basic Hooks

*Commonly used hooks, useState(), useEffect(), useContext()*

## Additional hooks

*For specific cases, useReducer(), useCallback(), useMemo(), useRef(), useImperativeHandle, useLayoutEffect, etc*

## Hooks

**Hooks** were added in React **v16.8**. Hooks lets us use *state* and other React features without writing a class component. Hooks are functions that let us **"hook into"** React *state* and lifecycle features from function components. Hooks **don't work** inside *class* components.

React provides a few *built-in* Hooks like *useState()*, *useEffect()*, etc. We can also create custom hooks to reuse stateful behavior between different components.

### Rules of Hooks

Hooks are JavaScript functions, but they impose two additional rules:

- Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions or nested functions.

- Only call Hooks **from React function components**. Don't call Hooks from regular JavaScript functions.

### Basic Hooks

Below is the list of commonly used built-in Hooks.

#### useState

This Hooks is used to add state to a function component. It returns a stateful value and a function to update it.

*Syntax*:
*const [**stateValue**, **updateFunction**] = **useState**(**initialValue**);*

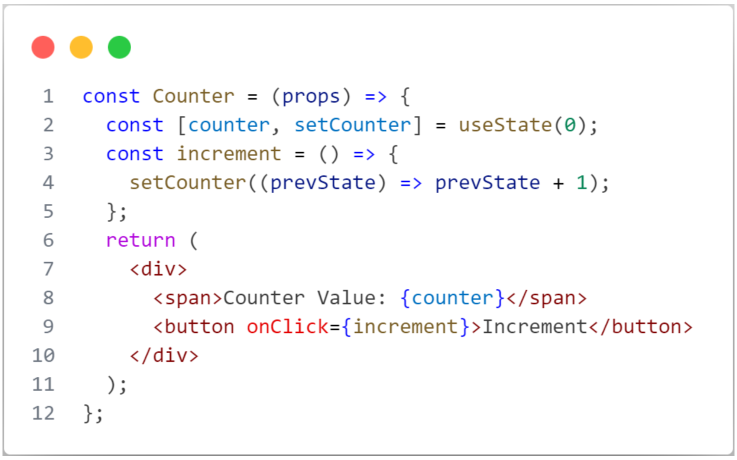
**useState:** This is the hook used to add *state* to a function component. It returns a **pair** - an *array* with two items. The first item is the current value and second is a function that updates it.

**stateValue:** It is the first value of the pair of values returned from the *useState()* function. This is the name of the state variable. In contrast to the class component's *state* being an *object*, the *state* of a function component can be any valid *type*. For example, it can be *numbers*, *strings*, *arrays*, etc.

**updateFunction:** This is the second value of the pair of values returned from the *useState()* function. This function is equivalent to *setState()* of the class component and used to update the respective *state* variable.

**initialValue:** It is the initial value of the *state* variable. The initial value is only assigned once, on the mount of the component and is ignored on further update events.

Let's build our Counter component using the **function component** and **useState()** *hook*.



```
const Counter = (props) => {
  const [counter, setCounter] = useState(0);
  const increment = () => {
    setCounter((prevState) => prevState + 1);
  };
  return (
    <div>
      <span>Counter Value: {counter}</span>
      <button onClick={increment}>Increment</button>
    </div>
  );
};
```

### Functional Updates

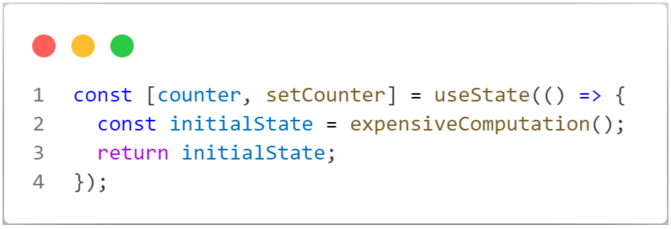
If the new *state* is computed using the previous *state* value, in such cases we can pass a *function* to the **updateFunction**. The function will receive the previous *state*, and return an updated *state*.

In the above code, we used functional updates.
***setCounter****((prevState) => prevState + 1);*

Alternative to functional updates is to pass a value to the **updateFunction**. Because React may batch *state* updates for performance benefits, therefore this syntax may not provide updated value.
***setCounter****(counter + 1);*

### Lazy initial state

The **initialValue** argument is the *state* used during the initial render. In subsequent renders, it is disregarded. If the initial *state* is the result of an expensive computation, we may provide a *function* instead, which will be executed only on the initial render.



```
const [counter, setCounter] = useState(() => {
  const initialState = expensiveComputation();
  return initialState;
});
```

### Bailing out of a state update

If we update a state Hook to the same value as the current *state*, React will bail out without rendering the children or firing effects. React uses the ***Object.is* comparison algorithm** for comparing values.

### Batching of state updates

React may group several *state* updates into a single re-render to improve performance. Before React 18, only updates inside React event handlers were batched. Starting with **React 18**, batching **is enabled for all updates by default**.

**Note**: React makes sure that updates from several *different* user-initiated events, like **clicking a button twice**, are always processed separately and **do not get batched**. This prevents logical mistakes.

## useEffect

Effects lets us **"step outside"** of *React* and synchronize the components with some external systems like a non-React widget, network call, or the browser DOM.

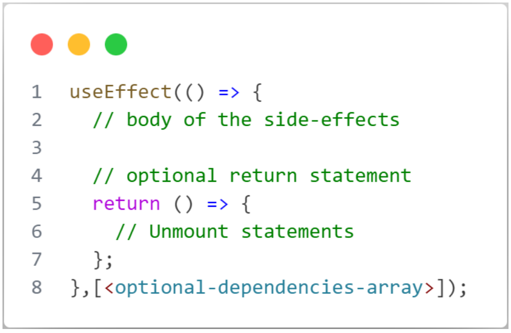
The ***useEffect*** is a React Hook that synchronizes a component with an external system. It is used to perform *side effects* in *function components*. Side effects can be calling APIs, setting up the subscription, setting timers, etc.

By using this hook, we tell React that our component needs to do something after *render*. React will remember the function and call it later after performing the DOM updates. React guarantees the DOM has been updated by the time it runs the effects.

If there is no external system involved (for example, if we want to update a component's state when some *props* or *state* change), we should **not use** *useEffect*. Removing unnecessary effects will make code easier to follow, faster to run, and less error-prone.

> **Note**: Hooks embrace JavaScript closures to access the function *state*.

*Syntax*:

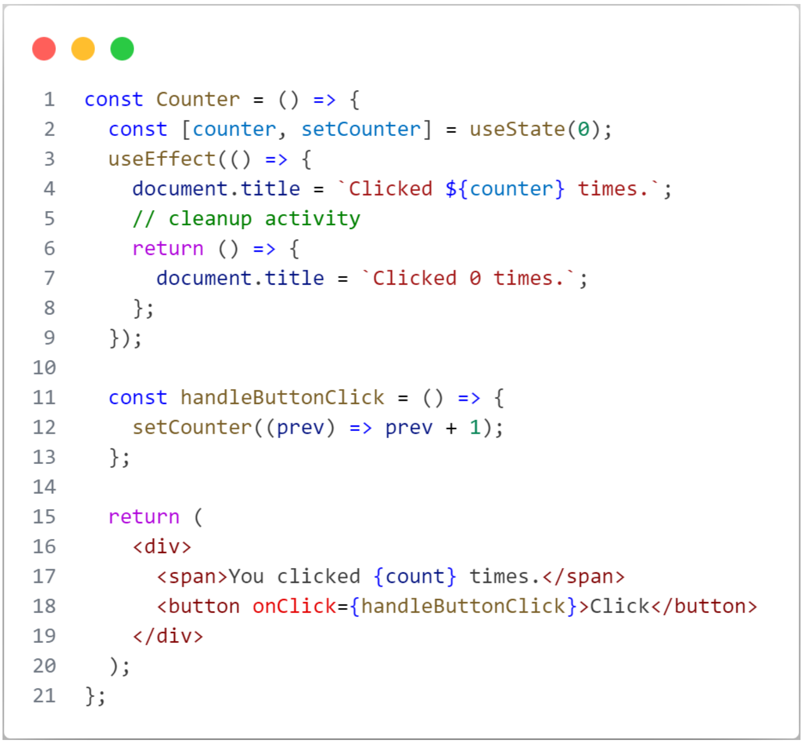


```
useEffect(() => {
  // body of the side-effects

  // optional return statement
  return () => {
    // Unmount statements
  };
},[<optional-dependencies-array>]);
```

**<optional-dependencies-array>**: This is the optional *Array* of dependencies, which instructs React, when to call this side-effect. React observes the changes in values of the dependency *array* items, and will call the side-effect statements, on change of any dependency.

***return () => { }***

*useEffect()* can optionally return a *function*, which will be called on component unmount or before running the next effect.



```
const Counter = () => {
  const [counter, setCounter] = useState(0);
  useEffect(() => {
    document.title = `Clicked ${counter} times.`;
    // cleanup activity
    return () => {
      document.title = `Clicked 0 times.`;
    };
  });

  const handleButtonClick = () => {
    setCounter((prev) => prev + 1);
  };

  return (
    <div>
      <span>You clicked {count} times.</span>
      <button onClick={handleButtonClick}>Click</button>
    </div>
  );
};
```

Compared to class components' lifecycle methods, this one hook is equivalent to ***componentDidMount()***, ***componentDidUpdate()*** and ***componentWillUnmount()*** combined together.

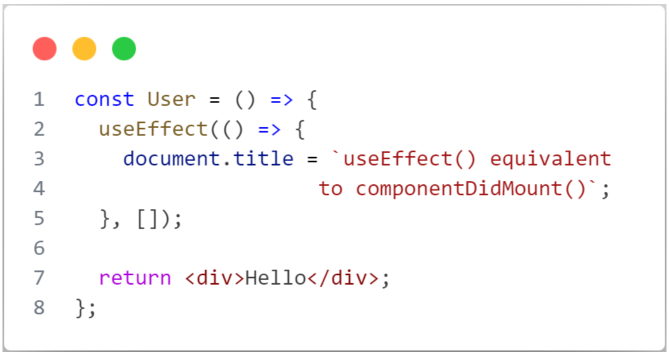
Let's learn how ***useEffect()*** can be used as *componentDidMount()*, *componentDidUpdate()* and *componentWillUnmount()*.

## As componentDidMount()

Passing empty **[]** dependency to *useEffect()* will make it equivalent to *componentDidMount()* and statements will be executed only once.

*Syntax*:

*useEffect(()= >{ },**[ ]**)*



```
const User = () => {
  useEffect(() => {
    document.title = `useEffect() equivalent
                     to componentDidMount()`;
  }, []);

  return <div>Hello</div>;
};
```

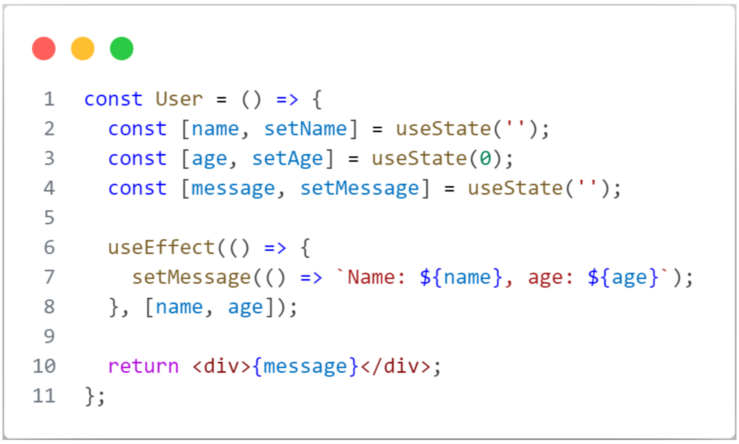
Above code will call ***useEffect()*** only once on **mount**.

### As componentDidUpdate()

Passing the dependency *array* to *useEffect()* will make it equivalent to *componentDidUpdate()* and statements will be executed on every update to the dependency items.

*Syntax*:

*useEffect(()= >{ },**[**dependency1, dependency2,..., dependencyN**]**)*



```
const User = () => {
  const [name, setName] = useState('');
  const [age, setAge] = useState(0);
  const [message, setMessage] = useState('');

  useEffect(() => {
    setMessage(() => `Name: ${name}, age: ${age}`);
  }, [name, age]);

  return <div>{message}</div>;
};
```

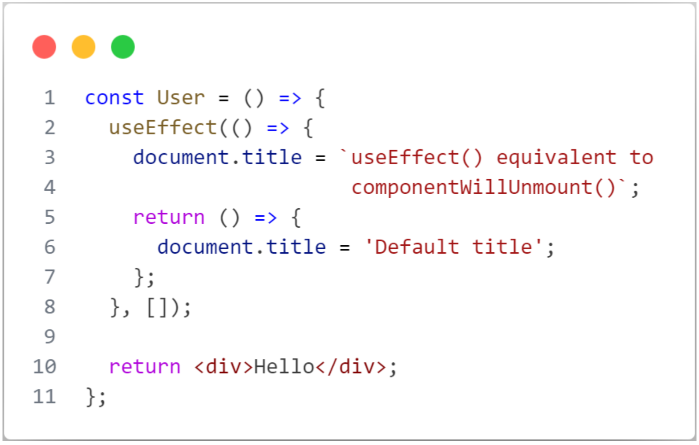
Above code will call ***useEffect()*** on **mount** as well as **on change** in value of ***"name"*** or ***"age"***.

### As componentWillUnmount()

*useEffect()* optionally returns a ***function***, which is called before running the *useEffect()* again. Returning this *function* along with **empty dependency []** will be equivalent to *componentWillUnmount()*.

*Syntax*:

*useEffect(() => { return () => {  // cleanup statements } },* ***[]****)*



```
const User = () => {
  useEffect(() => {
    document.title = `useEffect() equivalent to
                      componentWillUnmount()`;
    return () => {
      document.title = 'Default title';
    };
  }, []);

  return <div>Hello</div>;
};
```

**Note**: If *useEffect()* returns a *function* with some dependency or without any dependency, in such cases the returned *function* (cleanup method) will be invoked on every update. This may lead to performance issues, if not used wisely.

**Note**: Unlike *componentDidMount()* or *componentDidUpdate()*, effects scheduled with **useEffect() don't block the browser from updating the screen**. This makes the app **more responsive**. The majority of effects don't need to happen synchronously.

In uncommon cases where they do (such as measuring the layout), there is a separate *useLayoutEffect* Hook with an API identical to *useEffect()*.

## useContext

This hook is used to access Context values inside the function component. It returns the context value for the calling component.

*Syntax*:
*const value = useContext(**ContextObject**)*

**ContextObject**: It is the React Context object, returned from *React.createContext()* method.

The *useContext()* Hook accepts a context object "**ContextObject**" and returns the current context **value** for that context. The current context value is determined by the ***value prop*** of the nearest **<ContextObject.Provider>** above the calling component in the tree.

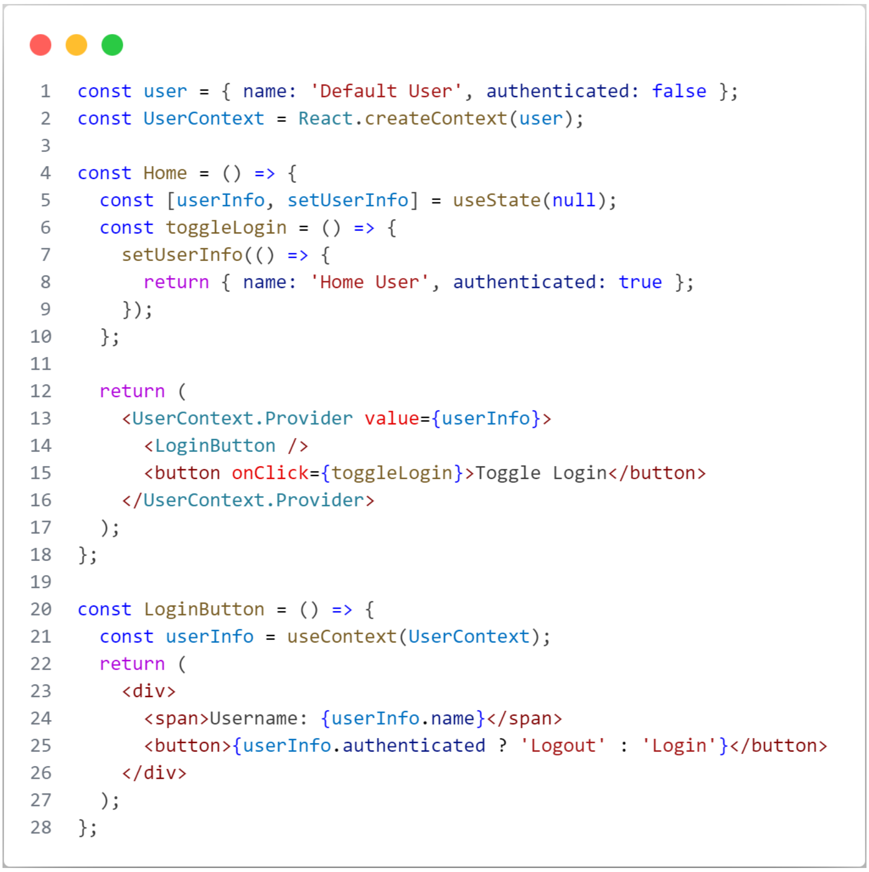
When the nearest **<ContextObject.Provider>** updates the ***value***, this Hook will trigger a re-render with the latest context value.

In the below code, the **LoginButton** component will render below elements on the mount of the component.

*<span>Username: Default User</span>*
*<button>Login</button>*

But once a user clicks on toggleButton in the **Home** component, the Home component will be re-rendered, because the value of the context will change. Now the LoginButton component will render the below elements.

*<span>Username: Home User</span>*
*<button>Logout</button>*



```
const user = { name: 'Default User', authenticated: false };
const UserContext = React.createContext(user);

const Home = () => {
  const [userInfo, setUserInfo] = useState(null);
  const toggleLogin = () => {
    setUserInfo(() => {
      return { name: 'Home User', authenticated: true };
    });
  };

  return (
    <UserContext.Provider value={userInfo}>
      <LoginButton />
      <button onClick={toggleLogin}>Toggle Login</button>
    </UserContext.Provider>
  );
};

const LoginButton = () => {
  const userInfo = useContext(UserContext);
  return (
    <div>
      <span>Username: {userInfo.name}</span>
      <button>{userInfo.authenticated ? 'Logout' : 'Login'}</button>
    </div>
  );
};
```

## Additional Hooks

The following Hooks are either variants of the basic ones or only needed for specific edge cases.

### useReducer

This hook is an alternative to *useState* Hook. It is usually preferable over *useState* when we have complex *state* logic that involves multiple sub-values or when the next *state* depends on the previous one. *useReducer()* also optimizes the performance of the component.

*Syntax*:
*const [**state**, **dispatch**] = useReducer(**reducer**, **intialArg**, init);*

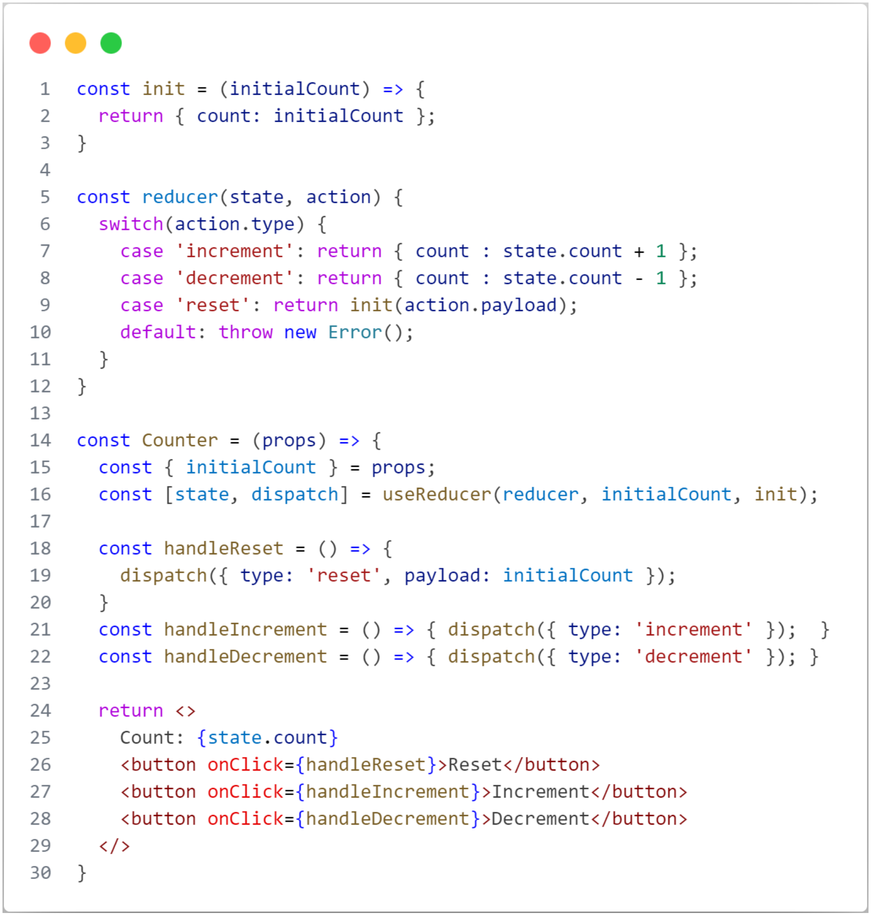
**state**: This variable holds the current value of the *state*.

**dispatch**: This method is used to trigger a change to the *state* value.

**reducer**: This is a method of type ***(state, action) => newState***. It accepts **"action"** from **dispatch**, uses the ***current state*** to compute **newState** and returns the **newState**.

**intialArg**: It is the initial value of the *state*.

*init*: This is an ***optional function***, used for lazy initialization of the *state*. If provided, the initial *state* will be set to ***init(initialArg)***. It lets us extract the logic for calculating the initial *state* outside the reducer.



```
const init = (initialCount) => {
  return { count: initialCount };
}

const reducer(state, action) {
  switch(action.type) {
    case 'increment': return { count : state.count + 1 };
    case 'decrement': return { count : state.count - 1 };
    case 'reset': return init(action.payload);
    default: throw new Error();
  }
}

const Counter = (props) => {
  const { initialCount } = props;
  const [state, dispatch] = useReducer(reducer, initialCount, init);

  const handleReset = () => {
    dispatch({ type: 'reset', payload: initialCount });
  }
  const handleIncrement = () => { dispatch({ type: 'increment' });  }
  const handleDecrement = () => { dispatch({ type: 'decrement' }); }

  return <>
    Count: {state.count}
    <button onClick={handleReset}>Reset</button>
    <button onClick={handleIncrement}>Increment</button>
    <button onClick={handleDecrement}>Decrement</button>
  </>
}
```

## useCallback

This hook is used to create a memoized callback, which will execute only on the change of the given dependencies.

*Syntax*:

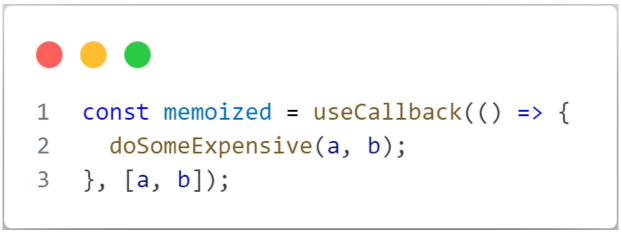
*const memoizedCallback = useCallback(() => {*

*// expensive expressions*

*}, **[/* dependency list */]**)*

The *useCallback()* will return a memoized version of the callback that only executes if one of the dependencies gets changed.

The ***useCallback(fn, deps)*** is equivalent to ***useMemo(() => fn, deps)***.



```
const memoized = useCallback(() => {
  doSomeExpensive(a, b);
}, [a, b]);
```

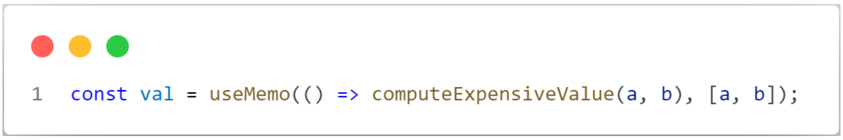
In the above code, **"doSomeExpensive"** method will be called on change on the given dependencies **"a"** and **"b"**.

## useMemo

The **useMemo** lets us cache the result of a calculation between re-renders. It will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

If no dependency array is provided, a new value will be computed on every render.

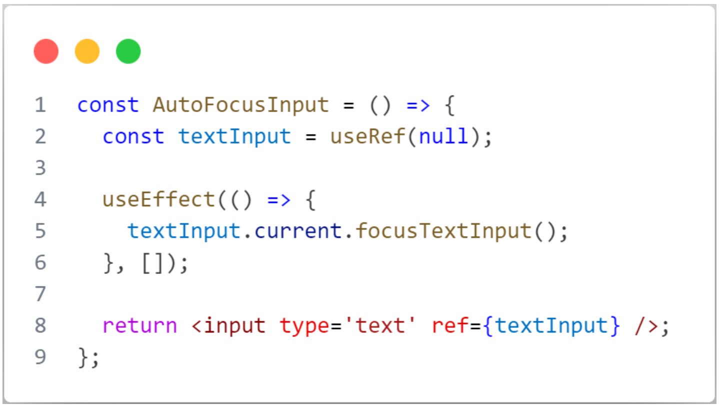
*Syntax*:
*const memoized = useMemo(() =>* ***expensiveFunction****, [/* dependencies */]);*



```
const val = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

## useRef

The **useRef** returns a mutable *ref* object whose ***".current"*** property is initialized to the passed argument (*initialValue*). The returned object will persist for the full lifetime of the component. This hook is used with **uncontrolled components**.



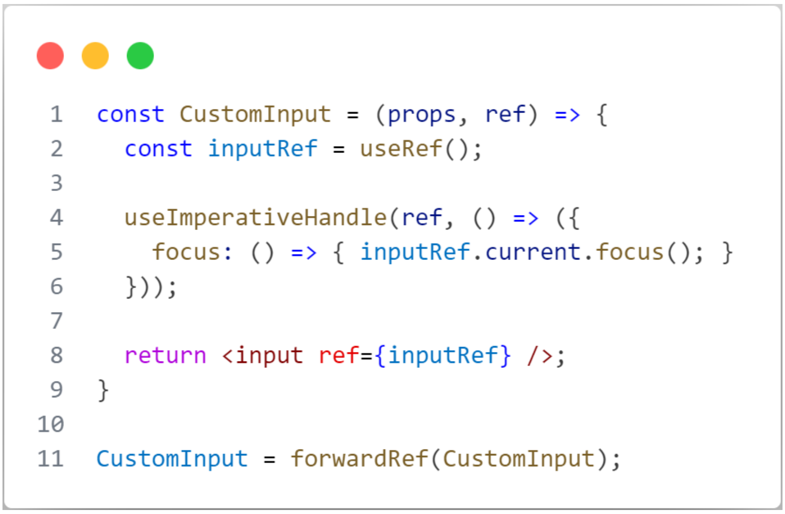
```
const AutoFocusInput = () => {
  const textInput = useRef(null);

  useEffect(() => {
    textInput.current.focusTextInput();
  }, []);

  return <input type='text' ref={textInput} />;
};
```

React will assign the ***"current"*** property with the DOM element when the component *mounts*, and assign it back to ***null*** when it *unmounts*. For *class components*, ***ref*** updates happen before ***componentDidMount()*** or ***componentDidUpdate()*** lifecycle methods.

## useImperativeHandle

This hook customizes the instance value that is exposed to parent components when using **ref**. The *useImperativeHandle* should be used with ***forwardRef***.

*Syntax*:
*useImperativeHandle(ref, createHandle, [deps])*



```
const CustomInput = (props, ref) => {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    focus: () => { inputRef.current.focus(); }
  }));

  return <input ref={inputRef} />;
}

CustomInput = forwardRef(CustomInput);
```

In above code, a parent component that renders **<CustomInput ref={inputRef} />** would be able to call ***inputRef.current.focus()***.

## useLayoutEffect

This hook is identical to ***useEffect***, but it fires **synchronously** after all DOM mutations. Use this hook to read layout from the DOM and synchronously re-render. Updates scheduled inside *useLayoutEffect* will be **flushed** synchronously, before the browser has a chance to paint.

Prefer the standard *useEffect* when possible to avoid blocking visual updates.

## Summary

1. Hooks, added in **v16.8**, allows adding state and other react features in function components. Hooks **don't** work in *class components*.
2. **Rules of Hooks**: Only call at **top level** and can be called from *function components*.
3. The ***useState()*** hook allows adding *state* to function components. Returns a **pair** with **current value** and an **update function**. State updates may be **batched** for performance benefits.
4. The ***useEffect()*** is used to perform *side-effects/lifecycle events*. It is equivalent to *componentDidMount()*, *componentDidUpdate()* and *componentWillUnmount()*.
5. The ***useContext()*** is used to access **Context** values inside the function component.
6. The ***useReducer()*** hook, an alternative to *useState()*, used to handle complex state logic. It optimizes performance of the component.
7. The ***useCallback()*** hook is used to create **memoized** callbacks, which is called on change of dependencies. The ***useMemo()*** hook can be used as an alternative to this, to achieve further performance benefits.
8. The ***useRef()*** hook is used to handle uncontrolled elements.
9. The ***useImperativeHandle()*** is used with *forwardRef()* to handle parent **ref**.
10. The ***useLayoutEffect()***, is identical to *useEffect*, but it fires **synchronously** after DOM updates.

CHAPTER 11

# React Router

## Preface to React Router

### What is React Router

*Routing library, keeps UI in sync with URL*

### BrowserRouter

*Uses HTML5 history API, follows standard URL structure*

### HashRouter

*Uses the hash of the URL, prefixed with # symbol*

### Routes, Route, RoutesConfig, Outlet

*Used to define routes, <Outlet> used to render matched Component*

### Nested Routes

*Create UI with persistent navigation, changing inner section with URL*

### Index Route

*Default child route with no path, renders in parent <Outlet>*

### Navigation

*Changing URL using <Link> or useNavigation()*

### Sharing Data

*Sharing URL data using useLocation(), useParams(), useSearchParams()*

## React Router

React Router is a routing library built on top of React that updates the UI, keeping it in sync with the URL. We'll see the implementation details for **version 6**.

### Installation

*npm install react-router-dom@6*

### BrowserRouter

This is a <*Router*> that uses the *HTML5* ***history*** API (*pushState*, *replaceState* and *postState* events) to keep the UI in sync with the URL. URLs built with BrowserRouter follow the standard URL structure. For example: ***"/language/React"***

An alternative to this is **<HashRouter>**, that uses the **hash** portion of the URL (i.e., *window.location.hash*).

> **Note**: *Hash history does not support* ***location.key*** *or* ***location.state****. These were supported in older versions, but it had some issues for implementing a couple of edge cases.*

<*BrowserRouter*> is the preferred way, but it may require some configuration changes to the server, to handle the **refresh of the page**.

On a server, it may be required to configure URL routing to **by-pass the server-side routing** and delegate the URL to the react application, so that BrowserRouter can render the expected UI.

### <Routes>

This is a parent route with child routes.

### <Route>

A Route Element of the structure *<Route path element >*, which is used to define a route in the route config.

### Route Config

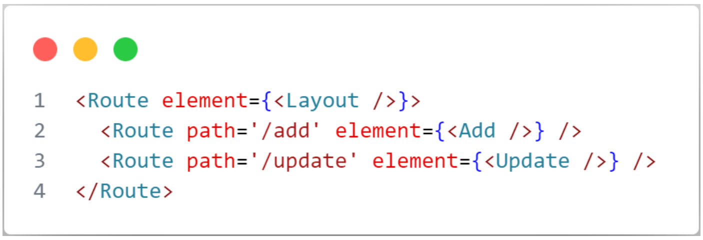
This is a tree of **route objects** that will be ranked and matched against the current location to create a branch of **route matches**.

### <Outlet>

It is a component that renders the matched routes element.

### Layout Route

This is a parent route **without a path**, used exclusively for grouping child routes inside a specific layout.



```
<Route element={<Layout />}>
  <Route path='/add' element={<Add />} />
  <Route path='/update' element={<Update />} />
</Route>
```

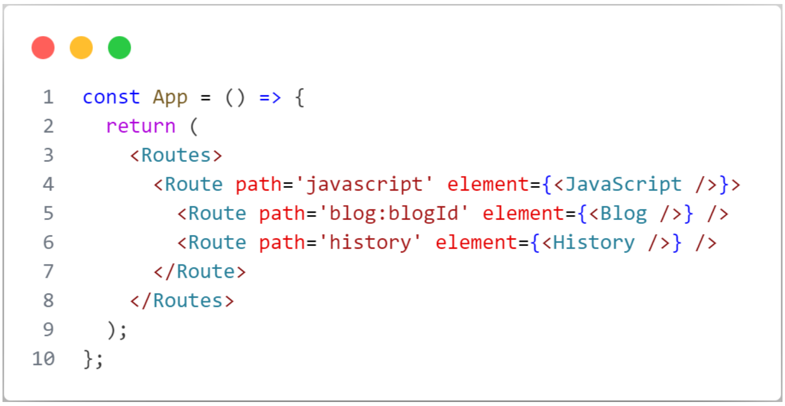
### Not Found Route

When no route matches with URL, we can render a not found component using the **path="*"**.

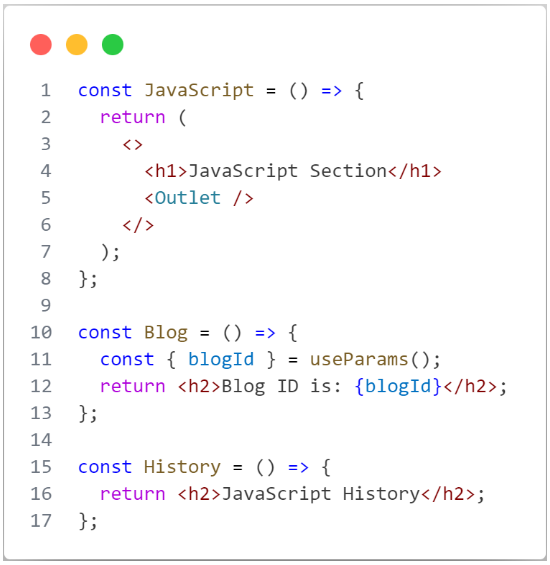
*<Route path="*" element={<NotFound />} />*

## Nested Routes

Routes can be nested inside another Route and their paths will also nest. Nested routes are perfect for creating UI that has persistent navigation in layout with an inner section that changes with the URL.



```jsx
const App = () => {
  return (
    <Routes>
      <Route path='javascript' element={<JavaScript />}>
        <Route path='blog:blogId' element={<Blog />} />
        <Route path='history' element={<History />} />
      </Route>
    </Routes>
  );
};
```



```jsx
const JavaScript = () => {
  return (
    <>
      <h1>JavaScript Section</h1>
      <Outlet />
    </>
  );
};

const Blog = () => {
  const { blogId } = useParams();
  return <h2>Blog ID is: {blogId}</h2>;
};

const History = () => {
  return <h2>JavaScript History</h2>;
};
```

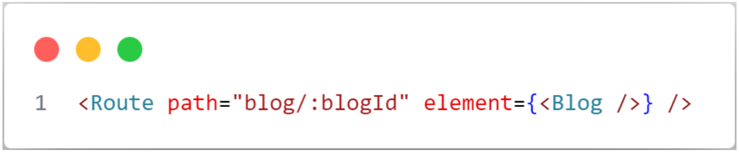
The route config will be defined with below given routes.

- “/javascript”
- “/javascript/blog/:blogId”
- “/javascript/history”

Observe that ***“/javascript”*** is prefixed to both child routes.

## Dynamic Segments

Segments of the URL can be dynamic placeholders that are parsed and provided to various APIs. The dynamic segments are added by prefixing with **“:”**(*colon*) symbol.

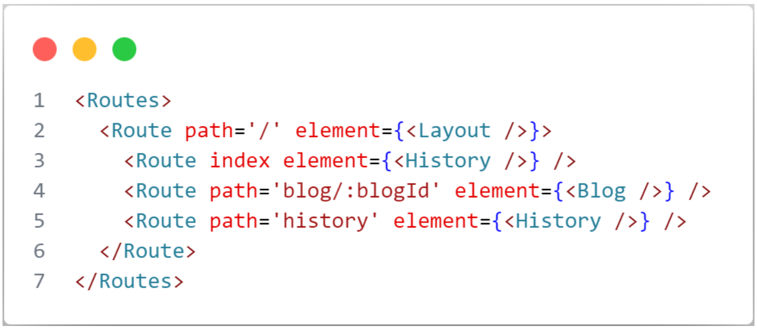


```
<Route path="blog/:blogId" element={<Blog />} />
```

**Note**: React Router’s nested routes were inspired by the routing system in *Ember.js circa 2014*.

## Index Route

A child route with **no path** that renders in the *parent’s <Outlet>* for the parent’s URL. Index routes can be thought of as the **“default child route”**. When a parent route has multiple children, but the URL is at the parent’s path, the index route will be displayed in the <Outlet>.

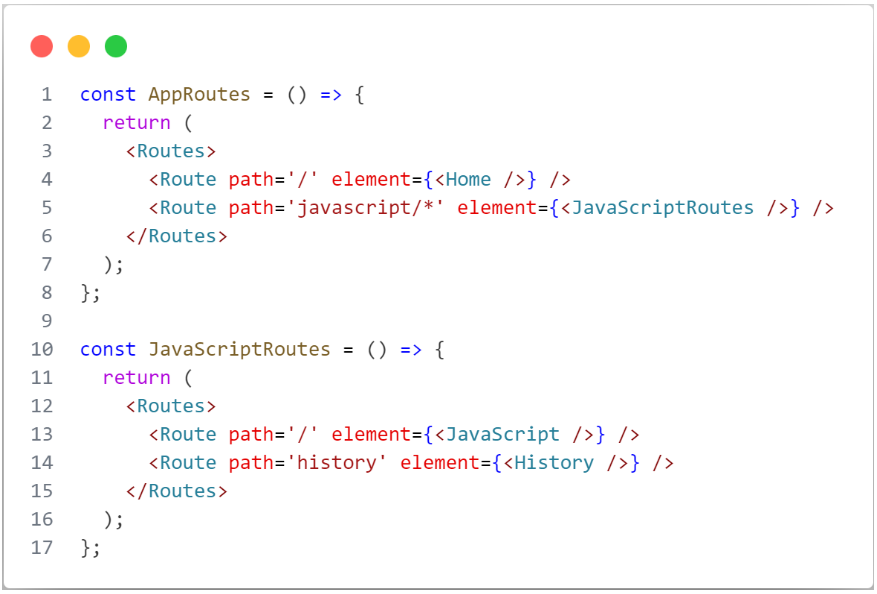


```
<Routes>
  <Route path='/' element={<Layout />}>
    <Route index element={<History />} />
    <Route path='blog/:blogId' element={<Blog />} />
    <Route path='history' element={<History />} />
  </Route>
</Routes>
```

## Descendant Routes

As we have learnt that routes can be nested, which means we can have multiple **<Routes>** elements in a *component tree*. Route with **path ending with "/*"** is called a descendant path, which tells React Router to use the given URL as the parent of the descendant routes.

Descendant Routes can be used to split routes in multiple files. This is helpful in defining module level routes.



```
const AppRoutes = () => {
  return (
    <Routes>
      <Route path='/' element={<Home />} />
      <Route path='javascript/*' element={<JavaScriptRoutes />} />
    </Routes>
  );
};

const JavaScriptRoutes = () => {
  return (
    <Routes>
      <Route path='/' element={<JavaScript />} />
      <Route path='history' element={<History />} />
    </Routes>
  );
};
```

## Navigation

When the URL changes, it is called *navigation*. There are two ways to navigate in React Router.

- ***<Link>*** Component
- ***useNavigate()*** Hook

## <Link>

A component used to render a link. It renders a **<a href>** tag making it web accessible compliant. React Router will prevent the browser's default behavior and tell the **history** to push a new entry into the **history stack**. When the user clicks on it, the **location** changes and the new matches will be rendered.

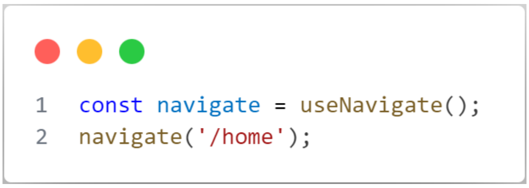
*<Link to="/home">Home</Link>*

### Relative Link

Relative <Link to> values (that **do not begin with a "/"**) are relative to the path of the route that rendered them.

## useNavigate()

This hook returns a *function* which is used to change the URL programmatically.



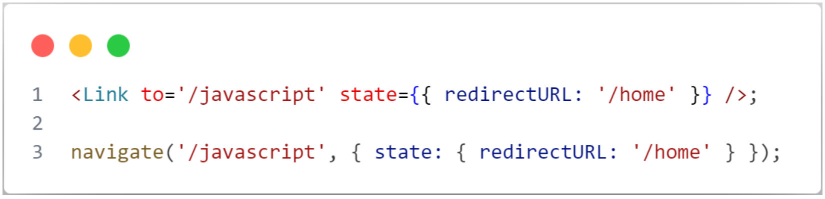
```
const navigate = useNavigate();
navigate('/home');
```

## Sharing/Accessing URL Data

A route may contain *location*, *state*, *route parameters* and *query string/search* parameters. React Router provides hooks for accessing these values.

### Adding state to URL

Values added in *state* are not visible on URL, instead these values are hidden that can be accessed programmatically.



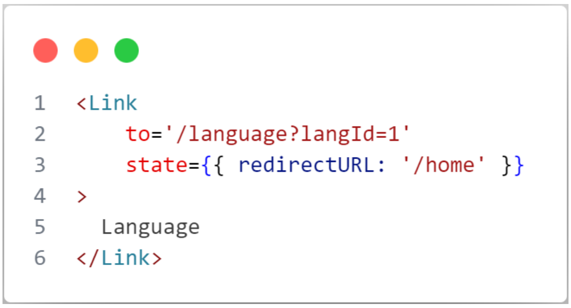
```
<Link to='/javascript' state={{ redirectURL: '/home' }} />;

navigate('/javascript', { state: { redirectURL: '/home' } });
```

## useLocation()

This hook returns a **location** *object* containing all related information for the current route. It contains *hash*, *key*, *pathname*, *search* and *state* attributes of the current route.

*const location = useLocation();*

*For below link the location object will hold below properties*:



```
<Link
    to='/language?langId=1'
    state={{ redirectURL: '/home' }}
>
  Language
</Link>
```

*location = {*

*__pathname__: "/language", __search__: "?langId=1",*

*__hash__: "", __state__:{ redirectURL : "/home" }, __key__: "default"*

*}*

## useParams()

This returns an *object* containing all the *route parameters* (if any). Route parameters are defined using **"/:key"** format.

*Example of route parameters*:

*<Route __path="language/:langId/:topicId"__ element={<Language />} />*

*Navigation link can be written as follows*:
*<Link **to="/language/1/20"**>Language</Link>*

*const params = useParams();*
*console.log(params);*

For given Link URL, params will contain below given values:
***{ langId:1, topicId: 20 }***

## useSearchParams()

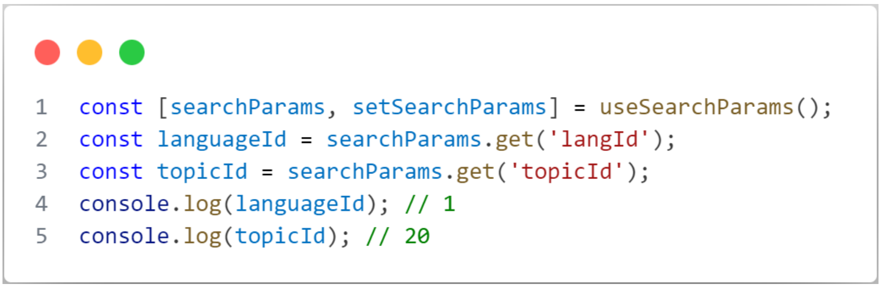
This hook is used to **read and modify** the query string parameters of the URL. It returns an **array of two values**: the current location search parameters and a *function* that may be used to update them.

*const [searchParams, setSearchParams] = useSearchParams();*

**setSearchParams** *function* works like *navigate*, but only for the search portion of the URL.

*Example of navigation link with search parameters*:
*<Link **to="/language?langId=1&topicId=20"**>Language</Link>*

The component matched with the above URL can read the query parameter values.



```
const [searchParams, setSearchParams] = useSearchParams();
const languageId = searchParams.get('langId');
const topicId = searchParams.get('topicId');
console.log(languageId); // 1
console.log(topicId); // 20
```

## Summary

1. **React Router** is a routing library to keep the UI in sync with the URL.
2. The **<BrowserRouter>** uses *HTML5 history* API.
3. **<Routes>** is the parent route for child <Route>. The **<Route>** accepts *route URL* as ***path*** and a component to render as ***element*** *prop*.
4. The *Route config* is a tree of route objects that will be ranked and matched against the current location.
5. The ***<Outlet>*** component is used to render the matched routes element.
6. The **Layout Route** is used for **grouping** child routes in specific layouts.
7. Not found component can be rendered for **path="*"**.
8. **Index Route** is a child route with **no path**, which would be the **default child route** and is rendered in the *parent's <Outlet>*.
9. **Descendant Routes** are routes with **path ending with "/*"**, is used to *split routes in multiple files* for implementing module level routes.
10. When the URL changes, it is called **Navigation**. React Router provides ***<Link>*** and ***useNavigate()*** for implementing navigation.
11. The ***useLocation()***, ***useParams()*** and ***useSearchParams()*** hooks are used to read url information like location, route parameters, query parameters, etc.

## Glossary

| **Term** | **Definition** |
|---|---|
| SPA | Single Page Application, that loads a single HTML page. |
| DOM | Document Object Model is a programming interface for the web. Represents documents as nodes and objects. |
| Virtual DOM | A lightweight JavaScript representation of the DOM. |
| React Fiber | A new reconciliation engine in React 16. Main goal is to enable incremental rendering of virtual DOM. |
| JSX | A syntax extension to JavaScript, enables writing HTML with JavaScript, and produces React elements. |
| Components | Isolated, reusable building blocks of UI, returns a React element to be rendered on page. |
| Props | Input values passed to the component, read-only data passed down from parent to child component. |
| State | Local values of the component, preserves values between renders. |
| Keys | A string or number that helps identify items in a list. It should be unique among sibling items. |
| Synthetic Events | React defined cross-browser compatible events, according to W3C specs. |
| Controlled Components | Components whose value is controlled by React. It accepts input props and update values via event handlers. |
| Uncontrolled Components | Components whose value is controlled by DOM. React refs are used to access DOM nodes. |
| Refs | Used to access DOM nodes or react elements in uncontrolled components. |
| Reconciliation | The algorithm React uses to diff one tree with another to determine which parts need to be changed. |

# References

https://ui-geeks.in an online UI learning platform.

https://react.dev React Js official documentation website.

https://developer.mozilla.org an open-source collaborative project documenting Web platform technologies.

# Where to go from here

Mastering the topics explained in this book is a great step towards your goal of deep diving React and React-Router.

I would suggest continuing your journey with the below action items.

Learn more about **Looping in JSX**, **Conditional Rendering**, **Render Props**, **Prop Drilling**, **DOM**, **Forms**, **Event Handling**, **Hooks**.

Learn how to apply styles to React Components using **CSS/SCSS/LESS**, **Styled Components**, **CSS in JS**.

Learn how to *manage state* using state management libraries like **Redux**, **MobX**, **Recoil**, etc.

Start building simple React applications like **Counter** App, **Calculator**, **Tic-Tac-Toe game**, **To-Do** App, etc.

Learn how to test React applications using **Jest**, **Enzyme**, **React Testing Library**.

Learn application frameworks built on top of React like **Next.Js**, **Gatsby**, etc.

# Liked the Content

If you liked the content of the book and want to support writing more useful content. Your smallest support would keep me motivated to keep writing.

**Support Us**

# Thank You

*A book is only the beginning for a writer. A reader completes it, and you are an example of this. Thank you for taking the time and reading the book.*

Would appreciate it if you can recommend this to your near and dear ones.

**Sunil Kumar**

**Cell**: +91 9818234200
**Mail**: skumar.mca2010@gmail.com
**LinkedIn**: sunil-kumar-83146843