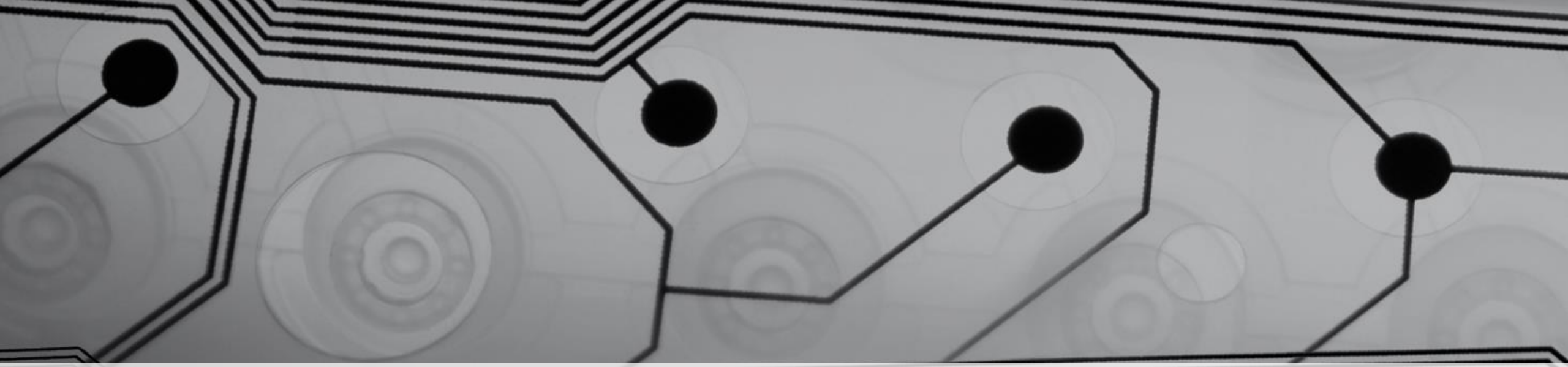


React.js

2020/21 COMP3322 Modern Technologies on WWW

Contents

- Introduction to React.js
- JSX
- React Elements
- Components
  - Function components and Class components
  - Stateless components and stateful components
  - Handling Events
  - Conditional Rendering
  - Controlled Components
- Virtual DOM

Intro to React.js

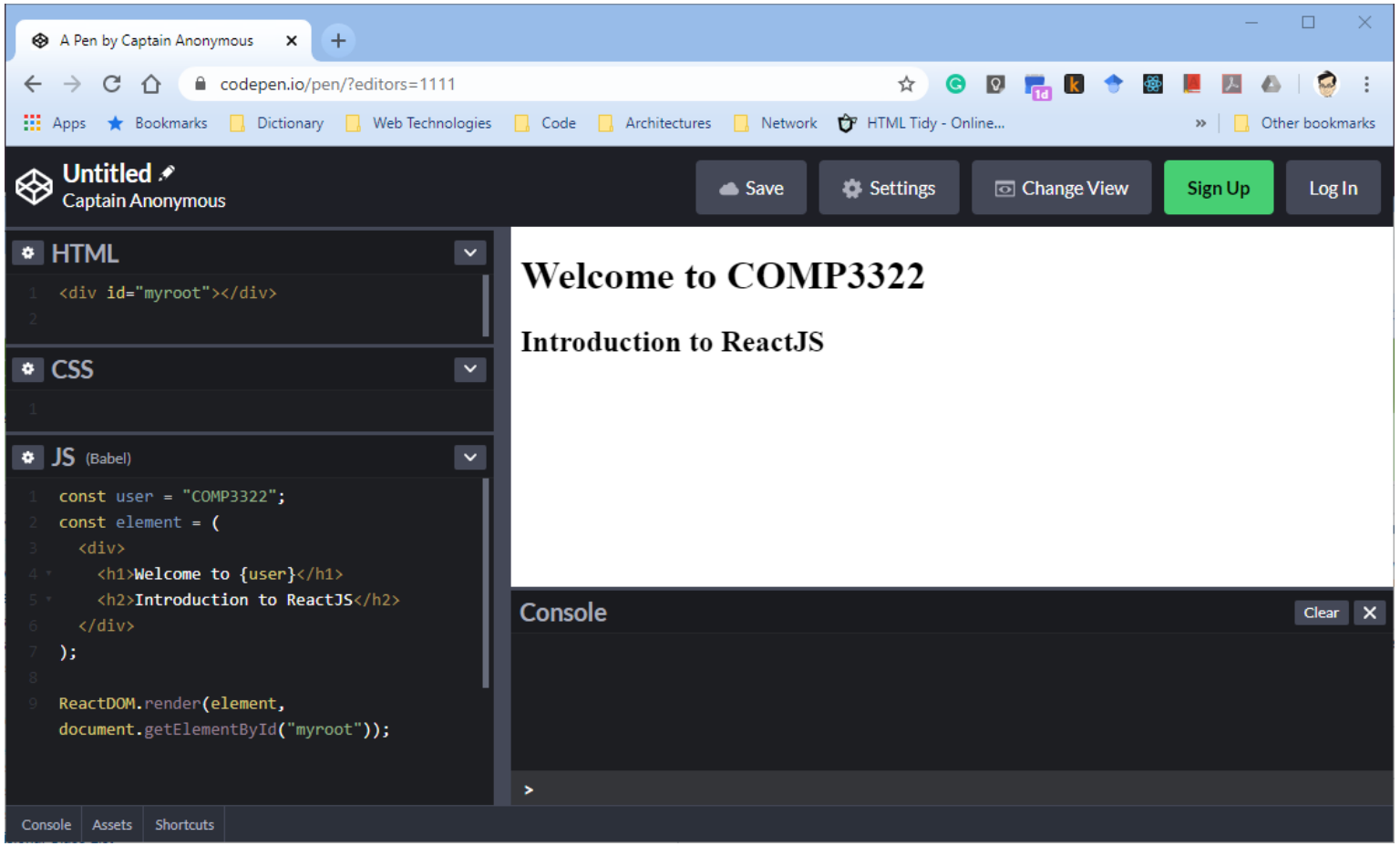
- What is React?
  - React is a popular JavaScript **front-end library** for building interactive user interfaces. It is not a framework.
- Why React?
  - Single page Web applications usually involve thousands lines of code. With traditional DOM structure, it is a challenging task to make changes in them.
  - In 2013, Facebook/Instagram released the React project – a UI **component-building** library to build user interfaces (UI) .
    - People consider React **is opinionated** as it defines the "right way" to design UI.
  - In the component-based approach, the entire application is divided into a small group of code fragments, which is known as components.

Intro to React.js

- Component-building
  - Allows us to compose complex UIs **from small and isolated pieces of code** called “components”.
  - Each component has its own structure, methods as well as API
  - Components work independently from one another and can be nested within other components to build complex UI.
  - Encourages us to **create reusable UI components.**

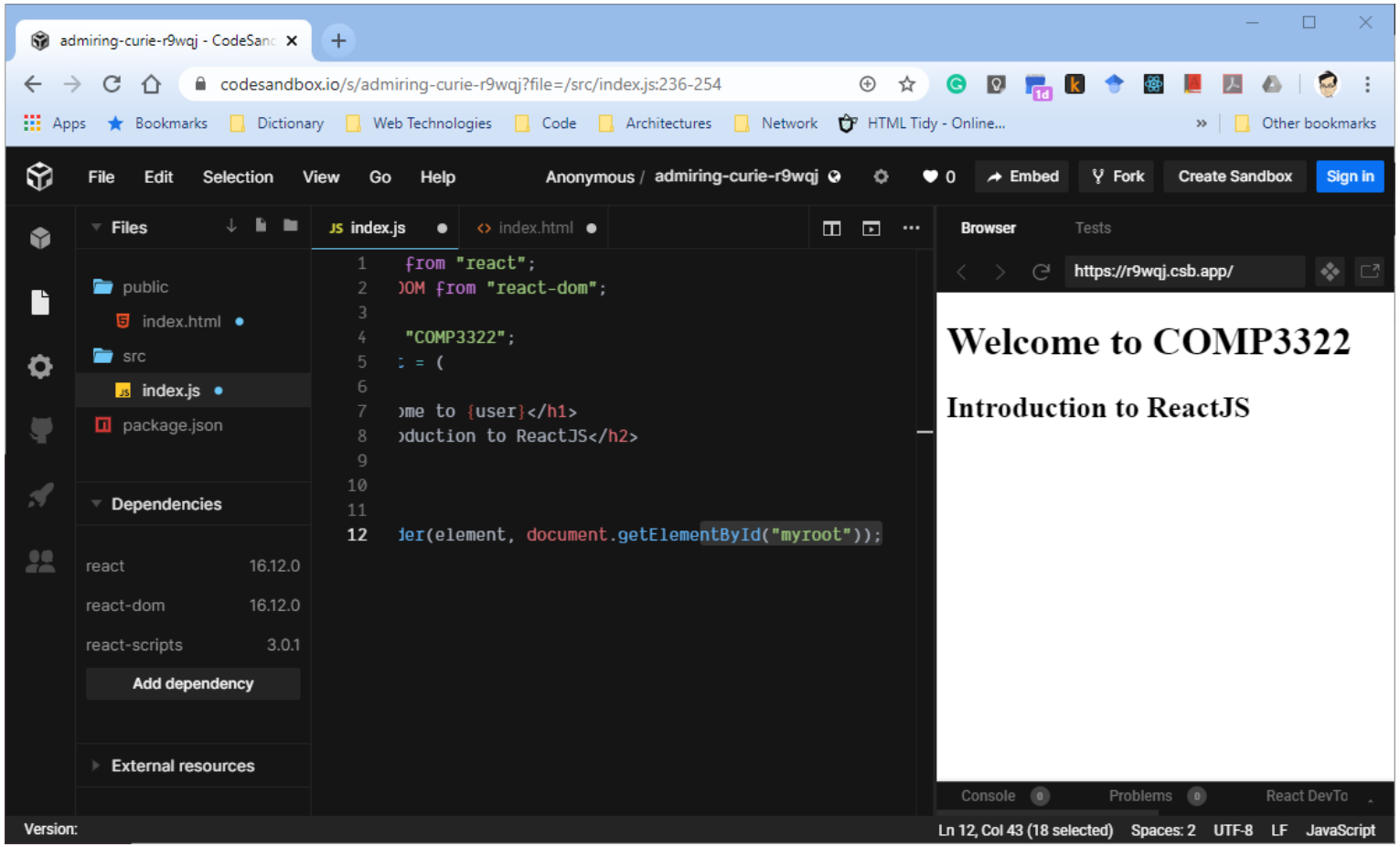
Online Development Platforms

- To kickstart your development, some online platforms are the suitable choices:
- Codepen.io



Online Development Platforms

- CodeSandbox (https://codesandbox.io/)



https://reactjs.org/docs/add-react-to-a-website.html

Add React to Web Page

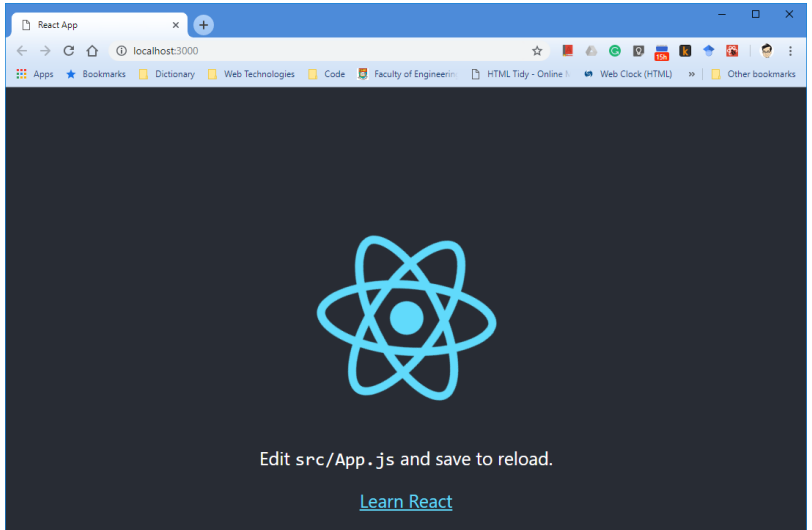
- We can add React to our existing website by including React-related JavaScript libraries to your page
- You can download an example html file from here:
  - https://raw.githubusercontent.com/reactjs/reactjs.org/master/static/html/single-file-example.html
  - That helps you kickstart your exploration.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Adding React</title>
    <script
src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
    <!-- Don't use this in production: -->
    <script src="https://unpkg.com/babel-
standalone@6.15.0/babel.min.js"></script>
  </head>
  <body>
    <!-- This is the place where your React element is rendered -->
    <div id="myroot"></div>

    <!-- This contains your React code -->
    <script type="text/babel">
      const course = "COMP3322";
      const element = (
        <div>
          <h1>Welcome to {course}</h1>
          <h2>Introduction to React.js</h2>
          <p>This is a simple React component</p>
        </div>
      );

      ReactDOM.render(element, document.getElementById('myroot'));

    </script>
  </body>
</html>
```

https://github.com/facebook/create-react-app

Create React App

- With your installed Node.js, to create a new app, you may use
  - with npm 5.2+ and higher `npx create-react-app my-app`
  - with npm 6+ `npm init react-app my-app`
- To start the react app, use 'npm start'
- You will use this method to create a React app in Workshop 5.




```
my-app
├── README.md
├── node_modules
├── package-lock.json
├── package.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

Basic Features

- React Features
  - JSX
  - React Elements
  - Components
    - Props and States
  - Virtual DOM

JSX

- JSX stands for JavaScript eXtension or JavaScript as XML.
- JSX is an XML-like syntax. The syntax is intended to be used by **preprocessors** to **transform JSX code** found in JavaScript files into standard **JavaScript objects** that a JavaScript engine will parse.
- It is not a must to use JSX to build React.js Apps
  - Because we can directly call **React.createElement()** to build the **React elements**.
  - If JSX is used, this is the task done by the transpiler.
- JSX greatly simplifies coding and maintenance of React Apps; this simplification is especially evident for declaring **deeply nested component relationships**.

With and Without JSX

```
<!-- This contains your React code -->
<script type="text/babel">
  const course = "COMP3322";
  const element = (
    <div>
      <h1>Welcome to {course}</h1>
      <h2>Introduction to React.js</h2>
      <p>This is a simple React component</p>
    </div>
  );

  ReactDOM.render(element, document.getElementById('myroot'));
</script>
```

```
<!-- This contains your React code -->
<script>
  const e = React.createElement;

  const course = "COMP3322";
  const element = e('div', null,
    e('h1', null, "Welcome to ", course),
    e('h2', null, "Introduction to React.js"),
    e('p', null, "This is a simple React component")
  );

  ReactDOM.render(element, document.getElementById('myroot'));
</script>
```

JSX

- Here is an example of JSX that being assigned to a variable.

```
const heading = <h1 className="main">Hello, React</h1>;
```

This is a JSX expression

- JSX is neither a string nor HTML, but it looks like HTML!!
- We can create and use our own XML-like tags.
- JSX is actually **closer to JavaScript**, not HTML, so we have to take note of the followings when writing JSX.
  - className is used instead of class for adding CSS classes, as class is a reserved keyword in JavaScript.
  - Properties and methods in JSX are **camelCase** – onclick will become onClick.
  - Self-closing tags must end in a slash – e.g. <img />

Embedding JS Expressions in JSX

- Any valid JavaScript expressions can also be embedded inside JSX using curly braces.

```
const name = 'James';
const heading = <h1>Hello, {name}</h1> ;
```

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}
const heading = (
  <h1>
    Hello, {(user)? formatName(user) : Stranger}!
  </h1>
);
```

Recommend wrapping JSX in parentheses if split it over multiple lines

Nested JSX tags

- If a JSX tag is empty, we can close it with />, like XML.

```
const element = <div className="sidebar" />
```

- JSX tags may contain nested tags.

```
const element = (
  <div>
    <h1>Header</h1>
    <h2>Content</h2>
    <p>This is the content!!!</p>
  </div>
);
```

If we want to return more elements, we need to wrap it with one container element.

https://reactjs.org/docs/react-api.html#createelement

Compile JSX

- JSX code is compiled into JavaScript code by transpiler such as Babel. Fundamentally, compile into the React.createElement() calls.

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

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

- The first part of a JSX tag determines the type of the React element.
  - If the first character of the JSX tag is a lowercase, it refers to the **standard HTML tag** element - DOM element.
  - If the first letter is capitalized, it refers to the Component element, which is the **user-defined** React element.

React Elements

- React.createElement() will return a React element.

```
React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

```
{
  type: 'h1',
  props: {
    className: "greeting",
    children: "Hello, world!"
  }
};
```

- React elements are the **smallest building blocks** of React apps.
- React element is a **plain object**, which **describes** what we want to see **about that element on the screen**.
  - It contains only information about the element type (for example, a 'p'), its properties (for example, its color), and any child elements inside it.
- **React elements are immutable** - once you create an element, you cannot change its children or attributes.

React Elements - DOM Elements & Component Elements

- The simplest type of React Elements is the **DOM element**.
  - It **represents** a DOM node, which React is going to render.

```
{
  type: 'h1',
  props: {
    className: "greeting",
    children: "Hello, world!"
  }
};
```

```
<h1 class="greeting">Hello, world!</h1>
```

- When the type of a React element is a Component, it is called a **Component element**.
  - An element describing a component is also an element, just like an element describing the DOM node. They can be **nested and mixed** with each other.
  - When React sees a component element, it knows to **ask that component what element it renders to**, given the corresponding props.
  - React will **repeat this process** until it reaches to the underlying DOM elements for every component encapsulate in the react element.

Rendering Elements

- React elements are just the descriptions of UI components.
- To render a React element, we use the **ReactDOM.render()** method.
- We have to tell the render() method where to display the element.
  - Applications built with React usually have a single root DOM node.

Components

- Components are the **reusable** building blocks of the React app.
  - Each component is responsible for describing a small, reusable piece of HTML.
  - A component may maintain **internal state**, **determines** how that component **renders & behaves**.
- A React component is a JavaScript **class** or **function** that optionally **accepts inputs** i.e. properties (props) and returns a **React element** that describes how a section of the UI (User Interface) should appear.
- Always start component names with a **capital letter**.
  - React treats components starting with lowercase letters as DOM tags.
- In React, we have mainly two types of components.
  - Functional Components
  - Class Components

Props and State

- Components need data to work with. There are two different types of data accessible to a component:
  - Props and State
- **Props** - It is **an object** passes as an input to a component.
  - React philosophy is that **props should be immutable** and **top-down**.
    - What this means is that a parent component can pass on whatever data it wants to its children as props, but the child component cannot modify its props.
- **State** - This is **an object** that is **owned by** the component where it is declared.
  - **Its scope** is limited to the current component.
  - A component can **initialize its state** and **update it** whenever necessary. The state of the parent component **usually ends up being props of the child component**.

Function and Class Components

- The simplest way to define a component is to write a JavaScript function.

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

  - All function components **accept a single props object argument** and returns a React element.

- Another way to define a component is by using an ES6 JavaScript class.

A class component must include **render()**, and the **return** can **only return** one parent element.

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

- The above two components are equivalent from React's point of view.

Rendering a Component

HTML

```
<div id="myroot"></div>
```

CSS

JS (Babel)

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

const element = <Greeting name="Everyone" />;
ReactDOM.render(element, document.getElementById('myroot'));
```

Hello, Everyone!!

- React calls the Greeting component with {name: "Everyone"} as the props.
- The Greeting component returns a <h1>Hello, Everyone!!</h1> element as the result.
- React DOM efficiently updates the DOM to match <h1>Hello, Everyone!!</h1>.

HTML

```
<div id="myroot"></div>
```

CSS

JS (Babel)

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


const element = <Greeting name="Everyone" />;
ReactDOM.render(element, document.getElementById('myroot'));
```

Hello, Everyo

Class Components

- Here is the recommended method to create a class component.

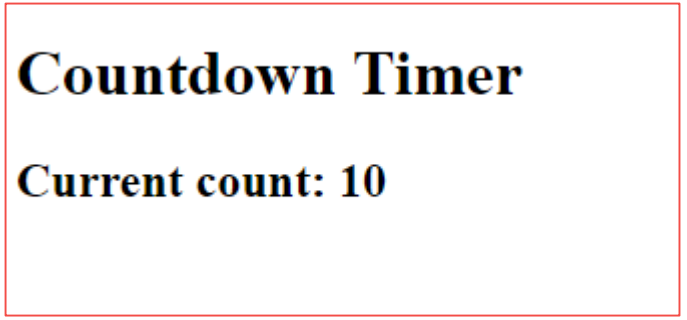
```
class Greeting extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <h1>Hello, {this.props.name}!!</h1>;
  }
}
```

- A constructor is defined which accepts props as input.
  - However, the constructor is **optional**
- Inside the constructor, we **must call** super(props) to pass down whatever is being inherited from the parent class.

Class Components - Stateful components

- The primary reason to choose class components over functional components is that they can have state.

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

  render() {
    return (
      <div>
        <h1>Countdown Timer</h1>
        <h2>Current count: {this.state.value}</h2>
      </div>
    );
  }
}
```



The state of a component is initialized in the constructor.

We can now access the state within the class methods including render()

Stateful Components

- We need to be able to **update the state** in an interactive application.
- ***Using state correctly***
  - To update the state, we'll use **this.setState()**, a built-in method for manipulating state.
    - Do not update the state directly.
    - Correct way: `this.setState({value: 9});`
  - Calling this.setState **causes React to re-render** your application and update the DOM, but updating this.state directly will not re-render the component.

Stateful Components

- ***Using state correctly***
  - State updates may be done asynchronously
    - React may batch multiple setState() calls into a single update for performance. Then the value of this.props and this.state may not be what you expected **at the time of calculation.**
    - Use the second form of setState() which receives the **previous state** as the 1st argument and **the props at the time the update is applied** as the 2nd argument.

```
this.setState({value: this.state.value-1});
```

Could work, but may suffer race condition.

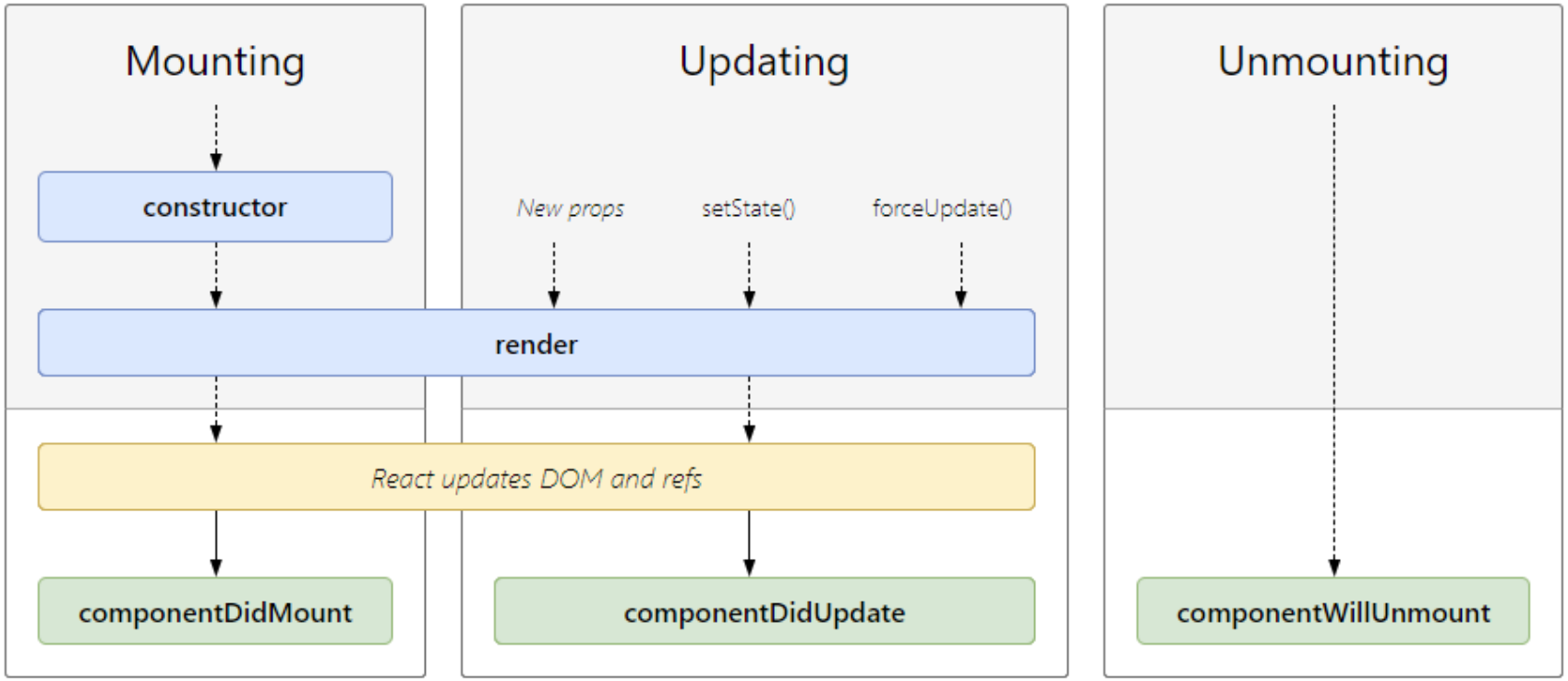
```
this.setState( (prevState, props) => {
        return {value: prevState.value-1};
      });
```

The correct way to make the update which depends on the previous state.

Component Lifecycle

- Each component has several "lifecycle methods" that allow us to run code at particular times in the process.

componentDid-Mount() is invoked immediately after a component is mounted (inserted into the tree).



componentDidUpdate() is invoked immediately after updating occurs.

componentWill-Unmount() is invoked immediately before a component is unmounted and destroyed.

Demo – A Countdown Timer

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

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

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    if (this.state.value > 0) {
      this.setState( (prevState, props) => {
        return {value: prevState.value-1};
      });
    } else {
      clearInterval(this.timerID);
    }
  }
```

set up a timer to call the component's tick() method once a second.

Update the value or stop the timer if value ≤ 0

```
  render() {
    return (
      <div>
        <h1>Countdown Timer</h1>
        <h2>Current count: {this.state.value}</h2>
      </div>
    );
  }
}

const element = <Countdown value='30' />;
ReactDOM.render(element,
document.getElementById('root'));
```

Handling Events

- Handling events with React elements is similar to handling events on DOM elements, except:
  - The name of the React events are in **camelCase** rather than in lowercase, e.g., onClick vs. onclick, onLoad vs. onload.
  - With JSX you **pass a function** as the event handler, rather than a string.
  - You cannot use "return false;" as an escape path to prevent default behavior of the element.

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

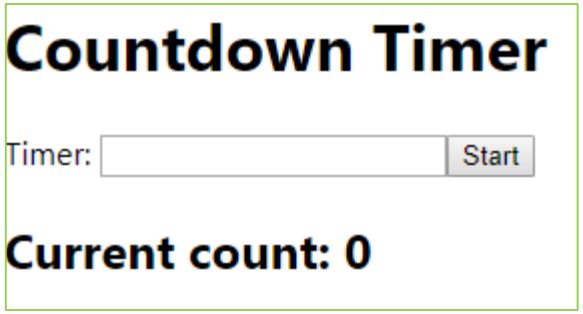
    - You must call preventDefault() to achieve that.
- It is interesting to see that React prefer adding inline listeners to the DOM elements rather than using addEventListener.

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

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

Handling Events

- When designing event handlers for the Class Components, a common pattern is to have the event handler (e.g., countdown) **as a method** in the component.
- Then, when installing the event handler in JSX callback, we use "this.countdown" to **refer to the method** in the class component.
- If the event handler **accesses the state** of the class component, it will encounter an error - the component's state is **undefined** when the function is actually called.
- It is because, by default, class methods in JavaScript are not bound to their class.
  - i.e., when the method is being called not within the class component scope, it cannot access the 'this'.

Demo - A Countdown Timer




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

  tick() {
    if (this.state.value > 0) {
      this.setState( (prevState, props) => {
        return {value: prevState.value-1};
      });
    } else {
      clearInterval(this.timerID);
      document.getElementById('timer').value = '';
    }
  }

  countdown() {
    var getinput = document.getElementById('timer').value;
    if (getinput == '') {
      alert("Please enter the time");
    } else {
      this.setState({value: getinput});
      this.timerID = setInterval(() => this.tick(), 1000 );
    }
  }
```

TypeError: Cannot read property 'setState' of undefined

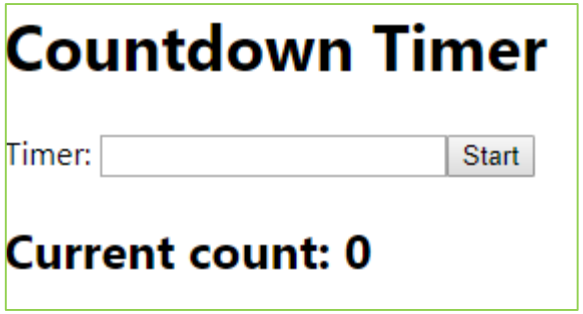
countdown
src/index.js:29

```
  26 |   if (!getinput) {
  27 |     alert("Please enter the time");
  28 |   } else {
> 29 |     this.setState({value: getinput});
  30 | ^   this.timerID = setInterval(() => this.tick(), 1000 );
  31 |   }
  32 | }
```

View compiled

```
  render() {
    return (
      <div>
        <h1>Countdown Timer</h1>
        <label for="timer">Timer: </label>
        <input id="timer" type="text" />
        <button onClick = {this.countdown}>Start</button>
        <h2>Current count: {this.state.value}</h2>
      </div>
    );
  }
}
```

A Countdown Timer

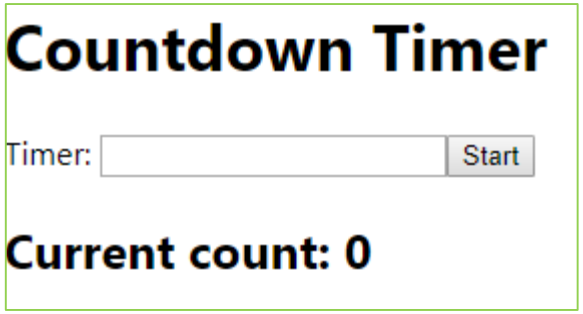



```
class Countdown extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 0};
    this.countdown = this.countdown.bind(this);
  }

  tick() {
    if (this.state.value > 0) {
      this.setState( (prevState, props) => {
        return {value: prevState.value-1};
      });
    } else {
      clearInterval(this.timerID);
      document.getElementById('timer').value = '';
    }
  }

  countdown() {
    var getinput = document.getElementById('timer').value;
    if (getinput == '') {
      alert("Please enter the time");
    } else {
      this.setState({value: getinput});
      this.timerID = setInterval(() => this.tick(), 1000 );
    }
  }
```

Using bind(), we add the scope of 'this' class component to the scope of the countdown method.

```
  render() {
    return (
      <div>
        <h1>Countdown Timer</h1>
        <label for="timer">Timer: </label>
        <input id="timer" type="text" />
        <button onClick = {this.countdown}>Start</button>
        <h2>Current count: {this.state.value}</h2>
      </div>
    );
  }
}
```

A Countdown Timer




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

  tick() {
    if (this.state.value > 0) {
      this.setState( (prevState, props) => {
        return {value: prevState.value-1};
      });
    } else {
      clearInterval(this.timerID);
      document.getElementById('timer').value = '';
    }
  }

  countdown() {
    var getinput = document.getElementById('timer').value;
    if (getinput == '') {
      alert("Please enter the time");
    } else {
      this.setState({value: getinput});
      this.timerID = setInterval(() => this.tick(), 1000 );
    }
  }
```

```
  render() {
    return (
      <div>
        <h1>Countdown Timer</h1>
        <label for="timer">Timer: </label>
        <input id="timer" type="text" />
        <button onClick = {(e) =>
this.countdown(e)}>Start</button>
        <h2>Current count: {this.state.value}</h2>
      </div>
    );
  }
}
```

```
<button onClick = {this.countdown.bind(this)}>Start</button>
```

Conditional Rendering

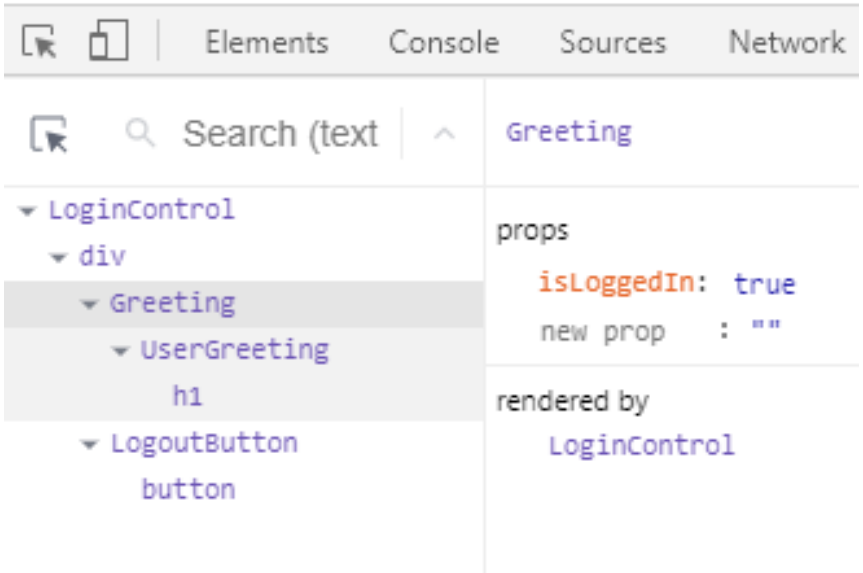
- As React is just a JavaScript library, we can use JavaScript **if** or the **conditional operator** to create elements representing the current state, and let React update the UI to match them.

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

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

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

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

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

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

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

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

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

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

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

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

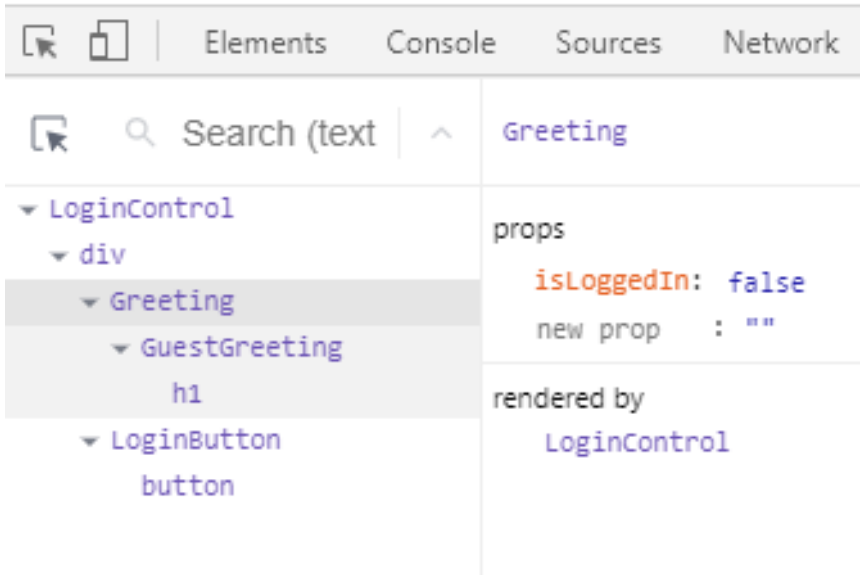
Welcome back!

Logout



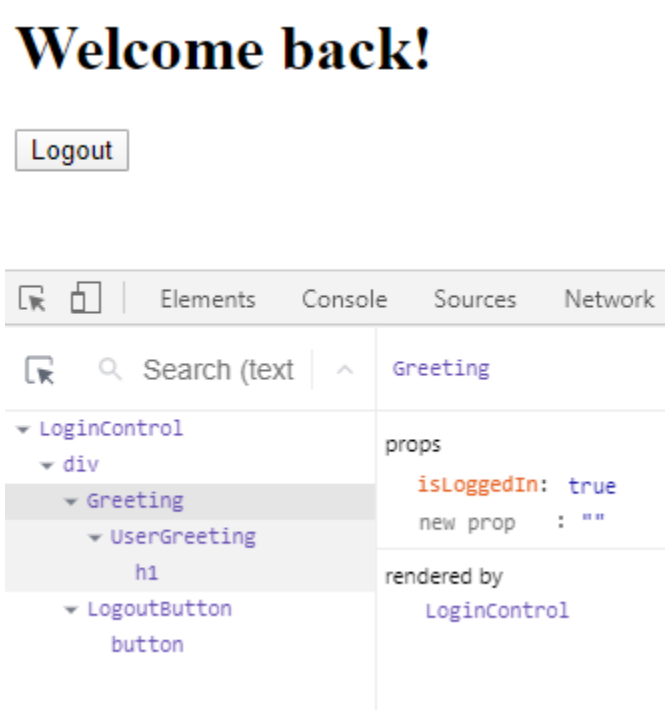
Please sign up.

Login



Communication Between Components

- Data sometimes needs to be able to move from one component to another component
- There are three basic cases
  - Parent component to child component
  - Child component to parent component
  - Between siblings components
- Parent to Child
  - This is the simplest scenario.
  - Simply use props to make the child inherit properties from its parent component



Communication Between Components

- Child to Parent
  - First, create a callback function in the Parent which will call by Child to pass back the data.
  - Secondly, pass this callback function to the child as a props.
  - Then, Child calls 'parentFunction' to pass the data.

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

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

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

```
button = <LogoutButton onClick={this.handleLogoutClick} />;
```

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

- Between siblings
  - Child1 to Parent and then Parent to Child2

Conditional Rendering

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

const messages = ['Msg1', 'Msg2', 'Msg3'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

Hello!
You have 3 unread messages.

- You may **embed any logical expressions in JSX** by wrapping them in curly braces.
- In JavaScript
  - (true && expression) ➔ always evaluates to expression
  - (false && expression) ➔ always evaluates to false
- Therefore, if the condition is true, the element right after && will appear in the output. If it is false, React will ignore and skip it.

```
const messages = [];
```

Hello!

Conditional Rendering

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

const messages = ['Msg1', 'Msg2', 'Msg3'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

Hello!
You have 3 unread messages.

- We can return null to hide a component from rendering.
- This is useful for a component to determine whether renders or returns a null depending on the value of props.

```
const messages = [];
```

Hello!

Form - Controlled Components

- In HTML, form elements such as <input>, <textarea>, and <select> typically **maintain their own state** and **update it based on user input**.
- How can React help maintaining the states of those form elements?
  - Controlled Component - A React component renders the form as well as controls what happens in the form when user input something.
- With a controlled component, we **associate a handler function** with an **input element**, which will be triggered when the user makes input to the form element.

The Countdown Timer

```
class Countdown extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 0};
    this.countdown = this.countdown.bind(this);
    this.getChange = this.getChange.bind(this);
  }

  tick() {
    if (this.state.value > 0) {
      this.setState( (prevState, props) => {
        return {value: prevState.value-1};
      });
    } else {
      clearInterval(this.timerID);
      document.getElementById('timer').value = 0;
    }
  }

  countdown() {
    if (this.state.value == 0) {
      alert("Please enter the time");
    } else {
      this.timerID = setInterval(() => this.tick(), 1000 );
    }
  }

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

```
  render() {
     return (
       <div>
         <h1>Countdown Timer</h1>
         <label for="timer">Timer: </label>
         <input id="timer" type="text"
onChange={this.getChange}/>
         <button onClick = {this.countdown}>Start</button>
         <h2>Current count: {this.state.value}</h2>
       </div>
     );
   }
}
```

Controlled Component

- React also adds the **value attribute** to <textarea> and <select> tags. In that way, a form can use the same way to capture the state changes like the text input.

```
<select>
  <option value="apple">Apple</option>
  <option value="grape">Grape</option>
  <option selected value="mango">Mango</option>
  <option value="melon">Melon</option>
  <option value="orange">Orange</option>
</select>
```

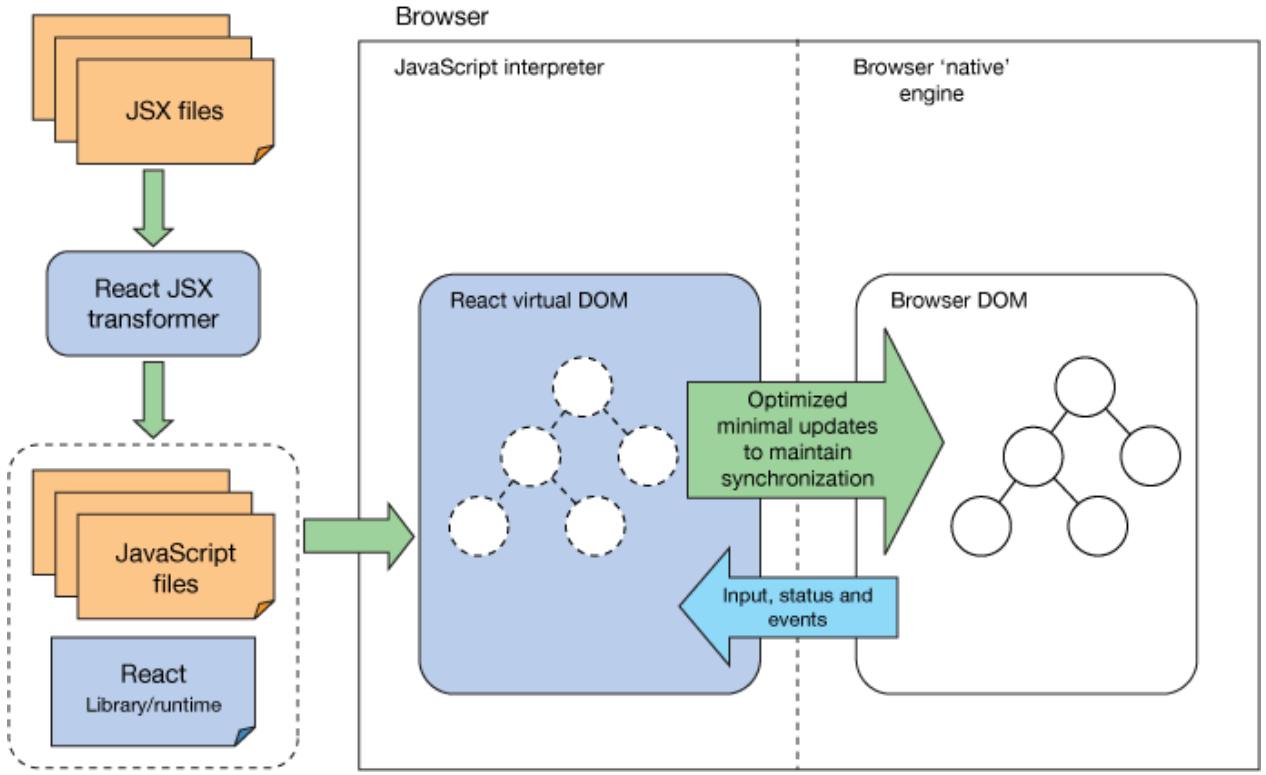
React, instead of using this selected attribute, uses a value attribute on the root select tag.

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

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

  handleInput(event) {
    const target = event.target;
    if (target.id == "select")
      this.setState({value: event.target.value});
    else {
      alert('Your favorite Fruit is: ' + this.state.value);
      event.preventDefault();
    }
  }
```

```
  render() {
    return (
      <form onSubmit={this.handleInput}>
        <label>Pick your favorite fruit: </label>
          <select id="select" value={this.state.value}
onChange={this.handleInput}>
            <option value="apple">Apple</option>
            <option value="grape">Grape</option>
            <option value="mango">Mango</option>
            <option value="melon">Melon</option>
            <option value="orange">Orange</option>
          </select>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

How does React.js work?

- Updating HTML element on a webpage involves using the DOM API. It needs to recalculate the layout and repaint the page.
- For website with many dynamic components, each of these updates triggers same set of computation which make the whole process inefficient.
- To optimize runtime performance, React components are first rendered into a managed virtual DOM.



Virtual DOM

- DOM manipulation is the heart of the modern, interactive web. Unfortunately, it is a lot slower than most JavaScript operations.
- In React, for every DOM object, there is a corresponding "virtual DOM object."
- When a React element is updated, the corresponding virtual DOM objects get updated.
  - This operation is much faster as in JavaScript.
  - Also multiple updates (of multiple elements) can be applied to the Virtual DOM.
- Once the virtual DOM has updated, then React compares the virtual DOM with a virtual DOM snapshot that was taken right before the updates. React then knows exactly which virtual DOM objects have changed.
- Now React just needs to inform the real DOM to update only those updated objects, with all the changes grouped into one.

Readings

- React Documentation
  - https://reactjs.org/docs/getting-started.html
  - Main Concept
    - 1. Hello World
    - 2. Introducing JSX
    - 3. Rendering Elements
    - 4. Components and Props
    - 5. State and Lifecycle
    - 6. Handling Events
    - 7. Conditional Rendering

References

- IBM Developer – Web Development
  - React: Create maintainable, high-performance UI components
    - https://developer.ibm.com/tutorials/wa-react-intro/
- Getting Started with React – An Overview and Walkthrough
  - https://www.taniarascia.com/getting-started-with-react/
- React: The Virtual DOM
  - https://www.codecademy.com/articles/react-virtual-dom