An Introduction to React

ITS290F

Getting Started

CodePen is assumed in running the React examples in this Lecture. Follow the steps below to config CodePen to work with React scripts.

1. Add the following CSS Link:
    - https://maxcdn.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css
2. Add the following JavaScript Links:
    - https://cdnjs.cloudflare.com/ajax/libs/react/16.8.3/umd/react.production.min.js
    - https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.8.3/umd/react-dom.production.min.js
    - https://unpkg.com/react-bootstrap@next/dist/react-bootstrap.min.js
3. Change the JavaScript Preprocessor to **`Babel`**

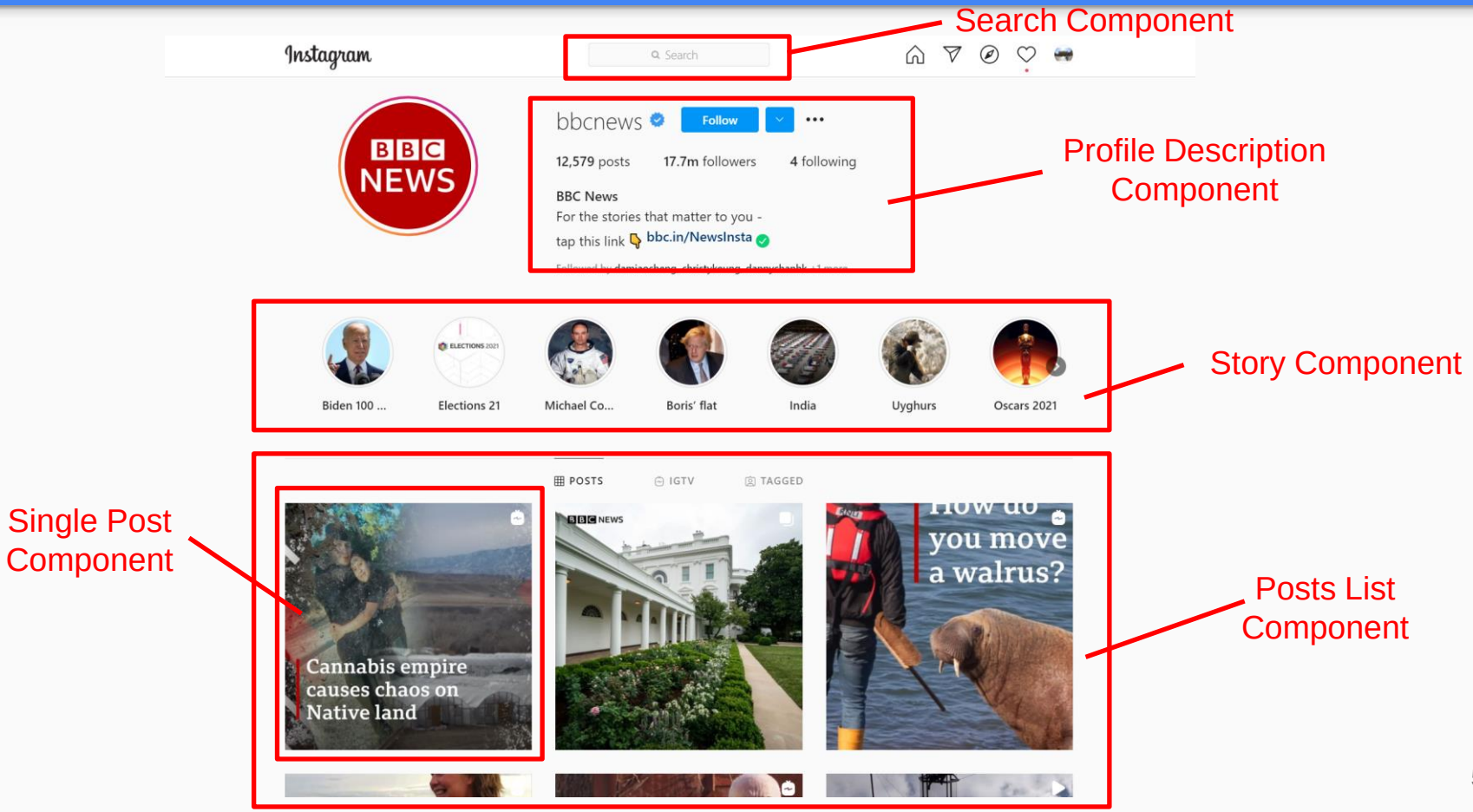
What is React?

- Reach is **a JavaScript library** for building fast and interactive user interfaces for the web as well as mobile applications.
- It is an open-source, **reusable component-based** front-end library
- In a model-view-controller architecture, React is the “view” which is responsible for how the app **looks and feels**.

History of React

- Created by Jordan Walke, a software engineer at Facebook
- First deployed on **Facebook's newsfeed in 2011** and later on **Instagram in 2012**
- Open-sourced at JSConf US in May 2013
- Facebook announced **React Fiber**, a new core algorithm of React library for building user interfaces in 2017

Example: An Instagram webpage which is entirely built using React



Why use React?

1. Simplicity
   - The **component-based** approach, **well-defined lifecycle**, and use of **just plain JavaScript** make React **very simple to learn**.
2. Easy to learn
   - Anyone with a basic previous knowledge in programming can easily understand React.
3. Native Approach
   - React can be used to create mobile applications (React Native). And it supports **extensive code reusability** is supported.

Why use React?

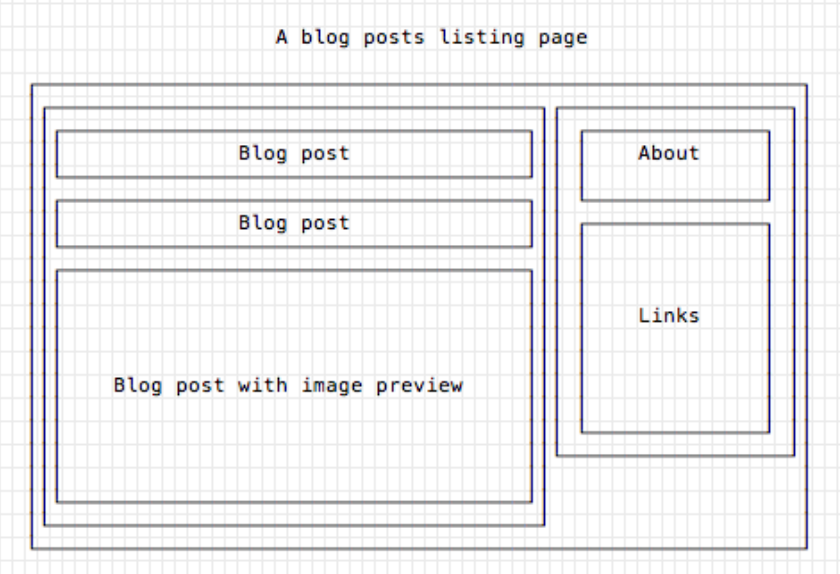
4. Data Binding
    - React uses **one-way data binding**. It's easier to debug self-contained components of large ReactJS apps.
5. Performance
    - React does not offer any concept of a built-in container for dependency. You can use various modules to inject dependencies automatically.
6. Testability
    - ReactJS applications are super **easy to test**. React views can be **treated as functions** of the state, so we can manipulate with the state we pass to the ReactJS view and take a look at the **output** and triggered actions, etc.

Main Features

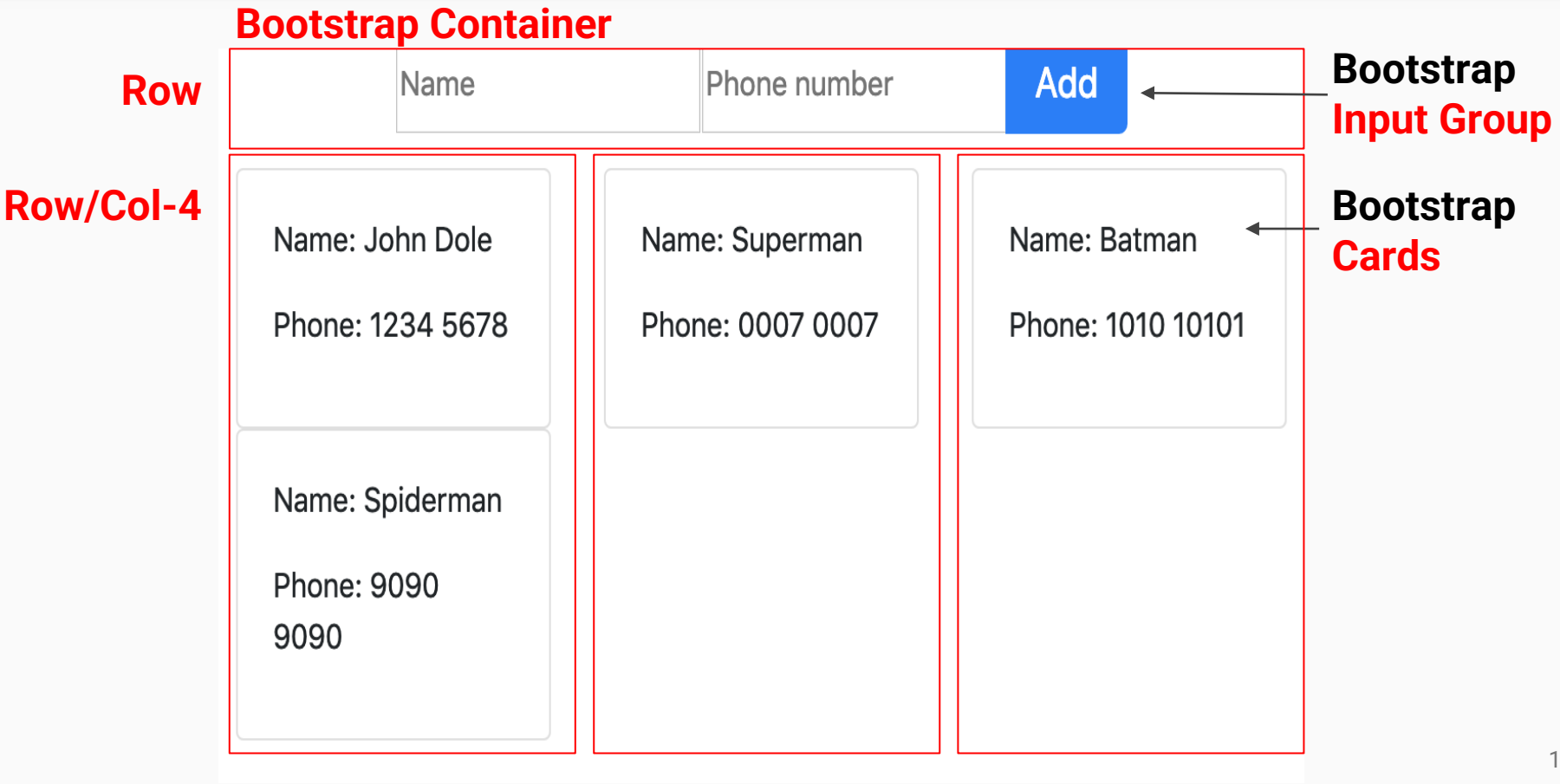
- Declarative for dynamic data
  - React enables developers to **declaratively describe their user interfaces** and **React will create the actual user interfaces** that represent dynamic data.
- Virtual DOM and reactive update
  - React uses the **Virtual DOM** to render an HTML tree virtually, and **only write the difference** between the new tree and the previous tree.

Declarative for dynamic data

- “React is declarative,” this is exactly what it means, we **describe user interfaces with React** and tell it **what we want** (not how to do it). React will take care of the “how” and **translate our declarative descriptions to actual user interfaces** in the browser.
- With React, we get to be declarative for HTML interfaces that represent **dynamic data**, not just static data.



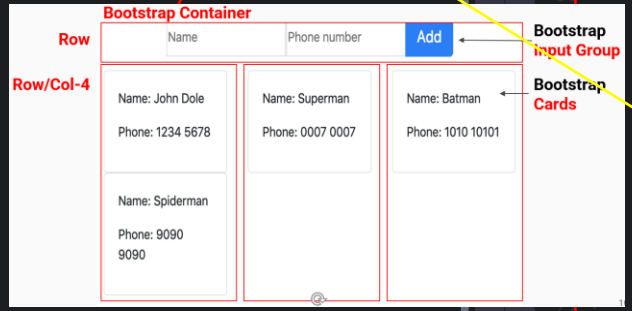
Example 1


Bootstrap Container
Row
Name
Phone number
Add
Bootstrap
Input Group
Row/Col-4
Name: John Dole
Phone: 1234 5678
Name: Superman
Phone: 0007 0007
Name: Batman
Phone: 1010 10101
Bootstrap
Cards
Name: Spiderman
Phone: 9090
9090

Example 1

HTML

```html
<div class="container-fluid">
    <div id="app"></div>
</div>
```



CSS

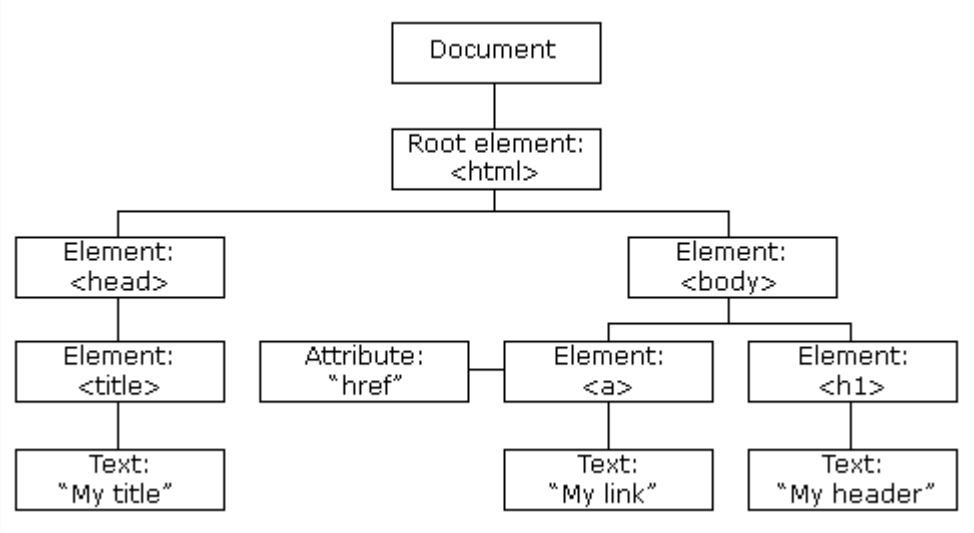
JS (Babel)

An excerpt of the class component App

```
  render() {
    return (
      <div>
        <div class="row">
          <div class="input-group input-group-lg d-flex justify-content-center">
            <input id="name" type="text" placeholder="Name" />
            <input id="phone" type="text" placeholder="Phone number" />
            <div class="input-group-append">
              <button class="btn btn-primary" type="button" onClick={this.addEntry}>
                Add
              </button>
            </div>
          </div>
        </div>
        <div class="row mt-3">{this.createPhoneBookEntries(this.state.phoneBook)}</div>
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById("app"));
```

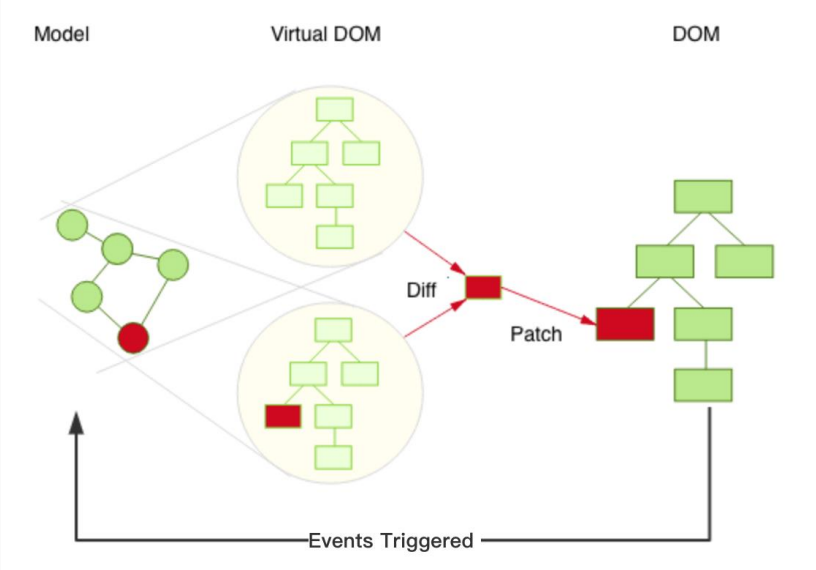
DOM

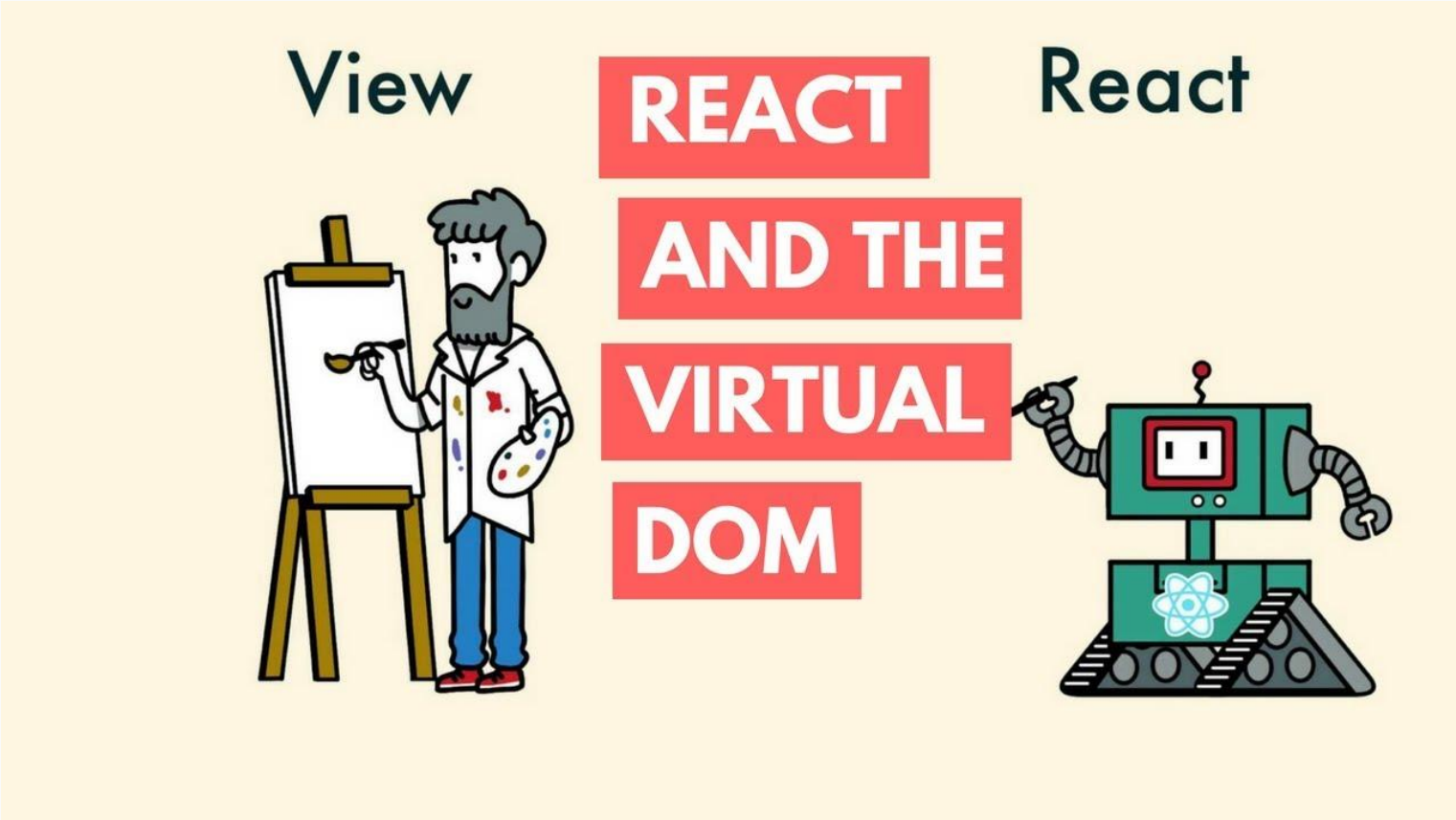
- Slow in update
- Update HTML directly
- Render **a new DOM tree** for any update
- DOM operation is expensive
- Wastage of system memory



React: Virtual DOM

- Faster in update
- Model cannot directly update HTML
- Render a virtual DOM tree for any update, which is then **compared** with the old DOM tree for the differences
- Update only the **differences** but not the whole DOM tree
- No wastage of system memory



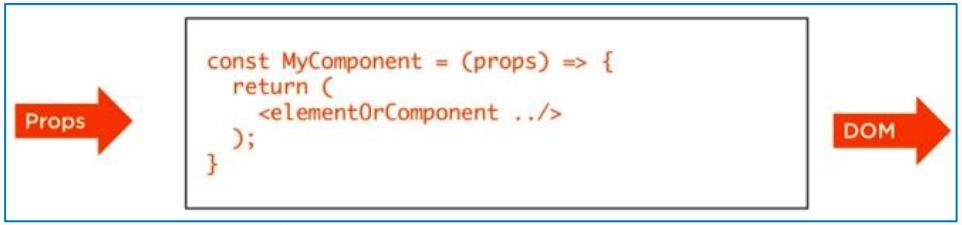


https://www.youtube.com/watch?v=BYbgopx44vo

React Components

- Components are like **classes** or **functions** in any programing languages, which can also be **reusable.**
- Their **input** are “properties” and “state”.
- Their **output** is the UI description, which is similar to HTML for browsers.

Class component



Function component

Example 2

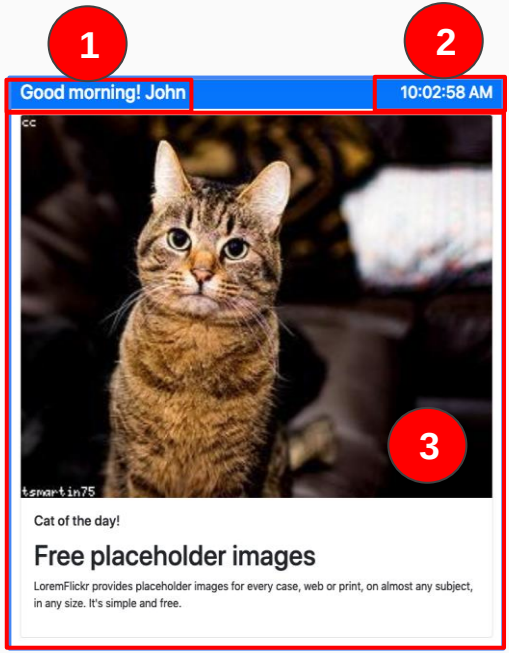
```
<div class="container-fluid">
  <div class="row bg-primary text-white">
      <div id="greetings" class="col-8 align-self-center d-flex justify-content-start"></div>
      <div id="clock" class="col-4 align-self-center d-flex justify-content-end"></div>
  </div>
  <div class="row mt-2">
      <div class="col d-flex justify-content-center">
          <div id="photo"></div>
      </div>
  </div>
</div>
```

1

2

3

```
/* Class and function implementation not shown */

ReactDOM.render(<Greetings name="John" />, document.getElementById("greetings"));
ReactDOM.render(<ClockFunction/>, document.getElementById("clock"));
ReactDOM.render(<CatOfTheDay />, document.getElementById("photo"));
```



React Components (cont.)

- React **components** are just state machines
- In React, you update a component's **state**, and then render a new UI based on this state.
  - React takes care of updating the DOM for you.
- Two types of React component
  - **Stateless**
  - **Stateful**

Stateless React Component

Stateless Component (Function) - Example

HTML

```
<div id='root' />
```

CSS

JS (Babel)

```
const HelloWorld = () => <h1>Hello, world!</h1>;

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

Hello, world!

`HelloWorld` component is just a function!

`ReactDOM.render()` creates the component, starts the framework, and injects HTML into a DOM node

Stateless Component (Class) - Example

HTML

```html
<div id='root' />
```

CSS

JS (Babel)

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

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

Hello, world!

Can use ES6 `class` and extend `React.Component`

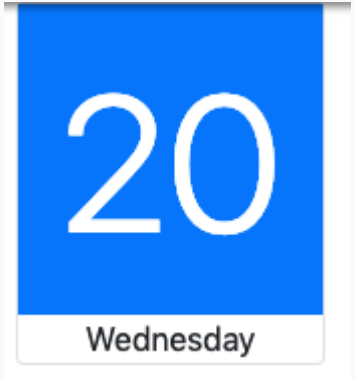
The `render` method is required. You can think of this as your template.

Date Component – Example 3

```
class SimpleCalendar extends React.Component {
  render() {
    let today = new Date();
    let date = today.getDate();
    let weekday = new Array(7);
    weekday[0] = "Sunday";
    weekday[1] = "Monday";
    weekday[2] = "Tuesday";
    weekday[3] = "Wednesday";
    weekday[4] = "Thursday";
    weekday[5] = "Friday";
    weekday[6] = "Saturday";
    return (
      <div class="card">
        <div class="card-body display-1 bg-primary text-white">
          {date}
        </div>
        <p class="card-text d-flex justify-content-center">
          {weekday[today.getDay()]}
        </p>
      </div>
    );
  }
}

ReactDOM.render(<SimpleCalendar />, document.getElementById("app"));
```

HTML

```
<div class="container-fluid">
  <div class="row mt-3">
    <div class="col-md-6 col-sm-12">
      <div id='app' class="d-flex justify-content-center"></div>
    </div>
  </div>
</div>
```



Two Components – Example 4

HTML

```html
<div class="container-fluid">
  <div class="row">
    <div class="col-md-4 col-sm-6">
      <div id="helloWorld" class="d-flex justify-content-center"></div>
    </div>
  </div>
  <div class="row">
    <div class="col-md-4 col-sm-6">
      <div id="simplecalendar" class="d-flex justify-content-center"></div>
    </div>
  </div>
</div>
```


```
ReactDOM.render(<HelloWorld />, document.getElementById("helloWorld"));
ReactDOM.render(<SimpleCalendar />, document.getElementById("simplecalendar"));
```

To add an extra UI component, we can simply add a container, i.e. `<div>` on the UI.

Then construct a class/function component for generating an output (UI description).

Using `prop`

Using the `props` parameter

HTML

```
<div id='root' />
```

CSS

JS (Babel)

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

ReactDOM.render(
  <HelloWorld name="JavaScript" />,
  document.getElementById('root')
);
```

Hello, JavaScript!

`props` can be used as an input parameter.

You can pass read-only **properties** to a React component via its *attributes*.

You can access this data with the `props` parameter inside of a JavaScript Expression `{ }`

`props` (array) – Example 5

HTML

```
<div id='root' />
```

CSS

JS (Babel)

```
class ShoppingList extends React.Component {
  render() {
    return (
      <ul class="list-group">{CreateShoppingItems(this.props.toBuyItems)}</ul>
    );
  }
}

const CreateShoppingItems = toBuyItems => {
  return toBuyItems.map(ShoppingItem);
};

const ShoppingItem = item => {
  return <li class="list-group-item">{item}</li>;
};

ReactDOM.render(
  <ShoppingList toBuyItems={["banana", "juice", "coffee"]} />,
  document.getElementById("root")
);
```

banana

juice

coffee

Resulting HTML

```
<div id='root' />
  <ul class="list-group">
    <li class="list-group-item">banana</li>
    <li class="list-group-item">juice</li>
    <li class="list-group-item">coffee</li>
  </ul>
```

Comparing React with native DOM

JS (Babel)

React

```
class ShoppingList extends React.Component {
  render() {
    return (
      <ul class="list-group">{CreateShoppingItems(this.props.toBuyItems)}</ul>
    );
  }
}

const CreateShoppingItems = toBuyItems => {
  return toBuyItems.map(ShoppingItem);
};

const ShoppingItem = item => {
  return <li class="list-group-item">{item}</li>;
};

ReactDOM.render(
  <ShoppingList toBuyItems={["banana", "juice", "coffee"]} />,
  document.getElementById("root")
);
```

- banana
- juice
- coffee

JS (Babel)

DOM

```
var toBuyItems = ["banana", "juice", "coffee"];
var root = document.getElementById("root");

var unorderedList = document.createElement("ul");
let att = document.createAttribute("class");
att.value = "list-group";
unorderedList.setAttributeNode(att);
root.appendChild(unorderedList);

for (i in toBuyItems) {
  var listNode = document.createElement("li");
  let att = document.createAttribute("class");
  att.value = "list-group-item";
  listNode.setAttributeNode(att);
  var itemNode = document.createTextNode(toBuyItems[i]);
  listNode.appendChild(itemNode);
  unorderedList.appendChild(listNode);
}
```

The above implementations produce the same output. Which one is better?

Stateful React Component

Let's Look at State

HTML

```html
<div id='root' />
```

JS (Babel)

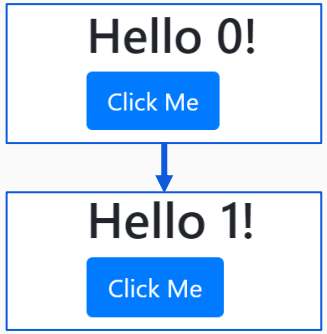
```
class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  handleClick = () => {
    this.setState({
      count: this.state.count + 1
    });
  };

  render() {
    return (
      <div class="container">
        <p class="h1">Hello {this.state.count}!</p>
        <button type="button" class="btn btn-primary btn-lg"
            onClick={this.handleClick}>Click Me</button>
      </div>
    );
  }
}

ReactDOM.render(<HelloWorld />, document.getElementById("root"));
```

Provide default `state` by assigning object from `constructor`

Update component state via `this.setState`

Access component state via `this.state`

Example 6



State: Example 7

HTML

```html
<div id='root' />
```

JS (Babel)

```js
class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      time: new Date().toLocaleTimeString()
    };
  }

  handleClick = () => {
    this.setState({
      time: new Date().toLocaleTimeString()
    });
  }

  render() {
    return(
      <div class="container">
        <button type="button" class="btn btn-primary btn-lg"
          onClick={this.handleClick}>
        Refresh  <span class="badge badge-light">{this.state.time}</span>
        </button>
      </div>
    );
  }
}

ReactDOM.render(
  <HelloWorld name="JavaScript" />,
  document.getElementById('root')
);
```

Provide default `time` in `constructor`

Update `time` in event handler

Access to `time`

Refresh 11:31:04 PM

Multiple States: Example 8

HTML

```html
<div id='root' />
```

CSS

JS (Babel)

```javascript
class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: this.props.name,
      textColor: this.props.textColor
    }
  }

  render() {
    return(
      <h1 style={{color:this.state.textColor}}>Hello, {this.state.name}!</h1>
    );
  }
}

ReactDOM.render(
  <HelloWorld name="JavaScript" textColor="red" />,
  document.getElementById('root')
);
```

There are two states.
You can specify states as many as you want

Hello, JavaScript!

Adding Lifecycle Methods to a Class

In applications with many components, it's very important to **free up resources** taken by the components **when they are destroyed**.

We want to **set up** a timer whenever the Clock is **rendered to the DOM for the first time**. This is called "**mounting**" in React.

We also want to **clear** that timer whenever **the DOM produced by the Clock is removed**. This is called "**unmounting**" in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts.

Hello, world!

It is 12:15:50 AM.

Adding Lifecycle Methods to a Class

`componentWillMount()` – Fired once, **before initial rendering** occurs. Good place to wire-up message listeners. `this.setState` doesn't work here.

`componentDidMount()` – Fired once, **after initial rendering** occurs. Can use `ReactDOM.findDOMNode`.

`componentDidUpdate()` - Fired after the component's updates are made to the DOM.

`componentWillReceiveProps()` – Fired when a component is receiving new props. You might want to this.setState depending on the props.

`shouldComponentUpdate()` - Fired before rendering when new props or state are received. return false if you know an update isn't needed.

`componentWillUnmount()` – Fired immediately before a component is unmounted from the DOM. Good place to remove message listeners or general clean up.

https://reactjs.org/docs/react-component.html

Lifecycle: Example 9

HTML

```html
<div id="root">
</div>
```

JS (Babel)

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

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

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

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

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

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

Hello, world!

It is 12:15:50 AM.

The `componentDidMount()` method runs after the component output has been rendered to the DOM. This is a good place to set up a timer.

We will tear down the timer in the `componentWillUnmount()` lifecycle method.

State: Arrays - The **Spread** (. . .) Operator – Example 10

HTML

```
<div id='app' />
<div id="x" />
```

CSS

JS (Babel)

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      x: []
    }
  }

  handleClick = (event) => {
    let newArray = [...this.state.x, Math.random().toFixed(2)];
    this.setState({
      x: newArray
    })
  }

  render() {
    return (
      <div>
        <button type="button" class="btn-primary" onClick={this.handleClick}>
          Click Me!
        </button>
        <div>
          {this.state.x.map(n=><li>{n}</li>)}
        </div>
      </div>
    );
  }
}

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

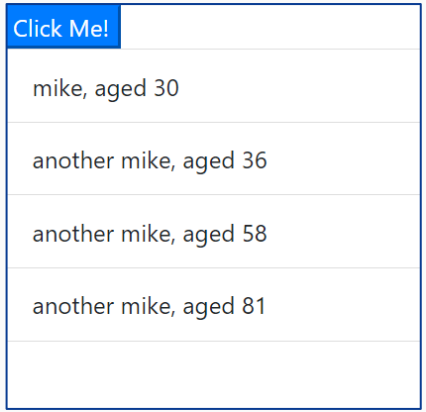
Click Me!

- 0.08
- 0.87
- 0.07
- 0.58
- 0.92
- 0.53

State: JSON Objects – Example 11

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      contact: [{name: "mike", age: 30}]  // array of json objects
    }
  }

  handleClick = (event) => {
    let newContact = {};  // this is a json object
    newContact['name'] = 'another mike';
    newContact['age'] = Math.floor(Math.random() * 100);
    let newArray = [...this.state.contact, newContact];
    this.setState({
      contact: newArray
    })
  }
```

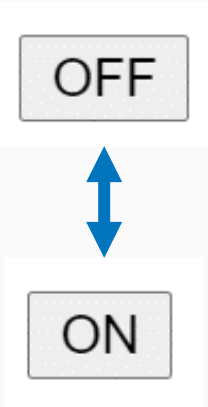
JSON (**JavaScript Object Notation**) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of **attribute–value pairs** and **arrays**.



Handling Events – Example 12

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

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



For example, the HTML:

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

is slightly different in React:

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

Passing Parameters to Event Handlers – Example 13

HTML

```
<div id='app' />
```

CSS

JS (Babel)

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

  handleClick = (message) => {
    alert(message);
  }

  render() {
    let message = "Good day!";
    return (
      <div>
        <button type="button" class="btn-primary" onClick={()=>this.handleClick(message)}>
          Click Me!
        </button>
      </div>
    );
  }
}

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

Click Me!

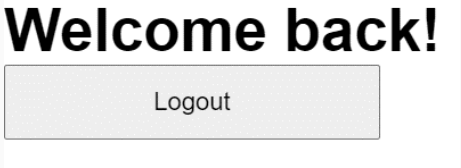
An embedded page at cdpn.io says

Good day!

OK

Conditional Rendering – Example 14

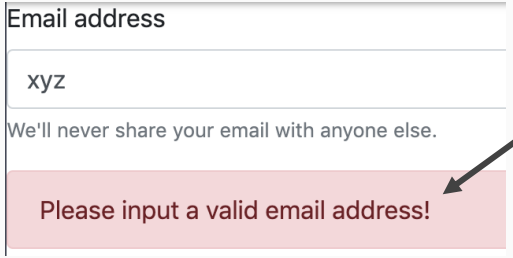
```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  let button;

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

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



The appearance of the message and the button are relied on the state `isLoggedIn`.

Conditional Rendering - `hidden` property – Example 15

```
render() {
  return (
    <form>
      <div class="form-group">
        <label for="email">Email address</label>
        <input type="email" class="form-control" id="email" onBlur={this.verify }placeholder="Enter email">
        </input>
        <small id="emailHelp" class="form-text text-muted">
          We'll never share your email with anyone else.</small>
      </div>
      <div class="alert alert-danger" role="alert" hidden={this.state.valid}>
        Please input a valid email address!
      </div>
    </form>
  );
}
```



Reference

https://elijahmanor.com/talks/react-to-the-future/

https://jaxenter.com/introduction-react-147054.html

https://reactjs.org/docs/state-and-lifecycle.html