

Building User Interfaces

# React 1

## An Introduction

Professor Bilge Mutlu

# Disclaimer

As with JS, this is not a comprehensive introduction to React, so below are links to great additional resources:

>> [ReactJS.org](https://reactjs.org)

>> [W3 Schools](https://www.w3schools.com/html/html5_react.asp)

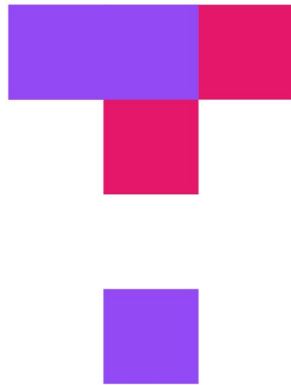
>> [Build with React](https://reactjs.org/docs/build-with-react.html)

>> [Tania Rascia's React Overview and Walkthrough](https://www.taniarascia.com/react-overview-and-walkthrough)

# What we will learn today?

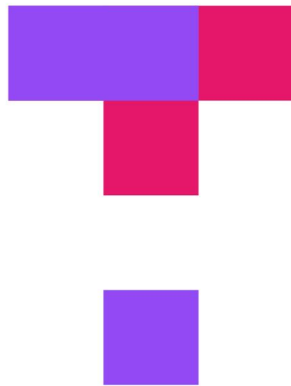
- >> History and overview of React
- >> Overview of building blocks
- >> Setting up a React project

# TopHat Attendance



**TOP HAT**

# TopHat Questions



**TOP HAT**

# Why should we use React?

# What is React?<sup>1</sup>

**Definition:** Also called ReactJS, React is a JS library for building user interfaces.

- >> Developed by Facebook, dating back to 2010.
- >> Started as an internal development tool, then open-sourced in 2013.

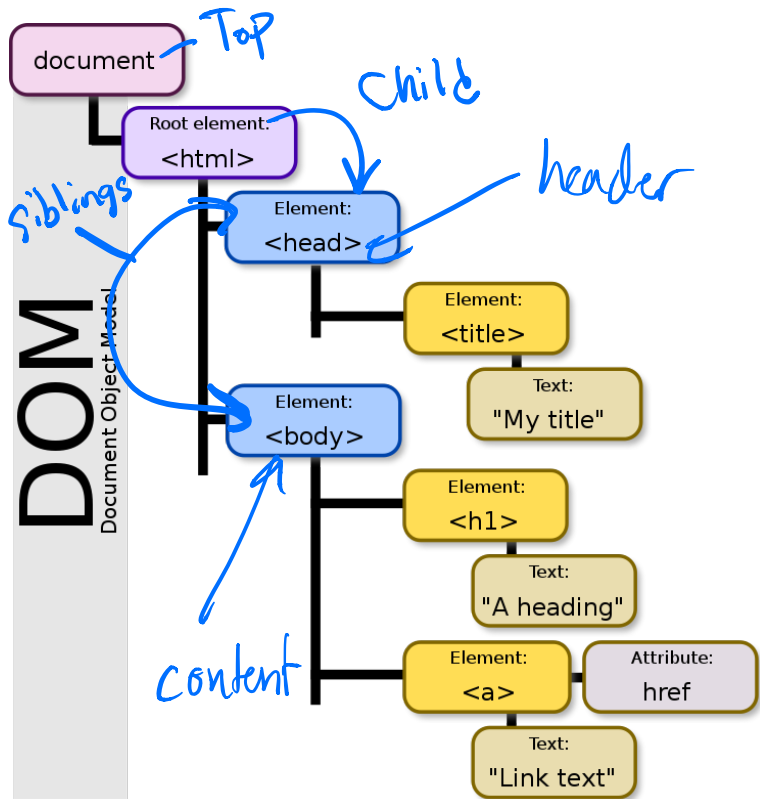
---

<sup>1</sup>[More on the history of React](#)

# Refresher: Document Object Model

**Definition:** Document Object Model (DOM) translates an HTML or XML document into a tree structure where each node represents an object on the page.

Source<sup>2</sup>



<sup>2</sup>[Wikipedia: DOM](#)



# Refresher: DOM Programming Interface

- » **Objects:** HTML elements, such as a paragraph of text.
- » **Property:** Value that we can get or set, such as the id of an element.
- » **Method:** An action we can take, such as adding or deleting an HTML element.

For JS to interact with user-facing elements, we first need to access them...

# Refresher: Accessing HTML elements

Most common way of accessing content is `getElementById()`.

```
<p id="userName"></p>
```

*unique ids*

```
<script>
```

```
    document.getElementById("userName").innerHTML = "Andy Schoen";
```

```
</script>
```

*singular response*

We can also find elements using tag name, class name, CSS selectors, and HTML object collections.

# Refresher: Manipulating HTML elements

Changing content:

```
document.getElementById("userName").innerHTML = "Andy Schoen";
```

Changing attributes:

```
document.getElementById("userImage").src = "Headshot.png";
```

```
document.getElementById("userName").style.color = "red";
```

# Refresher: DOM Events

DOM provides access to HTML events, such as `onclick`, `onload`, `onunload`, `onchange`, `onmouseover`, `onmouseout`, `onmousedown`, `onmouseup`, `formaction`.

We can register functions to events using *inline event handlers*, *DOM on-event handlers*, and *using event listeners*.

# What's wrong with this approach?<sup>3</sup>

- » Working with HTML DOM is slow
- » The DOM for *single-page applications (SPAs)* can be huge
  - » Needle in a haystack
- » Interactive applications require a large number of and frequent updates on DOM elements
  - » Inefficient updating

tons of ids, etc!

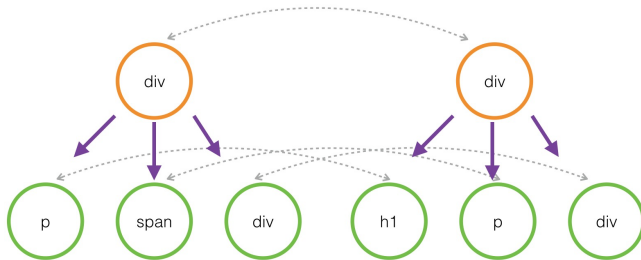
---

<sup>3</sup>React Kung Fu

# Solution: The *Virtual DOM*<sup>4</sup>

memory-based representation

**Definition:** The virtual DOM is a *virtual* representation of the user-facing elements that are kept in memory and synced with the real DOM when DOM elements are updated.



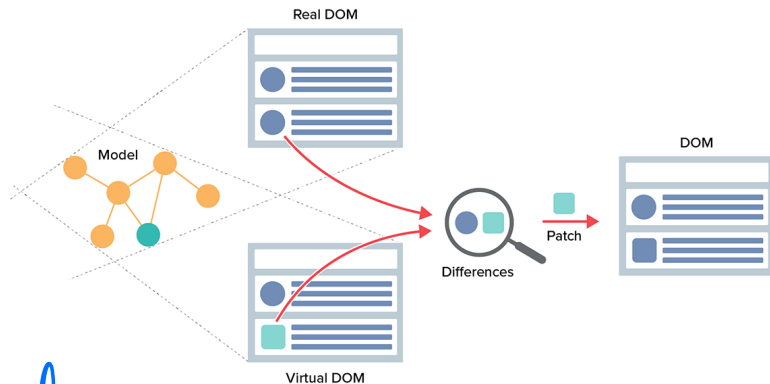
---

<sup>4</sup> [Image source](#)

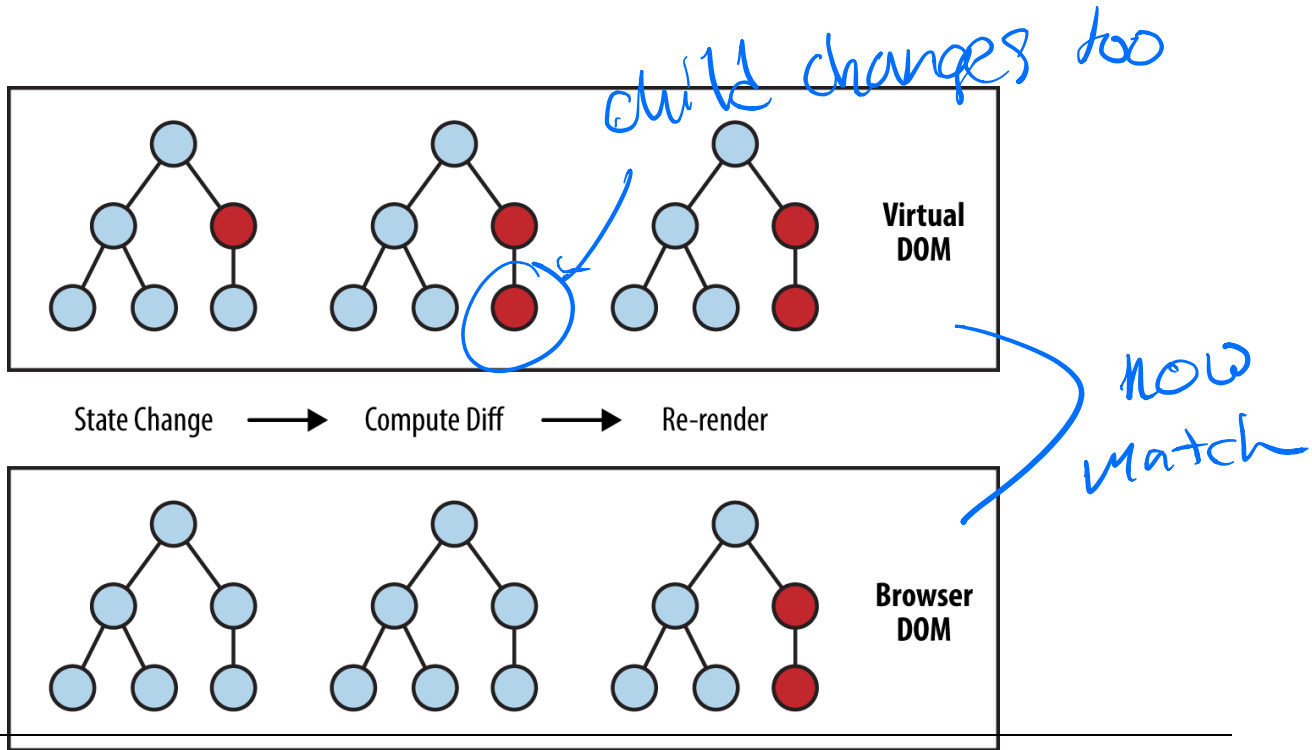
# More on the solution: *Reconciliation*<sup>5</sup>

**Definition:** *Reconciliation* is the process of *diffing* and syncing the virtual and real DOM to render changes for the user.

find differences  
and patch as  
needed



<sup>5</sup>[Image source](#)





# What are the benefits?

- » Incredibly fast, as only what is updated in the Virtual DOM is updated in the real DOM
- » Abstracts away interactions with DOM: *declarative programming*

you don't have to manually change the DOM. You just specify what it should be

# Detour: Declarative vs. imperative programming

**Definition:** *Imperative programming* expresses how the computation must flow.

- Programming how to get the outcome we want.

**Definition:** *Declarative programming* expresses the logic of a computation without describing its flow.

- Programming what we want the outcome to be.

virtual DOM

## Imperative example:<sup>7</sup>

```
var ages = [32, 45, 16, 67];
function checkEligibility() {
  for (i = 0; i < ages.length; i++) {
    if (ages[i] < 18 || ages[i] > 65) {
      console.log('Every one is not eligible');
      return false; } }
  console.log('Every one is eligible');
  return true; }
```

*every* checks that all entries are true

## Declarative example:

```
var ages = [32, 45, 16, 67];
function checkEligibility() {
  console.log(ages.every(age => age >= 18 && age <= 65)
    ? 'Every one is eligible' : 'Every one is not eligible'); }
```

*anonymous function*

*only specifies what it should log*

---

<sup>7</sup>See working example in [CodePen](#)

# Let's get back to React

- >> React assumes declarative programming
  - >> We only care about the outcome we want to see
- >> The ReactDOM library takes care of reconciliation and updating user-facing content under the hood

# wtf IS React's VIRTUAL DOM?

ONE of the reasons REACT IS so darn

# FAST\*

at updating changes TO OUR APP IS that it uses a VIRTUAL DOM

2

1

The React DOM

which is like a floating SECOND LAYER on top of the real DOM.

The real DOM

<sup>8</sup> Source: [www.illustrated.dev](http://www.illustrated.dev)

ALL the CODE we WRITE IN <React />

all THE COMPONENTS we CREATE

all THE PROPS we PASS IN

all THE STATES we UPDATE

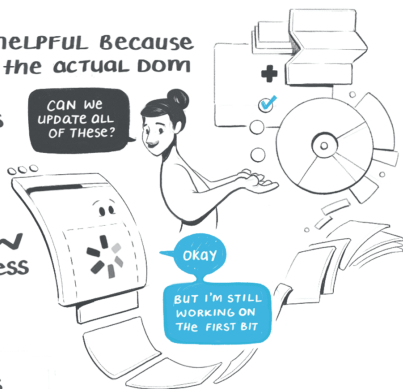
only interact WITH THIS VIRTUAL DOM

WHICH THEN TALKS TO THE REAL DOM FOR US

HEY BUDDY

HOW'S MONDAY GOING?

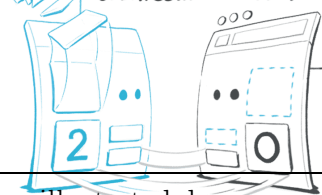
This is helpful because  
 passing the actual DOM  
 lots of changes  
 to our app's state  
 is a **SLOW**  
 process



whereas,  
 when we pass

THE  
**VIRTUAL DOM**

A PILE OF  
 UPDATES...



IT first compares its  
 own version  
 of the DOM to  
 THE REAL DOM  
 and notes all  
 the changes

AND THEN  
 IT RUNS A  
**"diffing algorithm"**



to calculate  
 how to relay  
 those changes  
 in the most  
**EFFICIENT**  
 way possible...

WITH THE LEAST  
 NUMBER OF  
 REQUESTS

OUR  
 1,000  
 UPDATES



JUST A  
 LITTLE  
 CHANGE...



(this process of  
 syncing up your app's  
 state with the real DOM  
 is called **reconciliation**)

Handy dev  
 buzzword

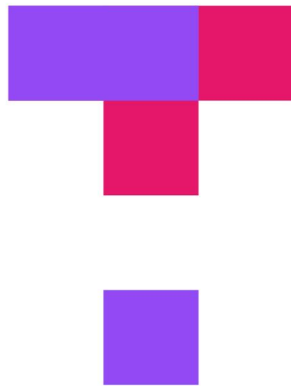
<sup>9</sup> Source: [illustrated.dev](https://illustrated.dev)

\* I know React's Speed is a hotly debated point.  
 TO CLARIFY: THE virtual DOM doesn't speed up  
 our initial render. It's only quicker at rendering  
 changes to state, compared to the bog-standard,  
 unoptimised DOM. It's also not the *absolute fastest*  
 solution to the issue of slow DOMs.

# Ok, that's why we should use React!

The name comes from *reacting* to events.

# TopHat Quiz



**TOP HAT**



# Building Blocks

# Core Components

React elements and components are the two fundamental building blocks React uses to represent and change all user-facing context and events.

# React Elements

**Definition:** A React element is a light, stateless, immutable, virtual representation of a DOM Element.

React elements are JS objects, thus browser-independent, until they are rendered. Once they are rendered, they become DOM elements.<sup>10</sup>

```
var root = React.createElement('div');  
ReactDOM.render(root, document.getElementById('example'));
```

---

<sup>10</sup>List of ReactElements

# React Elements, *Continued*

*If React elements are immutable, how do we update the page?*

Think of an element as a single frame of a movie, which represents the page at a certain point in time. To update the page, we create and render a new element.

# React.Component

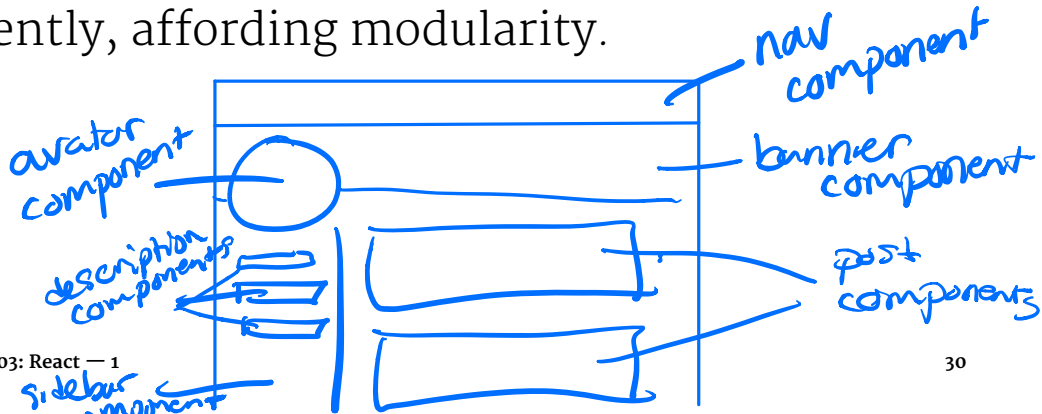
**Definition:** A React component is a function or class that accepts an input and returns a React element.

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

# React.Component, Continued

Components work like JS functions; they accept props and return React elements that correspond to what will be rendered in the DOM.

Each component is encapsulated (one component per file) and can operate independently, affording modularity.



# render()

**Definition:** `render()` returns a React element to be displayed on the page. Think of what you would like to see on the screen to determine what should go in `render()`.

There are two `render()` methods in React: `ReactDOM.render()` and `Component.render()`.

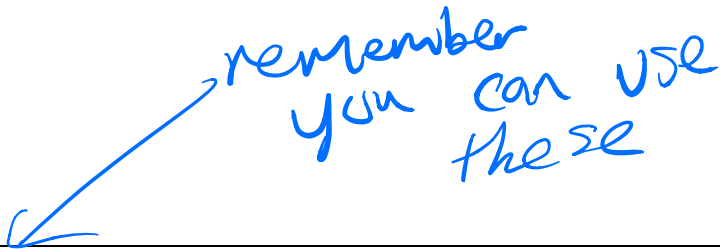
# ReactDOM.render()<sup>11</sup>

ReactDOM.render(element, container) mounts the declared element as a child to the specified container in the DOM.

```
const element = <h1>Welcome to React</h1>;
```

```
ReactDOM.render(element, document.getElementById('root'));
```

*remember  
you can use  
these*



---

<sup>11</sup>[See in CodePen](#)



# Component.render()<sup>12</sup>

Component.render() creates the virtual DOM representation of the contents of the React component. And then, we call ReactDOM.render() to mount the elements on the DOM.

```
class App extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1 className="App-title">Welcome to React</h1>  
      </div>  
    );  
  }  
}  
ReactDOM.render(<App />, document.getElementById("root"));
```

specify  
structure  
here

---

<sup>12</sup>[See in CodePen](#)

# props and states

**Definition:** `props`, or properties, are the arbitrary input provided into React components that utilize them to render content.

Components should never modify `props`; they are read-only and immutable.

**Definition:** `state` is similar to `props`, but they are fully private and controlled by the component.

`state` is what helps us keep track of changes in data.

# Usage: props<sup>13</sup>

Props are passed into functions as arguments, e.g.:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
const element = <Welcome name="Andy" />;  
ReactDOM.render(element, document.getElementById('root'));
```

The diagram consists of blue arrows. One arrow points from the `name="Andy"` attribute in the JSX element to the `name` prop in the `<Welcome />` call. Another arrow points from the `name` prop in the call to the `props` parameter in the `function Welcome(props)` definition. A third arrow points from the `props` parameter to the `{props.name}` interpolation in the `return` statement.

Reference using `this.props` within the component.

---

<sup>13</sup>[See in CodePen](#)

# Usage: state<sup>14</sup>

States are defined and manipulated within components, e.g.:

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

*must be modified like this!*

The state can be referenced using `this.state` and changed using `this.setState()`. More on this later...

<sup>14</sup>[See in CodePen](#)

# JSX

JavaScript + XML

**Definition:** A preprocessor step that adds XML syntax to JavaScript. JSX declarations produce React elements.

```
<div className="red">Text</div>;
```

... is compiled into ...

```
React.createElement("div", { className: "red" }, "Text");
```

Babel<sup>15</sup> is the preprocessor that compiles JSX into JS.

---

<sup>15</sup>[See example in Babel REPL](#)

# JSX, Continued

- >> JSX is not required, but makes React programming extremely effective
- >> Any JS expression can be included in JSX

## Detour: Naming Conventions<sup>16</sup>

**Definition:** camelCase involves writing phrases such that each word begins capitalized with no spaces/punctuation.

**Definition:** hyphen-case (aka *kebab-case*) involves writing phrases in lower case and using a hyphen as a separator.



---

<sup>16</sup>[Wikipedia](#)

# Detour: Naming Conventions, *Continued*

**Definition:** PascalCase capitalizes all words with no spaces/punctuation.

- >> ReactDOM and JSX use camelCase
- >> React components use PascalCase



# Setting up a React project

# Setting up a React project

## In the development environment

What you will need: *terminal, coding environment, Node.js*

```
npm install -g create-react-app
```

```
npx create-react-app <your-app-name>
```

```
cd <your-app-name>
```

```
npm start
```

*automatically generate a simple starting point*

*Change Directory to the app folder*

*Start the app*

# Setting up a React project

## In a sandbox

Simple React application:<sup>17</sup>

1. set sandbox settings to Babel preprocessor
2. import react and react-dom CDNs

React project:

- Create a project using a template / upload your project

---

<sup>17</sup> [See example in CodePen](#)

# What did we learn today?

- >> History and overview of React
- >> Overview of building blocks
- >> Setting up a React project