React Fundamentals

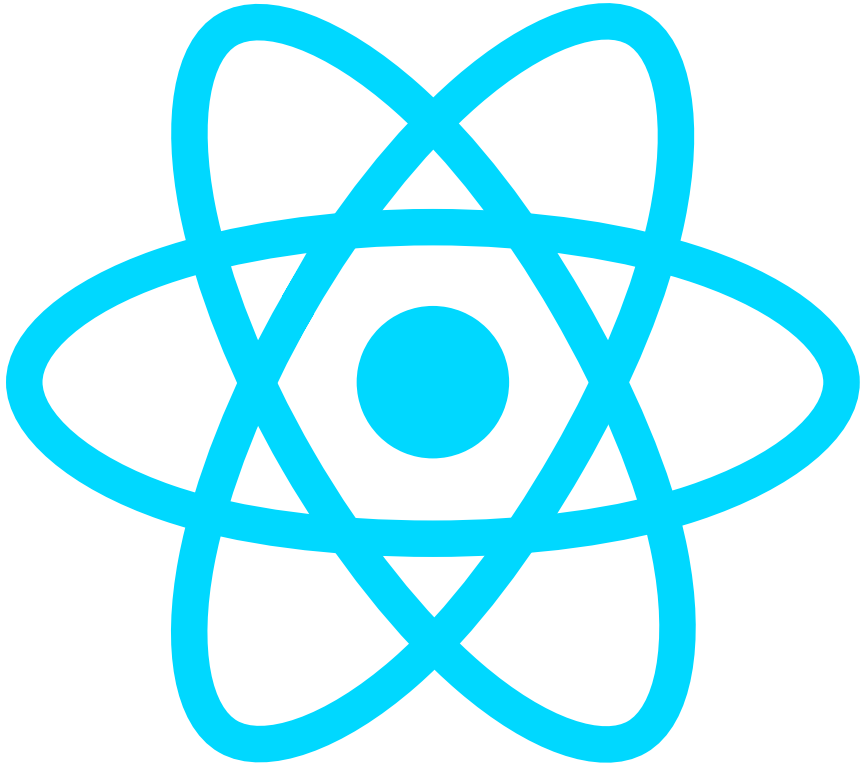
UI Frameworks

Peter Munro

React Fundamentals

React Fundamentals

React Fundamentals



What's React?

- A library for building component-based UIs in JavaScript
- Not an MVC framework
    - Facebook sometimes calls it "the V in MVC"

Why React?

- Conceptually simple
  - Ability to reason about our code
- Component-based
  - Components are composable and testable
- Manages the DOM for you
  - DOM manipulation is expensive
  - React only updates what it has to

History

- Facebook Ads Org
  - Client-side MVC using two-way data binding led to cascading updates which didn't scale
- Yet Facebook buddy list simply re-rendered entire list on online/offline events!
  - Jordan Walke wrote prototype
- Instagram (mainly Pete Hunt) refactored to standalone library
  - React in production use in both
- May 2013: React open-sourced
  - Feedback: "Huge step backwards" :-)
- March 2018: React 16.3 released

Who uses React?

Instagram
Fast beautiful photo sharing

facebook

KHAN
ACADEMY

NETFLIX

codecademy

BBC

...and many more

Components and Rendering

React Fundamentals

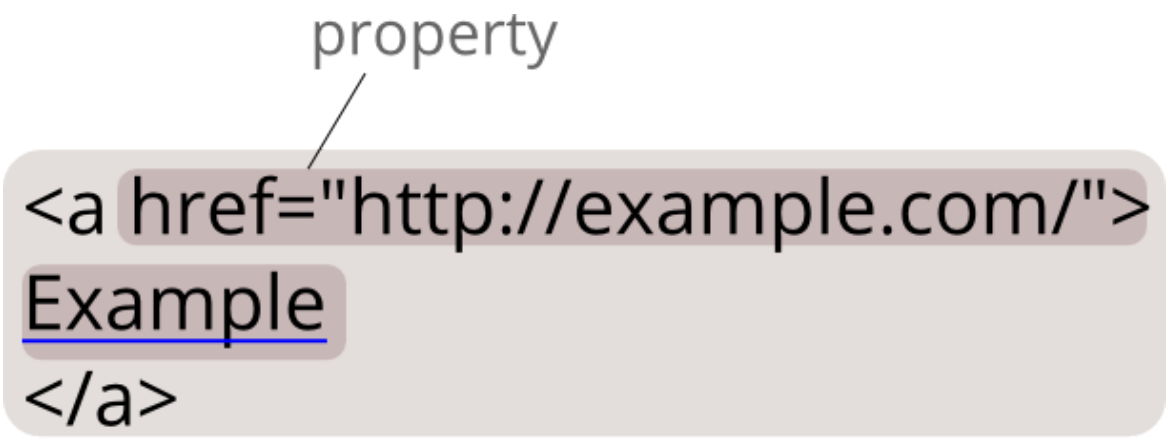
Components

- Components: great for composing UIs
- HTML provides us with simple components
  - h1, p, input, audio, ...
- For apps, why can't we have developer-defined 'complex' components?
  - ChatPanel, StarRating, StockGraph?

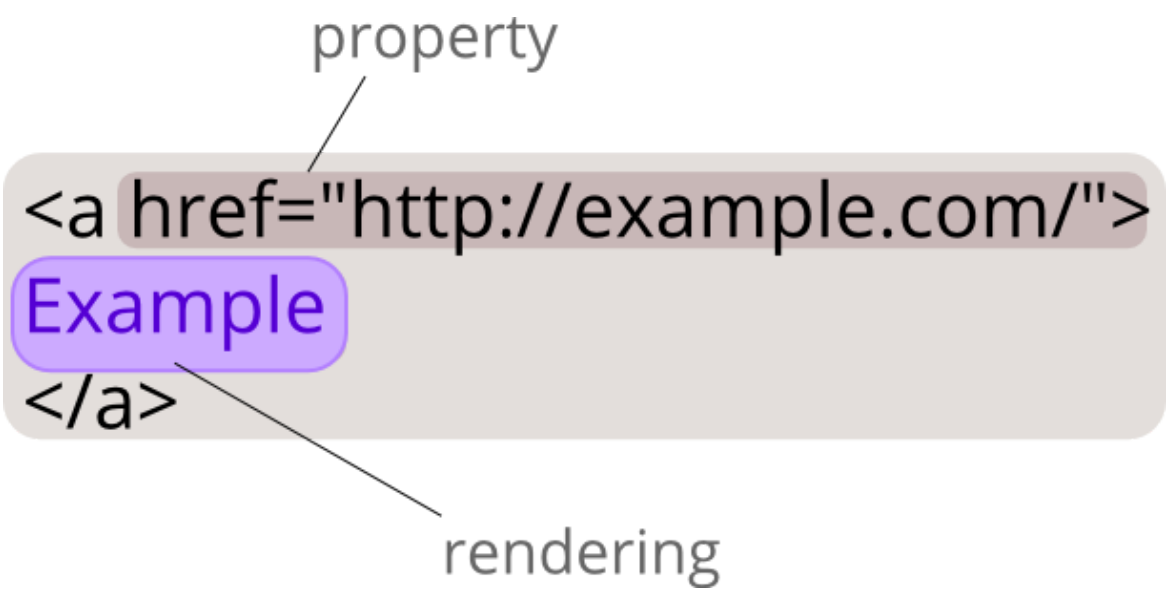
Components

```
<a href="http://example.com/">
Example
</a>
```

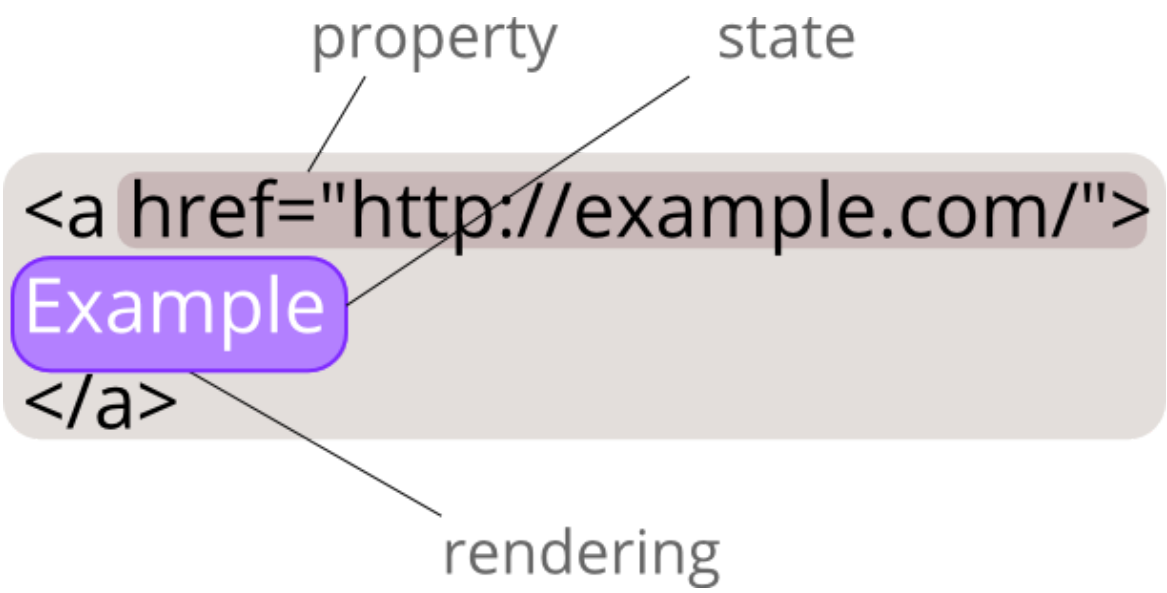
Components



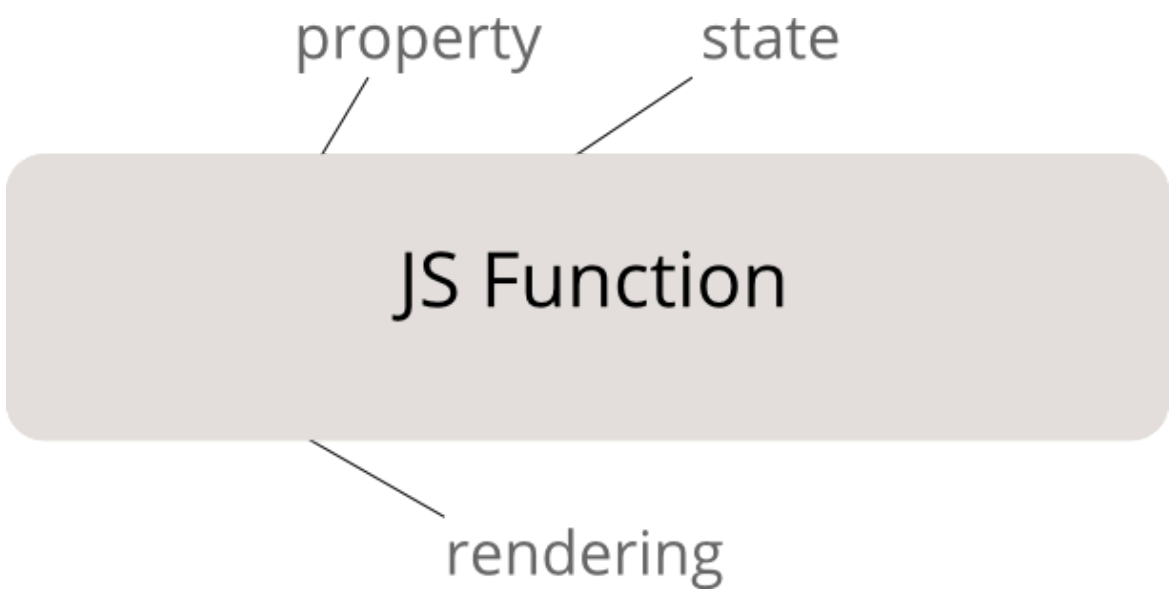
Components



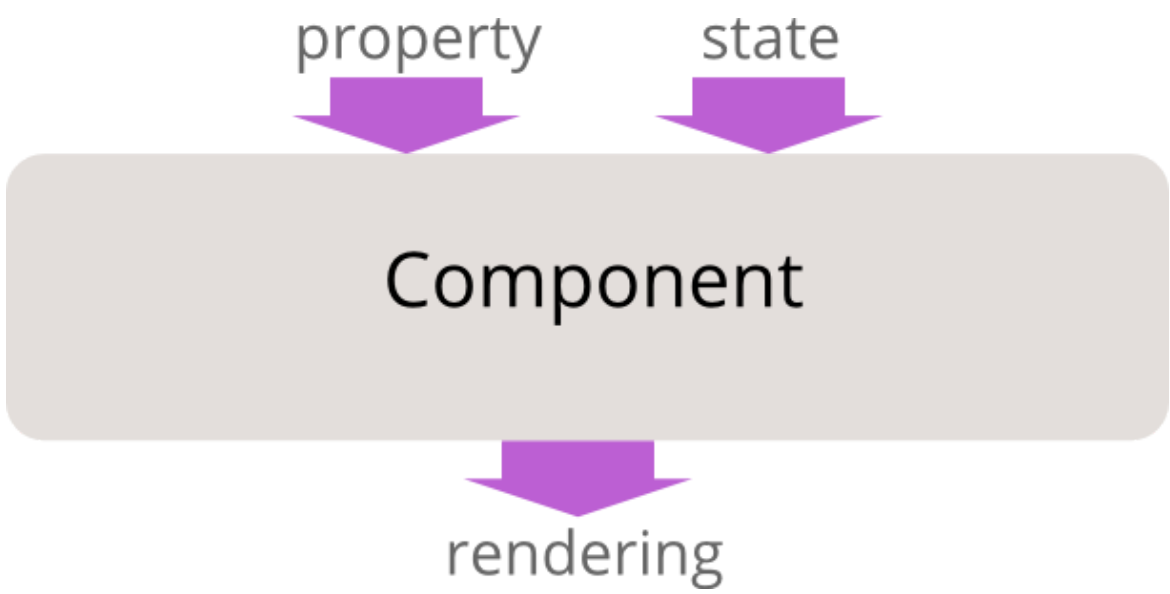
Components



Components



Components



- Component:
  - takes properties and state, and returns an HTML rendering

Simple Components

To create simple components:

JavaScript

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

This would create a JS object corresponding to:

HTML

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

Consequences

Rendering components using JS functions means:

- with the same inputs, they always produce the same outputs
- we can test them, improving code confidence in our code
- we can start using functional programming to build UIs

DOM Manipulation is Slow

Compared to plain JS objects, the DOM is slow:

JavaScript

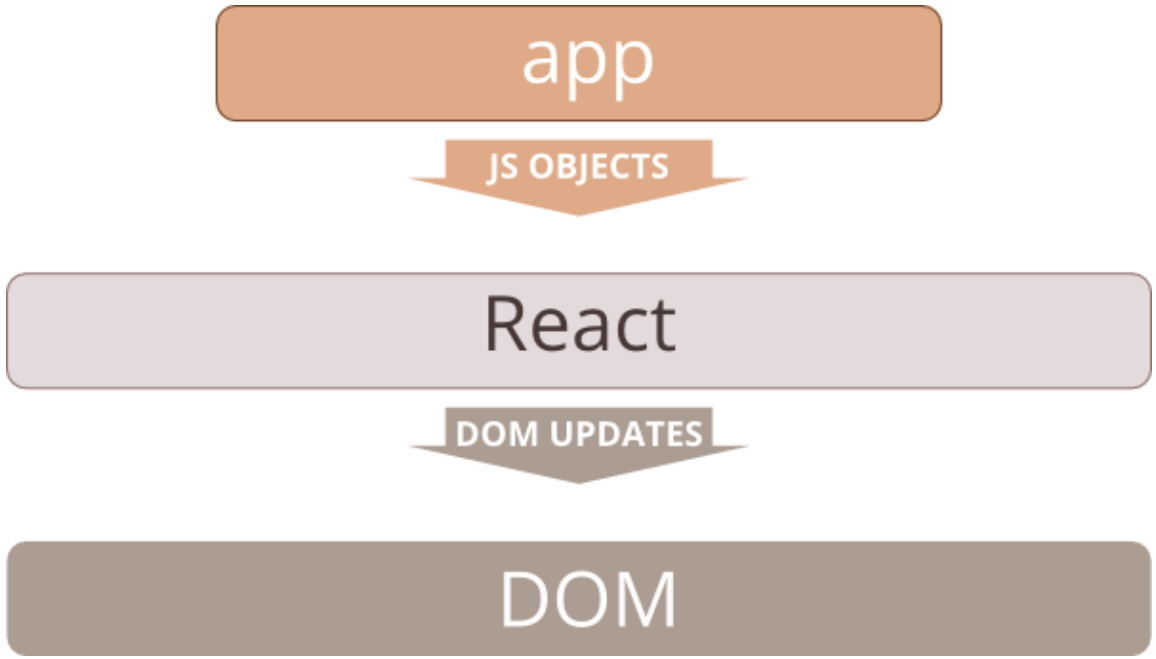
```
element.innerHTML = x;
```

- parse x as HTML
- ask browser extensions for permission
- destroy existing child nodes of element
- create child nodes
- recompute styles which are defined in terms of parent-child relationships
- recompute physical dimensions of page elements
- notify browser extensions of the change
- update Javascript variables which are handles to real DOM nodes

(From http://bit.ly/1EUdH6q)

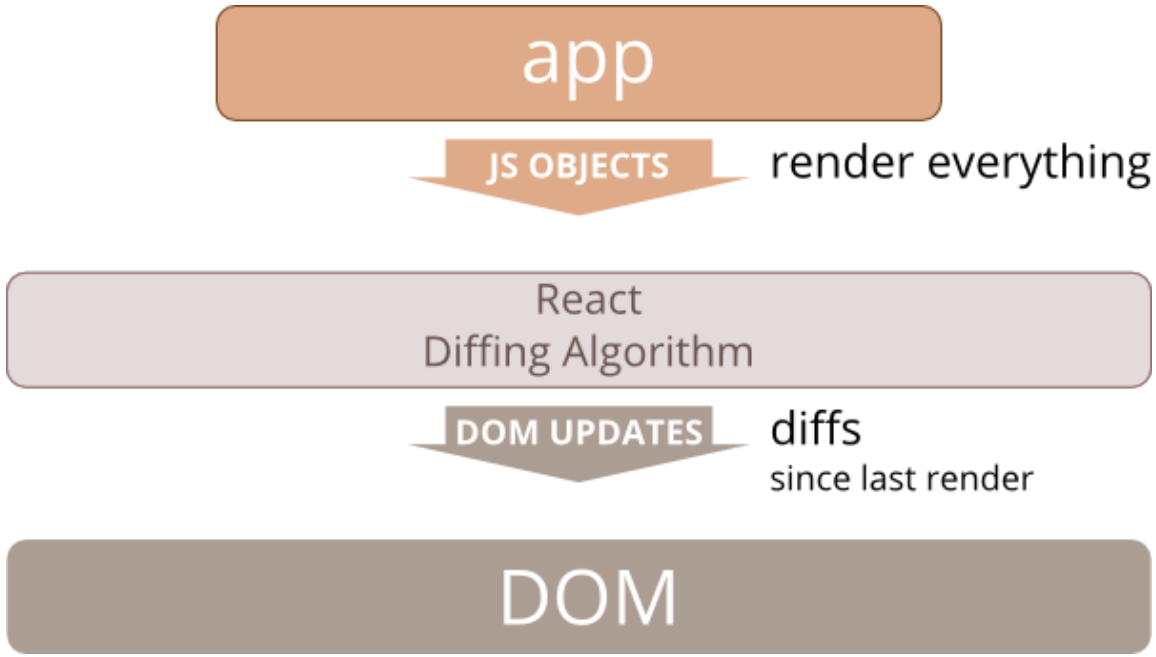
Rendering Lightweight Objects

- App describes how it wants UI to look
- Rendering creates lightweight JS objects, not DOM nodes - a **vDOM** or "Virtual DOM"
- App keeps no state in the DOM



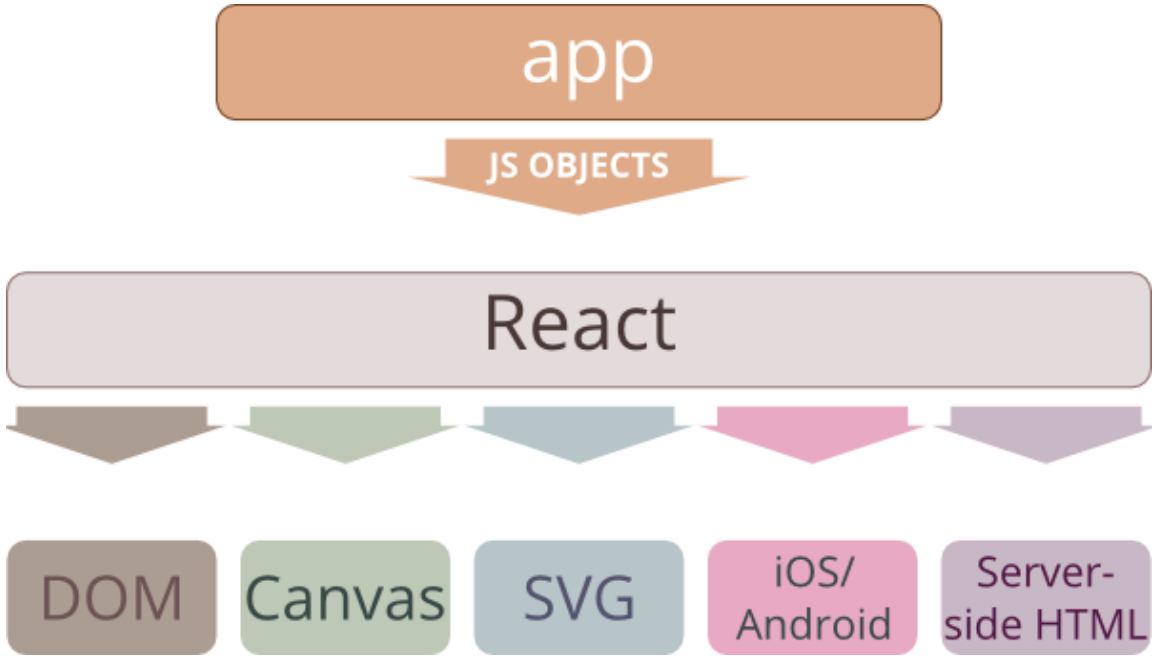
Re-Render Everything Every Time

- React imposes an API layer above the DOM
- App renders everything on every state change
- React updates just what's changed



Alternate Render Outputs

- React is pluggable
- Alternate renderers can be developed



Resources

The official React site:

- https://reactjs.org/

Reddit

- https://www.reddit.com/r/reactjs/

Getting Started: `create-react-app`

React Fundamentals

What's `create-react-app`?

`create-react-app` is a tool to help you get started quickly

- You don't need to configure Babel, Webpack etc
- It offers:
  - pre-configured webpack and webpack-dev-server
  - pre-configured Babel so you can use ES6, JSX and extensions (object spread and class properties)
  - vendor prefixing using Autoprefixer
  - static code checking using ESlint
  - test setup using Jest

`create-react-app` Commands

`create-react-app` provides these commands:

COMMAND	DESCRIPTION
`npm start`	Starts your development server
`npm run build`	Bundles the app into static files for production
`npm test`	Starts the test runner
`npm run eject`	'Eject's from `create-react-app` to give you control over your configuration. However, once you do this, you can't go back.

Defining a Component

React Fundamentals

Two Ways to Define Components

To define components, you can create either:

- an *SFC* (Stateless Functional Component); or
- a class

Creating a Stateless Functional Component (SFC)

To define a component using the function style:

JSX

```jsx
import React from 'react';
import './styles/Footer.css';

const Footer = () => (
  <footer className="footer-content">
    <div className="container">
      &copy; Acme Industries Inc, {new Date().getFullYear()}
    </div>
  </footer>
)

export default Footer;
```

Creating a Component Class

- To define a component class:

JSX

```
import React, { Component } from 'react';
import './styles/Footer.css';

class Footer extends Component {
  render() {
    return (
      <footer className="footer-content">
        <div className="container">
          &copy; Acme Industries Inc, {new Date().getFullYear()}
        </div>
      </footer>
    );
  }
}

export default Footer;
```

- Older versions of JavaScript (ES5) don't have classes

One Top-Level Component

Notice that:

1. There's *one* top-level component only
   - At least for now 😉
2. We `return` this single top-level element
   - We use parentheses around our tags if the tags are on a new line
3. We can add plain HTML into our components:

```
return (
  <div>
    <h1>My Component</h1>
    <p>This is the component's text</p>
    <MyCustomComponent special={true}>A custom component</MyCustomComponent>
  </div>
);
```

4. We `export` our component class at the module level

`render()` is Pure

- `render()` methods (and SFCs) are *pure functions*. They must **not**:
  - change the DOM
  - produce different values for the same state and props
  - have side effects, like calling `setTimeout` or calling HTML5 APIs
- But sometimes we do want to interact with the browser
  - Later we'll use *lifecycle methods*

SFCs vs Component Classes

So when writing a component, what's the difference between using a class versus a function?

SFC

- Simplicity
  - Basically just a render function
- Clarity
  - It's clear: props in, render out
- Can take props, but can't use state ("stateless"!)
- Can be defined where needed
  - even passed as a prop, 'inline'

Component Class

- (A little) more complex
- Defined as an ES6 class
  - A `render()` method to render
  - May have a constructor for initialisation
  - May have component "lifecycle" methods
- May have state (but can be stateless)
- May have event handlers

When to use SFCs

Use SFCs when:

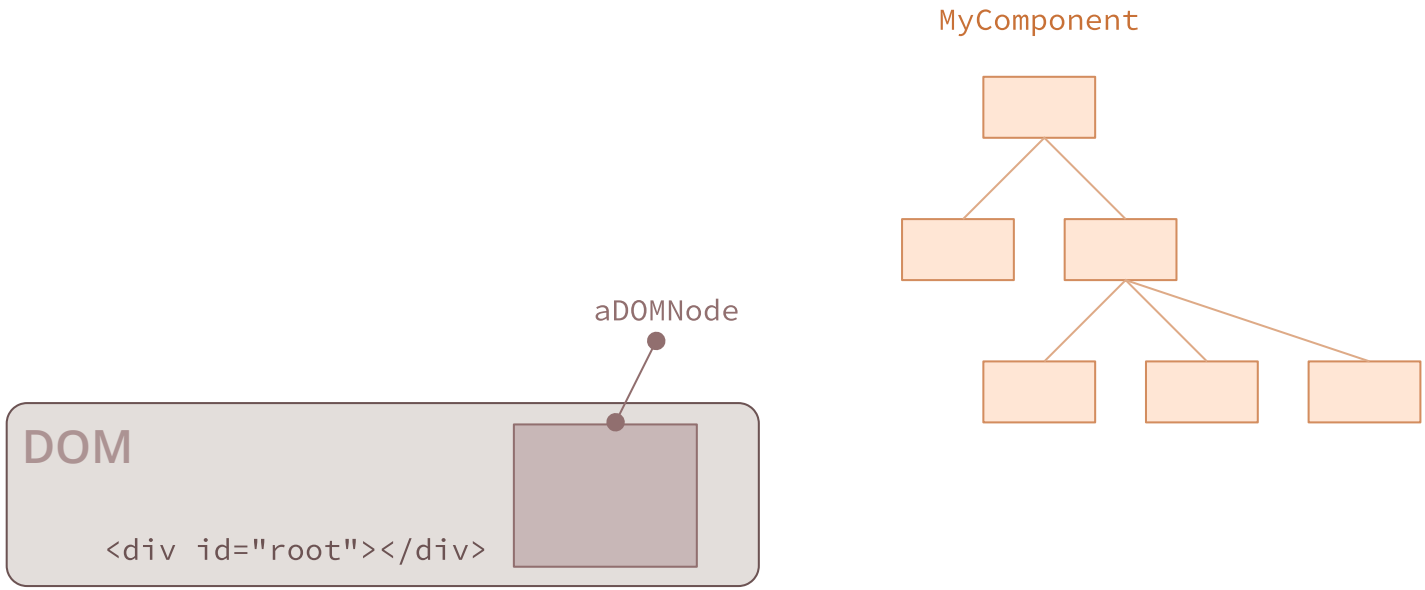
- your component just composes other components
- your component has no state
    - Given the same props, it returns the same HTML

The Top-Level Render

React Fundamentals

Rendering the Top-Level Component

Most React apps have a component tree, and *render* this tree into a single DOM node:

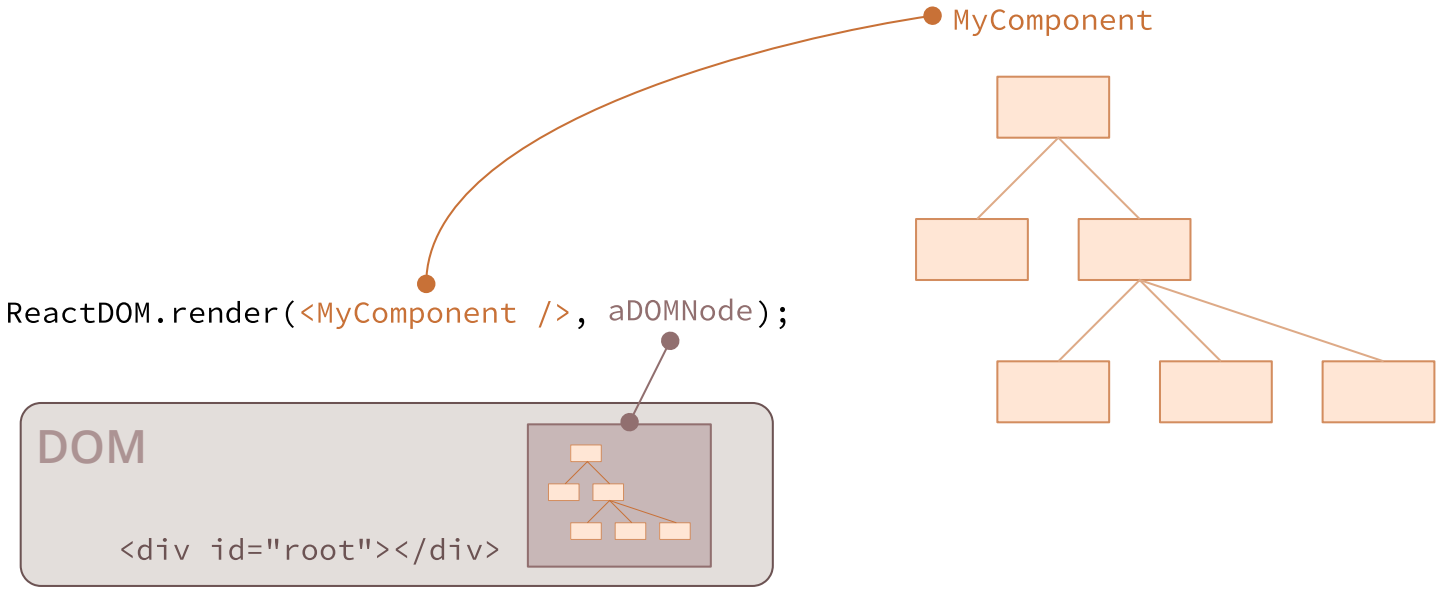


JavaScript

```
const aDOMNode = document.getElementById('root');
```

Rendering the Top-Level Component

To render a component or component tree, we call `ReactDOM.render()`:



JSX

React Fundamentals

What's JSX?

JSX:

- is a syntax embedded in JavaScript to represent rendered output
- looks similar to HTML
- is optional
- is converted into JavaScript function calls

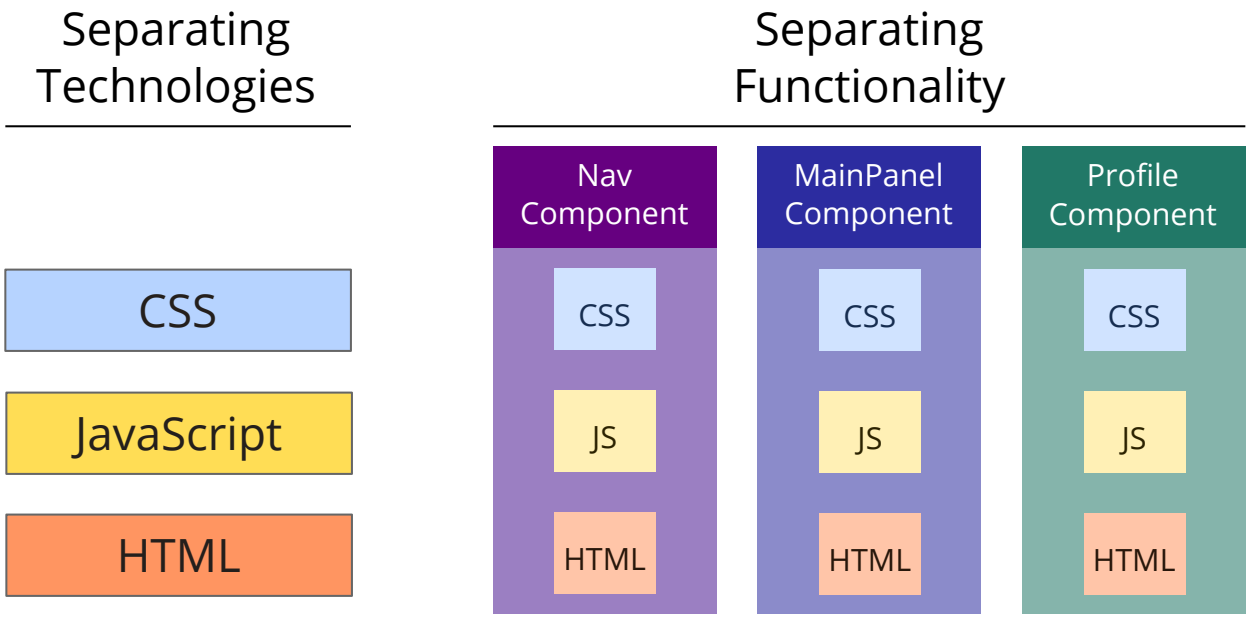
JSX

```
const Panel = props =>
  <div className="panel panel-warning">
    <div className="panel-heading">
      <h2>{props.title}</h2>
    </div>
    <div className="panel-body">
      {props.children}
    </div>
  </div>;
```

Why JSX?

- It makes our rendering code very easy to read and understand
- We specify *what* needs to be rendered (declarative), rather than how (imperative)
- We're not manipulating the DOM
  - We're just saying what we want the DOM to look like
- It's just JavaScript
  - No special syntax for `if` statements, loops or expressions. Just use JavaScript.

Separation of Concerns



- Separating across technology boundaries at first seems like a good approach
- However, components have their own behavior (JS), rendering (HTML) and often styles (CSS)
- These are grouped together within a component; and we separate across functional boundaries

JSX Compilation

- JSX source code cannot be understood by the web browser
  - We compile it to JavaScript when building our app

JSX

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

JavaScript

```
"use strict";

var Hello = function Hello() {
  return React.createElement(
    "div",
    null,
    "Hello, world!"
  );
};
```

- Note that `React` must be in scope (`import`ed) for the compiled code to run
- Popular JSX compilers: Babel, TypeScript

Rendering a DOM Element with a `class`

- We can't use `class="..."` as `class` is a reserved JavaScript keyword
- JSX uses `className` instead

JSX

```
const Welcome = () => (
  <h1 className="text-xs-center text-muted m-y-3">
    Welcome to your Account
  </h1>
);

export default Welcome;
```

Closing Tags and Self-Closing

- All elements must have a closing tag or be self-closing (like XML)
- You can use either form:
  - `<input type="email" />`
  - `<input type="email"></input>`

Wrong:

JSX

```jsx
return (
  <div>
    <img src="logo.png" alt="logo">
    <input type="email">
  </div>
);
```

Better:

JSX

```jsx
return (
  <div>
    <img src="logo.png" alt="logo" />
    <input type="email"></input>
  </div>
);
```

Using Expressions

- Place JavaScript expressions inside `{...}` braces
  - Any JavaScript expression is valid
- Specify nested elements just as you would in HTML
- Entities (`&copy;`) are preserved (unescaped)

JSX

```
const footerClasses = 'footer-content wrapper';

const Footer = () => (
  <footer className={footerClasses}>
    <div className="container">
      &copy; Acme Industries Inc, {new Date().getFullYear()}
    </div>
  </footer>
);

export default Footer;
```

Rendering `null`, `undefined`, `true`, `false`

- All of these can be safely returned
  - Each renders nothing
- They're useful when you don't want to render

JSX

```
class MyComponent extends Component {
  render() {
    // ...
    if (dataPending) {
      return null;
    } else {
      return (
        <div>
          <p>Content here...</p>
        </div>
      );
    }
  }
}
```

Beware of ASI

- JavaScript has Automatic Semicolon Insert (ASI)
  - This can lead to unexpected results

Wrong

JSX

```jsx
class MyComponent extends Component {
  render() {
    return
    (
      <div className="col-md-9">
        content here...
      </div>
    );
  }
}
```

Right

JSX

```jsx
class MyComponent extends Component {
  render() {
    return (
      <div className="col-md-9">
        content here...
      </div>
    );
  }
}
```

DOM Tags vs Component Tags

- The `<tagname>` can start with an uppercase or lowercase letter:
  - `<p>` is a *DOM Tag* and always renders a DOM element
  - `<Panel>` is a *Component Tag* and always renders a React component

JSX

```
const Error = props =>
  <Panel title="Temporary Error">
    An error occurred:
    <ul>
      <li>{props.error.message}</li>
      <li>{props.error.code}</li>
    </ul>
  </Panel>;
```

In this code:

- `Panel` generates a call to the function `Panel`
- `ul` and `li` generate DOM elements

Rendering Children

React Fundamentals

Rendering Children

- We know that JSX expressions are function calls
- So rendering children is just a matter of *choosing which functions to call*
- We can do this with plain JavaScript

JSX as an Expression

JSX can be used anywhere in your source file:

JSX

```
const container = <VictoryContainer title={this.props.title || 'Event chart'} />
```

JSX

```
const message = event.message && (event.message.replace(/\s*More info[\s\S]*/, ''))
                || <span className="text-muted">No message found</span>;
```

JSX

```
return (
  <div>
    {props.isFirst ? <NodeStartIcon /> : <NodeIcon />}
  </div>
);
```

Saving Components in a Variable

We can store JSX expressions in temporary variables:

JSX

```
render() {
  // ...
  let mycontent = <p>Content here...</p>;
  return (
      <div>
        { mycontent }
      </div>
    );
  }
}
```

Using `if...then` at the Component's Root

This example...

JSX

```jsx
render() {
  // ...
  if (dataPending) {
    return null;
  } else {
    return (
      <div>
        <p>Content here...</p>
      </div>
    );
  }
}
```

can be shortened to:

JSX

```jsx
render() {
  // ...
  return dataPending ? null : (
      <div>
        <p>Content here...</p>
      </div>
    );
  }
}
```

Is `if...then` an expression?

Can we do this?

JSX

```
render() {
  <div>
    {
      if (hasMorePages) {
        <p>Next Page</p>
      } else {
        <p>[Last Page]</p>
      }
    }
  </div>
}
```

- The problem here is that an `if` statement is not an expression
  - It doesn't yield a value, so this will not work

(Is `if...then` an expression?)

The solution is to use the ternary operator:

JSX

```
render() {
  <div>
    {
      hasMorePages
      ? <p>Next Page</p>
      : <p>[Last Page]</p>
    }
  </div>
}
```

- The ternary operator is an expression (yields a result), so can be used where a value is expected

Remember: JSX compiles to Function Calls

This:

JSX

```
var content = <Container>{isLoggedIn() ? <Nav /> : <Login />}</Container>;
```

...compiles to this:

JavaScript

```
var content = React.createElement(
  Container,
  null,
  isLoggedIn() ? React.createElement(Nav) : React.createElement(Login)
);
```

Using `&&` as a shorthand `if`

A common JavaScript idiom is:

JavaScript

```
let previouslyVisitedUser = true;
console.log(previouslyVisitedUser && 'Welcome back');          // "Welcome back"
```

We can use this in JSX:

JSX

```
class UserProfile extends Component {
  render() {
    let dataPending = true;
    return (
      <div className="col-md-9">
        {dataPending && <div>Fetching data...</div>}
        <p>Render retrieved data...</p>
      </div>
    );
  }
}
```

Rendering Children

When you create a component, between the opening and closing tags, you can add:

- child components
- further DOM elements

JSX

```
<MyComponent>
    <h1>Heading One</h1>
</MyComponent>
```

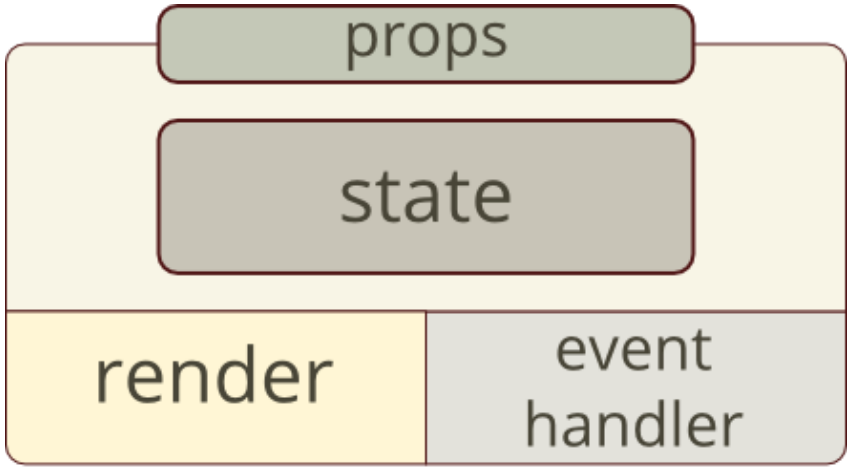
- `MyComponent` can see the children passed from above via `this.props.children`
  - It's normally an array of components, but for an only child it's the child instance
  - Use the `React.Children` utilities to access them

Data and State

React Fundamentals

Props and State

- Components:
  - can be supplied with **properties** or *props*
  - and can have **state** (ideally none)



Props

- think of HTML attributes
- properties passed in from outside
- considered immutable

Specifying Props

An owner component sets the props of the owned component:

JSX

```jsx
<StockCell symbol="AAPL" fundamentals={fundsObj} />
```

Using Props in a Component

JSX

```jsx
<p>{this.props.symbol}</p>
```

Props Example

Hello!

Here's the render() method:

JavaScript

```
import React, { Component } from 'react';

class Message extends Component {
    render() {
        return (
            <p>{this.props.message}</p>
        );
    }
}
```

JavaScript

```
ReactDOM.render(
    <Message message={randomMessage}/>,
    document.getElementById('message')
)
```

Property Validation

React Fundamentals

Property Validation

- The Problem:
  - Sometimes due to programmer error, input properties to components may be specified incorrectly
- The Solution:
  - Components can validate their properties
  - Property Validation is enabled during *development* mode
    - It'll display an error in the console
    - Production mode disables it for performance

Specifying PropTypes

- `propTypes` are mostly declarative
- You specify two things:
  - the property name
  - the validation rules (typically the expected type)
- Validators are simply functions that are given the props to validate
  - These functions are part of `PropTypes`

Validation Examples

Property Validation example:

```
class MyArticle extends Component {
  render() { ... }
});

MyArticle.propTypes = {
    // This component accepts an optional string prop named "description":
    description: PropTypes.string,

    // ...and a required enum prop named "category":
    category: PropTypes.oneOf(['News','Photos']).isRequired,

    // ...and a prop named "dialog" that requires an instance of Dialog:
    dialog: PropTypes.instanceOf(Dialog).isRequired
};
```

Prop Validation Types

You can have React check that property `foo` is:

DESCRIPTION	VALIDATOR

A specified type	`PropTypes.array`
	`PropTypes.bool`
	`PropTypes.func`
	`PropTypes.number`
	`PropTypes.object`
	`PropTypes.string`

A *single* React Element	`PropTypes.element`
A React Node (number, string element, or an array of those)	`PropTypes.node`
An instance of a class	`PropTypes.instanceOf(MyClass)`
An array of a certain type	`PropTypes.arrayOf(PropTypes.number)`
An object with property values of a certain type	`PropTypes.objectOf(PropTypes.number)`

(Prop Validation Types)

DESCRIPTION	VALIDATOR

Is one of a set of types	`PropTypes.oneOfType([`
	`  PropTypes.string,`
	`  PropTypes.number,`
	`  PropTypes.instanceOf(Message)`
	`])`

An object of a particular shape	`PropTypes.shape({`
	`  color: PropTypes.string,`
	`  fontSize: PropTypes.number`
	`})`

Required	Add `.isRequired`
Is one of a set of values	`PropTypes.oneOf(['EUR', 'USD', 'GBP'])`

Custom Prop Validation

- You can also write your own property validators:

JavaScript

```
MyComponent.propTypes = {
  // An optional string or URI prop named "href".
  href: function(props, propName, componentName) {
    const propValue = props[propName];
    if (propValue != null && typeof propValue !== 'string' &&
        !(propValue instanceof URI)) {
      return new Error(
        'Expected a string or an URI for ' + propName + ' in ' +
        componentName
      );
    }
  }
};
```

Events

React Fundamentals

Handling Events

To handle events, create an *Event Handler*

- An event handler is simply a function that responds to the event
- We'll also look at "binding" event handlers in the next chapter

DOM Events in HTML

Click Me - A traditional DOM event handler

```
<h3 onclick="alert('HTML event')">
  Click Me - A traditional DOM event handler
</h3>
```

Handling a Click Event in React

Click Me - A React event handler

```
import React, { Component } from 'react';

class ClickDemo extends Component {

  render() {
      return (
        <h3 onClick={ this.handleClick }>
          Click Me - A React event handler
        </h3>
      );
  }

  handleClick(e) {
      alert('React event - see console for SyntheticEvent details');
      console.log('React event:', e);
  }
}
```

React Events

- In HTML:
  - We do: `onclick="..."`
- In React:
  1. We use camelCase: `onClick`, not `onclick`
  2. No quotation marks! `onClick={...}`

Binding Event Handlers

Why 'bind' event handlers?

1. Most event handlers need to access `this`
    - They typically need to change some state
2. We also pass event handlers around

The Event Binding Backstory

Consider:

JavaScript

```
let mycar = {
  make: 'Maserati',
  speed: 0,
  display() { console.log(`My ${this.make} is traveling at ${this.speed}`)},
};

mycar.speed = 30;
mycar.display();      // "My Maserati is traveling at 30"

let foo = mycar.display;
foo();                // "My undefined is traveling at undefined"
```

The Event Binding Backstory -- 2

- In JS we can 'borrow' or pass a function and invoke it elsewhere
- JS has 4 different ways of invoking a function
    - Method invocation sets `this` to the object called upon
    - Function invocation sets `this` to `undefined`

Binding Approaches

You can use one of the following approaches:

1. Bind in the constructor
2. Use an arrow function in `render()`
3. Use a ES20XX (Stage 3) class instance property

State

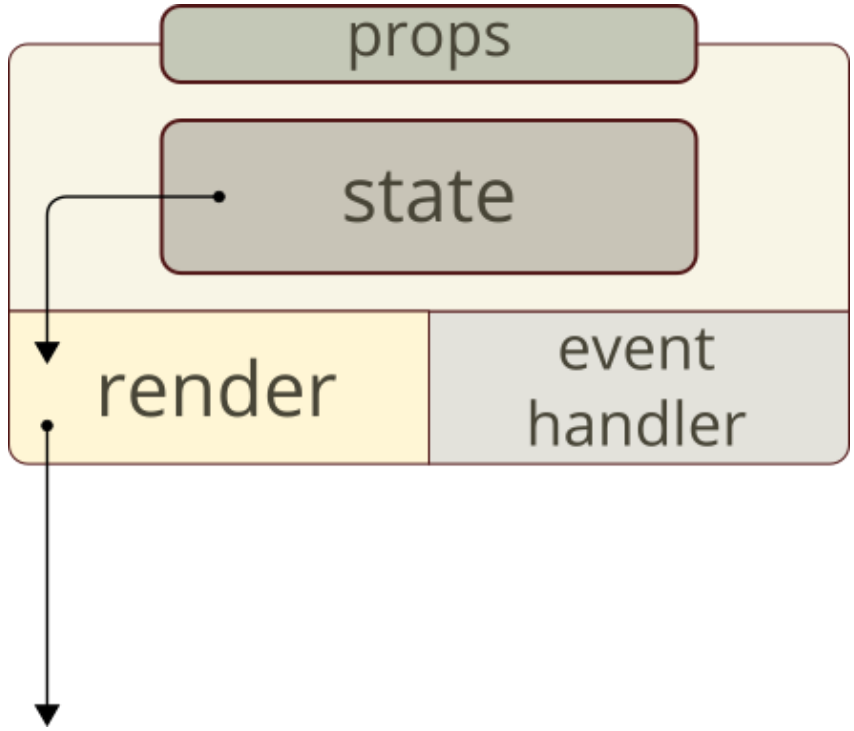
React Fundamentals

What's State?

- Internal data the component needs to render itself
- **Private** to a component
- Examples:
  - a boolean that says "I have read the Terms & Conditions"
  - a string that contains the value of a username input field
- Ideally, components would have minimal (or zero) state
  - However some components are best written using local state

Using State

- State is stored in a JavaScript object within a React component
- Whenever the state changes, React re-renders the component
  - Actually when `setState()` is called



Constructor

In ES6, set up your state as follows:

```
class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { counter: 1 };
  }
```

- This initialization is the one *and only* time we write directly to the `state` object
  - Any future state changes are made via `setState()`

`setState()`

```
setState({mykey: 'my new value'});
```

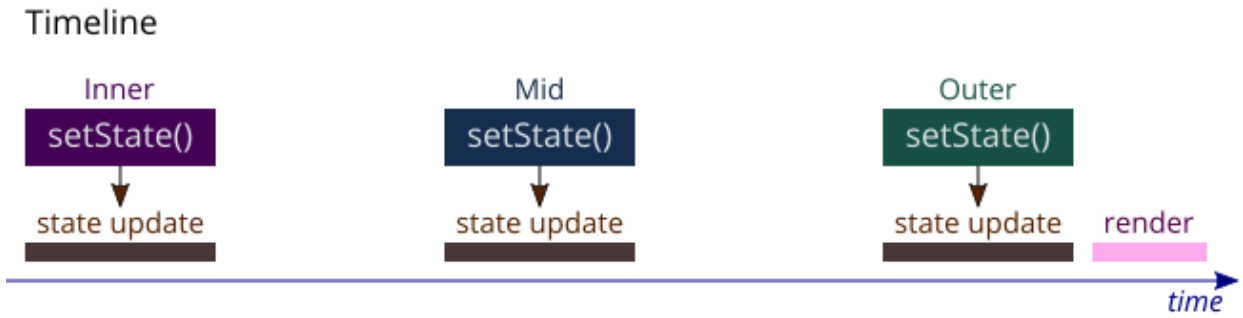
- Merges the state you provide into the current state
- *Never* change `this.state` directly!
  - Always use `this.setState()`
- To modify state, there are two forms:

JSX

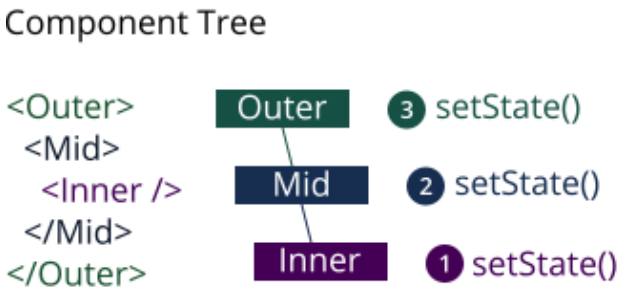
```jsx
// this form takes a function:
this.setState((state, props) => ({ counter: state.counter + 1 }));

// this form takes an object:
this.setState({ value: 42 });
```

State Updates are Batched

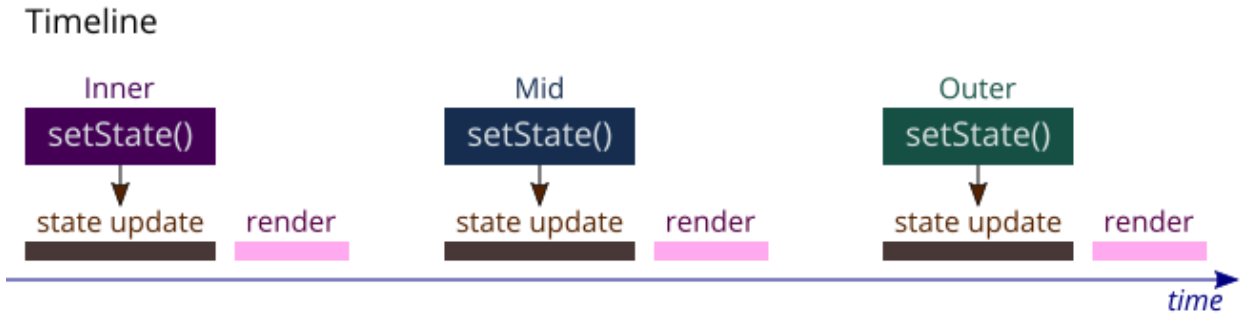
- To avoid multiple re-renders, state updates are *batched* together
  - Each `setState()` queues its state update
  - A series of updates is then processed during a single render



…and if we don't batch state updates?

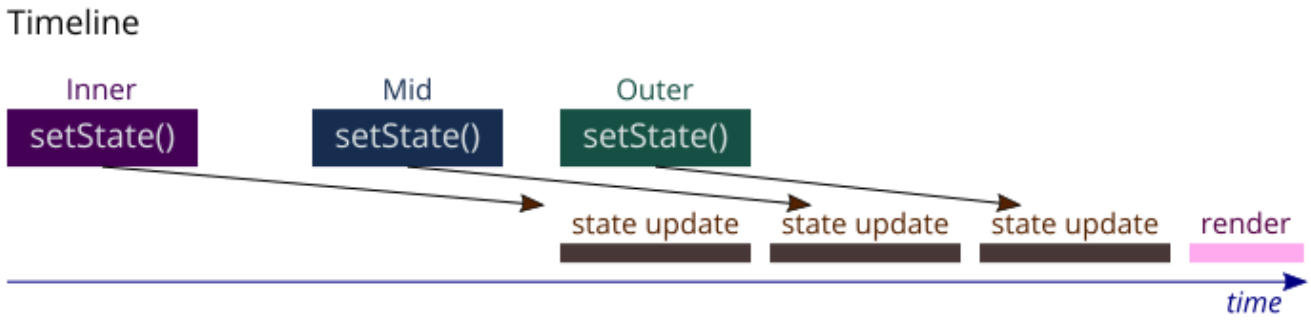


- Each component in this tree could call `setState()`, which triggers a render
- What if `Inner` updates its state, then notifies `Mid` which then updates its own?
  - We'd end up re-rendering twice, and similarly for `Upper`
  - This is obviously inefficient and would lead to an unresponsive UI



Treat State Updates as Asynchronous

- State updates are also likely to be *asynchronous* for performance
  - So you can't rely on the state having been updated immediately after `setState()`
  - But it will have happened before rendering



Treat State Updates as Asynchronous -- 2

So this means we can get:

```
// this.state.count was initialized to 0
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
// this.state.count is 1, not 2
```

See example on codepen

Interaction

React Fundamentals

Adding Interaction

0

JavaScript

```
class Counter extends Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };
        this.click = this.click.bind(this);
    }
    render() {
        return  <button id="counter" onClick={this.click}>
                    {this.state.count}
                </button>;
    }
    click() {
        console.log("click");
        this.setState(prevState => ({ count: prevState.count + 1 }));
    }
}
```

Updating a Field

Type something

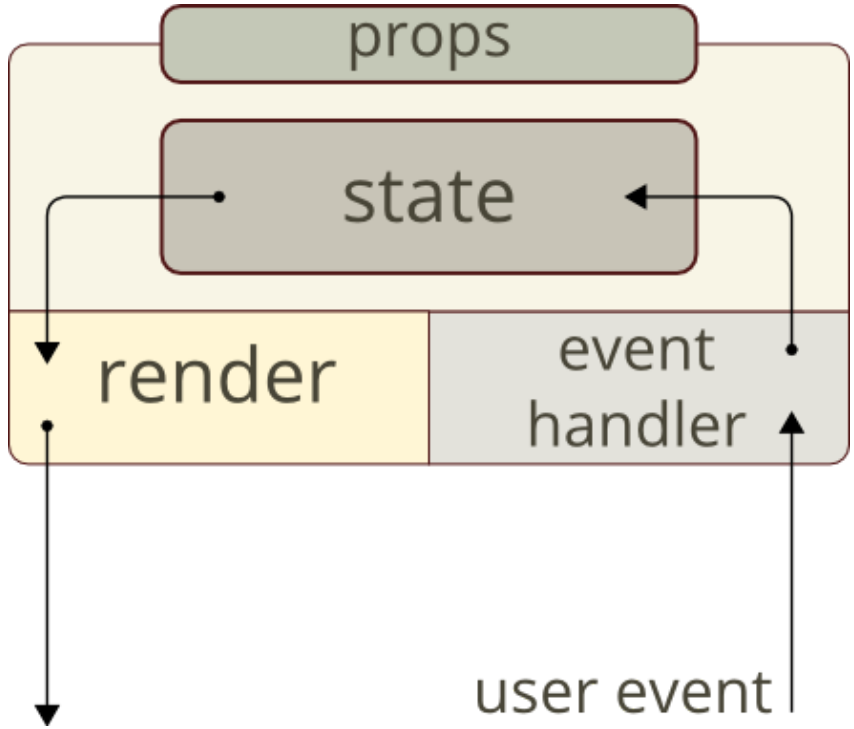
Start

JavaScript

```
setInterval(function() {
    log('react-interaction-log', 'redrawing...');
    React.render(<UpdatingField />,
            document.getElementById("react-interac
}, 2000);
```

Data Flow in React

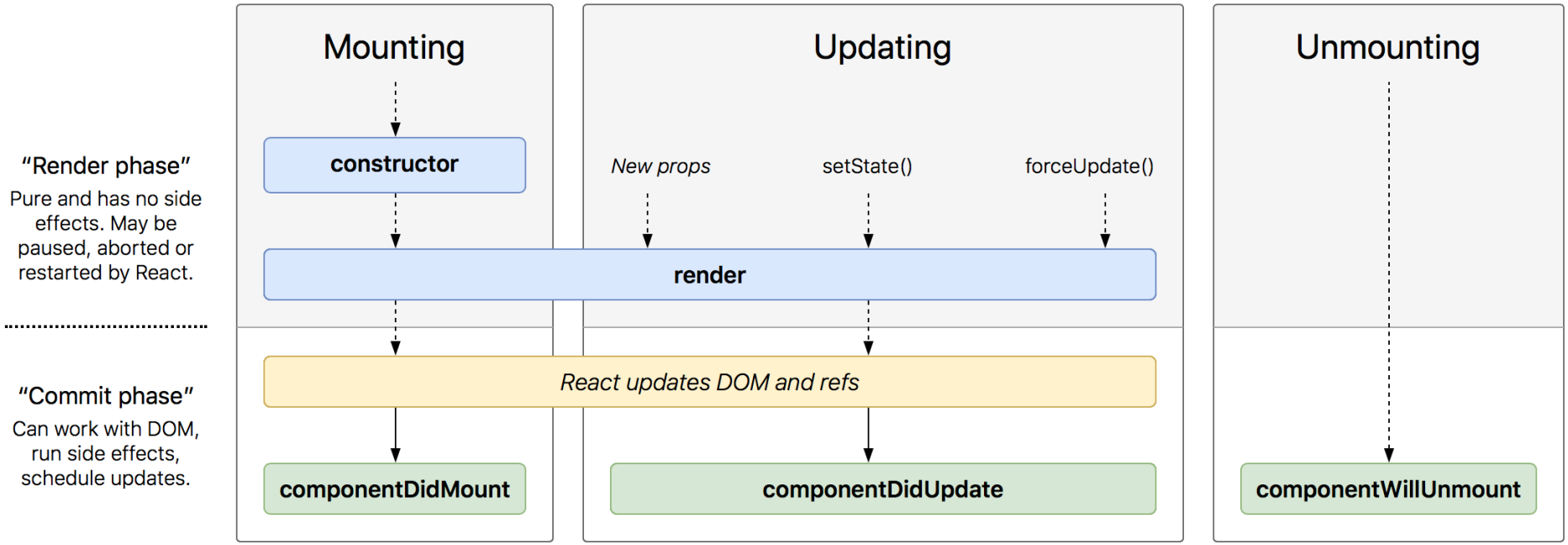
- React uses unidirectional data flow
  - Data flows in one direction
- Update state and re-render



Component Lifecycle

React Fundamentals

Component Lifecycle Methods - Overview



- diagram courtesy of Dan Abramov and Wojciech Maj

Component Lifecycle Methods

Mounting

- `constructor()`
  - the first "method" to be invoked
  - typically used for initialization, for example:
    - initializing state
    - setting up event handlers
- `render()`
  - a pure function that simply returns the output to render (typically React elements)

Component Lifecycle Methods

(Mounting)

- `componentDidMount()`
  - invoked once, immediately after initial rendering
  - children will have been rendered (and their `componentDidMount()`s will have been called), so you can access their refs
    - integrate with third-party libraries, set timers, make AJAX requests

(Component Lifecycle Methods)

Updating

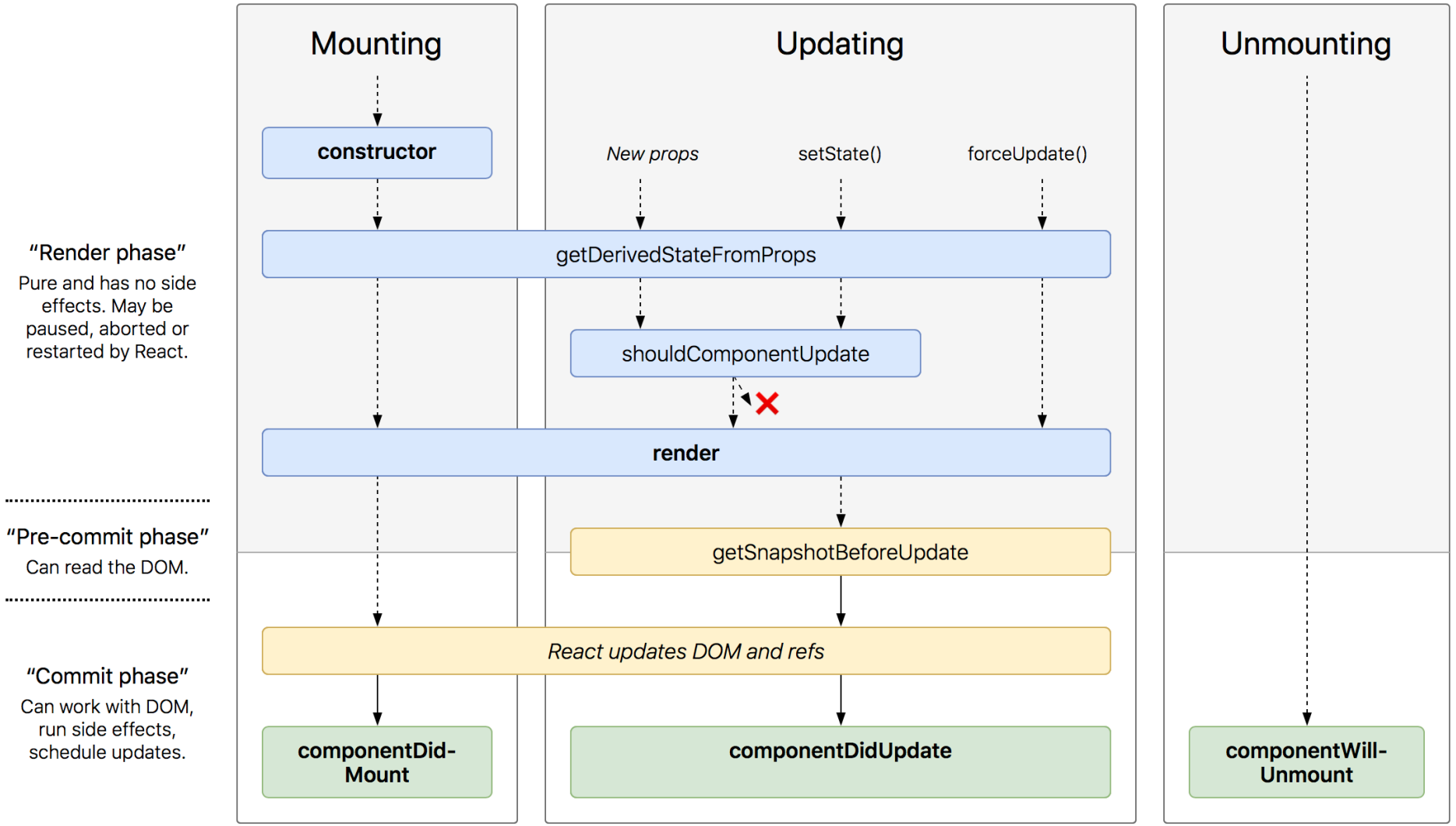
- `render()`
  - a pure function that simply returns the output to render (typically React elements)
- `componentDidUpdate(prevProps, prevState)`
  - called after the component has been rendered to the DOM
  - not called for the initial render

(Component Lifecycle Methods)

Unmounting

- `componentWillUnmount()`
  - called immediately before a component will be removed from the DOM
  - use this for any cleanup

Further Lifecycle Methods



Further Lifecycle Methods

Updating

- `static getDerivedStateFromProps(nextProps, currentState)`
  - called before every render
  - used to update state before the next render
    - often we update state based on the `nextProps`
  - it returns the new state as an object (similar to `setState()`)
  - use sparingly!
- `shouldComponentUpdate(nextProps, nextState)`
  - allows a component to veto re-renders
  - React renders a component and all its children when its state changes
    - return false to avoid rendering even to the VDOM
  - use only when you know its needed - don't optimize prematurely!

(Further Lifecycle Methods)

(Updating)

- `getSnapshotBeforeUpdate(prevProps, prevState)`
  - invoked after render, but before the output is finally committed to the DOM
  - why:
    - with async rendering, after `render()`, there's a potential delay before updating DOM
    - if the user interacts with the app in the meantime, scroll positions etc may have changed
    - enables you to capture latest interaction status from the DOM
    - more info in the RFC
  - return value is passed later as final parameter to `componentDidUpdate(prevProps, prevState, snapshot)`

Lifecycle Method Summary

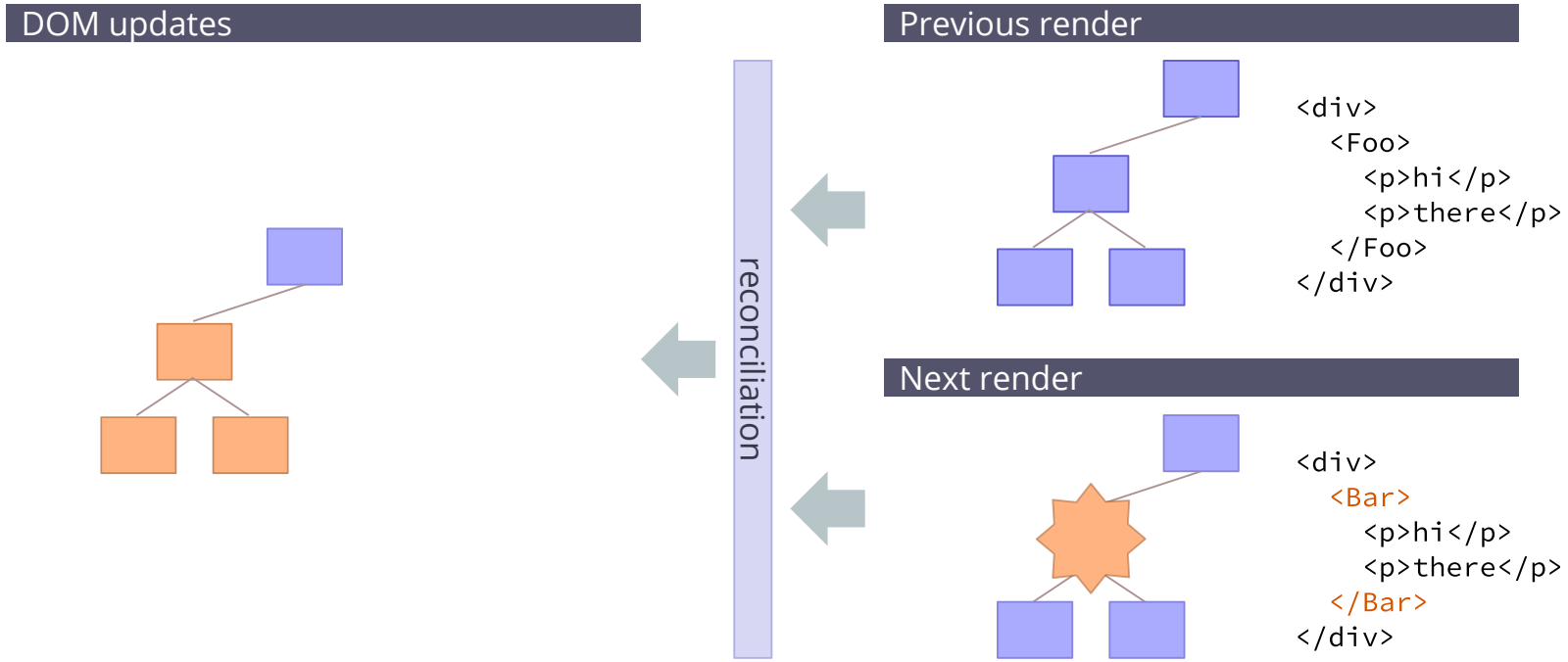
These methods are called:

METHOD	INITIAL RENDER	WHEN	CLIENT OR SERVER
`componentWillMount` (deprecated)	Yes	Before initial render	Both
`componentDidMount`	Yes	After initial render	Client
`componentWillReceiveProps` (deprecated)	No	On new props	Client
`getDerivedStateFromProps` (static, 16.x)	No	On new props	Client
`shouldComponentUpdate`	No	Before rendering; allows veto	Client
`componentWillUpdate` (deprecated)	No	Before rendering	Client
`componentDidUpdate`	No	After DOM update	Client
`componentWillUnmount`	No	Before unmount; for cleanup	Client
`getSnapshotBeforeUpdate` (16.x)	No		
`componentDidCatch` (16.x)	No	On error	Client

Rendering Lists and Reconciliation

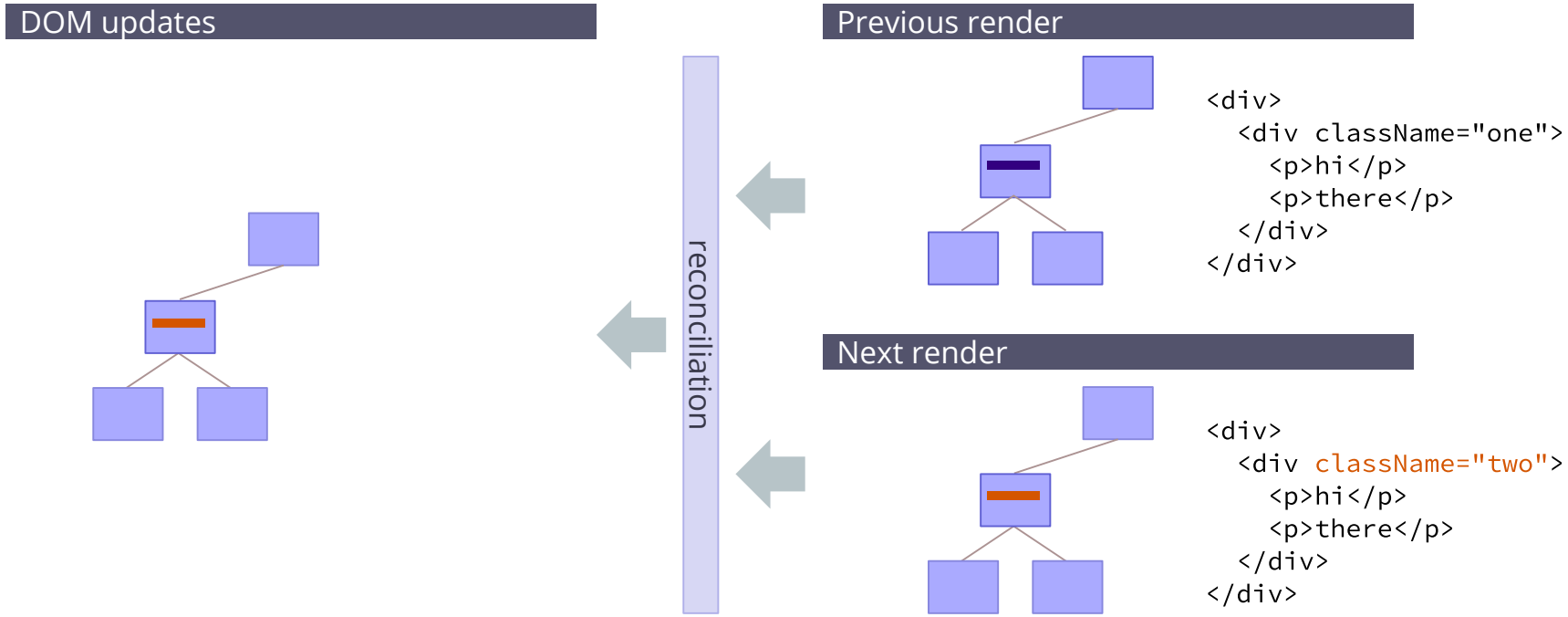
React Fundamentals

Reconciliation -- Different Types



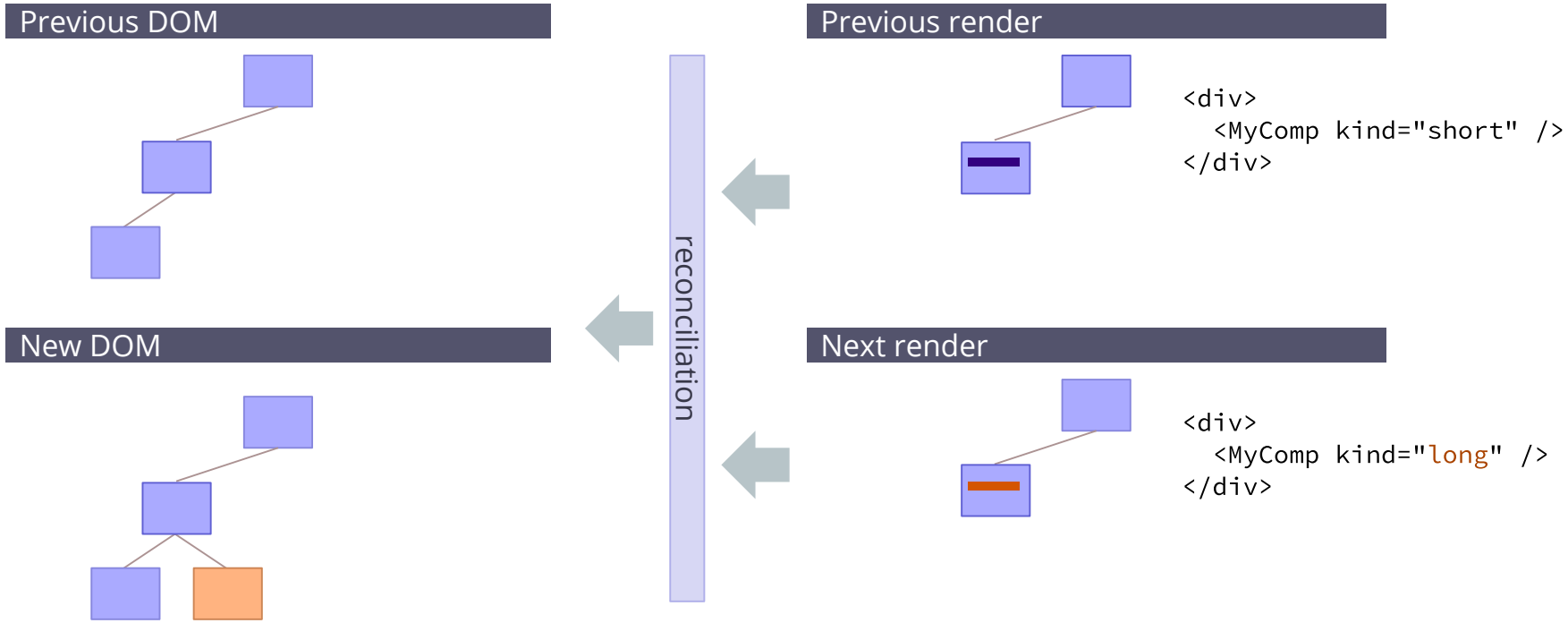
To render a different type, React scraps the old DOM subtree and builds a new one

Reconciliation -- Same DOM Types



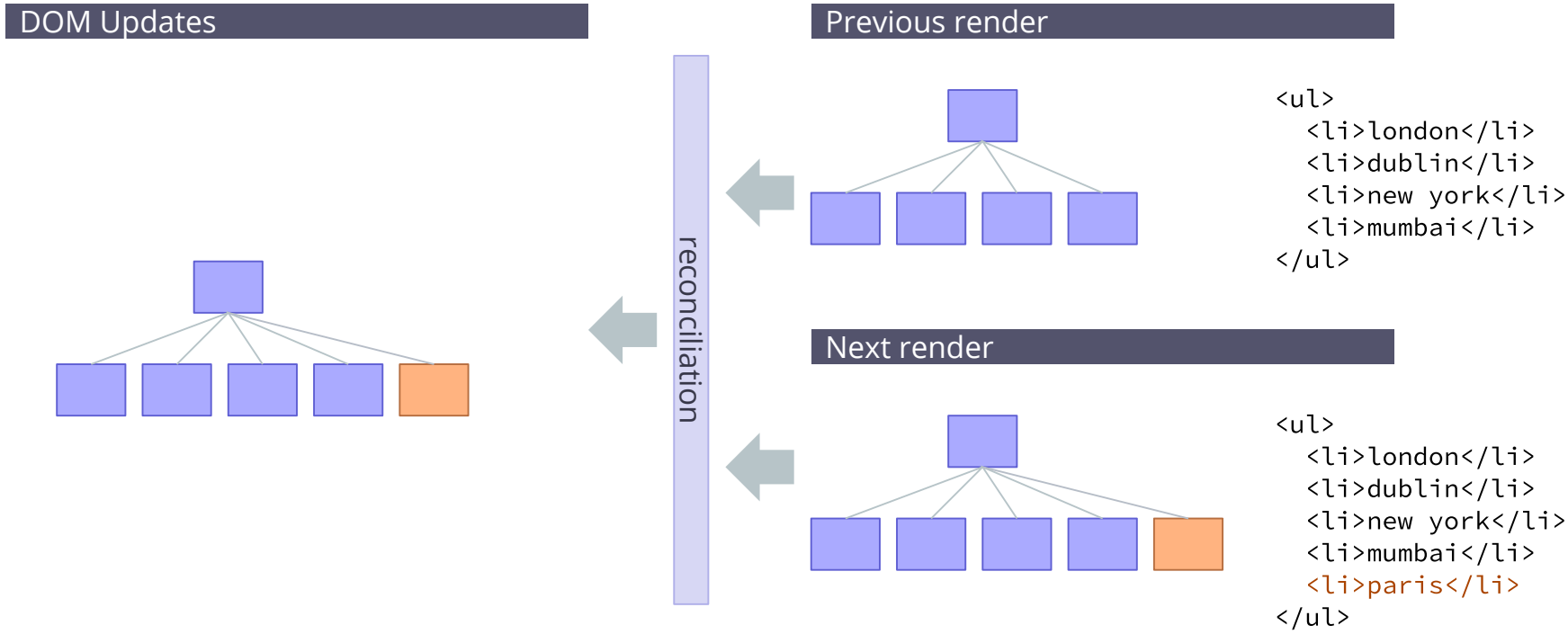
If a DOM element's type remains the same but its attribute changes, React just modifies the attribute in the DOM

Reconciliation -- Same Component Types



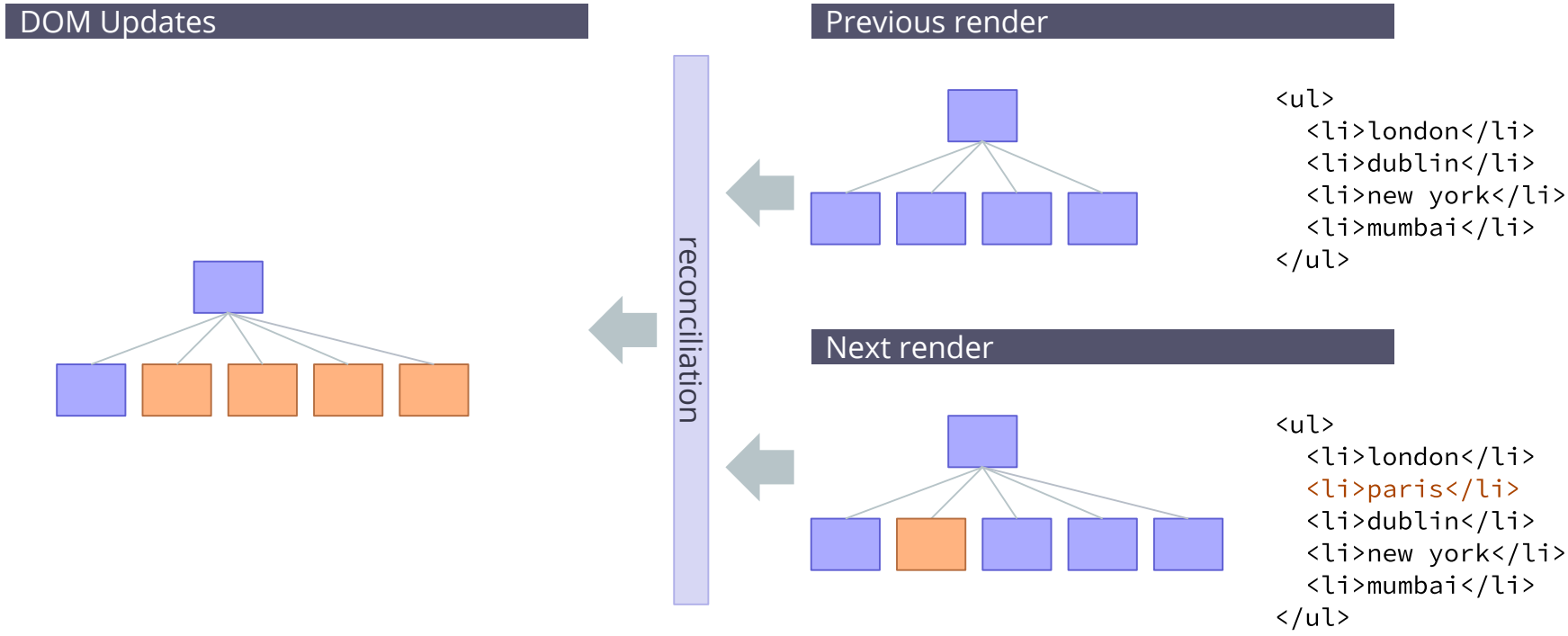
This Component is given new props, with which it renders more detail. React: (1) updates the component instance; (2) calls `getDerivedStateFromProps()` and `componentDidUpdate()`; (3) calls `render()`, recursively diffing the output with the previous render

Reconciliation -- Appending to a List



Appending to a list is as simple as adding an element

Reconciliation -- Inserting into a List



Inserting into a list is harder: React compares corresponding elements, finds they differ, and has to update *every* subsequent element!

Keys

To help React reconcile differences between renders:

- When creating multiple child components, provide a `key` for each one:

```
    <li key={author.id}>
```

- This enables React to match up correctly when adding or deleting
  - The key only needs to be unique within its parent
  - You can reuse a database key if your items have one

Stateful Children

- A stateful component holds state in `this.state`
- Having the VDOM re-use a different element can be a problem
- Hide elements (`display: none`) rather than remove them

Transferring Props

React Fundamentals

Props

- Props are like HTML attributes, but more flexible:
  - They can take object values
  - We can use them as input parameters to our components
  - We can 'merge' them from a parent component into a child component
- A common pattern:
  - Take a complex component (or set of components), and wrap it in a simpler interface
    - Examples: Griddle or FixedDataTable

Transferring Props

- We often need to transfer props down the component tree
- We typically pass them down explicitly and manually

Suppose we have:

```
const props = {
  color: "red",
  size: "medium",
  on: true
};
```

```
<Light color={this.props.color}
       size={this.props.size}
       on={this.props.on} />
```

The Spread Operator

- JSX provides the *spread operator*, similar to that in ES6

JSX
```
<Light {...props} />
```

- All props are transferred from parent to child
- To exclude props, you can use ES6 destructuring assignments
    - Then just pass in the `other` props:

JSX
```
const { checked, ...other } = props;
```

Forms

React Fundamentals

Forms

- Form components (like `<input>`, `<textarea>`, and `<option>`):
  - support "interactive" props
  - they're affected by user input
- Examples:
  - `value`, on `<input>` and `<textarea>` components
  - `checked`, on `<input>` components of type `checkbox` or `radio`
  - `selected`, on `<option>` components

Form Example

foo

Uncontrolled Components

- Uncontrolled components don't have a `value` prop

JSX

```
class InputUncontrolled extends Component {
  render() {
    return (
      <textarea className="biginput"></textarea>
    );
  }
});
ReactDOM.render(<InputUncontrolled/>,
  document.getElementById("form-uncontrolled"));
```

Controlled Components

- Components like `<input>` or `<textarea>` can have a `value`
  - If set, React *controls* the element
    - The `value` is maintained by React
    - Can't change it without an `onChange` handler!

JSX

```
render() {
  return (
    <textarea value="Here is some text"></textarea>
  );
}
```

Here is some text

(Controlled Components)

To enable changes to be made:

1. Add an `onChange` handler to get the element's value:
   - `<textarea onChange={this.handleChange} ...>`
2. Update this value in the component's `state`
   - `handleChange: function(e) { this.setState({value: e.target.value}); }`
3. Use the component's state to propagate this value back to the DOM element
   - `<textarea onChange={...} value={this.state.value}>`

Handling Errors

React Fundamentals

React's Error Behavior

React 15 and earlier

- On an error:
  - Displays the error in the console
  - Renders broken UI
    - Some parts may be displayed, others not
    - Unclear to user
    - Potentially allow the user to invoke illegal operations

React 16

- On an error:
  - the whole component tree is unmounted
- Rationale: better to reveal errors rather than render corrupted UI

`componentDidCatch()`

- Lifecycle method
- Called when a *child* component error occurs
  - Think of it like a `catch` clause in a programming language
- Catches errors in:
  - render methods
  - constructor
  - lifecycle methods
  - setState callbacks
- But not in:
  - event handlers
  - other asynchronous code

JavaScript

```
componentDidCatch(error, info) {
  this.setState({ hasError: true });
  myLogError(error, info);
}
```

Error Boundaries

An error boundary is:

- a component…
  - which implements `componentDidCatch()`
  - above which errors will not propagate further
  - which catches errors generated in its child components
  - which renders some fallback UI when an error occurs
  - which may optionally log the error
    - often to an online error reporting service

Using an Error Boundary

To handle errors in a component:

- Create an `<ErrorBoundary>` component you can use like this:

JSX

```
render() {
  return (
    <ErrorBoundary>
      <MyComponent />
    </ErrorBoundary>
  );
}
```

Creating an Error Boundary Component

- Your `<ErrorBoundary>` component should:
  - Render its children
  - If an error occurs (`componentDidCatch()` and `setState()`), render fallback UI

JSX

```jsx
class ErrorBoundary extends Component {
  render() {
    if (this.state.hasError) {
      return <div className="error">oops! Something went wrong</div>;
    }
    return this.props.children;
  }
  //...
}
```

Refs

React Fundamentals

Why Refs?

- First, a ref:
  - is an "escape hatch" out of React's Virtual DOM
  - is a JS reference to a real DOM element
  - enables DOM (and React) child elements to be accessed after rendering

But why??

Some use cases need DOM access. For example:

- Drawing on a `<canvas>` after it's been rendered
- Calling third party libraries that write directly to DOM nodes
- Managing `<audio>` or `<video>` elements (pause, skip etc)

How to use Refs

Here's an example that manages input focus:

JSX

```jsx
class MyComponent extends React.Component {
  divRef = React.createRef();

  render() {
    return <div><input type="text" ref={this.divRef} /></div>;
  }

  componentDidMount() {
    this.divRef.current.focus();
  }
}
```

Styling Components

React Fundamentals

Styling Components

- We're styling *components*, not pages or websites
    - Often we'd prefer a local scope (but with some global defaults)
- We'd like to build or use component libraries

What do we need from styles?

- Reuse of styles across components?
- The CSS cascade?
- Manipulate using JS?
- Isolation via local scoping?

Styling Approaches

- Vanilla CSS
- CSS-in-JS
    - Using styles in JSX
    - Styled Components
- CSS Modules

Vanilla CSS

- create-react-app uses this approach by default
- stylesheets are `import`ed by Webpack
- easy: `import "./MyComponent.css";`
- bear in mind that styles are globally scoped
    - so component-specific styles can leak out

Aside: Bootstrap

- react-bootstrap
    - a re-implementation of Bootstrap components in React

CSS-in-JS

- For a comprehensive list, see Michele Bertolli's css-in-js

A Popular Solution: Styled Components

Styled Components:

- uses JS tagged template literals to style your components

Design Goals:

- No build requirements
- Small and lightweight
- Supports global CSS
- Supports entirety of CSS
- Colocated
- Isolated
- Easy to override
- Theming
- Server side rendering
- No wrapper components

CSS Modules

- CSS is contained in smaller "module" files

JavaScript
```
import styles from "./style.css";
// import { className } from "./style.css";
```

Pros:

- Locally scoped
  - Though you can export to the global scope when needed
- Vanilla CSS
  - Not a standard

Cons:

- Styles need to be precompiled
  - Adding components from a library means you have to add the precompile step to your build

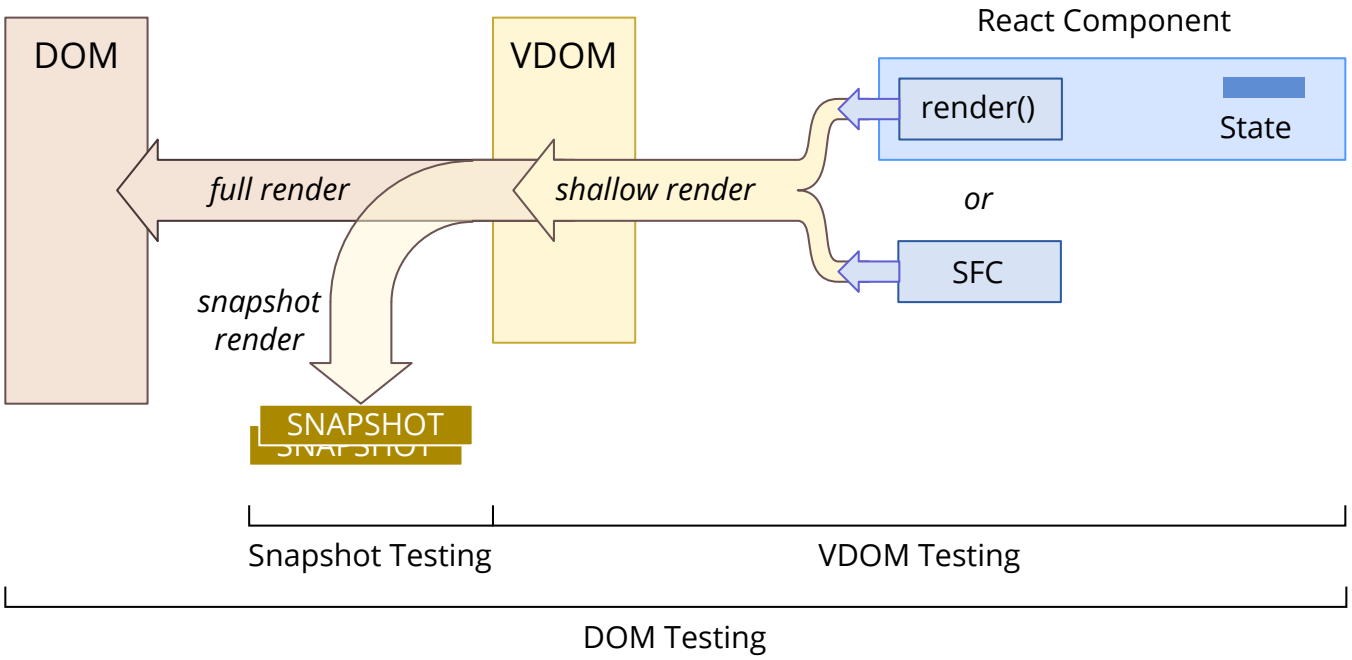
See https://github.com/css-modules/css-modules for more info

Testing

React Fundamentals

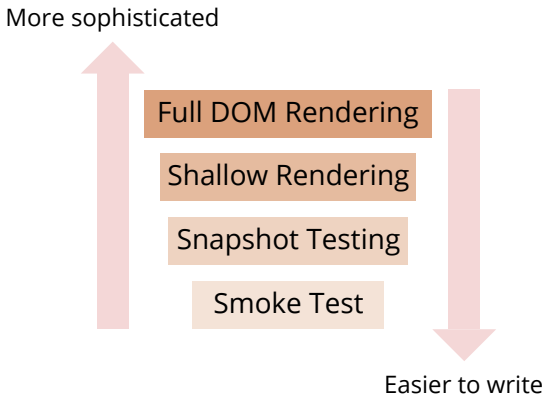
Testing Approaches

These approaches are common in React testing:



Levels of Testing - A *Rough* Guide

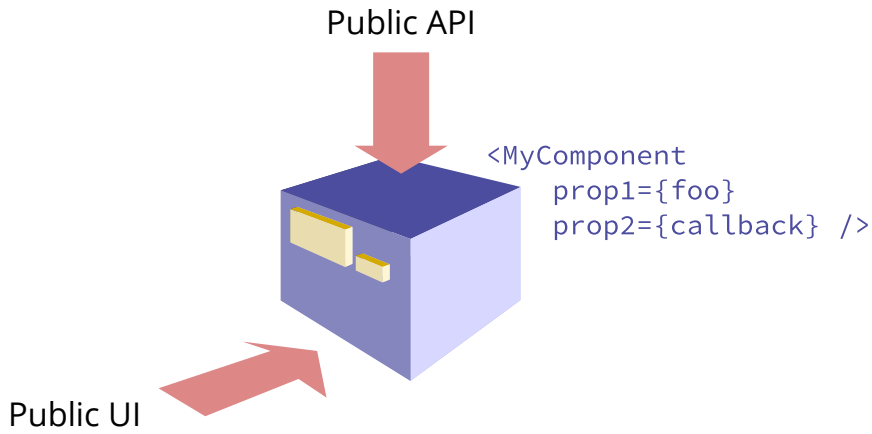
- **Full Rendering**
  - Test a component while rendering its children too
  - Renders to the DOM
- **Shallow Rendering**
  - Test a component in isolation from its children (and parent)
  - Renders to the VDOM, but only one level deep (doesn't render child React components)
- **Snapshot Testing**
  - Renders to the VDOM, writing the serialized rendering to a 'snapshot' file
  - Compares this snapshot with a previously-rendered one
- **Smoke Test**
  - Test that component renders without throwing an error



Components' Two Interfaces

Unlike traditional objects or functions, components have two interfaces:

- Public API
  - The component's name and props, including callbacks
- Public UI
  - The interface that the user interacts with



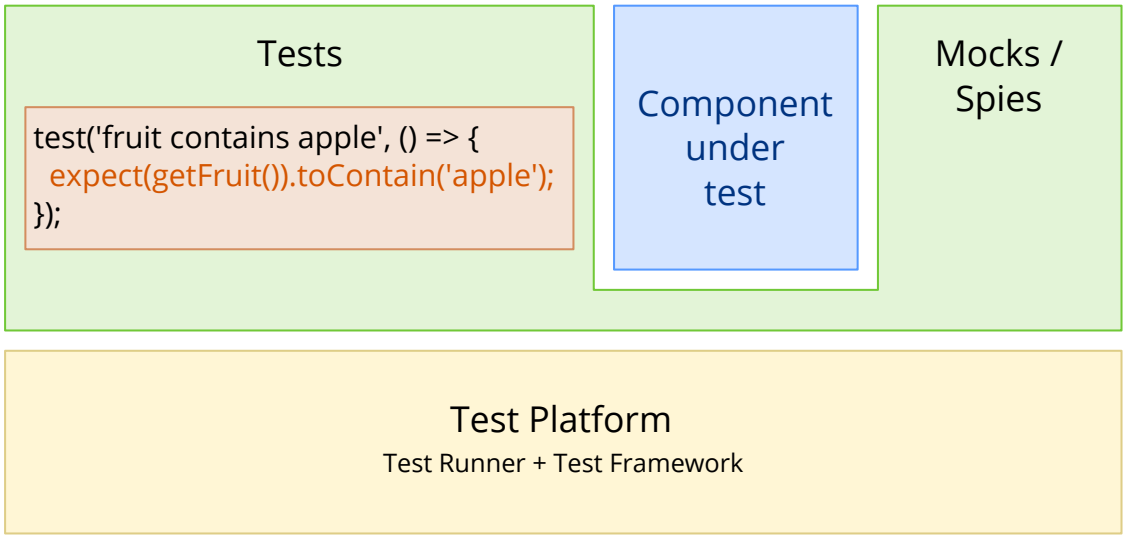
- This means that sometimes we will need to perform "white box" testing

Testing Principles

Guidelines

- Test the User-Visible Externals (see https://twitter.com/kentcdodds/status/974278185540964352)
- Abusing Snapshot Testing
    - Don’t rely completely on snapshot testing: have unit tests that work with snapshot tests as add-ons
- Don’t test internals! No state tests; no private method tests etc. brittle;

Testing Tool Overview



Introducing Jest

Jest:

- General-purpose testing platform
- Runs in node.js
- Includes:
  - Test framework (`describe()`, `it()`, `expect()` etc)
  - Assertions and matchers
  - Test runner

Writing Tests

- Name your tests as any of:
  - `MyComponent.test.js`
  - `MyComponent.spec.js`
  - `./__tests__/AnyNameForTheFileTest.js`

JSX

```
it('the best flavor is grapefruit', () => {
  expect(bestLaCroixFlavor()).toBe('grapefruit');
});
```

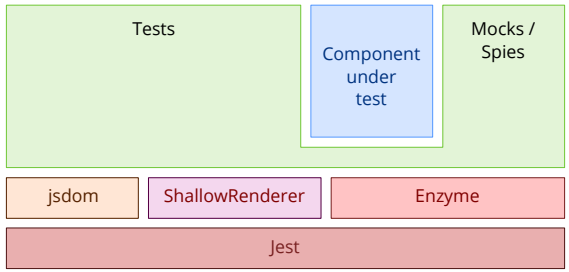
Running Tests

- To run:
  - `npm test` or `yarn test`
  - Starts jest in "watch" mode: re-runs tests whenever a source file changes

Enzyme

Enzyme:

- library that makes asserting and matching easier
- supports DOM rendering (via *jsdom*) as well as shallow rendering



DOM Testing with Enzyme

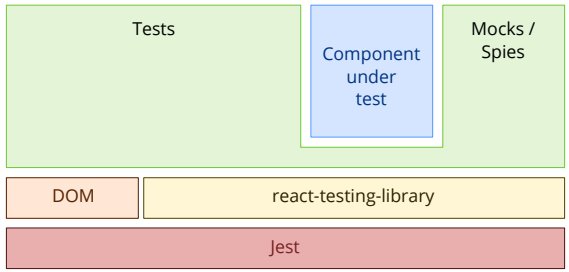
JSX

```jsx
it('should contain name', () => {
    const contactDom = mount(<Contact contact={person} />);
    contactDom.text().should.containEql('John Doe');  // TODO: use Jest
  });
```

react-testing-library

An alternate test library:

- use in place of Enzyme
- supports only full DOM rendering
- why?
  - *The more your tests resemble the way your software is used, the more confidence they can give you.* -- Kent C Dodds



Snapshot Testing

- on the first test, we record (serialize) the rendered output; then
- on subsequent tests, we compare the recorded output with the original

JSX

```
it("should render a contact's first and last name correctly", () => {
  const johnDoe = { firstName: 'John', lastName: 'Doe' };
  const tree = renderer
    .create(<Contact contact={johnDoe}>Instagram</Link>)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

- snapshot tests are assertions
  - recommend no more than one snapshot per test
- treat snapshots as code (see Jest's best practices)
  - commit and code review snapshots, so make them small!

Snapshot Testing -- Pros and Cons

Pros

- easy
  - no need to write assertions!
- quick to write
- show you that "something changed" (regression test)
- good for updating and testing legacy codebases

Cons

- they're a blunt tool
  - they can't highlight *why* something failed
- they don't convey the author's intention
  - we can't tell what the author wants the code to do!
  - also we can't use a TDD process (which requires tests to be developed first)
- they may provide a false sense of confidence in our code
  - we know our code hasn't changed, but is it doing the right thing?

Some Useful Development Tools

React Fundamentals

Some Useful Development Tools

- Examples:
  - codepen.io
  - codesandbox.io

The Good:

- Shareable!
- Embeddable
- Quick to set up
- Easy to use

The Bad:

- Typically not as feature rich or as fast as a local IDE
- Unavailable when you're offline
- Some company policies restrict code leaving the firm
  - Mandates against using cloud-based tools

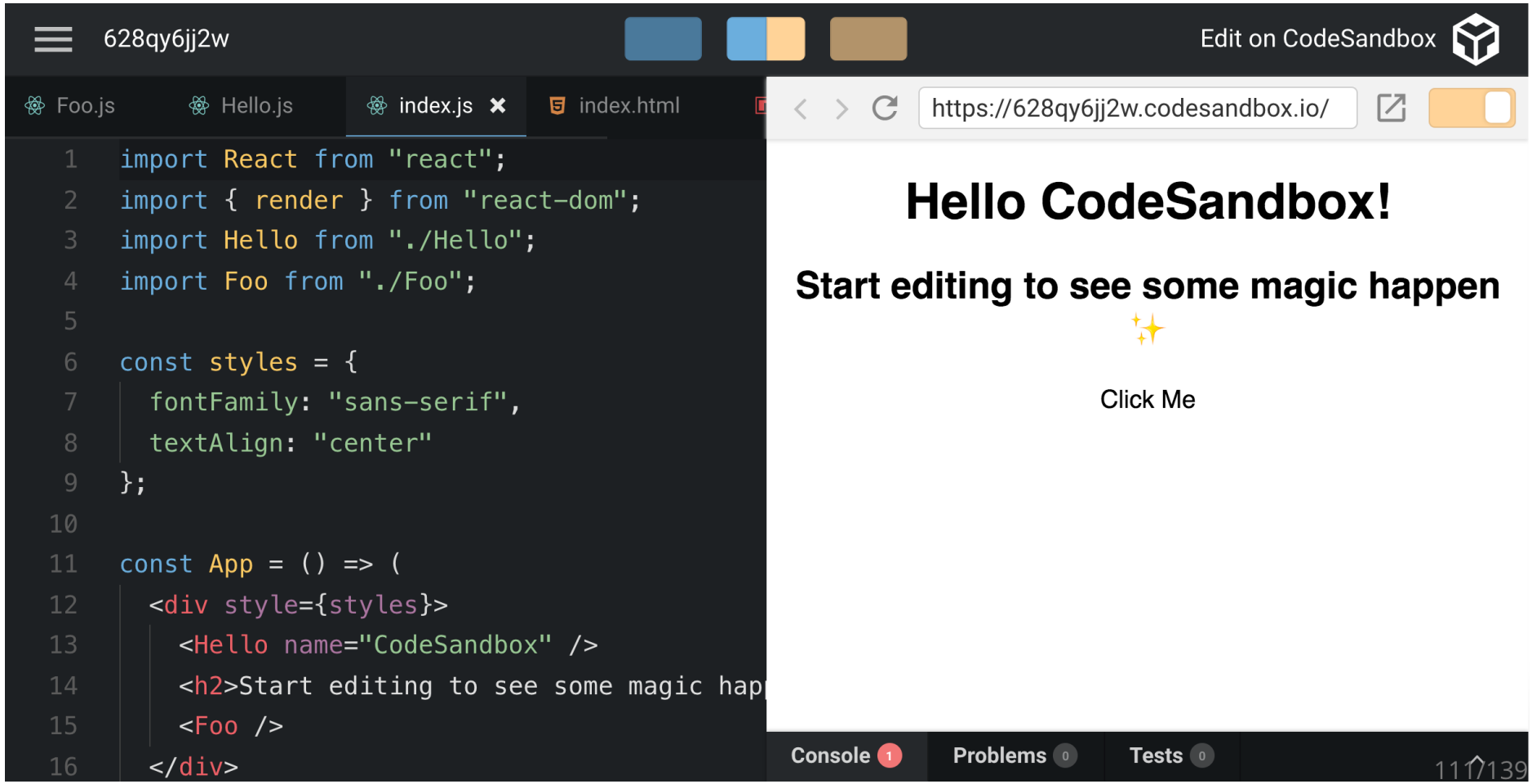
Codepen

Here's an example using codepen.io:

HTML CSS Babel Result EDIT ON CODEPEN

```
class SimpleButton extends React.Component {
```

Codesandbox



Local Development Tools

Useful for developing components in isolation:

- React Storybook
- React Styleguidist

These also offer:

- viewing different component states independently of your app
- deployment as a static app (no app server needed, just an HTTP server)
    - enables teams to collaborate and view components that have been developed

See the create-react-app documentation for more details

Exploring React Further

React Fundamentals

Exploring React Further

- Thinking in React
  - A useful blog post from Facebook
- Awesome React List
  - https://github.com/enaqx/awesome-react
- `immutable-js`
  - Immutable Persistent Collections
- React Native
  - Creating performant mobile (iOS/Android) apps with React
- Flux
  - Design Pattern for unidirectional flow
- Om
  - Functional ClojureScript layer on top of React

And many more...