

## SECTION 3: JSX. LIFECYCLES.

# JSX

- 🔔 JSX is a JavaScript syntax extension that looks similar to XML
- 🔔 XML has the benefit of balanced opening and closing tags. This helps make large trees easier to read than function calls or object literals.
- 🔔 After all JSX just makes code more readable and lets define virtualDOM in more convenient way
- 🔔 You don't have to use JSX with React. You can just use plain JS. However, we recommend using JSX because it is a concise and familiar syntax for defining tree structures with attributes.

## JSX

 You can use JSX to render HTML  
Tags:

```
var myDivElement = <div className="foo" />;  
ReactDOM.render(myDivElement,  
document.getElementById('example'));
```

 Or you can use JSX to render React  
Component

```
var MyComponent = React.createClass({/* ... */});  
var myElement = <MyComponent  
someProperty={true} />;  
ReactDOM.render(myElement,  
document.getElementById('example'));
```

## JSX TO JS

- 🔔 Since browser doesn't know how to interpret JSX it needs to be transformed to JS
- 🔔 Popular tools: babelify with react preset, webpack with react preloader (we will use)
- 🔔 JSX to JS transformations:

```
var Nav;  
// Input (JSX):  
var app = <Nav color="blue" />;  
// Output (JS):  
var app = React.createElement(Nav,  
{color:"blue"});
```

```
// Input (JSX):  
var person = <Person name={window.isLoggedIn  
? window.name : ""} />;  
// Output (JS):  
var person = React.createElement(  
  Person,  
  {name: window.isLoggedIn ? window.name : ""}  
);
```

# JSX EXPRESSIONS

- 🔔 With JSX you can create more semantic structures:  
Form, Form.Row,  
Form.Header:

```
var App = (
  <Form>
    <Form.Header />
    <Form.Row>
      <Form.Input/>
    </Form.Row>
  </Form>
);
```

🔔 Attributes: `<Person loggedIn={window.isLoggedIn ? true : false} />`

🔔 Boolean attr: `<Person loggedIn={true} />`    `<Person loggedIn={true} />`

🔔 Comments: `return /* child comment, put {} around */`

🔔 Styles: `<Person style={{color:black}} />`

🔔 Custom attr: `<div data-custom-attribute="foo" />`

🔔 Web Accessibility: `<div aria-hidden={true} />`

🔔 Logical: `let loggedIn = false;  
return (  
 {loggedIn || <authContainer/>}  
)`

🔔 Logical: `<div id={if (condition) { 'msg' }}>Hello World!</div>`

## JSX GOOD TO KNOW

- 🔔 Currently, in a component's render, you can only return one node!
- 🔔 if you have, say, a list of divs to return, you must wrap your components within a div, span or any other component.
- 🔔 Don't forget that JSX compiles into regular JS; returning two functions doesn't really make syntactic sense. Likewise, don't put more than one child in a ternary.
- 🔔 In JSX, `<MyComponent />` alone is valid while `<MyComponent>` isn't. All tags must be closed.

## LIFECYCLES

- 🔔 Render method in a React component needs to be a pure function
- 🔔 That means it needs to be stateless, it needs to not make any Ajax requests, etc. It should just receive state and props and then render a UI.
- 🔔 Though we can't do those things in the render method, they're still pretty critical for building a React app. So now the question, where should those things go?
- 🔔 To answer this question, we'll dive into React Life Cycle methods

# component LIFECycle

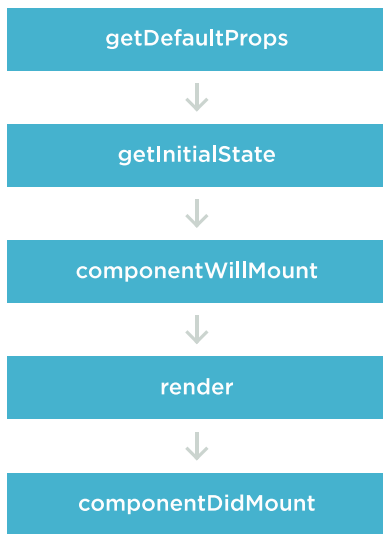
**Callbacks** that become possible to  
interact with Virtual DOM and  
**manipulate** the real DOM



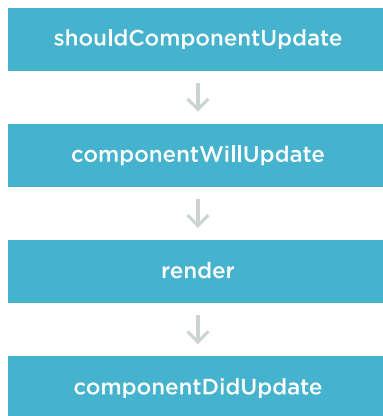
# COMPONENT LIFECYCLES

Whole bunch of methods:

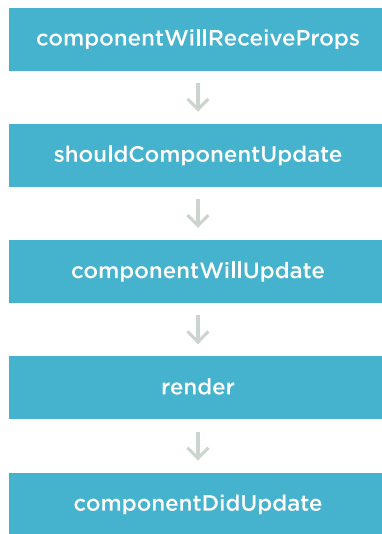
## First Render



## State Change



## Props Change



## Unmount



## MOUNTING: COMPONENTWILLMOUNT



This method is invoked once, both on the client and server, immediately before the initial rendering occurs.



If you call `setState` within this method, `render()` will see the updated state and will be executed only once despite the state change.

```
class ExampleComponent extends Component {  
  ...  
  componentWillMount() {  
    // Run code before component is initially rendered  
  }  
  ...  
}
```

## MOUNTING: COMPONENTDIDMOUNT

- 🔔 This method is invoked once, both on the client and server, immediately before the initial rendering occurs.
- 🔔 At this point in the lifecycle, you can access any refs to your children (e.g., to access the underlying DOM representation).
- 🔔 The `componentDidMount()` method of child components is invoked before that of parent components.

```
class ExampleComponent extends Component {  
  ...  
  componentDidMount() {  
    // Run code after component is initially rendered  
  }  
  ...  
}
```

## UPDATING: COMPONENTWILLRECEIVEPROPS

- 🔔 This method is invoked when a component is receiving new props. This method is not called for the initial render.
- 🔔 Use this as an opportunity to react to a prop transition before `render()` is called by updating the state using `this.setState()`. The old props can be accessed via `this.props`.
- 🔔 Calling `this.setState()` within this function will not trigger an additional render

```
class ExampleComponent extends Component {  
  ...  
  componentWillReceiveProps(nextProps) {  
    // Invoked when component is receiving new  
    props  
  }  
  ...  
}
```

## UPDATING: SHOULDCOMPONENTUPDATE

- 🔔 This method is invoked before rendering when new props or state are being received.
- 🔔 This method is not called for the initial render or when `forceUpdate` is used.
- 🔔 Use this as an opportunity to return `false` when you're certain that the transition to the new props and state will not require a component update.

```
class ExampleComponent extends Component {  
  ...  
  shouldComponentUpdate(nextProps, nextState) {  
    // Invoked before rendering as new props or  
    state are being received.  
  }  
  ...  
}
```

## UPDATING: COMPONENTWILLUPDATE

- 🔔 This method is invoked immediately before rendering when new props or state are being received. This method is not called for the initial render
- 🔔 You cannot use `this.setState()` in this method. If you need to update state in response to a prop change, use `componentWillReceiveProps` instead.
- 🔔 Use this as an opportunity to perform preparation before an update occurs.

```
class ExampleComponent extends Component {  
  ...  
  componentWillUpdate(nextProps, nextState) {  
    // Invoked immediately before rendering as new props  
    // or state are being received.  
  }  
  ...  
}
```

## UPDATING: COMPONENTDIDUPDATE

- 🔔 This method is invoked immediately after the component's updates are flushed to the DOM. This method is not called for the initial render.
- 🔔 Use this as an opportunity to operate on the DOM when the component has been updated.

```
class ExampleComponent extends Component {  
  ...  
  componentDidUpdate(prevProps, prevState) {  
    // Invoked immediately after updates are flushed to the  
    DOM  
  }  
  ...  
}
```

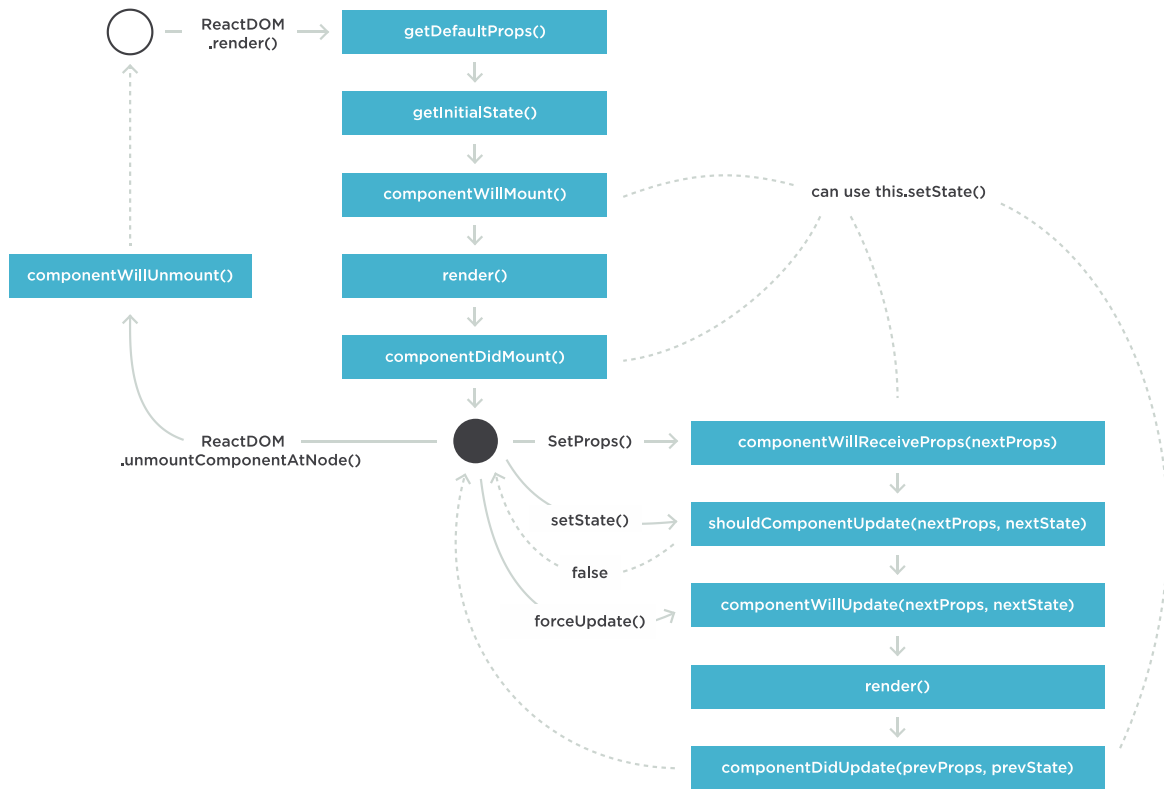
## UNMOUNTING: COMPONENTWILLUNMOUNT

- 🔔 This method is invoked immediately before a component is unmounted from the DOM.
- 🔔 Perform any necessary cleanup in this method, such as invalidating timers or cleaning up any DOM elements that were created in `componentDidMount`.

```
class ExampleComponent extends Component {  
  ...  
  componentWillUnmount() {  
    // Run code before component is dismounted from DOM  
  }  
  ...  
}
```



# LIFECYCLES



## REACT: EVENTS


 Lets consider two options:

 You want to add listeners do something in the dom, which is not under react.js control.

The way to do it:

```
//Some component subscribes for window resize
{
  componentDidMount: function() {
    window addEventListener('resize', this.handleResize);
  },
  componentWillUnmount: function() {
    window removeEventListener('resize', this.handleResize);
  },
  render: function() {
    return <div>Current window width: {this.state.windowWidth}</div>;
  }
}
```

## REACT: EVENTS

 Another option is event handler you manually pass to react elements. It works and looks quite close to what you actually do in pure JavaScript with HTML, but here you do it with JSX. Example:

```
var BannerAd = React.createClass({
  onBannerClick: function(evt) {
    // click happened
  },

  render: function() {
    // Render the div with an onClick prop (value is a function)
    return <div onClick={this.onBannerClick}>Click Me!</div>;
  }
});
```




# REACT: EVENTS

 List of available events

1. onCopy onCut onPaste
2. onCompositionEnd onCompositionStart onCompositionUpdate
3. onKeyDown onKeyPress onKeyUp
4. onFocus onBlur
5. onChange onInput onSubmit
6. onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp
7. onSelect
8. onTouchCancel onTouchEnd onTouchMove onTouchStart
9. onScroll
10. onWheel
11. onLoad onError
12. onAnimationStart onAnimationEnd onAnimationIteration
13. onTransitionEnd
14. onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata

## REACT: EVENTS

### SyntheticEvent

-  Your event handlers will be passed instances of SyntheticEvent, a cross-browser wrapper around the browser's native event
-  It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers
-  If you find that you need the underlying browser event for some reason, simply use the `nativeEvent` attribute to get it.