React JS

Created by :

Sangeeta Joshi

Agenda

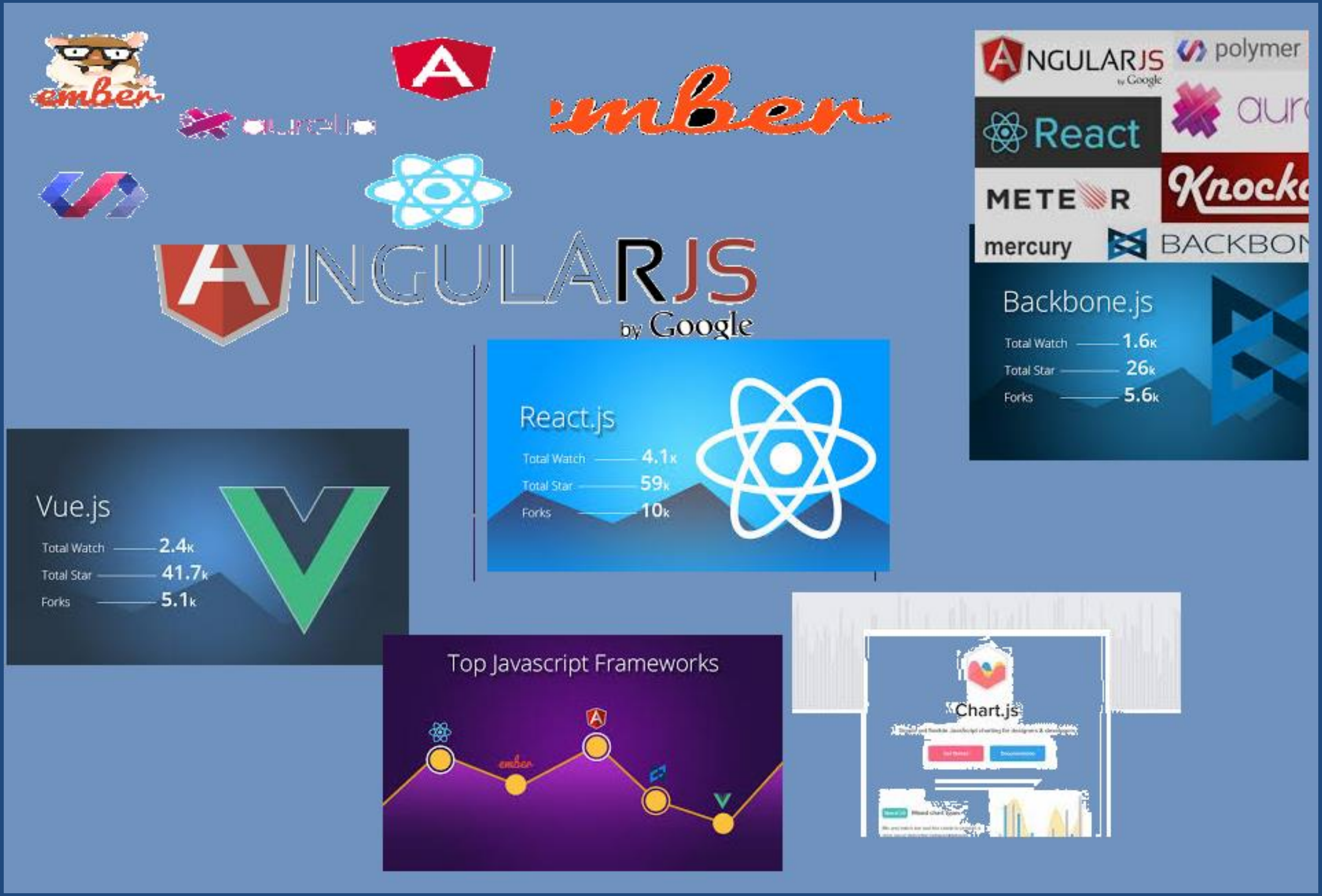
- React Overview & Component Architecture
- Starting with React
- Hello World
- Introduction to JSX
- Elements
- Components & Props
- Functional Components
- Components created using class
- State
- Component Life Cycle

What is React?

A JS Library for building User Interfaces

Target : Great User Experience

JavaScript Frameworks



Architectures

MVC

MVVM

MVW

What is React

- React is not a framework
- It’s a J S Library
- It’s a view rendering engine or a component model.
- React focuses on view layer i.e rendering

Focus

- Example : companies data ..top 3.. Etc

- React focuses on view layer i.e rendering

Components

Components : Declarative approach

- Reusable APIs
- Encapsulate Behavior (CSS,JS etc)
- Hides implementation details
- Something like pick a comp & drop it in page where you need it

Declarative Components

Components :

- More Template building
- Invokes some function
- No Explicit data binding

(As in angular : changes in data causes view updates & vice a versa)

Starting with React

React is the entry point to the React library.

- If you load React from a <script> tag, these top-level APIs are available on the React global.
- If you use ES6 with npm, you can write import React from 'react'.
- If you use ES5 with npm, you can write

  var React = require('react').

Diving in

Libraries to import:

React

ReactDOM

ReactDOM :

- glue between React and the DOM.
- Often, you will only use it for one single thing: mounting with *ReactDOM.render().*
- Another useful feature of ReactDOM is *ReactDOM.findDOMNode()* which you can use to gain direct access to a DOM element.
- If your app is "isomorphic", you would also use *ReactDOM.renderToString()* in your back-end code.

React:

for everything else, there's React. You use React to define and create your elements, for lifecycle hooks, etc

Elements

- smallest building blocks of React Apps
- describes what you want to see on the screen

Example:

```
    const element=<h1> Hello World</h1>;
```

Rendering element:

```
ReactDOM.render(element,
document.getElementIdBy(‘root’));
```

Hello World

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

http://codepen.io/sangeetaj/pen/KmNEax

Components & Props

- Components : lets to split UI into independent, reusable pieces where each piece can be considered in isolation.
- Conceptually, components are like JavaScript functions.
- They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Components

Always start component names with a Capital letter.

For example,

- <div /> represents a DOM tag,
- <Welcome /> represents a component and requires Welcome to be in scope.

JSX

```
const element = <h1>Hello, world!</h1>
```

- This funny tag syntax is *neither a string nor HTML*
- It is JSX
- It is a syntax extension to JavaScript.
- Use it with React to describe : ***what the UI should look like***.
- JSX may remind you of a template language, but it comes with the full power of JavaScript
- JSX produces React "elements".

JSX

- Embedding Expressions in JSX : { }

We can embed any JavaScript expression in JSX by wrapping it in curly braces.

For example :

- 2 + 2
- user.firstName
- formatName(user)

JSX is an Expression

- After compilation, JSX expressions become regular ***JavaScript objects***.
    - Can use JSX inside of if statements and for loops,
    - assign it to variables,
    - accept it as arguments,
    - return it from functions

JSX is an Expression

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

- Specifying Attributes with JSX

```
const element = <div tabIndex="0"></div>;
```

- use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

JSX

- Fundamentally, JSX just provides syntactic sugar for :

React.createElement(component, props, ...children) function.

JSX

- The JSX code:

```
 <MyButton
  color="blue" shadowSize={2}> Click Me
</MyButton>
```

compiles into:

```
React.createElement( MyButton,
 {color: 'blue', shadowSize: 2}, 'Click Me' )
```

Using Dot Notation for JSX Type

if MyComponents.DatePicker is a component, you can use it directly from JSX with:

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

Choosing the Type at Runtime

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Wrong! JSX type can't be an expression.
  return <components[props.storyType] story={props.story} />;
}
```

You cannot use a general expression as the React element type

Choosing the Type at Runtime

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

we will assign the type to a capitalized variable first:

Components & Props

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Conceptually, components are like JavaScript functions.
- They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Components

Always start component names with a Capital letter.

For example,

- <div /> represents a DOM tag,
- <Welcome /> represents a component and requires Welcome to be in scope.

Functional Components

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

- It’s a valid functional component as
  - it accepts a single "props" object with data
  - returns a React element.
- We call such components "functional" because they are literally JavaScript functions.

Demo - http://codepen.io/sangeetaj/pen/OmWgPW

props

- When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "*props*".

- const element = <Welcome name="Sara" />;

Components & Props

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

Components & Props

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Conceptually, components are like JavaScript functions.
- They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Props are Read Only

- Whether a component is a function or a class, it must never modify its own props.
- React is pretty flexible but it has a single strict rule:
- **All React components must act like pure functions with respect to their props**

Composing Components

- Components can refer to other components in their output.
- This lets us use the same component abstraction for any level of detail.

https://codepen.io/sangeetaj/pen/QvvmOg?editors=1111

Extracting Components

- split components into smaller components
- having a palette of reusable components pays off in larger apps.
- A good rule of thumb : if a part of UI is used several times (Button, Panel etc),

  or

  is complex enough on its own (App, FeedStory, Comment),

it is a good candidate to be a reusable component.

Class Components

- We can use an ES6 class to define component:

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

- Classes have some additional features

http://codepen.io/sangeetaj/pen/mmmxQP

State

State:

- application UIs are dynamic and change over time.
- State allows React components to change their output over time in response to :

  - user actions
  - network responses
  - and anything else

without violating “Props” rule.

State

components defined as classes have some additional features.

State:

- State is similar to props, but it is private and fully controlled by the component
- Local state : a feature available only to classes.

Using State Correctly

- Do Not Modify State Directly:

```
// Wrong
this.state.comment = 'Hello';
```

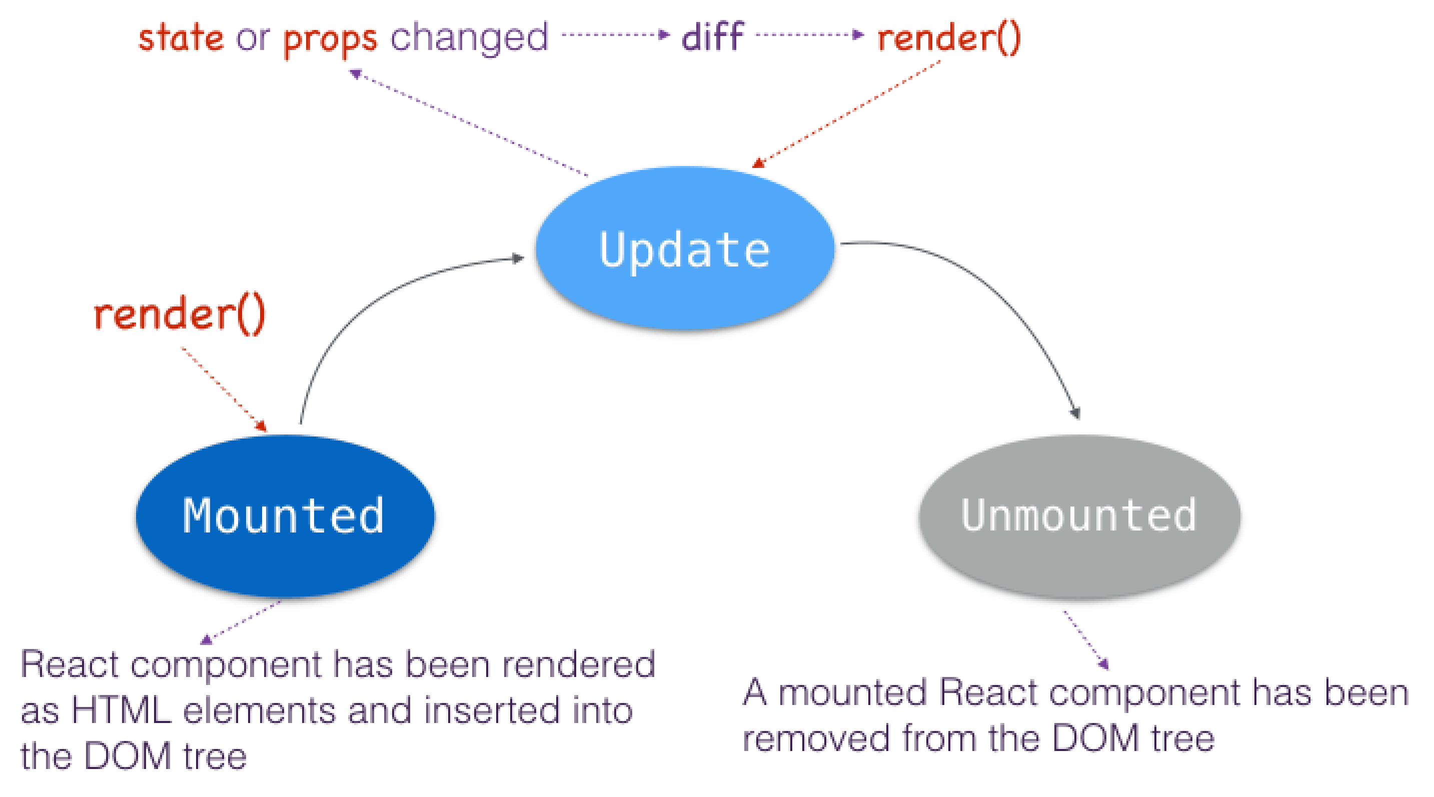
- Instead, use setState():

```
// Correct
this.setState({comment: 'Hello'});
```

- ***The only place where you can assign this.state is the constructor.***

Life Cycle

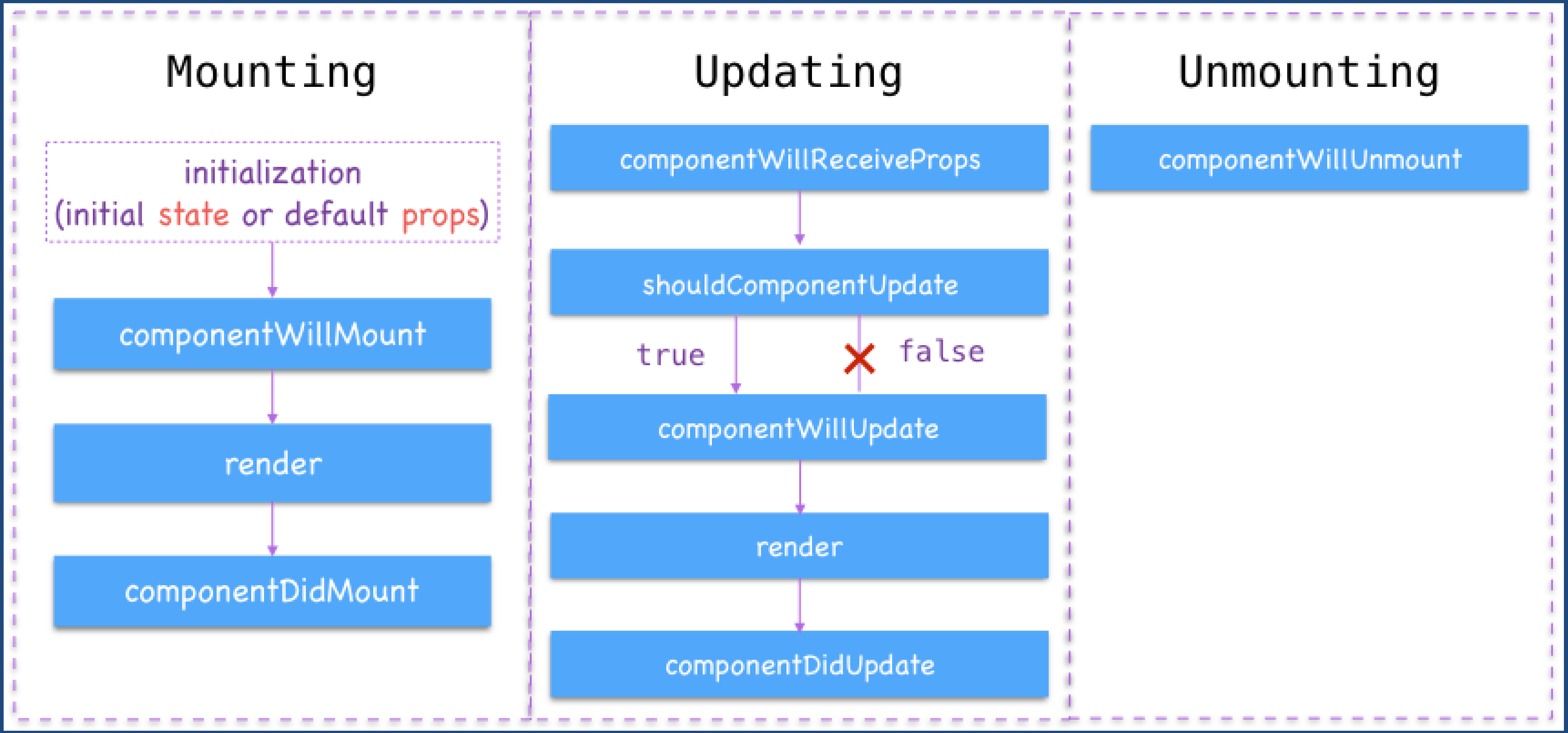
Three Stages : mounted ,update ,unmounted



React component has been rendered as HTML elements and inserted into the DOM tree

A mounted React component has been removed from the DOM tree

Life Cycle


Mounting
initialization
(initial state or default props)
componentWillMount
render
componentDidMount
Updating
componentWillReceiveProps
shouldComponentUpdate
true
false
componentWillUpdate
render
componentDidUpdate
Unmounting
componentWillUnmount

Life Cycle Hooks:Mounting:

Initializing State - ES6 : constructor

- ES5: *getInitialState()*

- gets called only once before component is mounted
- Initialization should typically only be done in a top level component, which acts as a role of **controller view** in your page.

Life Cycle Hooks

:Mounting:

Default Props:

The getDefaultProps() method is called only once before any instance of the component is created. So you should avoid using this.props inside getDefaultProps() method.

Life Cycle Hooks::Mounting

componentWillMount()

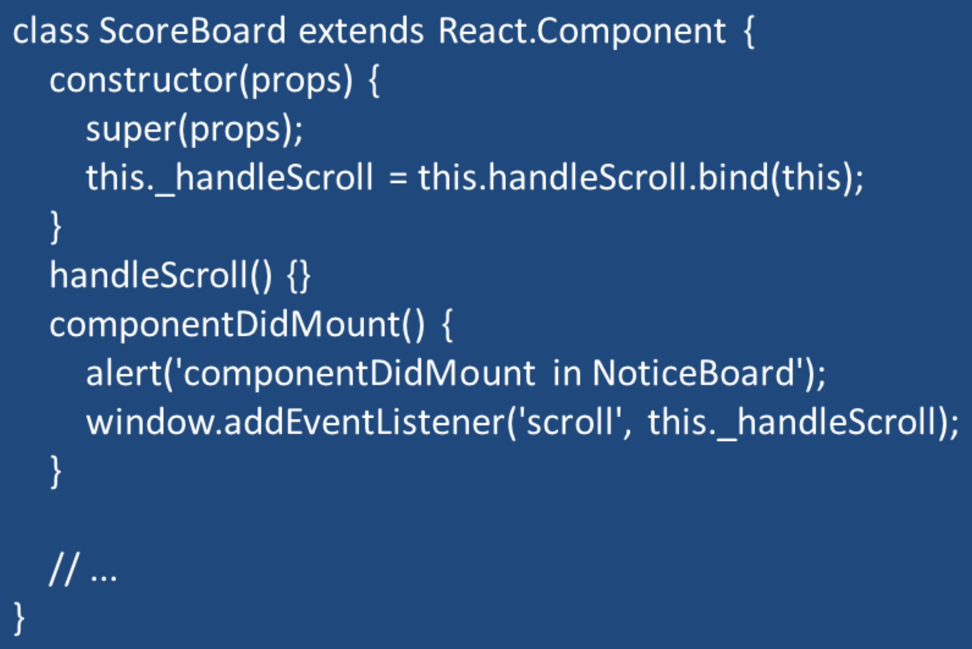
- The method is invoked only once before initial rendering.
- It is also a good place to set initial state value inside this method

```
     class SinglePlayer extends React.Component {
       componentWillMount() {
       this.setState({
             isPassed: this.props.score >= 60
        });

        alert('componentWillMount => ' + this.props.name);
         console.log('componentWillMount => ' + this.props.name);
      }

          // ...
}
```

Life Cycle Hooks::Mounting

componentDidMount()

- This lifecycle method will be invoked after rendering.
- It is the right place to access DOM of the component.



```
class ScoreBoard extends React.Component {
  constructor(props) {
    super(props);
    this._handleScroll = this.handleScroll.bind(this);
  }
  handleScroll() {}
  componentDidMount() {
    alert('componentDidMount in NoticeBoard');
    window.addEventListener('scroll', this._handleScroll);
  }

  // ...
}
```

Life Cycle Hooks::Updating

- componentWillReceiveProps()

  This method will be invoked when a component is receiving new props. It won't be called for the initial rendering.

- shouldComponentUpdate()

  will be invoked before rendering when new props or state are being received. This method won't be called on initial rendering.

  an opportunity to prevent the unnecessary rerendering considering performance. Just let shouldComponentUpdate() return false, then the render() method of the component will be completely skipped until the next props or state change.

Life Cycle Hooks::Updating

- componentWillUpdate()

  Invoked just before render(), but after shouldComponentUpdate() (of course, return a true). Not called for the initial rendering.

```
 class SinglePlayer extends React.Component {
componentWillUpdate(nextProps, nextState) {
alert('componentWillUpdate => ' + this.props.name);
console.log('componentWillUpdate => ' + this.props.name);
 }
   }
```

Life Cycle Hooks::Updating

- componentDidUpdate()
  - Invoked immediately after the component's updates are flushed to the DOM. This method is not called for the initial rendering.
  - You can perform DOM operations after an update inside function.

```
class SinglePlayer extends React.Component {
                componentWillUpdate(nextProps, nextState) {
                 alert('componentWillUpdate => ' + this.props.name);
                 console.log('componentWillUpdate => ' + this.props.name);
                  }
                    }
```

Life Cycle Hooks::Unmounting

- componentWillUnmount()
  - Invoked immediately before a component is unmounted or removed from the DOM.
  - Use this as an opportunity to perform cleanup operations. For example, unbind event listeners here to avoid memory leaking.

```
 class ScoreBoard extends React.Component {
      componentWillUnmount() {
        window.removeEventListener('scroll', this._handleScroll);
     }
}
```

Rendering

Two important things:

1. How initially things are rendered
2. How updates happen

A. Initial Rendering

Step 1 : Initial Rendering

(Unlike to other frameworks who work on creating DOM & wiring up events)

- render(){….} (just one function)

- a) Describe how your component looks at any point in time
- b) Does not return a string but returns:

"Representation of your view"

It means we generate markups as an Element & inject it into document

A. Initial Rendering

Step 1 : Initial Rendering ….continued

Two Pass Rendering :

1. Generate Mark up & inject it into DOM
2. Attach Events

Because of this separation, rendering can happen on Server or Client

B. Reconcile

~~Update~~ Reconcile

If data changes updates view

call render () ………get a representation

if something happens,again

call render () ……..get another representation

Compare two representations ,compute minimum differences and do minimum updates