Front End Notes

REACT JS Basics

By Sandesh Paudel

Table of Contents:

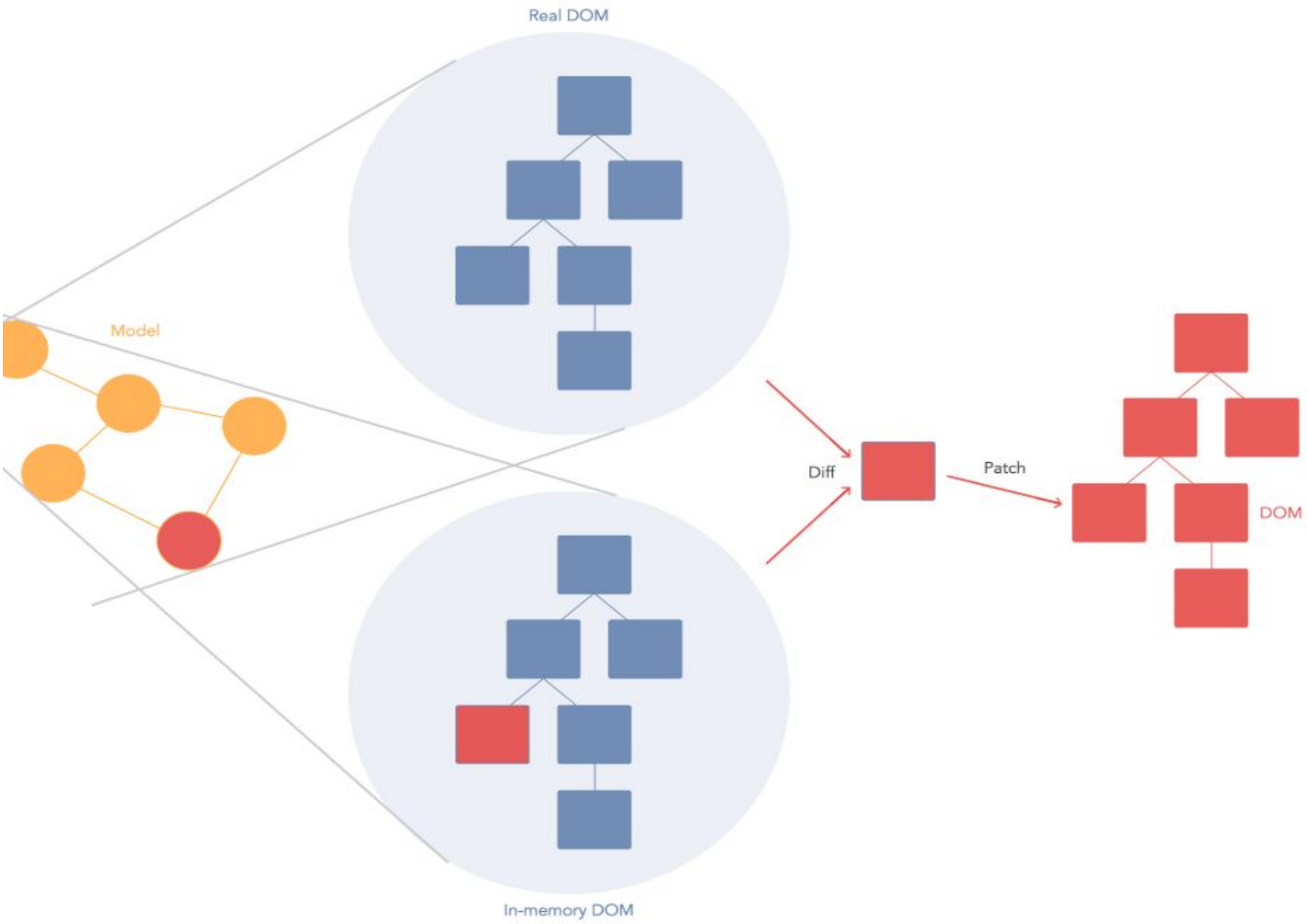
Why React?

- DOM manipulation is the heart of the problem in modern web apps. It is a bottleneck and most JS frameworks update the DOM much more than they have to!
-**Virtual DOM:** "virtual", representation of a UI **is kept in memory** and **synced with the "real" DOM** by a library such as ReactDOM. This process is called **reconciliation.**

In summary, here's what happens when you try to update the DOM in React:

1. The entire virtual DOM gets updated.
2. The virtual DOM gets compared to what it looked like before you updated it. React figures out which objects have changed.
3. The changed objects, and the changed objects only, get updated on the *real* DOM.
4. Changes on the real DOM cause the screen to change.

- React abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.



The JSX Syntax

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

- The right hand syntax is **neither a string nor HTML**. It is called **JSX,** a syntax extension to JS and it comes with the full power of JS.
- Something like HTML + JS. “template language inside JS”
- Produces React Elements

❖ **JS expressions inside JSX {}**

```
const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;

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

```
//JS Function
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez',
};

//Can display a retrun statement from JS function like so
const element = <h1>Hello, {formatName(user)}!</h1>;

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

❖ **JSX inside Javascript functions and control flow**

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

❖ **Attributes in JSX**

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

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

❖ **Safe against Injection Attacks**

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

❖ **Babel**: next generation JS compiler that recognizes JSX syntax.

❖ **Babel Magic:** JSX = React elements. These two are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

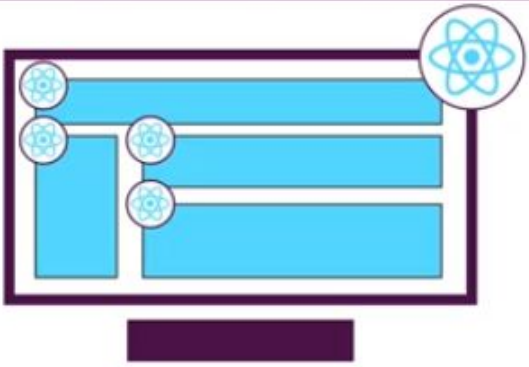
```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

React and SPA's (Single Page Apps)

Two Kinds of Applications

Single Page Applications

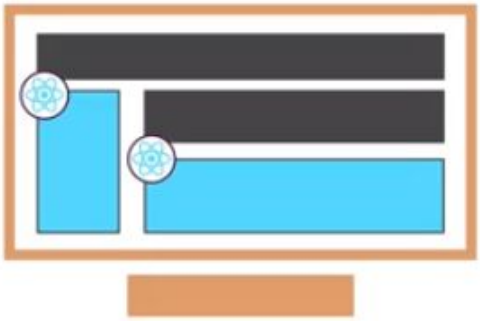
Only ONE HTML Page, Content is (re)rendered on Client



Typically only ONE ReactDOM.render() call

Multi Page Applications

Multiple HTML Pages, Content is rendered on Server



One ReactDOM.render() call per "widget"

The Next Generation JS

❖ **The Let and Const:** the next generation JS variables, a better alternative to var. **Let** is just a var that may change. **Const** are constants that do not change.

❖ **Arrow Functions**: [eliminates issue with **this** keyword] [functions are consts]
Vanilla js (left), Arrow Function (right)

```
function myFnc() {
  ...
}
```

```
const myFnc = () => {
  ...
}
```

```
function printMyName(name) {
  console.log(name);
}

printMyName();
```

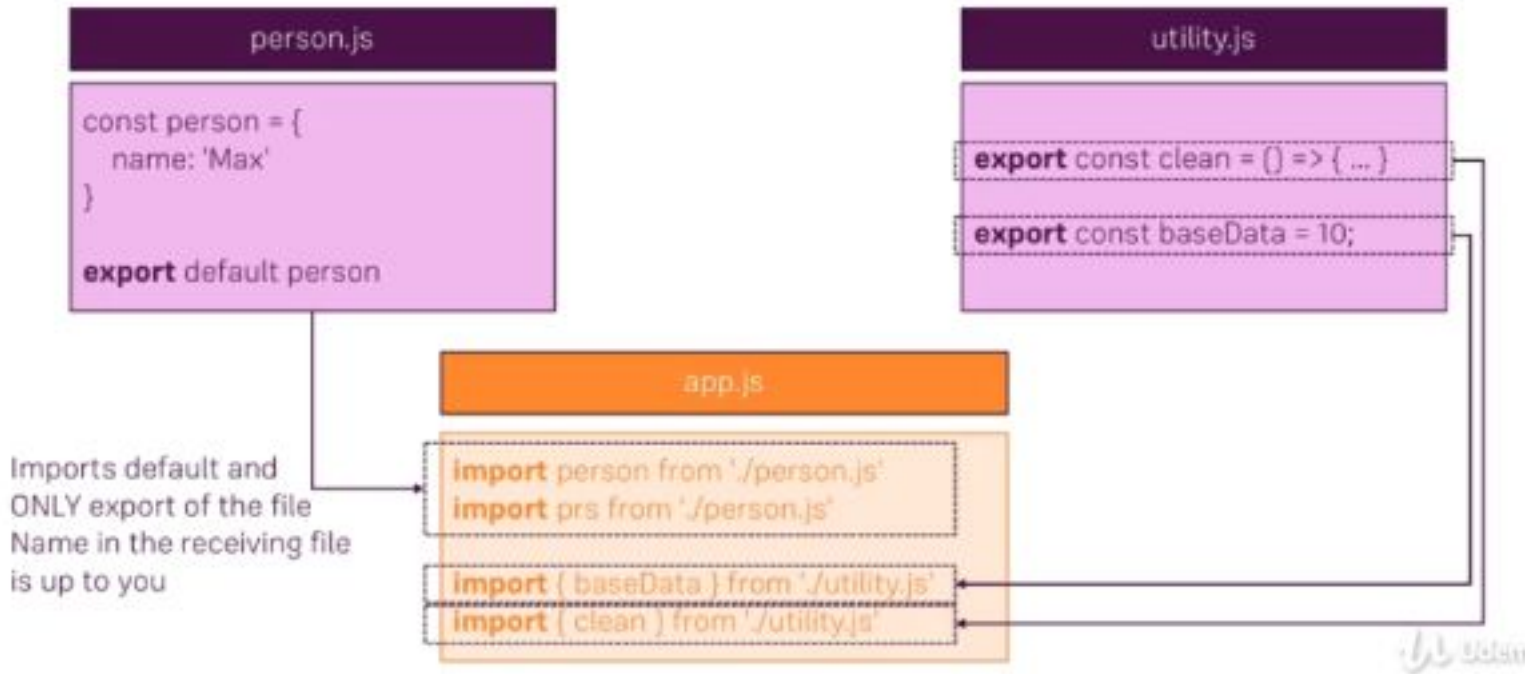
```
const printMyName = (name) => {
  console.log(name);
}

printMyName('Max');
```

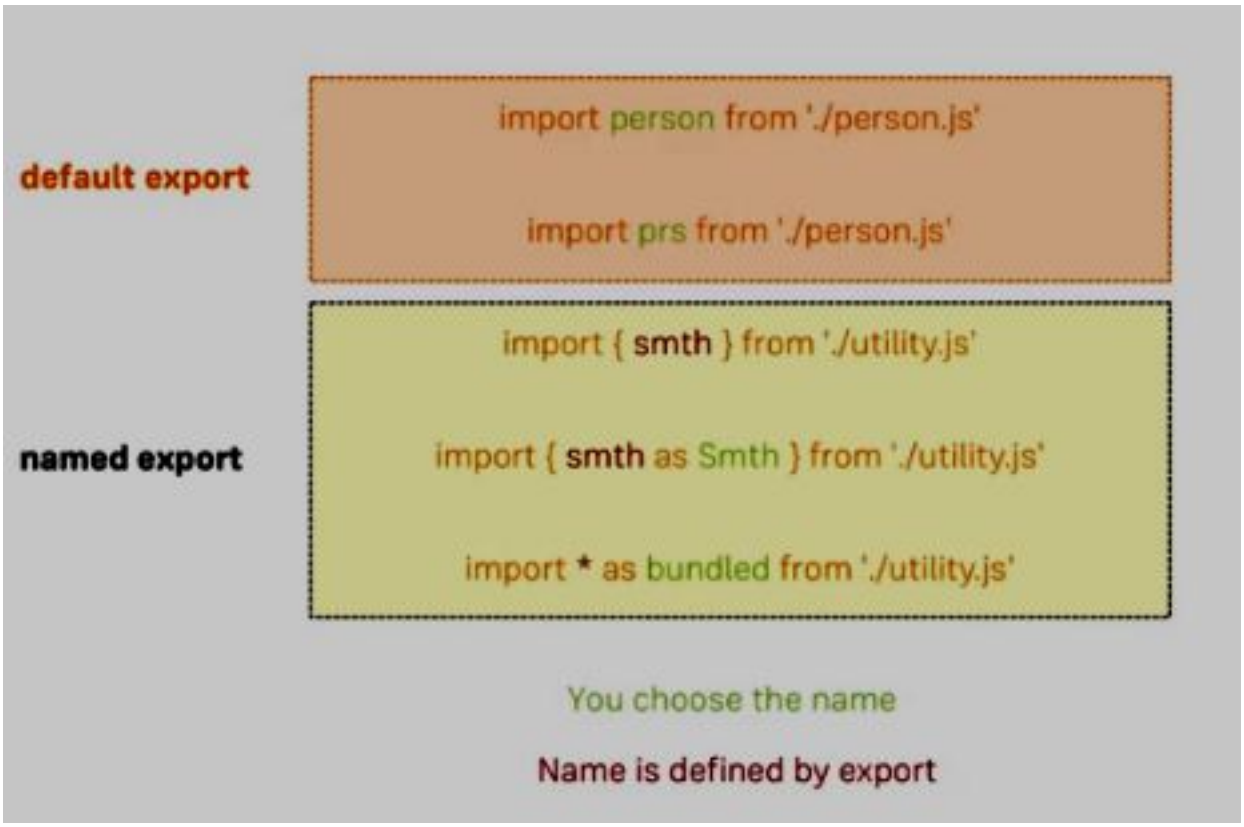
> Shorthand Return with Arrow Functions:

```
const multiply = (number) => {
  return number * 2;
}

console.log(multiply(2));
```

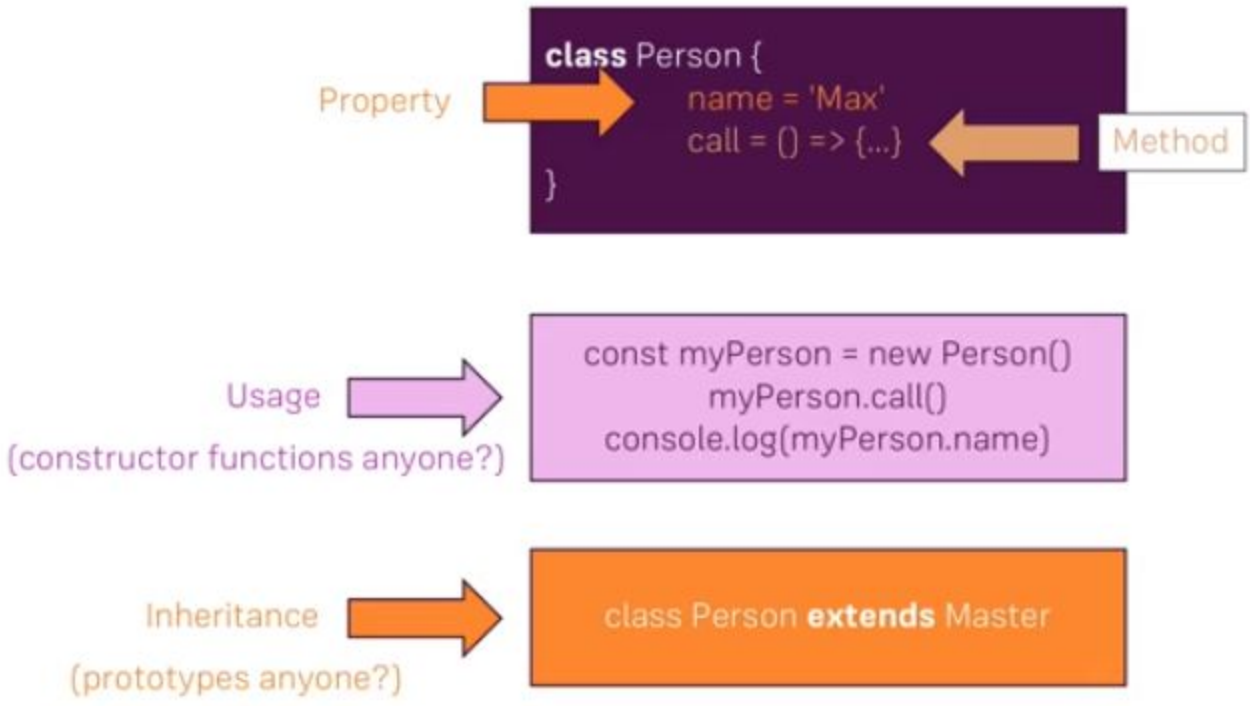
```
const multiply = (number) => number * 2;

console.log(multiply(2));
```

❖ **Exports and Imports (Modules)**

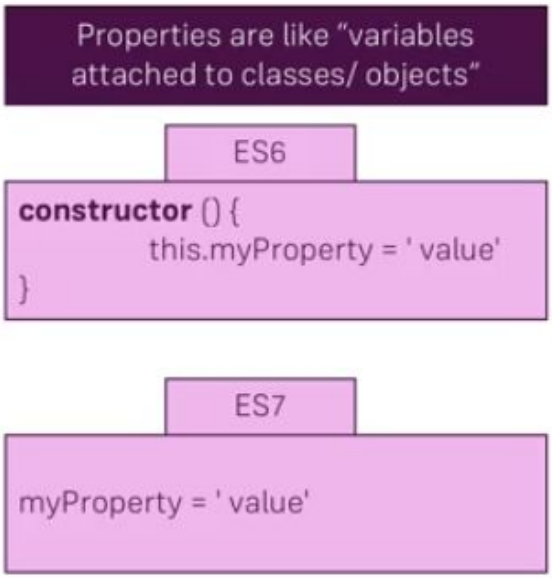


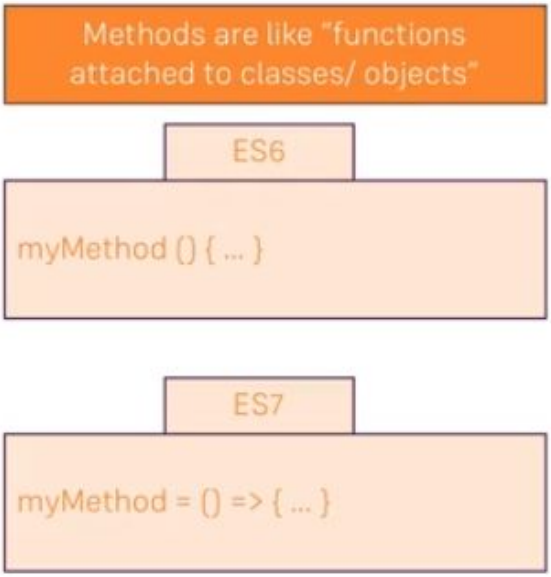

default export
import person from './person.js'
import prs from './person.js'
named export
import { smth } from './utility.js'
import { smth as Smth } from './utility.js'
import * as bundled from './utility.js'
You choose the name
Name is defined by export

❖ **Classes:**


Property
class Person {
name = 'Max'
call = () => {...}
}
Method
Usage
(constructor functions anyone?)
const myPerson = new Person()
myPerson.call()
console.log(myPerson.name)
Inheritance
(prototypes anyone?)
class Person extends Master

❖ Classes, Properties and Methods





Arrow Function: "Think of method as a property which stores a function as a value"

```
class Human {
  constructor() {
    this.gender = 'male';
  }

  printGender() {
    console.log(this.gender);
  }
}

class Person extends Human {
  constructor() {
    super();
    this.name = 'Max';
    this.gender = 'female';
  }

  printMyName() {
    console.log(this.name);
  }
}

const person = new Person();
person.printMyName();
person.printGender();
```

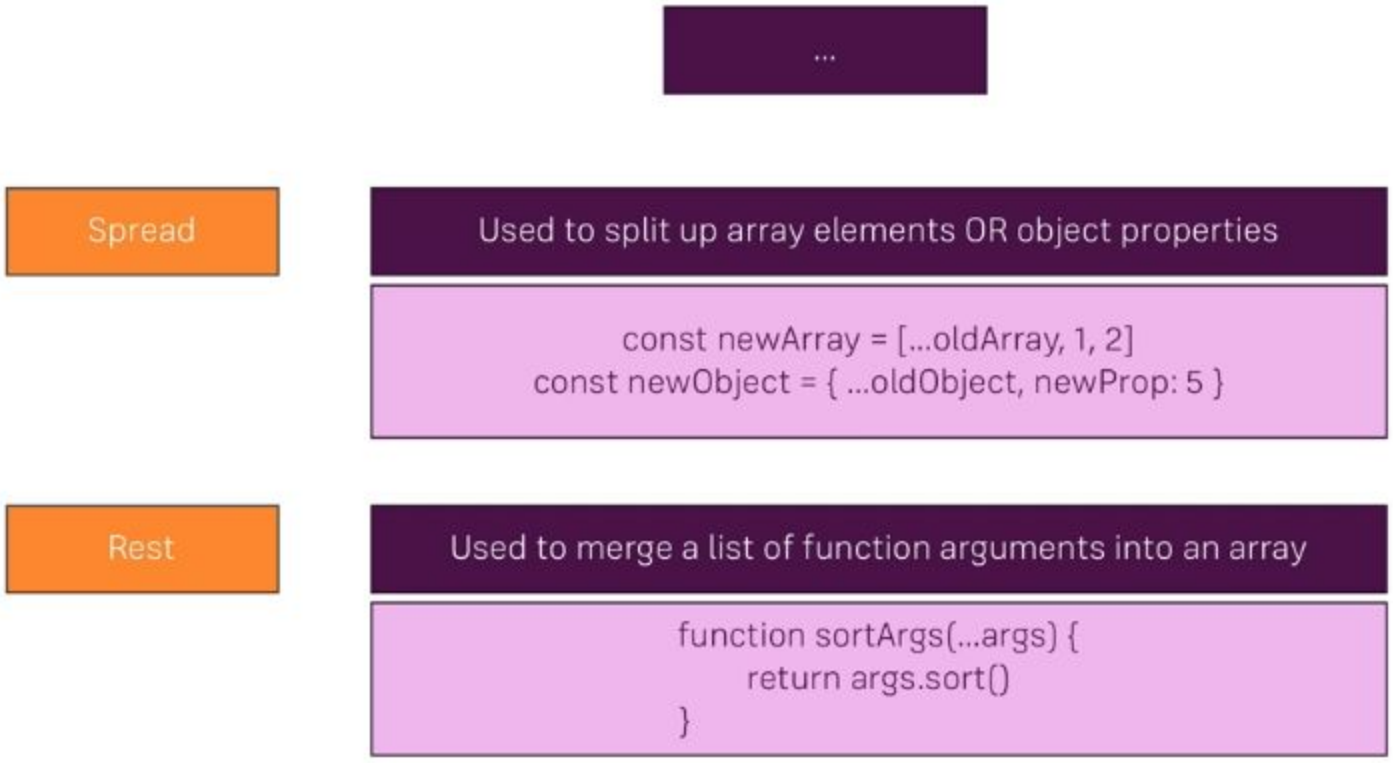
Some Notes:

- The person class inherits elements from the human class.
- The **super** declaration executes the constructor of its parent.
- Instantiation of class using **new** obj.

Next-Gen Transformations applied to left:

```
class Human {
  gender = 'male';

  printGender = () => {
    console.log(this.gender);
  }
}

class Person extends Human {
    name = 'Max';
    gender = 'female';

  printMyName = () => {
    console.log(this.name);
  }
}

const person = new Person();
person.printMyName();
person.printGender();
```

❖ **Spread and Rest Operators:** [const is used for arrays too]



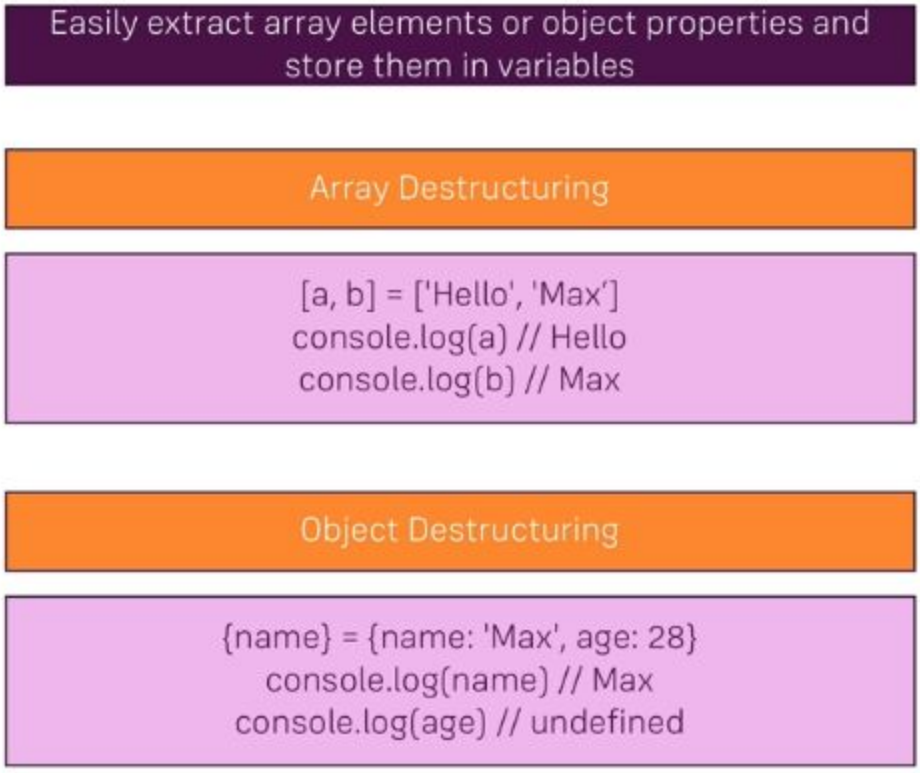
Rest Example:

```
const filter = (...args) => {
  return args.filter(el => el === 1);
}

console.log(filter(1, 2, 3));
```

Aside on Filter: the test is predicated by boolean true/false outcomes

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

❖ **Destructuring: pull out single array elements or object props** and **store them in variables**



Array Destruct Example:

```
const numbers = [1, 2, 3];
[num1, , num3] = numbers;
console.log(num1, num3);
```

Object Destruct Example:

```
var rect = { x: 0, y: 10, width: 15, height: 20 };


// Destructuring assignment
var {x, y, width, height} = rect;
console.log(x, y, width, height); // 0,10,15,20


rect.x = 10;
({x, y, width, height} = rect); // assign to existing variables using outer parentheses
console.log(x, y, width, height); // 10,10,15,20
```

❖ =

Reference and Primitive Types

❖ **Primitive Types: numbers, strings, booleans**

❖ **Reference Types:** objects and arrays are **reference** types. They are by default, "copied" by copying pointers to them in the memory. So it is not actual deep copy. We can do deep copy by using spread, however.

❖ **Deep copy of properties with spread operator:**

```
const person = {
  name: 'Max'
};

const secondPerson = {
  ...person
};

person.name = 'Manu';

console.log(secondPerson);
```

Array Functions and Useful JS Functions

- Executed on each elements, compiles an array for result

```
const numbers = [1, 2, 3];

const doubleNumArray = numbers.map((num) => {
  return num * 2;
});

console.log(numbers);
console.log(doubleNumArray);
```

```
[1, 2, 3]
[2, 4, 6]
>
```

> filter(): create a new array with elements that pass a funct test.

> map()

```
1 var array1 = [1, 4, 9, 16];
2
3 // pass a function to map
4 const map1 = array1.map(x => x * 2);
5
6 console.log(map1);
7 // expected output: Array [2, 8, 18, 32]
8
```

> **find():** returns the value of the first element in the array that satisfies test

```
var array1 = [5, 12, 8, 130, 44];

var found = array1.find(function(element) {
  return element > 10;
});

console.log(found);
// expected output: 12

```

> **findIndex():** returns the index of the first element that satisfies the test

```
var array1 = [5, 12, 8, 130, 44];

function findFirstLargeNumber(element) {
  return element > 13;
}

console.log(array1.findIndex(findFirstLargeNumber));
// expected output: 3

```

> **reduce():** applies a function against an accumulator and each element in the array to reduce it to a single value

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15

```

> **concat():** merge 2 or more arrays

```
var array1 = ['a', 'b', 'c'];
var array2 = ['d', 'e', 'f'];

console.log(array1.concat(array2));
// expected output: Array ["a", "b", "c", "d", "e", "f"]

```

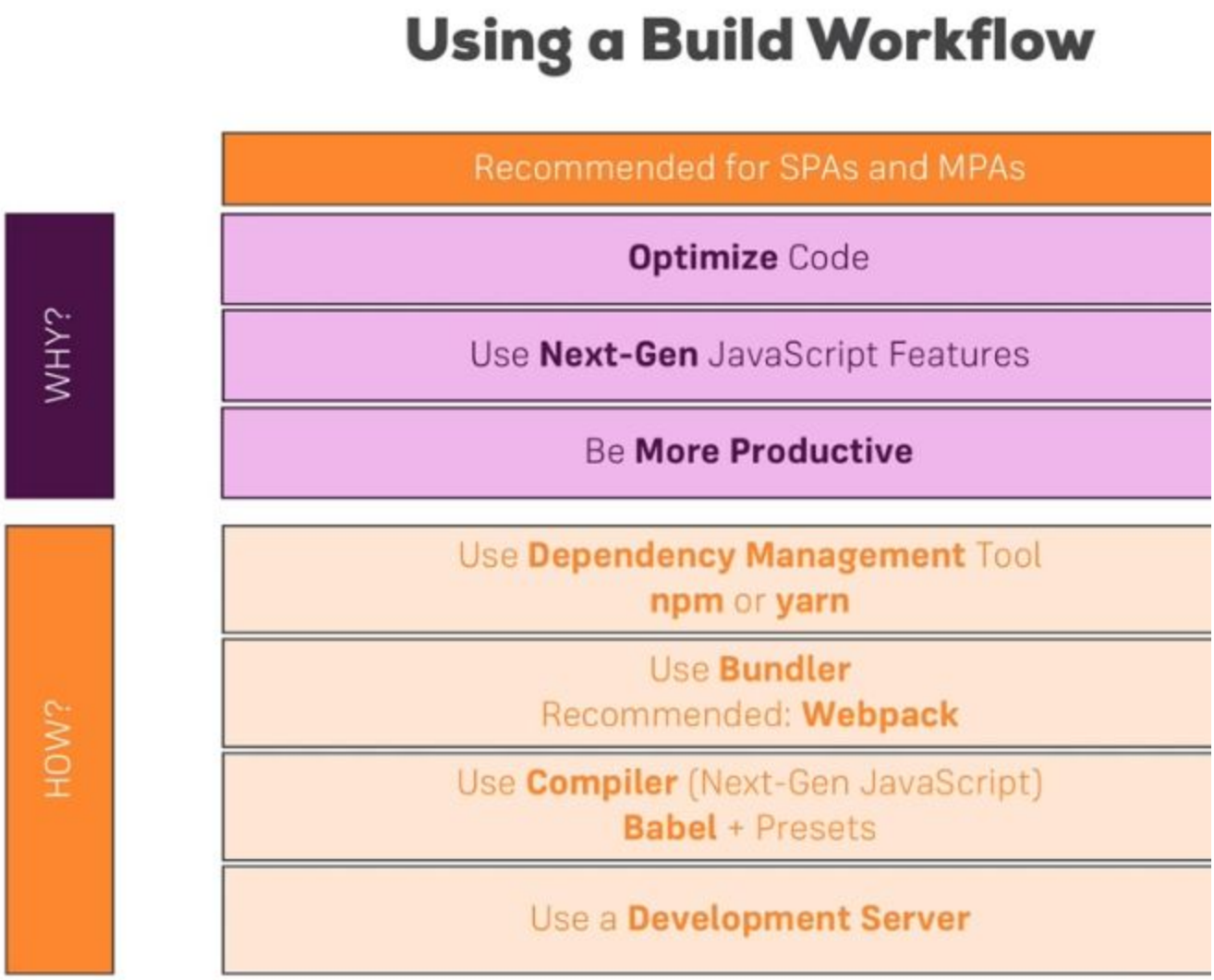
> **slice()**: returns a **shallow copy** of a portion of an array into a new array object

```
var animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

console.log(animals.slice(2));
// expected output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4));
// expected output: Array ["camel", "duck"]

console.log(animals.slice(1, 5));
// expected output: Array ["bison", "camel", "duck", "elephant"]

```

> **splice():** edit array by adding or removing elements

```
var months = ['Jan', 'March', 'April', 'June'];
months.splice(1, 0, 'Feb');
// inserts at 1st index position
console.log(months);
// expected output: Array ['Jan', 'Feb', 'March', 'April', 'June']

months.splice(4, 1, 'May');
// replaces 1 element at 4th index
console.log(months);
// expected output: Array ['Jan', 'Feb', 'March', 'April', 'May']

```

React Basics



> Node.js and NPM

Node.js is a JS runtime executable that can execute JS code that resides on the server. NPM is used to install and manage packages before they can be used/executed on Node.js. It also installs the packages' dependencies.

1) Element: what you want to see on the screen.

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

2) Rendering an element to the DOM inside the **:**

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

3) Elements are immutable: once created, cannot change its children or attribute. You must create the element all over again. The secret sauce is that

the reactDOM only updates the changes made in the snapshots of states for the actual DOM.

Components

- **Components** are like java functions. They **accept arbitrary inputs (props)** and **returns the React-elements** describing what should appear on the screen.
- Lets you split the UI into independent reusable pieces and think about each piece in isolation.

❖ **Functional Components:**

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

❖ **Class Components:**

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

❖ **Rendering a Functional Component**: custom HTML tags support on top of regular HTML tags

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

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Let's recap what happens in this example:

1. We call `ReactDOM.render()` with the `<Welcome name="Sara" />` element.
2. React calls the `Welcome` component with `{name: 'Sara'}` as the props.
3. Our `Welcome` component returns a `<h1>Hello, Sara</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Sara</h1>`.

Note: Always start component names with a capital letter.
React treats components starting with lowercase letters as DOM tags. For example, `<div />` represents an HTML div tag, but `<Welcome />` represents a component and requires `Welcome` to be in scope.

You can read more about the reasoning behind this convention here.

❖ **Rendering a class component in App.js:**

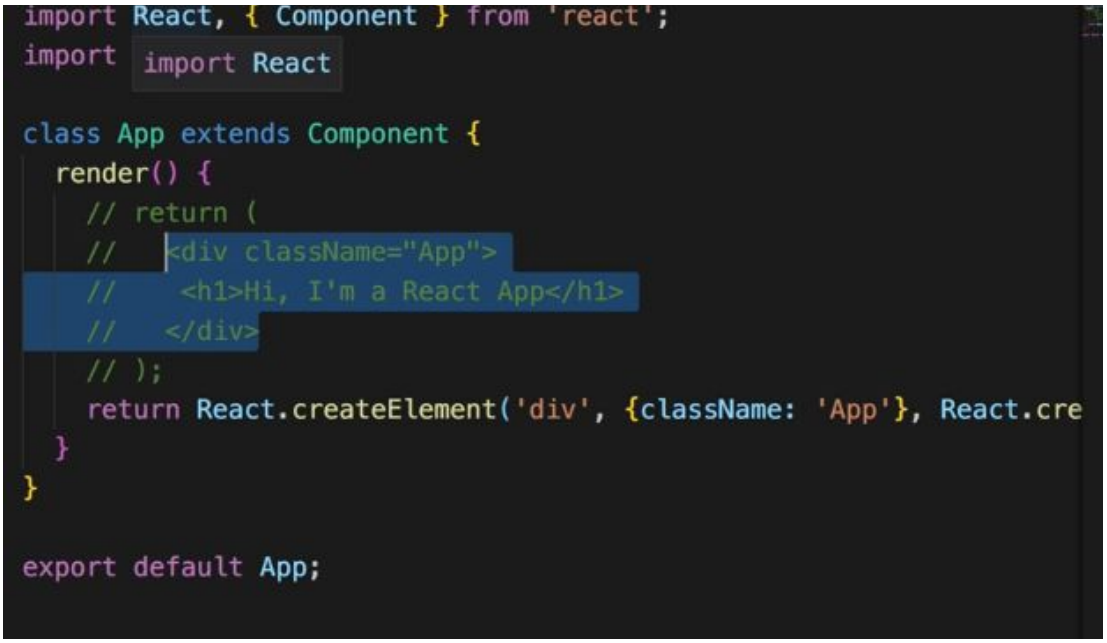
```
import React, { Component } from 'react';
import './App.css';

//import the Person component
import Person from './Person';

//App is a component that is exported from this class
class App extends Component {
  render() {
    return (
      <div>
        <h1> Hi, I am a react app </h1>
      </div>

      //injecting one component inside another
      <Person />
    );
  }
}

export default App;
```

Import App from App.js as Apprename (to index.js)

```
//App component is renamed into Apprename
import Apprename from './App';
import registerServiceWorker from './registerServiceWorker';

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

*Note: ReactDOM can render both a **component** and an **element.***

❖ **React.createElement('element', {propname: 'value'}/null, nest for children):** equivalent to JSX. The JSX gets converted to React.createElement() during compilation.



❖ **More wisdom on components:**

Components are the **core building block of React apps**. Actually, React really is just a library for creating components in its core.

A typical React app therefore could be depicted as a **component tree** - having one root component ("App") and then an potentially infinite amount of nested child components.

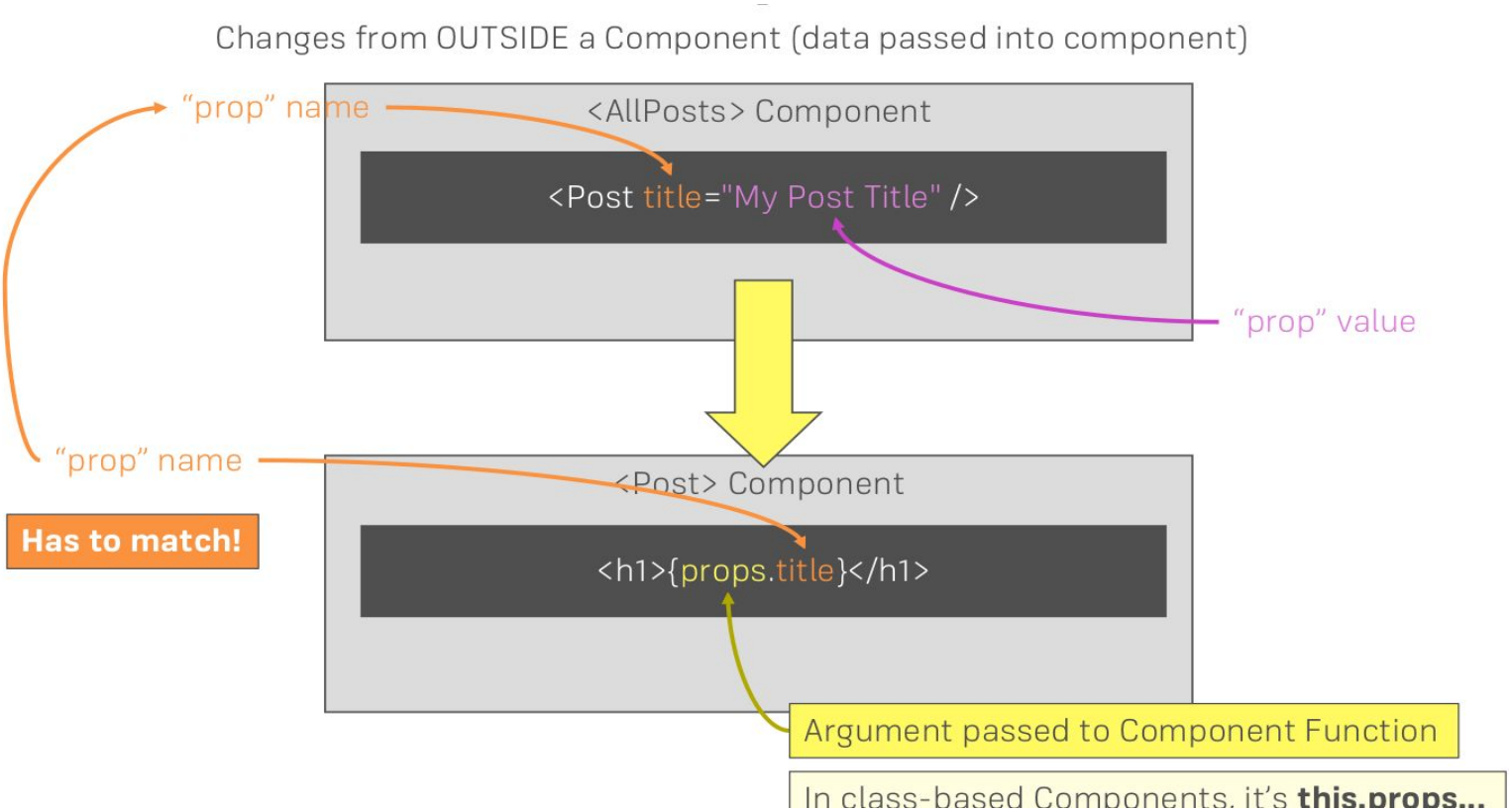
Each component needs to return/ render some **JSX** code - it defines which HTML code React should render to the real DOM in the end.

JSX is NOT HTML but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls.

Props: attribute passing through a component to another component. reusability.

```
import React, { Component } from 'react';
import './App.css';
import Person from './Person/Person';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hi, I'm a React App</h1>
        <p>This is really working!</p>
        <Person name="Max" age="28" />
        <Person name="Manu" age="29" >My Hobbies: Racing</Person>
        <Person name="Stephanie" age="26" />
      </div>
    );
    // return React.createElement('div', {className: 'App'}, React.
  }
}

export default App;
```

```
import React from 'react';

const person = ( props ) => {
    return (
        <div>
            <p>I'm {props.name} and I am {props.age} years old!</p>
            <p>{props.children}</p>
        </div>
    )
};

export default person;
```

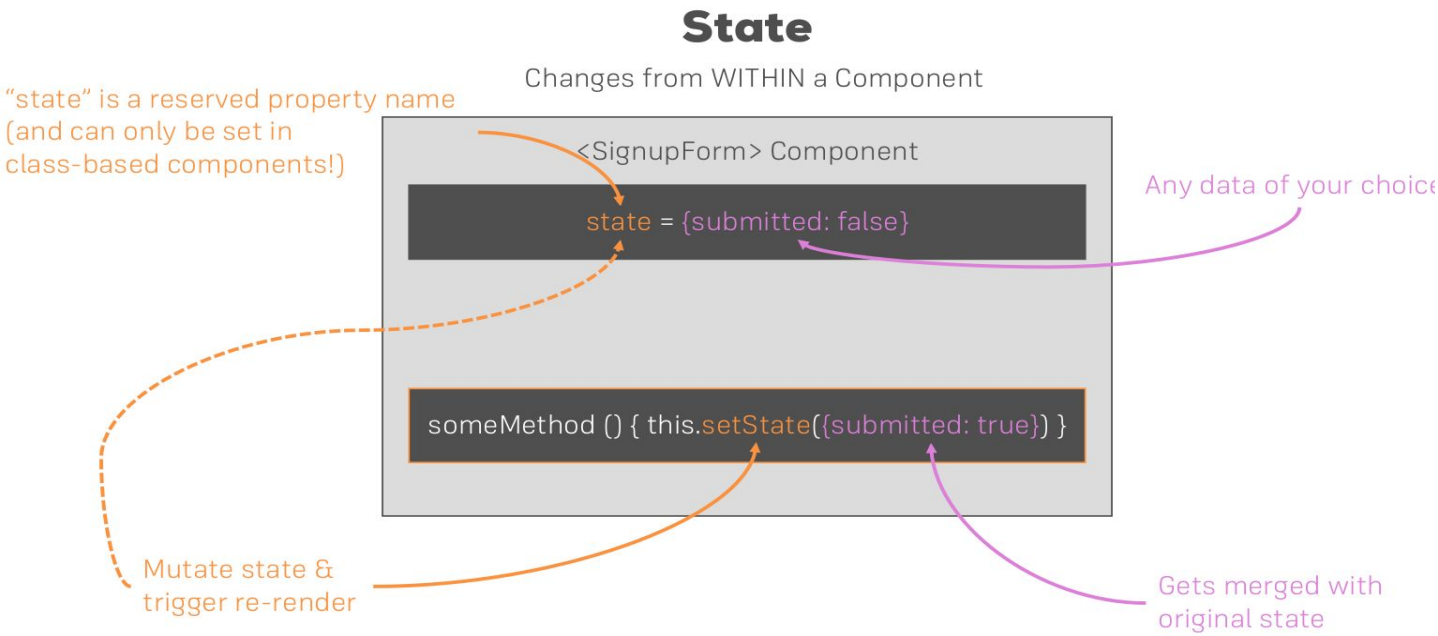


> All React components must act like pure functions with respect to their props.

- Meaning components do not change their props.

State:

- State change re-renders the DOM. Only for class components, not function components. It is the change that a class component can do it itself.



- **Event Handlers can this.setState().**

Putting Everything Together

★ **Creating a React Project:**

- `create-react-app my-app`

★ **Starting the node.js server**

- cd ./directory
- npm start

★ **Index.js attaches App to render everything**

```
//importing react no component
import React from 'react';
//for the DOM.render()
import ReactDOM from 'react-dom';
//import component
import App from './App';
import registerServiceWorker from './registerServiceWorker';

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

★ **App.js that handles the Virtual DOM**

- Imports

```
//Importing React and Components
import React, { Component } from 'react';
//Importing another component from another class file
import Card from './prodcard.js';
//Importing CSS
import './App.css';
```

- Declaring Class Components, injecting imported component with props

```
class App extends Component {
  render() {
    return (
      <div className="card">

        //<Componentname prop0="value" prop1="value".../>
        <Card name="All White" price="$500"/>
        <Card name="Retro Faboulous" price="$1000"/>
      </div>
    );
  }
}
```

- Default the default component for export

```
export default App;
```

★ Reusable Component PostCard

```
//imports for a component in react
import React, {Component} from 'react';

import './body.css';

//example of props and state in a Class Component
class PostCard extends Component{

    render(){

        return(//Your JSX goes here

            <div>
                <h1> Product Item: {this.props.name}  </h1>
                <h2> Item Price: {this.props.price} </h2>

            </div>
        );
    }

}

//props inn a Function COmponent
const Person= (props) => {
    return( //Rendered JSX goes here

        <div>
                <h1> Person Item: {props.name}  </h1>
                <h2> Person Price: {props.price} </h2>
        </div>
    )
};
export default Person;
```

★ **States and Event handlers. Remember, the setState function re-renders the DOM**

```
class App extends Component {

  //state
  state = {
    prodarray:[
    {name: 'hello kitty', price: '$40'},
    {name: 'Jadoo', price: '$400'}
    ]

  }


  //event handler function (using the arrow funct syntac)
  eventHandler = () => {

    //changing the state with setState()
    //setstate merges the state with this
    this.setState({prodarray:[{name: '3', price: '$300'}]})
  }


  render() {
    return (
      <div className="card">
        {/*calling the event handler function*/}
        <button onClick={this.eventHandler}> Clicky! </button>
```

Events:

1) Mouse

```
onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp
```

2) Keyboard

```
onKeyDown onKeyPress onKeyUp
```

3) Clipboard

```
onCopy onCut onPaste
```

More Events: https://reactjs.org/docs/events.html#mouse-events

Some nifty Tricks

> Passing Methods to children components as props [useful when you want to trigger a state change from a stateless child component]

-Parent:

```
<Person
  name={this.state.persons[1].name}
  age={this.state.persons[1].age}
  click={this.switchNameHandler} >M
```

-Child:

```
const person = ( props ) => {
    return (
        <div>
            <p onClick={props.click}>I'm {props.name}
    onClick
        </div>
    )
};
```

> Passing arguments to eventHandlers: method.bind(this, ...args);

```
<button onClick={this.switchNameHandler.bind(this, 'Maximili
```

```
switchNameHandler = (newName) => {
  // console.log('Was clicked!');
  // DON'T DO THIS: this.state.pers
  this.setState( {
    persons: [
      { name: newName, age: 28 },
```

> Propagating change to the state level

```
nameChangedHandler = (event) => {
  this.setState( {
    persons: [
      { name: 'Max', age: 28 },
      { name: event.target.value, age: 29 },
      { name: 'Stephanie', age: 26 }
    ]
  } )
}
```

```
<Person
  name={this.state.persons[1].name}
  age={this.state.persons[1].age}
  click={this.switchNameHandler.bind(this, 'Max!')}
  changed={this.nameChangedHandler} >My Hobbies: Racing</Pe
```

```
const person = ( props ) => {
    return (
        <div>
            <p onClick={props.click}>I'm {props.name} and I am {pro
            <p>{props.children}</p>
            <input type="text" onChange={props.changed} />
        </div>
    )
};
```

> Two-way binding:

```
<input type="text" onChange={props.changed} value={props.name} />
```