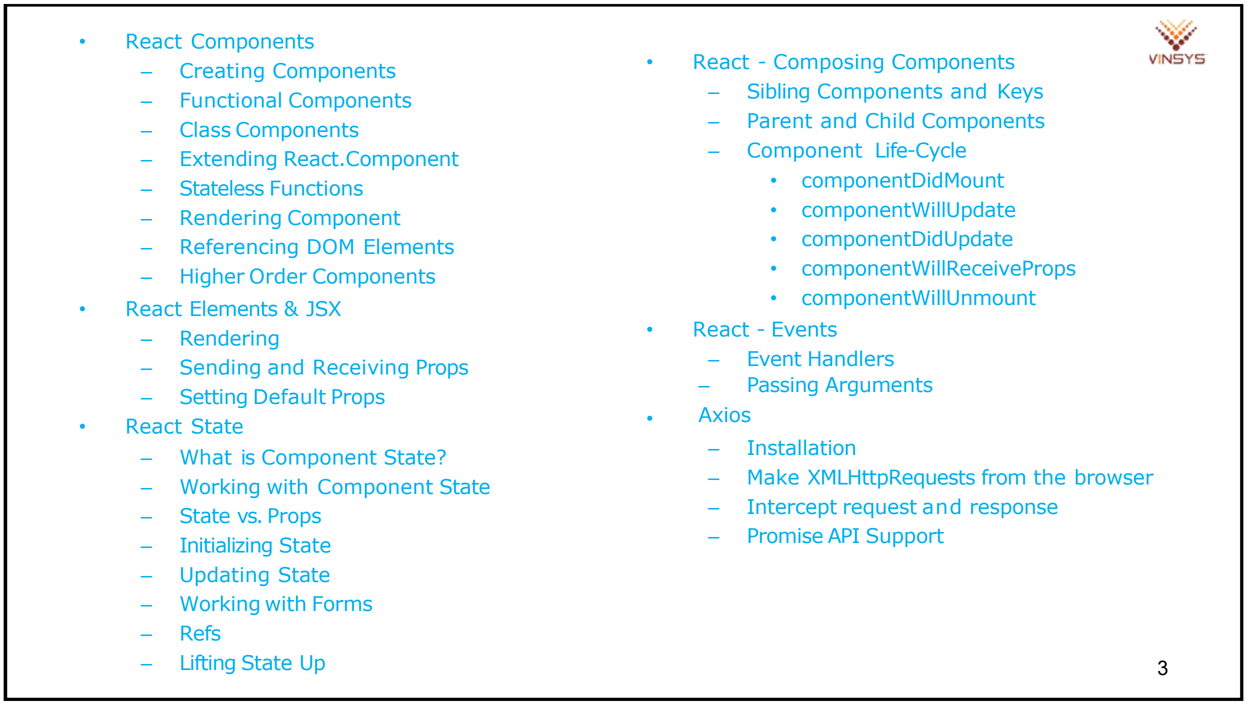
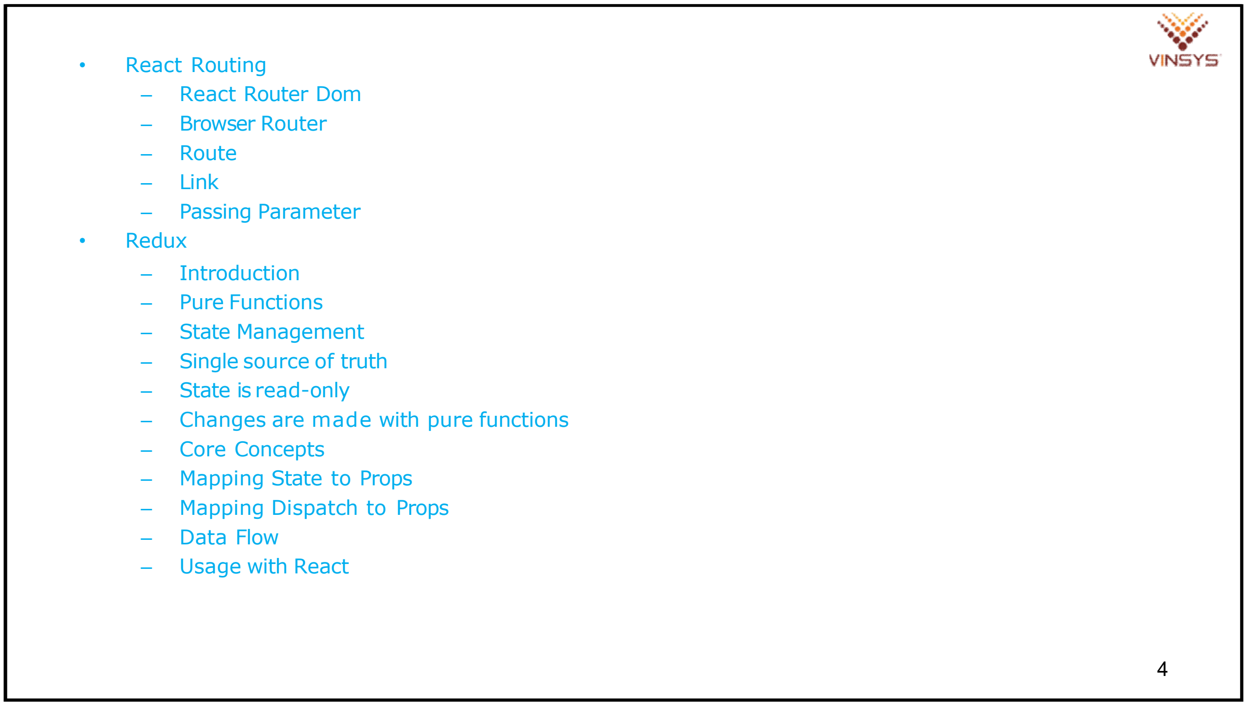
React

- Overview
  - Understand the View Technology
  - MVC – Model View Controller
  - MVVM - Model–view–viewmodel
  - What is React?
  - Understanding View Technology
  - Single Page Applications
  - Multi Page Applications
  - React and SPA
  - Node and NPM basics
- ES6 Fundamentals
  - Default Parameters
  - Arrow Functions
  - Template Literals
  - Multiline String
  - Scoping using Let and Const Keywords
  - Spread Syntax and Rest Parameters
  - Destructuring
  - Promises
  - Async / Await
  - Classes and Objects
- React Basics
  - DOM vs Virtual DOM
  - How does the Virtual DOM work?
  - Introduction
  - Installation and basic setup
  - React CLI tool
  - Basic Setup
- JSX
  - What is JSX?
  - Babel
  - React.createElement
  - Embedding Expressions in JSX
  - Attributes with JSX
  - Props
  - JSX Children



- React Components
  - Creating Components
  - Functional Components
  - Class Components
  - Extending React.Component
  - Stateless Functions
  - Rendering Component
  - Referencing DOM Elements
  - Higher Order Components
- React Elements & JSX
  - Rendering
  - Sending and Receiving Props
  - Setting Default Props
- React State
  - What is Component State?
  - Working with Component State
  - State vs. Props
  - Initializing State
  - Updating State
  - Working with Forms
  - Refs
  - Lifting State Up
- React - Composing Components
  - Sibling Components and Keys
  - Parent and Child Components
  - Component Life-Cycle
    - componentDidMount
    - componentWillUpdate
    - componentDidUpdate
    - componentWillReceiveProps
    - componentWillUnmount
- React - Events
  - Event Handlers
  - Passing Arguments
- Axios
  - Installation
  - Make XMLHttpRequests from the browser
  - Intercept request and response
  - Promise API Support



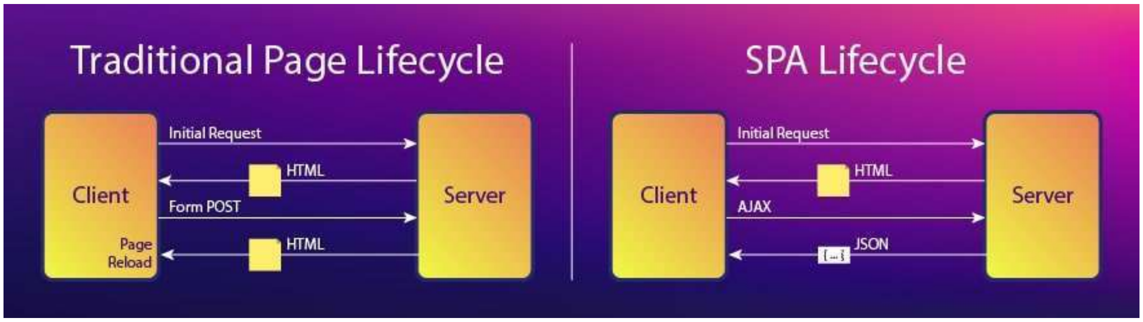
- React Routing
  - React Router Dom
  - Browser Router
  - Route
  - Link
  - Passing Parameter
- Redux
  - Introduction
  - Pure Functions
  - State Management
  - Single source of truth
  - State is read-only
  - Changes are made with pure functions
  - Core Concepts
  - Mapping State to Props
  - Mapping Dispatch to Props
  - Data Flow
  - Usage with React

Traditional Web Applications

- Every time the app calls the server, the server renders a new HTML page.
- Triggers a page refresh in the browser

SPA – Single Page Application

- In an SPA, after the first page loads, all interaction with the server happens through AJAX calls. (Async. JavaScript & XML)
- These AJAX calls return data in JSON format.
- All UI interaction occurs on the client side, through JavaScript



Javascript

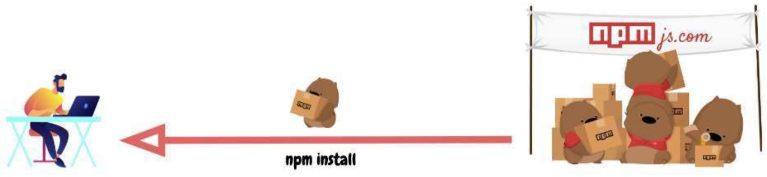
- Scripting language for web pages
- Single threaded
- Runs inside browser
- Javascript Engines
  - Spidermonkey (Firefox)
  - JavaScriptCore also called Nitro (safari)
  - Chakra - Edge
  - V8 - Chrome

Node JS

- JavaScript runtime built on Chrome's V8 JavaScript engine.
- Not a programming language
- Event-driven, non-blocking

NPM

- Package manager for the Node JavaScript platform.
- www.npmjs.com hosts thousands of free packages
- $ npm install <package name>

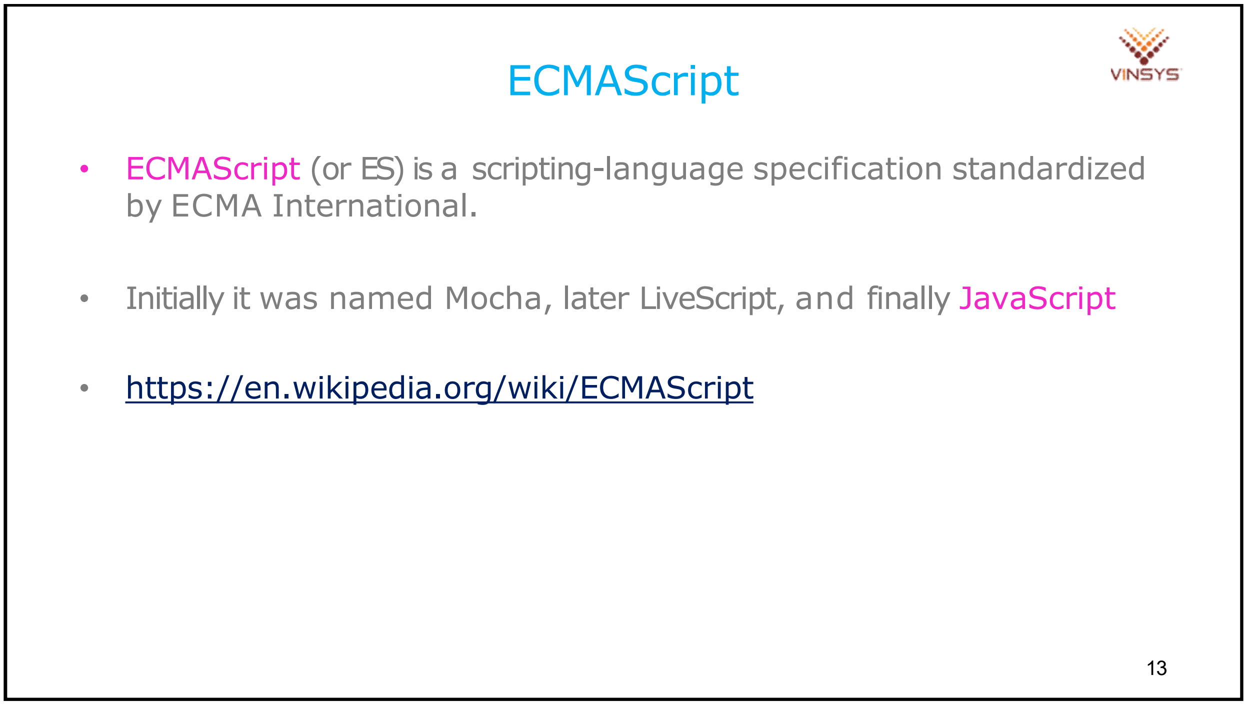


npm scripts

```
{
 "scripts": {
   "build": "tsc",
   "format": "prettier --write **/*.ts",
   "format-check": "prettier --check **/*.ts",
   "lint": "eslint src/**/*.ts",
   "pack": "ncc build",
   "test": "jest",
   "all": "npm run build && npm run format && npm run lint && npm run pack && 
npm test"
 }
}
```

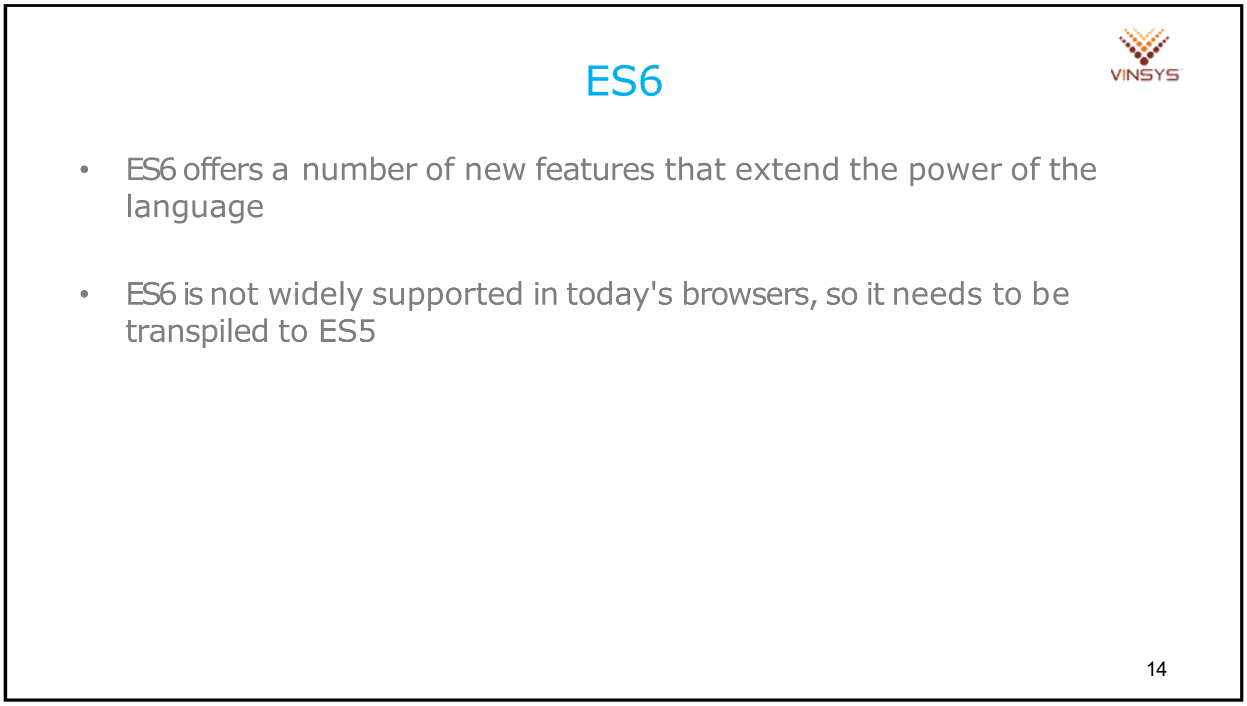
Yarn

- Yarn is a package manager for your code
- A package contains all the code being shared as well as a package.json file which describes the package.
- npm install –global yarn
- yarn –version
- yarn init
- yarn add <package>



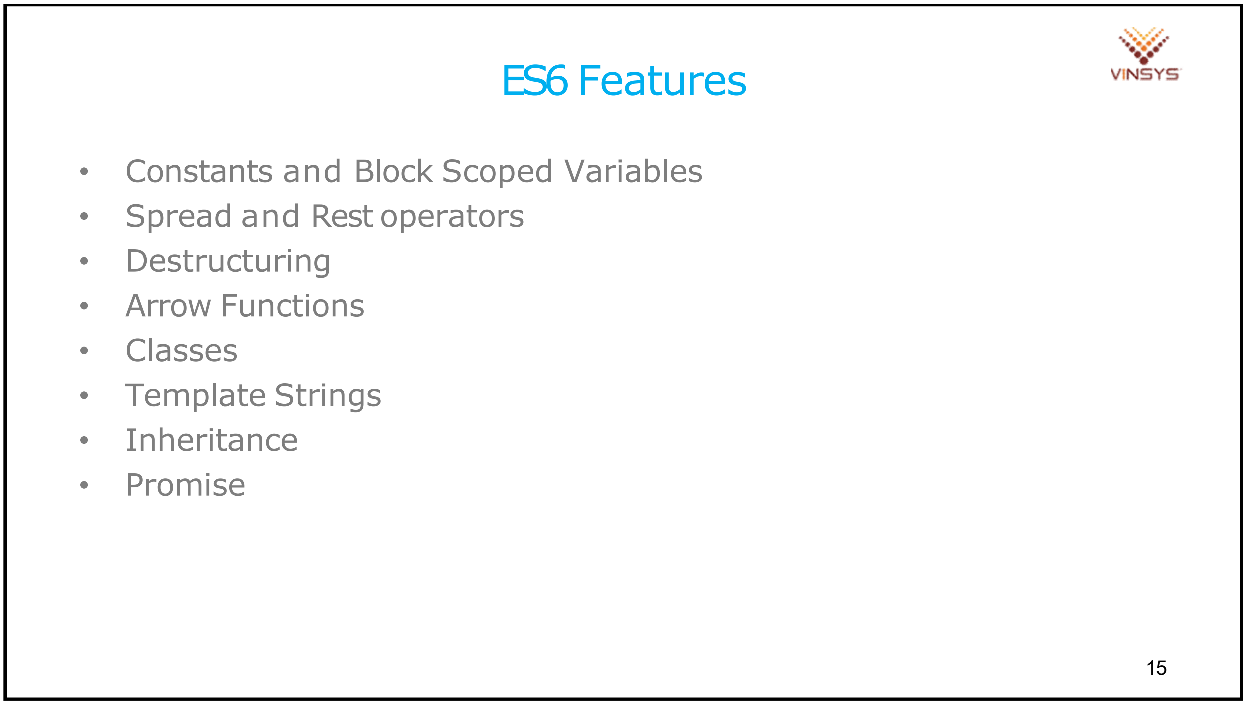
ECMAScript

- ECMAScript (or ES) is a scripting-language specification standardized by ECMA International.
- Initially it was named Mocha, later LiveScript, and finally JavaScript
- https://en.wikipedia.org/wiki/ECMAScript



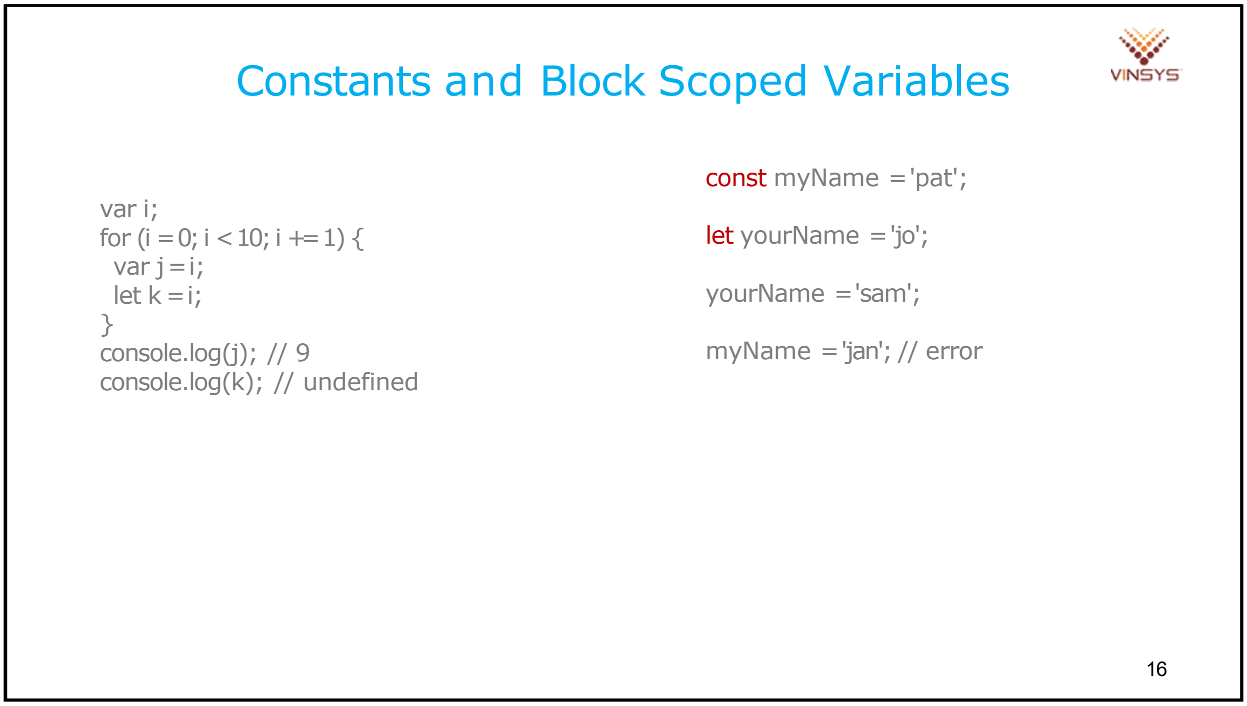
ES6

- ES6 offers a number of new features that extend the power of the language
- ES6 is not widely supported in today's browsers, so it needs to be transpiled to ES5



ES6 Features

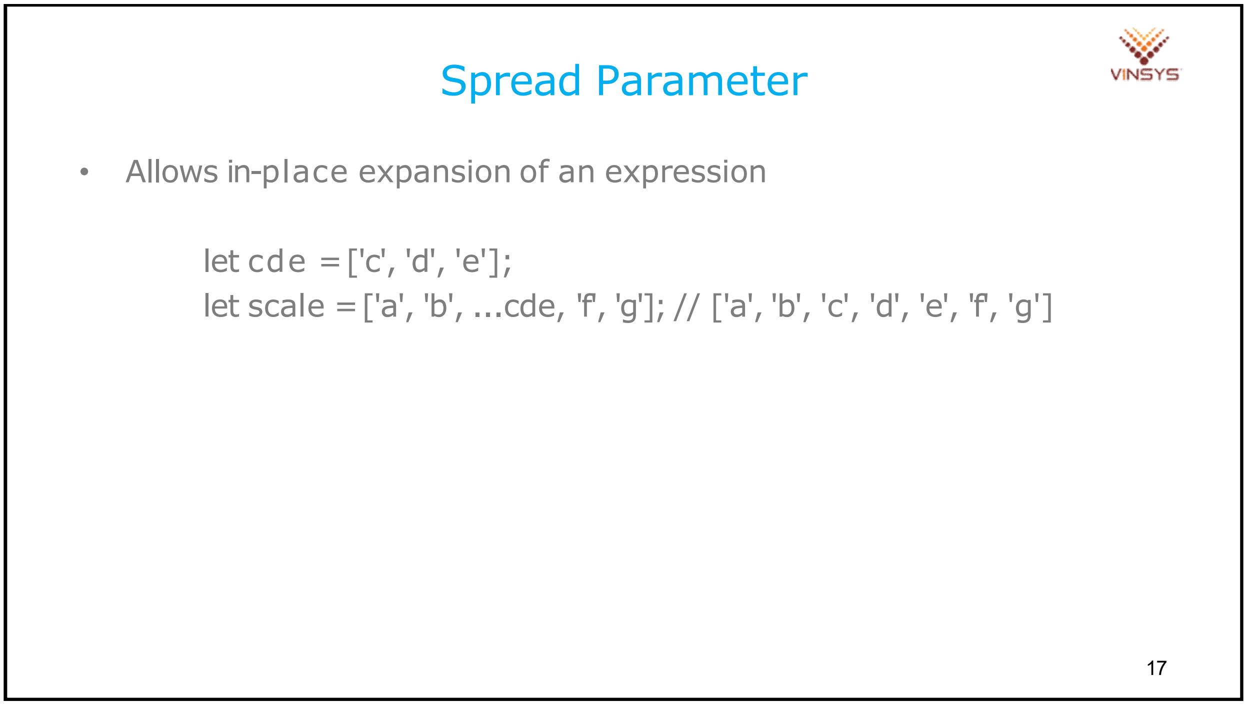
- Constants and Block Scoped Variables
- Spread and Rest operators
- Destructuring
- Arrow Functions
- Classes
- Template Strings
- Inheritance
- Promise



Constants and Block Scoped Variables

```
var i;
for (i = 0; i < 10; i += 1) {
  var j = i;
  let k = i;
}
console.log(j); // 9
console.log(k); // undefined
```

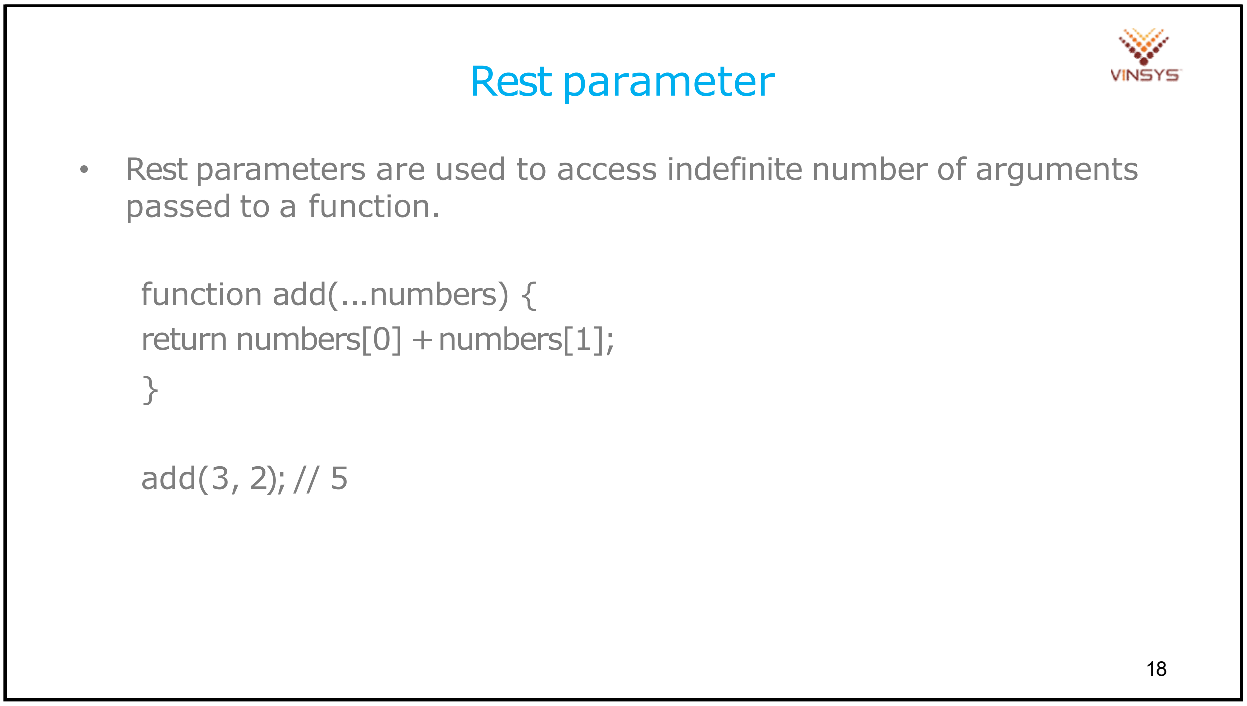
```
const myName = 'pat';

let yourName = 'jo';

yourName = 'sam';

myName = 'jan'; // error
```



Spread Parameter

- Allows in-place expansion of an expression

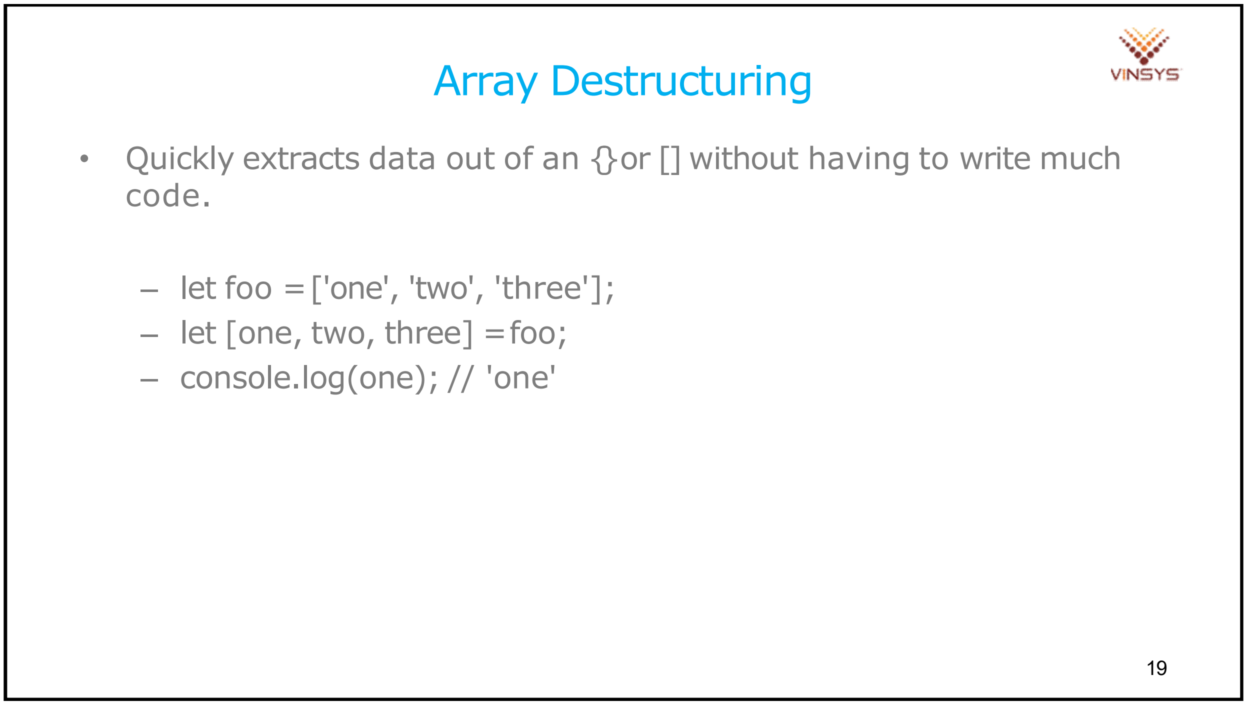
```
let cde = ['c', 'd', 'e'];
let scale = ['a', 'b', ...cde, 'f', 'g']; // ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```



Rest parameter

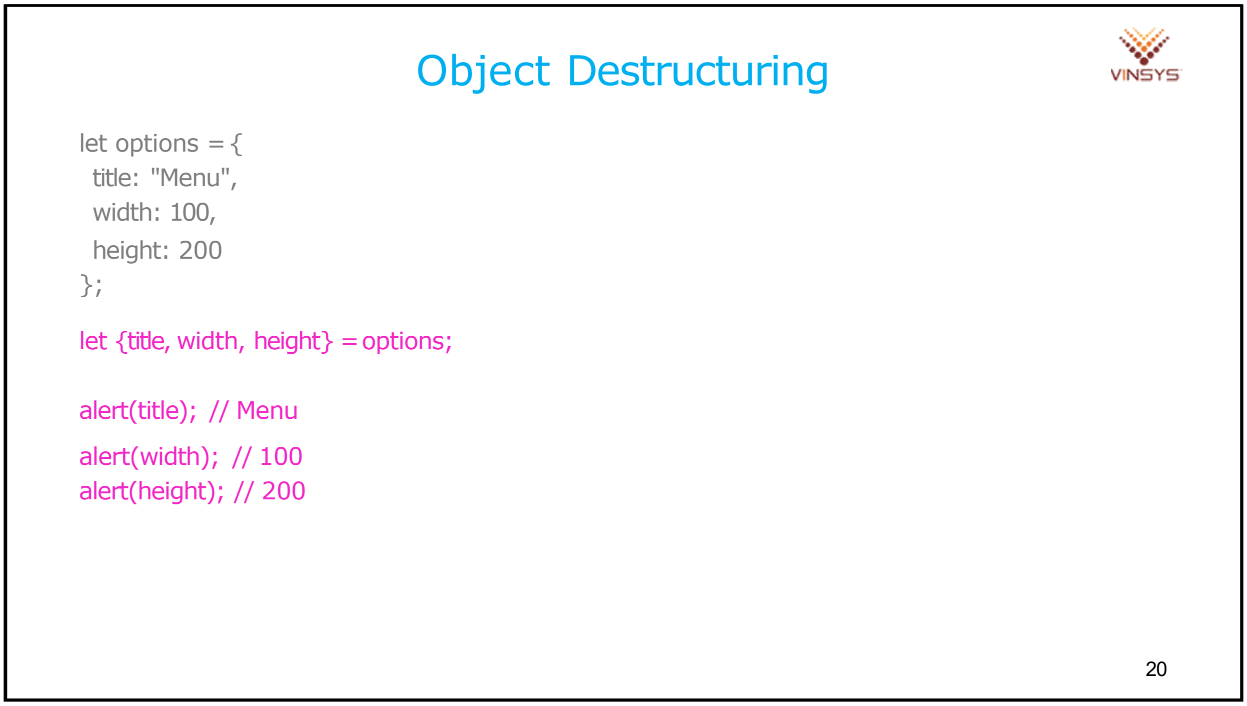
- Rest parameters are used to access indefinite number of arguments passed to a function.

```
function add(...numbers) {
return numbers[0] + numbers[1];
}

add(3, 2); // 5
```



Array Destructuring

- Quickly extracts data out of an {}or [] without having to write much code.
  - let foo = ['one', 'two', 'three'];
  - let [one, two, three] = foo;
  - console.log(one); // 'one'



Object Destructuring

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};
let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Arrow Functions

- The new "fat arrow" notation can be used to define anonymous functions in a simpler way.

```
var hello = function() {
      return "Hello World!";
    }

var hello = () => {
      return "Hello World!";
    }
```

this

```
class Employee {
public name:string;
public address:string;
display1():void
{
setTimeout(function()
{
console.log(this.name); },1000);
}}
display2():void
{
setTimeout(() =>
{console.log(this.name); },1000);
}}
```

```
let emp:Employee = new Employee();
emp.name=“Sunny";
emp.address="Pune"
emp.display1();
emp.display2();
```

Classes

- Classes are a new feature in ES6, used to describe the blueprint of an object

```
class Hamburger {
constructor() {
    // This is the constructor.
}
listToppings() {
    // This is a method.
}
}
```

Object

- An object is an instance of a class which is created using the new operator.

```
let burger = new Hamburger();
burger.listToppings();
```

Template Strings

```
var name = 'Sam';
var age = 42;
console.log('hello my name is ' + name + ' I am ' + age + ' years old');

var name = 'Sam';
var age = 42;
console.log(`hello my name is ${name}, and I am ${age} years old`);
```

Inheritance

```
// Base Class : ES6
class Bird {
  constructor(weight, height) {
    this.weight = weight;
    this.height = height;
  }
  walk() {
    console.log('walk!');
  }
}
```

```
// Subclass
class Penguin extends Bird {
  constructor(weight, height) {
    super(weight, height);
  }
  swim() {
    console.log('swim!');
  }
}
// Penguin object
let penguin = new Penguin(...);
penguin.walk(); //walk!
penguin.swim(); //swim!
```

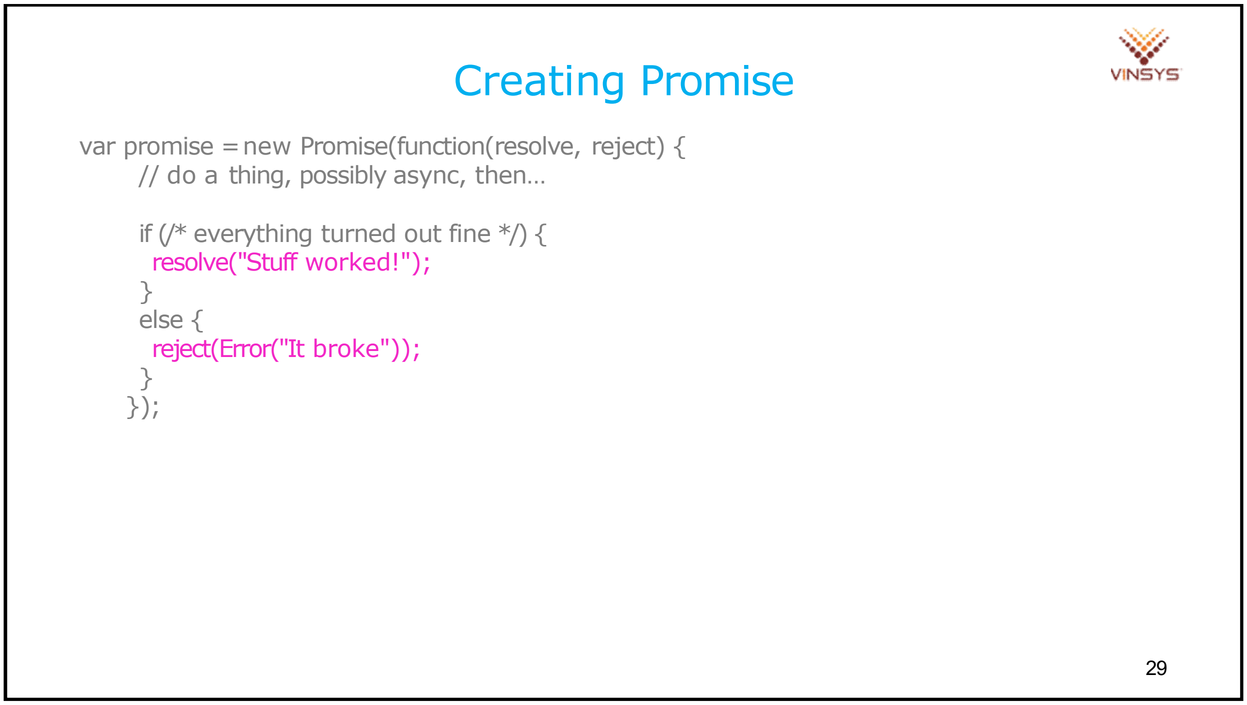
Promise

- JavaScript Promises are a new addition to ES6
- The promise constructor takes one argument, a callback with two parameters, resolve and reject.
- Do something within the callback, then call resolve if everything worked, otherwise call reject.

Syntax

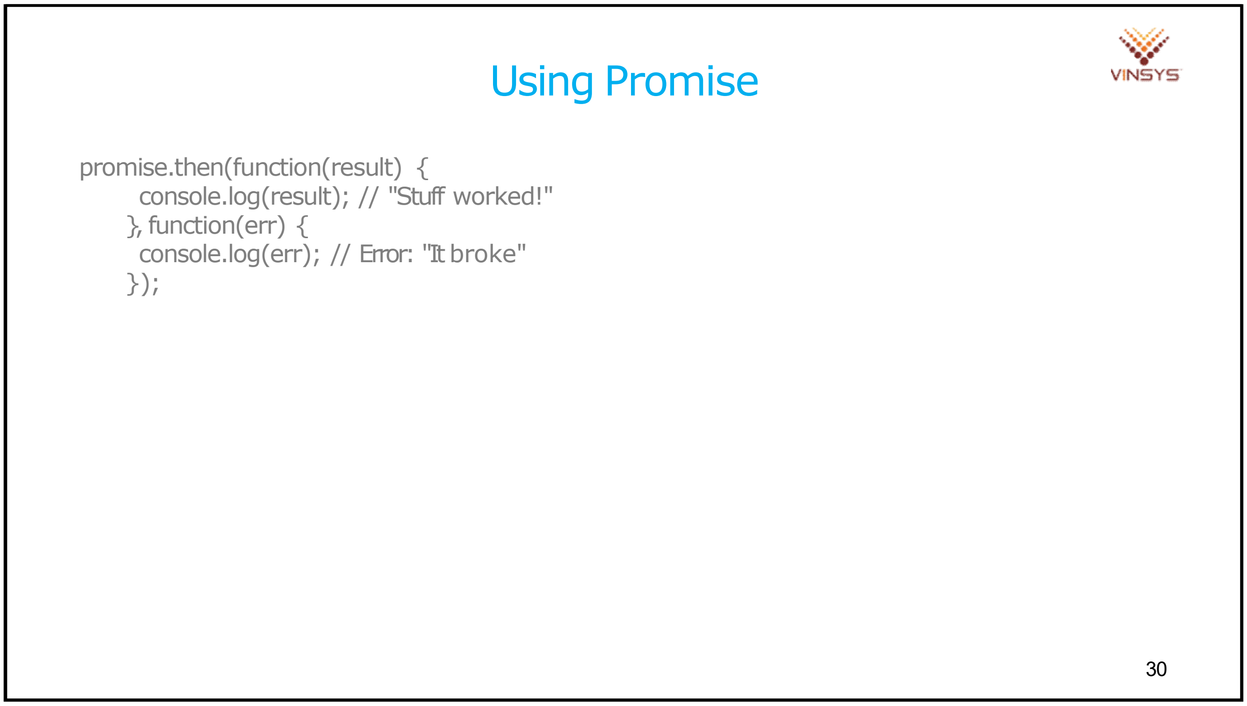
```
var mypromise = new Promise(function(resolve, reject) {

        // asynchronous code to run here
         // call resolve() to indicate task successfully completed
         // call reject() to indicate task has failed
})
```

- resolve(value) — if the job finished successfully, with result value.
- reject(error) — if an error occurred, error is the error object.



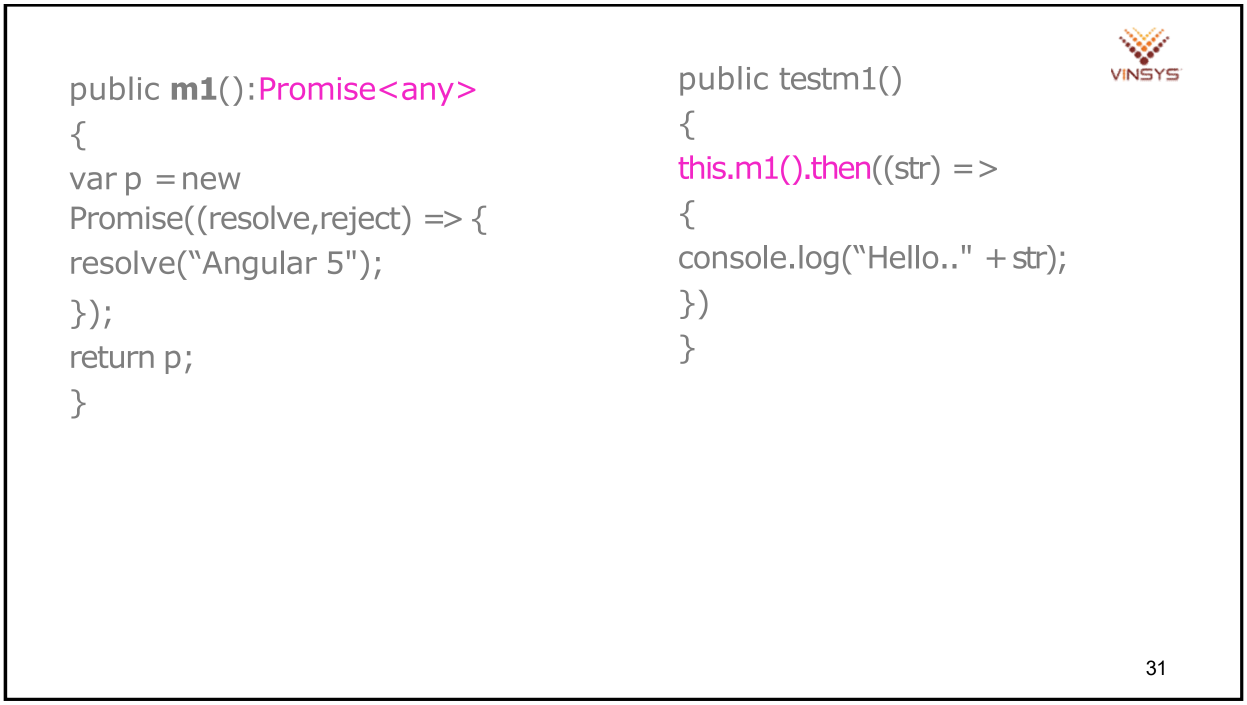
Creating Promise

```
var promise = new Promise(function(resolve, reject) {
        // do a thing, possibly async, then…

        if (/* everything turned out fine */) {
          resolve("Stuff worked!");
        }
        else {
          reject(Error("It broke"));
        }
      });
```



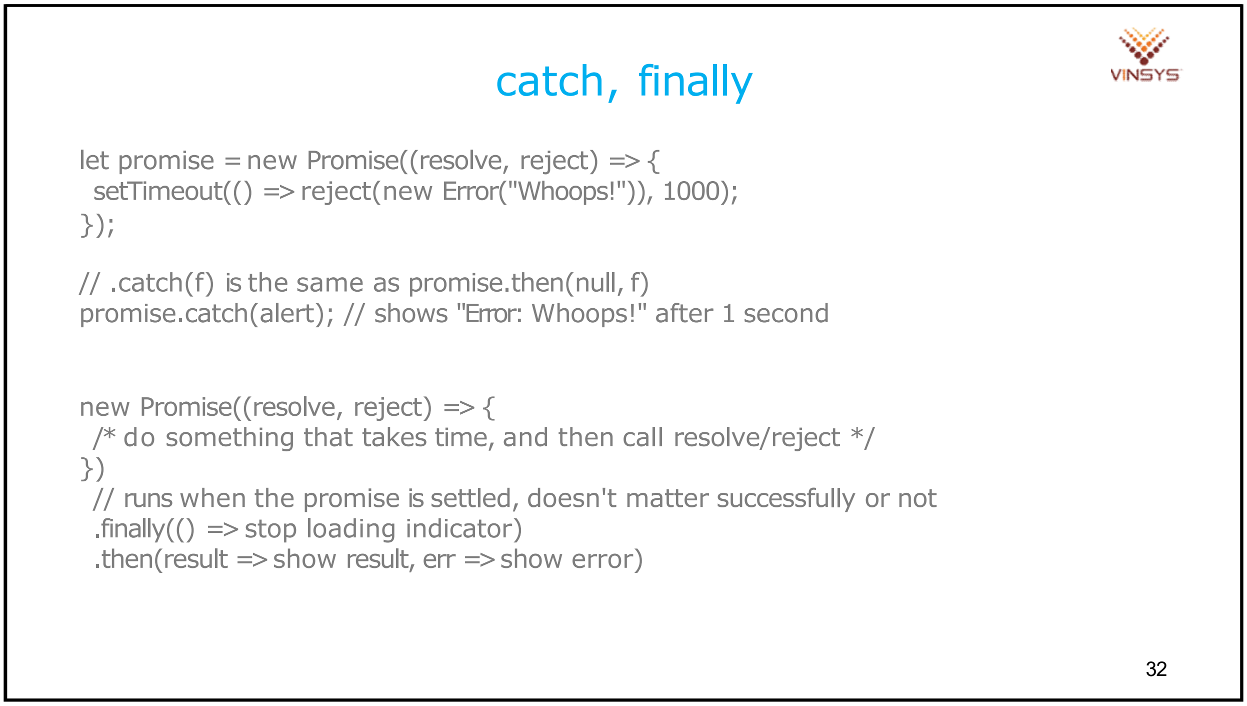
Using Promise

```
promise.then(function(result) {
        console.log(result); // "Stuff worked!"
      }, function(err) {
        console.log(err); // Error: "It broke"
      });
```



```
public m1():Promise<any>
{
var p =new
Promise((resolve,reject) => {
resolve(“Angular 5");
});
return p;
}
```

```
public testm1()
{
this.m1().then((str) =>
{
console.log(“Hello.." + str);
})
}
```



catch, finally

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second


new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve/reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
  .finally(() => stop loading indicator)
  .then(result => show result, err => show error)
```

async/await

- Special syntax to work with promises in a more comfortable fashion

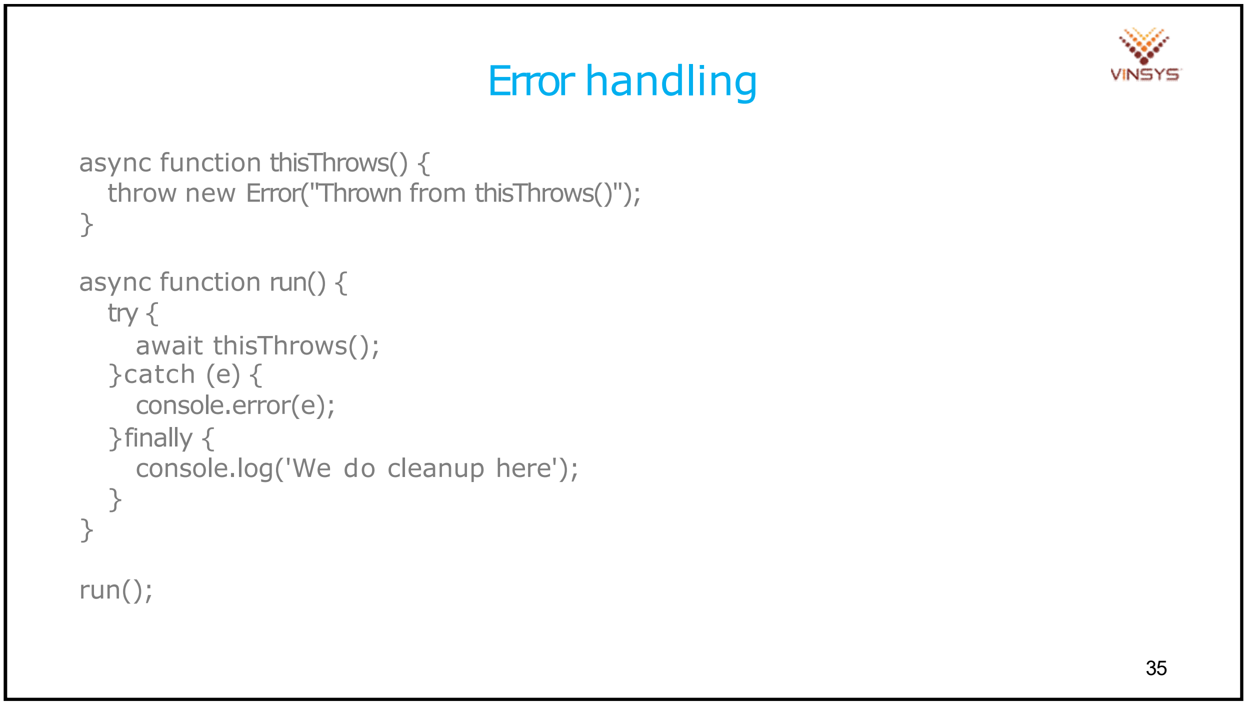
```
async function f() {
 return 1;
}

 f().then(alert); // 1
```

- “async” before a function always returns a promise

await

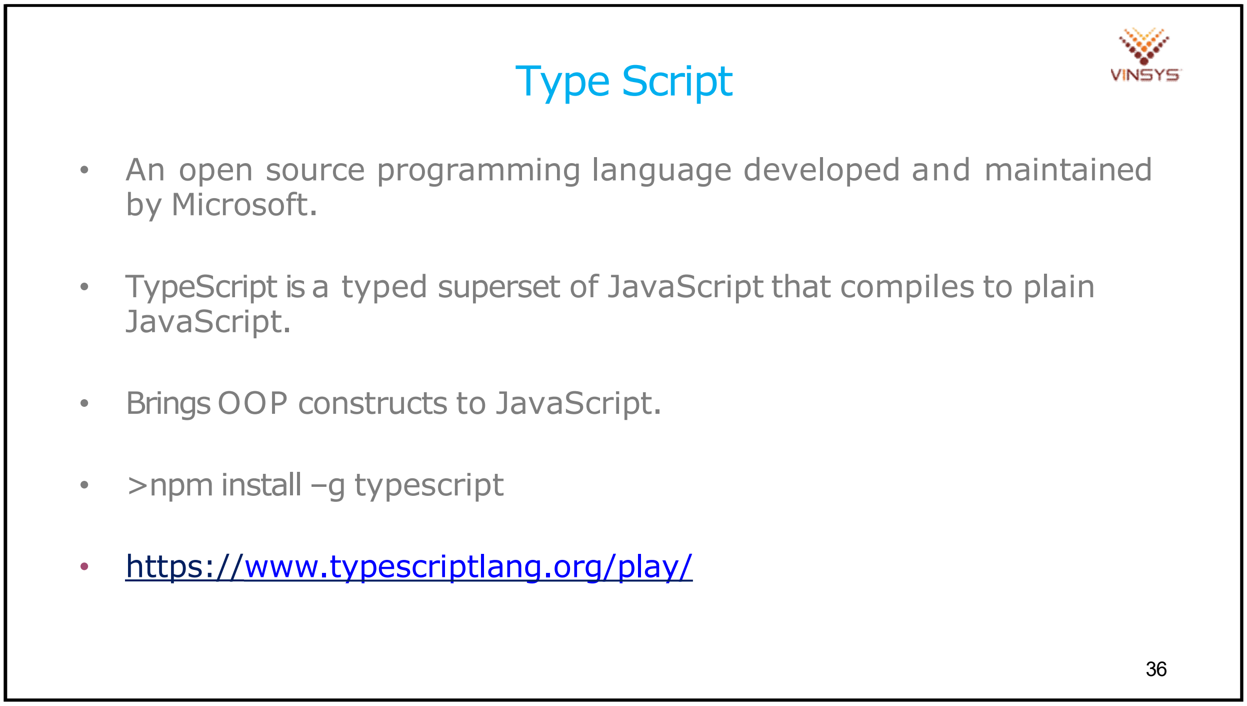
- await makes JavaScript wait until that promise settles and returns its result.

```
async function f() {

 let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("done!"), 1000)
 });

 let result = await promise; // wait until the promise resolves (*)

 alert(result); // "done!"
}

f();
```



Error handling

```
async function thisThrows() {
   throw new Error("Thrown from thisThrows()");
}

async function run() {
   try {
      await thisThrows();
   }catch (e) {
      console.error(e);
   }finally {
      console.log('We do cleanup here');
   }
}

run();
```



Type Script

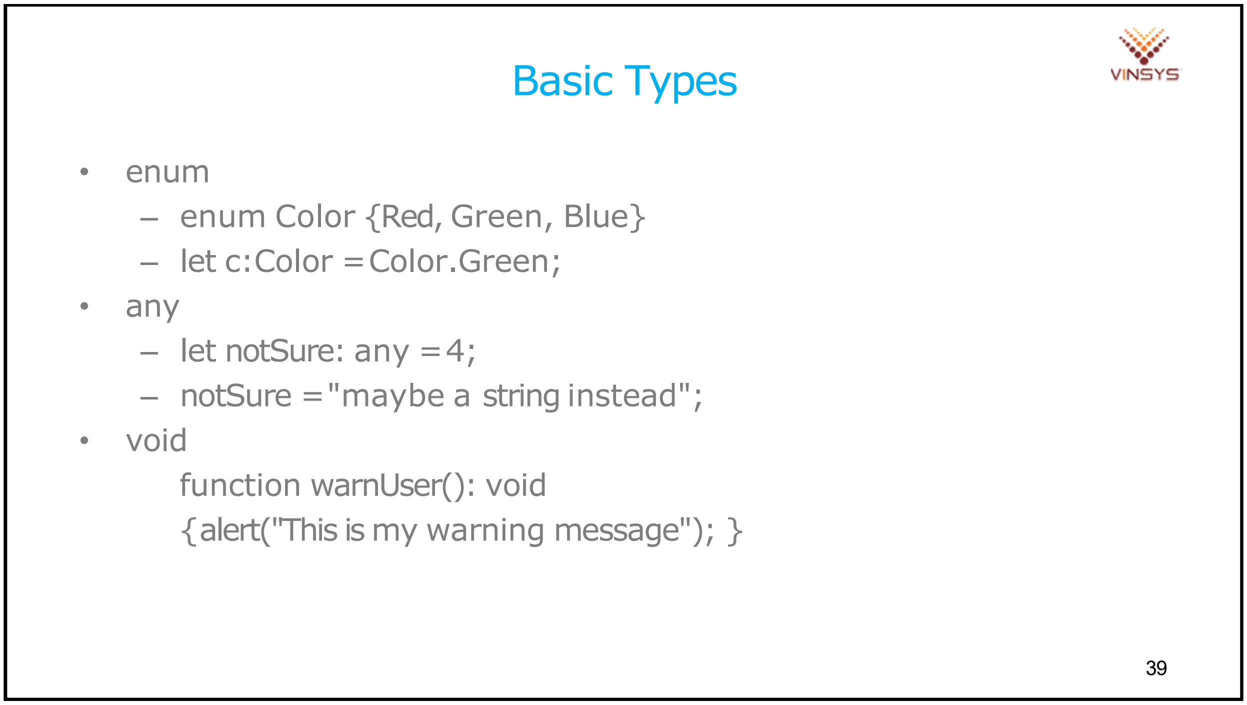
- An open source programming language developed and maintained by Microsoft.
- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
- Brings OOP constructs to JavaScript.
- >npm install –g typescript
- https://www.typescriptlang.org/play/

Static Typing

- A very distinctive feature of TypeScript is the support of static typing.
- Code written in TypeScript is more predictable, and is generally easier to debug.
- Declare the types of variables, and the compiler will make sure that they aren't assigned the wrong types of values.

Basic Types

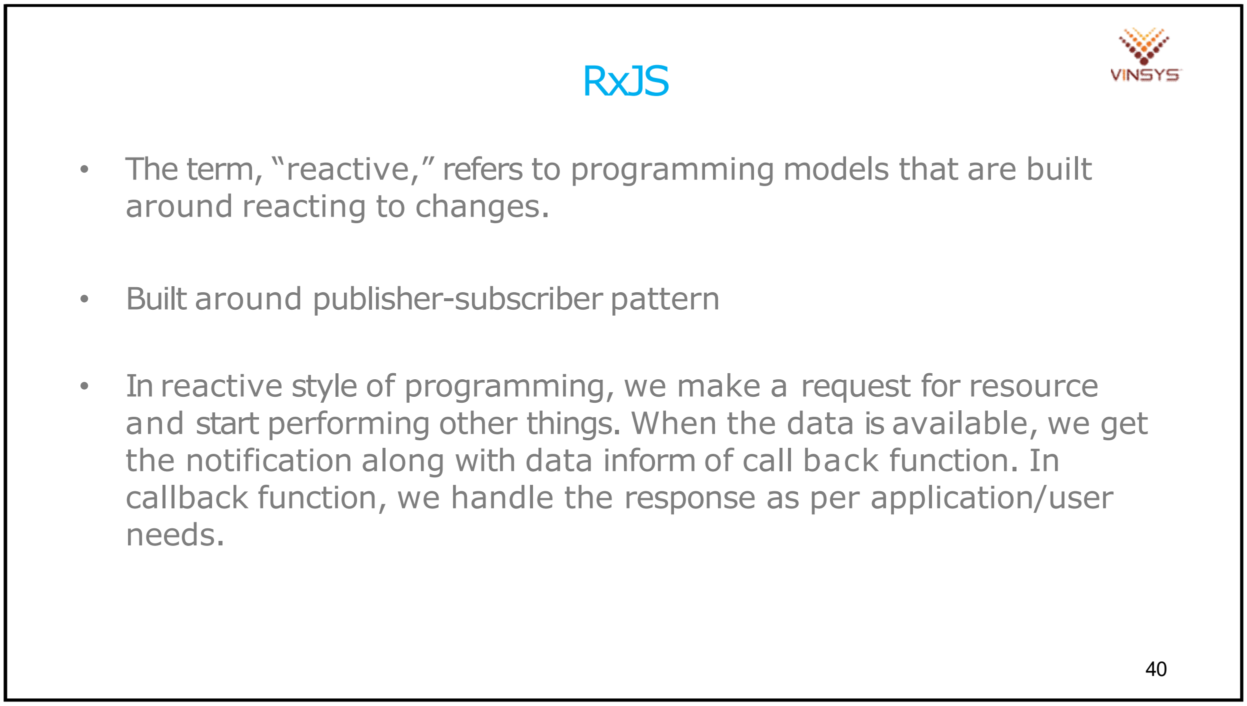
- Boolean
  - let isDone: boolean = false;
- Number
  - let num: number = 6;
- String
  - let color: string = "blue";
- Array
  - let list: number[] = [1, 2, 3];
  - let list: Array<number> = [1, 2, 3];



Basic Types

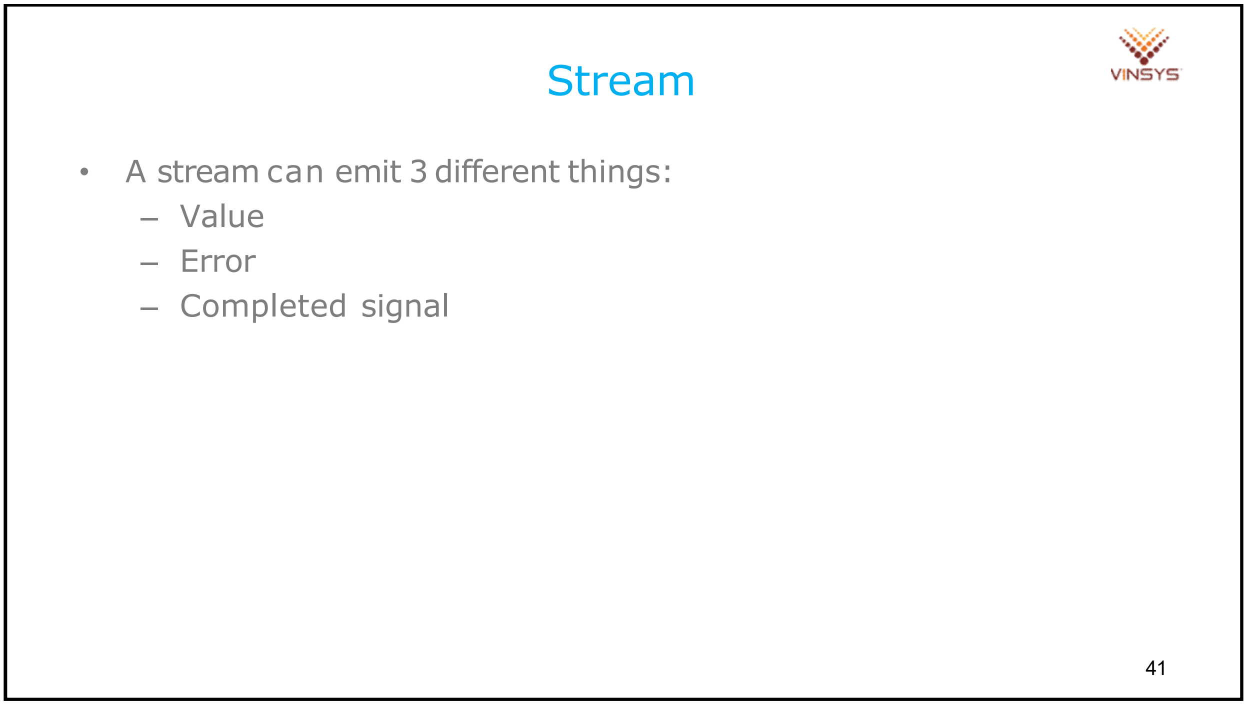
- enum
  - enum Color {Red, Green, Blue}
  - let c:Color = Color.Green;
- any
  - let notSure: any = 4;
  - notSure = "maybe a string instead";
- void

```
function warnUser(): void
{alert("This is my warning message"); }
```



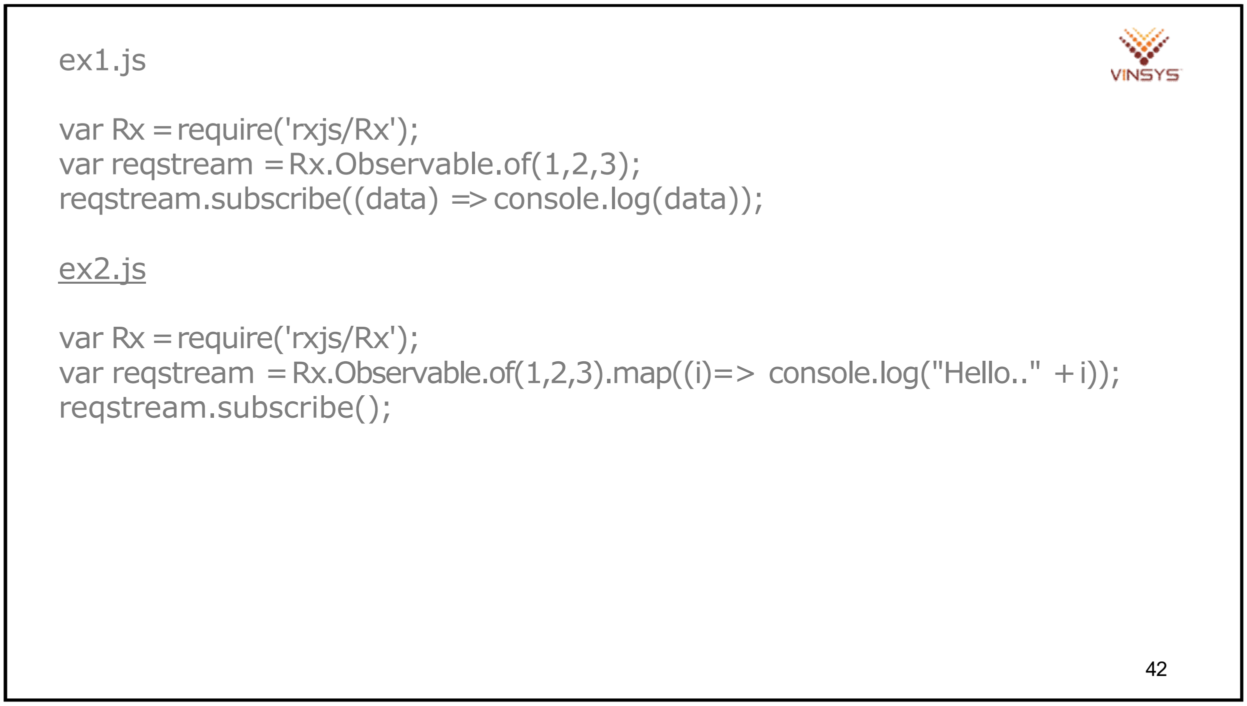
RxJS

- The term, "reactive," refers to programming models that are built around reacting to changes.
- Built around publisher-subscriber pattern
- In reactive style of programming, we make a request for resource and start performing other things. When the data is available, we get the notification along with data inform of call back function. In callback function, we handle the response as per application/user needs.



Stream

- A stream can emit 3 different things:
  - Value
  - Error
  - Completed signal



ex1.js

```
var Rx = require('rxjs/Rx');
var reqstream = Rx.Observable.of(1,2,3);
reqstream.subscribe((data) => console.log(data));
```

ex2.js

```
var Rx = require('rxjs/Rx');
var reqstream = Rx.Observable.of(1,2,3).map((i)=> console.log("Hello.." + i));
reqstream.subscribe();
```

```
class ContactService {
contacts = [
   new Contact("John","John@gmail.com","983344562"),
   new Contact("Charles","charles@gmail.com","983344123")
];

public findAll(): Observable<Array<Contact>> {
      var obs = Observable.create((o) => {
         o.next(this.contacts);
         o.complete();
      });
            return obs;
}
```

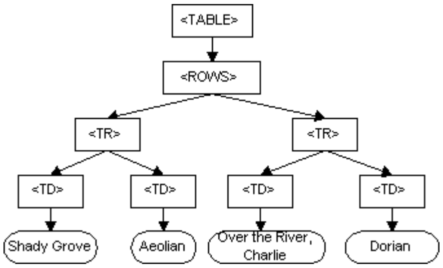
```
var contactService<ContactService> = new ContactService();


this.contactService.findAll().subscribe((data) => {
     console.log(data);
   });
```

What is React?

- React is a declarative, efficient, and flexible JavaScript library for building user interfaces.
- React abstracts away the DOM from you, offering a simpler programming model and better performance.
- Uses virtual DOM which is a JavaScript object.

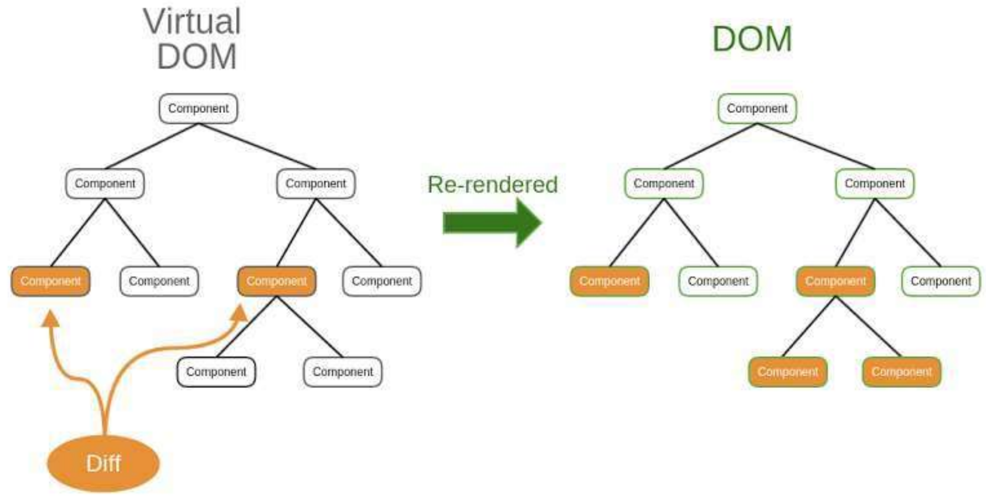
DOM

- The Document Object Model (DOM) is a programming API for HTML and XML documents.
- Defines the logical structure of documents and the way a document is accessed and manipulated.



Virtual DOM

- The Virtual DOM is an abstraction of the HTML DOM.
- It is lightweight and detached from the browser-specific implementation details.
- Think of the virtual DOM as React’s local and simplified copy of the HTML DOM.



Install

```
>npm install -g create-react-app
>create-react-app my-app
>cd myapp
>npm start
```

- Or

```
>npx create-react-app myapp
```

JSX

- Syntax extension to JavaScript.
- JSX is an XML/HTML-like syntax used by React

```
const element = <h1>Hello, world!</h1>;
const element = <div tabIndex="0"></div>;
const element = (
  <div>
  <h1>Hello, world</h1>
  <h3>This is a test</h3>
  </div>
 );
```

Syntactical Sugar

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

compiles to :

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

```
<h1>Hello World!!!</h1>
var xyz = React.createElement('h1',null,’Hello World!!!')
```

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

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

Why JSX?

- Funny tag syntax is neither a string nor HTML.

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

- Syntax extension to JavaScript.

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

```
// Using JS with React.createElement
React.createElement('form', null,
React.createElement('div', {'className': 'form-group'},
  React.createElement('label', {'htmlFor': 'email'}, 'Email address'),
  React.createElement('input', {'type': 'email', 'id': 'email', 'className': 'form-control'}),
 ),
 React.createElement('button', {'type': 'submit', 'className': 'btn btn-primary'},
   'Submit')
)

// Using JSX
<form>
 <div className="form-group">
  <label htmlFor="email">Email address</label>
  <input type="email" id="email" className="form-control"/>
 </div>
 <button type="submit" className="btn btn-primary">Submit</button>
</form>
```

Attributes with JSX

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

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

Props in JSX

- Passing JavaScript expression

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

- Passing String Literals

```
<MyComponent message="hello world" />
<MyComponent message={'hello world'} />
<MyComponent message="&lt;3" />
<MyComponent message={'<3'} />
```

- Spread Attributes

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}
function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

Children in JSX

- Content between those tags is passed as a special prop : **props.children**
- String Literals
  `<MyComponent>`**Hello world!**`</MyComponent>`
- JavaScript Expressions as Children
  `<MyComponent>{'foo'}</MyComponent>`

Rendering Elements

- A ReactElement is a light, stateless, immutable, virtual representation of a DOM Element.
- An element describes what you want to see on the screen
- ReactElements can be render into the “real” DOM

```
<div id="root"></div>

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

```
const Watch = (props) => <div>{props.hours}:{props.minutes}</div>;
ReactDOM.render(<Watch hours="Hello" minutes="World"/>,

          document.getElementById('app'));
```

Simple (stateless) React components

```
// Simple (stateless) React component
const Headline = () => {
  return <h1>React Cheat Sheet</h1>
}
const Intro = () => {
  return (<div>
    <Headline />
    <p>Welcome to the React world!</p>
  </div>)
}
const Intro = () => (
  <div>
    <Headline />
    <p>Welcome to the React world!</p>
  </div>
)
```

User-Defined Components Must Be Capitalized

Smart Component

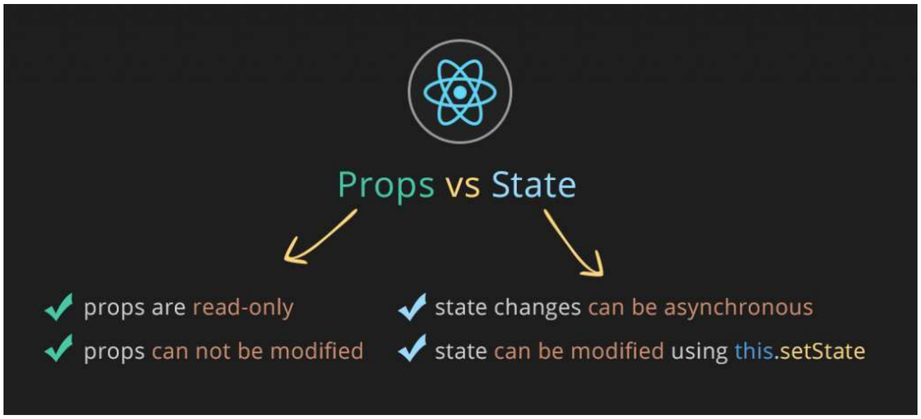
```
class App extends React.Component {
   render() {
      return (
         <h1>React Cheat Sheet</h1>
      )
   }
}
```

Example Component

```
import React, {Component }from "react";
class ExampleComponent extends Component {
constructor() {
super();
this.state = {
articles: [
{title: "React Redux Tutorial for Beginners", id: 1 },
{title: "Redux e React: cos'è Redux e come usarlo con React", id: 2 }
]
};
}
render() {
const {articles }=this.state;
return <ul>{articles.map(el => <li key={el.id}>{el.title}</li>)}</ul>;
}
}
```

```
class App extends React.Component {
  // fires before component is mounted
  constructor(props) {
    // makes this refer to this component
    super(props);

    // set local state
    this.state = {
      date: new Date()
    };
  }
  render() {
    return (
      <h1>
        It is {this.state.date.toLocaleTimeString()}.
      </h1>
    )
  }
}
```

State vs Props



Props vs State

✓ props are read-only

✓ props can not be modified

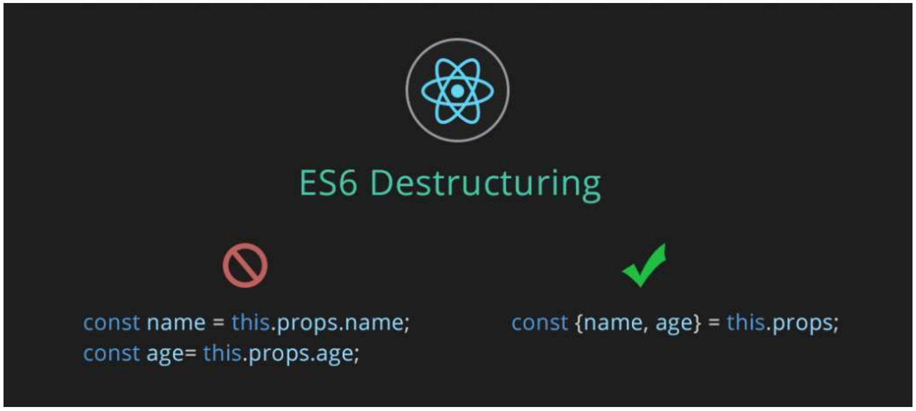
✓ state changes can be asynchronous

✓ state can be modified using this.setState

component that receives props

```
// Component that receives props
const Greetings = (props) => {
   return <p>You will love it {props.name}.</p>
}
<Greetings name={Peter}>
```

Destructuring



ES6 Destructuring

⊘

```
const name = this.props.name;
const age= this.props.age;
```

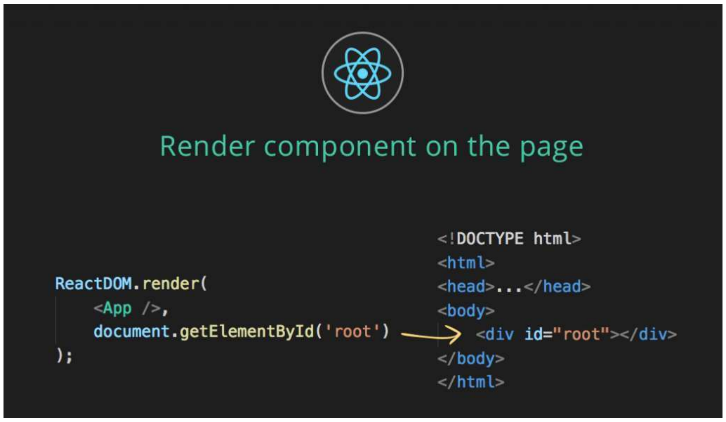
✓

```
const {name, age} = this.props;
```

```
const name = this.props.name;
const age = this.props.age;
const isLoggedIn = this.state.isLoggedIn;
const username = this.state.username;
```

```
const {name, age} = this.props;
const {isLoggedIn, username} = this.state;
```

Rendering



Render component on the page

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

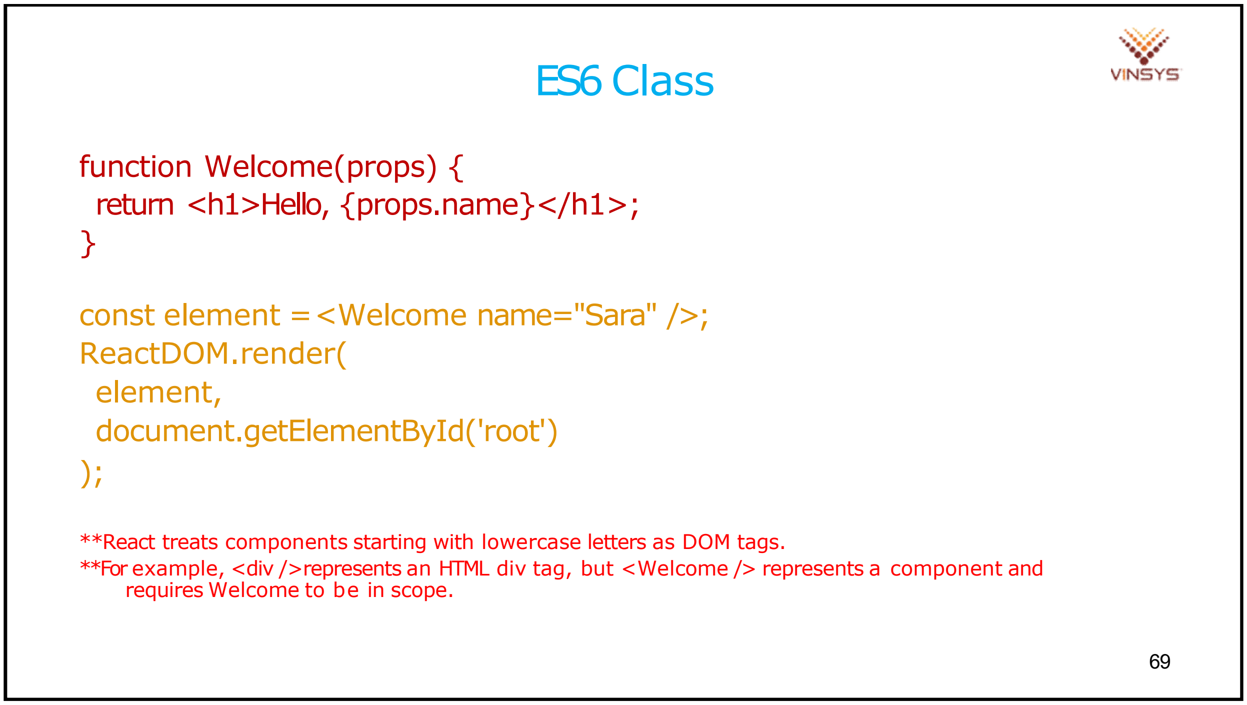
```
<!DOCTYPE html>
<html>
<head>...</head>
<body>
  <div id="root"></div>
</body>
</html>
```

```
import React from 'react'
import ReactDOM from 'react-dom';
import App from './js/components/App';

/ /  Render component into the
//DOM - only once per app
ReactDOM.render(
   <App />,
   document.getElementById('root')
);
```

Components and Props

- Components let you split the UI into independent, reusable pieces.
- The simplest way to define a component is to write a JavaScript function

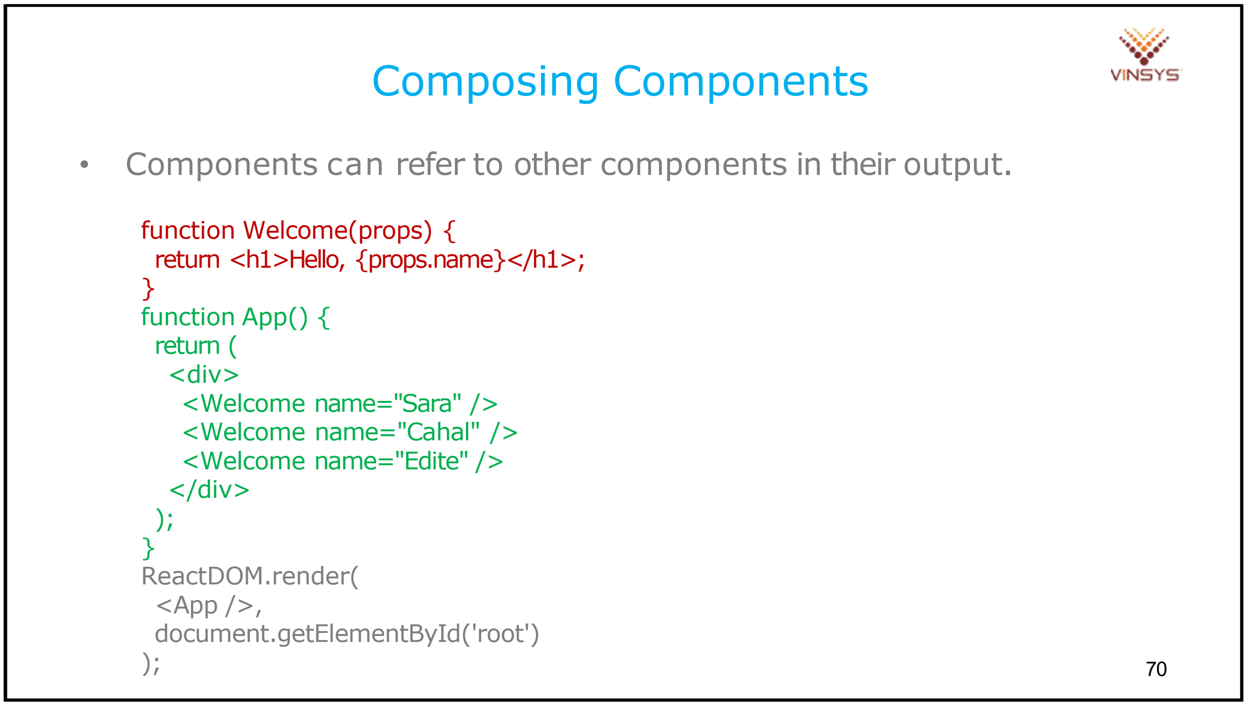
```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```



ES6 Class

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

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

**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.



Composing Components

- Components can refer to other components in their output.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

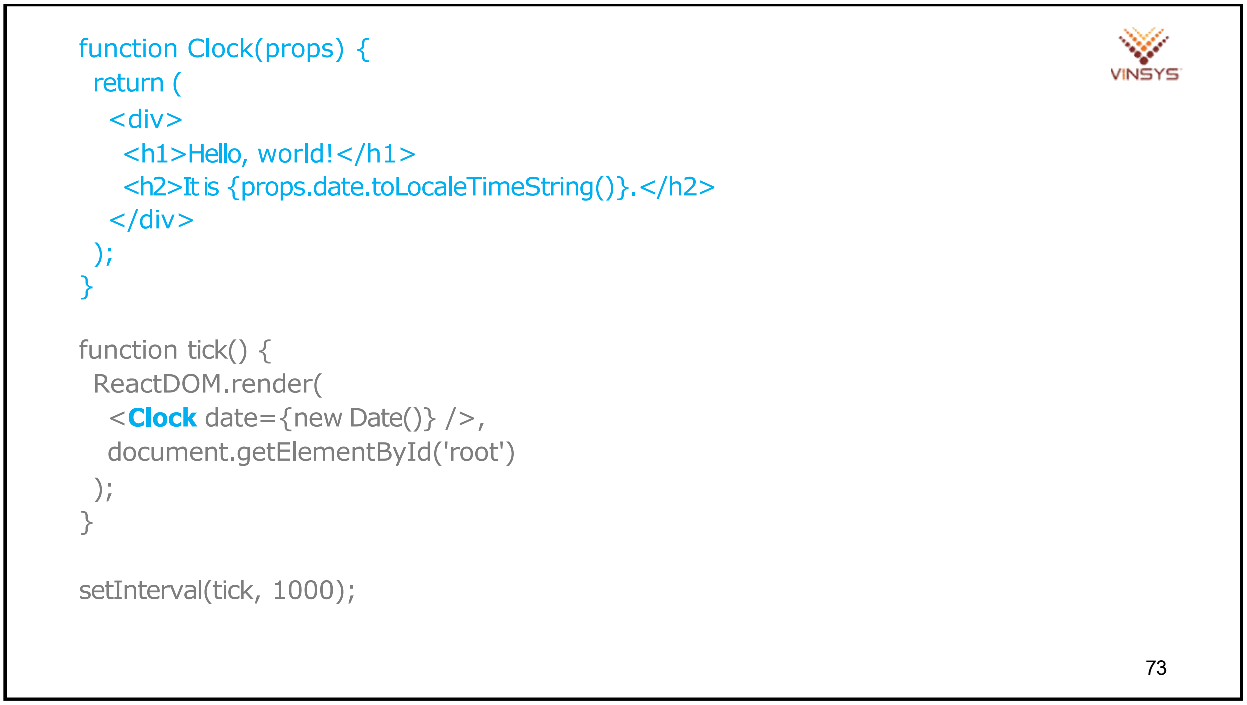
Pure Function

- Whether you declare a component as a function or a class, it must never modify its own props.

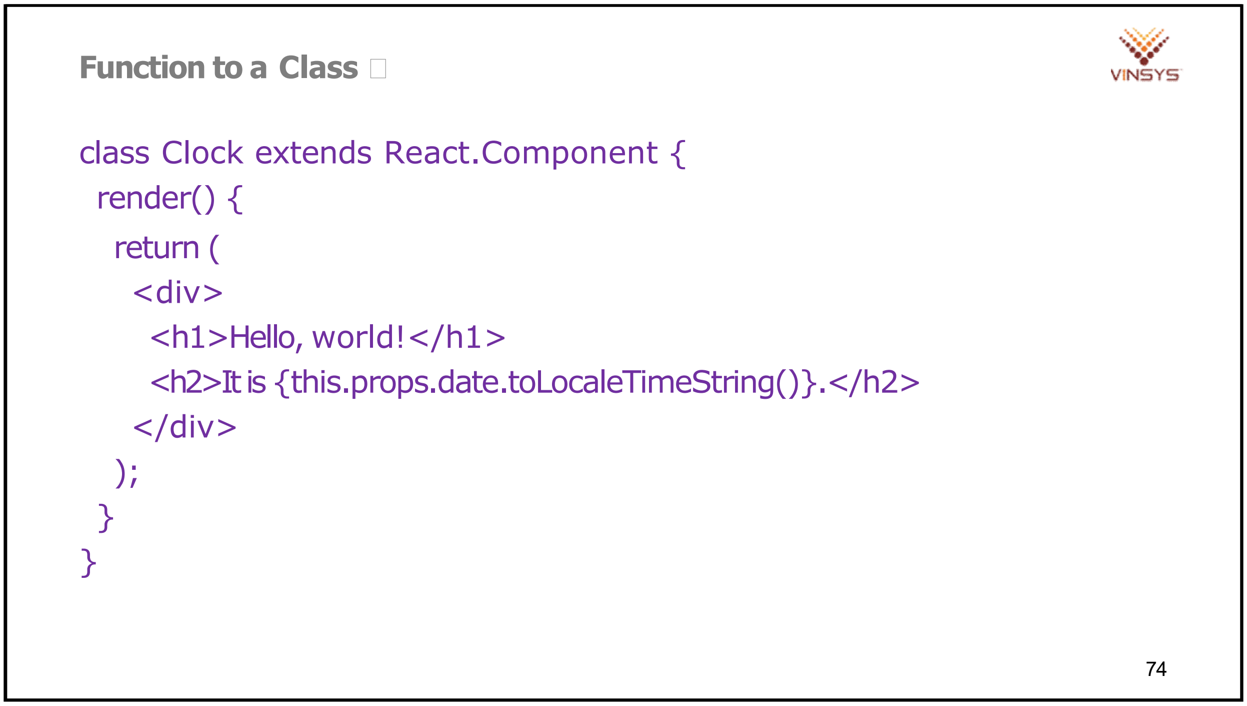
```
function sum(a, b) {
return a +b;
}

function withdraw(account, amount) {
account.total -=amount;
}
```

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

```
function tick() {
 const element =(
  <div>
   <h1>Hello, world!</h1>
   <h2>It is {new Date().toLocaleTimeString()}.</h2>
  </div>
 );
 ReactDOM.render(
  element,
  document.getElementById('root')
 );
}
setInterval(tick, 1000);
```



```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```



Function to a Class □

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Local State

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

 render() {
  return (
   <div>
    <h1>Hello, world!</h1>
    <h2>It is
   {this.state.date.toLocaleTimeString()}.</h2>
   </div>
  );
 }
}
```

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

Lifecycle - Mounting

- Methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- constructor()
- static getDerivedStateFromProps()
- render()
- componentDidMount()

- UNSAFE_componentWillMount()

Lifecycle - Updating

- static getDerivedStateFromProps(props, state)
- shouldComponentUpdate(nextProps, nextState)
- render()
- getSnapshotBeforeUpdate(prevProps, prevState)
- componentDidUpdate(prevProps, prevState, snapshot)

- UNSAFE_componentWillUpdate()
- UNSAFE_componentWillReceiveProps()

***componentDidUpdate() will not be invoked if shouldComponentUpdate() returns false.*

Lifecycle - Unmounting

- This method is called when a component is being removed from the DOM:

- componentWillUnmount()

Error Handling

- These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- static getDerivedStateFromError(error)
- componentDidCatch(error, info)

State Updates are Merged

- When you call setState(), React merges the object you provide into the current state.

- this.state = { posts: [], comments: [] };

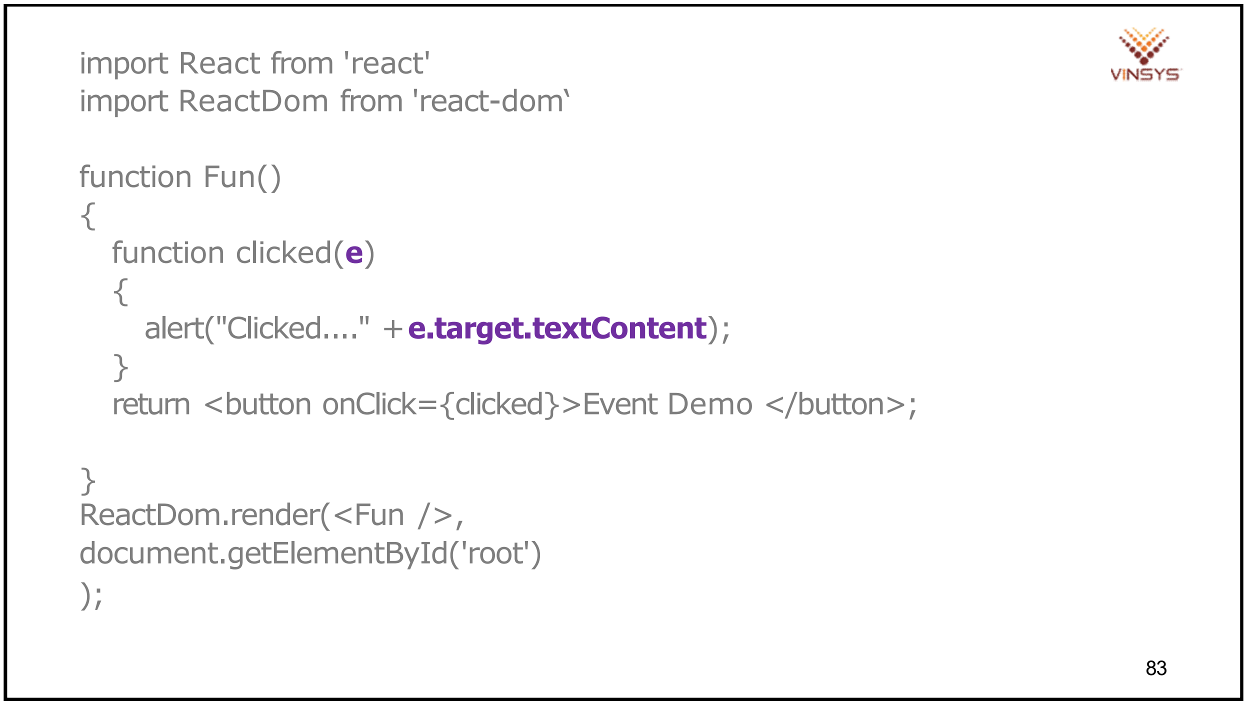
- this.setState({ posts: response.posts });

Events

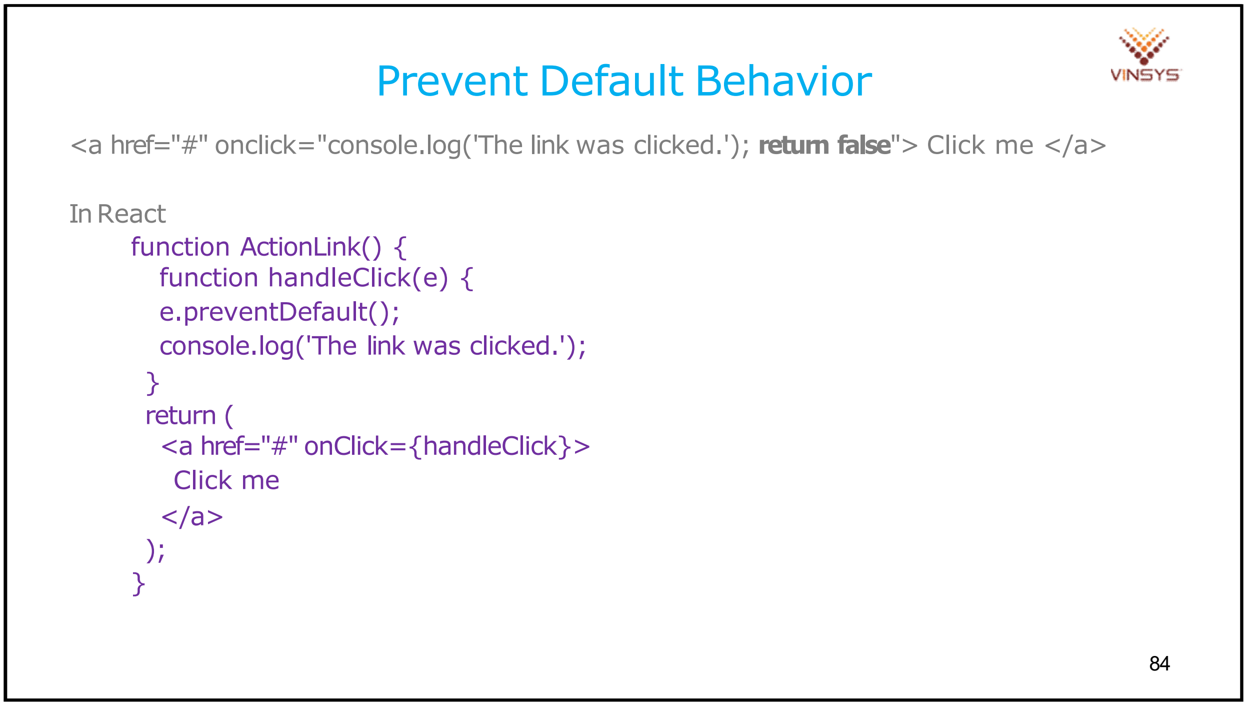
- Handling events with React elements is very similar to handling events on DOM elements.
- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

```
<button onclick="activateLasers()"> Activate Lasers </button>
<button onClick={activateLasers}> Activate Lasers </button>
```

```
import React from 'react'
import ReactDom from 'react-dom`

function Fun()
{
    function clicked()
    {
    alert("Clicked....");
    }
return <button onClick={clicked}>Event Demo </button>;
}
ReactDom.render(<Fun />, document.getElementById('root')
);
```



```
import React from 'react'
import ReactDom from 'react-dom`

function Fun()
{
   function clicked(e)
   {
      alert("Clicked...." + e.target.textContent);
   }
   return <button onClick={clicked}>Event Demo </button>;

}
ReactDom.render(<Fun />,
document.getElementById('root')
);
```

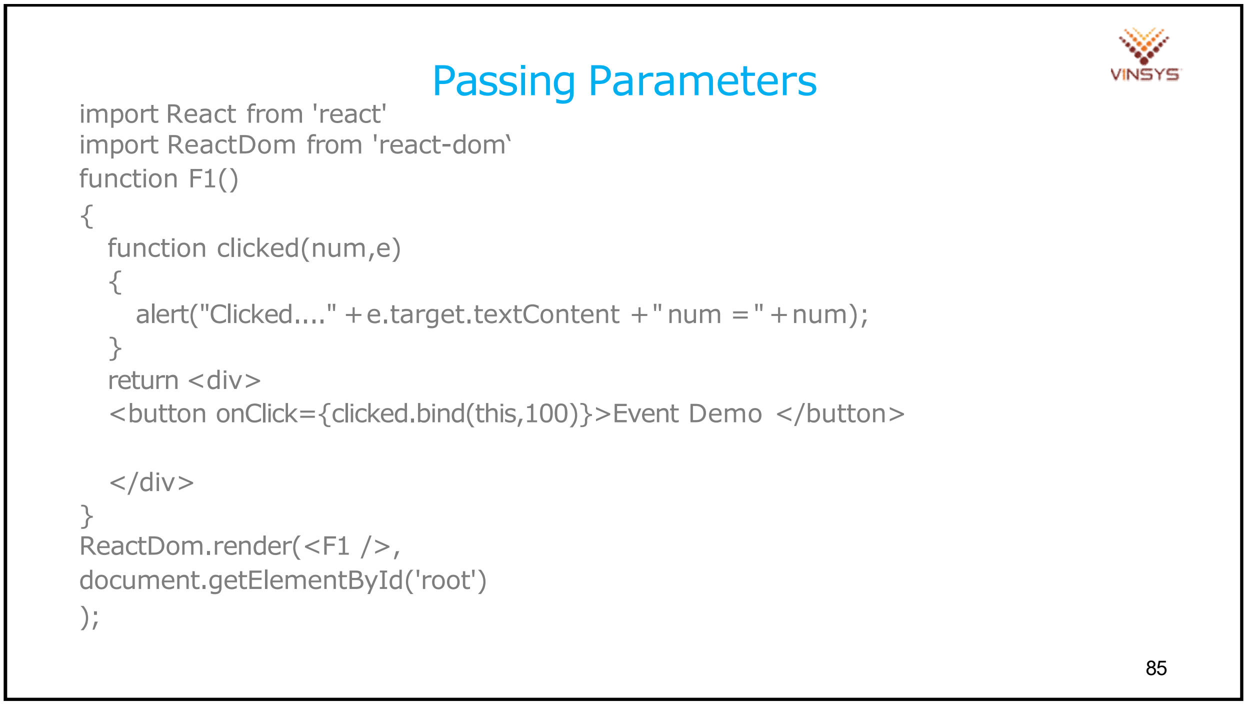


Prevent Default Behavior

<a href="#" onclick="console.log('The link was clicked.'); **return false**"> Click me </a>

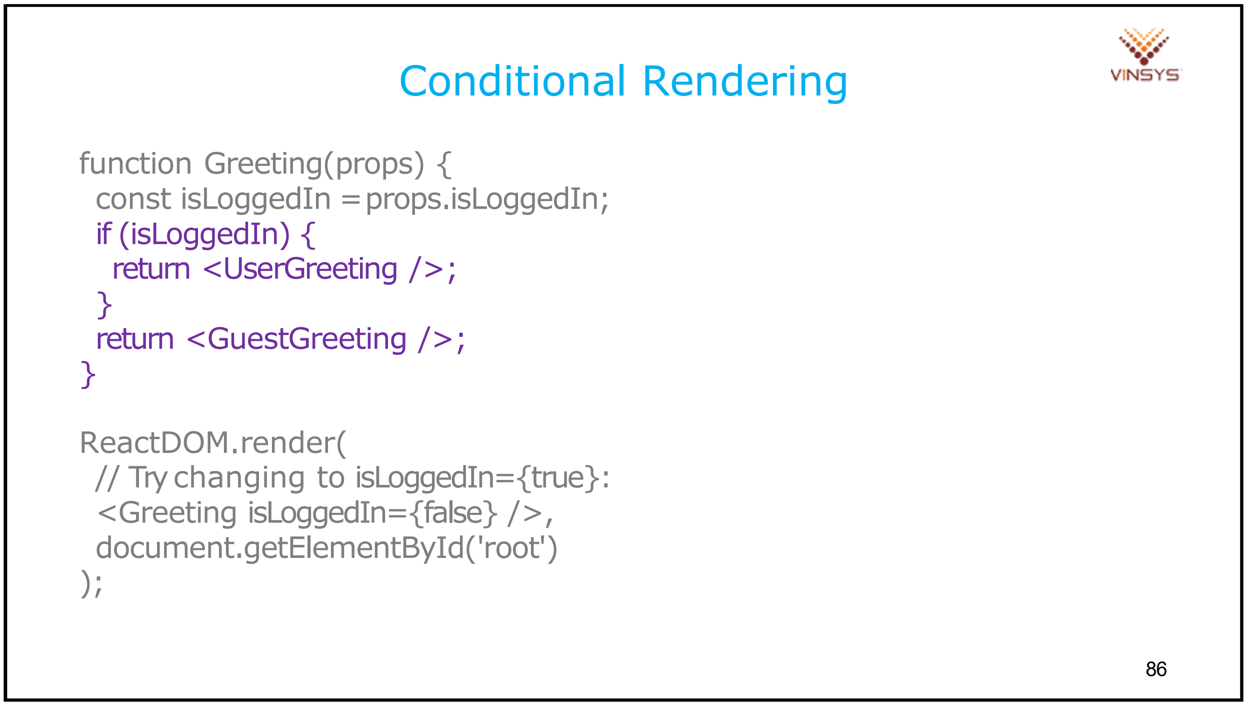
In React

```
      function ActionLink() {
         function handleClick(e) {
         e.preventDefault();
         console.log('The link was clicked.');
        }
        return (
          <a href="#" onClick={handleClick}>
            Click me
          </a>
        );
      }
```



Passing Parameters

```
import React from 'react'
import ReactDom from 'react-dom‘
function F1()
{
   function clicked(num,e)
   {
      alert("Clicked...." +e.target.textContent +" num =" +num);
   }
   return <div>
   <button onClick={clicked.bind(this,100)}>Event Demo </button>

   </div>
}
ReactDom.render(<F1 />,
document.getElementById('root')
);
```



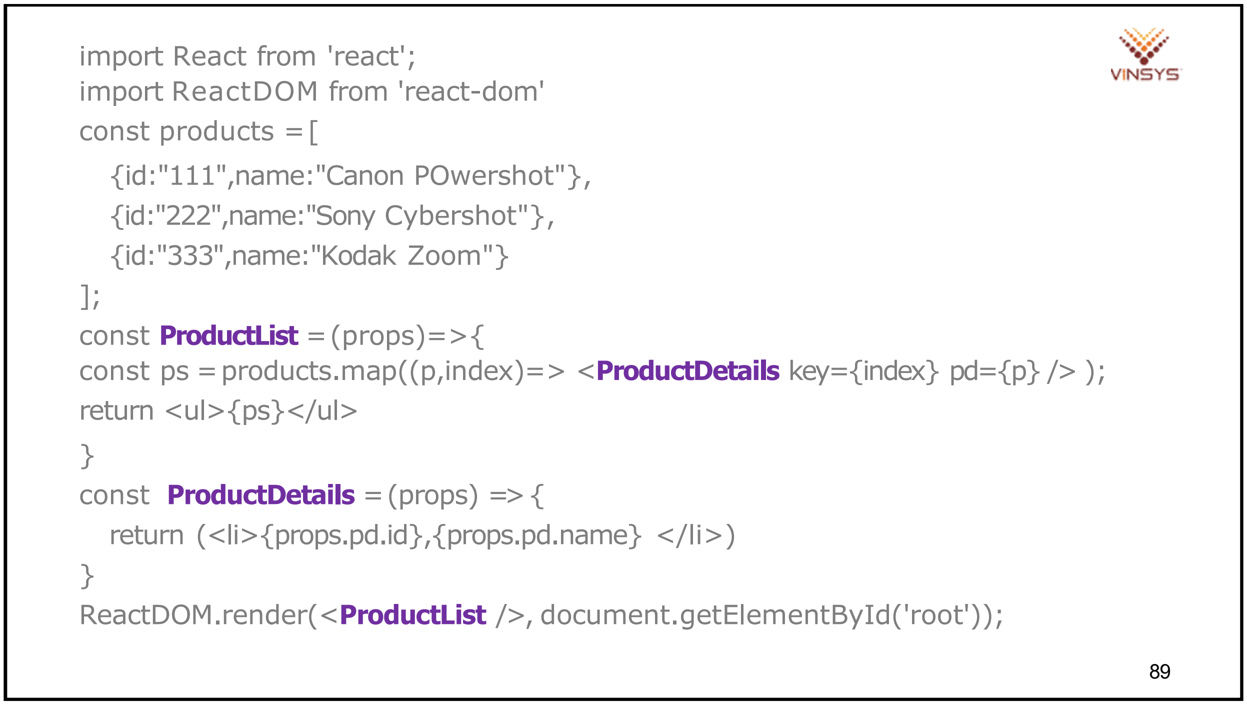
Conditional Rendering

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

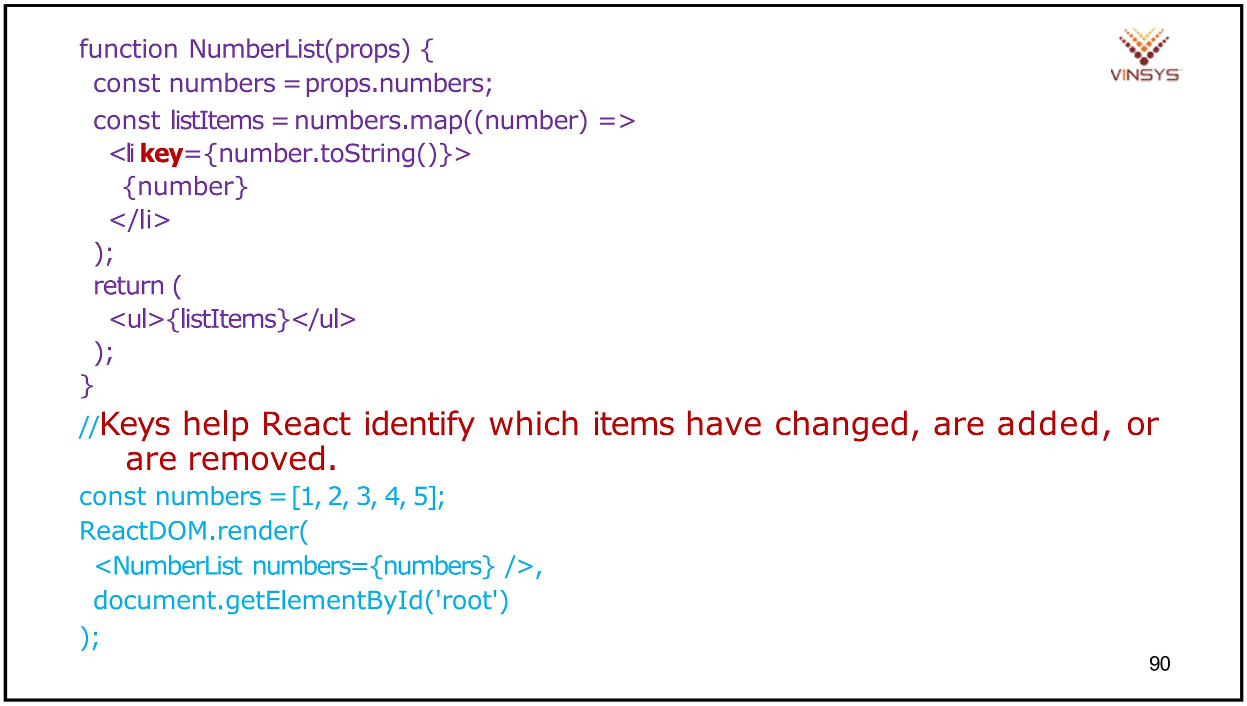
Lists

```
import React from 'react'
import ReactDOM from 'react-dom'
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
<li>{number}</li>
);

ReactDOM.render(
<ul>{listItems}</ul>,
document.getElementById('root')
);
```

```
var Products = () => {
   var names = ["Sony Cybershot", "Canon Powershot", "Kodak Zoom"];

   var namesList = names.map(function(name){
                return <li>{name}</li>;
               })

   return  <ul>{ namesList }</ul>
}
ReactDOM.render(
   <Products />,
   document.getElementById('root')
);
```



```
import React from 'react';
import ReactDOM from 'react-dom'
const products = [
   {id:"111",name:"Canon POwershot"},
   {id:"222",name:"Sony Cybershot"},
   {id:"333",name:"Kodak Zoom"}
];
const ProductList = (props)=>{
const ps = products.map((p,index)=> <ProductDetails key={index} pd={p} /> );
return <ul>{ps}</ul>
}
const  ProductDetails = (props) => {
   return  (<li>{props.pd.id},{props.pd.name}  </li>)
}
ReactDOM.render(<ProductList />, document.getElementById('root'));
```



```
function NumberList(props) {
 const numbers = props.numbers;
 const listItems = numbers.map((number) =>
   <li key={number.toString()}>
     {number}
   </li>
 );
 return (
   <ul>{listItems}</ul>
 );
}
//Keys help React identify which items have changed, are added, or
    are removed.
const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
 <NumberList numbers={numbers} />,
 document.getElementById('root')
);
```

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

Form

- Default HTML form

```
<form>
<label> Name: <input type="text" name="name" />
</label> <input type="submit" value="Submit" />
</form>
```

- Form elements such as <input>, <textarea>, and <select> typically maintain their own state

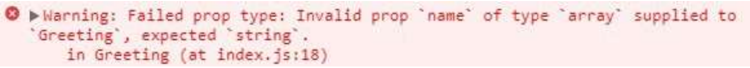
```
return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input type="text" value={this.state.value} onChange={this.handleChange} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
```

```
import React from 'react';
import ReactDOM from 'react-dom';
class SelectComp extends React.Component {
   cities = ["Pune","Mumbai","Delhi","Bangalore"]
    handleChange = (event) => {alert(event.target.value);}
     render() {
        return (
            <form >
               <select name="city" value="Pune" onChange={this.handleChange}>
                {
                   this.cities.map((city)=> <option value={city}>{city}</option>)
                }
               </select>
            </form>
        )
     }
}
ReactDOM.render(<SelectComp />, document.getElementById('root'));
```

PropTypes

- PropTypes exports a range of validators that can be used to make sure the data you receive is valid.

```
import PropTypes from 'prop-types';
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}
Greeting.propTypes = {
  name: PropTypes.string
};
```



```
▶Warning: Failed prop type: Invalid prop `name` of type `array` supplied to
`Greeting`, expected `string`.
    in Greeting (at index.js:18)
```

```
ReactDOM.render(<Greeting name={[100]} />,document.getElementById('root'));
```

- PropTypes.array
- PropTypes.bool
- PropTypes.fun
- PropTypes.number
- PropTypes.object
- PropTypes.string
- PropTypes.symbol

Default Prop Values

- Define default values for your props

```
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    ); }}
// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};
// Renders "Hello, Stranger":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

Mandatory

```
Person.propTypes = {
  firstName:PropTypes.string.isRequired,
  lastName:PropTypes.string.isRequired,
  country:PropTypes.string
};
```

Refs

- Refs provide a way to access DOM nodes or React elements created in the render method.

When to Use Refs

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Creating Refs

- Refs are created using React.createRef() and attached to React elements via the ref attribute.
- Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
this.myRef = React.createRef();
```

Adding a Ref to a DOM Element

```
<input type="text‟ ref={this.textInput} />
```

Accessing Refs

- When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

```
const node = this.myRef.current;
```

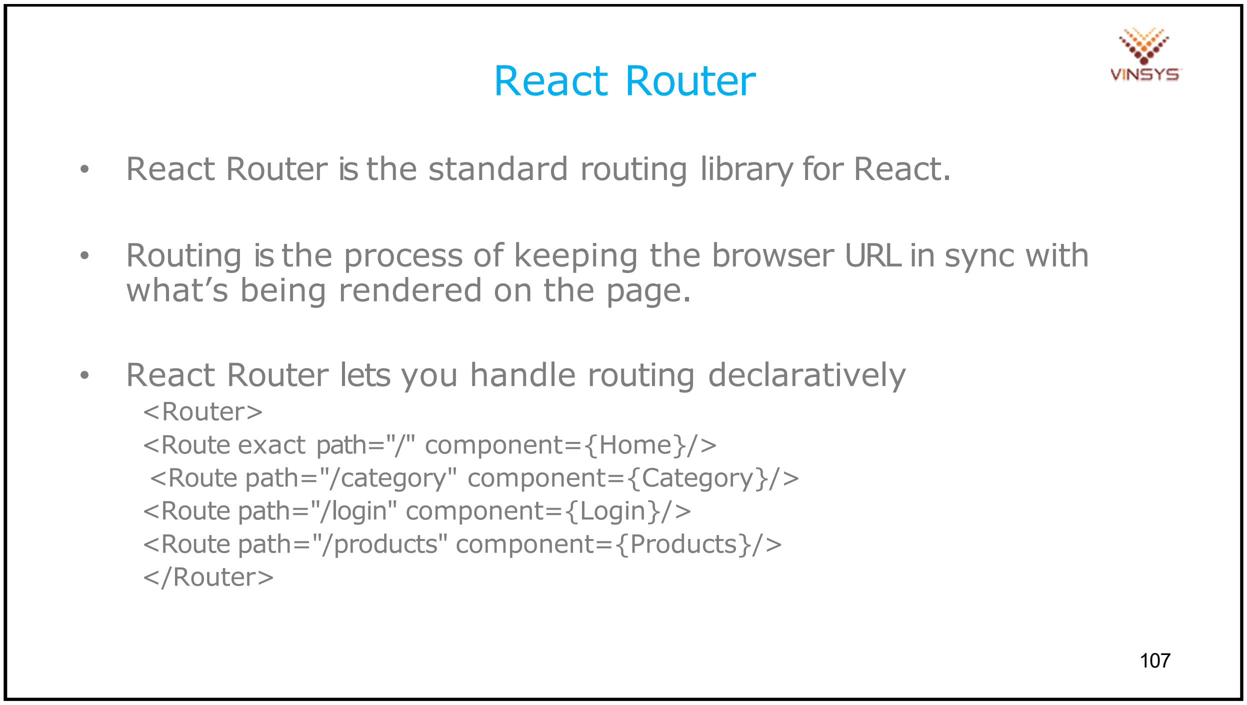
value of the ref

- When the ref attribute is used on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.
- When the ref attribute is used on a custom class component, the ref object receives the mounted instance of the component as its current.
- You may not use the ref attribute on function components because they don’t have instances.

Callback Refs

- Instead of passing a ref attribute created by createRef(), you pass a function.
- The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.
- Refs are guaranteed to be up-to-date before componentDidMount or componentDidUpdate fires.

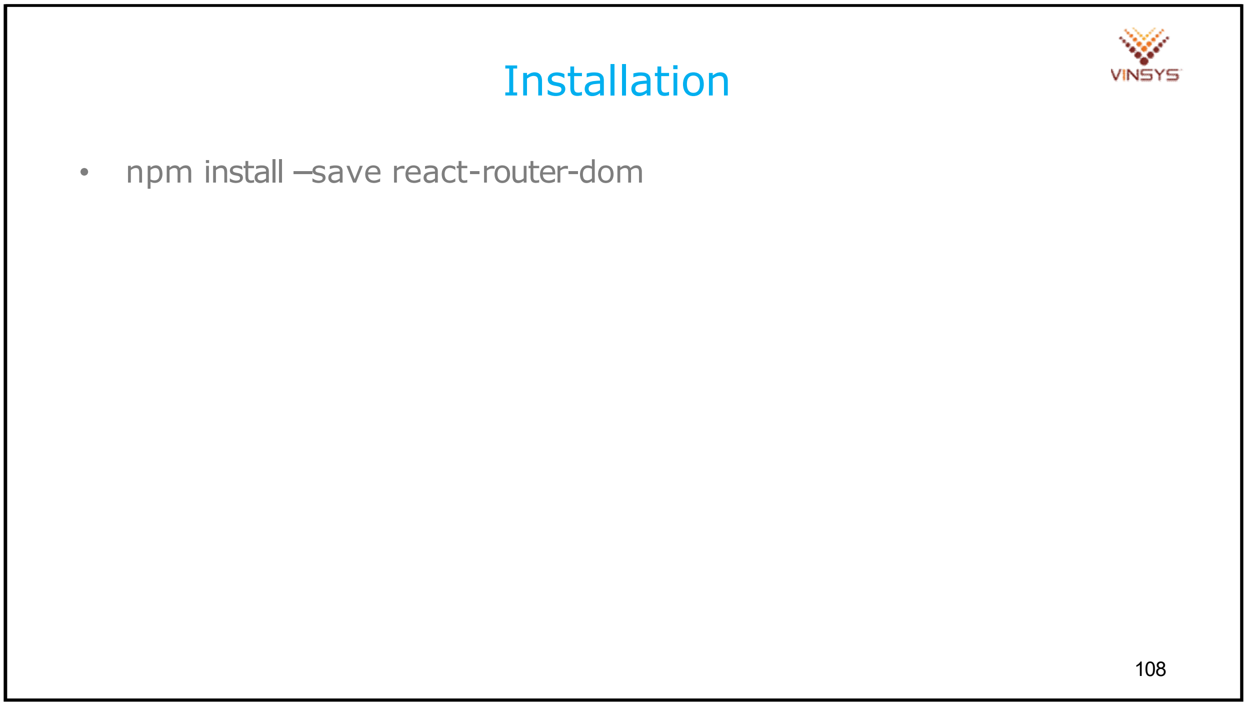
```
this.setTextInputRef = element => {
    this.textInput = element;
  };


<input type="text“  ref={this.setTextInputRef} />
```



React Router

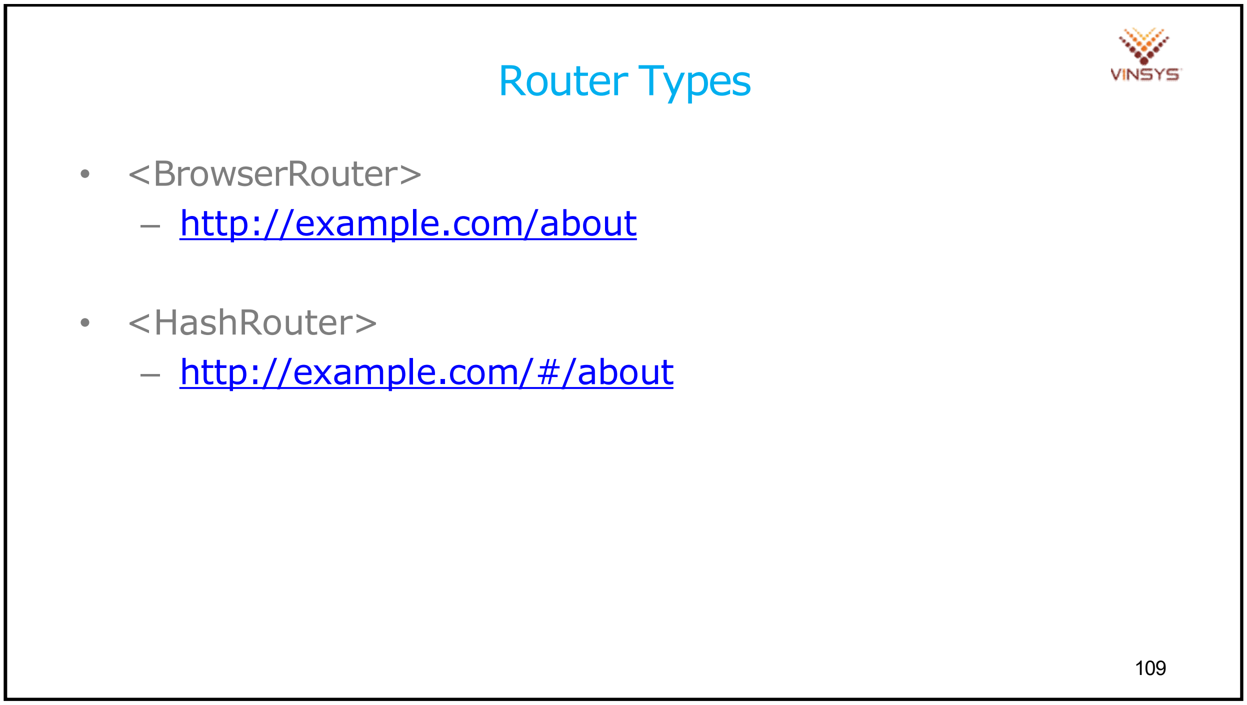
- React Router is the standard routing library for React.
- Routing is the process of keeping the browser URL in sync with what’s being rendered on the page.
- React Router lets you handle routing declaratively

```
<Router>
<Route exact path="/" component={Home}/>
 <Route path="/category" component={Category}/>
<Route path="/login" component={Login}/>
<Route path="/products" component={Products}/>
</Router>
```



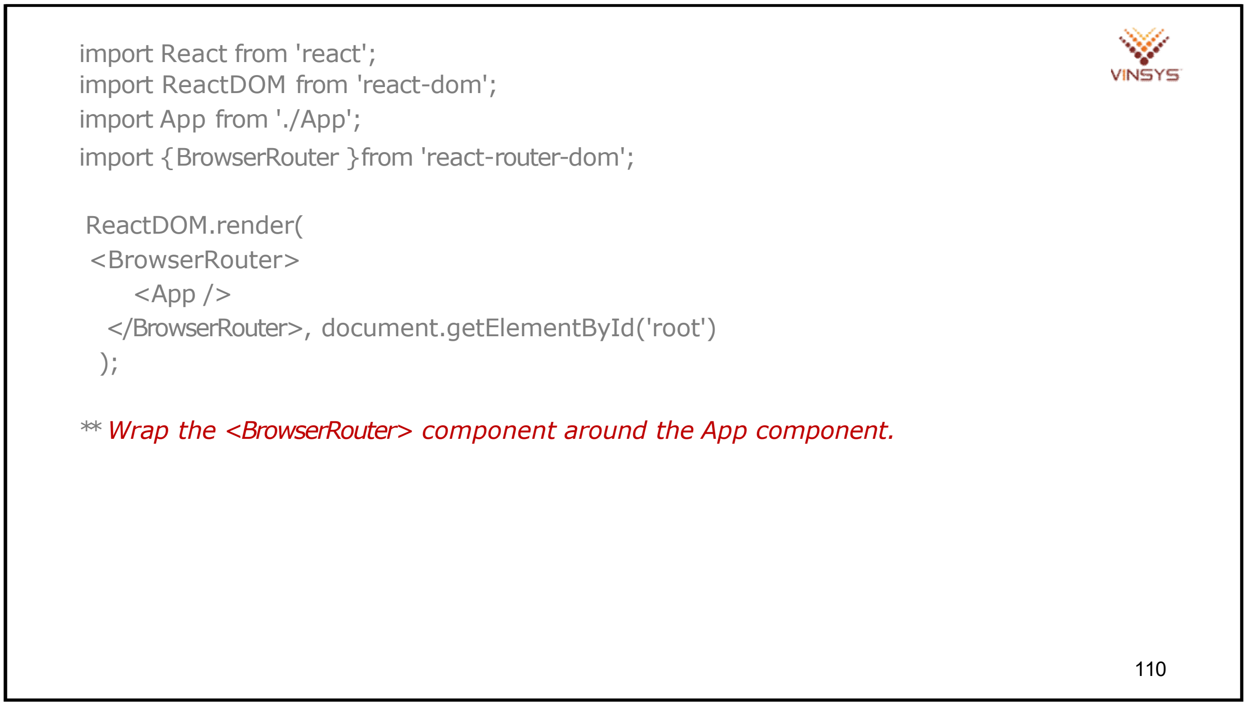
Installation

- npm install –save react-router-dom



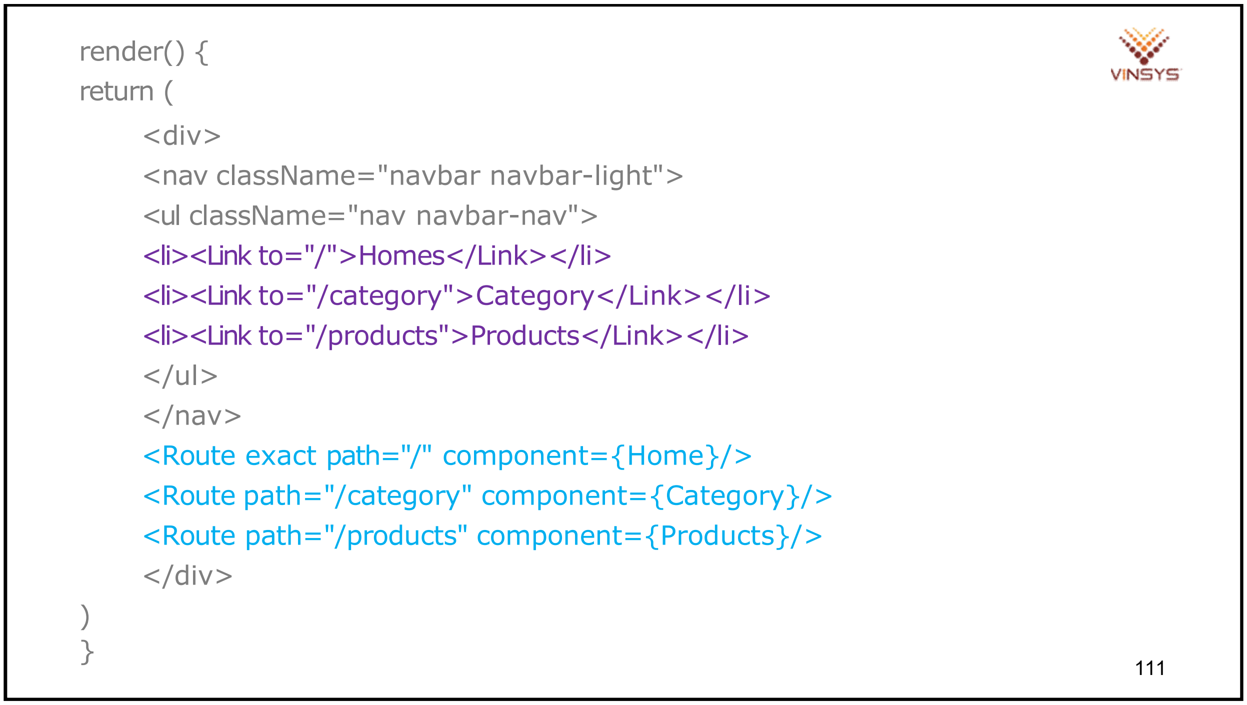
Router Types

- <BrowserRouter>
  - http://example.com/about
- <HashRouter>
  - http://example.com/#/about

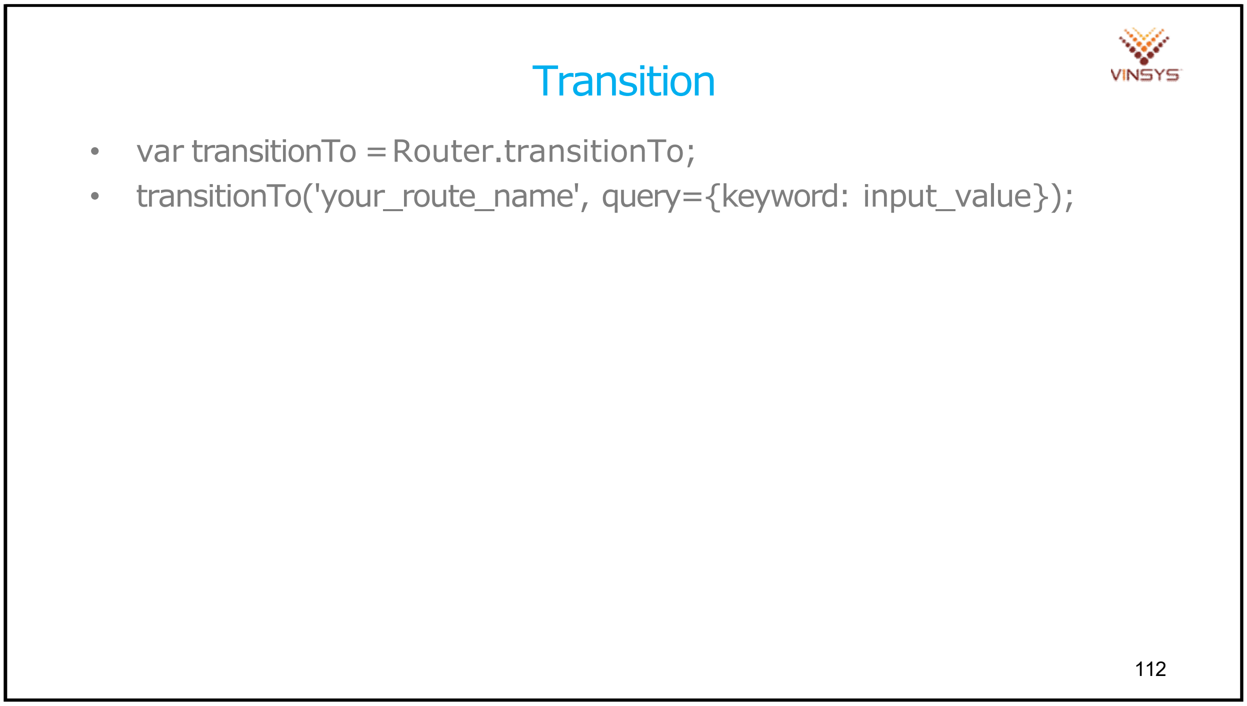


```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import {BrowserRouter }from 'react-router-dom';

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

** *Wrap the <BrowserRouter> component around the App component.*

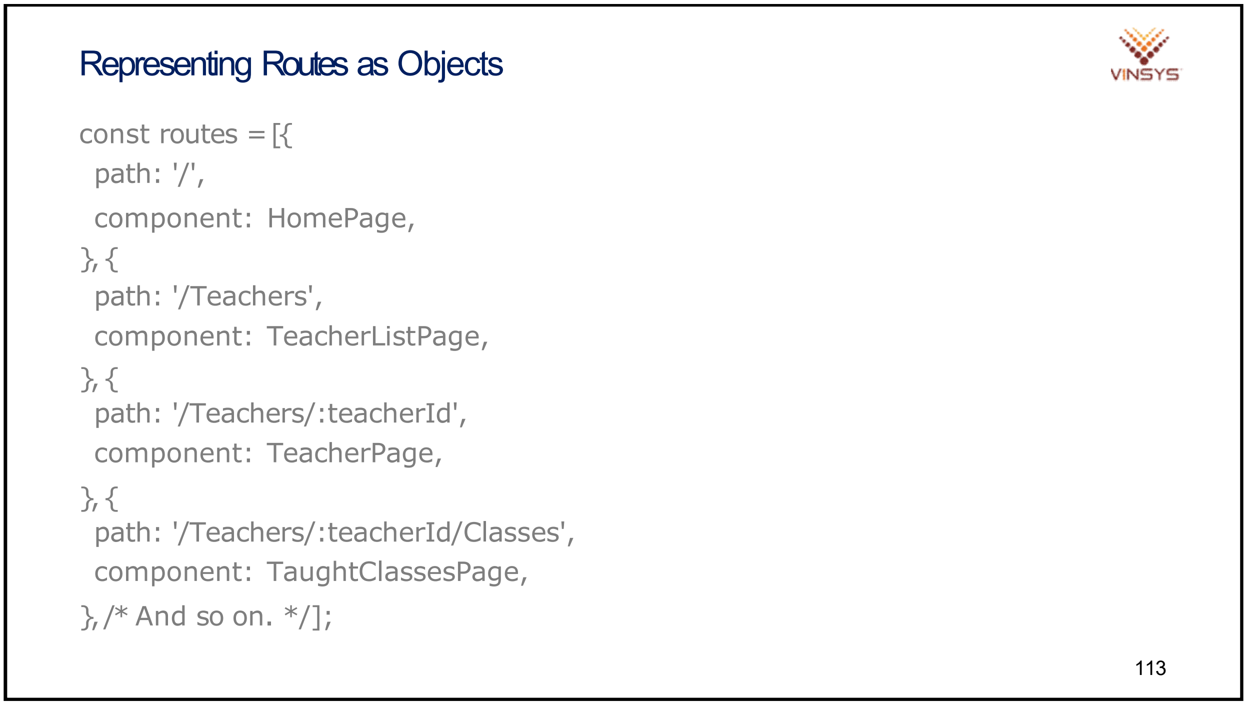


```
render() {
return (
      <div>
      <nav className="navbar navbar-light">
      <ul className="nav navbar-nav">
      <li><Link to="/">Homes</Link></li>
      <li><Link to="/category">Category</Link></li>
      <li><Link to="/products">Products</Link></li>
      </ul>
      </nav>
      <Route exact path="/" component={Home}/>
      <Route path="/category" component={Category}/>
      <Route path="/products" component={Products}/>
      </div>
)
}
```



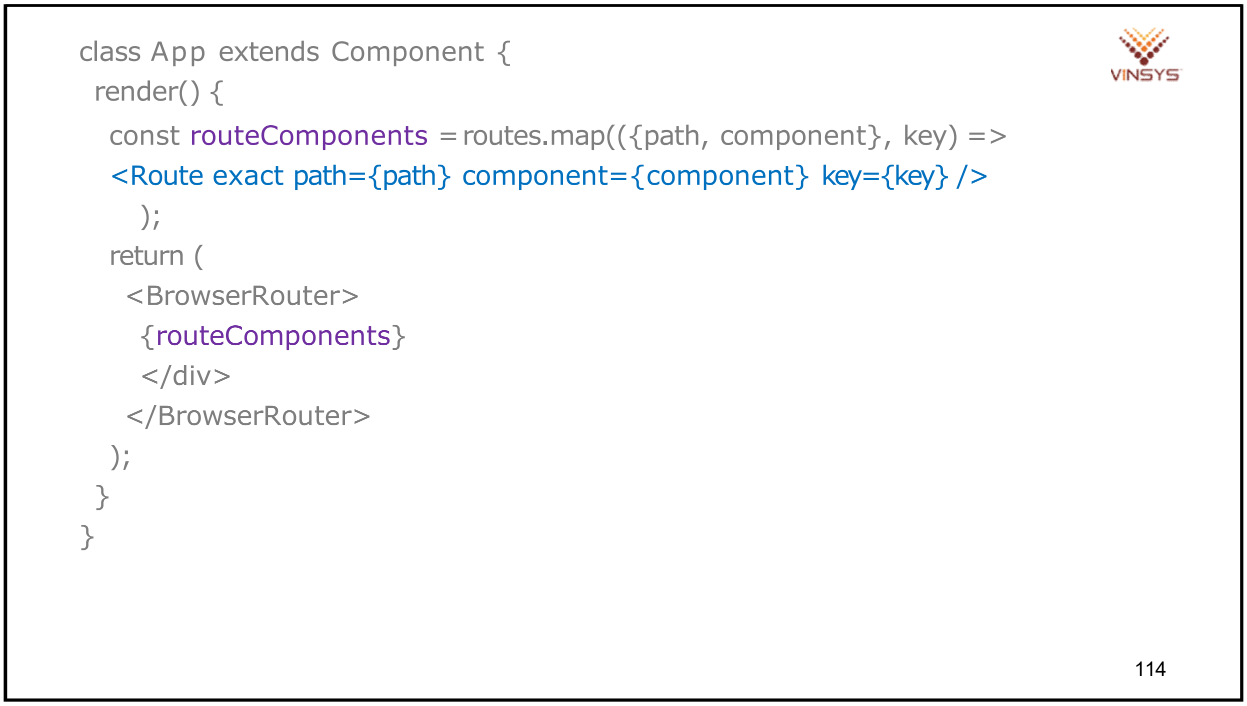
Transition

- var transitionTo = Router.transitionTo;
- transitionTo('your_route_name', query={keyword: input_value});

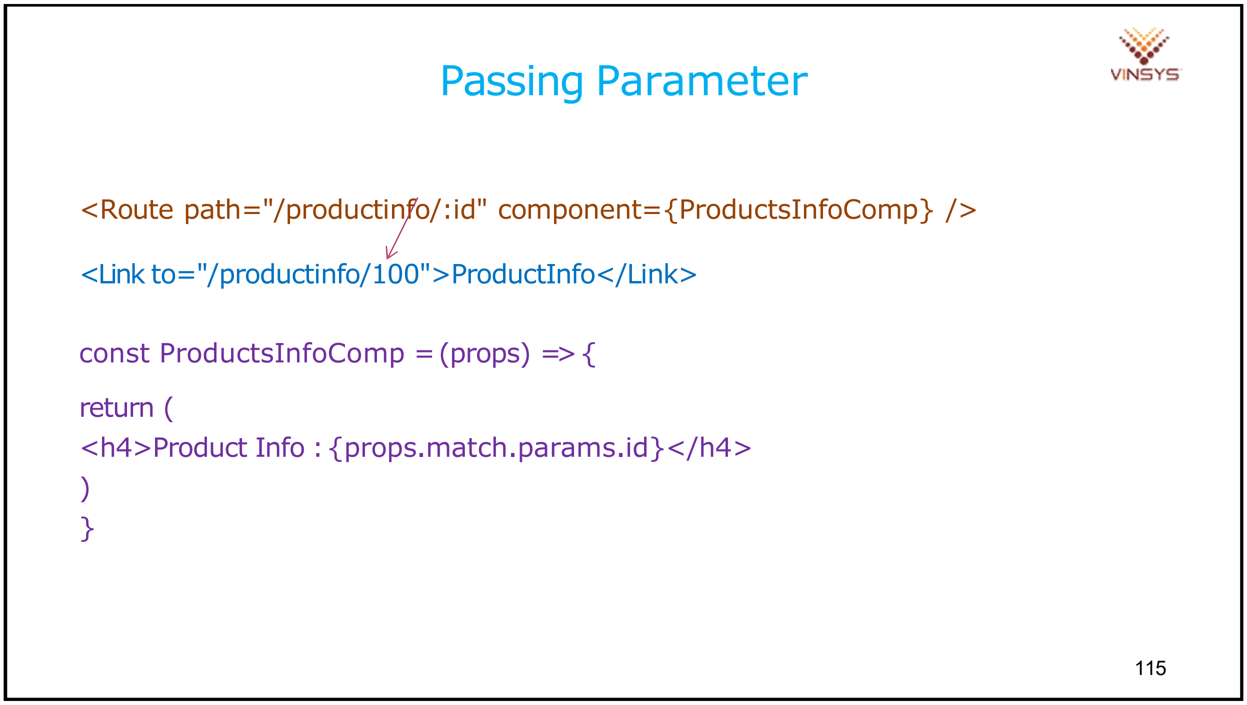


Representing Routes as Objects

```
const routes = [{
 path: '/',
 component: HomePage,
}, {
 path: '/Teachers',
 component: TeacherListPage,
}, {
 path: '/Teachers/:teacherId',
 component: TeacherPage,
}, {
 path: '/Teachers/:teacherId/Classes',
 component: TaughtClassesPage,
}, /* And so on. */];
```

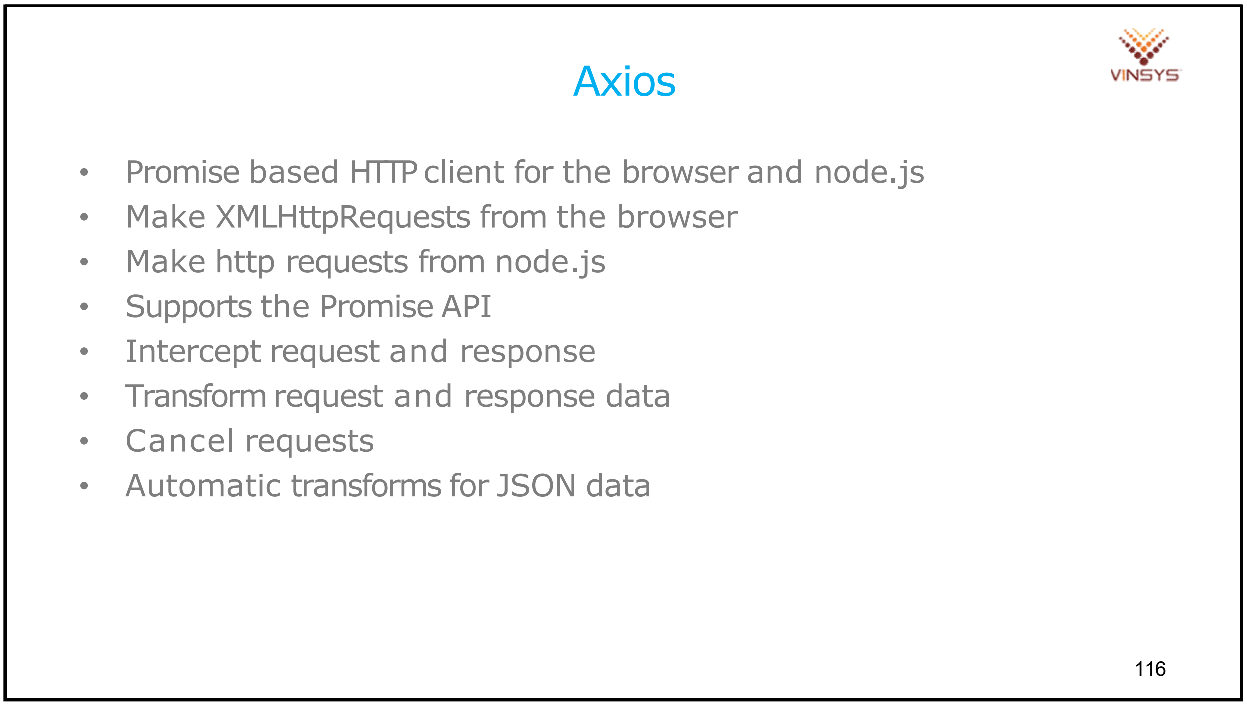


```
class App extends Component {
 render() {
  const routeComponents = routes.map(({path, component}, key) =>
  <Route exact path={path} component={component} key={key} />
    );
  return (
   <BrowserRouter>
    {routeComponents}
    </div>
   </BrowserRouter>
  );
 }
}
```



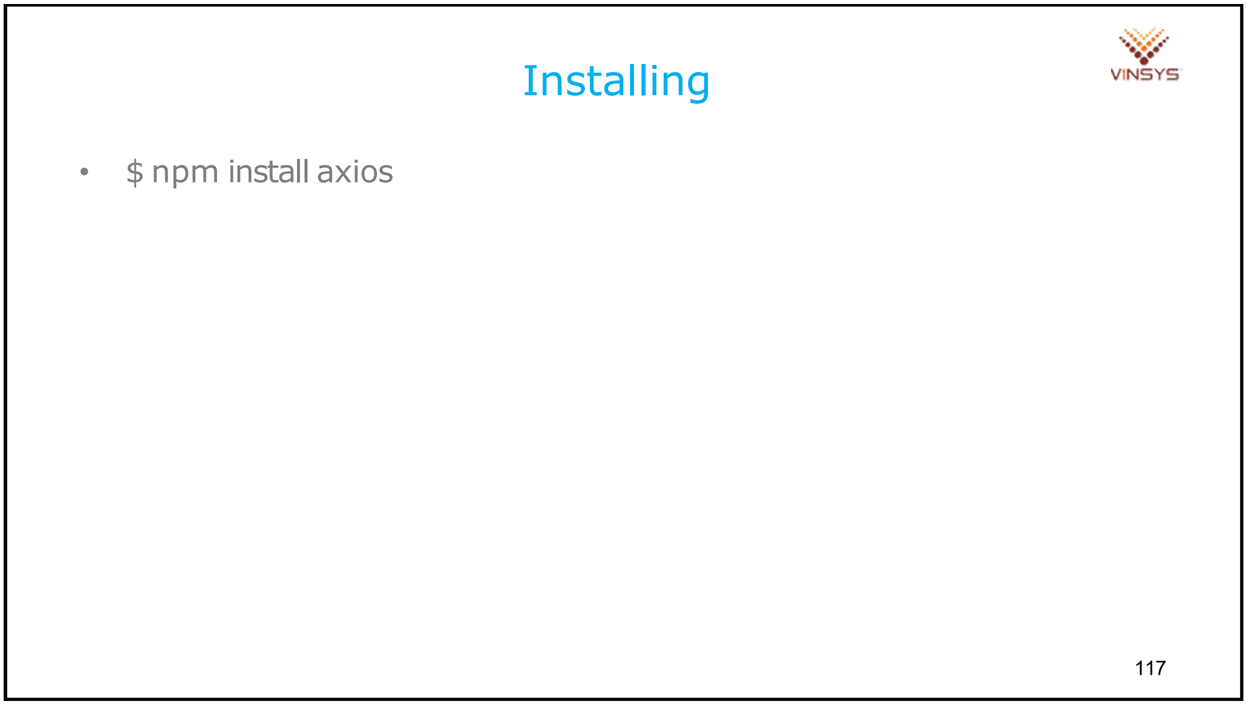
Passing Parameter

```
<Route path="/productinfo/:id" component={ProductsInfoComp} />

<Link to="/productinfo/100">ProductInfo</Link>

const ProductsInfoComp = (props) => {

return (
<h4>Product Info : {props.match.params.id}</h4>
)
}
```



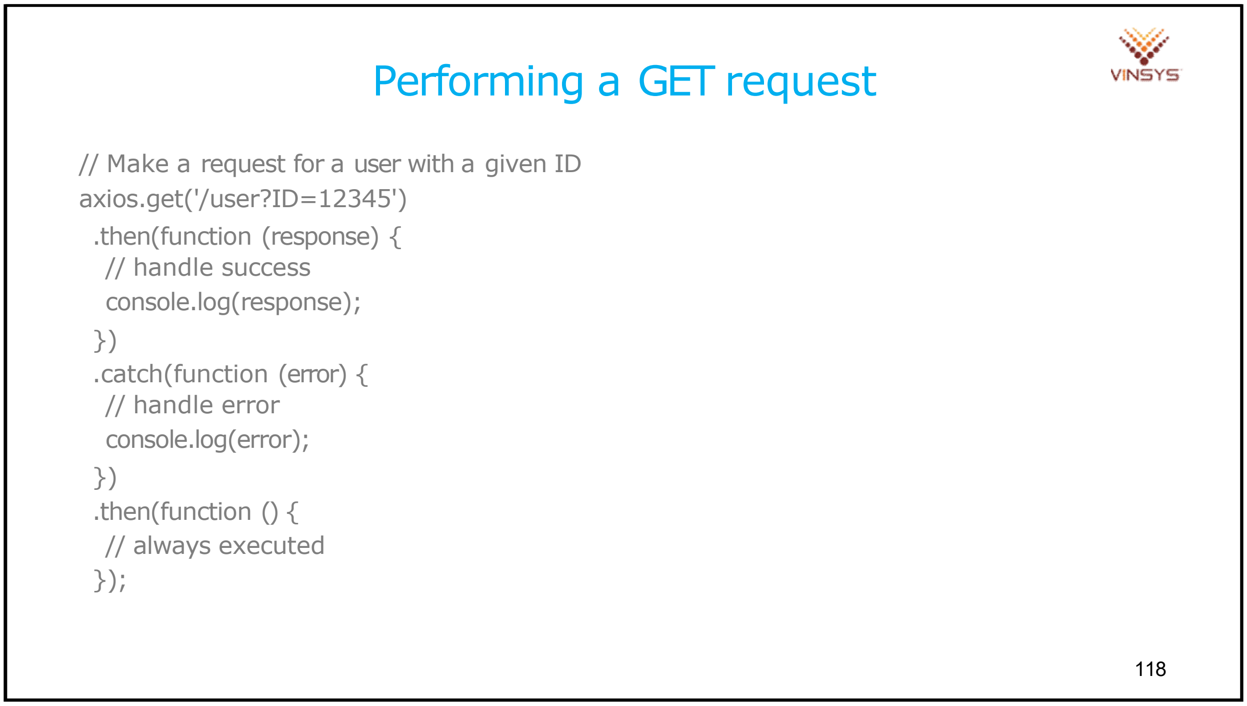
Axios

- Promise based HTTP client for the browser and node.js
- Make XMLHttpRequests from the browser
- Make http requests from node.js
- Supports the Promise API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data



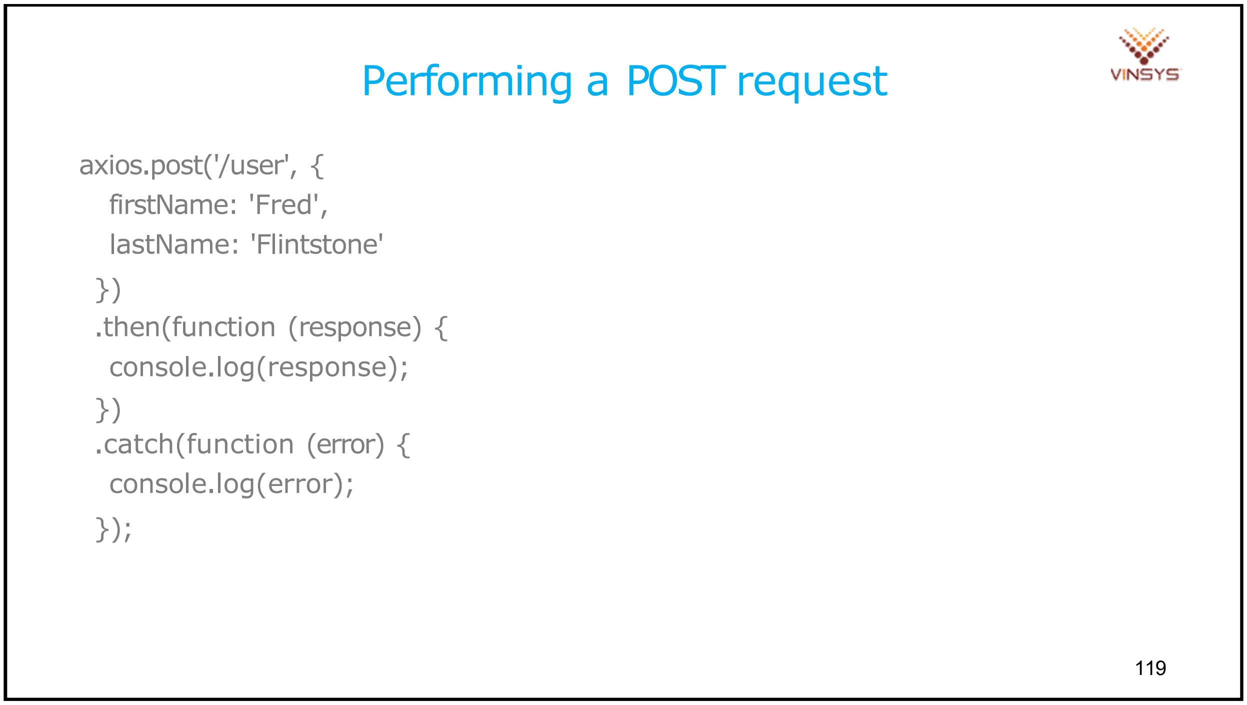
Installing

- $ npm install axios



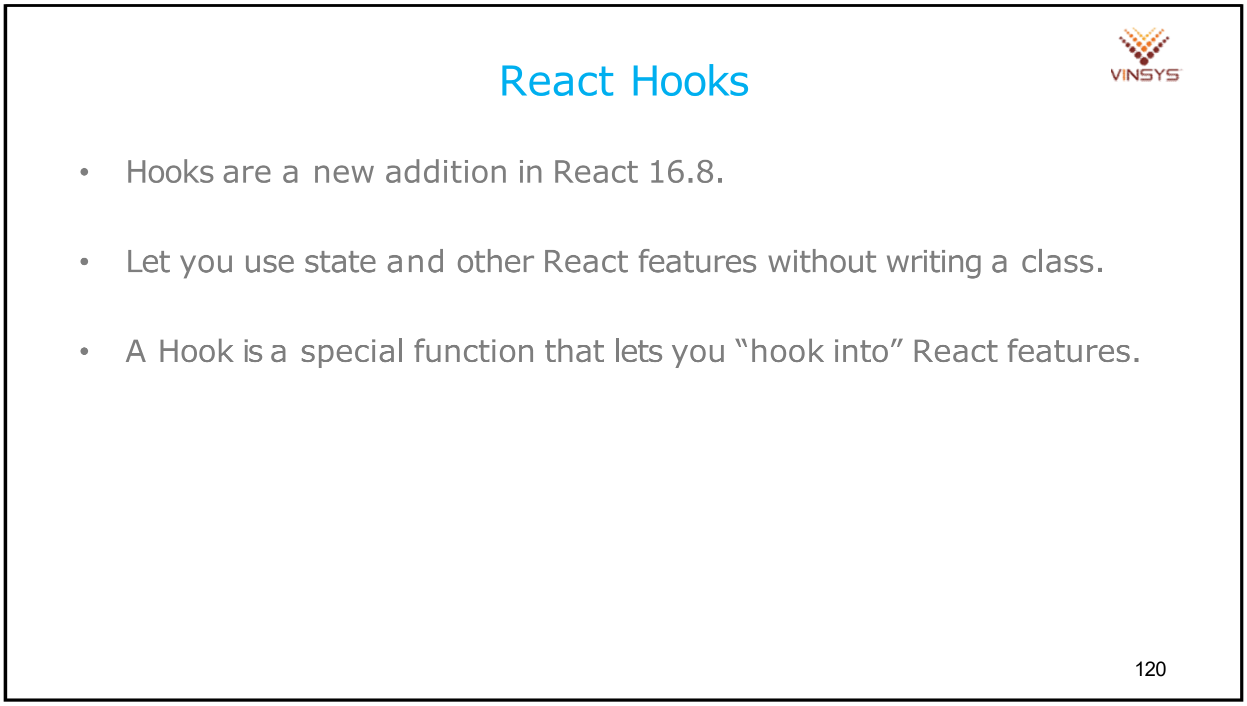
Performing a GET request

```
// Make a request for a user with a given ID
axios.get('/user?ID=12345')
 .then(function (response) {
  // handle success
  console.log(response);
 })
 .catch(function (error) {
  // handle error
  console.log(error);
 })
 .then(function () {
  // always executed
 });
```



Performing a POST request

```
axios.post('/user', {
  firstName: 'Fred',
  lastName: 'Flintstone'
 })
 .then(function (response) {
  console.log(response);
 })
 .catch(function (error) {
  console.log(error);
 });
```



React Hooks

- Hooks are a new addition in React 16.8.
- Let you use state and other React features without writing a class.
- A Hook is a special function that lets you “hook into” React features.

Hooks

- useState
- useEffect
  - Allows you to register a function which executes after the current render cycle.
- useContext
- useReducer
- useCallback
  - Allows you to prevent the re-creation of a function
- useMemo
  - exporting component asReact.Memo() function will render only if there will be a change in props or state.
- useRef
- useImperativeHandle
- useLayoutEffect
- useDebugValue

useEffect

- By default, effects run after every completed render,

- Cleaning

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

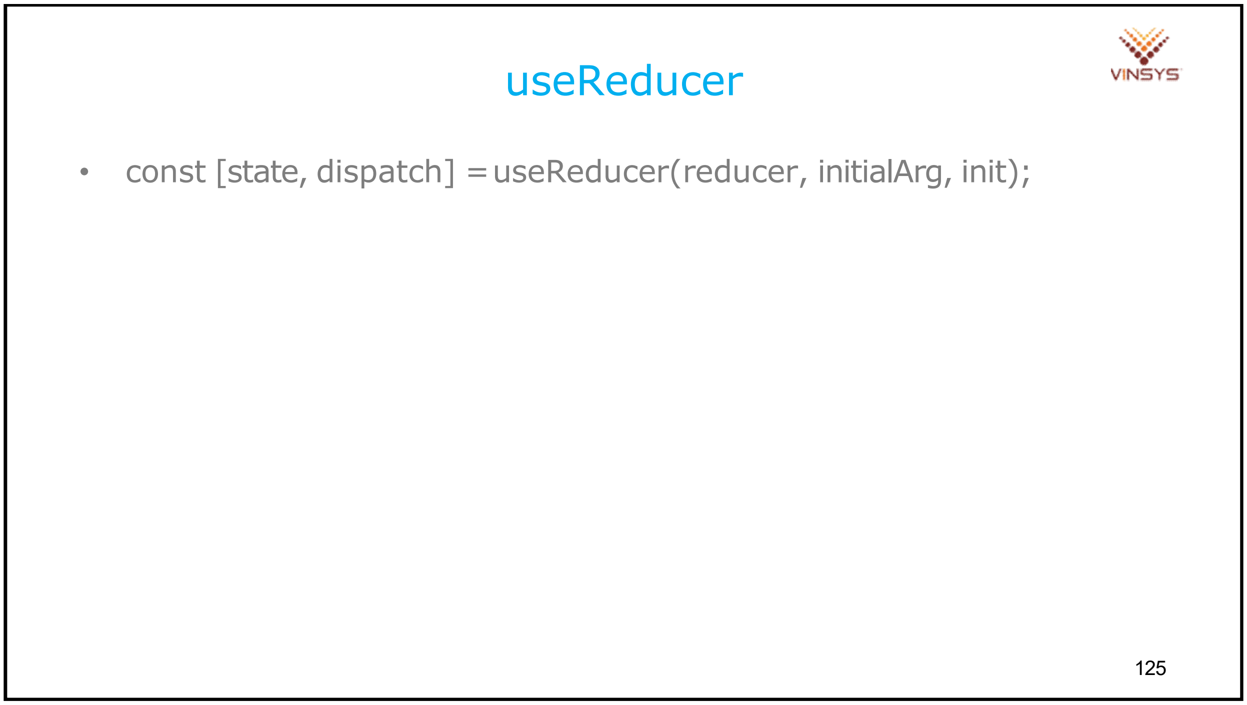
- Conditionally firing an effect

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

//don’t need to create a new subscription on every update, only if the source prop has changed.

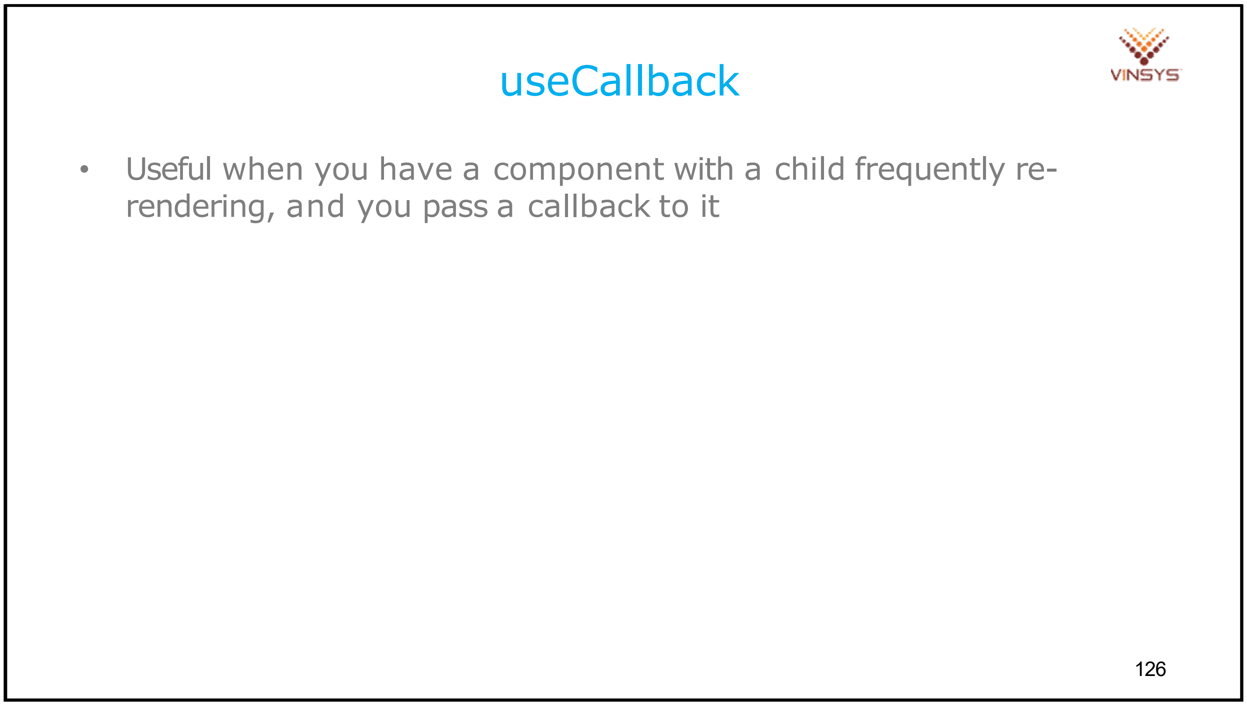
useContext

- const value = useContext(MyContext);



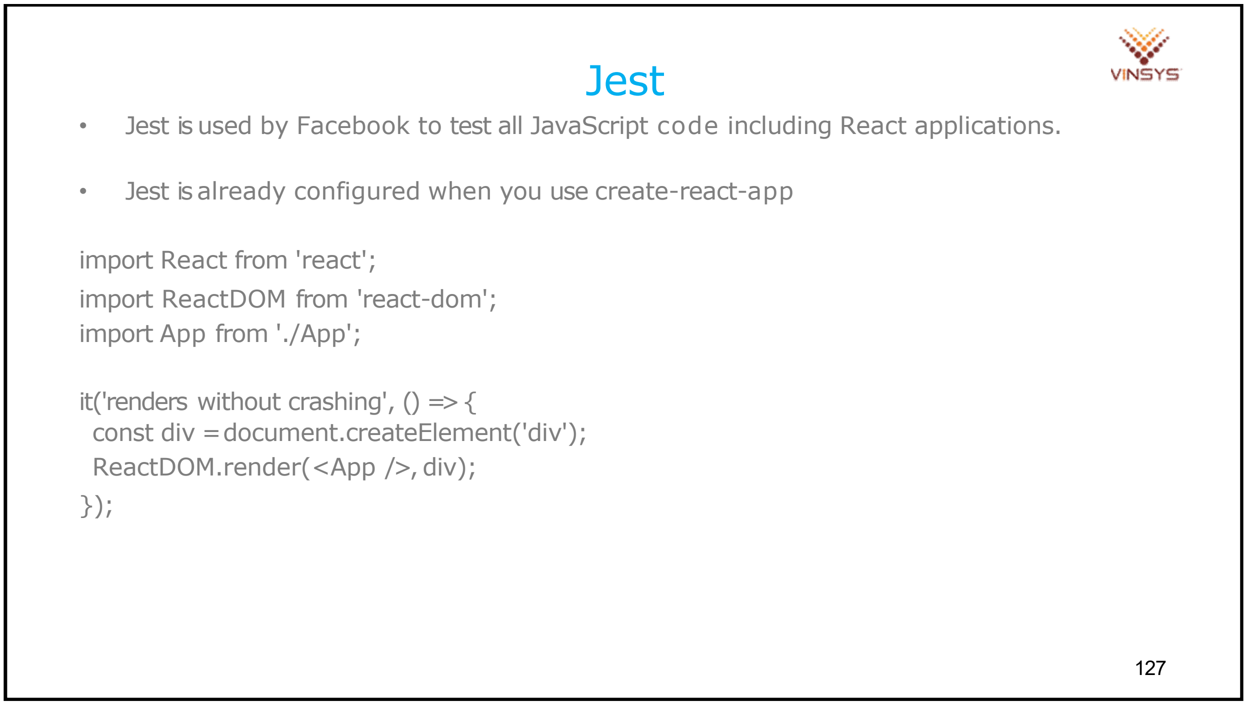
useReducer

- const [state, dispatch] = useReducer(reducer, initialArg, init);



useCallback

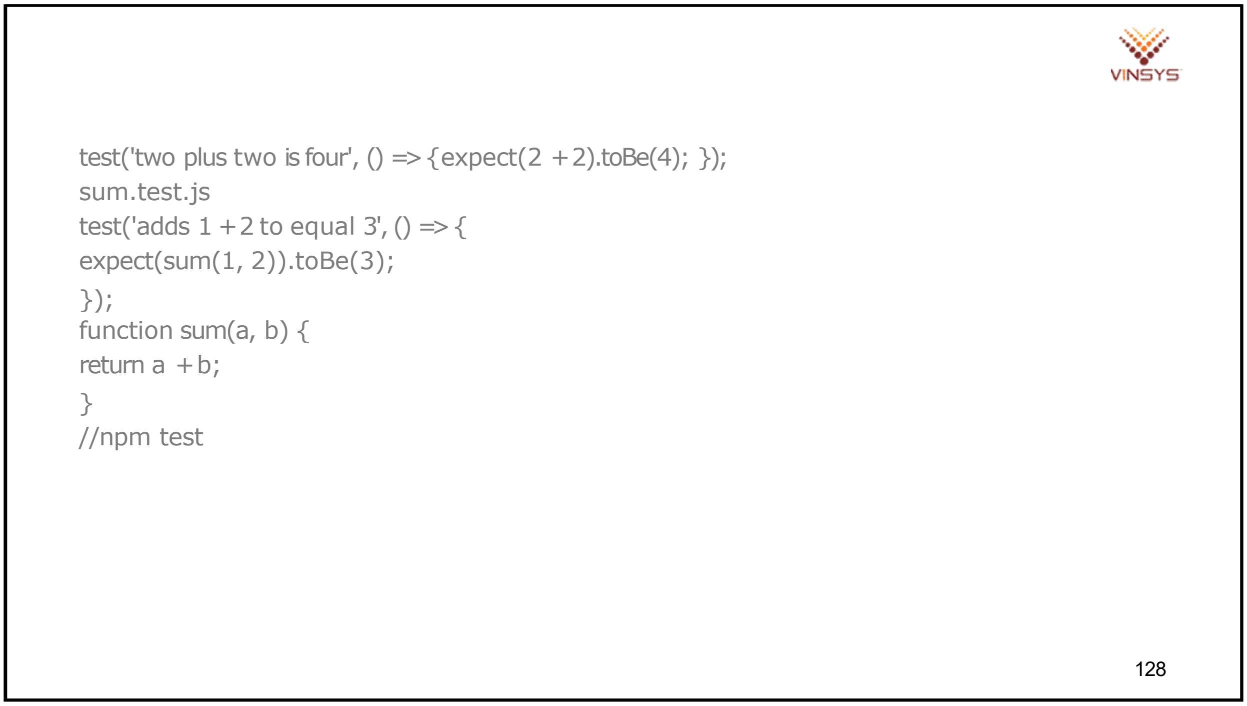
- Useful when you have a component with a child frequently re-rendering, and you pass a callback to it



Jest

- Jest is used by Facebook to test all JavaScript code including React applications.
- Jest is already configured when you use create-react-app

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```



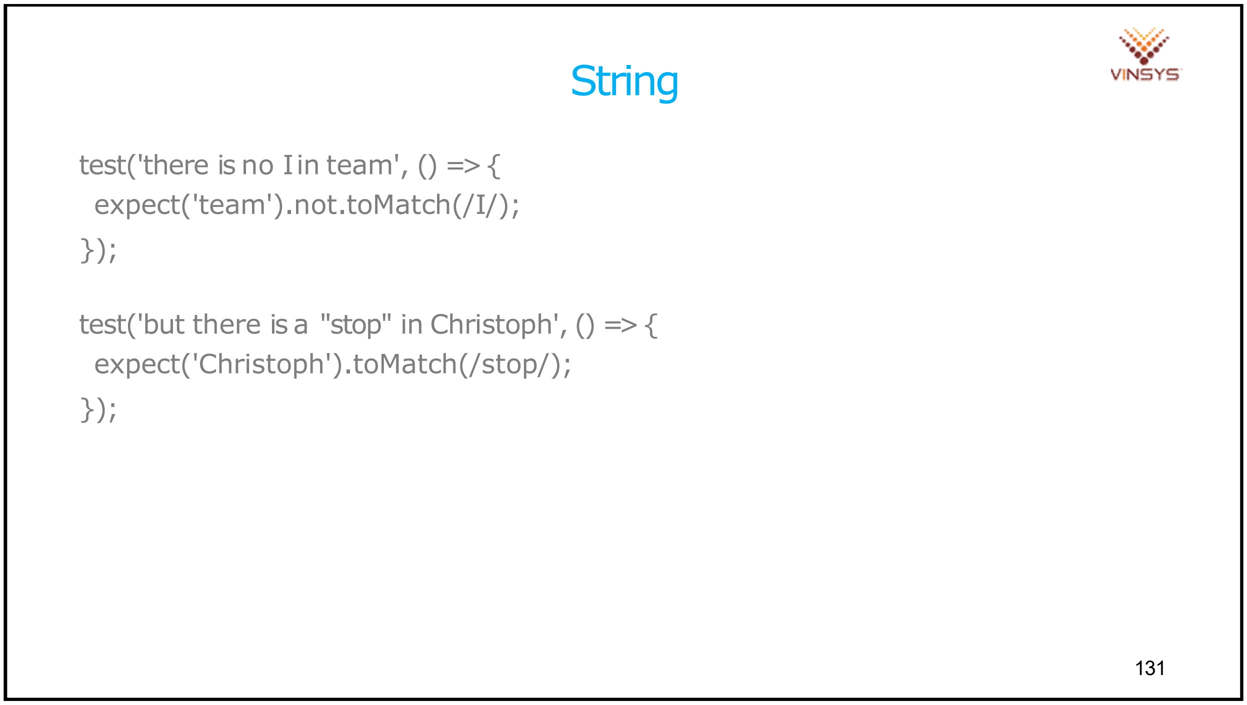
```
test('two plus two is four', () => {expect(2 + 2).toBe(4); });
sum.test.js
test('adds 1 + 2 to equal 3', () => {
expect(sum(1, 2)).toBe(3);
});
function sum(a, b) {
return a + b;
}
//npm test
```

Truthiness

- toBeNull matches only null
- toBeUndefined matches only undefined
- toBeDefined is the opposite of toBeUndefined
- toBeTruthy matches anything that an if statement treats as true
- toBeFalsy matches anything that an if statement treats as false

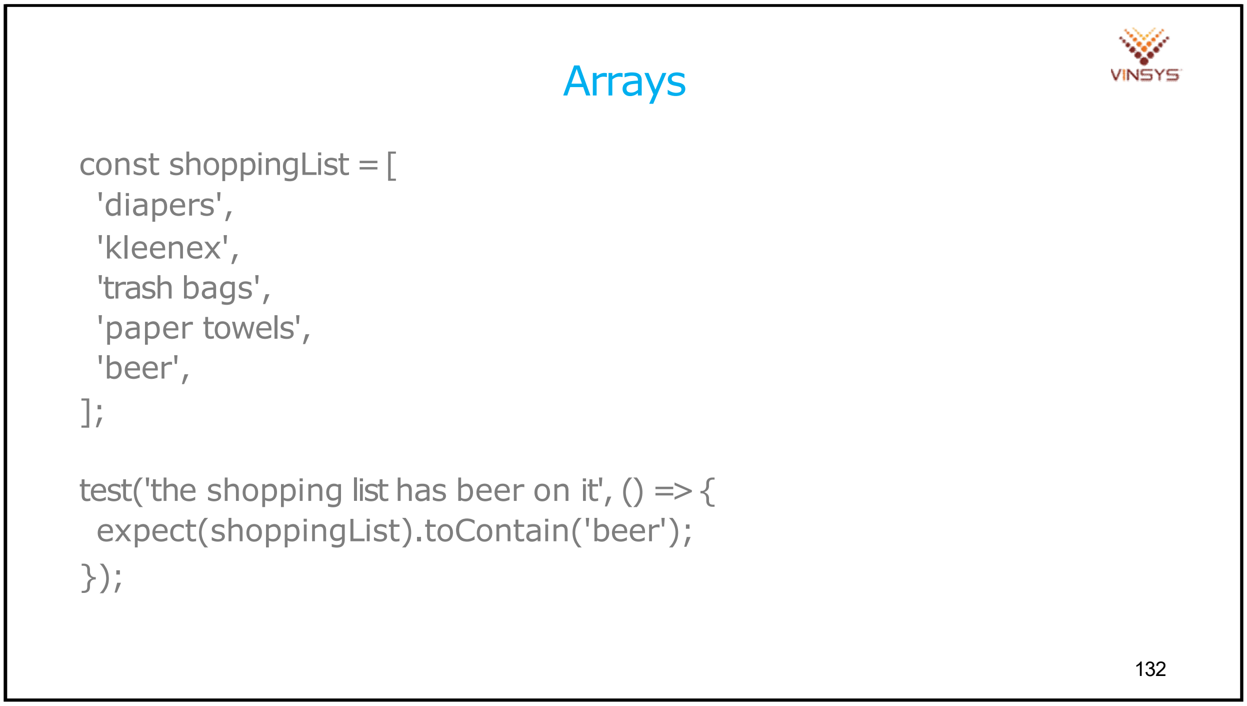
Numbers

```
test('two plus two', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);

  // toBe and toEqual are equivalent for numbers
  expect(value).toBe(4);
  expect(value).toEqual(4);
});
```



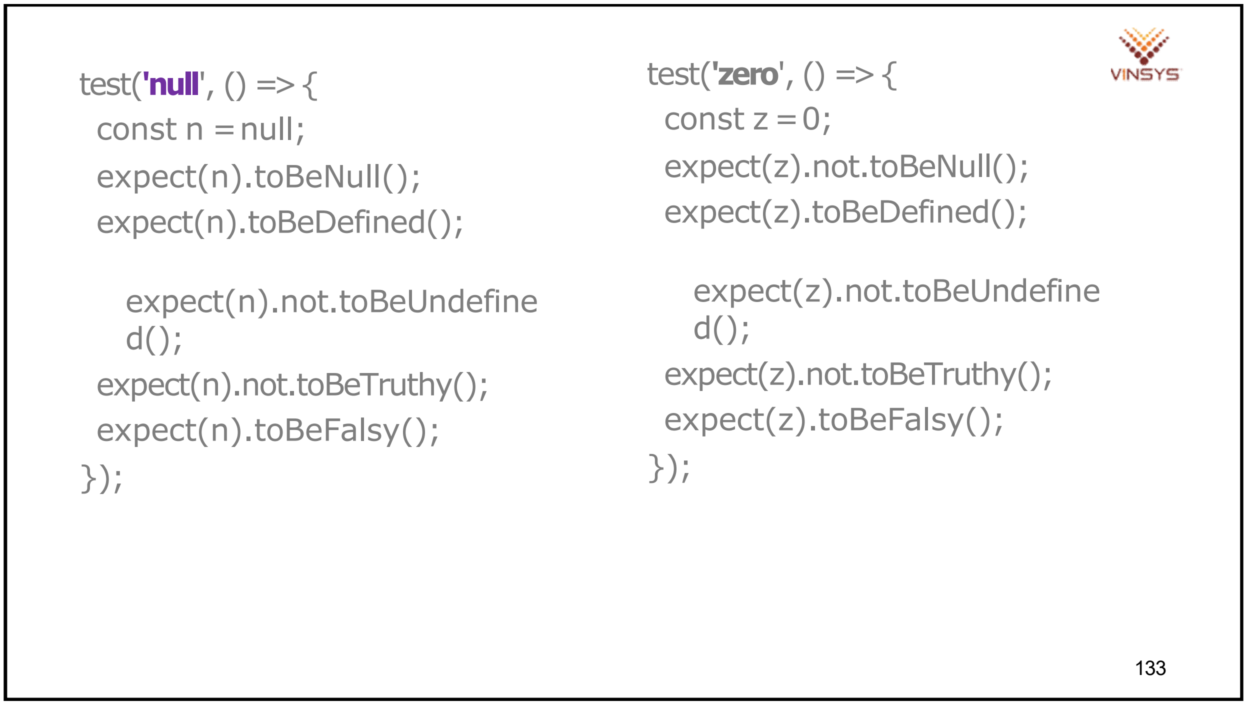
String

```
test('there is no I in team', () => {
  expect('team').not.toMatch(/I/);
});

test('but there is a "stop" in Christoph', () => {
  expect('Christoph').toMatch(/stop/);
});
```



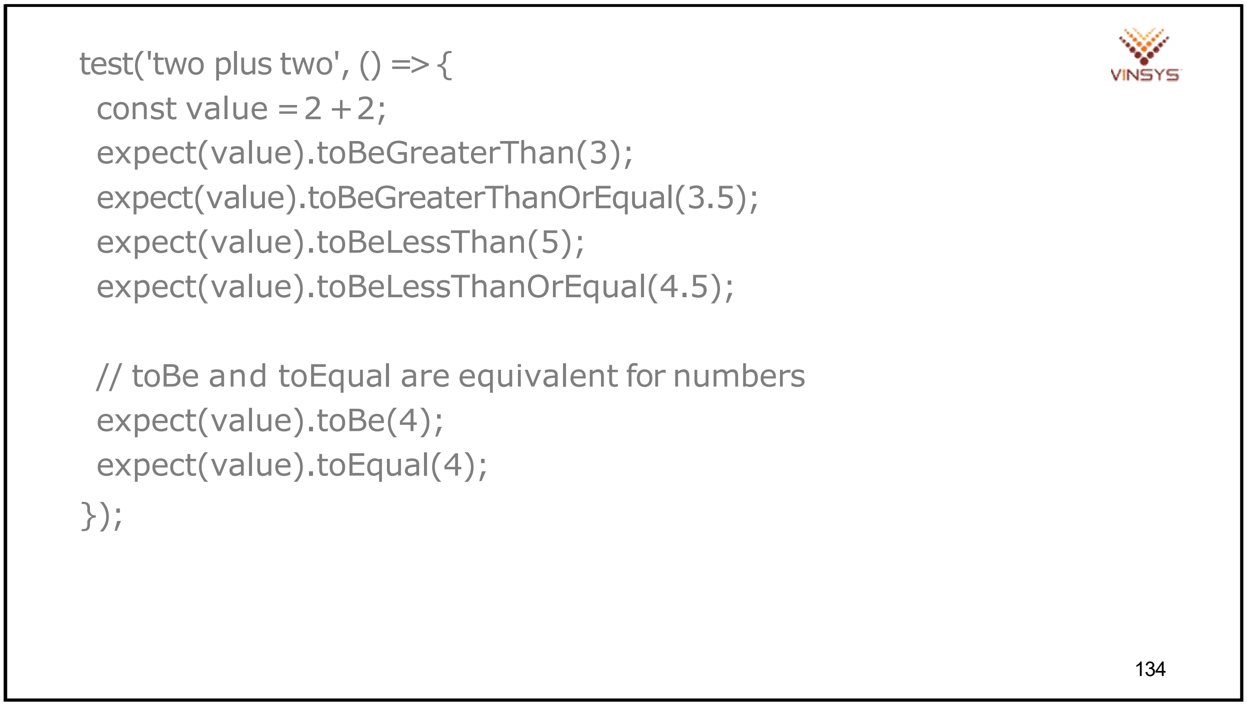
Arrays

```
const shoppingList = [
  'diapers',
  'kleenex',
  'trash bags',
  'paper towels',
  'beer',
];

test('the shopping list has beer on it', () => {
  expect(shoppingList).toContain('beer');
});
```



```
test('null', () => {
  const n = null;
  expect(n).toBeNull();
  expect(n).toBeDefined();

    expect(n).not.toBeUndefine
    d();
  expect(n).not.toBeTruthy();
  expect(n).toBeFalsy();
});
```

```
test('zero', () => {
  const z = 0;
  expect(z).not.toBeNull();
  expect(z).toBeDefined();

    expect(z).not.toBeUndefine
    d();
  expect(z).not.toBeTruthy();
  expect(z).toBeFalsy();
});
```

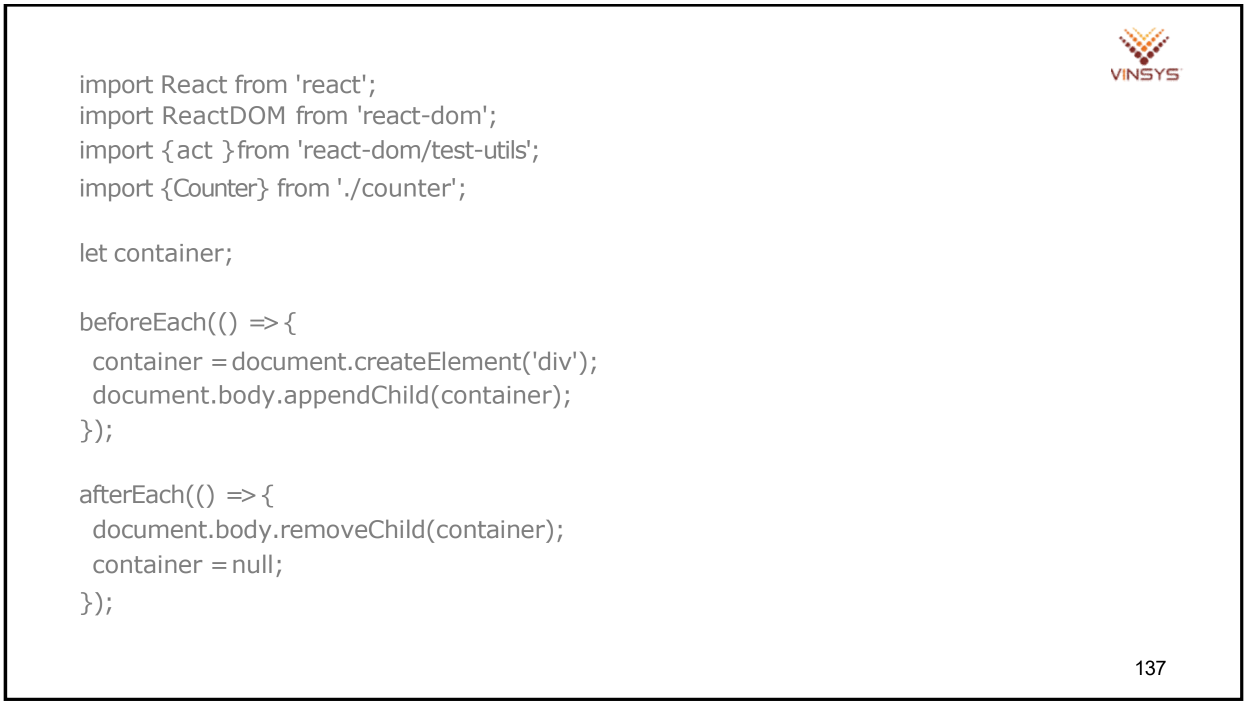


```
test('two plus two', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);

  // toBe and toEqual are equivalent for numbers
  expect(value).toBe(4);
  expect(value).toEqual(4);
});
```

React Testing Library

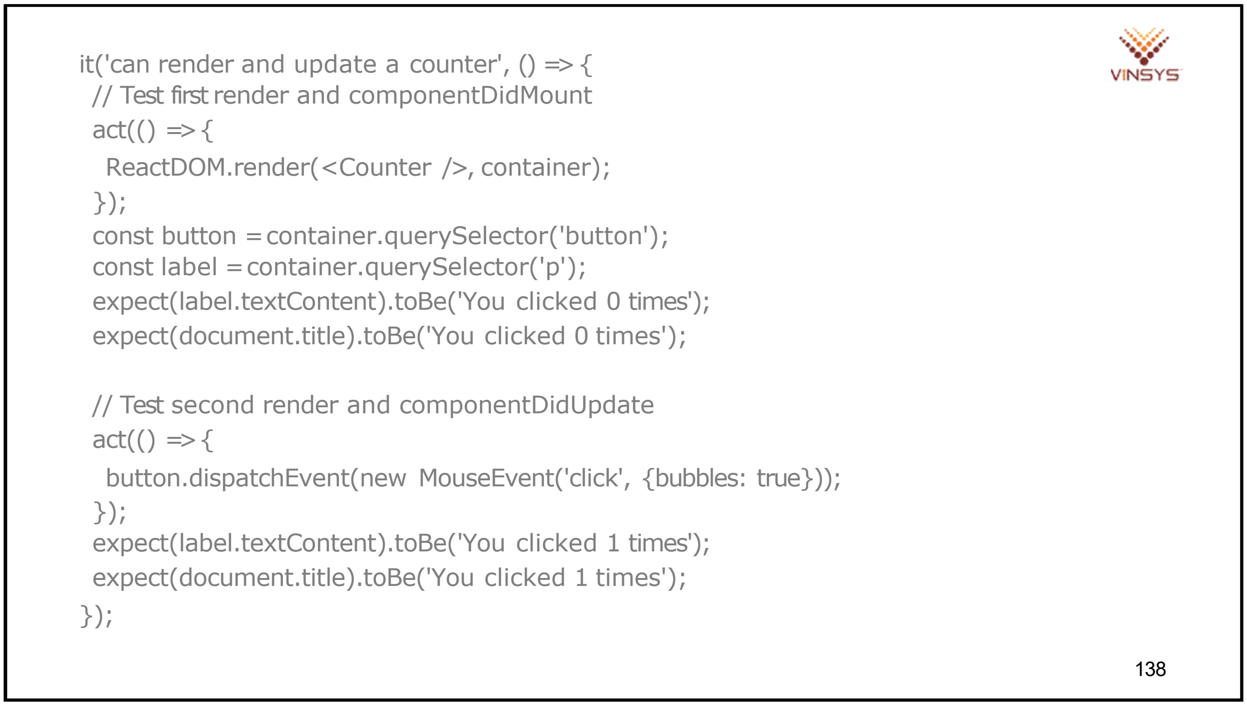
- React Testing Library builds on top of DOM Testing Library by adding APIs for working with React components.

```
export class Counter extends React.Component {
  constructor(props) {
   super(props);
   this.state = {count: 0};
   this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
   document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
   document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
   this.setState(state => ({
     count: state.count + 1,
   }));
  }
  render() {
   return (
     <div>
      <p>You clicked {this.state.count} times</p>
      <button onClick={this.handleClick}>
       Click me
      </button>
     </div>
   );
```



```
import React from 'react';
import ReactDOM from 'react-dom';
import { act } from 'react-dom/test-utils';
import {Counter} from './counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});
```

137

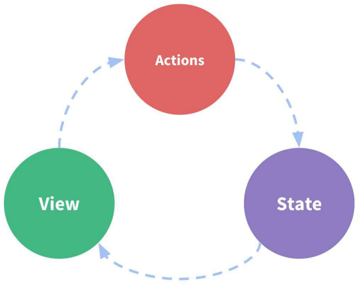


```
it('can render and update a counter', () => {
  // Test first render and componentDidMount
  act(() => {
    ReactDOM.render(<Counter />, container);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and componentDidUpdate
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

138

Redux

- Redux is a pattern and library for managing and updating application state.
- Redux helps you deal with shared state management



Redux Application Data Flow

- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state

Single Source of Truth

- The global state of your application is stored as an object inside a single store.
- Any given piece of data should only exist in one location, rather than being duplicated in many places.

React-Redux

- React-Redux is our official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.

The Redux Store

- A "store" is a container that holds your application's global **state**.

```
const initialState = {
  value: 0
}
```

Reducers

- A reducer is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state: (state, action) => newState

- You can think of a reducer as an event listener which handles events based on the received action (event) type.

- *They should only calculate the new state value based on the state and action arguments*

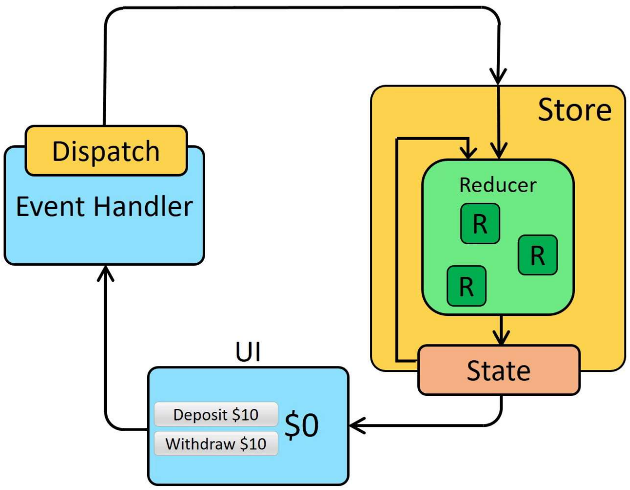
Actions

- An action is a plain JavaScript object that has a type field.
- You can think of an action as an event that describes something that happened in the application.

```
const addTodoAction = {
  type: 'todos/todoAdded',
  payload: 'Buy milk'
}
```

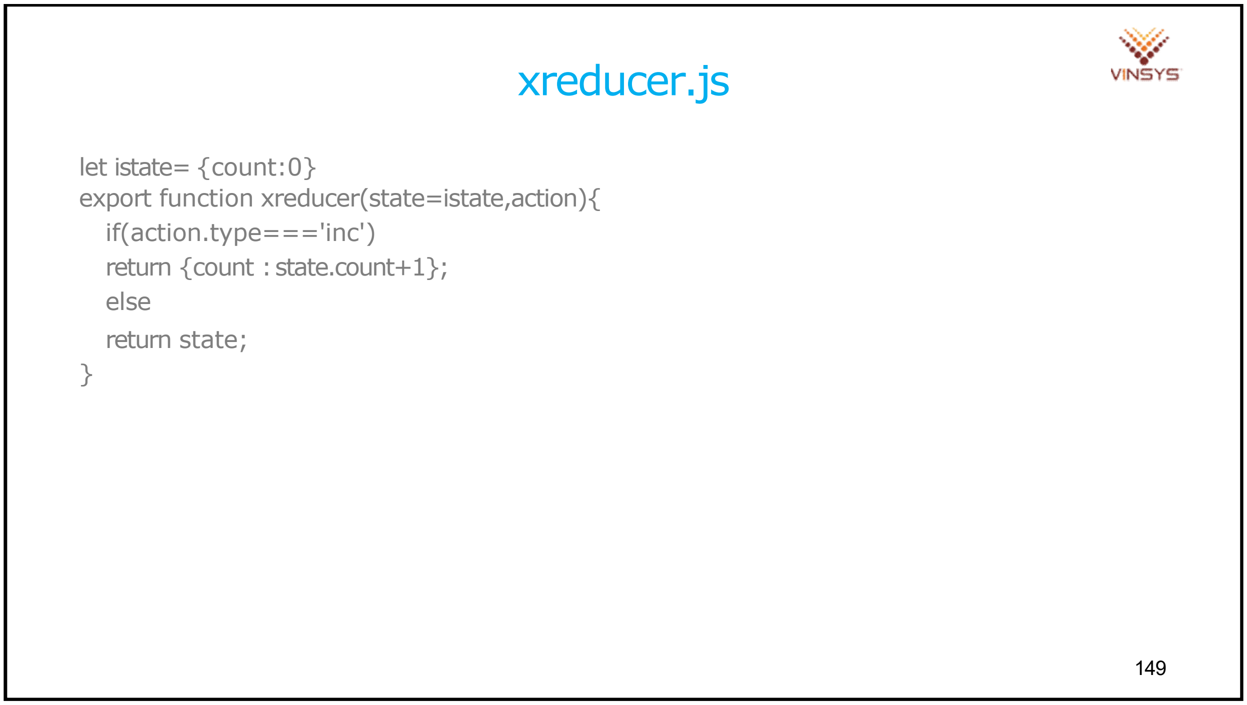
Dispatch

- The Redux store has a method called dispatch.
- The only way to update the state is to call store.dispatch() and pass in an action object.



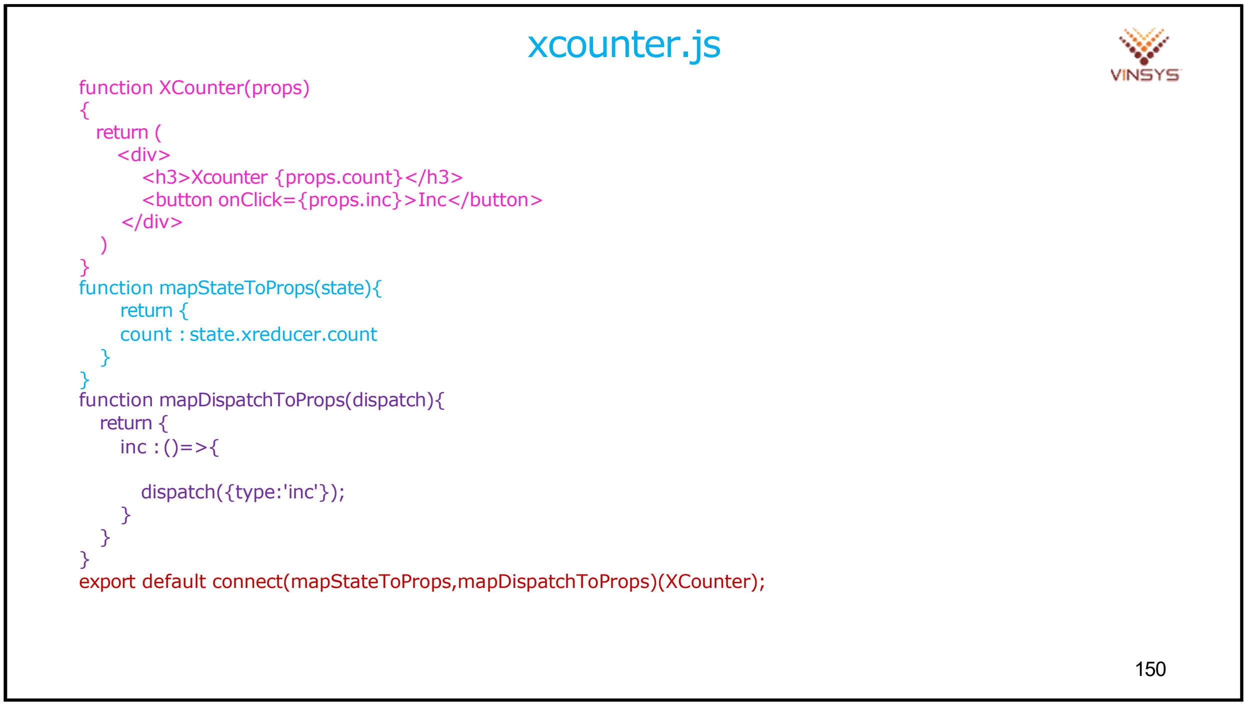
Install

- npm install redux
- npm install react-redux



xreducer.js

```
let istate= {count:0}
export function xreducer(state=istate,action){
   if(action.type==='inc')
   return {count : state.count+1};
   else
   return state;
}
```



xcounter.js

```
function XCounter(props)
{
  return (
     <div>
        <h3>Xcounter {props.count}</h3>
        <button onClick={props.inc}>Inc</button>
     </div>
   )
}
function mapStateToProps(state){
      return {
      count : state.xreducer.count
   }
}
function mapDispatchToProps(dispatch){
   return {
     inc : ()=>{

        dispatch({type:'inc'});
     }
   }
}
export default connect(mapStateToProps,mapDispatchToProps)(XCounter);
```

App.js

```
let store = createStore(xreducer));

export default () => {

  return (
    <Provider store={store}>
      <XCounter></XCounter>
</Provider>
  )
}
```

Thank You