IRIS

REACT.JS Book

Solid understanding of a hard to learn JavaScript library

WRITTEN AND EDITED BY GREG SIDELNIKOV

IRIS

REACT.JS Book

Solid understanding of a hard to learn JavaScript library

React.js Tutorial Book

React is definitely one of the hardest things I had to learn from the pool of JavaScript libraries. In part, it is because it assumes knowledge of so many other things associated with professional web development. Namely, Node.js, NPM, mastery of babel command line tools, EcmaScript (5 and 6,) JSX and possibly WebPack.

This book also makes an assumption. That you're completely new to React programming. It creates a challenge for me as the author, because it's my responsibility to speak about React in a way that is still meaningful, without overwhelming you with large blocks of code that look complicated and hard to follow.

Having said this, my strategy for the presentation format of this book follows a simple pattern of mixing explanations with simple code examples. Primarily, this book is a narrative designed to gradually delve into understanding the React library.

I'll just put it out there right now that this book alone will not be enough to become an expert React programmer. No publication about React can claim this, due to sheer scope of principles you need to understand in order to get going and start writing real applications with React. Then there is also syntactical differences between EcmaScript 5, 6 and soon 7. For this reason React may turn out to be one of the most unusual learning experiences you have ever had when it comes to JavaScript.

But breaking through the hard parts is worth the battle. And I hope this book will prove to be practical in accomplishing this task.

Written and edited by Greg Sidelnikov
greg.sidelnikov@gmail.com

Table of Contents

A Note From the Author

Welcome to the introduction chapter of my book.
This is React Gems a book by Greg Sidelnikov.

That's me, the author!

First, thank you for showing interest in React Gems. This tutorial book was self published by myself through my JavaScript web development newsletter. It will walk you step by step through the process of setting up a development environment and learning Reactive programming from start to finish.

As of 2017, what I call "reactive" programming has become an alternative way of thinking about web applications. It provides software patterns that make your web app extremely responsive to user input and generally feel faster to the end-user (people who are using your application.)

When I was studying React it took me about 3 weeks for something to occur to me. React is unlike any other JavaScript library I have ever programmed with. It assumes a lot of previous knowledge. Because of this I had to write this book a certain way, that gradually delves into the process of understanding it.

Studying React is not about just memorizing a few methods here and there and using them as you wish. React itself is built on *software principles* that usually become known only to professional software engineers who have worked in the industry for a long time. This creates a challenge for authors like me who are trying to explain React to JavaScript newcomers.

I chose to solve this problem by adopting a more fluid style of explanations that border between discussions and source code and that make an attempt to create balance between both. I didn't want to show long React programs and write a few comments explaining what each

line does. That would only complicate the learning process. Instead, I broke down the narrative of the book into components (no pun intended!) that deal with real problems you will be faced with when studying React for the first time.

Programming with React library is a lot more than just learning about and using React ***objects***, ***components***, ***states***, ***props*** and ***methods***. The entire book is dedicated to explanations of what those are. Have you ever heard of props or states in other libraries? States… probably, you did. But not so much ***props*** (React's way of referring to "properties" of an object, which are in fact ***arguments/parameters*** passed to React objects. It's a mystery of why they decided to call this *props*. Because even what they *represent* comes from HTML ***attributes***, not properties.)

React makes understanding both (props and states) explicit requirements. They're almost like sub-features that you absolutely must master in order to become a versatile React library programmer.

It's the quirky details like these that requires previous experience with JavaScript, without which (and explanations of which) it would be much more difficult to learn React programming. Thankfully I am aware of this and the book is written with all of this in mind. At times React assumes knowledge of software principles not inherent to React library itself. The passages of this book will gradually uncover them throughout the book as well.

This book is not a list of object names, methods and functions with side explanations of what they do. You can look them up on Facebook's official React documentation page including long lists and explanations of every single method.

Rather, the book will deal with the gradual learning process and understanding of the said principles involved in React programming.

Before we begin I'd like to show you some of the things I discovered that. Below you will see a list of the ***hardest things about learning React*** followed by most important reasons why you would want to learn

it.

Main Roadblocks to Learning React

Here, I'm going to make an attempt to speak of the commonplace difficulties faced with learning React library from scratch. These are the ones that make learning React not as intuitive as other far straightforward libraries you're already used to.

I'll try to do it without making React come off as a bad choice, because… all in all, this is indeed a book about how to advance as a React programmer. However, if we at least briefly talk about the hard parts now, it will make it easier to sink in later on as we move forward.

1. **New Design Pattern.** New, and in many cases *unfamiliar*. If you're coming to React from traditional pre-2015 JavaScript world of programming, it might take time to adjust to ***modular*** and ***declarative*** design adopted by React library. Sure, if you've ever worked on a large JavaScript application you might have already dealt with long lists of .js files and treated them as separate modules. But for large majority of programmers this isn't true. React requires that we use *modular design* for maximum application management efficiency.

2. **JSX** language throws a lot of people off. It's like writing HTML inside JavaScript and at first it looks nothing short of bizarre, if you're encountering it for the first time ever:

For example:

```
var divblocks = <div>
  <div>Another nested tag</div>
</div>;
```

Like wait, what? Are you seeing what I'm seeing? We have just relentlessly typed HTML into a JavaScript statement, without quotes surrounding the HTML, and even have separated them using line breaks... and still got away with it.

JSX processor, which processes XML as if it's HTML and turns it into a

valid JavaScript statement we can toss around does all the dirty work for us behind the scenes. In reality, this type of code is "transpiled" from XML (that looks like HTML) to EcmaScript 5, so browsers can understand it. This means the actual code the browsers sees will not even look this way. This is just for us, while we're programming our app. But… we can write the code in this format without a hitch.

How it is used is discussed throughout the book. But basically, this format is interchangeable with React component structure. What essentially you have just created is actually a valid React component pattern.

For this reason, JSX is often used with React as a supplementary library. You can create the same statement we've seen in example above, by writing it using React methods. Using JSX is not a requirement, and you can get away without it.

However without it React programming becomes sort of like eating a pizza without cheese. The two work together to provide a tasty experience, even if it takes trial and error and time to learn how to bake the pie.

3. **Babel & Transpiling** code from EcmaScript 6 to an earlier version (ES5) is a bit of an overhead and seems like an unnecessary thing to do as well. Because eventually browsers like Chrome and Safari will have full support for ES6 and this process will become automatic anyway.

Babel allows us to write this code:

```
import someClass from "someClass.js"
```

And this will grab ***someClass*** class definition from the file ***someClass.js***, where it is defined. As of this writing, Chrome, Firefox and none of the other browsers support ES6 import keyword.

However, babel allows us to start learning and writing ES6 code even now, before official support is even rolled out in the modern browser.

Until then it will just "transpile" it back to ES5, behind the scenes, so that the code can actually run in any browser today.

The good news is that installing babel plugins forces you to learn how to use the *command line* tools. Something traditional JavaScript developers are usually unfamiliar with. Which, I think the skill you should be learning anyway.

Using command line is a "must-have" skill in professional software development environment (outside of personal projects in some arbitrary folder on your Desktop). And... it's one of those extra steps that might potentially prevent delving directly into Reactive programming, if not taken. As much of a hassle as it may be.

Good Reasons for learning React

1. ***It's fast***. This is probably the top reason JavaScript programmers are looking to start writing their applications using React library. Utilizing a virtual copy of the DOM, it is designed to perform operations virtually before they are rendered to the standard JavaScript DOM tree. This isn't just a numbers game. It's surprising that moving UI operations to ***virtual DOM*** actually changes how your application feels for the end-user. You've probably already started noticing this phenomenon across the web on websites such as Netflix.

2. ***Modular design***. React supports modular design that is often associated with building large, scalable applications. The reason for this is the advent of EcmaScript 6 specification that came out in 2015 which provides new syntax that favors modular methodology, for example the ***require*** *(ES5)* and ***include*** *(ES6)* keywords tie together not only files alone but assume class-based relationship between objects. And even though as of this writing ***include*** keyword is not supported in modern browsers (even Chrome) we can begin writing modular programs by either using require, or transpiling our ES6 code to ES5 (that almost all browsers do understand by now) before sending it to the browser for rendering. When browsers catch up, we will no longer have to transpile. When this takes place, React will become an even more common library you'll see everywhere. Eventually, modular JavaScript program design will become the norm with or without React.

3. ***Scalability***. In React each HTML element is *treated* like a programmable component that can be nested within other components. Child components have parent components associated with them. The relationship between parent and children components is established through something known as props. Scalability of React programs is achieved by making each component a conglomerate of HTML, CSS and JavaScript code.

4. ***Flexibility***. React is more than just a UI view library as opposed to what

you will probably hear at first when you start figuring it out. And that isn't entirely wrong. React gives us ability to create lightning-fast interactive components that can be used as elements of our application. However, React's utilizes the principle of ***render targets*** which allows you to send result of your code to just about anywhere, included *but not limited to* front-end view.

5. ***Popularity among Employers***. React was developed by programmers at Facebook after years of studying web applications. More or less, it originated from a set of proven software engineering philosophies. Although, many of them most traditional JavaScript programmers will not be familiar with. When you are learning React, you are learning computer programming principles, not just a list of functions and methods, which is the case with other libraries such as jQuery. In 2016 JavaScript has become the leading language for front-end web development. Having React on your resume gives you an edge over your competition if you're looking for employment in software industry. The average Sr. JavaScript programmer in San Francisco Bay Area makes $120-150K a year.

Reactive Programming

What is this reactive programming anyway? React is a small but powerful library written in JavaScript. You can use it by including it on your web page by using SCRIPT tag. Just like you would any other JavaScript library, like jQuery for example.

If you are looking to be prepared for employment at an actual software company, simply adding React to your web page isn't enough. It almost demands that you *understand* other areas of web development that go beyond simple JavaScript programming. The design of this book is dedicated to help nurture that process.

By using React components, which will be shown throughout many examples in this book, you will build an entire application. During many parts of this book I will refer to building applications using the React library as "reactive" programming.

Reactive programming is nothing more than a phrase made up to represent an alternative way of thinking about building modern web applications. *Reactive programming* is not inherent to programming using the React.js library itself. It's just a methodology and it happens that React.js library certainly adopts it as core philosophy.

What I mean by this is that it's the type of programming methodology that will focus a lot on logical functionality of your program. Think about it as a more advanced form of programming compared to working with DOM in JavaScript, where ***each visual element*** of your app not only has events associated with them, but can possibly contain custom programs that you will write yourself which will determine their function.

While React is considered to be mostly dealing with the view areas of your application, don't believe the myths that React is ***strictly*** the "view" in the MVC (model-view-controller) pattern you're probably already familiar with. If not, just know that the Model View Controller

is a programming pattern that separates code from the visual elements of an application. You've probably heard the phrase "separation of concerns" in the past. But the point is that you can either install React, learn and get used to its programming style and get a few UI elements designed, or you can make it the reason you learn in-depth subjects of web application development. The latter, is the approach I have taken while writing this book.

I hope you enjoy this book, learn the React library and explore the possibilities offered by modern software development that will train you for making real-world web applications.

The Main Principles of Reactive Programming

Before we even go any further, it's important to mention what we are even doing here. We can refer to programming with React as ***functional templating***. Not to be confused with functional programming languages. Here I simply mean that your React components have a specific function to perform and that this is a way to customize your components with unique capabilities. Whereas HTML provides us with a set of elements (as HTML tags) that handle the layout of our web application and provide interfacing with JavaScript code via attributes, React is that… and a lot more.

React gives us ability to create compact blocks of code that tell us at a glance exactly how your component will render on the screen all in one place, and often in a single line of code. This gives tremendous opportunities for giving the programmer an immense amount of control over making applications that create visual elements and at the same time, give him (or her) the ability to program their behavior.

React programs usually do not deal with separate CSS stylesheets. Interfacing with the style is provided via virtual DOM. We'll see how that works later on in the book.

Compared to traditional HTML, CSS and JavaScript approach, where we would have to scout around the source code just to find different bits and pieces of just one single element on the web page, React gives us a way to use programmable templates.

In standard Ajax-based HTML applications, we're used to seeing code such as demonstrated in the following example:

```
// HTML
<div class = "LoginForm">
   <div class = "Name">
      Enter username here.
   </div>
```

```
</div>
```

And here is the counterpart jQuery ajax example:

```
// Ajax counterpart
$.ajax( { url: "/login.php",
        data: { "username": "user1",
                "password": "abfgh123" },

    // login.php returned some data
    success: function( msg ) {

        // Adjust DOM data
        $(".Name").html( msg );
    }
);
```

This is a purposely simplified example. But you can see how even then, the code is located in two separate blocks. One is the HTML layout definition. It is almost like a reusable template for LoginForm with some dynamic functionality.

The second block of code is the actual program that executes an Ajax statement. This line of code will load data from a PHP script based on the data passed to it (username and password) and returns some type of a message in ***msg*** variable.

Last, jQuery is used to modify the DOM by replacing the contents of div whose class is Name with the message received from the script.

For a while this methodology worked just fine. Over the last several years, this is how we're used to working with dynamic elements on a web page. But the brilliance of React, is that it takes all that and gives us programmatic control over the HTML and dynamic data, all packed in a neat **render** function:

```
render: function() {
```

```
  return
  <div>
    {
      This.state.name ? this.state.name :
      <span>Enter username here</span>
    }
  </div>;
}
```

Notice how HTML and JavaScript code is intertwined without having to open separate tags. Both exist within the same JavaScript statements and used almost interchangeably without any problem. We'll get to how this is done in a few moments.

Of course this isolated example makes almost no sense outside of being used in a real-world application. But it shows a few important principles.

This react code that borrows its JavaScript-cross-HTML syntax from JSX language gives us a means to know what the output of a dynamic element will look like by briefly looking at the code. React here is trying to be everything to everyone. And in some cases you will notice some patterns that almost feel like bringing the ***back end*** logic into the ***front end***.

The ternary operator **? :** is used in this example. And basically, it's saying that if the property name exists (not ***null*** or ***undefined***) then we should output it. If it doesn't exist, we should output generic "Enter username here" as the default message.

This brings HTML template methodology to a different wavelength. Here, we are designing and programming the component at the same time.

Gems and Working in the Mine

It's only the first chapter, and I am already delving into complexities of React programming. However, let's get back to basics and break down some important principles into smaller tutorial pieces first. It will help us later down the road.

Just like with my jQuery Gems book, we will start with a collection of "gems" which is prerequisite knowledge that will provide quick insightful bits of information required to move forward. Because the software industry is in transitional times right now between EcmaScript 5 and 6, it will help us get common quirks out of the way and stay focused on what's important when we get to the rest of the lessons in the book.

If these gems are skipped, certain parts of the book later on might not make a lot of sense. Especially if you've embarked on studying React as your first contact with JavaScript programming.

But the gems is only one part of the book. They are designed to reduce as much turbulence as possible and help you ease into the world of modern JavaScript programming with a hint of reactive principles.

You simply must understand these principles before descending deeper into the mine. The gems will sharpen your pick axe. The second part of the book will show you the rest of the cave.

React on the Palm of Your Hand

To make your journey easier, I've included this section here representing each separate aspect of a React program. Fasten your seatbelt and get ready to absorb a lot of new principles! We're truly taking off into the world of React programming now:)

This is a bird's eye view explanation going over the most important concepts involved in building React programs. I already mentioned that learning React is fragmented. And it assumes previous knowledge of

advanced programming patterns.

In this section, I'll try to break them down into reasonable chunks that can be analyzed individually. And as we move forward, this will help you put the pieces of the puzzle together. Think of the principles described here are the ideas fueling the core engine of React.

Just because they are neatly outlined here, it doesn't mean that learning React will be easy. But it certainly makes it easier to go through the rest of the book by taking your time to familiarize yourself with them.

The Essentials

If you're familiar with jQuery, you've used the dollar sign ($) object for selecting HTML elements and applying various operations on them.

But what is a similar building block of React when it comes to HTML ***elements*** (not components.)? It must be the **createElement** method.

```
React.createElement(type, props, children);
```

And here are a few examples of how you would use createElement in practice:

```
let link = React.createElement('a', {href: '#'}, "Save");
let nav = React.createElement(MyNav, {flat: true}, link);
```

Here, we programmatically created HTML elements using React method createElement. Then we have taken the created ***link*** element and placed it within **nav** element to be one of its children.

Of course in this case link is the only child, but multiple links can be used as well, as additional arguments separated by comma:

```
let link1 = React.createElement('a', {href: '#'}, "Save");
let link2 = React.createElement('a', {href: '#'}, "Delete");
let nav = React.createElement(MyNav, {flat: true}, link1, link2);
```

You can go like this forever, adding as many children as required by your application.

Notice nav variable uses **MyNav** without double quotes. This is perfectly fine. When we use ***custom*** elements (not the standard ones like ***a***, ***div***, ***table***, etc…) that we make up ourselves, we must use leading upper case letters. Therefore, the navigation element is called **MyNav**. As long as we use the leading capital letter React knows we're

referring to a custom element. No need to use surrounding quotes here.

In **JSX** it will translate to something like **<MyNav />**

Eventually with time as you develop your application, you'll arrive at full structures similar to one depicted in the following example. Note, that here it is a little different as I am actually using React statements directly as children arguments:

```
let nav = React.createElement(MyNav, {flat: true},
    // start children of nav
    React.createElement('a', {href: '#'}, "Save"),
    React.createElement('a', {href: '#'}, "Load"),
    React.createElement('a', {href: '#'}, "Delete"),
    React.createElement('a', {href: '#'}, // AlertContaner
        // start children of AlertCounter link
        React.createElement('span', {alerts: 0}, "No Alerts"))
);
```

Note that some children are simple strings of text. They are used as text nodes here. The stuff that goes in between the brackets of tags should also have a representation. And that's what they're for. In this case children can be used as the text supplied to the link's innerHTML, when they are not used for specifying nested elements.

Also, here ***createElement*** is how you would create a React element in ES5 without JSX. I'll explain this in a moment. This is version ES5, for coding pre-2015. While I advocate the use of ES6, I wanted to show you how it's done in ES5 first. Moreover, ES6 is backwards-compatible with ES5. So, this means that a lot of the time how you write react programs will depend on your personal preference.

What's important to understand, for now, is that ***createElement*** takes three arguments: ***type***, ***props***, ***children***.

Here, **type** is the name of the element. For example A, H1, DIV, etc.

Then we have ***props***, which are properties you pass to this component. Props are not states, but they can modify states. We'll draw this distinction later in more detail.

Finally, ***children*** are just hierarchical elements to be found within its parent elements. Create chains of them and type them right into ***createElement*** function as N-number of arguments at the end.

Another key that unlocks deeper understanding of React here is that all of the code above has its equivalents in ***JSX***. The two can be used interchangeably. This is the part that actually is exciting, once you truly start understanding how it works.

So, while in ES5, you would use ***createElement*** function, in JSX (arguably its common use is with the rest of ES6 syntax) you could create ***exactly the same functionality*** by simply writing it out as follows:

```
let link1 = <a href = "#">Save</a>;
```

Now you see where I am going with this? JSX turns HTML into JavaScript statements. Here **link1** is object-like representation of the element which is exact equivalent to React code we created earlier with ***createElement*** method.

And yes, **link1.href** will contain "#" by simply declaring it as JSX. Once I realized that this is how it works JSX started to make a lot more sense and I actually started to feel comfortable using it.

It's like a two-way road between HTML and JavaScript where attributes automatically become properties.
And here is the more complex example, this time in JSX once again:

```
let nav = <MyNav>
    <a href = "#">Save</a>
    <a href = "#">Load</a>
    <a href = "#">Delete</a>
```

```
   <a href = "#">
      <span alerts = "0">No Alerts</span>
   </a>
</MyNav>;
```

We have just recreated virtual DOM using JSX language using a single JavaScript statement! You can pass the value of nav around as a statement into functions. It's almost like another way of creating a mini DOM structure.

Now you (hopefully) see that this is the very structure of virtual DOM in React. It's like a secondary DOM we're going to be working with, changing, modifying, editing, deleting and adding children from, etc.

We will update the actual primary JavaScript DOM object which is responsible for visible elements in our application… but ***only when it actually needs updating***. None of the virtual DOM operations we're performing will be associated with spending time updating the original DOM. And this is what improves performance of React applications.

The best thing, React does this for us automatically. As a React programmer, you are not required to understand how React does this internally. You just focus on design of your application.

Note, of course if you type JSX code directly into your JavaScript application, it will not even compile. You must be running JSX processor. To do this you would usually initialize it from command line, so it constantly runs in the background.

Babel has a JSX pugin. I also recommend learning how to make babel watch your files. What this means is, you can set babel up so that it auto-transpiles your .js files (and places them into "build" or "production" directory) automatically, soon as the file changes. This babel functionality is called "**watch**". It first must be enabled. I explain how to do this on Windows PC in my video tutorial:

React Tutorial on YouTube that helps you set up your development

environment, install Node.js, NPM, babel, JSX and help you start watching your js files:

https://www.youtube.com/watch?v=tXaNvGcjEi0

You can get away with simply adding babel.js to your SCRIPT tag at the top or bottom of your web page. JSX will transpile just fine. But this will significantly slow down processing of your application. It should be used only for testing.

The video, on the other hand is only 58 minutes long, and if you're serious about setting up a professional web development environment I recommend breezing through it on 2x speed option in YouTube player which should only take about 25 minutes. It's like a mini introduction course.

Finally, we're drawing this discussion to an end by showing that you can also clone existing elements using ***cloneElement***.

```
React.cloneElement(element, props, children);
```

It works in the exactly same way as createElement. Just pass an existing element as element argument. Then, whatever props and children you pass to corresponding parameters, will be merged into ones already available on the original element. This is reminiscent of extending object functionality in traditional Object-Oriented programming. Here, we're somewhat imitating OOP functionality.

Created elements can now be actually rendered to the DOM. You cannot render to virtual DOM. The ***render*** function is the magick that takes care of this process within React library internals. This is when our element "copy and pasted" from our virtual DOM into the actual DOM that will be visible on the screen.

React provides ***render*** method on the main **ReactDOM** object as shown below. In reactive programming ***ReactDOM*** is the second most important object after ***React***.

```
ReactDOM.render(reactElement, domContainerNode);
```

Here *domContainerNode* is the parent element on which reactElement will be rendered, re-rendered or attached for the first time (mount.)

You can use render multiple times, if you need to update properties of the component.

Here's what a real-life scenario might look like. In this example we're taking a ReactElement and rendering it on a DOM node:

```
ReactDOM.render(
   React.createElement("div"),
   document.getElementById("container")
);
```

If you have an element already associated with a variable name, you can look it up using *ReactDOM*.**findDOMNode**:

```
ReactDOM.findDOMNode( element );
```

This statement returns the DOM node associated with the given element. Note, that this will only work only after element has been rendered to the DOM with the render method. Until then, it is not findable in the DOM, even if it was already created.

Special Props

Here we will briefly talk about special prop names, of which there are a few. These should be treated almost like reserved keywords in JavaScript, except in React. They all have a special meaning.

Children

The prop name **children** is automatically added to this.props by ***React.createElement***

className

This prop corresponds to the HTML's class attribute. You must use **className** in your components/elements instead of **name** to avoid clashing with HTML's reserved keyword.

htmlFor

Same as *className* only for the "for" attribute. ***htmlFor*** should be used in your React element anywhere where you need to use "for" attribute.

key

The **key** prop uniquely identifies a ReactElement. It's used with elements in ***arrays***.

ref

The **ref** prop accepts a callback function which will be called:

1. With the component instance of DOM node on mount.
2. With ***null*** on unmount and when the passed in function changes.

style

The **style** prop accepts an **object** of styles, instead of a string. In React, there is no mechanism for specifying CSS styles as a text string. This is

one of the limitations of using JSX.

But this limitation can also be looked at as an advantage, because CSS styles can now be represented programmatically, where each property of the CSS-style object represent CSS property pairs.

It might feel a little awkward at first but this style actually adapts much better to the overall scheme of React programming.

PropTypes

React has a special property called ***PropTypes***. Typechecking is its primary purpose. As your app becomes large in size, type checking can help catch various bugs.

Here is an example. Let's say we already have a React component created, and named ***ReactComponent***. By default typechecking is enabled in development mode. It is actually not turned on in production code.

```
ReactComponent.propTypes = {
   name: React.PropTypes.string
};
```

What React.PropTypes does is it turns on validation for the type of the data stored in a property. By default React has a number of data validators, and if a ***Numeric*** data type is passed to a prop defined as ***String*** (as shown in example above) then you will see a warning in JavaScript developer's console in Chrome or other browsers.

The propTypes in React are defined as follows:

```
any
array
bool
element
func
node
number
object
string
```

Class Component Options

We'll speak about components a lot more throughout the rest of the book. They are the building blocks of your react application. But remember that this is "React on the Palm of Your Hand" section! Let's briefly overview the full scope of a React component, its options and methods available by default in React library. Components are defined by ***createClass*** method existing on the main React object:

```
var MyComponent = React.createClass({

   displayName: 'MyComponent',

   /* This is the core of your React Class
      options and lifecycle methods will be written here */

   render: function() {
      // Once rendered, let's return a newly created element
      return React.createElement( /* … */ );
   }

});
```

This is the basic structure of a component class.

Component Options

Components hold the **propTypes** object and two methods that return an object:

1. **propTypes** *object* mapping prop names to types
2. **getDefaultProps** *function* returning object
3. **getInitialState** *function* returning object

Lifecycle Methods

1. **componentWillMount**
2. **componentDidMount**

3. **componentWillReceiveProps**
4. **shouldComponentUpdate**
5. **componentWillUpdate**
6. **componentDidUpdate**
7. **componentWillUnmount**

These 7 methods can be implemented in your React class. Not all components classes will require using all of the seven functions. This will depend on your components' purpose.

Each is basically an event that executes at a particular time during the lifecycle of a React component. Most of this functionality is to do with checking mounting and update state of the component.

In addition to these methods, of course you can also add your own custom functions, depending on what your component is supposed to be doing.

Component Instances

Component instances are the instantiated objects from the class representation we've just taken a look in the previous section. Here are a few hints as to their implementation:

1. The **this** keyword within a component refers to itself.
2. Stateless functional components do not have component instances.
3. One component instance may persist over multiple equivalent ReactElements.

Properties

1. Property ***this.*props** contains any props passed to React.createElement through tag attributes.
2. The ***this.*state** property contains state set by *setState* and *getInitialState* methods.

Methods

1. ***setState***(***changes***) applies the given changes to ***this.state*** for re-rendering the DOM component.
2. ***forceUpdate()*** immediately re-renders the component to the DOM.

Gems

In this section I will present several "gems" that expand on reactive programming with a few explanations of important principles. Each gem is a focused discussion on a particular subject.

Gem 1 - React Components

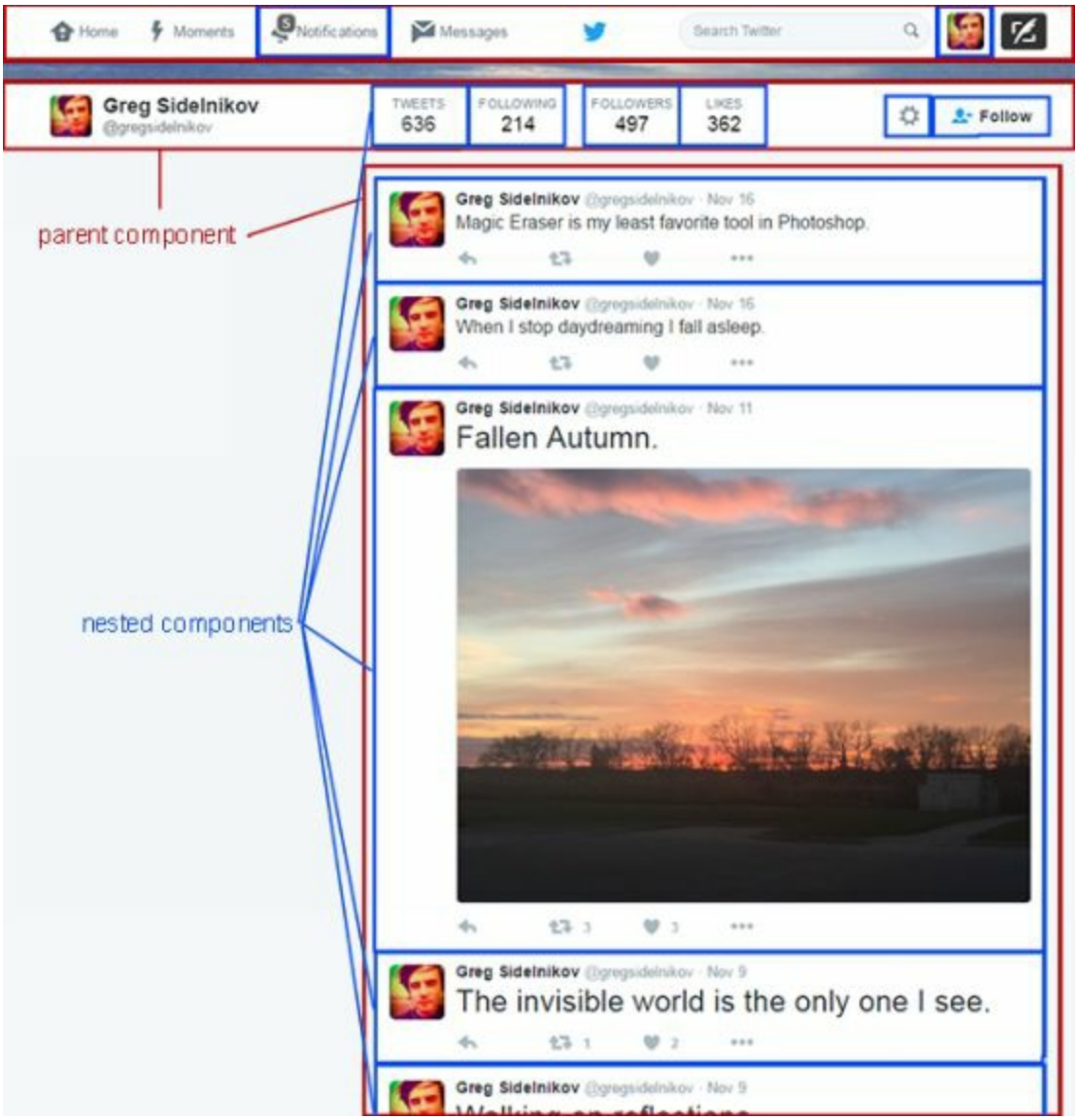
Components are the building blocks of React applications.

Because React is primarily used with the application ***view*** (the interactive User Interface dynamics) in mind, React components can be thought of as visual blocks that your web application is composed of. React treats each component as a reusable template.

In React, each component is a bundle of HTML, CSS and JavaScript code which identifies one particular block of your application.

For example, here is my Twitter page @gregsidelnikov looked at from the vantage point of React components.

Programmatically, React components follow in the foot tracks of nested HTML tags. They have parent and children, and the parent can also be a child of another component up the hierarchy chain.



Think of component design as the blueprint of your entire application. Before you start writing the first line of code in your reactive application, it's a great idea to map it all out on a piece of paper. It's just an intuitive way of thinking about your page design as a web application with unique elements.

And just like the first time you were learning about regular DOM back in the day and realized that each HTML element (such as P, DIV, TABLE and others) is actually an object in a long tree-like chain of the Object Model, think of react components as visual areas that are also tied to some sort of software function. This time, this functionality will be

custom-determined by you and the requirements of your application.

From these fundamentally basic principles of React, you can now see why Facebook and Instagram love using it. It's lightning-fast, it helps break down feed-like elements into software blocks composed of components which are just bundles of CSS, HTML and JavaScript code. This design model is a lot more in touch with app-like behavior rather than just informational websites that use table-like layouts.

Components are hierarchical. They can be children of other components. For the most part, components are tailored for reuse in your application. They contain patterns of code that can be adapted to a wide variety of cases. React code usually follows a pattern of creating chains of these components that work one within another, while retaining fast rendering speed and flexibility.

Gem 2 - Render Method

Chances are that to some degree, you've already split up your app into compartments when you were building regular HTML-JavaScript applications. Then, you used Ajax events to populate each view separately, based on what type of data changed in each HTML element cell.

But React takes this approach one step further by requiring **render()** method. The render method must be attached to each React component you create. There is no way around it. It's part of the pattern that React components demand that is being used.

Even though React is so much more than the view layer - making the ***render method*** explicit to programming with React, is the reason why React is often considered to be primarily a view-based library. We will see why this is true throughout the book, but also why this isn't *entirely* true.

This isn't fair to say that React programming is necessarily view-only programming. In fact, the choice of ***render targets*** tells us the opposite. React is a programming pattern that follows fundamental philosophy that actually doesn't require a view.

You can render results of a react component either to a browser or to a string of text (using a method made just for that, called *React.***renderToString**,) JSON object or anywhere else you wish. In this case rendering isn't rendering at all. Because the word render implies visualization of data. So React is more like a set of principles, rather than "a way to make User Interfaces" as many beginners tend to believe after brief familiarity with the library.

While you can use any third party library in addition to React, for example database IO libraries, React still offers a lot in the way of custom programming that goes behind the view elements. In fact, it demands that you understand how these programming principles work.

The method ***render()*** which is usually attached to the React's own "virtual" DOM is the method that will be executed when your application is loaded into the browser for the first time. And consequently, only when the data in the element it represents should be updated. However, this is where it gets interesting. Until the render method is called, the react's component isn't even "mounted" to the actually JavaScript DOM, nor will it be displayed. And it's possible to create React components without mounting them. We'll cover mounting in greater detail in Gem 6.

We know that render triggers an update of the data set (or state of the component). But when should the data be updated? This will depend on the type of data shown in your components but also a few generic function that are responsible for updating it at **key times** while it's being used. Later in the book we'll get to the functions responsible for giving us control over these events.

In a real scenario, the elements themselves could be an alert icon notifying the end-user of an increase in number of incoming alerts, the number of profile followers or profile views.

Gem 3 - Virtual DOM and Bandwidth Salvation

React makes us think thoroughly about what each element will represent, because each component is tied to some sort of software function that will, in real time, update the data in each component without updating the entire layout. And in React this process is made more efficient in comparison to traditional applications. You don't have to reload the entire page like on regular HTML websites.

This makes your application feel different, and it saves bandwidth. if you think about it, the non-reactive websites we've been building until 2017 is really just a lazy man's design idea. Reloading ***everything*** just to see if there is a new alert isn't exactly the best application design. Yet, this is how websites like Facebook have been working all of this time. And bit by bit, it's starting to change. This is why they are moving toward reactive implementations.

But Reactive programming isn't just a bunch of ajax calls that pull the data and replace an HTML element. It's much more sophisticated.

And because fast access to *only relevant* components is usually the key to good application design this is where ***Virtual DOM*** comes in. Virtual DOM is really a quirky way to provide a fast way of modifying the DOM, while abandoning slower old-fashioned alternatives.

The Virtual DOM is hands down one of the greatest advantages of choosing to write your application with the React library. In fact, when you add react library to your web project, it behooves you to also add ***react.dom*** library. The two are just inseparable. And all reactive JavaScript applications by default also include React DOM library.

This is where React's Virtual DOM makes a big difference on the usability of your application. It's what allows websites have that native application feel, where each element responds instantly to a touch event.

You've probably already seen this starting to happen throughout the internet on sites like Netflix,, Facebook and Instagram. You can actually

feel the difference. Where actions happen instantly, seemingly without having to wait for HTTP requests to load first every time you press a button.

The application responds instantly. React doesn't eliminate HTTP bandwidth, it just minimizes it and focuses only on whether the data really should be transferred or not just because of a button click event, for example.

Of course sites like Netflix and Amazon preload some of the data (first 15 seconds of a movie trailer, for example) regardless of whether the user clicks on the play button or not. But together with other "reactive" principles it makes a big difference on UI responsiveness.

We're still building the app from familiar HTML elements, but the response rate to each event is brought about by a DOM that is significantly faster than even jQuery's implementation.

There is nothing special you have to do in order to start using virtual DOM. By utilizing React component templating pattern, you're automatically using react's own virtual DOM traversal. It transparently imitates standard DOM model by providing camel-case (like ***className*** *and* ***fontFamily***) methods and properties matching the names that we're already familiar with.

Gem 4 - Two Distinct Ways of Initializing a React Class

As of the time this book is being written, the web is still somewhat transitioning from EcmaScript5 to EcmaScript6. When EcmaScript 2015 (ES6) came out about two years ago, it caused a lot of turbulence in the JavaScript world. And divided programmers into "the old way" and "the new way" categories. And some prefer to use one style over the other based on the circumstance without any problem. Neither one is either better or worse.

However, in the end both ES5 and ES6 are executing exactly the same code. Most of the difference is syntactic. Surely working in two different languages at the same time might create unnecessary headaches, especially when learning React for the first time.

There simply isn't "one standard way" of doing anything in a loosely typed language such as JavaScript. This has always been the case.

It is extremely important to make this distinction, when it comes to creating React components. There isn't one way of doing it.

In this gem, I will demonstrate creation of two exactly the same React components in two different ways. The difference here, is the difference between EcmaScript5 and EcmaScript6 itself.

For the most part, one significant difference is in the usage of **constructor** (ES6) and a specialized init state function method (ES5). Let's take a look.

EcmaScript5 example looks as follows:

```
var NewComponent = React.createClass({
   getInitialState() {
      return { /* initial state */ };
   }
});
```

Notice that in EcmaScript5 the constructor keyword does not exist. And

this is why in this case React provides **getInitialState** function. If you are writing your app in ES5, you should use this method to initialize default state of your component.

We'll get to states in just a moment. Because it's such an important subject. States are tied to react components. You should draw a mental association between the component and its state while planning your application. Each component is responsible for having a state.

Above we demonstrated ES5 syntax. I won't tell you which one of the two you should use.

Eventually ES6 is going to become the newly adopted standard within a few years, but many programmers still prefer ES5 formatting. Both still work as JavaScript supports ES5 as a subset of ES6 for compatibility with future applications.

If this works for you, use EcmaScript6 as demonstrated below, now using the new keywords not available available in ES5: **class**, **extends**, **super** and **constructor**:

```
class NewComponent extends React.Component( {
    constructor(props) {
        super(props);
        this.state = { /* initial state */ };
    }
} );
```

It's same thing, except using ES6 syntax. If you're coming from Java, C# or C++ languages, you might be more familiar with this class creation syntax and might prefer it over the outdated ES5 format.

Note another difference here is that we're using **.Component** method to create our component, instead of **.createClass**. But essentially, the same exact thing takes places under the hood.

I know that having two different ways of doing the same thing can sound perplexing. But here you have a choice of using ES5 and have your code execute in all browsers out of the box, or use ES6 syntax, and transpile it back to ES5. However, the advantage of learning the relatively new ES6 syntax is that your application will look cleaner and more intuitive.

It's just the period of time we're faced with right now, as JavaScript specification matures. Until browsers fully adopt EcmaScript 6, studying this syntax format gives you an advantage of learning something that isn't yet in wide use, but that will definitely become the next standard over the next few years.

My personal preference is to start using EcmaScript 6 syntax. This road might be a little rocky but in order stay on top of things when it comes to becoming a better JavaScript programmer, it is a wise thing to do. And these types of things always take hard work and effort. The good news is that once you're past the learning curve, this will become second nature.

Gem 5 - States & Life Cycles

In React, each component has a state. A state is an object with values. A state is exactly what it sounds like. It's the state of that component in terms of one or more flags, parameters and settings usually specified by using numerics, booleans, arrays or maps.

The Life Cycle of a React component has a lot to do with states. And React provides several default methods that help us write efficient applications. These methods are briefly outlined below.

Each component will usually control its own states via following helper methods. They are designed to modify or update component's state at a unique time during the application flow or component's *Life Cycle*.

All of these methods are attached to the component's object like shown below. This is just a brief example demonstrating the placement of these methods within your React object.

Let's take a look!

```
var MyComponent = React.createClass( {

  displayName:              'MyComponent',

  // This is almost like a component's constructor!
  getInitialState:                function() { … },

  // Component mounting events
  componentWillMount:       function() { … },
  componentDidMount:        function() { … },
  componentWillUnmount:             function() { … },

  // Component updating events
  componentWillReceiveProps:      function() { … },
  componentWillUpdate:              function() { … },
```

```
  componentDidUpdate:        function() { … },
  shouldComponentUpdate:     function() { … }

  // Render the component to the DOM
  render: function() { /* return something */ }

});
```

That's a whole lot of ***Will***s and ***Did***s. But what do they all mean? These methods are like the command center of our component! Below I'll provide brief descriptions for which events during the lifetime of a component each one of these methods is responsible for.

getInitialState functions like component constructor. It is called before the component is rendered on the screen.

componentWillMount called before render() method *just once*. It is never called again throughout component's life cycle. Sometimes used synonymously as component's *constructor* method.

componentDidMount called just after the component has been rendered for the first time.

componentWillUnmount called when the component is about to be destroyed. This is a place to clean up memory and delete state.

componentWillReceiveProps is called when props are passed to the component. For example if you have ***Form*** component and ***TextInput*** component that allows user to change its value by typing text into it. Imagine if ***TextInput*** has an ***onchange*** event that passes in some ***props*** to the component. Soon as the new text is entered into the input field, the value is passed to the main component as props. This is the moment when ***componentWillReceiveProps*** is triggered.

componentWillUpdate is called right before the render method is

executed on a component. It gives us a chance to handle any state changes before next render update.

componentDidUpdate is called soon after render() method has finished updating the DOM.

shouldComponentUpdate is a default method that always returns true. It gives us control over whether the component's element should update in DOM. This isn't always the case. For example, if the new results of some state or property in a component equals the value of what it was previously set to anyway, it's probably not worth updating the DOM. Why send the data to the view if it did not change even after being updated?

These methods are crucial to understanding virtual DOM. As we're slowly peeling principles of react programming one by one, it will all start making sense.

For now just know that a react component is usually ***mounted*** to a render target. This is why some of the default methods mentioned above use the word "mount" instead of traditional *setter* or *getter* methods like "set" or "get" you would expect. Mounting is an idea unique to programming with React and the components created within React's virtual DOM ecosystem.

We've already talked about components theoretically. But what is a component in practice? Remember that your application is made up of compartmentalized elements. Components can literally be anything from buttons, text input fields, check boxes to action status areas for displaying text. You will determine the design of your component based on the purpose of your application.

Because we've just discussed methods that deal with "mounting" of components, it is natural to explore what it means in our next gem!

Gem 6 - Component Mounting

React is a library that values programming principles. One of them is the idea of **components** as elements. A component is a programmatic (declarative) way to render what you want in your application.

But React doesn't actually care about where or how you choose to render those elements. Whether it is JavaScript DOM or elsewhere. This can sound surprising at first, because React is known for being primarily as a front end library.

That is because React uses something known as ***render targets***. That is, you determine how the results of a react's render method will be displayed or passed on elsewhere.

In other words, you can say that React is **agnostic** when it comes to the output of data or states stored within its components.

For example… you can output data from a component to file writing function. In other words, React's rendering target is not limited just to UI elements, even though throughout most of this book and react-style programming in general this is what you will be dealing with for the most part.

In previous gem we've talked about how each **component** takes care of its own state. A state could be a boolean flag indicating whether a button is highlighted. Or whether a navigation menu option is selected.

Now, if you've been paying attention this is where it all starts coming together. A react component is a fusion between its **state** and its **rendering** method. We're given programmatic control over the two most important functions of a UI element in one place.

But what in the world is ***mounting***? In React we are using a virtual representation of the DOM for performance reasons.

By simply creating a component, no DOM elements will be even touched. This is great news but how could it be? This is because using native DOM would increase memory imprint of our application.

However with React's ***virtual DOM***, elements are created only when they are required to be used by the application.

I'll explain this in more detail.

React is an isomorphic framework. It's functionality can be mapped to arbitrary output. Not just JSX-style HTML UI elements.

The virtual DOM provided by react library is exactly just that: virtual. It doesn't physically exists within actual JavaScript DOM until it actually needs to be used by the application.

Just think about it this way. The JavaScript's DOM is a physical tree of HTML elements stored in memory. It is populated while the page is loading into your browser.

Usually, when a web page is loaded into the web application, each DOM branch represented by HTML element brackets, loads up into the primary root DOM element regardless of how it will be used, or whether it will even be shown on the screen. The footer will be loaded into DOM, even if it is displayed below the fold in the browser.

React, on the other hand, doesn't even touch that DOM tree. As discussed in one of the previous gems, it adds or removes components from or to ***its own virtual DOM***, which is stored completely in memory. An element becomes physically represented in memory of ***the render target*** usually only when the component is purposely used within your application. It's almost like copying and pasting virtual DOM to a render target, whatever it may be. In the case of web applications, it is of course the JavaScript DOM tree.

I'll demonstrate it with the following example:

```
var component =
React.createElement( ComponentDefinition );
```

At this moment component is an instance of the React type ***ComponentDefinition*** which I made up just for this example. In reality, it would correspond to a component like ***LoginScreen***,

NavigationMenu or ***Button***. And it would contain the standard component definition code we've already seen in earlier gems in the book.

It's simply a React type. It is not yet physically existent anywhere in the primary JavaScript's document (DOM) object. Aside of being a react node it is nowhere else to be found. This is because as of yet it has no physical purpose in terms of being displayed in your application.

But this is true only until **render** method is executed on this component. Here, ***container*** is just some is of a DIV or other HTML element:

```
React.render(component, container);
```

The render method **mounts** this ***component*** to the DOM. The reason this process is referred to as *mounting* is similar to virtual hard drives. If you've ever mounted a virtual CD-Rom or hard drive on your computer, you know how it works. The hard drive doesn't physically exist. For the same reason, this is why this type of software is called ***virtual*** hard drive. Or virtual CD-Rom. And this is why virtual drives have features like "mount this drive."

Likewise, this is the same principle behind virtual DOM in React. Until it is mounted, it cannot be operated upon. This is what makes React so much faster than regular DOM operations. And this is probably one of the top reasons for using React in the first place. React stores a fast in-memory representation of the DOM.

But this virtual DOM is not even associated with the JavaScript's DOM. It exists side by side, and renders into the actual DOM on the screen only when it becomes necessary. Usually, when the component representing an element is actually updated with new incoming data.

Essentially, React is designed to update components on the screen only when they need to be. Another advantage of doing it this way is that it separates your application from the back end, while effectively executing functions that are responsible for updating the screen almost instantaneously as they occur.

What you choose to use as a communication bridge between the back end and the front end is entirely up to you. React doesn't care about it. Its purpose is strictly limited to updating the view as fast as possible.

This ***mounting mechanism*** make React a versatile library. Because at its basic principle, React also doesn't care whether the final result is rendered in the DOM or elsewhere.

You can render React output into an XML document, treat it as custom object data, or return a JSON. It combines familiar programming principles and tries to be "everything for everyone." This is what makes React an incredibly powerful tool for programming User Interfaces as well as fiddling around with custom software implementations.

Now, chances are that if you're coming to React from traditional pre-2015 JavaScript background… unlike studying jQuery, learning React is a much more involved process. It pertains to learning ***software development principles*** rather than learning a set of method names and you're ready to go!

However, it does have a few methods that are becoming standardized among React programmers. We'll explore them as we move forward. Simply knowing them, without understanding the fundamental philosophy behind React is just not enough.

Gem 7 - Node.js & NPM

When I said that studying React involves learning much more than just React library, it was certainly true. At the very least, working with React requires that you install the Node library.

A whole another book can be written just about Node alone, but with regard to React library, we need to install it for several reasons:

1. To provide a ***localhost*** environment for our React application. Installing Node gives you the ability to run your applications from http://localhost on your local machine.
2. Node automatically comes with Babel and JSX support (JSX is installed as a Babel plugin.)

Installing Node and NPM is surprisingly easy on both Mac (via the terminal) and the PC. On the PC it's as easy as downloading the Node installation file from their official website. Run the installation program and you're all set. Just don't forget to check the babel checkbox during installation, so it's also included.

On a Mac, you can download Node via command line. Pop open your Terminal application and type:

```
$ wget http://nodejs.org/dist/node-v0.4.4.tar.gz
$ tar -xzf node-v0.4.4.tar.gz
$ cd node-v0.4.4
$ ./configure
$ sudo make install
```

That's it! The configuration log will run and then you're done.

On Ubuntu, you might want to run this additional command to install essential components:

```
$ apt-get -y install build-essential
```

It looks like at this point node is installed and ready to be utilized for giving us a ***localhost*** http address where we can test our React code.

However, the good news is that if you already have Apache server installed on your localhost machine, and it's mapped to your development folder, using node for providing localhost environment is not necessary. You already have it set up.

In this case, you can just start writing React programs right inside your existing Apache httpdoc folder, or whatever folder it is mapped to on your machine. Just create a new folder in that directory for example "my-react-app", and as usual your project will be available for access via browser under something like:

http://localhost/**my-react-app**

However, it is highly recommended that you do install Node. If for any reason, it's because it comes with command line version of Babel and JSX plugins, which will be required for proper development environment set up.

The installation is really straightforward on a Mac in terminal. But on PC it is just a bit more challenging, especially if you've never done this before.

If you are still unsure on how to do any of this on a PC I recorded a 58-minute supplementary tutorial on YouTube, available for free at the following location:

React JS Tutorial 1 - For Beginners | ReactJS JavaScript ES6 Node.js, NPM, Babel Transpiler, Webpack

https://www.youtube.com/watch?v=tXaNvGcjEi0

It's worth going through regardless of where you are in the development process. But it's a must-watch if you're just starting out.

Watching this tutorial should get you started with writing React applications on a PC via the localhost address. So in that video tutorial I walk you through the entire process, step by step. I don't want to

rewrite the details in this book to take any more space. You can watch it at 2x speed and get through it in just about 25 minutes.

Gem 8 - JSX Walkthrough

If you've started studying React already, you should have heard a lot about JSX. You've also probably heard that it is not necessary to use it with React. And that much is true. But, it is recommended that you do. It just makes things so much easier, even though its syntax can appear a little scary at first to traditional JavaScript developers.

JSX is known for being a language that lets you "write HTML inside JavaScript." This is true to some degree. The desired effect is that we can treat HTML tags as a JavaScript statement. However, JSX actually uses a form of XML which only resembles HTML, because it uses tag names that match those of standard HTML tags, such as DIV, UL, TABLE and so on. Remember that even HTML is just a distant brother of XML.

Although HTML as a JavaScript statement might look a bit dinosauric at first, once you grasp the idea and get used to it, it really becomes second nature. It's definitely something completely new and not usually done (and not even compile in most browsers today) but let's take a look at what it actually looks like:

```
var scoreBoard = { Player1: 10, Player2: 25 };
<div className = "class1" />;
<SomeCounter count = { 3 + 5 } />
<ScoreBoardUnit data-index = "2">
   <h1>Score Leaderboard</h1>
   <Scoreboard className = "results" scores = {scoreBoard} />
</ScoreBoardUnit>;
```

This code naturally will not compile in your standard browser unless Babel's JSX plugin is helping us transpile it back into ES5, the specification we can be sure all modern browsers do understand. In which case, it will compile into following React elements.

```
var scoreBoard = { Player1: 10, Player2: 25 };
React.createElement( 'div', { className: 'class1' }, null)
React.createElement( SomeCounter, { count: 3 + 5 }, null)
React.createElement( ScoreBoardUnit, { 'data-index': 2 },
   React.createElement( 'h1', {}, 'Score Leaderboard'),
   React.createElement( Scoreboard, {
      className: 'result',
      Scores: scoreBoard }, null)
) // close React.createElement( ScoreBoardUnit...
```

These JSX and React code examples can be used synonymously. As long as you have JSX transpiling your code in the background, it will work and you can interchange between the two. And while JSX is highly recommended for combining with React it is not necessary for using with or compiling React programs. Just use *createElement* syntax without JSX, if you must.

However if you wish to get it done right, to get started with the transpiling process, I recommend watching the tutorial from Gem 7 to get familiar with the complexity of transpiling process via Babel.

It takes a bit of patience to set up Babel and JSX, but it's so worth it. Just go through the video tutorial and follow instructions on how to set it up on your own machine.

Once your files are actively transpiled into ES5, you can begin using JSX in your own programs.

As you can see from the example above, this is like writing HTML tags while treating them as JavaScript statements. Of course, this code takes place within some **.js** file itself. This isn't HTML.

Note how **DashboardUnity** is really just an XML tag. It contains children, just as you would expect from any other standard XML code. The reason it works is because it's processed by JSX language processing mechanism. It's a bit like learning a new language, but because it's likely you already know HTML syntax, it's not difficult to

figure it out.

The only primary difference is that curly brackets { } can now also be used right within JSX statements under tag attribute names. The whole tag Scoreboard is treated as an object. And its attribute **scores** is just its property which contains the value specified earlier in code by a JavaScript object **scoreBoard**, which is an JSON-literal object notation for creating objects.

This might seem a bit awkward at first, but this gives you tremendous control over your application. It unifies **document.getElementById** (or jQuery's equivalent of $("#id").attr("scores")) which is the method you would otherwise have to use just to access a property, into a new format.

This format allows not only writing attributes directly into your HTML elements (as JSX) and assign them to already-existing variables/objects, but also treat HTML as a JavaScript statement itself. This is great, because imagine that you can now return HTML code as a JavaScript statement from a function, and then pass it on to some other abstract method in your application. It's like tossing HTML between functions and preparing it for output in one of the containers in your application.

One more thing. You will notice that JSX uses className instead of just "class" as in traditional DOM object. The JSX docs explain it as "Since JSX is JavaScript, identifiers such as **class** and **for** are discouraged as attribute names." So this is why class has become **className**, and for has become **htmlFor**.

To learn more about JSX you should check out the official JSX documentation. Again, another book could be written just about the subject, and we're trying to get the basic out of the way so we can focus on React throughout the rest of this book.

Writing React.js Applications

Welcome to the second part of React Gems. In this section we are

leaving the gem format behind us because the first part of the book was designed to simply get familiar with the most common principles of React programming and modern software development.

This theory will help us investigate the practical aspects of programming with React library. So let's get going and start by writing our very first React component.

Lesson 1 - Writing your first React Component

Below you are seeing a compact example of what the very first React component you might write would look like. It simply renders the generic Hello World message into a DIV element. And it fits on just one line of code.

```
React.createClass({render: function() {
      return (
        <div>Hello World.</div>
      )
   }
});
```

Here, we are using main **React** library object's method ***createClass***. This is how components are created. But what's this code in blue? It looks like HTML, without double quotes surrounding it!

Components are usually stored in variables.

For example:

```
var HelloWorld = React.createClass( { .. } );
```

Storing components in variable names can help us later on when we create parent-child associations between our elements. Which is just a way to programmatically create relationships between nested components.

Some of this code will appear a bit peculiar especially if you're coming from traditional JavaScript programmer background. It's the JSX code highlighted in blue that seems like it doesn't fit in.

But remember that React programming is associated with using additional tools. React library by itself is supposed to be just one part of your entire application. No one writes React apps using React library by itself. It's just one single stone in the wall.

In React we can write HTML inside JavaScript. This is the reversal of the traditional "JavaScript inside HTML" idea we've gotten used to over the years, where we write JavaScript inside HTML's attribute tags (like ***onclick*** = "***this.style.color='red';***") for example.

But React gives us ability to do the opposite. We'll find out why as we move forward. For now let's expand on creating components and then I'll provide a brief explanation how JSX helps us deal with blocks of HTML and treat them as return values.

Having said that... let's move onto the complete example of creating our first React component and rendering its content within a specific HTML element:

```
// Add required library code
var React = react("react");
var ReactDOM = require("react-dom");

// Create our first react component "HelloWorld"
var HelloWorld = React.createClass( {
   render: function() {
          return (
          <div>
          Hello World.
           </div>
      )
   }
});

// Draw "HelloWorld" component inside HTML element whose id is
"app"
ReactDOM.render(<HelloWorld />,
document.getElementById("app"));
```

Notice that our newly created component HelloWorld has its own **render** method. But the ReactDOM object which acts like primary rendering engine also has a **render** method of its own. The correlation between the two is that **ReactDOM.render** calls **render** method of the component it's passed as the first argument. Which, in this case is our HelloWorld component.

Here, we first include the aforementioned React and ReactDOM libraries. You will be required to do that when starting a new reactive application from scratch. This is what initializes React.

Next, once we have React included in our project, we will create a new instance of React class, using the React.**createClass** method. This isn't the only way to initialize a React app but we'll stick to this example in this chapter.

Finally, we call ReactDOM's own method **.render**. And <**HelloWorld**/> here is the variable name **HelloWorld** we assigned the dinosauric React code to. Basically, the React's **render** method takes two arguments.

You simply pass **what** you want to render as the first argument and **where** you want to render it as the second (in this case it's the HTML element whose ID is "app".) So from now on Whatever we assigned to **HelloWorld** variable will be rendered in an element like [div id = "**app**"]

And this is really the basic idea behind React programming. You build components and make them do things within specific HTML elements. Virtual DOM automatically takes care of fast rendering in the front view of your application.

But there is one more thing...

The HTML code highlighted in blue is the part that will be written into the component's rendering area. Yes, we simply type HTML direct into JavaScript statement. Again, this functionality is enabled by installing the JSX plugin from your **node.js** via **babel**.

The process of setting this up is a bit complicated. But this is shown in

more detail in my YouTube tutorials and also later in this book.

Congratulations on creating your first React component! I wanted to keep it simple and not overwhelm you with too much stuff that still needs to take place to set up React. But in my future chapters we will continue to gradually unpack programming with React library and exactly just what it entails.

JSX

In React, writing HTML within JavaScript is achieved by open source JSX engine. You can browse JSX source code by looking it up at the official GitHub repository at:

https://github.com/jsx/JSX

According to the official website JSX is a *statically-typed*, *object-oriented* programming language designed to run on modern web browsers. But when it comes to React, we will be using it in order to ***write HTML tags in JavaScript format***. Which is kind of quirky and unusual. But primarily, this gives us the ability to return blocks of HTML code as though it was a JavaScript statement. Which you will find will prove to be very useful and intuitive.

JSX simply translates HTML syntax back to JavaScript. This might seem really strange at first, but it's like JavaScript now has its own internal HTML processor. This way of thinking is a good starting point of getting familiar with JSX. As we move forward in the book we will explore it in more intricate detail as part of the source code examples.

While tempting, it is completely unnecessary to fully understand how JSX does this. It takes care of this internally, and helps us focus on writing the code. That's exactly what it's for!

JSX does have limitations and it's not entirely pure HTML you'd expect. It's a derivative of XML, not HTML because XML is more strict. Which means it is better for designing libraries. But primarily, because it makes it easier to parse and understand.

JSX does not have an inline CSS parser. So while you can use HTML elements with attributes, ***style*** attribute will not be one of them. You can still use other attributes like ***name***.

In fact, we probably don't really want a full blown HTML processor for our JavaScript code. Because it would make things way too complicated. HTML has too many edge cases. This would make rewriting a complete HTML interpreter like a lot more work and make

working with JSX disorienting. Let's face it, the idea of writing HTML within JavaScript is already a little far fetched as is. But it really gives the leverage we need for writing code faster, by utilizing it primarily for treating HTML as return values and tossing them around between components.

Primarily influenced by original JavaScript DOM API, JSX uses "camel case" name formatting. For example **className**. (Camel case is a name format for properties with first word in lower case and one or more consequent words starting with capital letter.)

The reason className is used in JSX, is because this is what the original JavaScript DOM's property name is. For example:

```
var cn = document.getElementById("my_id").style.className;
```

This is standard JavaScript code. But JSX uses its own virtual DOM system which tries to closely imitate it. It's not trying to reinvent the wheel and create its own original processing language. And so properties follow the standard format we're already familiar with from working with DOM API object in JavaScript.

But there are a few other quirks. For example, you cannot use the "**className**" keyword as HTML attribute name. Let's consider the following example which creates a root component, consisting of just one UL element, and calling it's class "my-list" by assigning it with className property.

```
var root = React.createElement('ul', {
    className: 'my-list' },
child);
```

This is often mistaken for a JSX problem, thinking that JSX prohibits the use of JavaScript reserved keywords. While this is a somewhat true assessment, the problem stems from JSON notation, not JSX. The

common solution for this problem is to use quotes around property name:

```
var root = React.createElement('ul', {
    'className': 'my-list'
}, child);
```

We haven't gone deep into the trenches of React programming to talk about parent and child relationships between components. But because in this example we're using the third parameter ***child***, I wanted to say a few words about what it means.

In the following chapters you'll see how parent and child associations work between components. But basically, for now just know that whatever element or elements you pass as children as 3rd argument in ***createElement*** method, will become available as "**props**" within that element through **this.*props*.*children*** property.

This way you can work with children components from within your parent class. Which is usually the core of your entire app that contains all of the other secondary components. Parents can have multiple children.

We'll talk about what ***props*** are a lot more later. For now, just know that ***props*** (or properties) are to *react components* what ***arguments*** are to a *function*. You simply pass them into the object as custom values required for functionality of that reactive element.

Also remember that you can't paste just about any HTML to use it as your JSX statements. JSX version of HTML is very similar but not 100% accurate HTML specification. It closely resembles it, but there are a few differences.

For reasons like these, JSX might throw you off at first. I struggled with it for a while myself, but when I realized the main purpose was to help write code faster, I thought it made things a lot easier in the long run once I adapted to it and started using it habitually.

It's almost like re-learning years of working with HTML standard. The hard part is knowing limitations of JSX. Because you can get frustrated with not knowing why things don't work when they don't, even though HTML you use in JSX looks properly formatted.

It definitely challenges you to think in a new way about designing some parts of your application. But bear with these example, and eventually it will sink in and start to make a lot more sense.

Props

In my previous chapter I spoke of React ***components***. But there is a lot more to it. Previously, we created a simple component, similar to one that looks just like this:

```
ReactDOM.render(
    <H1>Hello</H1>,
    document.getElementById('root')
);
```

Notice that we are typing HTML tags <H1> right into render's method as an argument without surrounding quotes. This is syntactic style offered by JSX. It normally will not compile in JavaScript, but intelligent IDE's like PHPStorm are aware of JSX syntaxing and will not treat it as an error.

As long as you have JSX included in your project (usually via babel .js library or its counterpart command-line plugin, the latter being the one I recommend using on your production server) this will compile just fine.

Now that we know that this will work only as long as you have babel transpiler (babel.min.js) included in your project. (In order words, don't attempt writing this in Notepad in some folder on your "Desktop" your development environment must be properly set up in order for this to compile, or at the very least babel.js should be included in SCRIPT tag on your page.) It uses internal JSX processor to convert this inline HTML into JavaScript.

Function Purity

Before we go into React props, I just wanted to mention one thing. In React props are read-only. And this has a lot to do with a particular

type of a function called **pure function**. In traditional software development a pure function is one which does not internally change its arguments which are passed to it.

For example,

```
// pure function; does not modify argument values
function sum(a, b) { return a + b; }
```

This is considered to be a pure function, because its body does not modify any of the parameters that were passed as arguments. On the other hand an example of an impure function is shown below:

```
// impure function; modifies value of the passed argument "total"
function sub(total, less) { return total -= less; }
```

In React, **props** take on the "pure" ideology. This means you can pass them, under the condition that your react component will not be changing their values from within itself. This is exactly what makes react props **read-only.** It's good to stick to that rule.

Props are derived from HTML element properties. It's a bit odd, that they are called props. Because in HTML elements, properties are single-keyword flags like DISABLED or CHECKED.

The better name for props would be *attributes*. Because they resemble the name=value pair of HTML **attributes** not props. But, that's just the way it is. React.js developers have chosen this naming convention! Don't shoot the messenger:)

If you've been following our discussion from the previous chapter you should now know what **components** are. They are combinations of HTML, CSS and JavaScript code blocks that determine functionality of one visual area of your application. They are responsible for updating things like feed messages, alerts and various state updates.

We also know that a component usually has a **render** method to display

it in your app.

Working with React.js the next thing you will run into are **props**. Props are like function arguments. Except you pass them to your component, not functions. The principle is the same, however.

Let's create a component and pass a **prop** to it.

```
// Create a Hello component
var Hello = React.createClass({
   render: function() {
      return (
         <DIV>Hello, {this.props.name}</DIV>
      )
   }
});
// Now call "render" method to display this component in your #app
ReactDOM.render(<Hello name = "Jason"/>,
document.getElementById("app"));
```

Here I added a prop called **name** and assigned it value of "Jason". Notice that it's contained within a self-closing tag <Hello **/>.**

Within the component itself, we can refer to it as **this.props.name** to print out its value, instead of generic "Hello World" example from previous tutorial.

Functional Components

Above we have demonstrated a standard React component. It has all of the complexities of one: it's created using **createClass** function, and contains a **render** method. But the following code is actually also considered a valid React component, even though it's nothing more than a standard JavaScript function:

```
function Hello(props) {
   return <H1>Hello, { props.name }</H1>;
}
```

What makes it a valid React component is the fact that it returns a **React element**. React elements can be rendered on the screen. A lot of your reactive code will be returning such statements. It's just a way of bundling HTML elements and returning them as if it was a regular JavaScript value.

This is a big part of React-style programming and it's easy to get used to once you've actually programmed a few of them yourself. At first it will feel awkward, but once you get used to it you'll really see the versatility of this style of programming.

Remember that in JavaScript everything is an object, even functions. This is why components like this are called **functional components**. In a way, it's really just a different way of thinking about JavaScript functions.

This is a valid way to define a React component using EcmaScript5 specification, not ES6 like in the previous example above. However, both are equivalent from the point of view of writing reactive applications.

It's up to you as the programmer to know the differences between ES5 and ES6, which can take a whole book of its own to justify righteously. However, throughout the tutorials in this book, I'll try to use every opportunity to explain the differences between the two.

What's more, let's take a look at a piece of code that accomplishes exactly the same thing of defining a component only this time using syntax of EcmaScript6 specification:

```
class Hello extends React.Component {
```

```
    render() {
        render <H1>Hello, { this.props.name }</H1>
    }
}
```

Note that we're using ES6 ***class*** keyword. We also use the new keyword ***extend*** to derive this object from the standard **React.Component** which is the default component object supplied by React library itself.

A Wolf in Sheep's Clothing

If you know how class inheritance works, well this is how it is imitated in JavaScript. I say imitated, because under the veil, JavaScript ***still uses prototype-based inheritance***. Which can be a blessing and a curse.

Inheritance standards defined by JavaScript are not really real, compared to more mature languages such as C++ or Java. In other words, you can forget creating advanced object structures such as "interfaces" in JavaScript with its loose typing.

Strict programming languages are usually better at providing a fully functional Object-Oriented Programming environment. And here, EcmaScript developers decided to dress up JavaScript syntax a bit to make the wolf look a little more like a sheep.

But most programmers agree that this syntax improves programming in JavaScript. At least visually.

One other major difference of making components this way is access of ***props.name*** property via ***this*** keyword variable. Otherwise it does exactly what the functional component from previous example does.

Native Babel vs JavaScript Babel

I tried to subtly mention this in a previous chapter. But this deserves a brief explanation before we move forward. I don't want you to deceptively see babel at work, without knowing that it might not be properly set up for real-world production environment.

It's best to run babel as a stand alone application in the background, as part of your development environment. This comes with Node installation by default. Simply download Node from Google and keep clicking "Next" button to finish installation.

Adding babel directly via SCRIPT tag is possible and works for testing things out. But because babel's JavaScript library is slower, it is advised to install the native version that works from command line. JavaScript babel library is just slow at processing something like this and defies React programming, one of the sole purposes of which is to optimize your app for speed.

NPM

Command line babel application comes together with Node.js installation. So if you install Node, you will automatically have babel packages. You just need to install them separately from command line using NPM tool, which also comes with Node by default.

While learning React.js, it's perfectly normal to simply add babel via SCRIPT tag. It's just not fast enough for production environment when your app goes live. In any case, all babel does is "transpile" your EcmaScript6 and JSX code back into EcmaScript5, so you can run it in any browser that doesn't have full support for ES6 yet. (None of them do, as of this writing.)

Transpiling Source Code

You've probably heard programmers on forums, Twitter and reddit use the word transpiling but may have not fully grasped the idea behind it.

Usually, transpiling is the process of compiling code in one language into source code in another language. It's just compiling between languages. However, when it comes to JavaScript, we will be transpiling between two different specifications of EcmaScript, 5 and 6.

This gives us the ability to write code in EcmaScript 6, have it transpile to EcmaScript 5, so that modern browsers can actually run your code. Most browsers at the time of this writing do not have support for ES6, not even Chrome.

Eventually, when ES6 support is rolled out for all browsers it will be safe to simply discontinue using babel for the purpose of transpiling between EcmaScript specs. The code will work out of the box by simply writing it within your standard JavaScript files.

But until then, we might as well use this opportunity to learn about a fun source code tool. There are always new standards coming out, and the art of transpiling is probably not going away anytime soon.

* * *

In this chapter I wanted to introduce you to basic principles of React programming that go beyond creation of components alone. Here, I mentioned them briefly in the context of our discussion on components and props. As we move forward, we'll keep going deeper into the principles of reactive programming. This time around, already having brief familiarity with these principles, it will help us advance forward to writing practical React applications.

Lesson 2 - Nested Components

Nested components in React.js help you create more complex view element structures. I have previously talked about creation of React components. And just as a reminder, here is how you would go about creating one.

```
var Component = React.createClass({ render: function(){
    return (
        <Something></Something>
    )
}});
```

This is just a basic component that returns a set of <Something> HTML tags. I know, you're probably still getting used to returning something that looks like an HTML tag without quotes around it, but by this time it should start looking more familiar :)

As you may have imagined, this is the type of code structure you will be working a lot with in React. Most other things stem from it. If you've ever worked with jQuery, this component structure can be comparable to jQuery's object-selector syntax:

```
$(".class").on("click", function() {/*do something*/} );
```

Where we have a callback function return upon the event has finished executing. In react, we're returning an HTML element that will usually be placed inside our application's structure on the front end.

In case of React components, it's just a JavaScript object that contains render function, and has a return statement that returns transpiled JSX code. That's what converts <Something> tag into EcmaScript5 that browsers can understand.

What gets to be in render function and in return statement depends on the

purpose of your application. In this example, for the sake of clarity, let's call these objects **Parent** and **Child** respectively. An actual pair of names if you were writing an application could be something like Friend and FriendList. Or Customer and CustomerProperty.

Parent with a Single Child

Using this pattern, you can implement a basic view functionality.

First, let's create the least building block of the pattern - the child.

```
var Child = React.createClass( {
   render: function() {
   return
<BUTTON onClick = { this.props.onClick }> { this.props.text } </BUTTON>
   }
});
```

In this case the child is a basic component that returns a button element. The button provides an onClick event that will be executed by the method **this.props.onClick**.

Let's now create its counterpart **Parent** class, and use **Child** as the object it will return. I highlighted Child to make sure it's easy to see how the Child component is tied to Parent.

```
var Parent = React.createClass({
   render: function() {
      return (
         <Child onClick = { this.handleChildClick } text = { this.state.childText } />
      );
   },
   handleChildClick: function( event ) {
      // In this method, you can access props
      // that were passed to the Child, using this.state property
      var child_text = this.state.childText;

     // To access target element of the click:
      var target = event.target;
   }
});
```

Notice that you can access child's text from **handleChildClick** method within Parent container. The Parent and Child containers are tied by the props passed into the text attribute on the Child container.

Whatever you pass into the Child (which is located within Parent) will be automatically accessible via the parent container. So props are really what establishes the relationship between the two.

React is a relatively new library. However, it does use one important JavaScript function that you simply must know how to use, if you ever plan on using Parent and Child dynamic in your reactive application.

And this standard JavaScript function is the **map** method, that natively exists on all objects of type Array. If you've ever worked with JavaScript arrays before, you may have at least heard of this function. All it does is creates a new array, calls the callback function on each item, and fills the new array with the result of modifications that the callback function has applied to each item from the original array.

So in short, **Array.map** takes all items and applies a custom operation on them. Then, it simply returns the new, modified data set as a new array. How does this help us in terms of writing Parent-Child components in React library? I can demonstrate this by providing the following example where we will use multiple Children in a single Parent.

Parents with Multiple Children

I'll keep the Child component exactly the same for this example, so we can keep things simple. Notice the new usage of array's map function within the newly updated Parent component:

```
var Parent = React.createClass({
   /* Set defaults for multiple children array */
   getChildrenState: function() {
      return { childrenData: [
         { childText: "Click me 1!", childNumber: 1 },
         { childText; "Click me 2!", childNumber: 2 }
      ]};
   },
   render: function() {
      return (
         var children = this.state.childrenData.map(function(data, index) {
            return <Child onClick = { this.handleChildClick.bind(null, childData) }
                        text = { childData.childText } />;
         }.bind(this));
         return <DIV>{children}</DIV>;
      );
   },
   handleChildClick: function( event ) {
      // In this method, you can access props
      // that were passed to the Child, using this.state property
      var child_text = this.state.childText;

     // To access target element of the click:
      var target = event.target;
   }
});
```

Now, I purposely created these generic examples to show you the association between child and parent, because it's not as trivial as simply nesting HTML elements in one another. In react, this link is established programmatically, as shown in the examples above.

In a real world scenario, you will be building components that represent actual elements in your application, like alerts, friends lists, messages, comments and buttons. But these examples that were shown here are the basic building blocks of reactive programming.

I know it might seem like a hassle to learn all of this, but really, once you program a few of these components, you will see the versatility of React, because it's like embedding custom programming into your everyday HTML elements, having full control over the data flow, and utilizing Virtual DOM which is lightning fast at updating only areas of applications that need updating.

I hope you enjoyed this tutorial and it has proven to be insightful. I talk a lot more about these things in my new book React Gems, which is just about to come out.

Lesson 3 - Handling Component Events

Up until this point we have only looked at basic composition of React programs. We've created components and looked at their internal structure.

In this lesson we will go deeper into the structure and functionality of React Components. In addition to **render**, we will explore three new methods, each executing at a distinct time during the life of a component. This gives us increased creative control over the application.

Always remember that a React component has **state**. The whole purpose of components is to be tied to a custom state represented by that component's object. A **state** is an ***object with data***. But components also have helper functions that make it easier to control how these states will work in a given context.

In this lesson we will take a look at some of these methods.

Method 1 - getInitialState

Each component is in charge of rendering its own state but the method **getInitialState** is always called before our render function. This has similar functionality to an object constructor function. Here we set the default values.

```
var Example = React.createClass( {
   getInitialState: function() {
      return {
         state: 0;
      }
   }
});
```

You'll often see **getInitialState** used throughout React components. It resets the default object tied to the component. Here you could read values from a database or reset state to its default value before rendering this component on the screen.

You can use this method interchangeably with the default **constructor** method provided in EcmaScript 6:

```
class Example extends React.Component( {
   constructor: function() {
      this.state: 0;
   }
});
```

The only other notable difference here is that we're using the **this** keyword to attach state to the main object.

To access the state when the component is rendered on the screen we must use **componentDidMount** method:

Method 2 - componentDidMount

componentDidMount is called automatically by React when a component is rendered. By adding it to your custom React component, you are overloading it. Overloading means, adding your own functionality to the method that already exists in the default React component object, from which you are extending your own custom component. That's the whole purpose of using **extend** keyword. We're extending default functionality of React's component object already provided by the library.

When I first heard of using "mounting" function within my React component, I didn't understand what it meant. Are we not simply attaching a bunch of helper methods to our primary React component?

We've already spoken of mounting earlier in the book. If you still need to brush up on the process, I recommend making sure you've gone over ***Gem 6 - Component Mounting*** previously discussed.

componentDidMount is the method responsible for setting the state after component has been mounted.

```
var Example = React.createClass( {
   componentDidMount: function() {
      this.state = 1;
   }
});
```

It simply means that the virtual DOM has become associated with a DOM node that will be actually updated on the screen. This method is always called after executing **render** command from the component. They are synonymous and can probably be used interchangeably. It ensures and tells us that the element actually already exists within virtual DOM. The **render** method has already been called on it!

Method 3 - componentWillUnmount

And finally, **componentWillUnmount** is called just before the component is removed from the page and is about to be destroyed.

```
var Example = React.createClass( {
   componentWillUnmount: function() {
      this.state = null;
   }
});
```

Notice, we're resetting the state here back to ***null***. The component is destroyed and we no longer need its data. This is the place where you would clean up memory used by your component, if any. For example uninitialize timers.

Example of a Complete Application written in React

We've covered enough principles to write a complete React application. I intentionally tried to make all examples in this book simple. Complexity usually comes from getting creative with programming, and if I just dumped a large program into this book, you it would probably have a discouraging effect rather than motivational.

Here, I will build a simple search app that looks up results from an array set. If the typed characters match anything on the list, they will be filtered out and all other entries will be dismissed.

This app is a lot like the Google search auto-look up feature where items drop down as you type your phrase into the query input box. Below I'm listing the full source code.

Here, I am not going to include Cascading Style Sheet part of the app. This can be taken care of separately. We want to focus strictly on the React implementation.

The app is surprisingly compact in comparison to everything you'd have to do using standard JavaScript code, provided what it actually does.

Creating Search List

Let's create the list of potential search results by using object literal format in JavaScript:

```
// First, let's define the list of all searchable animals
var libraries = [
   { name: 'Wolf', url: 'http://localhost/wolf'},
   { name: 'Panther', url: 'https://localhost/panther'},
   { name: 'Fox', url: 'http://localhost/fox'},
   { name: 'Giraffe', url: 'http://localhost/giraffe'},
   { name: 'Elephant', url: 'http://localhost/elephant'},
   { name: 'Ferret', url: 'http://localhost/ferret'},
   { name: 'Lion', url: 'http://localhost/lion'},
   { name: 'Lioness', url: 'http://localhost/lioness'},
   { name: 'Tiger', url: 'http://localhost/tiger'},
   { name: 'Tigress', url: 'http://localhost/tigress'},
   { name: 'Cat', url: 'http://localhost/cat'},
   { name: 'Dog', url: 'http://localhost/dog'},
];
```

Writing the React Application

```
// Let's create our search component
var SearchComponent = React.createClass( {

   // Reset input element to a blank space
   getInitialState: function() {
      return { searchString: '' };
   },

   handleChange: function(e) {
      // Commenting the following line out, will result in the text box not changing its value, because in React input boxes cannot change independently of the value that was assigned to it. In this case it's the this.state.searchString.
      this.setState({searchString: e.target.value});
   },
```

```
  // Render this component
  render: function() {

    var libraries = this.props.items,

      searchString = this.state.searchString.trim().toLowerCase();

    If (searchString.length > 0) {
      // We are searching. Filter the results.
      libraries = libraries.filter(function(l){
        return l.name.toLowerCase().match( searchString );
      });
    }

    // Return this component as a JSX statement
    return <div>
            <input type = "text" value={this.state.searchString} onChange =
{this.handleChange} placeholder = "Type here" />

            <ul>
              { libraries.map(function(l) {
                return <li>{l.name} <a href = {l.url} > {l.url}</a></li>
              }) }
            </ul>
          </div>;
  }
});

// Render the SearchComponent component on the page
ReactDOM.render(
  <SearchComponent items = { libraries } />,
  document.getElementById('container')
);
```

Place this code into the main ***.js*** file of your application and run it. It will result in basic drop down functionality. Typing partial animal names will filter them out from the main list. Ideally, you would load this list from a database or some other data storage mechanism.

As you can see, depending on the type of an application you are designing (and in this case, the application is very simple) you may or may not use any of the mounting or updating methods just based on the

nature of the elements your component represents.

I hope this provides enough ammunition to start writing your own React application.

Just as the case is with JSX, programming in React assumes knowing many different things. One of those things is the EcmaScript 6 specification. This is important because ideally, as part of learning how to make React applications you want to transition from old-style ES5 syntax to ES6, the standard that in about one to two years will be normally supported in all modern browsers such as Chrome, Firefox and Safari without having to transpile the code by using additional tools.

The rest of the book will deal with going over some of the most important changes to JavaScript language that were applied in EcmaScript 6.

EcmaScript 6 - Things You Need to Know

When React was created, it started out when ES5 specification was in the spotlight. It was still new and fresh in 2015 and even throughout 2016.

However, by 2017 it started to progressively adapt to ES6 syntax. And this is why it's important to talk about the significant changes that were applied to this new standard while studying React.

When you are learning React, you are learning a lot more than just React. It works in ES5 just fine, but that syntax is becoming outdated. And this is where many programmers can get puzzled, because essentially the same library might not look the same in all the different examples you're seeing on the Internet and in books.

If you don't know React, chances are you probably don't know much about EcmaScript 6 at all. I know this is a dangerous assumption to make, but I think it can be justified by things you will learn in this chapter.

There really isn't anything wrong with ES5. But it isn't going away anytime soon. And I think when writing educational books at a time when the industry is still transitioning from one specification into another it's important to occasionally juggle between the two.

But we've come to a place of the book where we will now investigate the primary differences and additions to JavaScript in EcmaScript 6 specification. At the end of the book I will also provide cheat sheets for many of the subjects discussed here.

The var and let keywords

In ES6 it's commonly agreed that instead of using ***var*** keyword we now use ***let*** keyword. There are reasons for this, which are described below:

- **There is no hoisting when using let keyword.** If you're familiar with JavaScript's hoisting mechanism, you will spot a difference when using the new **let** keyword.

  Hoisting is when JavaScript moves all variable definitions up the scope in which they are defined. This little-known process is done behind the veil. But it is in fact true:

```
{
console.log( a ); // undefined; but only because it is hoisted
var a = 1;
}
```

  And the **let** example:

```
{
console.log( a ); // ReferenceError: "a" is not defined at all
        let a = 1;
}
```

In the example above the result coming out of console when making an attempt to print out value of a variable is undefined. In first case, it's because while the variable is hoisted. But in the second example, because it is not.

- **In global scope.** The ***let*** keyword and the old ***var*** keyword are still scoped to the *global scope*. But the ***let*** keyword is not attached to the global window object:

```
var    a = 1;                  // will be attached to global window object
let    b = 1;                  // will not be attached to global window object
console.log(window.a);  // 1
console.log(window.b);  // undefined
```

❖ **In function scope.** The ***let*** and ***var*** keyword are still identical within function scope:

```
function helloThere() {
let x = 1;// identical scoping
var y = 1;      // identical scoping
}
```

❖ **In for statements.** The ***let*** keyword is only visible inside for-loop while ***var*** is visible to entire function's scope. So, the ***let*** keyword limits visibility of variables defined within ***closest enclosing block*** specified by { and } brackets. Below, let's first take a look at how ***var*** traditionally handles scoping:

```
// var inside a function and for-loop example
function helloThereIterator() {
        // i is theoretically visible here too.
for (var i = 0; i < 10; i++) {
// i is visible here.
        }
 }
```

Perhaps this is not the best example of **var** keyword (because the for loop is where it's initially defined) because while it is available everywhere in the function. But it still demonstrates the principle.

Traditionally, var is known to exist for access within the brackets { } of a function and hidden from function's scope but only unless it is the window object. This is the basic rule. But JavaScript is complicated like that... and if the variable is not found within the function itself, only then the **window** object is checked, which is outside of the function itself.

It is your responsibility as a programmer to determine whether that variable should or should not be available from within the global scope. This is an example of where JavaScript tries to be "everything for everyone."

While this looseness is helpful in many cases and makes the learning curve

almost non existent, in the long run you will bump into these weird cases, especially when your program reaches certain length in large projects.

So now, let's take a look at the same example. The only thing we did was swap the **var** over with **let** keyword. This produces different scoping expectations.

```
// let example
function helloThereIterator() {
        // i is not visible here
for (let i = 0; i < 10; i++) {
// i is visible here, and in callbacks within this scope itself
        }
 }
```

- Redeclaration is not allowed with **let** keyword

The **var** keyword (in strict mode) will let you redeclare its own value in the same scope. But this is not true with **let** keyword:

```
'use strict';
var a = 1; // Ok.
var a = 2; // Ok. a is now 2;
```

```
'use strict';
let a = 1; // Ok.
let a = 2; // Syntax error: identifier a  is already declared.
```

- The ***let*** keyword can also be used to avoid problems with closures. It will bind a new value instead of keeping a reference to the old one.

```
"use strict";
for (let i = 1; i < 10; i++) {
$.ajax("url": "script.php", function(msg) {
console.log(i); // the variable i has lost its original scope here, it's no longer associated with the i variable defined in the for loop
        } );
}
```

This code demonstrates a common JavaScript problem that happens

quite a lot when using the **var** keyword in a for loop in combination with an ajax call. Here, because i is seemingly redefined within ajax's callback function, it is no longer associated with the same **i** variable that was defined within the for loop.

This classic problem is usually avoided by passing i variable into the function as a parameter, needlessly creating a new instance of the variable.

However, this is not true when **let** is used to defined the same loop. It effectively resolves this common problem. Here is the same example, only this time using **let**.

```
"use strict";
for (var i = 1; i < 10; i++) {
$.ajax("url": "script.php", function(msg) {
console.log(i); // here i is the one that was originally defined in for loop
        } );
}
```

This definitely saves many headaches associated with for loops and callback functions.

❖ Moreover, the **let** keyword allows us to keep the value of the variable to the scope it was defined in, as shown in this isolated example:

```
{
let  a = 1;
};

console.log( a ); // Reference error: a is undefined
```

❖ In other words, it can be said that a variable defined using **var** keyword is known throughout the function in which it is defined, since the moment it is defined.

Whereas the **let** keyword is only known by the scope block in which it is

defined and all other scopes within it even if it's a callback function. This doesn't solve any major software problems but it certainly helps us get things done in a more intuitive way, especially when dealing with the web of multiple scopes in our script's program structure.

The new const keyword

Const is exactly like **let**. The only difference is that once you define a value using **const** it cannot be redefined again later on in your program. Doing this will produce an error:

```
let a = 1;
a = 2; // this is ok, we simply reassigned the value of a variable

const a = 1;
a = 2; // error, this is a constant and cannot be reassigned!
```

How is this useful? Constant variables are traditionally used in computer programs to define values that will not change at runtime throughout your application. They are great for flag values and app configuration settings that should not change.

Oh, and one more thing… When it comes to objects only… **const** keyword, while prevents re-assignment does not actually make the assigned object immutable. In contrast to many other languages, making an object const would make its properties immutable as well.

Not so in JavaScript.

Destructuring assignment

EcmaScript 6 introduces us to ***destructuring assignments***. This is new JavaScript functionality. The previous explanations of let keyword laid the groundwork for discussing this next subject. So let's take a look at how they work:

```
let {a, b} = obj;
```

In this example two variables **a** and **b** are created. Then, properties **obj.a** and **obj.b** are taken from the object named **obj**, and assigned to those variables respectively. So this is the equivalent of doing something like:

```
let a = obj.a;
let b = obj.b;
```

This format takes a lot more space. *Destructuring assignments* save space and provide better readability for our code. Get used to them! :)

For-of loop

This is a completely new type of a loop in EcmaScript 6. Let's take a look at how it works:

```
for (let [key, value] of map) { ... }
```

This mighty for loop will iterate through values stored in map. Unfortunately (like many things in JavaScript) the **for-of** loops do not work on objects (and their properties). This is almost a bewildering design decision. So, what do **for-of** loops can work on?

The for-of loops are designed exclusively to work with **Array**, **Map** and **Set** objects only.

The Map object

Just like an Array object, ES6 adds a new one, called Map. Maps are nothing more than a simple key/value map. They are a lot like associative arrays in PHP.

Maps should not be used everywhere. Only where you previously used objects as ***collections***. That's what they're made for. And using objects should still be primary to your JavaScript program design.

Maps have a ***.size*** property which determines its size. Unlike objects, where you would have to calculate the size of an object manually. Use them together with for-of loops when you have collections of data paired by key/value association!

Arrow Functions

One of the most favored features in EcmaScript6 is the arrow functions. I think you should start using them at earliest opportunity. Arrow functions are just JavaScript functions written in a more compact syntax and that are slightly different than regular functions created using the function keyword. Here is the basic arrow function syntax format, which at first may seem a little odd:

name = (params) => returned expression

For example, you can write something like this using an arrow function:

x = (a, b) => { **console**.log("This is arrow function x with params = " + a + " and " + b) };

But this is only a function definition. It can be executed just like a regular function as follows:

x(5, 7);

One radical difference compared to regular JavaScript functions is in how arrow functions treat the **this** keyword. It solves fundamentally the same (or similar) problem that **let** keyword solves for regular scopes. We need to get into a slightly more involved discussion here.

JavaScript ***arrow function*** is a feature relevant only to EcmaScript 6. It did not exist prior to this specification. It adds convenience to making anonymous functions within objects easier to use while retaining the value of the ***this*** object inside anonymous event handlers.

Arrow functions also modifies the functionality of the ***this*** object used within nameless callback functions within object scope definition by automatically binding it to topmost local scope of the object. It reminds me a little of hoisting (I briefly talk about hoisting in my other tutorial).

Arrow functions help us escape ***this/that*** gymnastics (If you're not familiar with what I'm referring to keep on reading, this tutorial explains it) inherent to

original scope model we've been used to.

Arrow functions are also known as "fat arrow" functions due to their syntax.

```
(params) => { statement };
```

Just think of it as a different syntax of defining a JavaScript function.

The rest of this tutorial will explain the difference between an older JavaScript example and how arrow functions can be used to improve code readability and internally tweak object functionality.

The aim of arrow functions is to solve a fundamental problem that stems from original JavaScript language design dealing with object scoping and the automatic ***this*** keyword binding.

As someone who has developed a canvas-based game engine library in JavaScript that required creation and instantiation of Line, Rectangle, Sprite and many other useful objects (a library to aid the development of a browser-based computer game) like many other web developers, I have faced the following issue.

Let's consider we have this simplified version of an object called Sprite for making game character and enemy object sprites!

Also let's create a new "ship" object of type Sprite and instantiate it.

```
/* A significantly simplified Sprite object in JavaScript  */
var Sprite = function()
{
   this.angle = 5;
   this.draw = function() {
      alert(this.angle); // display sprite's angle
   }
};
```

```
var ship = new Sprite(); // instantiate ship

ship.draw(); // draw the ship with this.angle rotation angle
```

Notice the ***draw*** function. In this simple example we output the current angle of rotation for our sprite. In real world circumstance the code would be actually more complicated. But this is just an example to demonstrate arrow functions in JavaScript.

I know that programming the inner mechanics of JavaScript functions have always eluded programmers. The desire to "experiment" with how things work and go by feelings and hacky results is tempting. In the example above the "this" keyword is the same within ***Sprite's own scope*** and scope of its ***draw*** function.

We specified 5 in Sprite's scope. Then by referring to it from Sprite's ***this.draw*** function we simply pull its value. Sounds easy! And works as expected. But let's not forget about anonymous event functions (also known as callback functions) within JavaScript objects.

Shall we take a look now? I am building on top of the code we already saw in the previous example. So I am repeating it here. But there is a new addition. I added ***setTimeout*** function, which takes an anonymous function and executes it in 1 second (1000 milliseconds) after Sprite object is instantiated. This is normal behavior of a JavaScript function-object constructor.

Notice ***setTimeout*** now encompasses the following line:

```
alert(this.angle); // display sprite's angle from setTimeout event function

/* A significantly simplified Sprite object in JavaScript  */
var Sprite = function()
{
   this.angle = 5;
   this.draw = function() {
     alert(this.angle); // display sprite's angle
   }
```

```
  // New! The following code will execute as part of the
   // object construction process:
   var timer = setTimeout(function() {
       alert(this.angle); // display sprite's angle from here
   }, 1000);
};

var ship = new Sprite(); // instantiate ship

//ship.draw(); draw the ship with this.angle rotation angle
```

I temporarily removed **ship.draw()**; method by commenting it out. This way only the code highlighted in green will be executed, when Sprite object is instantiated.

The "this" keyword ***inside an event handler*** is automatically assigned to the object on which that event happened. Let's run this program and see what happens, in comparison to our previous example.

The alert message will now return:

Undefined

That's because ***this.angle*** within ***anonymous function() { ... }*** no longer refers to the "this" object from Sprite object. And there is no such property as "angle" attached to the anonymous function (that was passed as an argument to ***setTimeout*** function.)

```
/* A significantly simplified Sprite object in JavaScript  */
var Sprite = function()
{
   this.angle = 5;
   this.draw = function() {
     alert(this.angle); // display sprite's angle
   }
  // New! One solution is to create a "that" variable still in this scope
```

```
    var that = this;

   // The following code will execute as part of the
    // object construction process:
    var timer = setTimeout(function() {
        alert(that.angle); // display sprite's angle from here
    }, 1000);
};
var ship = new Sprite(); // instantiate ship

// ship.draw(); draw the ship with this.angle rotation angle
```

All I did in this example was assign ***this*** object to the newly created ***that*** variable within main Sprite object's scope. Before ***timer*** was defined. So in reality we now have something like ***this.that*** within Sprite.

I also changed ***this*** to ***that*** within ***setTimeout***(function() { ... });

Let's run our code this time and see what happens now:

5

The value of this.that is 5. This happened when we bound ***this*** object to variable called ***that***.

I am sure you have used the trick I just described in the examples above before. If not, well I'm glad you know about it now. Most programmers discover it by instinct. Because it takes time to learn the difference in JavaScript scoping rules between function objects and anonymous events within those functions.

And after all these explanations with a big smile on our face this finally brings us to our big final question!

What are Arrow Functions in JavaScript?

And what is the difference between an arrow function and anonymous functions in JavaScript?

An arrow function in JavaScript is an *anonymous* function with shortened syntax. The only two differences vs standard anonymous function is in syntax and how ***this*** object is treated within its scope. A small detail, which slightly changes in how the two actually operate when it comes to binding of this object.
Let's compare a regular JavaScript anonymous function with an arrow function equivalent. We already know what anonymous functions like ***function(){...}*** in JavaScript look like. But arrow functions actually have a distinct syntax.

Notice that both do exactly the same thing, execute some code. Except ***arrow functions*** automatically take care of the cumbersome this/that juggling we have just experienced in the previous examples. Arrow functions create their own local ***this*** object that doesn't change. So we never have to rename or bind it to another variable such as ***that****, like in previous example.*

This is the new syntax in EcmaScript 6 that defines new ***arrow function*** in JavaScript:

```
() => { };
```

A word of caution. ***Arrow functions*** in JavaScript ***do not work with "new" operator***. They are anonymous and are usually used for events.

But there also exist syntactic variations:

```
(a, b) => { ... }    // Multiple parameters
   a => { ... }    // Single parameter
```

Note that params can include a set of multiple parameters separated by comma. In which case you must always use parenthesis. For single parameter syntax, you don't have to use parenthesis.

In a real world-scenario we can also assign arrow functions to a variable name. In this case the anonymous arrow function becomes named. The function ***returns the result*** from within its body's scope.

```
var add = (a, b) => { return a + b; }
```

Having said that...

```
add(1, 2); // returns 3
```

But there is another, scope-less variation syntax for defining arrow functions:

```
var add = (a, b) => a + b;
```

Notice that the return value here is represented by a ***regular JavaScript statement***.
This will produce exactly the same result. We removed ***return*** statement. But the function will still return the value of ***a + b***. In this case a + b is a *JavaScript statement*, without a scope body. By default, an arrow function specified without a scope returns the value of the *statement* even if return keyword is not specified.

And finally let's put it all together and rewrite our original example of the Sprite object. Only now using an arrow function. You can see that there is no necessity to create and set binding of ***this*** to ***that*** variable anymore. Arrow functions take care of that for you, for us!

```
/* A significantly simplified Sprite object in JavaScript  */
var Sprite = function()
{
   this.angle = 5;
   this.draw = function() {
     alert(this.angle); // display sprite's angle
   }

   var timer = setTimeout( () =>
   {
```

```
        // note the use of arrow function here
        alert(this.angle); // display "5" using "this", not "that".
    });
};

var ship = new Sprite(); // instantiate ship

// ship.draw(); draw the ship with this.angle rotation angle
```

We simply switched standard anonymous **function(){...}** commonly used as the event handler.

And replaced it with arrow function **() => {...}**

From that moment on ***this*** keyword within the new anonymous event function started to correctly refer to ***this*** keyword from the original Sprite object. Without us having to write any extra binding code.

And so... I think this concludes our discussion of arrow functions. I hope you enjoyed this tutorial. Please share it with your friends, on Twitter or on your Facebook page.
Developer Post is a small website dedicated to simple explanations of subjects that otherwise appear complex. And hopefully this helps someone out there write better JavaScript code.

If you want to learn more about ES6 Harmony, head to What's new in EcmaScript 6 aka Harmony (opens new tab) to get familiar with the basics.

According to this Sitepoint article, arrow functions are the most favorite feature in ES6.

This Mozilla arrow function tutorial helped me become familiar with arrow functions. I used it as my first reference while writing this article.

Immediately Invoked Arrow Functions (IIAF)

If you are familiar with jQuery or JavaScript plugin development, you've probably indirectly come across IIAF's and have already seen the often obscure syntax of Immediately Invoked Function Expressions (IIFEs) that looks like this:

```
( function () { ... }() );
```

Here we execute the function at the moment of its definition.

You will find this syntax used often at the bottom of the HTML page just above the closing body tag. This technique is used to wait until the HTML elements have finished loading until starting to execute JavaScript.

In the same way, arrow functions can be used to achieve equivalent result.

```
( () => { ... }() );
```

This syntax is referred to as Immediately Invoked Arrow Functions. The parentheses surrounding it make the function execute at that place in your program's execution flow.

Backtick (``) Template Strings

Backtick is just another name for the quirky quote character, also known as ***grave accent***.

Statements like **${ expression }** can now be interpolated by using backtick template strings. It's simply a string surrounded by the third set (after single and double quotes) of quotes located just to the left of "1" key on your keyboard.

One of the best things about backticked strings is that they can be split over multiple lines. A much needed feature that was lacking in JavaScript.

These types of strings are also called ***template literals***. Below several examples are shown:

```
`string text`;
  `string text on line one
 Can continues here without semicolons`;

`another text string with $(expression) within it`;
```

```
`string text`;

// just like regular quotes,
// grave accent quotes can also be back-slashed, see below:
// the following statement will return true

`\`` === "`";
```

You can also use both regular single quote (') and double quote (") characters within grave accents.

New Array methods

The following are the most important additions to the Array objects in ES6.

Let's define our common Array object as follows:

```
let arr = new Array();
```

In ES6, we have these new array operation methods:

find

```
arr.find(callback[, thisArg]);
```

Returns the value of the first item which, when passed to callback, produces a "truthy" value.

findIndex

```
arr.findIndex(callback[, thisArg]);
```

The **findIndex** method is exactly the same as **find**. The difference is that it will return the ***index*** of the first item which produces a "truthy" value, not the value.

fill

The **fill** method initializes or "fills" all of the elements at every index of an array. It can initialize or set only a sector of your array by taking optional parameters ***start*** and ***end***:

```
arr.fill(value[, start = 0[, end = this.length]]);
```

By default start is ***0*** and end is the ***length*** of the array.

copyWithin

This new array method copies a sequence of items ***within itself*** as an array, starting at the position indicated by ***start*** and starting from index indicated by ***target***:

```
arr.copyWithin(target, start[, end = this.length] );
```

Here is a practical example:

```
var arr1 = [1,1,1, 2,2,2, 3,3,3];

// copy first 3 indices to last 3
arr1.copyWithin(6, 0, 3);

alert(arr1); // 1,1,1, 2,2,2, 1,1,1
```

This code, apparently copies the "***1,1,1***" sequence, starting at ***0*** index in the array and copies it over to the last 3 indices, that start at ***target*** index of **6**. This is a lot like memory operations in C++, except with arrays.

New Built-in Classes

Built in classes are the pre-existing objects that come packaged with JavaScript. For example the familiar ones are ***Object***, ***Array***, ***Function***, ***Date***, ***Math*** and a few others… EcmaScript 6 adds several new built in classes which are listed below:

1. **Map** a set of key/value pairs. Almost like an array.
2. **Set** is like map, except where each stored value is unique.
3. **Symbol** is used to make private object/class properties.
4. **Promise** manages an event that will take place at a future time.

Promises are interesting because all events usually take place at a future time anyway. What's so different about a promise? The proper explanation would be to say that they will take place regardless of whether the structure of your

HTML document changes. These special events, called "promises" guarantee that the event will take place at some time in the future.

One of the problems promises solve is the classic case when HTML element's events (such as onclick) are overwritten by code that updates visual aspects of your application. Because this happens often in an actively updated front-end view, some events might not be triggered at all, just because DOM was rewritten.

A promise can exist "in the future" regardless of what happens to DOM structure of your site, so long as the integrity of the original DOM remains the same, or similar to one that existed at the moment when the event was executed.

Thank You

This ***tutorial book*** was independently published by me and distributed to subscribers of my free JavaScript newsletter. This couldn't have been a possibility without your support.

Have you ever tried writing and pushing your own books? Hours of research and mental processing goes into crafting a technical book!

This is why, I wanted to dedicate this section to just saying thank you for buying a copy of ***React Gems***. As you may know, your purchase includes free life-time book updates.

You will continue receiving book updates including new chapters automatically, every time I add, modify or edit significant parts of the book which I do plan doing throughout 2017.

An updated version of this ***tutorial book*** will be sent to the PayPal e-mail address you used to buy this edition. New updates happen every 1-2 months or upon availability (a quality update takes time to make) as I gather more educational material.

Free updates are already included with your purchase and you don't have to do anything to continue receiving them.

Thanks for reading and your continued support!

Author,
Greg Sidelnikov
greg.sidelnikov@gmail.com

Book update 1 January 6th, 2017

Edited and improved chapters:
"A note from the author" (improved text)
"Main Roadblocks to Learning React" (deepened insight on JSX and babel)
"EcmaScript 6 - Things You Need to Know" (fixed ***let/var*** examples)
Book length change: +2 pages.

Book update 2 January 10th, 2017

Edited and improved chapters:
Fixed typographical error on page 37 (changed of to or)