Guía sobre React

Comenzando a Programar Interfaces Web con React

9 de agosto de 2021

Índice general

Entorno de Desarrollo

Para comenzar a programar en React recomendamos Visual Studio Code (VS Code). Y para ser más productivo, especialmente si eres nuevo en React, sugerimos que instales la extensión VS Code ES7 React / Redux / React-Native / JS snippets que proporciona la generación rápida de estructuras sintácticas de JavaScript y React. También sugerímos instalar Prettier, que es un formateador de código JavaScript / React.

Para instalar una extensión, en Visual Studio Code, vaya al menú Archivo, luego Preferencias y luego Extensiones. Verá un cuadro de búsqueda que le permitirá encontrar las extensiones que desea instalar.

Finalmente, realmente recomiendo configurar VS Code para formatear los fuentes al momento de guardar. Puede hacerlo yendo al menú Archivo, luego a Preferencias y luego a Configuración. En el cuadro de búsqueda, escriba .Editor: Formato al guardar"(Format On Save). Esto formateará su código inmediatamente después de guardarlo.

Parte I

Introducción

Capítulo 1

Conceptos Esenciales de JavaScript

Si deseamos dominar React y eso significa comprender por qué y cómo funcionan ciertas cosas, debemos aprender algunos conceptos específicos de JavaScript.

En este capítulo explicaremos los conceptos de JavaScript y las construcciones sintácticas necesarias para crear un camino sólido que nos permitan dominar React. Si desea profundizar en más detalles sobre algunos de los temas que se explican aquí u otros sobre JavaScript, recomendamos que visiten el sitio web Mozilla [1]. Esta sección se basa en conceptos explicados allí. Habiendo dicho eso, comencemos.

De la web Developer Mozilla JavaScript Documentation [1], JavaScript se define como:

"JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado justo-a-tiempo (just-in-time) con funciones de primera clase. Si bien es más conocido como un lenguaje de scripting (secuencias de comandos) para páginas web, y es usado en muchos entornos fuera del navegador, tal como Node.js, Apache CouchDB y Adobe Acrobat. JavaScript es un lenguaje de programación basada en prototipos, multiparadigma, de un solo hilo, dinámico, con soporte para programación orientada a objetos, imperativa y declarativa (por ejemplo programación funcional)."

Si Usted es un desarrollador Java, C# o C++, esta definición puede sonar un poco intimidante. Es por ésto que es necesario aprender algunos conceptos, especialmente aquellos que no están disponibles de forma natural en los lenguajes compilados. Para comenzar con esos conceptos, primero explicaremos las construcciones básicas del lenguaje, luego explicaremos lo

que significa que un lenguaje tenga **funciones de primera clase** y sea **basado en prototipos**, **multi-paradigma**, **de un solo subproceso** y **dinámico**.

The JavaScript language is governed by a standard under the responsibility of ECMA[2]. ECMAScript is the name of the language specification. The standardisation allows vendors to write interpreters and developers to be able to run their programs on any vendor interpreter. So, it is a great thing. Particularly, in 2015 there was a major release known as ES6, ECMAScript 6 or ECMAScript 2015. Most of the syntactical constructions that we will study in this section were implemented in this release.

Lets then begin learning. Any piece of code written in this section will run using the node interpreter, invoked using the console in the following way:

```
$ node yourjsfile.js
```

Using Visual Studio Code, you can go to the menu "Terminal."and then "New Terminal". It will open a small terminal window where you can run your scripts.

Printing text on the screen must be the very first thing you learn every time you start playing with a new programming language. This book is not the exception. You can print text on the screen using the *Console* object like the following example:

```
1 console.log("Coding in React!");
```

Console was created mainly for debugging purposes, it should not be used in production. It gives you access to the Browser's debug console. It is also not part of the standard, but most modern browsers and nodejs supports it.

1.1. Variables

Lets move to **variables**. You can declare a variable using the *let* keyword:

```
1 let myFirstVariable;
```

Declaring a variable without initialising it will assign the *undefined* special value to it and that is what you will see if you print it. Try it!. Lets give it an initial string value:

```
let myFirstVariable = "Hello value!";
```

Now if you print it you will see the string. You can also declare a variable using the *const* keyword:

```
const myFirstConst = "Hello constant value!";
```

As you might have guessed, declaring a variable with *const* will not allow you to change the value of the variable once it has been initialised. If you do it the interpreter will thrown an error. Try it!.

JavaScript is a **dynamic language**, among many other things that we will discuss later, that means that the type of a variable can be change it at runtime. Opposed to static (or compiled) languages where the type of a variable is defined at compile-time and cannot be changed during execution.

```
//my type is string
let changeMyType = "Hello String!";
//now it is number
changeMyType = 100;
```

1.2. Functions

Lets move now to **functions**. In the following code snippet we are declaring a function and after that we are calling it:

```
function saySomething(string) {
  console.log(string);
}

saySomething("Hello Function!");
```

If you just need to have a function that gets called as soon as it is declared, you can use the syntax below. This is called IIFE (Immediately-invoked Function Expression).

```
(function saySomething(string) {
  console.log(string);
})("Hello Function!");
```

In JavaScript functions always return something. If you don't explicity return something from the function using the *return* keyword it will return *undefined*.

```
let x = saySomething("Hello Function!");
//x is undefined
```

In addition, functions are **first-class** objects. The most well known first class object of programming languages are *variables*. Variables are denominated first class objects because it can be assigned, it can be passed as argument to a function or method, it can be returned from a function, etc. So, having functions as first-class objects means that all those things that you can do with variables are possible with functions too. See at the example below where on line 6 we are assigning the function to the variable `say`. And then on line 10 we are using it to invoke the function.

```
function returnSomething(string) {
  return "This is it: " + string;
}

//assigning a function to a variable
let say = returnSomething;

//calling the function
returnSomething("Hello js!");
say("Hello again!");
```

Both `say` and `returnSomething` points to the same place which is the first statement in the body of the function. In the next example, on line 12 we are invoking a function and passing the `returnSomething` function as argument. Note how then is invoked on line 8 and its return value returned.

```
function returnSomething(string) {
  return "This is it: " + string;
}

//receives a function as parameter
//invokes it and return the value
function saySomethingMore(fn) {
  return fn("Hey !");
}

//passing a function as argument
saySomethingMore(returnSomething); //"This is it: Hey !"
```

Functions can also be asigned to variables just in its declaration as the next example illustrate:

```
//assigning the function
const returnSomething = function (string) {
  return "This is it: " + string;
};

returnSomething("Hey !"); //"This is it: Hey !"
```

JavaScript provides another and a bit less verbose way to declare functions called **arrow functions**. Let's see some examples:

```
//arrow function with no parameters
const arrowf1 = () => {
  return "arrowf1 was invoked!";
};

//arrow function with one parameter
//parenthesis is not necesary here
const arrowf2 = param => {
  return "this is the argument: " + param;
};

//arrow functions with one statement
//in the body won't need return
const arrowf3 = (a, b) => a + b;
```

1.3. Arrays

Arrays is another very important construction that we will use massively. This is how you can declare an array:

```
//an empty array
let empty = [];

//an array
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
```

The elements of an array can be accessed by its index, where the index of the first element is 0.

```
//an array
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
```

```
family[0]; //Jóse
family[1]; //Nicolas
family[2]; //Lucia
family[3]; //Enrique
```

Adding an element at the end of the array:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

//adding an element at the end of an array
family.push("Pablo");
```

And if you want to add the elements of an existing array to another non empty array, you can use what is known as *spread syntax*:

```
let myParents = ["EnriqueR", "Susana"];
let JoseParents = ["Eduardo", "Graciela"];
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
let all = [...myParents, ...JoseParents, ...family];
//[
//  'EnriqueR', 'Susana', 'Eduardo', 'Graciela',
//  'Jóse', 'Nicolas', 'Lucia', 'Enrique'
// ]
```

Spread syntax is also available for functions to accept an indefinite number of arguments:

```
function restParams(param1, param2, ...params) {
 //params is [3, 4, 5]
}
restParams(1, 2, 3, 4, 5);
```

To simply iterate over an array you can use the following `for` construction:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
for (let element of family) {
  console.log("regular for: ", element);
}
```

And in addition we have a set of very useful methods. Let's see first how we can iterate over an array using the `.forEach` method:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

family.forEach(function (value, index, array) {
  //value is the element being processed
  //index is the index of the current value
  //array is the entire array
  console.log(value, index, array);
});
```

Note that the `.forEach` method accepts as parameter a function that accepts three parameters. `value` which is the element being processed, the `index` which is the index of the value being processed and `array` which is the array that we are looping. If you are only interested in the elements you can just do this:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

family.forEach((value) => {
  //do something with the value here
});
```

Another very interested method is `.filter`. Similar to the previous one, it receives a function with the same parameters. It will return a new array (shorter or equals than the original) with the elements that evaluates to true to a condition.

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

const members = family.filter((member) => {
  return member.length > 5;
});

//members = ['Nicolas', 'Enrique']
```

Note that we are passing an arrow function to the `.filter` method with a condition testing the length of each element in the `family` array. Those element whose length is greater than 5 will be part of the returned new array. Also note that the family array is not changed at all.

The last method we will see is `.map`. This method receives a function, same as the previous two methods, and it will return a new array with the result of applying the function to every element. It will always return an array of the same length as the one we are processing. As we will see later, `.map` is very used in React to add style markup to the elements of arrays.

```
let numbers = [1, 2, 3, 4, 5, 6, 7];
const double = numbers.map((element) => {
  return element * 2;
});
//double = [2,  4,  6, 8, 10, 12, 14]
```

Array methods can be combined to produce the desire results. Look at the example below. We are applying first the `.filter` function to get an array only with odd numbers filter out from the numbers array and to that we are applying `.map` to transform it into even numbers.

```
let numbers = [1, 2, 3, 4, 5, 6, 7];
const chain = numbers
  .filter((element) => {
    return element % 2 !== 0;
  }) //[1, 3, 5, 7]
  .map((element) => {
    return element * 2;
  });
//chain = [2, 6, 10, 14]
```

If you have an array with few elements, instead of working with indexes, there is a very convenience way called **destructuring** that allows you to assign each element of the array to named variables. See below:

```
let [one, two, three] = [1, 2, 3];
//one = 1
//two = 2
//three = 3

//same as the previous
let fewNumbers = [1, 2, 3];
[one, two, three] = fewNumbers;

//and here using spread syntax
let [a, b, ...rest] = [1, 2, 3, 4, 5];
//a = 1
//b = 2
//rest = [3, 4, 5]
```

1.4. Objects

There are several ways to create objects in javacript. We will study those that will be used further in the book when coding in React. The first way to create objects that we will see is called **Object Literal**. And object literal is created wrapping within curly braces a collection of comma-separated *property*:*value* pairs.

```
1  //an object literal
2  let mi = {
3    name: "Enrique",
4    surname: "Molinari",
5    sports: ["football", "tennis"],
6    address: {
7      street: "San Martin",
8      number: 125,
9    },
10   allSports: function () {
11     console.log(this.sports);
12   },
13 };
14 //this is an empty object
15 let obj = {};
```

As you can see an object literal can be composed not only of simple property-value pairs but also for arrays, other objects like *address* and functions (called methods). Additionally, since ES6, you can create object literals with what is called *computed* property names, like shown below on line 6:

```
1  let aproperty = "phone";
2  //an object literal with a computed property name
3  let mi = {
4    name: "Enrique",
5    surname: "Molinari",
6    [aproperty]: "+54 2920 259031"
7  };
```

Every time the JavaScript interpreter evaluates an object literal a new object is created.

You can access the properties of an object using the **dot notation** as the example below shows:

```
1  console.log(mi.name); //Enrique
2  console.log(mi.sports[0]);//football
3  console.log(mi.address.street);//San Martin
4  console.log(mi.phone);//+54 2920 259031
5  mi.allSports(); //invoke the function and prints the sports array
```

You can add properties (and remove too) dynamically to an object. In the example below, on lines 3 and 4 we are adding the properties `x` and `y` (with their corresponding value) to the `obj` object.

```
1  let obj = {a: 1, b: 2};
2  //add properties to the obj object
3  obj.x = 3;
4  obj.y = 4;
```

The spread syntax also works with objects, see below:

```
1  let obj1 = {
2    a: 1,
3    b: 2,
4  };
5  let obj2 = {
6    c: 3,
7    d: 4,
8  };
9  let obj3 = { ...obj1, ...obj2 };
10 //obj3 = { a: 1, b: 2, c: 3, d: 4 }
```

And if you want to create an object from some declared variables, you can do this:

```
1  let a = 1,
2    b = 2;
3  let obj4 = {
4    a,
5    b,
6  };
7  //obj4 = { a: 1, b: 2 }
```

So far we have seen object literal syntax and what you can do with it. But what if you don't know how many object you will need to create? You need something like a *class* from class-based languages like Java or C++.

In JavaScript, we have what is called **constructor functions**. As we will see later, JavaScript have added classes to the language, but they are just syntactic sugar on top of functions.

A constructor function name by convention start with a capital letter. Lets see how to create and use them:

```
1  function Book(name, authors, publishedYear) {
2    this.name = name;
3    this.authors = authors;
4    this.publishedYear = publishedYear;
5    this.fullName = function () {
6      return this.name + " by " + this.authors + ". " + publishedYear;
7    };
8  }
9
10 thisBook = new Book("Coding in React",
11                       ["Enrique Molinari"], 2021);
12 thisBook.fullName(); //Coding in React by Enrique Molinari. 2021
13
14 archBook = new Book("Coding an Architecture Style",
15                       ["Enrique Molinari"], 2020);
16 archBook.fullName(); //Coding an Architecture Style by Enrique Molinari. 2020
```

As you can see, the function `Book` looks like a class's *constructor* of a class-based language, in which, in addition, we are able to declare right there methods like `fullName()`. We define properties and we initialise them with the function parameters on lines 2, 3 and 4. On line 5 we define a method. After that, on lines 10 and 14 we are creating two instances of two different books and then invoke the `fullName()` method. Constructor functions is the syntactical construction that JavaScript offers to create object oriented software.

1.4.1. A Prototype-Base Language

Now that we know how to create objects literals, constructor functions and create instances from them, it is time to explain what it means for JavaScript to be a **prototype-based** language. Prototype-based languages are a style of object oriented programming in which objects are created without creating *classes*. That is why they are also called *classless* languages, in contrast to *class-based* object oriented languages (like Smalltalk, Java, C++ and C# to name a few). In prototype-based languages there are no

classes, just objects. We don't have that difference between classes and objects. That difference between a *static* definition of a blueprint (a class) and their inheritance relationship (which cannot be changed at runtime) vs the *dynamic* instantiation (object creation). And not having classes vs objects becomes evident in some situations like the ones we will see next. Defining methods in constructors functions in the way we did before it is not ideal due to for each instance that we will create we are adding the method `fullName()` to it. This can be illustrated with the code below:

```
thisBook = new Book("Coding in React",
                          ["Enrique Molinari"], 2021);
archBook = new Book("Coding an Architecture Style",
                          ["Enrique Molinari"], 2020);
//printing thisBook
//Book {
//  name: 'Coding in React',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2021,
//  fullName: [Function (anonymous)]
//}
//printing archBook
//Book {
//  name: 'Coding an Architecture Style',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2020,
//  fullName: [Function (anonymous)]
//}
```

As you can see in the previous example code, the two instances of the Book constructor function includes, in addition to the the property names (and their values), the function implementation code. Source code is not shared across the instances like it is in class-based languages. This is an implementation detail of the language which if you are not aware of it might lead to inefficient programs. And what about *inheritance* which is a valuable language resource used by developers? If there are no classes, do we have inheritance? Yes, we have something similar to the classic inheritance that you know from class-based languages. The difference, among others, is that this relation is dynamic, meaning that the inheritance relationship in prototype-based languages can be changed at runtime (as opposed to class-based languages where inheritance is a static relationship that cannot change at runtime). Here is where we have to introduce the concept known as

prototype. We can deal with the duplication in the definition of the method and implement inheritance using it.

Each object in JavaScript can have a **prototype** object, to *inherit* properties and method from it. If you call a property or method in an object and is not defined there, it will **delegate** that call to its prototype. Since that prototype object might have a prototype object too, this delegation will follow until is found or fail with an error. This is called as **prototype chain**.

Each constructor function have access to a special property called prototype, that can be accessed using dot notation: `Book.prototype`. And when you create an instance, you can also access (while in general is not necessary) to this property using: `thisBook.__proto__` or which is the same: `Object.getPrototypeOf(thisBook)`.

Knowing this we can improve our `Book` constructor function defined above in the following way:

```
function Book(name, authors, publishedYear) {
  this.name = name;
  this.authors = authors;
  this.publishedYear = publishedYear;
}
Book.prototype.fullName = function () {
  return this.name + " by " + this.authors + ". " + this.publishedYear;
};

thisBook = new Book("Coding in React",
                        ["Enrique Molinari"], 2021);
archBook = new Book("Coding an Architecture Style",
                        ["Enrique Molinari"], 2020);
//printing thisBook
//Book {
//  name: 'Coding in React',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2021
//}
//printing archBook
//Book {
//  name: 'Coding an Architecture Style',
//  authors: [ 'Enrique Molinari' ],
//  publishedYear: 2020
```

```
25 //}
```

In the example above we have defined the method `fullName()` in the prototype of the constructor function (line 6). After that, on lines 10 and 12 we are creating two instances and then printing them. Now as you can see the `fullName()` method is not there because it now belongs to their prototype object, shared by the two `Book` instances: `thisBook` and `archBook`. So, what happen if we execute the following:

```
1 thisBook.fullName();
```

JavaScript will first find the `fullName()` method in the `thisBook` instance. As it is not defined there, JavaScript will then look in their prototype object and because its there it will be called.

Every prototype chain will end up points to `Object.prototype`. So, if you execute the following:

```
1 thisBook.valueOf();
```

JavaScript will try to find the method `valueOf()` in the `thisBook` instance. Then on their prototype and finally on the prototype object of their prototype, which is `Object.prototype`. The method is there, so it is called.

Lets now create a basic inheritance example. We are going to create a new `EBook` constructor function that will inherit from `Book`.

```
1  function EBook(filesize, name, authors, publishedYear) {
2    Book.call(this, name, authors, publishedYear);
3    this.filesize = filesize;
4  }
5  let eBook = new EBook(2048, "Coding in React", ["Enrique
   ↪  Molinari"], 2021);
6
7  //printing eBook:
8  //eBook:  EBook {
9  //  name: 'Coding in React',
10 //  authors: [ 'Enrique Molinari' ],
11 //  publishedYear: 2021,
12 //  filesize: 2048
13 //}
```

Above we have defined the `EBook` constructor function. On line 2 we are invoking the `Book` constructor function using the `call` method which allows

us to set the value of `this` as the current object. This is, somehow, analogous to the use of `super(...)` inside a constructor in a class to instantiate the parent class and initialise their private members. On line 5 we are creating an instance of `EBook` and if we print the instance on the console we can see that now we have all the properties from the `Book` constructor functions on the `eBook` object. However, we don't have yet the `fullName()` method that was defined in the `Book.prototype`. To inherit that method in the `EBook` instances we have to set the `Book.prototype` object as the prototype of the `eBook` instance. We do that below:

```
1  Object.setPrototypeOf(eBook, Book.prototype);
2
3  //Another way of doing the same as above is this:
4  //thisEBook.__proto__ = Book.prototype;
5  //However __proto__ is deprecated
```

`Object.setPrototypeOf` is a method where the first parameter is the instance to have its prototype set and the second parameter is the prototype object to be set.

1.5. Classes

Yes! JavaScript has classes and their syntax is pretty similar to most of the class-based languages you might know. They were added to the language in 2015 as part of the EcmaScript 6. The thing is that classes in JavaScript are a syntactic sugar on top of constructor functions and prototype inheritance. Behind the scene, everything works like a prototype-base language, even if you define instances from classes. That is why it is important to understand the previous sections.

We will implement our `Book` and `EBook` constructor functions with prototype inheritance from the previous section but using classes.

```
1  class Book {
2    constructor(name, authors, publishedYear) {
3      this.name = name;
4      this.authors = authors;
5      this.publishedYear = publishedYear;
6    }
7
8    //this method gets added to the Book.prototype
9    fullName() {
```

```
10        return this.name + " by " + this.authors + ". " +
          ↪  this.publishedYear;
11      }
12    }
13
14    class EBook extends Book {
15      constructor(filesize, name, authors, publishedYear) {
16        super(name, authors, publishedYear);
17        this.filesize = filesize;
18      }
19    }
```

What we did in the above example with classes, the `EBook` and `Book` inheritance relationship is the same as what we did with the `EBook` and `Book` constructor functions in the section before. See the following code that demonstrate this:

```
1   let ebook = new EBook(2048, "Coding in React", ["Enrique,
    ↪  Molinari"], 2021);
2   //Book.prototype is the prototype of the ebook instance
3   console.log(Book.prototype.isPrototypeOf(ebook)); //true
4   //fullName method is found on the prototype
5   console.log(ebook.fullName()); //Coding in React by Enrique,
    ↪  Molinari. 2021
6   //EBook is a function not a class
7   console.log(typeof EBook); //function
```

In the example above, we first create an instance of `EBook` and then on line 3 we verify that `Book.prototype` is the prototype of the `ebook` instance. This means that the inheritance relationship was implemented as prototypes just like functions.

1.6. The Multiple Meanings of *this*

It is important to understand how `this` works in JavaScript as it depends on where it is used, how it behaves. We have been using `this` in the examples from the previous sections in constructor functions and in classes. And we have not mentioned anything about it because in those examples it behaves just like you know from class-based languages like Java or C#. However, there are some details you should know specially if you your classes for your React components.

So far, if you use `this` in constructor functions and classes, and creates the instances using the `new` keyword, `this` is bind to the object being instantiated. However, specifically with constructor functions this won't work as expected if you just call the function like in the next example:

```
1  function Constr(param) {
2    this.param = param;
3  }
4
5  Constr(2); //this is global object window
6  console.log(window.param); //prints 2
```

Calling the constructor function as we are doing on line 5 will bind `this` to the *window* global object. So, this example works perfect, but what it does is probably something you don't expect. This example will end up adding the `param` property to the *window* object and assigning it the value 2.

So, `this` will bind to the *window* global object if you use it in regular functions. As we have seen, in classes works as expected, it will bind to the object being instantiated. However, if you need to assign or pass as an argument a method from a class that method will lose the binding of `this`. Let study the next example:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4    }
5    saySomething() {
6      console.log(this.name + " is talking...");
7    }
8  }
9  let enrique = new Person("Enrique");
10 enrique.saySomething(); //Enrique is talking...
11
12 let o = enrique.saySomething; //assigning to a variable
13 o(); //TypeError: Cannot read property 'name' of undefined
```

In the previous example we are defining a class `Person`. Then on line 9 we are creating an instance `enrique` and on line 10 we are invoking the `saySomething()` method. When the method is invoked since `this` is bind to the object `enrique`, `this.name` which was initialised to the `"Enrique"` string will works and prints ."Enrique is talking...". However, on line 12 we

are assigning the method to a variable and then using the variable to invoke the method on line 13. At the invocation of the `saySomething()` method (by the `o()` call), `this` is `undefined`, it is not bind to the object `enrique`. Then we will get a *TypeError* message saying that `name` is not a property of `undefined`.

In order to fix this, we have to explicitly bind the value of this as we see next:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4      this.saySomething = this.saySomething.bind(this);
5    }
6    saySomething() {
7      console.log(this.name + " is talking...");
8    }
9  }
10 let enrique = new Person("Enrique");
11 enrique.saySomething(); //Enrique is talking...
12
13 let o = enrique.saySomething; //assigning to a variable
14 o(); //Enrique is talking...
```

On line 4 above we are explicitly binding the value of `this`, which in the constructor is the object being instantiated, to the `saySomething` method. In this case when `saySomething` is invoked by the `o()` call it will work as expected. Another way to fix this is by declaring the methods as *arrow functions*.

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4    }
5    saySomething = () =>  {
6      console.log(this.name + " is talking...");
7    }
8  }
9  let enrique = new Person("Enrique");
10 enrique.saySomething(); //Enrique is talking...
11
12 let o = enrique.saySomething; //assigning to a variable
13 o(); //Enrique is talking...
```

Defining the method using the *arrow function* syntax will work because arrow functions retain the `this` value of the enclosing lexical scope which in this case is the class. However, arrow functions in classes will not get added to the prototype of the object being instantiated, which means, as we have already discussed, that every instance will have its own copy of the method.

1.7. Modules

The 6th edition of ECMAScript in 2015 has added, in addition to classes, the possibility of defining modules. Before this release there were other options to create modular JavaScript programs using tools like RequireJS [4], among many others. Now we have this functionality supported natively by modern browsers.

Its use is pretty simple. You can define functions, classes, objects, constants in a JavaScript file and `export` those that you want to be reused by others. And in addition a client module must `import` those that needs to reuse. Lets see some code examples.

```
1  //this is my complex-module.js module
2
3  export function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  export let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 export class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
```

In the module *complex-module.js* we are exporting a function, and object and a class. We can also write the same using a different syntax:

```
1  //this is my complex-module.js module
2
3  function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
21
22 export { obj, ASimpleClass, complexThing };
```

Everything can be `exported` at the end just like we are doing on line 22 above. Of course, you don't have to export everything from a module, just those abstractions that represent the public API of your module. Let's see below how you can `import` the abstraction from the *complex-module.js*.

```
1  //this is my main-module.js module
2
3  import { complexThing, obj, ASimpleClass } from
   ↪  "./module/complex-module.mjs";
4
5  //calling the imported function
6  complexThing();
7
8  //printing the imported object
9  console.log(obj);
10
11 //instantiating the imported class
12 let o = new ASimpleClass("Enrique");
13 o.print();
```

As you can see on line 3 we are importing the three abstractions that the module exports. And the we are just able to use them as if the were declared in the *main-module.js* file.

There is a common practice to define a `default export` abstraction from a module in order that client module can import those a bit easier. In the code below, on line 22, we are exporing the class as our default exported abstraction from the module.

```
1  //this is my complex-module.js module
2
3  function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
21
22 export default ASimpleClass;
23 export { obj, complexThing };
```

On line 3 below, note how we are importing the default exported abstraction with a different name (it is an alternative, but we can use the same name). Note also that there are no curly braces.

```
1  //this is my main-module.js module
2
3  import AClass from "./module/complex-module.mjs";
4  //This line below is the same as the one above
5  //import { default as AClass } from
   ↪  "./module/complex-module.mjs";
```

```
6  import { obj, complexThing } from
   ↪  "./module/complex-module.mjs";
7
8  let o = new AClass("Enrique");
9  o.print();
10
11 //... more code here
```

1.8. Single Thread Language

As per the definition about JavaScript we gave at the beginning of this chapter, we know that JavaScript is a **single threaded** language. This means that the execution of a statement is one at a time. However, as we know, JavaScript supports *asynchronous* operations, like Ajax calls. So, how does this work? The thing is that the JavaScript interpreter receives some help from the Browser, where somehow, as we will see later, are returned to the JavaScript interpreter as *callbacks*. There are a bunch of operations that the JavaScript interpreter delegates its execution to the Browser. These operations are called **Web APIs**. Among these Web APIs, we can find events (onclick, onmouseover, etc), fetch and XMLHttpRequest (ajax calls), setTimeout, etc.

To handle this, the execution environment of JavaScript has use: a *call stack*, the browser's Web APIs, a *callback queue* and its *event loop*. Lets see the following simple example:

```
1  console.log("starting");
2
3  setTimeout(() => {
4    console.log("callback");
5  }, 1000);
6
7  console.log("finishing");
```

Here is the list of how this program is executed in a browser:

1. Statement on line 1 is pushed on to the *call stack* and executed. `"starting"` is printed on the console.

2. The `setTimeout` on line 3 is delegated to the browser's Web API to be executed. Which basically wait for one second. However execution of the program continue, does not wait because the execution was delegated to the browser's Web API.

3. Statement on line 7 is pushed on to the *call stack* and executed. `"finishing"` is printed on the console.

4. After the one second elapsed from the `setTimeout` function, the callback arrow function passed as the first argument is then pushed into the *callback queue*. Since there are no more statements to be executed on the *call stack*, the *event loop* get from the top of the *callback queue* the arrow function and push it into the *call stack*. Finally it gets executed. `"callback"` is printed on the console.

Note that all the callback functions that end up in the *callback queue* gets executed after the call stack is empty and not before. To be very clear with this, look a the example below.

```
1  console.log("starting");
2
3  setTimeout(() => {
4    console.log("callback");
5  }, 0);
6
7  console.log("finishing");
```

Note that on line 5 we are passing `0` seconds to the `setTimeout` function telling the Web API to not wait to push the callback arrow function into the *callback queue*. In any case, the result and the order of the console messages is the same as the example before: `"starting"`, `"finishing"`, `"callback"`.

We can expect exactly the same behaviour from the example below that perform an ajax call:

```
1  console.log("starting");
2  fetch("https://jsonplaceholder.typicode.com/posts/1")
3    .then((response) => response.json())
4    .then((json) => console.log(json));
5  console.log("finishing");
```

`fetch` is delegated to the Web API which performs an ajax call. Once the server respond, the callback arrow functions are pushed into the *callback queue*. The first callback function, on line 3, transform the response obtained from the server to json and the next callback function on line 4 prints that json in the console. Only after printing on the console the text `"finishing"` on line 5, is when those callbacks functions are pushed into the *call stack* and executed. You can find a more detailed explanation about the JavaScript interpreter and how it works, in the great talk by Philip Roberts[5].

1.9. The Promise Object and the async/await Keywords

The Promise Object was introduced in Javascript in ES2015. This object represents an asynchronous operation that might finish successfully or fail. See below how to create an instance of a Promise:

```
1  let p = new Promise(function (resolve, reject) {
2    //function to be executed by the constructor
3  });
4
5  //do something with p
```

As we can see above, the Promise constructor receives a function, called *executor*, that will be invoked by the constructor. The *executor* function receives two additional functions as parameters. The body of the *executor* function perform typically an asynchronous operation and finish by calling the `resolve(value)` function if everything goes well or `reject(reason)` otherwise. See how this is done bellow:

```
1  let p = new Promise(function (resolve, reject) {
2    //long async operation
3    setTimeout(() => resolve("finished"), 1000);
4  });
```

On the example above using `setTimeout` we are simulating an operation that takes one second to finish. After that operation finish it will call the `resolve` function passing as value the string `"finished"`. What can we do with that then?

The Promise object have the `then(handleResolved)` method that receives a function that allows you to work with the parameter passed when you invoke the `resolve` function like we do on line 3 below:

```
1  let p = new Promise(function (resolve, reject) {
2    //long async operation
3    setTimeout(() => resolve("finished"), 1000);
4  });
5
6  p.then((value) => console.log(value));
```

As we can see above, on line 6 we are passing the `"finished"` string as parameter named `value` to the function passed to the `then(handleResolved)`. Then that value is just printed. What is important to note here is that the `handleResolved` function passed to the `then(...)` method is executed only once the promise is resolved.

Suppose now something goes wrong withing the executor, then it can be handled in a different way, see below:

```
1  let p = new Promise(function (resolve, reject) {
2    //long async operation
3    setTimeout(() => reject("can't be done..."), 1000);
4  });
5
6  p.then((value) => console.log("success: " + value))
7   .catch((value) => console.log(value));
```

Note that on line 3 now we are calling the `reject(reason)` function passing the `reason` value as the string `"can't be done"`. Then, on line 7 note how the function passed to the `catch` method is the one that gets called.

During this book, and usually in React we don't write promises, but we use them frequently. By using the `fetch` method to retrieve data from an external API, we have to deal with a promise. Look at the example below:

```
1  function fetchPost() {
2   fetch("https://jsonplaceholder.typicode.com/posts/1")
3    .then((response) => response.json())
4    .then((json) => console.log(json));
5  }
6
7  fetchPost();
```

As you can see on line 2 above, the `fetch` method call returns a promise, which allows us to call the `then` method on it, to work with the fetched data.

Another way to write this is by using the `async` and `await` keywords. These keywords were added to Javascript on ES2017 allowing us to write asynchronous code in a synchronous way. Let's rewrite one of our previous examples to take advantage of these keywords. First we will create a function that return a promise. Functions that return promises are (usually) asynchronous:

```
function thePromise() {
  return new Promise(function (resolve, reject) {
    //long async operation
    setTimeout(() => resolve("finished"), 1000);
  });
}
```

To call these functions the `await` keyword is used provoking to stop the execution until the promise is `resolved` or `rejected`. The call code must be a function declared with the `async` keyword. See below:

```
async function testingKeywords() {
  console.log("before");
  const data = await thePromise();
  console.log("after");
  console.log(data);
}
```

Note that we declared the function `async`, line 1, and we use the `await` keyword before calling the asynchronous function on line 3. It is important to understand that the order of the messages printed on the console is the same as the order of the sentences inside the `testingKeywords()` function, as opposed to what we have discussed before on section 1.8.

To finish this section we are going to rewrite the `fetchPost()` function to use these new keywords. See below:

```
async function fetchPost() {
  let data = await
   ↪  fetch("https://jsonplaceholder.typicode.com/posts/1");
  data = await data.json();
  console.log(data);
}

fetchPost();
```

As you can see, again, we are declaring the function `async`, and in this case the `fetch` call on line 2, is prepended with the `await` keyword. Prepended the `await` to the `fetch` function means that instead of returning a Promise object, it returns (if the promises resolve, you can use a try/catch block to handle errors) a Response object. That allows us to call on line 3 directly to the `json()` method of the Response object. As that method returns a

Promise, we also prepend the sentence with the `await` keyword, giving us a Javascript object that is finally printed on the console.

It is important to note that `await` can only be used inside an `async` function.

Parte II

Understanding React

Capítulo 2

Essential React Concepts

In this chapter we will learn the core concepts behind react.

2.1. React Principles

I have to start saying that I *love* React! I love it because I love designing software in a professional way. With that I mean with practices that keeps application's source code *modifiable* after several years of changing it. We say a software is modifiable, if I know on every change, where that change will impact. The best feature of React is their components construction. Yes... components. That piece of software very well described in text by several books, but very rare provided by programming languages in a good way. React components challenge the design pattern found in most of the best known frameworks and libraries, which are based on logic separated from markup,by using template languages to style data. React components suggest that UI logic and markup must be naturally grouped in the same abstraction. And that is their power, what makes React so great and the reason I love it. And among other features provided by React, understanding how to build application by assembling components is the main goal of this book.

After reading this book, I really recommend to see the explanation about react design principles by Pete Hunt: React - Rethinking Best Practices.

2.2. Creating a React Project

To start with React we will first create a React project using a tool called Create React App. With this tool as you will see, you can create a starter

React project without configuration. In order to use this tool, install on your local development machine the latest LTS version of Node.js.

After that, run the following command on the console to create your first React project:

```
$ npx create-react-app coding_in_react
```

When finish, if everything went fine, you will see a message like:

```
Success! Created coding_in_react at /home/react/coding_in_react
```

Run the following commands to start your application:

```
$ cd coding_in_react
$ npm start
```

That will open your default browser with the URL http://localhost:3000/, with a sample application running. Lets now open VS Code to see what is inside our project folder. To do that, you can type on your console the following:

```
$ cd coding_in_react
$ code .
```

This will open VS Code with the project folder `coding_in_react` ready to be used. Lets have a look at the project folder structure below. There is a brief explanation of what is each item.

```
coding_in_react
├── node_modules This folder contains packages installed by the
│                Create React App tool.Any package that we install
│                will end up here.
├── public This folder contains the index.html file (among others)
│          which creates the main skeleton layout of your web
│          application.
├── src This is the folder where your React source files will reside.
└── package-lock.json Here you will find the exact version of each
                      of the packages (and the dependencies of the
                      packages) that your project depends on.This file
                      should be changed only by using npm commands.
```

```
├── package.json Here you will find the packages that your project
│                depends on.This file should be changed only by
│                using npm commands.
└── README.md This is a Markdown file that contains documentation
              about the project.
```

In the next section we will start coding in React using this project. I will refer to the folders and files there when necessary.

2.3. React Components

React applications are built by creating components. Component is probably one of the most confusing terms in software engineering. So, we will provide our definition of what a component is in React. A React component is a self contained piece of software that is created by using the JavaScript `class` definition or the JavaScript `function` definition. It manage its own state and ideally perform a single task. It might collaborate with other components and knows how to paint itself on the browser.

In the previous paragraph I said that React components knows how to paint itself in the browser and they can be defined using the `class` or the `function` syntactical constructions. If you use a `class` you have to add a method called `render`, which is the method invoked by the React runtime to paint the component in the browser. And if you use a `function`, you just provide there the implementation of its render.

Lets create our first React component using a JavasScript `class`:

```
1  import { Component } from "react";
2
3  export default class Person extends Component {
4    render() {
5      return <p>This is a <strong>Person</strong>
       ↪  Component</p>;
6    }
7  }
```

You first need to `import` the `Component` class from React Core, because your JavaScript classes must `extends` from it. And finally you have to define the `render` method that is in charge to paint the component on the browser. Note that this method just `return` HTML (or at least looks like HTML as we see later). See below the same component as the on above but using a `function` definition instead of a class:

```
1  export default function Person() {
2    return <p>This is a <strong>Person</strong> Component</p>;
3  }
```

2.4. Rendering Components

I have my first component, how do I get it render into the DOM? To answer to this question we will learn some React concepts.

Take a look again at the component we have created in section 2.3, the "HTML"snippet that the function component return (or the render method in the clas based component) is not really HTML. It is called JSX, which stands for **JavaScript XML**. It is a *syntax extension* to JavaScript and what the React team recommends to use to paint your components on the browser.

In JSX, you can embed any valid JavaScript Expression between curly braces. As an example, below you can see the variable `name` declared and initialised (on line 2) and used inside the JSX syntax (line 6).

```
1  export default function Animal() {
2    let name = "Eze the Dog";
3
4    return (
5      <p>
6        This is <strong>{name}</strong>
7      </p>
8    );
9  }
```

Browsers does not understand JSX syntax. To make it work we have to translate JSX into JavaScript, using a compiler like Babel. If you create your React application using the `create-react-app` tool like we did before, you get this covered and you can forget completely about this.

Once this compilation is done, JSX gets translated into a JavaScript expression which at execution time, is evaluated along with the expressions defined by you in curly braces and painted on the browser.

As JSX are expressions, it is possible to see a JSX piece of code as a first class citizen. Which allows you to do, for instance, what you see below:

```
1  function passingAsArgument(jsx) {
2    return jsx;
```

```
3   }
4
5   export default function Animal() {
6     let name = "Dog";
7     //assign a JSX block to a variable
8     let jsx = passingAsArgument(
9       <p>
10        This is a <strong>{name}</strong>
11      </p>
12    );
13
14    return jsx;
15  }
```

In the example above, we are calling a function passing a JSX expression as an argument and the return of that function (which is just this same argument) is assigned to the variable `jsx` on line 8.

JSX is also the way you use to render your own components. In the VS Code project we have created in section 2.2, create a JavaScript file called `src/Person.js` with the `Person` component we have created in section 2.3. Then, open the file `src/index.js` and paste the following:

```
1   import React from "react";
2   import ReactDOM from "react-dom";
3   import Person from "./Person";
4
5   ReactDOM.render(
6     <React.StrictMode>
7       <Person />
8     </React.StrictMode>,
9     document.getElementById("root")
10  );
```

The `src/index.js` file is our JavaScript main module (the entry point), so it is the place to describe the UI of our application. In the main file we have to call the `ReactDOM.render` function, which expects as their first argument a JSX expression, and then the DOM element in which the JSX expression will be rendered. If you check the `public/index.html`, you will see the markup `<div id="root"></div>` we are referencing on line 9 above.

> When you you create a React project with `create-react-app` as we did, there are several things that happens unders the hood. One is

the translation of JSX to JavaScript as we mentioned. But there are few others that are important to understand. `create-react-app` uses Webpack, a static module bundler to help improve the performance of your application by combining multiple static files into one, which reduces the number of request the browser requires to do to the server to get static assets. Specially when you build a single page application with JavaScript tools like this becomes very important. So, Webpack unders the hook, takes the `src/index.js` as the entry point to produce the bundler. I use the `public/index.html` as the HTML template, which change to inject an `<script>` tag with the path to the js bundler. This magic happens when you run `npm start` and `npm run build`. You can check this explanation directly from one of the React core developers.

On line 7 we are telling React to render our Person component. That will instantiate the Person `class` and execute the `render` method, in the case of the class based component, or execute the Person `function` in the case of the function based component. In either case, the output is inserted into the DOM, generating what is shown below:

```
1 <div id="root">
2  <p>This is a <strong>Person</strong> Component</p>
3 </div>
```

In the `src/index.js`, the `<Person/>` component is wrapped by `<React.StrictMode>` component which helps us in development informing us about potential problems with our React code. Have a look at strict mode in React official docs.

Note that component names, in this case Person, start with capital letter. React sees components starting with capital letter as custom components and that requires the function or class definition to be in scope. And it sees components starting with lowercase letter as DOM tags, like div, p, strong, etc.

2.4.1. Styling

Here we describe how you can add styling to your React components. In the example below we are adding style to a component using a CSS file.

./StyleDemo.css

```
1  .big {
2    background-color: red;
3    height: 200px;
4    width: 200px;
5    font-size: 30px;
6    color: blue;
7  }
8
9  .small {
10   background-color: green;
11   height: 40px;
12   width: 40px;
13   font-size: 14px;
14   color: black;
15 }
```

Below, on line 1 we are first importing the CSS file we shown above which is located in the same directory of the component. Note that on lines 8 and 9 we are using the `className` to assign specific style to elements, instead of `class` used in HTML.

```
1  import "./StyleDemo.css";
2  import React, { Component } from "react";
3
4  export default class StyleAComponent extends Component {
5    render() {
6      return (
7        <>
8          <div className="small">Hola Mundo</div>
9          <div className="big">Hola Mundo</div>
10       </>
11     );
12   }
13 }
```

This component will paint on the browser two squares, one green and small and the other red and bigger. You might need to choose between different styles based on a `prop` value:

```
import React from "react";
import ReactDOM from "react-dom";
import StyleAComponent from "./StyleAComponent";

ReactDOM.render(
  <React.StrictMode>
    <StyleAComponent square="small" />
    <StyleAComponent square="big" />
  </React.StrictMode>,
  document.getElementById("root")
);
```

```
import "./StyleDemo.css";
import React, { Component } from "react";

export default class StyleAComponent extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <>
        <div className={this.props.square}>Hola Mundo</div>
      </>
    );
  }
}
```

While not recommended, you can also use inline styling. The style attribute of JSX elements accepts a JavaScript object with camelCased properties, as demostrated below:

```
import React, { Component } from "react";

export default class StyleInLine extends Component {
  constructor(props) {
    super(props);
  }

  render() {
```

```
      const colorBoxStyle = {
        width: "50px",
        height: "50px",
        border: "1px solid rgba(0, 0, 0, 0.05)",
        backgroundColor: this.props.color,
      };

      return (
        <>
          <div style={colorBoxStyle}>Hola Mundo</div>
        </>
      );
    }
  }
```

2.5. Props

In React you can pass arguments to components. These arguments are transformed into a single JavaScript object with a special name: `props` (which stands for properties). Let's add to our class based Person component the usage of props:

```
export default class Person extends Component {
  render() {
    return (
      <>
        My name is
        <strong>
         {this.props.name + " " + this.props.surname}
        </strong>
      </>
    );
  }
}
```

And now the equivalent but using the function definition of the component:

```
export default function Person(props) {
  return (
    <>
      My name is
```

```
5         <strong> {props.name + " " + props.surname}</strong>
6       </>
7     );
8   }
```

> You might note the use of an empty tag `<>...</>` in the components definition above. React component must always return an element. That force you to wrap everything into a root element and usually for these cases we use the `<div>` element as our root. But that introduce an extra node in the DOM. That is why React introduced what is called fragments.

Note that in our Person component, the `props` object has two properties: `name` and `surname`. In the class based component we accessed to them using `this.props` and in function based component just use the argument of the function.

How do you then pass `name` and `surname` to the component? You have to define them as JSX attributes, lets see that below:

```
1   import React from "react";
2   import ReactDOM from "react-dom";
3   import Person from "./Person";
4
5   ReactDOM.render(
6     <React.StrictMode>
7       <Person name="Enrique" surname="Molinari" />
8     </React.StrictMode>,
9     document.getElementById("root")
10  );
```

When React sees attributes in a user-defined component, like what we have on line 7 above, it passes them as the single object `props`.

> If you define a `constructor` in a class-based component, the constructor must receive `props` as argument and the first implementation line must be `super(props)`.

2.6. State

Suppose that we want to build a CountDownLatch component. Starting from a positive integer, on each second it is decremented until arrives to zero. Using what we have learned so far, we can create the component below:

```
1  import React, { Component } from "react";
2
3  export default class CountDownLatch extends Component {
4    render() {
5      return <h1>{this.props.startFrom}</h1>;
6    }
7  }
```

The component just paint the property `props.startFrom` wrapped in an `<h1>` tag on the browser. Then on the `src/index.js` we can do something like this:

```
1  import React from "react";
2  import ReactDOM from "react-dom";
3  import CountDownLatch from "./CountDownLatch";
4
5  function countDown(number) {
6    ReactDOM.render(
7      <React.StrictMode>
8        <CountDownLatch startFrom={number} />
9      </React.StrictMode>,
10     document.getElementById("root")
11   );
12   if (number === 0) {
13     clearInterval(intervalId);
14   }
15 }
16
17 let startFrom = 10;
18 let intervalId = setInterval(() => {
19   countDown(startFrom);
20   startFrom = startFrom - 1;
21 }, 1000);
```

The function on line 5, render the CountDownLatch component on the browser passing the number to be painted as props. Then we have starting on line 17 the `setInterval` JavasScript function which execute the arrow function (lines 19 and 20) every second. So, every second, the CountDownLatch component is repainted on the browser, each time with a the number passed as prop decremented by one. Until it arrives to zero where the `clearIntervale()` is executed stopping the count down.

However, in the way we have implemented this, the logic that does the decrements, update the browser every second and stop the count down is outside the component. We can implement our component in a much better way incorporating the logic *inside* the component. This will give us a much more reusable CountDownLatch component. In order to do this we must add **state** to our component. The **state** in React components is not like classic state you know from objects in the object oriented paradigm. React components *react* to state changes performing the render again. Lets see how we add state to our CountDownLatch component:

```
1  import React, { Component } from "react";
2
3  export default class CountDownLatch extends Component {
4    constructor(props) {
5      super(props);
6      this.state = {
7        startNumber: this.props.startFrom,
8      };
9    }
10
11   render() {
12     return <h1>{this.state.startNumber}</h1>;
13   }
14 }
```

`state` is managed and controlled by the component. On line 12, we have changed the use of props in `this.props.startNumber`, to the use of `state`. And on the constructor starting on line 4, we are initializing the `state` with the value from the `props` comming from a JSX attribute. Note that `state` is a JavaScript object, but with special meaning for React. Then on the `src/index.js` we add the JSX elements to paint our component on the browser:

```
1  import React from "react";
2  import ReactDOM from "react-dom";
3  import CountDownLatch from "./CountDownLatch";
4
5  ReactDOM.render(
6    <React.StrictMode>
7      <CountDownLatch startFrom={10} />
8    </React.StrictMode>,
```

```
9     document.getElementById("root")
10  );
```

As you can see on line 7 we are passing the value `10` as `props` which is used to initialize the `state` in the component's constructor. So far, if we run this, we will have just this markup `<h1>10</h1>` painted on the browser. Now we have to add the logic that decrements our number. To do this, we will take advantages from what React calls **lifecycle** methods. These methods are present only in class-based components and they are hooks that you can use plug your logic. Calling them hooks can be confused because React Hooks is a big new topic that we will cover in the next section, but in object oriented frameworks literature hooks are called to those extension points that you have to customise the framework. And this is exactly that, they are a set of methods that React calls at *certain moments* and you can use to implement component's specific logic.

On of these lifecycle method that we will use is `componentDidMount()`. This method is called by React after the `constructor` is executed and after `render` is executed. So, it is called just after our component is rendered into the DOM for the first time. It is called the *mounting* moment. This seems to be the perfect moment to set up the `setInterval` that performs the decrement. Let's see how this looks:

```
1   import React, { Component } from "react";
2
3   export default class CountDownLatch extends Component {
4     constructor(props) {
5       super(props);
6       this.state = {
7         startNumber: this.props.startFrom,
8       };
9     }
10
11    componentDidMount() {
12      this.intervalId = setInterval(() => {
13        this.setState((state) => ({
14          startNumber: state.startNumber - 1,
15        }));
16      }, 1000);
17    }
18
```

```
19      render() {
20        return <h1>{this.state.startNumber}</h1>;
21      }
22    }
```

On line 11 we have added the method `componentDidMount()` where we use to set up the decrement of the component. Note that each second the arrow function, passed as the first argument of the `setInterval` function, that decrements the `state.startNumber` by one is executed. To update the state we have to use the special `this.setState()` method, which triggers the *re-render* of the component. So, on each second the `this.state.startNumber` is decremented by one and the component is painted again on the screen.

What is still missing in our component is to stop the countdown when it arrives to zero. To implement this, we have use another lifecycle method called `componentDidUpdate(prevProps, prevState)`. This method is executed when there is an update on the `state` of the component, after render again the component on the DOM. By parameter receives the previous values from the props and the previous value from the state. Lets add then the logic to stop the countdown on this method to finishing our component.

```
1   import React, { Component } from "react";
2
3   export default class CountDownLatch extends Component {
4     constructor(props) {
5       super(props);
6       this.state = {
7         startNumber: this.props.startFrom,
8       };
9     }
10
11    componentDidMount() {
12      this.intervalId = setInterval(() => {
13        this.setState((state) => ({
14          startNumber: state.startNumber - 1,
15        }));
16      }, 1000);
17    }
18
19    componentDidUpdate() {
20      if (this.state.startNumber === 0) {
```

```
21          clearInterval(this.intervalId);
22        }
23      }
24
25      render() {
26        return <h1>{this.state.startNumber}</h1>;
27      }
28    }
```

On line 19 above, we have added the lifecycle method `componentDidUpdate()`. As we can see on figure 2.1, after the mounting phase is finished, we have the updating phase, triggered by calling the `setState()` method, which first invoke the `render` and after that the `componentDidUpdate()`. There, if we arrive to zero we stop the countdown by calling the `clearInterval()` function. Remember that the arrow function we have set up in the `setInterval()` on the `componentDidMount()` method will be executed every second, triggering the updating phase each time due to the call to the `setState()` method.

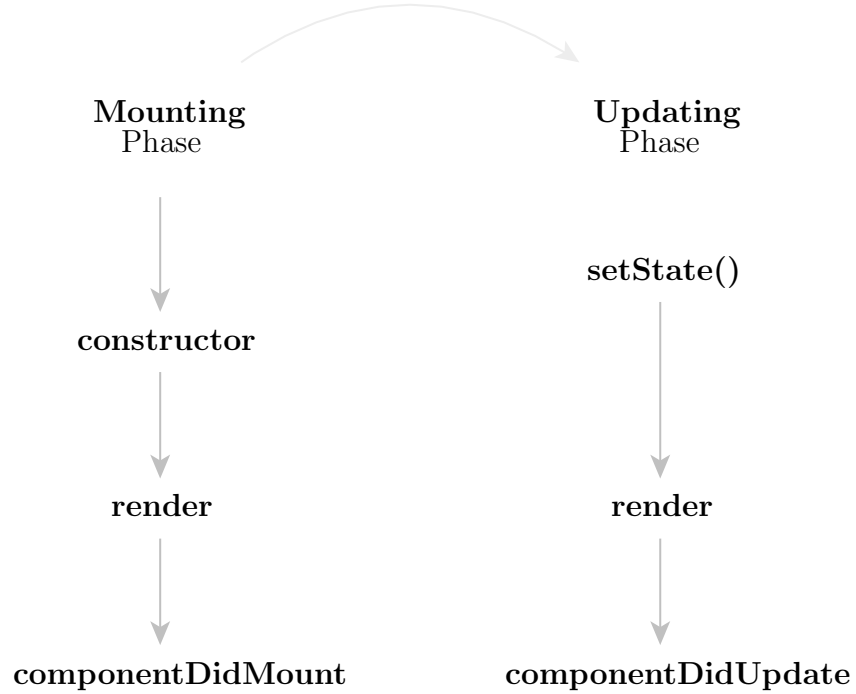

Figura 2.1: Order of execution of lifecycle methods used in the CountDownLatch component

We have seen how to use state and the lifecycle methods to create a self-encapsulated and reusable component. And we have reviewed how changes

in component's state triggers the updating phase which among other things, re-render of the components on the browser. There are some more lifecycle methods in React. Dan Abramov has shared a picture to summarize all the lifecycle methods available. Note from the picture that there is a third phase called unmounting. That phase has a single lifecycle method called `componentWillUnmount()` which is executed before the component is unmounted from the DOM. This method is used for clean up or close resources: subscriptions, sockets, timers, etc.

Few important notes to manage state in the correct way.

- The constructor is the only place where you can change the state directly. In all other places use `setState()`.

```
1  //only do this inside constructor
2  this.state.startNumber = 4;
```

- State Updates may be asyc: React may batch multiple `setState()` calls into a single update for performance. Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state. In our CountDownLatch component we are updating the state on line 14 based on their previous value. In these cases you should not trust on the state current value. Use the one passed as argument on the setState() method, like below:

```
1  //Don't do this
2  this.setState(() => ({
3    startNumber: this.state.startNumber - 1,
4  }));
5
6  //Do this
7  this.setState((state, props) => ({
8    startNumber: state.startNumber - 1,
9  }));
```

- In the `componentDidUpdate(prevProps, prevState)` lifecycle method you might need to call the `setState()` method. But make sure you wrap that call in a condition to avoid an infinite loop. Usually, the condition is based from the previous values of the props and/or state with the current values of them.

2.7. Dealing with Events

Defining events in React is not that different from defining them on HTML. The big different is that in React you define the events on JSX, which is then translated to JavaScript, so there are two important things you should know:

- Event names in JSX are defined in camelCase.
- You have to pass a function to handling the event.

Look at the example below:

```
<button onClick={clickMe}>Click me</button>
```

Note that the event name `onClick` is written in camelCase. As we have mentioned, in JSX between curly braces you can define any valid JavaScript expression. In this case we are passing the function `clickMe` to be used to handling the event. As we have seen in section 1.2, functions are first-class citizens in JavaScript allowing us to do that. Let's create a function based component to use some events:

```
export default function EventExample() {
  function onOver() {
    console.log("onOver...");
  }

  function clickmeLink(e) {
    //this is necesary to prevent the default
    //link behavior
    e.preventDefault();
    console.log("link clicked...");
  }

  return (
    <div>
      {/*events are camelCase*/}
      {/*and they receive a function not a string*/}
      <a href="#" onMouseOver={onOver}
      ↪  onClick={clickmeLink}>
        click this link
      </a>
```

```
20        </div>
21      );
22    }
```

On lines 2 and 6 we are defining the functions to handle the events `onClick` and `onMouseOver`, which just print on the console some text. Note that on the function `clickmeLink(e)` on line 6 we are receiving as argument an instance of the event, but is not the browser's native event, it is an instance of SyntheticEvent, a React wrapper to make events works identically across browsers. And in this case we use it to prevent the default behaviour of clicking an anchor element.

Now, lets see how this example can be translated into a class based component:

```
1   import React, { Component } from "react";
2
3   export default class EventExample extends Component {
4     constructor(props) {
5       super(props);
6       this.onOver = this.onOver.bind(this);
7       this.clickmeLink = this.clickmeLink.bind(this);
8     }
9
10    onOver() {
11      console.log("onOver...");
12    }
13
14    clickmeLink(e) {
15      //this is necesary to prevent the default
16      //link behavior
17      e.preventDefault();
18      console.log("link clicked...");
19    }
20
21    render() {
22      return (
23        <div>
24          {/*events are camelCase*/}
25          {/*and they receive a function not a string*/}
26          <a href="#" onMouseOver={this.onOver}
            ↪  onClick={this.clickmeLink}>
```

```
27              click this link
28            </a>
29          </div>
30        );
31      }
32    }
```

Since event handlers are class methods, on line 26 we can see that they are passed using `this`. However, as we have explained in section 1.6, if we assign or pass as an argument a method, we lose the binding of `this`, in this case, to the component instance. That is why we have to explicitly set this binding as we do on lines 6 and 7. Other than that, it is pretty similar to what we did on the function based component.

Here is another example, in this case we change the state on the event handler:

```
1   import React, { Component } from "react";
2
3   export default class ColorSelect extends Component {
4     constructor(props) {
5       super(props);
6       this.state = {
7         value: "white",
8       };
9       this.colorChanged = this.colorChanged.bind(this);
10    }
11
12    colorChanged(e) {
13      this.setState({
14        value: e.target.value,
15      });
16    }
17
18    render() {
19      const colorBoxStyle = {
20        width: "30px",
21        height: "30px",
22        border: "1px solid rgba(0, 0, 0, 0.05)",
23        backgroundColor: this.state.value,
24      };
```

```
25
26        return (
27          <>
28            <label for="colors">Choose your favourite color:
              ↪  </label>
29            <select name="colors" onChange={this.colorChanged}>
30              <option>Options...</option>
31              <option value="blue">Blue</option>
32              <option value="red">Red</option>
33              <option value="green">Green</option>
34              <option value="yellow">Yellow</option>
35            </select>
36            <div style={colorBoxStyle}></div>
37          </>
38        );
39      }
40    }
```

In the example above we have a drop down list with colors. When you choose one color, the `onChange` event (line 29) is triggered, calling the method `colorChanged()` (line 12). The `state` gets updated with the color selected. Changing the state makes the render to be executed, painting the square box (line 36) now colored with the chosen color.

It is important to clarify that React is very efficient in updating the DOM. Only the exact portion of the DOM that has changed is the one that is updated. The rest remain untouched. You can check what I'm saying by running this component inspecting the DOM with the Browser DevTool. This is achieved by React using what is called as Virtual DOM and the reconciliation.

We will look at more events later on the book. Here is the full list of supported events.

Bibliografía

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[2] https://www.ecma-international.org/

[3] https://nodejs.org/

[4] https://requirejs.org/

[5] http://latentflip.com/loupe/