

# Xseed

---

Curso de React

# Clase 03 -

ES6 (EcmaScript 2015) - Conditional Rendering

---

by Diego Cáceres

# Repaso

---

# Tipos de componentes

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
class HelloWorld extends React.Component {
  render () {
    return (
      <div>
        Hello {this.props.name}
      </div>
    )
  }
}
ReactDOM.render(<HelloWorld name='Diego' />,
  document.getElementById('app'))
```

**Class Component**

```
import React from 'react';
import ReactDOM from 'react-dom';
function HelloWorld (props) {
  return (
    <div>Hello {props.name}</div>
  );
}
ReactDOM.render(<HelloWorld name='Diego' />,
  document.getElementById('app'))
```

**Functional Component** (Stateless Component)

# Props son Read-Only

No importa si se trabaja con **Class Component** o **Functional Component**, en cualquiera de los casos los valores que se reciben en *props* no se pueden modificar, son **sólo de lectura**.

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

Las **funciones puras** son un concepto relacionado con la programación funcional. Una función es pura cuando no modifica sus *inputs*, y devuelve siempre el mismo resultado para el mismo *input*.

## Manejo del State (1)

El *state* es similar a las *props*, pero es **privado** para el componente y **controlado** completamente por el componente.

No se puede acceder al *state* de un componente desde otro; y si esto se logra mediante algún truco de JavaScript, igualmente va en contra de los conceptos de React.

Como se mencionó anteriormente, el estado local es una característica que sólo tienen los componentes de tipo **Class**. Es una práctica común crear un *Functional Component* y que más adelante haya que convertirlo en un *Class Component* por necesidad de manejar estado.

## Manejo del State (2)

La forma de declarar el *state* inicial en un componente es mediante el **constructor**, que no es propio de React sino de la `class` de JavaScript introducida en ES6. Es importante notar que este es en el único lugar donde se va a asignar el *state* con un simple `=`

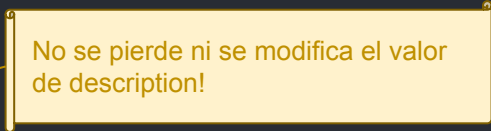
```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      description: 'Soy un ejemplo'  
    };  
  }  
  render() { <div> Hello World! </div> }  
}
```

El state es un Objeto que contendrá todas las propiedades que se quieran manejar localmente dentro del componente.

## Manejo del State (5)

La forma correcta es utilizar la función **setState** que recibe un objeto, con el cual actualiza el estado del componente. Sólo actualiza las propiedades que contiene el nuevo objeto, por lo que no es necesario que contenga las propiedades que no se modificaron.

```
constructor(props) {  
  super(props);  
  this.state = {  
    description: "Soy un ejemplo",  
    comment: "Comentario inicial"  
  };  
}  
  
// En algún lugar del componente:  
this.setState({  
  comment: "Comentario actualizado"  
})
```



No se pierde ni se modifica el valor de description!



# State y uso del bind

En React es común que se utilice el `bind` para que las funciones definidas dentro de un Class Component, siempre estén 'bindeadas' al mismo contexto, es decir, al contexto del componente.

```
class InputExample extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { text: "" };  
    this.change = this.change.bind(this);  
  }  
  change(ev) {  
    this.setState({ text: ev.target.value });  
  }  
  render() {  
    return <input type="text" value={this.state.text} onChange={this.change} />;  
  }  
}
```

Si no se "bindea" `change`, al utilizar `this.setState` no estaría definido

ECMAScript 2015 o ES2015  
También llamado ES6

---

# Nuevas características de JavaScript

ECMAScript 2015 (6ta edición) es una versión de la especificación del lenguaje **ECMAScript**, conocida simplemente como **ES2015 (ES6)**, y fue aprobada en junio del 2015.

Sigue siendo algo nuevo a pesar de ser del 2015, principalmente porque ES5 había salido en 2009. Con la creciente popularidad del lenguaje JavaScript ahora se están agregando más frecuentemente nuevas características al mismo.

Tanto es así que ya salió ES2016 (ES7) y ES2017 (ES8). Sin embargo, aún son muy poco soportadas por los navegadores, por ende durante el curso no se verán estas versiones.

# ES6

Se revisarán las siguientes características:

- Arrow functions.
- `var`, `let` y `const`.
- Default values.
- Object Destructuring.
- Enhanced Object Properties.
- Spread operator "...".
- String templates.
- Imports / Exports.

*\* Estas no son las únicas características nuevas en ES6.*

# Arrow functions

---

# Arrow functions (1)

Es una nueva forma de **declarar las funciones**, simplemente indicando los parámetros entre paréntesis. Ya no se necesita la palabra `function`, pero se agrega una flecha `=>`

```
// ES5:
var createGreeting = function(name, message) {
    var result = message + ' ' + name;
    return result;
}

// ES6:
var arrowGreeting = (name, message) => {
    var result = message + ' ' + name;
    return result;
}
```

## Arrow functions (2)

Cuando se tiene sólo una línea de código dentro de la función, se puede escribir todo en una línea sin necesidad de escribir `return` ni de encapsular el cuerpo de la función entre llaves `{ ... }`

```
// Hasta ahora no cambia tanto, pero podemos escribir menos código:
var arrowGreeting = (name, message) => message + ' ' + name;
// Esto automáticamente retorna el resultado de la sentencia: message + ' ' + name;

// Si sólo tenemos un parámetro, no necesitamos los paréntesis:
var arrowGreeting = name => 'Bienvenido: ' + name;

// Otro ejemplo:
var squared = x => x * x;
```

## Arrow functions (3)

Un típico problema en JavaScript es el valor de la keyword `this`. Cada función en JavaScript define su propio valor de `this`. En este caso, se resuelve guardando su valor en una variable `self` para después accederlo desde el contexto de la función `crecer`.

```
function Persona() {  
    var self = this; // Algunas personas prefieren `that` en lugar de `self`.  
    self.edad = 0;  
    setInterval(function crecer() {  
        // La función callback apunta a la variable `self` la cual contiene el objeto esperado.  
        self.edad++;  
    }, 1000);  
}
```



## Arrow functions (4)

Si la función que se le pasa a `setInterval` se define con una *arrow function* no se tiene ese problema, ya que el valor de `this` dentro del *arrow function* se mantiene igual al del contexto actual.

```
function Persona() {  
    this.edad = 0;  
    setInterval(() => {  
        this.edad++; // this apunta al objeto Persona  
    }, 1000);  
}  
  
var p = new Persona();
```

var, let *y* const

---

## Definición de variables: `var`, `let` y `const` (1)

En JavaScript siempre se utiliza **`var`** para declarar variables. En el siguiente ejemplo se puede pensar que en consola saldrá “Sunday” porque el resto está en un bloque, pero en realidad es la misma variable que se re-assigna.

```
var message = 'Sunday';
{
    var message = 'Monday';
}
console.log(message); // Imprime Monday
```

Lo que ya existe en JavaScript es “*function scoping*”, donde las variables existen dentro de la función únicamente. Con el resto de los bloques (**`for`**, **`if`**, etc) no hay *scope* y sería la misma variable.

```
var message = 'Sunday';
function dayOfWeek() {
    var message = 'Monday';
}
console.log(message); // Imprime Sunday
```

## Definición de variables: `var`, `let` y `const` (2)

En ES6 se introduce la palabra clave `let` que sí tiene scope dentro de los bloques.

```
let message = 'Sunday';  
  
{  
  
    let message = 'Monday';  
  
}  
  
console.log(message); // Imprime Sunday
```

## Definición de variables: `var`, `let` y `const` (3)

Otro ejemplo. Se tiene un array de funciones y un **for** loop donde para cada valor de `i` se inserta una nueva función al array, cuya finalidad es loguear el valor de `i` en consola.

Podría esperar que al recorrer el array y ejecutar las funciones se impriman los números del 0 al 9. Sin embargo, lo que sucede es que el número "10" se imprime diez veces, porque es la misma variable `i` que se reasigna.

```
var fs[];
for (var i = 0; i < 10; i++){
    fs.push(function() {
        console.log(i);
    });
}
fs.forEach(function(f) {
    f(); // Va a imprimir siempre 10.
});
```

## Definición de variables: `var`, `let` y `const` (4)

Si en vez de `var` se utiliza `let`, entonces se obtiene el resultado esperado, ya que en cada loop se crea un nuevo `i` para el bloque.

```
var fs[];
for (let i = 0; i < 10; i++) {
    fs.push(function() {
        console.log(i);
    });
}
fs.forEach(function(f) {
    f(); // Va a imprimir desde 0 a 9.
});
```

## Definición de variables: `var`, `let` y `const` (5)

Hasta ES5 cuando se desea declarar algo como constante, como buena práctica se declara en mayúsculas, pero eso no impide que se vuelva a modificar su valor en el código.

```
var PI = 3.14;  
PI = 10;  
console.log(PI); // Imprime 10
```

En ES6 se introduce la palabra clave **const** para declarar variables que no se quiere que se les pueda modificar el valor. Son **Read-Only**.

```
const PI = 3.14;  
PI = 10; // -> Esto tira un error de Read Only en consola  
console.log(PI);
```

## Definición de variables: `var`, `let` y `const` (6)

Es importante notar que la variable no es constante, sino que es una referencia constante.

Si se declara un objeto con **`const`** aún se le puede asignar propiedades sin que tire error de Read-Only, pero no se puede re-asignar el objeto a un string por ejemplo.

```
const person = {};  
  
person.age = 26; // Funciona.  
  
person.name = 'Mike'; // Funciona.  
  
person = 'Mike'; // NO funciona -> Tira un error de Read Only en consola.
```

Al igual que `let`, `const` sólo existe dentro del bloque donde es declarado.

```
if (true) { const foo = 'bar'; }  
  
console.log('Foo value:', foo); // -> Not defined error
```



# Default values

---

# Default values (1)

En ES6 se introduce la posibilidad de declarar valores por defecto en los parámetros de la declaración de una función.

```
function greeting(message, name) {  
    console.log(message + ', ' + name);  
}  
greeting(); // Loguea: undefined, undefined  
// ES6:  
function greeting(message, name = 'Mike') {  
    console.log(message + ', ' + name);  
}  
greeting(); // Loguea: undefined, Mike  
greeting('Hello'); // Loguea: Hello, Mike
```

## Default values (2)

También se puede utilizar para declarar funciones por defecto.

```
function greet(complete = function() {  
    console.log("complete");  
}) {  
    complete();  
}  
greet();  
  
// Y usando arrow functions:  
function greet(complete = () => console.log("complete")) {  
    complete();  
}  
greet();
```

# Object Destructuring

---

# Object Destructuring (1)

En JavaScript para acceder a una propiedad dentro de un objeto, se utiliza el punto ".".  
En ES6 también se puede hacer a través de "destructuring".

```
var person = {  
  name: 'Mike'  
}  
  
console.log(person.name);  
  
// Con Destructuring:  
var { name } = person;  
console.log(name);
```

## Object Destructuring (2)

Esto también es útil cuando se tiene un objeto con varias propiedades y solo interesan algunas.

```
var person = {  
  name: 'Mike',  
  lastname: 'Ford',  
  city: 'London',  
  age: 30  
};  
  
var {name, city} = person;  
console.log(name);  
console.log(city);
```

## Object Destructuring (3)

Un típico caso es cuando se tiene una función que retorna un objeto, pero solo se requieren algunas de sus propiedades.

```
function getFooBarOwner() {  
  return {  
    name: 'Mike',  
    lastname: 'Ford',  
    city: 'London',  
    age: 30  
  };  
}  
  
var {name, city} = getFooBarOwner();  
console.log(name);  
console.log(city);
```

# Object Destructuring (4)

También es posible cambiarle el nombre a las variables que se crean con respecto al nombre que tienen las propiedades del objeto.

```
function getFooBarOwner() {  
  return {  
    name: 'Mike',  
    lastname: 'Ford',  
    city: 'London',  
    age: 30  
  };  
}  
  
var {name:firstname, city:location} = getFooBarOwner();  
console.log(firstname);  
console.log(location);
```



## Object Destructuring (5)

Un uso común en React es utilizar esto en los parámetro de una función (o de un Componente para quedarse solo con lo necesario de Props)

```
function myFunction(person) { // ES5
  return (person.name + ' ' + person.lastname);
}

function myFunction({name, lastname}) { // ES6
  return (name + ' ' + lastname);
}

// Un Componente
function NickName({ nickName }) {
  // En vez de recibir props y usar props.nickName
  return <h3>NickName: {nickName}</h3>;
}
```

# Enhanced Object Properties

---

# Enhanced Object Properties - Shorthand Properties

Es una manera rápida de agregar o modificar propiedades de un objeto a partir de variables que ya se tienen, sin necesidad de decir el nombre de la propiedad.

```
let firstname = 'Mike';
let lastname = 'Ford';
let person = {firstname, lastname};
// En ES5 hubiera sido:
var person = {firstname: firstname, lastname: lastname};

// También podemos hacerlo con objetos
let sportsTeam = 'Barcelona';
let fan = {person, sportsTeam};
console.log(fan);
// { person: {firstname: Mike, lastname:Ford}, sportsTeam: Barcelona}
```

# Enhanced Object Properties – Method Properties

En ES6 se tiene soporte para declarar métodos como propiedades de un objeto.

```
// ES5:
var obj = {
  foo: function (a, b) {
  }
}

// ES6:
let obj = {
  foo (a, b) {
  }
}
```

# Enhanced Object Properties – Computed Properties

En ES6 también se cuenta con la posibilidad de utilizar computed properties, lo cual puede ser muy útil. Básicamente se puede acceder a una propiedad de un objeto mediante un valor calculado.

```
let obj = {  
  foo: "bar",  
  [ "baz" + quux() ]: 42  
}  
  
let person = {  
  name: "",  
  lastName: ""  
}  
  
let field = "name";  
person[field] = "Diego"; // Modifica la propiedad name.
```

# Spread Operator ...

---

# Spread Operator ...

Este nuevo operador permite tomar un array y separarlo en cada uno de sus ítems (spread = propagar).

```
console.log([1,2,3]); // [1, 2, 3]
console.log(...[4,5,6]); // 4 5 6
// Esto es util para unir de forma facil dos arrays
let first = [1,2,3];
let second = [4,5,6];
first.push(second);
console.log(first); //[1, 2, 3, [4, 5, 6]]
// Si en vez de eso uso spread
let first = [1,2,3];
let second = [4,5,6];
first.push(...second);
console.log(first); //[1, 2, 3, 4, 5, 6]
```

# Spread Operator ...

También se puede utilizar para separar un array en los parámetros que recibe una función.

```
let first = [1,2,3];  
let second = [4,5,6];  
  
function addThreeNumbers(a, b, c) {  
    console.log( a + b + c );  
}  
  
addThreeNumbers(...first); // 6  
addThreeNumbers(...second); // 15
```



# String Templates ``

---

# String Templates

Hasta ES5 es muy común el concatenar variables con strings de la siguiente forma.

```
let name = 'Diego';  
let greeting = 'Hello ' + name + ', welcome aboard';  
console.log(greeting); // Hello Diego, welcome aboard.
```

En ES6 se puede utilizar el tilde al revés `` (Alt + 96) para envolver un *string template*, dentro del cual se puede acceder a evaluar JavaScript dentro de `${ .. }`

```
let name = 'Diego';  
let greeting = `Hello ${name}, welcome aboard`;  
console.log(greeting); // Hello Diego, welcome aboard.
```

# String Templates

También se pueden utilizar expresiones dentro de las llaves y serán evaluadas. Otro detalle, es que respeta las líneas en blanco y los tabs que se agreguen al texto.

```
let x = 5;  
var eq = `${x} * ${x} = ${x*x}`;  
console.log(eq); // 5 * 5 = 25
```

```
let name = 'Diego';  
let greeting = `  
Hello  
    ${name},  
  
        welcome aboard`;  
console.log(greeting);
```

En la Consola se imprimirá:

```
Hello  
    Diego,  
  
        welcome aboard
```

# Imports - Exports

---

# Imports - Exports (1)

En ES6 se puede de forma muy simple exportar e importar valores y funciones, sin necesidad de acudir a global namespaces.

```
// En src/utils/math.js:
function square(a) { return a*a; }
export { square };

// En src/index.js:
import { square } from 'utils/math.js';
console.log(square(5)); // 25

// Otra forma de exportar:
// En src/utils/math.js
export function square(a) { return a*a; }
```

## Imports - Exports (2)

Se puede hacer esto para varias funciones y a la hora de importarlas también se les puede dar un alias.

En el archivo `src/utils/math.js`:

```
export function square(a) { return a*a; }  
export function multiply(a,b) { return a*b; }
```

En el archivo `src/index.js`:

```
import {  
    square as squareFunction,  
    multiply  
} from 'utils/math.js';  
  
console.log(squareFunction(5)); //25  
console.log(multiply(5, 3)); //15
```

## Imports - Exports (3)

También es posible importar todas las funciones de un archivo bajo un namespace, lo cual es útil si tenemos varias funciones que queramos utilizar, por ejemplo, si importo todas las funciones para invocar una API.

En el archivo `src/utils/math.js`:

```
export function square(a) { return a*a; }  
export function multiply(a,b) { return a*b; }
```

En el archivo `src/index.js`:

```
import * as math from 'utils/math.js';  
  
console.log(math.square(5)); //25  
console.log(math.multiply(5, 3)); //15
```

# Conditional Rendering

---



# Conditional Rendering

En React se pueden crear distintos componentes para encapsular en cada uno el comportamiento necesario. Luego, se puede utilizar **Conditional Rendering** para dibujar solo alguno de ellos, dependiendo del estado de la aplicación.

```
const UserGreeting = (props) => { return <h1>Welcome back!</h1>; }
const GuestGreeting = (props) => { return <h1>Please sign up.</h1>; }

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

# Conditional Rendering

También se pueden utilizar variables para almacenar los componentes y después mostrar (o no) parte de un componente sin que afecte el resto del *output*.

```
render() {  
  const isLoggedIn = this.state.isLoggedIn;  
  let body = null;  
  if (isLoggedIn) {  
    body = <App />;  
  } else {  
    body = <LoginComponent />;  
  }  
  return (  
    <div>  
      {body}  
    </div>  
  );  
}
```

# Conditional Rendering - Operador Lógico

Como todo lo que se escriba entre `{ }` será evaluado como una expresión JavaScript, también se puede utilizar el operador lógico de JavaScript `&&` lo que permite lograr condicionales más cortos.

```
function Mailbox(props) {  
  const unreadMessages = props.unreadMessages;  
  return (  
    <div>  
      <h1>Hello!</h1>  
      {unreadMessages.length > 0 &&  
        <h2>  
          You have {unreadMessages.length} unread messages.  
        </h2>  
      }  
    </div>  
  );  
}
```

# Conditional Rendering - Inline If - Else

También se puede utilizar el operador condicional

`condition ? true : false`

```
render() {  
  const isLoggedIn = this.state.isLoggedIn;  
  return (  
    <div>  
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.  
    </div>  
  );  
}
```

# Conditional Rendering - Inline If - Else

Se pueden agregar expresiones un poco más complejas, aunque hay que tener cuidado de que el código siga siendo legible.

```
render() {  
  const isLoggedIn = this.state.isLoggedIn;  
  return (  
    <div>  
      {isLoggedIn ? (  
        <LogoutButton onClick={this.handleLogoutClick} />  
      ) : (  
        <LoginButton onClick={this.handleLoginClick} />  
      )}  
    </div>  
  );  
}
```

# Ejercicios

---

# Ejercicio 1 – TicTacToe

Partiendo de la solución del ejercicio de TicTacToe, modificarlo para que utilice sintaxis de ES6, sin que deje de funcionar.

1. Considerar los `var`, no debería quedar ninguno que no se pueda cambiar por un `let` o un `const`.
2. Considerar las `function` definidas `inline`, seguramente sea mejor cambiarlas por *arrow functions* (por ejemplo en el caso que se guarde `this` en `self`, no es más necesario).
3. Cambiar el uso de `slice` por el *Spread Operator* de *arrays*.

## Ejercicio 2

Editar el archivo `src/App.js`. Lograr que `Clock` deje de ser un Functional Component, para que tenga estado y sea quien maneje la hora a mostrar, para que funcione como un reloj cuando se presiona el botón, y de esa forma no muestre siempre la misma hora.

1. Primero se debe definir `Clock` como una subclase de `Component`, con su método `render`, y cambiar `props` por `this.props`, y verificar que siga funcionando.
2. Definir un constructor con el estado inicial, determinando qué estado necesita manejar `Clock`.
3. Dejar de mostrar la hora recibida por `props` y cambiarla por la del estado local. También se puede cambiar en `App` cuando se utiliza `Clock` por `<Clock />`.
4. Implementar un método para actualizar el valor de la hora, con el botón actualizar (más adelante se implementará una solución que se actualice automáticamente).

**No olvidarse del bind.**



## Ejercicio 2 – Parte 2

Ahora se procederá a automatizar para que se actualice sola la hora y cambiar el botón simplemente para pausar el reloj y reanudarlo.

1. Renombrar el método `updateTime` a `tick`.
2. Agregar la siguiente línea al final del constructor para crear un intervalo que ejecuta la función `tick` cada 1 segundo (Ahora el reloj funciona solo):

```
this.interval = setInterval(() => this.tick(), 1000);
```

3. Crear otro método en el Componente llamado `onPress`, "*bindearlo*" y llamarlo desde el click del botón. El botón debería cambiar su texto a "Pausar/Reanudar" (si no se "bindea", no se tendrá acceso a `this.interval`).
4. Implementar `onPress` para que si `this.interval` no es `null`, limpie el intervalo y lo iguale a `null`, o de lo contrario vuelva a crear el intervalo. Nota, para limpiar un intervalo en JavaScript se utiliza la función `clearInterval(...)`.

## Ejercicio 3

- Editar el archivo `src/App.js`
- Completar el componente `Greeting` para que muestre `UserGreeting` o `GuestGreeting` según si el usuario está logueado o no.
- Completar el *render* de `LoginControl` para que muestre `LoginButton` o `LogoutButton` según si el usuario está logueado o no.
- Implementar `handleLoginClick` y `handleLogoutClick` y pasarlos como props a los botones según corresponda.

## Ejercicio 4

- Editar el archivo `src/App.js`
- Completar el componente `Card` para que muestre la información recibida por props. Deme manejar un state interno, para determinar si muestra los tabs o no, y para determinar qué tab es el seleccionado para mostrar. Además debe mostrar un mensaje por defecto si no recibe description.

