JavaScript que hay que saber para usar React

Enrique Barra Arias
Álvaro Alonso González

var vs. let vs. const

```
function addPoints(){
        var points = 50
        var winner = false

        if (points > 40) {
          var winner = true
                console.log(winner) // true
        }

        console.log(winner)
         > true
        if (points > 40) {
           winner = true
                console.log(winner) // true
        }
        console.log(winner)
        > true
}
```

Una variable (**var**) representa un nuevo identificador en nuestro código, al que podemos hacer referencia usando su nombre.

No se declara el tipo de variable, sino que se infiere por el valor.

Sólo se puede utilizar dentro de su **scope**, que es la función dentro de la cual se define.

var vs. let vs. const

```
function addPoints(){
        var points = 50
        let winner = false

        if(points > 40) {
          let winner = true
                console.log(winner)
                > true
        }
        console.log(winner)
        > false

        if(points > 40) {
          winner = true
        }
        console.log(winner)
        > true
}
```

Una variable (**let**) sólo se puede utilizar dentro del bloque de código en el que se declara (un condicional, un bucle, una función, etc.)

No se puede sobreescribir dentro del mismo bloque.

var vs. let vs. const

```
let name = 'Alice'
const handle = 'alice'

name = 'Alice Cooper' // ☑
handle = '@alicecooper' // ✗ TypeError: Assignment to constant variable.
```

Una constante (**const**) se comporta como una **let**, solo que una vez que se le ha asignado un valor, no se puede sobreescribir.

Shorthand property names

```
const a = 'hello'
const b = 42
const c = [true, false]

// Forma tradicional
let obj = {a: a, b: b, c: c}

// Forma abreviada
let obj = {a, b, c}

console.log(obj)
```

Si las variables/constantes tienen el **mismo nombre** que la clave a la que vamos a asignar en un objeto, podemos utilizar esta sintaxis abreviada

```
> { "a": "hello", "b": 42, "c": [true, false] }
```

Objetos y destructuring

```
const data = {
   id: 2,
   name: "Pepe",
   isAdmin: false
};

// Forma tradicional
let id = data.id // data["id"]
let name = data.name
let isAdmin = data.isAdmin
console.log(id)
>  2

// Forma desestructurada
let {id, name, isAdmin} = data
console.log(id)
>  2
```

Tenemos un objeto a cuyas propiedades queremos acceder.

Podemos crear una variable nueva con el nombre que queramos o, si usamos la **forma desestructurada**, crear en una sóla línea una variable para cada propiedad utilizando su nombre original.

Spread operator

```
const data = { id: 2, name: "Pepe", isAdmin: false };

let newData = { ...data, age: 5 };
console.log(newData);

> {id: 2, name: "Pepe", isAdmin: false, age: 5}
```

Nueva propiedad

```
let updatedData = { ...data, name: “Lola” };
console.log(updatedData);

> {id: 2, name: "Lola", isAdmin: false}
```

Sobreescribir propiedad existente

Arrays y destructuring

```
const foo = ['one', 'two', 'three'];
const [red, yellow, green] = foo;
console.log(red); // "one"

let a, b, rest;
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(rest); // Array [30,40,50]

function f() {
  return [1, 2];
}
let a, b;
[a, b] = f();
console.log(a); // 1
console.log(b); // 2
```

Tenemos un array a cuyos valores queremos acceder.

Si usamos la **forma desestructurada**, crear en una sóla línea una variable para cada valor utilizando su nombre original.

Las funciones también son objetos

```
//Declaración de la función
let myFunction = function() {
   console.log("Hello!");
}

//Imprimimos el resultado de ejecutar la función myFunction();
 > "Hello!"

//Imprimimos la función, que es un objeto (esto sirve de poco)
myFunction;
 > ƒ () {
   console.log("Hello!");
  }
```

IMPORTANTE: las funciones se ejecutan con su nombre y poniendo () o pasándoles sus parámetros en esos paréntesis, por ejemplo:
suma(3,4)
ó
llamaServer(url, obj)

Arrow functions

```
// Forma tradicional
function divide (a, b)  {
   return a / b;
}
const divide = function (a, b) {
   return a / b;
}

// Arrow function
const divide = (a, b) => a / b
const divide = (a, b) => {
            // ...
      return a / b;
}
```

Las arrow functions nos permiten declarar funciones de forma más abreviada.

IMPORTANTE: Las arrow functions heredan el objeto **this** del contexto donde están. En cambio las funciones "tradicionales" tienen su propio **this**.

Arrow functions - sintaxis

```
// sintaxis completa
const myfun = (param1, paramN) => {
  const a = 1;
  return a + param1 + paramN;
}
// solo un parámetro y solo 1 expresion
const myfun = param => expression
const myfun = (param) => expression

// varios parámetros y una expresión
const myfun = (param1, paramN) => expression
// parámetros por defecto
const myfun = (param1=5, paramN=7) => expression
```

- Si tiene más de un parámetro o parámetros por defecto es obligatorio el uso de paréntesis.
- Si tiene más de una expresión es obligatorio el uso de corchetes y la palabra “return” indicando lo que devuelve.

Operador ternario

```
let msg;
if (data.isAdmin) {
  msg = ':)';
} else {
  msg = ':(';
}


// Usando el operador ternario
const msg = data.isAdmin ? ':)' : ':('
```

Con el operador ternario, el primer valor después de la interrogación (?) es el que se devuelve si la expresión se evalúa a **true**. Si se evalúa a **false**, se devuelve el valor después de los dos puntos (:)

Parámetros por defecto

```
function add(a, b) {
  b = b === undefined ? 0 : b
  return a + b
}

const add = (a, b = 0) => a + b

console.log(add(2, 2));
> 4
console.log(add(2));
> 2
```

En muchas ocasiones, es necesario comprobar que los parámetros que se le pasan a una función existen y son correctos. Esta comprobación se suele realizar al principio del cuerpo de la función.

Para facilitar un gran número de casos de uso, JavaScript permite asignar un valor por defecto si no se pasa ningún parámetro.

Clases

```
class Question {
  constructor(ans) {
    this.answer = ans || 42;
  }
  setAnswer(ans){
    this.answer = ans;
  }
  getAnswer() {
    return this.answer;
  }
}

let myQuestion1 = new Question(32);
let myQuestion2 = new Question();
console.log(myQuestion1.getAnswer());
> 32
console.log(myQuestion2.getAnswer());
> 42
myQuestion1.setAnswer(33);
```

Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.

Tienen un método constructor que se llama para inicializar un objeto de la clase, que acepta parámetros iniciales.

Adicionalmente podemos definir otros métodos de la clase.

Clases: Herencia

```
class TrueFalseQuestion extends Question{
  constructor(ans) {
    super(ans);
    this.answer = ans || false;
  }
}
let tfq1 = new TrueFalseQuestion(true);
console.log(tfq1.getAnswer());
> true
let tfq2 = new TrueFalseQuestion();
console.log(tfq2.getAnswer());
> false
```

Una clase ECMAScript solo puede tener una clase padre, que se indica con la palabra **extends**.

Cuando vamos a modificar un método de la clase padre, llamamos al método **super**.

Podemos hacer uso de todas las funciones que ha definido la clase padre.

ESModules

```
// config.js
export const mode = 'development';
export const port = 3000;

// myLib.js
export default function add(a,b) {
  return a + b
}

// myApp.js
import addFunction from './myLib';
import {mode, port} from './config;
```

Con **export** hacemos nuestro código accesible desde otros archivos

Podemos exportar tantas cosas como queramos, pero sólo uno con la palabra **default**

Para usar el código exportado en otro fichero, utilizamos **import**. Al importar el default, podemos asignarle el nombre que queramos. En cambio, el resto de exports se importan entre llaves y usando su nombre original

Métodos de arrays

```
let products = [
    {
        name: "Toothpaste",
        price: 3.56
    },
    {
        name: "Ham",
        price: 9.99
    },
    {
        name: "Sunscreen",
        price: 6.45
    }
];
```

find

some

every

includes

map

filter

reduce

Métodos de arrays: map

La función **map** se aplica sobre un array y devuelve **otro array** de las mismas dimensiones, en el que cada elemento es el resultado de ejecutar la función que se le pasa como parámetro, a cada elemento del array original.

```
products.map(product => product.name)
> ["Toothpaste", "Ham", "Sunscreen"]

products.map((product, index) => {
  return (index + 1) + ". " + product.name;
})
> ["1. Toothpaste", "2. Ham", "3. Sunscreen"]
```

Métodos de arrays: some

La función **some** se aplica sobre un array y devuelve **true** si alguno de los elementos del array devuelve **true** al aplicarle la función que se le pasa como parámetro.

```
products.some(product => product.price < 5)
> true

products.some(product => product.price > 100)
> false
```

Métodos de arrays: filter

La función **filter** se aplica sobre un array y devuelve **otro array** que contiene sólo los elementos del array original que hayan dado como resultado **true**, al aplicarles la función que se le pasa como parámetro.

```
products.filter(product => product.price < 5)
> [{name: "Toothpaste", price: 3.56}]
```

Métodos de arrays: reduce

La función **reduce** se aplica sobre un array y devuelve un único valor **acc** resultado de aplicar la función que se le pasa como parámetro a cada elemento del array y a **acc**. Se puede proveer un valor inicial para **acc**.

```
products.reduce((acc, el)=>{return acc + el.price}, 0)
> 20.00
```

Fetch y Promesas

```
fetch(url)
        .then(res => res.json())
        .then(users => console.log(users))
        .catch(e => console.error(e))

fetch(url, {
           method: 'POST',
           body: JSON.stringify({ name: 'Alice'}),
           headers: {
             "Content-type": "application/json; charset=UTF-8"
                }
        })
        .then(res => console.log(res.status === "ok" ? "OK": "Error"))
        .catch(e => console.error(e))
```

Fetch y Promesas

```
console.log("A");
fetch(url)
    .then(res => console.log("B")
    .catch(e => console.error("C"))

console.log("D");
```

¿En qué orden aparecerán A,B,C,D si no se produce ningún error en la petición?

```
> A
> D
> B
```

Fetch con Async y Await

```
async function getUsers(url) {
    try {
      let result = await fetch(url);
      let users = await result.json();
      return users;
    } catch (error) {
      return error;
    }
}
```

Referencias

- **JavaScript to know for React**

  https://kentcdodds.com/blog/javascript-to-know-for-react

  https://www.freecodecamp.org/news/top-javascript-concepts-to-know-before-learning-react/

- **Arrow functions**

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

- **var vs let vs const in JavaScript**

  https://tylermcginnis.com/var-let-const/

- **Arrays**
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array