Módulo 10 - Conceptos Avanzados ES6 - Teoría

Let & Const

Var

Hasta ahora la única forma que teníamos para declarar variables era mediante la palabra reservada `var` :

```
var teacher = "Javi";
```

Características

Recordemos que las variables declaradas con `var` tienen las siguientes características:

1.- Es reasignable, esto es, podemos cambiar el valor que contiene tantas veces como queramos:

```
var teacher;
console.log(teacher); // undefined
teacher = "Javi";
console.log(teacher); // Javi
teacher = "Lissette";
console.log(teacher); // Lissette
teacher = "Daniel";
console.log(teacher); // Daniel
```

2.- Se pueden redeclarar variables con el mismo nombre:

```
var teacher = "Javi";
var teacher = "Lissette";
var teacher = "Daniel";
console.log(teacher); // Daniel
```

3.- Su **ámbito** o *scope* es de **función y no de bloque**. Esto significa que no podemos usarla fuera de dicho ámbito. Visto al contrario, también se podría entender como que una vez que se abandone dicho ámbito, la variable no puede ser usada, muere.

En este ejemplo, cuando se declara una variable `var` en una función, podremos usarla dentro de dicha función pero no podremos acceder a ella desde fuera, desde un ámbito superior:

```
function whoIsTeachingToday() {
  var teacher = "Javi";
  console.log(teacher);
}

whoIsTeachingToday(); // Javi
console.log(teacher); // Error!
```

Recuerda que un bloque (if-else, bucles, etc) no supone un ámbito para `var` por lo que si podremos acceder a ella:

```
if (true) {
  var teacher = "Javi";
  console.log(teacher); // Javi
}

console.log(teacher); // Javi
```

4.- Cuando una variable `var` no está declarada dentro de ninguna función (su ámbito natural), se declara globalmente en el objeto `window` :

```
var teacher = "Javi";
console.log(window.teacher); // Javi
```

Inconvenientes

Sin embargo, `var` presenta una serie de inconvenientes, quizá derivados de su exceso de flexibilidad, que hacen que su uso sea propenso a errores en ciertas circunstancias. Esto nos exige analizar con cuidado nuestro código para ver como se comportaría, en definitiva, nos exige "pensar demasiado". Algunos ejemplos son los siguientes:

Debido a su ámbito (y también a un mecanismo inherente a `var` llamado hoisting) sucede que podemos acceder a una variable `var` incluso antes de haberla declarado:

```
console.log(teacher); // Esperariamos un error, pero nos da undefined
var teacher = "Javi";
```

Y además podemos acceder a ella desde fuera si ha sido declarada en un bloque, lo cual puede resultar anti natural sobre todo cuando se ha trabajado previamente con otros lenguajes:

```
console.log(teacher); // Esperariamos un error, pero nos da undefined

if (true) {
  var teacher = "Javi";
  console.log(teacher); // Javi
}

console.log(teacher); // Seria más natural no poder acceder a teacher // Javi
```

El hecho de que las variables `var` se creen de forma global puede ser problemático ya que podríamos machacar variables globales previamente existentes, bien porque provengan de un fichero distinto o bien porque ya formaban parte del objeto `window`. Ejemplo:

```
console.log(window.alert); // Existe la propiedad window.alert
var alert = "Something went wrong!";
console.log(alert); // Something went wrong!.
console.log(window.alert); // He sobreescrito window.alert
```

Pero es que además puede conducir a situaciones realmente problemáticas como esta:

```
var sayNumber = [];
for (var i = 0; i <= 3; i++) {
  sayNumber[i] = () => console.log(i);
}
sayNumber[0](); // 4
sayNumber[1](); // 4
sayNumber[2](); // 4
sayNumber[3](); // 4
```

En este ejemplo cabría esperar que cada llamada a la función nos diese índices consecutivos. Sin embargo, recordemos que un bucle `for` no supone ámbito para `var` y por tanto el índice `i` en este ejemplo sería global. Por tanto, cada vez que se ejecuta alguna de las funciones almacenadas en nuestro array `sayNumber`, todas ellas acceden a la misma variable `i` global, cuyo valor tras finalizar el bucle será el último índice incrementado. Para verlo más claro, el código anterior sería equivalente a:

```
var sayNumber = [];
var i;
for (i = 0; i <= 3; i++) {
  sayNumber[i] = () => console.log(i);
}
console.log(i); // 4
sayNumber[0](); // 4
sayNumber[1](); // 4
sayNumber[2](); // 4
sayNumber[3](); // 4
```

ES6: Let y Const

Debido a los inconvenientes vistos anteriormente, se hizo necesario contar con una forma simple y natural de declarar variables con ámbito de bloque, y de este modo nacieron las palabras reservadas `let` y `const`.

1.- Mientras que `let` es reasignable, `const` no lo es. Esta es la única diferencia entre estos 2 tipos nuevos de variables.

```
let teacher;
console.log(teacher); // undefined
teacher = "Javi";
console.log(teacher); // Javi
teacher = "Lissette";
console.log(teacher); // Lissette
```

```
const teacher = "Javi";
console.log(teacher); // Javi
teacher = "Lissette"; // Error!
```

Una nota importante respecto a las variables `let` es que no se pueden inspeccionar antes de haber sido declaradas, ya que realmente no se inicializan hasta que no se declaran. Esto con `var` si podía hacerse. A este fenómeno se le conoce como *temporal dead zone*:

```
console.log(uno); // undefined
var uno;

console.log(dos); // Error!
let dos;
```

Por lo demás, comparten las mismas características.

2.- En ambos casos, no son redeclarables:

```
let teacher = "Javi";
let teacher = "Lissette"; // Error!
```

```
const teacher = "Javi";
const teacher = "Lissette"; // Error!
```

3.- En ambos casos su ámbito es de bloque. Esto significa que cualquier bloque, tanto if-elses, switches, bucles, como funciones, suponen un *scope* local para estas variables:

```
if (true) {
  const teacher = "Javi";
  console.log(teacher); // Javi
}

console.log(teacher); // Error! teacher tiene ámbito local dentro del bloque if
```

4.- En ambos casos, no crean variables globales, sino como mucho a nivel de fichero, que representará el *scope* superior si no se han declarado dentro de ningún bloque. No más confusiones con propiedades del objeto `window`.

```
const teacher = "Javi";
console.log(teacher); // Javi
console.log(window.teacher); // undefined
```

¿Cuando usar let? ¿Cuando usar const?

A menos que necesites reasignación, en cuyo caso deberás usar `let`, utiliza siempre `const`. Por tanto, como norma general, comienza siempre utilizando `const`, y sólo cuando necesites reasignar dicha variable, promocionala a `let`.

Lo que si debes hacer a partir de ahora es olvidarte de `var`.

Resumen

	var	let	const
Reasignable	Si	Si	No
Redeclarable	Si	No	No
Ámbito	Función	Bloque	Bloque
Global	Si	No	No

Spread

Antes de abordar la definición de `spread` recordemos que un **iterable** en JavaScript era una estructura de datos a modo de colección, conformada por diversos items o elementos que podían recorrerse. El ejemplo más claro son los `arrays`, pero también son iterales, por ejemplo, los `strings` ya que no dejan de ser una colección de caracteres.

`Spread` es un nuevo y potente operador de JavaScript super que nos permite extender o difundir los elementos de un objeto iterable (cada item de una colección) en aquellos sitios donde se espere cero, uno o más elementos. Es decir, con *spread* podemos **repartir** cada item de una colección a la posición donde corresponda.

Esto se entenderá mucho mejor con los ejemplos prácticos.

Spread en llamadas a funciones

Con el operador `spread` podremos repartir los diferentes items de una colección a cada uno de los argumentos de entrada de una función, por ejemplo:

```
const names = ["Javi", "Lissette", "Dani"];

const sayNames = (name1, name2, name3) => {
  console.log(name1);
  console.log(name2);
  console.log(name3);
};

// sayNames("Javi", "Lissette", "Dani"); // Forma literal
// sayNames(names[0], names[1], names[2]); // Forma tradicional

sayNames(...names); // Con spread operator
```

O por ejemplo:

```
const names = ["Javi", "Lissette", "Dani"];

const greetPeople = (greeting, name1, name2, name3) => {
  console.log(greeting + " " + name1 + ", " + name2 + " y " + name3);
};

greetPeople("Buenos días", ...names);
```

Spread en arrays

`Spread` nos va a permitir crear arrays a partir de otros de forma super potente y expresiva sin tener que recurrir a bucles. Veamos:

Podemos inicializar un array a partir de otro de forma inmediata:

```
const original = [1, 2, 3];
const copy = [...original]; // Extendemos cada elemento del array origen en el array destino
console.log(copy); // [1, 2, 3]
```

De forma tradicional, tendríamos que haber hecho:

```
const original = [1, 2, 3];
const copy = [];
for (let i = 0; i < original.length; i++) {
  copy.push(original[i]);
}
console.log(copy);
```

Pero es más, podemos jugar a crear nuevos arrays a partir de otros, concatenando, intercalando, etc:

```
const threeAndFour = [3, 4];
const oneToFive = [1, 2, ...threeAndFour, 5];
console.log(oneToFive); // [1, 2, 3, 4, 5]
```

```
const weekend = ["sábado", "domingo"];
const workDays = ["lunes", "martes", "miércoles", "jueves", "viernes"];
const week = [...workDays, ...weekend];
console.log(week); // ["lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"]
```

Spread en objetos

A partir de ES2018 (ES9) también se puede utilizar el operador `spread` para trabajar con objetos, de modo que al igual que con los arrays o cualquier otro iterable, podemos inicializar objetos a partir de otros, mezclar, etc:

```
const original = {
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const copy = { ...original };
console.log(copy);
```

```
const id = {
  name: "Javi",
  surname: "Calzado",
};
const details = {
  age: 36,
  phone: 654123456,
};

const user = { ...id, ...details }; // Mezclamos en un nuevo objeto
console.log(user);
```

El orden en el que se haga el spread de las propiedades importa mucho:

```
const user = {
  name: "Javi",
  age: 36,
};

const newUser1 = {
  name: "Lissette",
  ...user,
};
console.log(newUser1); // {name: "Javi", age: 36}

const newUser2 = {
  ...user,
  name: "Lissette",
};
console.log(newUser2); // {name: "Lissette", age: 36}
```

Rest

El operador `rest` es similar a `spread`, de hecho su notación es la misma (...), pero con un comportamiento opuesto: en lugar de extender o repartir elementos, los agrupa y los ofrece en formato de array. Rest se utiliza para agrupar argumentos de entrada de una función en un array. Veamos ejemplos:

Ejemplo básico de funcionamiento. Fíjate como `myArguments` es una variable local a la función que contiene todos los argumentos que se le han pasado en la llamada a dicha función:

```
const myFunction = (...myArguments) => {
  console.log(myArguments);
  console.log(myArguments.length); // Es un array!
};

myFunction("uno", 2, true);
```

Este operador nos va a permitir hacer funciones muy flexibles, cuyo número de argumentos de entrada no tenga que ser fijo ni estar previamente definido:

```
const greetPeople = (greeting, ...names) => {
  let nameListString = "";
  for (const name of names) {
    nameListString += " " + name;
  }
  console.log(greeting + nameListString);
};

greetPeople("Buenos días"); // Buenos días
greetPeople("Buenos días", "Javi"); // Buenos días Javi
greetPeople("Buenos días", "Javi", "Lissette"); // Buenos días Javi, Lissette
```

Otro ejemplo muy útil donde se ve el valor del operador rest:

```
const sum = (...numbers) => {
  let result = 0;
  for (const n of numbers) {
    result += n;
  }
  return result;
};

console.log(sum()); // 0
console.log(sum(1)); // 1
console.log(sum(1, 2)); // 3
console.log(sum(1, 2, 3)); // 6
```

Destructuring

Se le llama `destructuring` a un sintaxis avanzada de JavaScript que nos permite extraer de forma cómoda y fácil:

- Elementos de un array.
- Propiedades de un objeto.

Y almacenarlos en variables, todo ello de la manera más compacta posible.

Destructuring sobre arrays

Veamos como extraeríamos elementos a variables en un array de forma clásica:

```
const teachers = ["Javi", "Lissette", "Jaime", "Victor", "Dani"];
const primero = teachers[0];
const segundo = teachers[1];
const tercero = teachers[2];
console.log(primero, segundo, tercero);
```

y ahora usando *destructuring*:

```
const teachers = ["Javi", "Lissette", "Jaime", "Victor", "Dani"];
const [primero, segundo, tercero] = teachers;
console.log(primero, segundo, tercero);
```

El ejemplo primero es equivalente a este segundo, pero gracias al *destructuring* podemos extraer múltiples elementos a las variables que queramos en una sola línea y de una sola vez.

Además, podemos omitir elementos en el *destructuring* y 'capturar' solo aquellos que nos interesan, por ejemplo:

```
const teachers = ["Javi", "Lissette", "Jaime", "Victor", "Dani"];
const [, , tercero] = teachers;
console.log(tercero);
```

Es muy común utilizar el *destructuring* en argumentos de una función que sean de tipo array, para capturar o extraer rápidamente alguno de sus elementos, por ejemplo:

De forma clásica haríamos:

```
const teachers = ["Javi", "Lissette", "Jaime", "Victor", "Dani"];
const getSecondTeacher = teachers => {
  const second = teachers[1];
  return second;
};
console.log(getSecondTeacher(teachers));
```

y aplicando *destructuring*:

```
const teachers = ["Javi", "Lissette", "Jaime", "Victor", "Dani"];
const getSecondTeacher = ([, second]) => second;
console.log(getSecondTeacher(teachers));
```

Destructuring sobre objetos

Todo lo anterior es aplicable también sobre objetos. Es decir, se puede hacer *destructuring* para 'capturar' propiedades de objetos. Veamos:

Dado un objeto con diversas propiedades, de forma clásica podríamos extraer algunas propiedades a variables del siguiente modo:

```
const user = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const id = user.id;
const name = user.name;
const age = user.age;
console.log(id, name, age);
```

aplicando *destructuring* simplificamos y compactamos la extracción de propiedades:

```
const user = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const { id, name, age } = user;
console.log(id, name, age);
```

También es muy utilizado el *destructuring* sobre objetos en argumentos de función:

Si de forma clásica haríamos:

```
const userSample = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const getUserFullName = user => user.name + " " + user.surname;
console.log(getUserFullName(userSample));
```

con destructuring tendríamos:

```
const userSample = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const getUserFullName = ({ name, surname }) => name + " " + surname;
console.log(getUserFullName(userSample));
```

Dos características avanzadas del destructuring sobre objetos son:

1.- Podemos cambiar el nombre de las propiedades que capturamos:

Es útil para evitar colisiones como esta:

```
const id = 2834; // variable "id" previamente declarada.
const user = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const { id, name, age } = user; // Error! id ya está declarada anteriormente
console.log(id, name, age);
```

en cuyo caso haríamos:

```
const id = 2834; // variable "id" previamente declarada.
const user = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const { id: userId, name, age } = user; // Capturo la propiedad id y la almaceno en userId
console.log(userId, name, age);
```

podemos cambiar multiples nombres si queremos:

```
const user = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
};
const { id: userId, name: userName, age: userAge } = user;
console.log(userId, userName, userAge);
```

2.- Se puede hacer destructuring en profundidad, para capturar propiedades de objetos que sean profundas (más allá del primer nivel).

Supongamos que nuestro usuario es un objeto con objetos anidados dentro:

```
const user = {
  id: 4451234,
  name: "Javi",
  surname: "Calzado",
  age: 36,
  location: {
    country: "Spain",
    city: "Málaga",
    postalCode: 29017,
  },
};

const getUserNameAndCountry = ({ name, location: { country } }) => name + " from " + country;

console.log(getUserNameAndCountry(user));
```

También se puede combinar destructuring de arrays y objetos:

Objetos como elementos de arrays:

```
const classes = [
  { teacher: "Javi", subject: "Destructuring", duration: 3 },
  { teacher: "Lissette", subject: "Rest/Spred", duration: 2 },
];
const secondClassDuration = ([, { duration }]) => duration;
console.log(secondClassDuration(classes));
```

O al contrario, arrays como propiedades de un objeto:

```
const lemoncode = {
  teachers: ["Javi", "Lissette", "Jaime"],
  type: "Company",
  foundation: 2015,
};
const getFirstTeacher = ({ teachers: [first] }) => first;
console.log(getFirstTeacher(lemoncode));
```

Combinando Destructuring y Rest

Una aplicación de estas técnicas muy frecuentemente usada es la combinación de *destructuring* con el operador *rest* para agrupar los elementos restantes de un array tras hacerle destructuring, o las propiedades restantes de un objeto tras hacerle destructuring. Veamos ejemplos:

Ejemplo básico tanto con arrays:

```
const sampleArray = ["uno", 2, true];
const [firstItem, ...restItems] = sampleArray;
console.log(firstItem, restItems);
```

como con objetos:

```
const sampleObject = {
  id: 43,
  name: "Javi",
  age: 36,
};
const { id, ...rest } = sampleObject;
console.log(id, rest);
```

Manipulando arrays sin bucles

Hasta ahora hemos usado los bucles para manejar estructuras iterables o colecciones. Sin embargo, JavaScript nos proporciona métodos con las utilidades más comunes y usadas para manipular dichas colecciones sin necesidad de recurrir a bucles.

Usaremos el siguiente array como ejemplo:

```
const bookCollection = [
  { isbn: 23453, author: "J. Simmons", pages: 250, title: "The Dark", read: true },
  { isbn: 56456, author: "Peter Black", pages: 120, title: "Feed", read: false },
  { isbn: 43243, author: ["A. Smith", "F. Gant"], pages: 340, title: "Fire", read: true },
  { isbn: 23223, author: undefined, pages: 260, title: "Eve", read: true },
  { isbn: 89232, author: "Anna Willis", pages: 610, title: "The Run", read: false },
];
```

Veamos las utilidades:

Every

Devuelve `true` si **todos los elementos** del array cumplen con una condición dada. La condición la expresamos en forma de función (*callback*):

```
const allBooksRead = books => books.every(book => book.read);

console.log(allBooksRead(bookCollection)); // false
```

Some

Similar a `every`, pero en este caso devuelve `true` si **al menos un elemento** del array cumplen con una condición dada. La condición la expresamos en forma de función (*callback*):

```
const anyBookRead = books => books.some(book => book.read);

console.log(anyBookRead(bookCollection)); // true
```

Find

Obtiene el primer elemento del array que cumpla una condición dada. Dicha condición se expresa, nuevamente, mediante una función (*callback*).

```
const findEveBook = books => books.find(book => book.title === "Eve");
console.log(findEveBook(bookCollection));

const findUnknownAuthorBook = books => books.find(book => !book.author);
console.log(findUnknownAuthorBook(bookCollection));
```

Versión FindIndex

```
const findEveBook = books => books.findIndex(book => book.title === "Eve");
console.log(findEveBook(bookCollection));

const findUnknownAuthorBook = books => books.findIndex(book => !book.author);
console.log(findUnknownAuthorBook(bookCollection));
```

Filter

Te permite extraer del array los elementos que cumplan cierta condición. Devuelve un nuevo array con dichos elementos. La condición de filtrado la expresamos en forma de función (*callback*) que debe devolver `true` para quedarse con el elemento en cuestión o `false` para descartarlo (filtrarlo):

```
const unreadBooks = books => books.filter(book => book.read === false);
console.log(unreadBooks(bookCollection));
```

```
const booksOver300Pages = books => books.filter(book => book.pages >= 300);
console.log(booksOver300Pages(bookCollection));

const multiAuthorBooks = books => books.filter(book => Array.isArray(book.author));
console.log(multiAuthorBooks(bookCollection));
```

Map

Junto con `reduce`, esta es una de las utilidades estrella para manipular arrays. `Map` permite aplicar una función transformadora elemento a elemento. Por tanto, genera un nuevo array, donde cada elemento ha sido transformado (modificado) por dicha función.

```
const getTitleCollection = books => books.map(book => book.title);
console.log(getTitleCollection(bookCollection));

const addOwner = (owner, books) =>
  books.map(book => {
    return { ...book, owner };
  });
// const addOwner = (owner, books) => books.map(book => ({...book, owner}));
console.log(addOwner("Javi", bookCollection));
console.log(addOwner("Lissette", bookCollection));
```

Reduce

`Reduce` es una utilidad hermana de `map` y junto a ella son las manipulaciones estrella de los arrays. Tanto es así que se suele decir que cualquier manipulación de un array podría expresarse en términos de `map` y `reduce`. `Reduce` es similar a `map`, es decir, aplica una función transformadora elemento a elemento, pero en vez de devolver un array, va acumulando el resultado y nos devuelve un único dato.

```
// De un array, devolvemos un número.
const getTotalPages = books => books.reduce((sumPages, book) => sumPages + book.pages, 0);
console.log(getTotalPages(bookCollection));

// De un array, devolvemos un string.
const getTitlesString = books => books.reduce((titles, book) => titles + " " + book.title, "");
console.log(getTitlesString(bookCollection));

// De un array, devolvemos un número.
const getNumberUnread = books =>
  books.reduce((unreadNum, book) => {
    if (!book.read) unreadNum++;
    return unreadNum;
  }, "");
console.log(getNumberUnread(bookCollection));
```

Otras utilidades interesantes

ForEach

Ejecuta un bucle internamente, permite iterar elemento a elemento:

```
const showTitles = books => books.forEach(b => console.log(b.title));
showTitles(bookCollection);
```

Sort

Ordena los elementos de un array en base a una función de comparación (*callback*). La función de comparación acepta 2 elementos a comparar como argumentos (a y b) y debe devolver:

- Valor negativo (-1), si, en función del criterio de comparación, a debe colocarse antes que b.
- Valor positivo (1), si, en función del criterio de comparación, b debe colocarse antes que a.
- Valor cero (0) si, en función del criterio de comparación, a y b deben dejarse tal cual.

```
const sortByLength = books => books.sort((a, b) => (a.pages > b.pages ? 1 : -1));
sortByLength(bookCollection);
console.log(bookCollection);
```

Join

Concatena todos los elementos de un array en un `string` y lo devuelve. Admite utilizar un carácter separador para la concatenación:

```
const sample1 = ["Javi", "Lissette", "Dani"];
const sample2 = ["Texto", 43, true];
console.log(sample1.join("-"));
console.log(sample2.join(" & ")); // Hace casting a string de los elementos.
```

Ejemplos con múltiples utilidades

```
const getAllReadPages = books =>
  books.filter(book => book.read).reduce((sumPages, book) => sumPages + book.pages, 0);
console.log(getAllReadPages(bookCollection));

const extractUnreadBooksSorted = books =>
  books
    .filter(book => !book.read)
    .map(book => ({ title: book.title, pages: book.pages }))
    .sort((a, b) => b.pages - a.pages);
console.log(extractUnreadBooksSorted(bookCollection));

const getSortedTitles = books =>
  books
    .map(book => book.title)
    .sort() // Si no le paso función de comparación sigue ordenación alfabética por defecto
    .join(", ");
console.log(getSortedTitles(bookCollection));
```

Conclusiones

- Utiliza `const` para declarar variables a partir de ahora.
- Si necesitaras reasignar la variable, promociónala a `let`.
- Utiliza `spread` para inicializar cómodamente arrays u objetos a partir de otros, o bien mezclarlos, clonarlos, etc. También aprovéchate de la potencia de `spread` para repartir elementos como argumentos de entrada de una función.
- `Rest`, con la misma notación que `spread`, se utiliza para agrupar argumentos de una función en un array, permitiéndote trabajar con múltiples argumentos sin tener que definirlos previamente.
- Para extraer propiedades de objetos o elementos de arrays rápidamente y almacenarlos en variables, utiliza la sintaxis de `destructuring`.
- Haz uso de las funciones de utilidad que te ofrecen los arrays para manipularlos sin tener que recurrir a bucles.

Recursos

- Javascript Basics: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types
- Var: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var
- Let: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
- Const: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
- Spread: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
- Rest: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
- Destructuring: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- No mas bucles FOR: https://lemoncode.net/lemoncode-blog/2017/6/22/javascript-es6-no-mas-bucles-for