

JAVASCRIPT

JS

Contenido

★ FETCH

- then()
- catch()

★ Web Storage

- localStorage
- sessionStorage

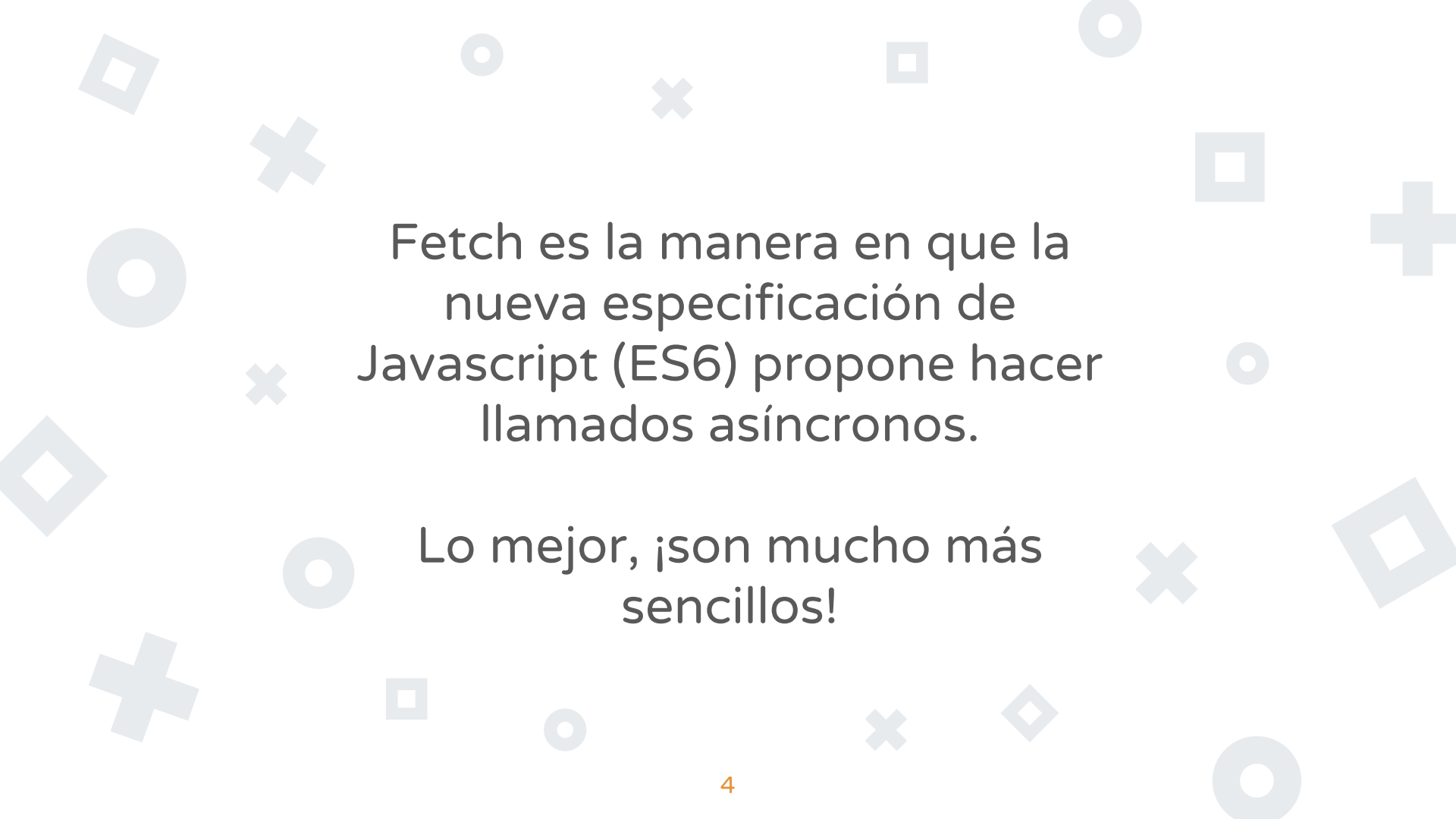
★ ES6 - ECMAScript 2015

- let & const
- Template literals
- Arrow functions
- Spread operator
- Destructuring
- Default parameters



FETCH

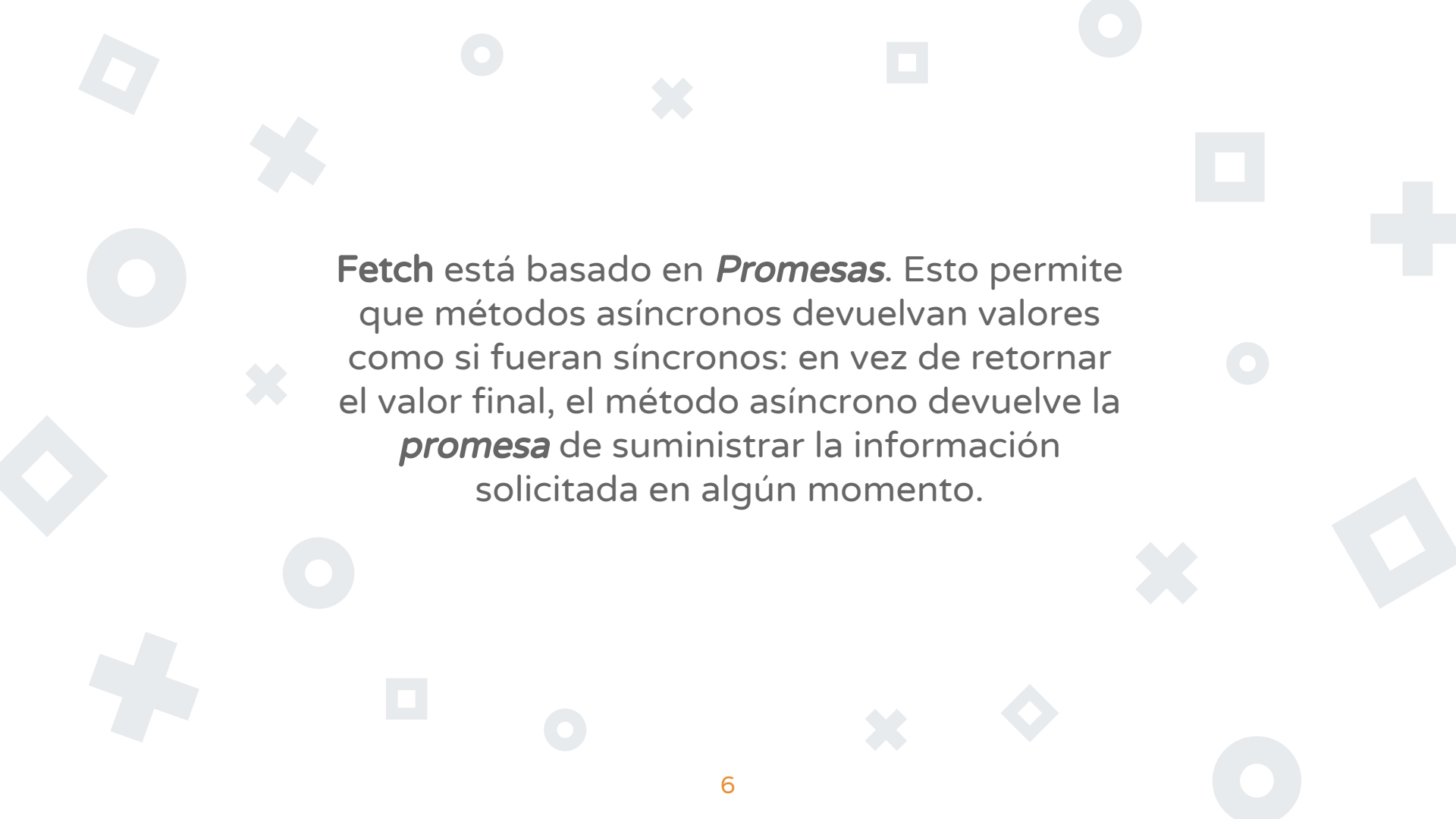
Llamados Asincrónicos de ES6
¡Más sencillos!



Fetch es la manera en que la
nueva especificación de
Javascript (ES6) propone hacer
llamados asíncronos.


Lo mejor, ¡son mucho más
sencillos!

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    // do stuff with data;
  })
  .catch(function (error) {
    console.log("The error was: " + error);
  })
```



Fetch está basado en ***Promesas***. Esto permite que métodos asíncronos devuelvan valores como si fueran síncronos: en vez de retornar el valor final, el método asíncrono devuelve la ***promesa*** de suministrar la información solicitada en algún momento.

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    // do stuff with data;
  })
  .catch(function (error) {
    console.log("The error was: " + error);
  })
```



Fetch recibe como primer parámetro la URL del endpoint al cual estamos haciendo el llamado asíncrono.

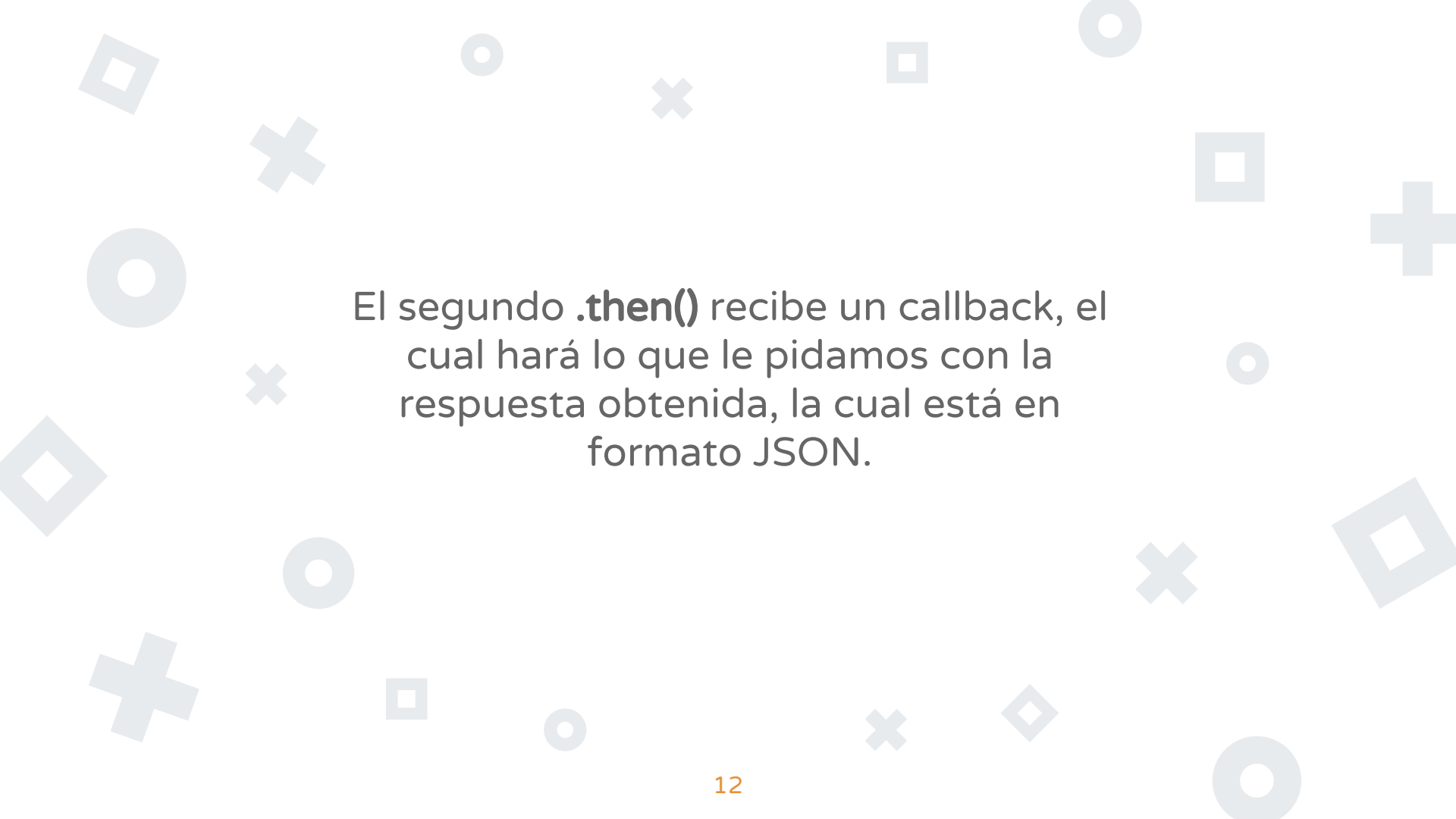

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    // do stuff with data;
  })
  .catch(function (error) {
    console.log("The error was: " + error);
  })
```

El primer **.then()** recibe un callback, el cual retornará la respuesta del llamado asíncrono en formato JSON.

Indispensable esto:

```
return response.json();
```

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    // do stuff with data
  })
  .catch(function (error) {
    console.log("The error was: " + error);
  })
```



El segundo **.then()** recibe un callback, el cual hará lo que le pidamos con la respuesta obtenida, la cual está en formato JSON.

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    // do stuff with data
  })
  .catch(function (error) {
    console.log("The error was: " + error);
  })
```

El **.catch()** atraparé los errores, si los
hubiere, e imprimirá en consola el tipo de
error obtenido.

¿Y para enviar datos vía POST?

- ✕ Si queremos enviar datos vía POST con **fetch()** necesitamos pasar como segundo parámetro un Objeto Literal con la siguiente información:

```
fetch("http://localhost/post.php", {  
    method: 'POST',  
    body: datosDelFormulario  
})  
  
    .then(function (response) {  
        return response.text();  
    })  
    .then(function (data) {  
        // do stuff with data;  
    })
```


Pero ¿Y qué es datosDelFormulario?

Es una variable que guarda la instancia del objeto **FormData()** al cual, después de ser instanciado, le tenemos que insertar los datos deseados en un String tipo JSON.

```
var campos = {  
    nombre: "Ada",  
    apellido: "Lovelace",  
}  
var datosDelFormulario = new FormData();  
datosDelFormulario.append('datos', JSON.stringify(campos));  
  
fetch("http://localhost/post.php", {  
    method: 'POST',  
    body: datosDelFormulario  
})
```

De esta manera en `$_POST` de PHP ahora tendremos una posición llamada **datos**, en la cual está el String JSON que guarda los datos enviados con **fetch()**.



WEB STORAGE

Persistencia de los datos

Web Storage es una herramienta que JS nos da para poder almacenar información en el navegador del usuario. Es algo como `$_COOKIE` y `$_SESSION` de PHP.

La información se almacena en el cliente, NO en el servidor.



Web Storage nos provee de dos
métodos para almacenar información:

- localStorage
- sessionStorage

Web Storage

.localStorage

Guarda información sin tiempo de expiración. Sirve para leer y escribir datos. Es un objeto literal en el cual podemos setear propiedades y valores.

```
localStorage.setItem("userName", "Juana"); // Setea el atributo userName  
localStorage.getItem("userName"); // Juana  
localStorage.removeItem("userName"); // Elimina el atributo userName
```

// Al igual que las COOKIES, localStorage es información única que se guarda por dominio y navegador.

Web Storage

.sessionStorage

Guarda información mientras se mantenga abierto el navegador. Sirve para leer y escribir datos. Es un objeto literal en el cual podemos setear propiedades y valores.

```
sessionStorage.setItem("userID", 32); // Setea el atributo userID  
sessionStorage.getItem("userID"); // 32  
sessionStorage.removeItem("userID"); // Elimina el atributo userID
```

```
// Al igual que las SESSION, sessionStorage se borra al cerrar la ventana  
del navegador.
```




ECMAScript 6

Es la nueva especificación de lenguaje

ES6 o ECMAScript 2015 es la nueva especificación del lenguaje.

Trae consigo una cantidad de nuevas y poderosas funcionalidades que hacen de JS un lenguaje mucho más potente.

ECMAScript 2015

let

Sirve para definir variables. A diferencia de **var**, **let** es una variable de bloque. Esto nos permite crear variable con el mismo nombre sin sobrescribir sus valores.

```
let name = "Ada Lovelace";  
if(true) {  
    let name = "Tim Berners Lee";  
    console.log(name); // "Tim Berners Lee"  
}  
console.log(name); // "Ada Lovelace"
```

ECMAScript 2015

const

const nos sirve para declarar constantes. Es decir, es una *"variable"* que **NUNCA** podrá cambiar su valor. A menos que sea un objeto literal.

```
const DNI = 23456987; // Crea una constante llamada DNI
DNI = "PAS-324567"; // ERROR: Assignment to constant variable.
const STUDENT_DATA = {
  code: "FS345618TN"
}
STUDENT_DATA.email = "adalovelace_ok@gmail.com"; // Permitido
```

ECMAScript 2015

Template literals

Nos permite una nueva y mejor manera de "concatenar" diferentes valores en un string. Usa comillas francesas `` e interpolación de variables.

```
let price = 950.45;  
let product = "Remera para nena";  
let message = `La ${product} cuesta ${price}`;  
console.log(message); // "La Remera para nena cuesta 950.45"
```

// Dentro de la interpolación \${} podemos operar con cualquier funcionalidad de JS

ECMAScript 2015

Arrow Functions

Una nueva forma de escribir una función en la cual no es necesaria la palabra reservada *function* ni incluso, en ocasiones, el *return*.

```
let suma = (n1, n2) => n1 + n2;  
let sayHello = () => "Hello world!";  
let contactForm = document.querySelector(".contact-form");  
contactForm.addEventListener("submit", e => e.preventDefault())
```

// Para usar como callback es una gran herramienta

ECMAScript 2015

Spread operator (...)

Nos permite "esparcir" datos dentro de un Array u Objeto de manera sencilla. Podemos copiar información de un lugar y trasladarla a otro fácilmente.

```
let arrayOne = ["Ada", "Grace"];  
let arrayTwo = ["Tim", "Vin"];  
arrayOne = [...arrayOne, ...arrayTwo]; // Ada, Grace, Tim, Vin  
let arrayThree = [...arrayOne, "Brendan"]; // Ada, Grace, Tim, Vin, Brendan  
  
// Funciona de manera similar para los objetos literales
```

ECMAScript 2015

Default parameters

Una manera sencilla de definir valores para los parámetros que se pasan a una función.

```
let sayHello = (name = "Stranger") => `Hello ${name}`;  
sayHello("Ada"); // Hello Ada  
sayHello(); // Hello Stranger
```

// Una herramienta muy funcional, pues dentro de la función, podemos implementar toda la lógica deseada.

¡A practicar!

Practica Integradora



```
// To Do  
console.log("Practice Time");
```