educaciónIT
CODE YOUR FUTURE

JavaScript desde cero

Módulo 5

educaciónIT
CODE YOUR FUTURE

La evolución de ECMAScript

La evolución de ECMAScript

JavaScript nació en **1996**, de la mano de la empresa **Netscape Inc.**, creadora del navegador web Netscape. Su planteo de ser un lenguaje fácil de comprender, hizo que sus *web browsers* competidores, comenzaran a crear sus "adaptaciones" del lenguaje JS.

Y, para no desvirtuar el mercado, Netscape convirtió a JS en un estándar regulado, el cual pasó a manos de ***ECMA International***. Esta organización, evoluciona de forma contínua el lenguaje JS, con nuevas características que lo adapten a necesidades modernas.

Veamos a continuación su evolución y la importancia que esta tiene.

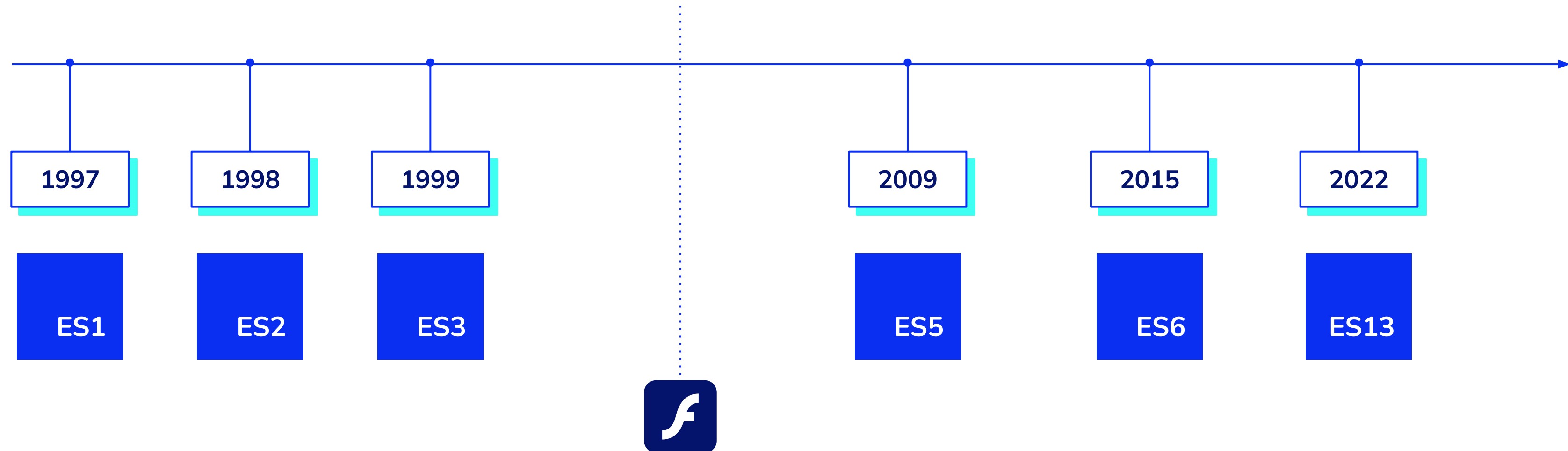
ES1

Un año después de la estandarización del lenguaje JS, **1997**, nació **ECMAScript versión 1**. Esto consiguió que, todo *web browser* que se jactara de dar soporte al lenguaje JS, debía respetar el estándar delineado por *Ecma International*.

Esta movida desalentó la creación de “alternativas” a JS, dado que los *web browsers* se volverían complejos y pesados por tener que dar soporte a HTML, CSS, JS más “otras variantes”.

Evolución y siguientes versiones

A la par de la evolución tecnológica del *hardware* y de las velocidades de Internet, **EcmaScript comenzó a mejorar el lenguaje JavaScript, de forma evolutiva**:



- Si bien, la evolución de ES estuvo casi interrumpida entre **1999 y 2009**, por la gran popularidad que tuvo ***Flash Player*** en el desarrollo web de sitios interactivos, en esta época JS adquirió grandiosas herramientas como la capacidad de interactuar con servidores de *backend* integrando el paradigma **AJAX** y **XMLHttpRequest**.

- Su quinta versión, **ES5**, trajo cambios funcionales dentro del lenguaje, que lo hicieron cada vez más interesante.

- Finalmente, en **2015**, llegó **ES6**, la cual generó un **cambio incipiente en novedades dentro de este lenguaje**.

- Desde este año, JS evoluciona a una nueva versión cada 365 días. Es por ello que ES6 es la más comentada en noticias de este lenguaje y hasta en descripciones de puestos laborales.

JavaScript moderno

JavaScript moderno

Dentro de la gran revolución que trajo **EcmaScript versión 6**, encontramos algunas de las funcionalidades que vimos a lo largo de este curso:

- `querySelector` y `querySelectorAll`.
- Manejo de eventos con `addEventListener`.
- ***Template String + Literals*** y el uso del carácter ***backtick*** ``` `` ```.

Veamos algunas otras funcionalidades:

- Operador ternario.
- Operador lógico *AND*.
- Operador lógico *OR*.
- Operador *nullish coalescing*.

Operador ternario

El operador ternario se integra a JS para permitir **simplificar** 5 líneas de código que estructuran un **`if - else`** convencional, **en una sola línea de código**.

Este operador existe en otros lenguajes de programación, aparte de JavaScript:

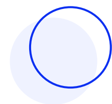
- *PHP.*
- *Java*.
- El lenguaje *C.*

Veamos un ejemplo
en las próximas pantallas.

```
const button = document.querySelector("button button-cart")
button.disabled = true

const confirmar = confirm("¿Confirmas la compra de tu carrito?")

if (confirmar === true) {
    button.classList.add("icon-spinner")
else {
    button.disabled = false
}
```



Su estructura combina el signo `?` que reemplaza la estructura correspondiente a `if {}`.

La estructura correspondiente a `else {}` es reemplazada por el signo `:` (**dos puntos**).

Debemos tener en cuenta que, en ambas condiciones, sólo podemos resolver con **una única línea de código**.

```
const confirmar = confirm("¿Confirmas la compra de tu carrito?")

confirmar ? button.classList.add("icon-spinner") : button.disabled = false
```

Si tenemos **múltiples líneas** de código que se ejecutan tanto en el bloque `if {}` como en el bloque `else {}`, podemos **encerrarlas en funciones para luego invocarlas más fácilmente.**

```
const confirmar = confirm("¿Confirmas la compra de tu carrito?")

confirmar ? confirmarCompra() : retornarAlHome()
```

Además, **el operador ternario funciona con retorno implícito** por lo cual, al evaluar la condición, es posible retornar cualquier valor como resultado, tanto con la expresión '**?**' como con la expresión '**:**'

```
const confirmar = confirm("¿Confirmas la compra de tu carrito?")

button.disabled = confirmar ? true : false
```

Operador lógico *AND*

El operador lógico *AND* representado por dos caracteres *ampersand* '**&&**', se utiliza para **confirmar la evaluación positiva de dos condiciones dentro de un bloque `if`.**

Veamos un ejemplo a la derecha:

```
let username = ''

if (usuario.trim() !== '' && password.trim() !== '') {
    username = validarUsuario()
}
```

Desde **ES6**, este operador lógico se puede utilizar también para **simplificar un condicional** representado por un **`if`** simple.

Ejemplo:

```
const carrito = retornarContenidoCarrito()

if (carrito.length > 0) {
    cargarCarritoEnPantalla()
}
```

```
const carrito = retornarContenidoCarrito()

(carrito.length > 0) && cargarCarritoEnPantalla()
```

Operador lógico *OR*

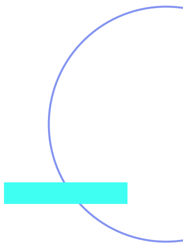
El operador lógico *OR* representado por dos caracteres *pipe* '||', también es utilizado en los condicionales múltiples para **evaluar el cumplimiento de una condición, u otra**.

En ES6, nos ayuda, entre otras cosas a **simplificar estructuras de asignación de valores a diferentes datos**.

En el ejemplo de la siguiente pantalla, se usa el condicional `if - else`, para asignar un valor u otro a un texto en pantalla.

```javascript
const parrafo = document.querySelector("span.mensaje-bienvenida")
const username = obtenerNombreDeUsuario()

if (username !== '') {
    parrafo.textContent = `Bienvenido ${username}`
} else {
    parrafo.textContent = `Bienvenido invitado`
}
```



El operador lógico *OR* funciona como una **opción de *fallback***:

- si un valor **no cumple** con un dato específico almacenado (izquierda),
- podremos definir un **valor alternativo del lado opuesto** del operador lógico (derecha).

```
const parrafo = document.querySelector("span.mensaje-bienvenida")
const username = obtenerNombreDeUsuario()

parrafo.textContent = `Bienvenido ${username || 'invitado'}`
```

Tabla de valores *falsy*

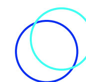
Esta comparación que realiza el operador lógico *OR*, se basa en lo que se denomina: *tabla de valores falsy*.

Cualquier valor que se compare y que retorne algunos de estos resultados, **el operador lógico *OR* permitirá definir un valor *fallback* alternativo al esperado**.

Valor *falsy*	Descripción
`‘’`	Cadenas de texto vacías.
`0`	El número cero.
`null`	Valor nulo.
`undefined`	Valores *undefined*.
`false`	Valor *booleano* falso.

Operador *nullish coalescing*

El operador *nullish coalescing* está representado por el doble signo **??**, y **se comporta igual que el operador lógico *AND*, con la diferencia de que acepta menos valores que la *Tabla de valores falsy*** vista anteriormente.



```
const valor = null

const valorPredeterminado = valor ?? "Valor predeterminado"
console.log(valorPredeterminado)
```

En este caso, el operador *nullish coalescing*, toma solamente como valores *nullish* a los tipos de datos `null` y `undefined`.

Los valores **`0`**, **caracteres vacíos** `‘’`, o ***booleano* del tipo *false***, son considerados como valores óptimos para este operador.

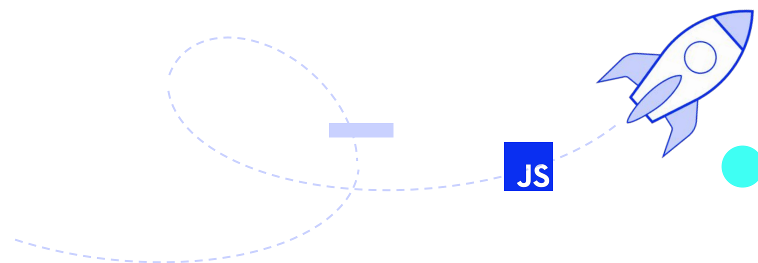
Valor *nullish*	Descripción
`null`	Valor nulo.
`undefined`	Valores *undefined*.

Conclusión

Todos los **operadores modernos** explicados en las diapositivas anteriores, fueron integrados desde ES6, en el año 2015.

No es obligatorio utilizarlos, pero sí es importante que te familiarices con ellos, porque son **sumamente requeridos en el mercado laboral** actual y, además se utilizan de forma natural en **reemplazo de los operadores condicionales tradicionales**, dentro de React, Angular, Vue, y del resto de *frameworks* JS modernos.

También es bueno que sigamos de cerca la evolución de JS en cuanto a novedades dado que, desde el año 2015, JS lanza nuevas características todos los años.



educaciónIT
CODE YOUR FUTURE

¡Sigamos trabajando!