

Desarrollo Backend

Bienvenid@s

Principios del lenguaje JavaScript II

Clase 03



Pon a grabar la clase



Temario

- Array de elementos
 - Métodos asociados
- Ciclos de iteración
 - por conteo (for)
 - por repetición (while - do while)
 - interrupción de los ciclos
 - salteo en la iteración
- Objeto literal
 - propiedades
 - métodos
- Array de objetos



Principios del lenguaje JS

Principios del lenguaje JS

Ten presente que, en estas primeras clases, estamos realizando un repaso del lenguaje JS general, porque necesitamos que tengas presente y que comprendas muchas de las bases de este lenguaje. Este repaso te ayudará luego a construir aplicaciones de servidor (backend), mucho más efectivas.

Además, todo aquel estudiante que no conoce en profundidad a JavaScript o que proviene de otro lenguaje de programación, puede entender más rápidamente los fundamentos y particularidades de JS.



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Principios del lenguaje JS

Luego de repasar los fundamentos básicos del lenguaje JS, continuamos adentrándonos en el universo del lenguaje de programación, para así estar a tono con el entorno de programación de Node JS, y poder sacar el mejor partido de este fabuloso lenguaje.



Principios del lenguaje JS


Algunas recomendaciones para implementar la mejor y más moderna experiencia en el uso de JavaScript:

- utiliza **arrow functions** para crear funciones
- simplifica las instrucciones lo más posible
- implementa el operador lógico **OR** junto a falsy
- olvida el uso de **var**, declara variables siempre con **let**
- todo lo que no deba cambiar, decláralo utilizando **const**
- segmenta el código en varios archivos, lo más que puedas
- mantén el orden de tu código
 - espaciado simple, solo para separar bloque de procesos
 - comentarios en el código de no más de una línea



Principios del lenguaje JS

```
código ordenado
```



```
const usuario = 'Joe McMillian';

if(usuario!==''){


  console.log('Bienvenid@' + usuario);
}else{

  console.log('No se reconoce el usuario');

}
```

Código poco legible y difícil de seguir
en su lógica planteada.

```
código ordenado
```



```
const usuario = 'Joe McMillian';

if (usuario !== '') {
  console.log(`Bienvenid@ ${usuario}`);
} else {
  console.warn('No se reconoce el usuario');
}
```

Ten la precaución de **ordenar**, **indentar**,
sumar **buenas prácticas**, y **separar bien**
la lógica de tu código desde el “*vamos!*”.

Principios del lenguaje JS

Si tienes dificultad para leer el código, recurre a CTRL+ y CTRL- para aplicar zoom sobre el IDE VS Code. También puedes acceder a la configuración del espacio de trabajo y cambiar la tipografía de fuentes y el tamaño predeterminado de estas.

Acondiciona tu entorno de la forma que más cómoda te resulte.

Array de elementos

Array de elementos

Los arrays de elementos son **estructuras que permiten agrupar una serie de datos en un único lugar**. En otros lenguajes de programación se puede llamar también *colecciones, arreglos, matrices o vectores*.

Los arrays son útiles para, por ejemplo, contener información que se requiere procesar o utilizar en un mismo lugar, toda junta, o por partes.



Array de elementos



Array de elementos

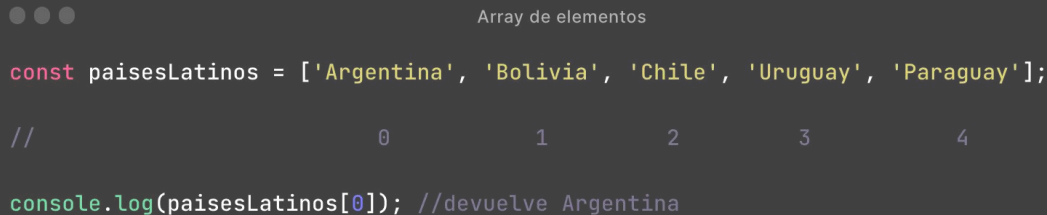
```
const paisesLatinos = ['Argentina', 'Bolivia', 'Chile', 'Uruguay', 'Paraguay'];
```

En JavaScript, los arrays pueden contener múltiples valores, y no uno solo. Aunque, aún con esta flexibilidad, se suele recomendar en base a buenas prácticas propias de otros lenguajes de programación, utilizar un mismo tipo de valor en los elementos contenidos.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Array de elementos



```
Array de elementos

const paisesLatinos = ['Argentina', 'Bolivia', 'Chile', 'Uruguay', 'Paraguay'];

//                                0           1           2           3           4

console.log(paisesLatinos[0]); //devuelve Argentina
```

Cada uno de los elementos del array, recibe un valor numérico denominado **índice** (*index, en inglés*). Esta posición permite leer fácilmente el contenido de dicho valor dentro del array utilizando simplemente el nombre del array, e indicando el índice dentro de los corchetes.

Array de elementos

```
Array de elementos

const paisesLatinos = ['Argentina', 'Bolivia', 'Chile', 'Uruguay', 'Paraguay'];

console.log(paisesLatinos.length); //devuelve el número 5
```

A través de la propiedad **.length**, el array nos devuelve el total de elementos que contiene. En el caso del ejemplo, devolverá el número **5** como valor.

Los arrays también cuentan con varios métodos para manipular su contenido de la mejor forma posible. Veamos a continuación algunos de los más importantes.

Array de elementos

Método	Descripción
<code>array.push('elemento')</code>	agrega un nuevo elemento en la última posición del array.
<code>array.unshift('elemento')</code>	agrega un nuevo elemento en la primera posición del array.
<code>array.pop()</code> - <code>array.shift()</code>	quita un elemento del array (el último o primero, respectivamente)
<code>array.slice(inicio, indice)</code>	crea un nuevo array copiando los elementos desde la posición inicio , hasta la posición indice indicada
<code>array.splice(inicio, totElem)</code>	corta los elementos del array desde la posición inicio , contando la cantidad indicada mediante la variable totElem
<code>array.indexOf('elemento')</code>	devuelve la posición numérica en el índice del array, donde se encuentra el elemento indicado. Si no existe, devuelve -1 como resultado
<code>array.includes('elemento')</code>	nos permite validar si un elemento existe o no dentro del array. Devuelve true o false , según el caso
<code>array.reverse()</code>	invierte los elementos del array. El último va a la primera posición, y así sucesivamente
<code>array.sort()</code>	ordena los elementos del array [A-Z] o [0-9], según corresponda
<code>array.concat(otroArray)</code>	concatena dos arrays en uno solo

Arrays de elementos

Existen otros tantos métodos para trabajar con array de elementos. Estos son también efectivos para interactuar con un array de objetos, el cual repasaremos más adelante.

Estos métodos están englobados bajo lo que se conoce como Funciones de Orden Superior, y poseen una complejidad media.

Ciclos de iteración

Ciclos de iteración

Como todo lenguaje de programación, JavaScript también cuenta con ciclos de iteración. Estos ciclos permiten repetir una tarea ***N*** cantidad de veces.

Esta repetición se realiza de forma controlada (*de principio a fin*), o de forma indefinida (*hasta que alguien indique cuándo terminar*).



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Ciclos de iteración

Las repeticiones controladas se dan de la mano del **ciclo for**. En este se estructura una serie de parámetros, que le indican cuándo comienza y cuándo termina.

Este tipo de iteración se la denomina: “*Ciclo por conteo*”.

Veamos un ejemplo a continuación:



Ciclos de iteración



Ciclos por conteo

```
const paisesLatinos = ['Argentina', 'Bolivia', 'Chile', 'Uruguay', 'Paraguay'];  
  
for (let i = 0; i < paisesLatinos.length; i++) {  
  console.log(paisesLatinos[i]);  
}
```

En este ejemplo, sabemos que el array dispone de 5 elementos.

En el ciclo **for**, le indicamos que recorra el array desde la primera posición, hasta la última (*indicada por la propiedad **.length***).

Así es como implementamos un “*ciclo por conteo*”.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Ciclos de iteración

La estructura de repetición (*iteración*) de forma indefinida, se da de la mano de **while**. En este otro caso, el ciclo while ejecutará una tarea determinada, nunca, o repetidas veces.

En este caso, **while** evalúa una expresión la cual debe dar como resultado un valor booleano del tipo **true**.

Allí, el **ciclo while** comienza a iterar, y seguirá haciéndolo hasta tanto ese valor booleano cambie a **false**.

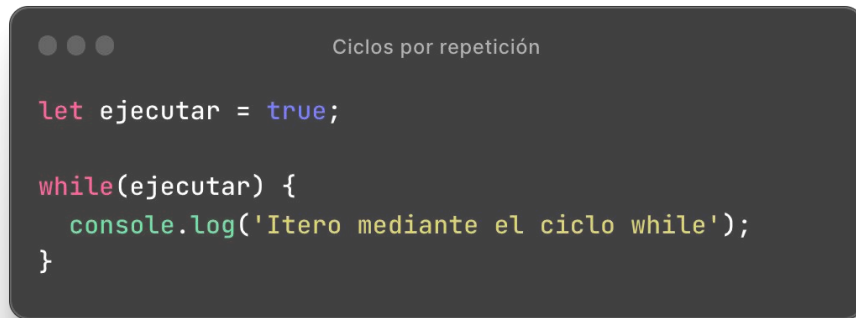


Ciclos de iteración

El **ciclo while** inicia su ejecución porque se cumple la condición de la variable ejecutar la cual, al evaluarla, devuelve un valor **true**.

Debemos tener cuidado con este ejemplo, porque no estamos controlando dentro del ciclo **while**, el cambio de valor de la variable.

El no cambiar su valor, lleva este ciclo a lo que se conoce como **loop infinito**.

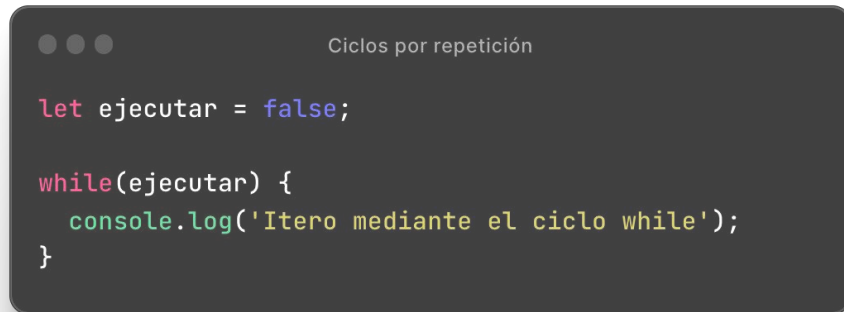


```
let ejecutar = true;

while(ejecutar) {
  console.log('Itero mediante el ciclo while');
}
```

Ciclos de iteración

En este otro caso, la variable **ejecutar** inicia su valor en **false**. Dada esta condición, el ciclo **while** no iterará en ningún momento, porque, al ejecutar dicha línea de código, la condición esperada no se está cumpliendo.



```
Ciclos por repetición

let ejecutar = false;

while(ejecutar) {
  console.log('Itero mediante el ciclo while');
}
```

Ciclos de iteración

La estructura de repetición (*iteración*) de forma indefinida, se da de la mano de **do - while**.

A diferencia del ciclo **while simple**, la expresión evaluada se hace al final del ciclo **do**, lo que conlleva a que este ciclo se ejecute al menos una vez.

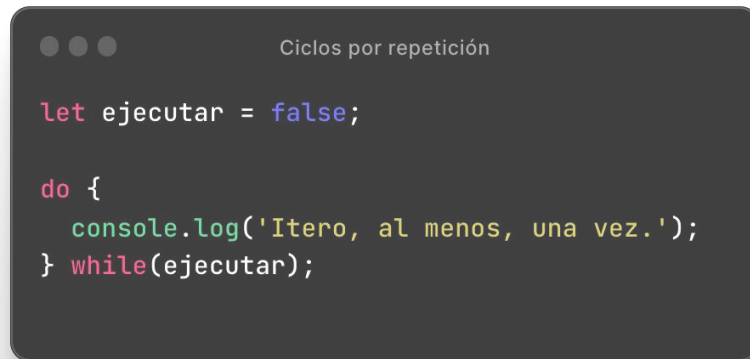


UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Ciclos de iteración

De igual forma, si la variable **ejecutar** se inicializa con el valor booleano **true**, debemos controlar dentro de la iteración, que en algún momento cambie. Sino caeremos también en un ***loop infinito***.



```
let ejecutar = false;

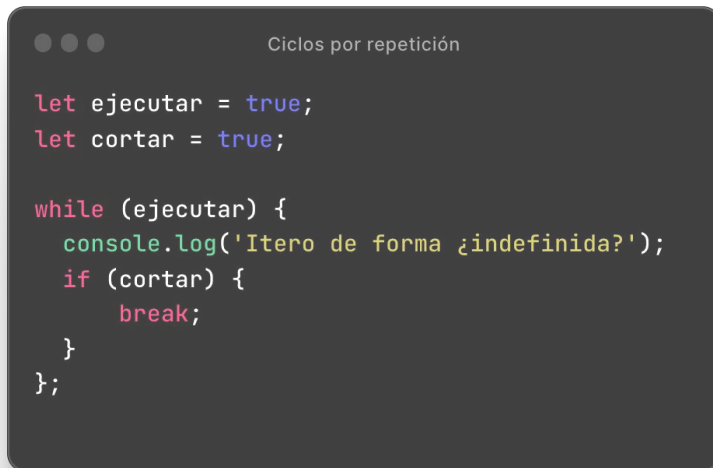
do {
  console.log('Itero, al menos, una vez.');
```

```
} while(ejecutar);
```

interrupción de los ciclos de iteración

Interrupción de los ciclos de iteración

La sentencia **break** es implementable, también, dentro de un ciclo de iteración, por conteo o por repetición. La misma nos permite interrumpir la iteración en base a una segunda expresión evaluada la cual devuelve true como resultado.



```
Ciclos por repetición

let ejecutar = true;
let cortar = true;

while (ejecutar) {
  console.log('Itero de forma ¿indefinida?');
  if (cortar) {
    break;
  }
};
```

Interrupción de los ciclos de iteración

La sentencia **break** también funciona dentro del ciclo for. En este caso, si el índice del array que se está iterando se encuentra en el valor **3**, entonces interrumpimos el ciclo.

```
Ciclos por repetición

let paisesLatinos = ['Argentina', 'Bolivia', 'Chile', 'Uruguay', 'Paraguay'];

for (let i = 0; i < paisesLatinos.length; i++) {
  console.log(paisesLatinos[i]);
  if (i === 3) {
    break;
  }
}
```

Interrupción de los ciclos de iteración

La sentencia **continue** hace lo propio dentro de un ciclo, pero en lugar de interrumpirlo, saltea una de las iteraciones. En este ejemplo, cuando el país que se está iterando coincide con la expresión evaluada, no lo imprime en consola. Lo saltea, pero continuando con la iteración del siguiente elemento del array.

```
Ciclos por repetición

let paisesLatinos = ['Argentina', 'Bolivia', 'Chile', 'Uruguay', 'Paraguay'];

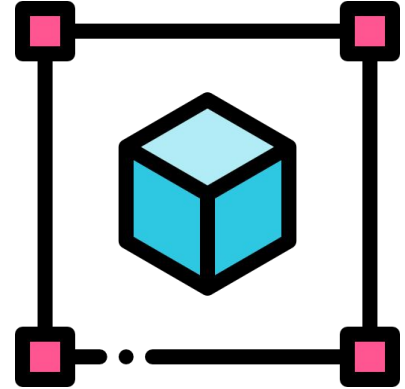
for (let i = 0; i < paisesLatinos.length; i++) {
  if (paisesLatinos[i] === 'Chile') {
    continue;
  }
  console.log(paisesLatinos[i]);
}
```

Objeto literal

Objeto literal

La evolución del paradigma de objetos en JavaScript es muy extensa. Todo se inició con el objeto literal, luego nacieron las funciones constructoras y, finalmente, las clases JS basadas en objetos.

Pero en esta ocasión nos concentraremos en la primera señal de objetos JavaScript: el **objeto literal**.

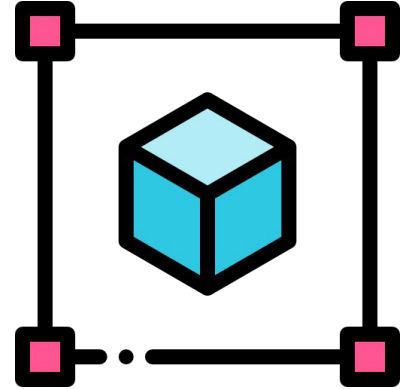


Objeto literal

La **estructura de un objeto literal** se compone de la misma estructura de objetos que conforman el lenguaje JS.

Estos poseen una serie de propiedades que conforman al objeto en sí y, eventualmente, pueden incluir uno o más métodos asociados para realizar determinadas tareas.

Veamos un ejemplo a continuación:



Objeto literal

Su estructura parte de un nombre definido con la primera inicial en mayúsculas. Luego, dentro de la notación `{ }`, definimos cada propiedad que le da vida seguida del valor predeterminado. Cada propiedad + valor, se separa de la siguiente, utilizando una coma.



Objeto literal

```
const Producto = {  
  codigo: 123,  
  nombre: 'Notebook 14 pulgadas FHD',  
  importe: 115000,00,  
  stock: 22  
}
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Objeto literal

El uso del objeto literal es directo. No requiere que lo instanciamos previo a su utilización.

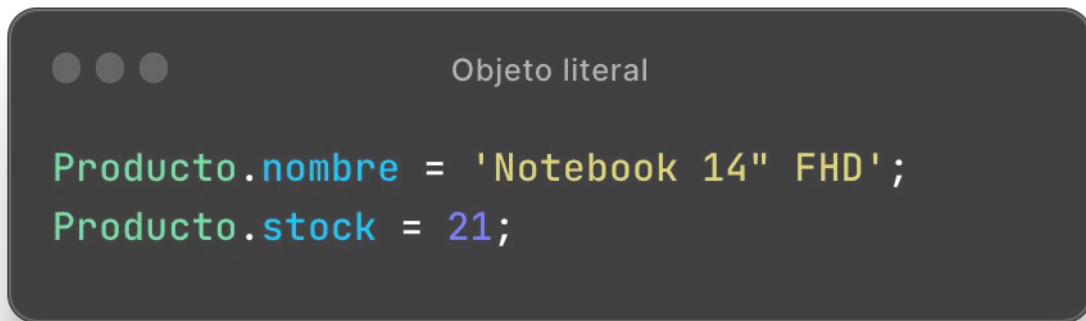


Objeto literal

```
console.log(Producto.nombre); //imprime 'Notebook 14 pulgadas FHD'  
console.log(Producto.stock); //imprime 22
```

Objeto literal

Para cambiar cualquier valor previamente asignado, solo describimos **Objeto.propiedad = valor**.



```
Objeto literal

Producto.nombre = 'Notebook 14" FHD';
Producto.stock = 21;
```

Objeto literal

```
Objeto literal

const IVA = 1.21

const Producto = {
  codigo: 123,
  nombre: 'Notebook 14 pulgadas FHD',
  importe: 115000,
  stock: 22,
  importeFinal() {
    return parseFloat(this.importe * IVA);
  }
}
```

En los objetos literales también podemos definir métodos. Estos se definen directamente, sin utilizar la palabra reservada **function**, dentro de la notación **{ }**, y utilizando la coma como separador.

Objeto literal

Y también se utilizan de forma directa,
como cualquier otro método de objeto JS.

A dark-themed terminal window with a title bar containing three window control buttons and the text 'Objeto literal'. The terminal displays two lines of code: 'Producto.importeFinal()' in green and '//' devuelve 139150 in light blue.

```
Objeto literal

Producto.importeFinal()
// devuelve 139150
```

Objeto literal

Los objetos literales son la forma más rápida de estructurar una Entidad dentro de un lenguaje de programación, sin tener que caer en tener que instanciar a dicho objeto, previo a su uso.

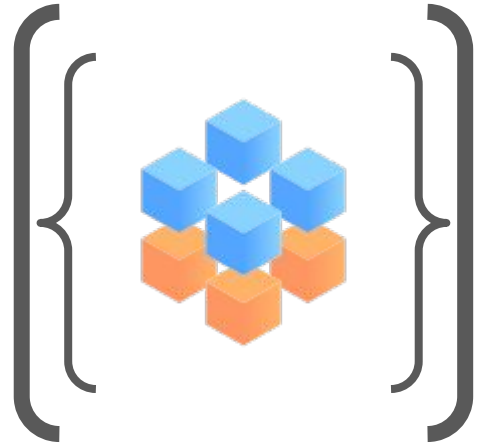
Un claro ejemplo de ello podría ser el objeto `Math()` de JS, de uso directo.

Array de objetos literales

Objeto literal

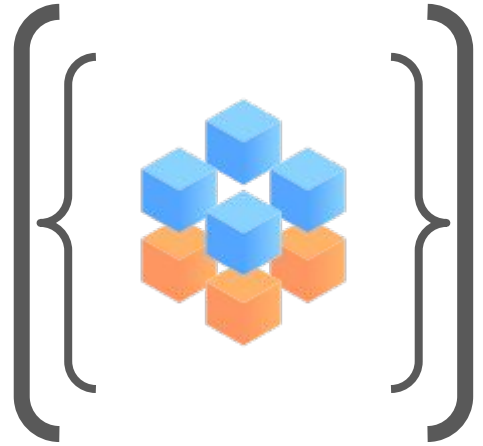
Un **array de objetos literales** combina en una misma ecuación, un objeto literal con todo el poder que este puede tener, y la estructura de arrays, que agrupan de forma efectiva un conjunto de elementos.

A través de los arrays de objetos literales, podremos manipular información estructurada de acuerdo a la necesidad de nuestros proyectos.



Objeto literal

La estructura de **array de objetos literales** es la que le dio vida a la **estructura de datos JSON**, actualmente utilizada para intercambiar información entre aplicaciones frontend y aplicaciones de backend.



Objeto literal

```
Objeto literal

const productos = [{id: 1, nombre: 'Notebook 14" FHD', importe: 115000},
                   {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000},
                   {id: 3, nombre: 'Macbook Air 13', importe: 745000},
                   {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000},]
```

Representación de una estructura basada en array de objetos. Como podemos ver, la misma responde a un orden similar a cuando estructuramos datos de un mismo tipo en una planilla de cálculo o base de datos.

Objeto literal

Los array de objetos literales no requieren que respetemos un orden de datos 100% estructurados. Como podemos ver en la representación anterior, la estructura puede variar entre uno y otro objeto literal. Si bien nos puede parecer algo extraño, esta flexibilidad es una característica natural en bases de datos del tipo **noSql**.



Objeto literal

```
const productos = [{id: 1, nombre: 'Notebook 14" FHD', importe: 115000, stock: 25},  
                   {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000, stock: 19},  
                   {id: 3, nombre: 'Macbook Air 13', importe: 745000, discontinuo: TRUE},  
                   {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000},]
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Objeto literal

Otra característica propia de los objetos literales que también se dispone dentro de un array de objetos, es la posibilidad de que un objeto literal contenga un array interno. Este puede ser un array de elementos, como también un array de objetos.

Podemos apreciar un ejemplo, en el caso de las manzanas y su stock.



Objeto literal

```
const productos = [{id: 1, nombre: 'Banana', importe: 450, stock: 25},  
                   {id: 2, nombre: 'Manzana', importe: 380, stock: [roja: 19, verde: 40]},  
                   {id: 3, nombre: 'Pera', importe: 440, stock: 70},  
                   {id: 4, nombre: 'Durazno', importe: 820},]
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Objeto literal

```
Objeto literal

const productos = [{id: 1, nombre: 'Banana', importe: 450, stock: 25},
                   {id: 2, nombre: 'Manzana', importe: 380, stock: 59},
                   {id: 3, nombre: 'Pera', importe: 440, stock: 70},
                   {id: 4, nombre: 'Durazno', importe: 820},]

for (let i = 0; i < productos.length; i++) {
  console.log(`${productos[i].nombre} - stock: ${productos[i]?.stock || 0}`);
}
```

Podemos pensar que tanta flexibilidad se vuelve una complicación a la hora de escribir código. La buena noticia es que, hoy, JS cuenta con las mejoras suficientes en el lenguaje como, por ejemplo, el **acceso condicional a un objeto** o **acceso condicional a una propiedad**, que nos ayudan a minimizar posibles errores en la lógica de nuestras aplicaciones.

Objeto literal

Objeto literal

```
const productos = [{id: 1, nombre: 'Banana', importe: 450, stock: 25},  
                   {id: 2, nombre: 'Manzana', importe: 380, stock: 59},  
                   {id: 3, nombre: 'Pera', importe: 440, stock: 70},  
                   {id: 4, nombre: 'Durazno', importe: 820},]  
  
for (let i = 0; i < productos.length; i++) {  
  console.log(`${productos[i].nombre} - stock: ${productos[i]?.stock || 0}`);  
}  
  
//IMPRIME:  
//  Banana - stock: 25  
//  Manzana - stock: 59  
//  Pera - stock: 70  
//  Durazno - stock: 0
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Objeto literal

Los array de objetos literales cuentan con la mayoría de los métodos de arrays de elementos, los cuales nos permiten llevar a cabo tareas de mantenimiento de los datos. Por ejemplo, el método **.push()** nos permite incorporar un nuevo objeto al array actual.

```
Objeto literal

const agregarProducto = () => {
  const nuevoProd = {id: 5, nombre: 'Ananá', importe: 930, stock: 16}
  productos.push(nuevoProd)
  console.clear()
  console.table(productos)
}
```

Objeto literal

Para una inserción efectiva, recurrimos a armar una estructura de objeto literal independiente, a la cual le definimos los valores de cada propiedad de forma dinámica, y luego nos queda pasarla como parámetro al método **push()**.

```
Objeto literal

const agregarProducto = (cod, nombreProd, importe, uni)=> {
  const nuevoProd = {id: cod, nombre: nombreProd, importe: importe, stock: uni}
  productos.push(nuevoProd)
  console.clear()
  console.table(productos)
}

agregarProducto(5, 'Ananá', 930, 16)
```


Objeto literal

```
app.js
app.js > ...
21 const agregarProducto = (cod, nombreProd, importe, unidades)=> {
22   const nuevoProd = {id: cod, nombre: nombreProd, importe: importe, stock: unidades}
23   productos.push(nuevoProd)
24   console.clear()
25   console.table(productos)
26 }
27 agregarProducto(5, 'Ananá', 930, 16)
28
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

(index)	id	nombre	importe	stock
0	1	'Banana'	450	25
1	2	'Manzana'	380	59
2	3	'Pera'	440	70
3	4	'Durazno'	820	
4	5	'Ananá'	930	16

Algunos métodos de array más avanzados, para buscar o filtrar contenido del array, verificar existencia, conocer su índice, y demás, los veremos más adelante bajo el concepto de **Funciones de Orden Superior**.

Espacio de trabajo

Espacio de trabajo

Pongamos en práctica este repaso general de la sintaxis básica y moderna de JavaScript, a través de un conjunto de ejercicios.

Tiempo estimado: 20 minutos. 

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

```
const questions = [ 'dudas', 'consultas', '🤔' ]
```



```
> node gracias.js
```