

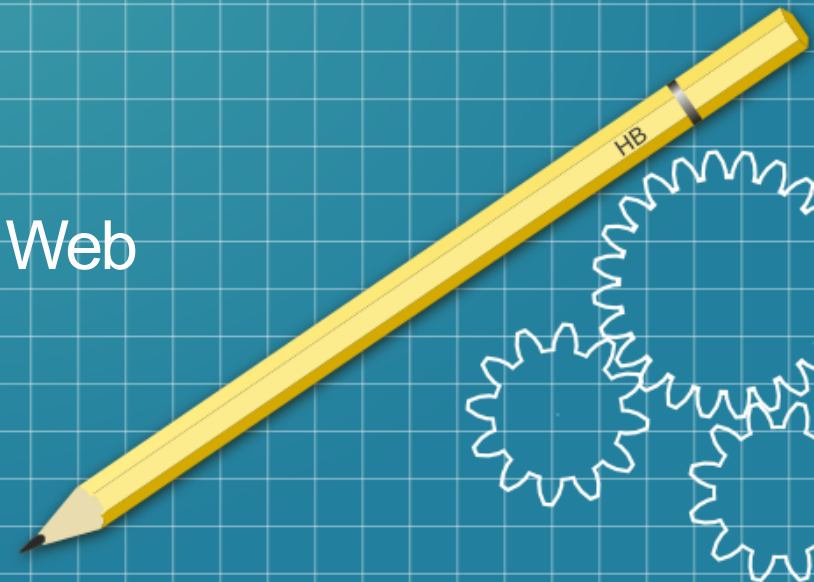


FPA FORMACIÓN
PROFESIONAL
ANDALUZA

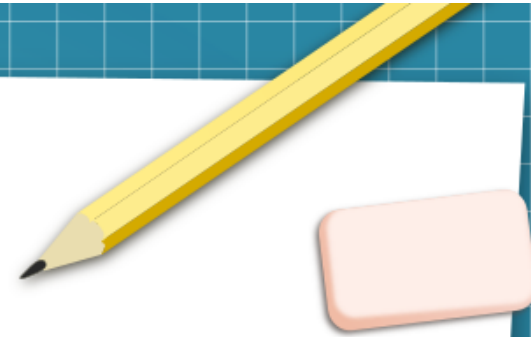
UD3.- ESTRUCTURAS DEFINIDAS POR EL USUARIO EN JAVASCRIPT.

2º curso – CFGS Desarrollo de Aplicaciones Web

Profesora: María Ojeda García



Objetivos de la unidad



- Poder crear **funciones** personalizadas para realizar tareas específicas que las funciones predefinidas no logran hacer.
- Comprender el objeto Array de JavaScript y familiarizarse con sus propiedades y métodos.
- Crear objetos personalizados diferentes a los objetos predefinidos del lenguaje.
- Definir propiedades y métodos de los objetos personalizados.
- Uso de módulos.
- Conocer y aplicar los patrones de diseño en JavaScript

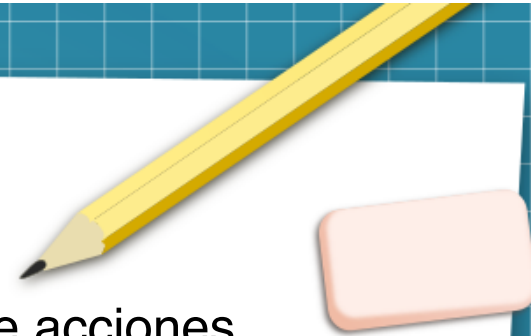
3.1. FUNCIONES

Las funciones son líneas de código que realizan una serie de acciones y vienen a resolver uno de los principales problemas de la programación estructurada (todo el programa en un único bloque de código).

El uso de funciones tiene una serie de ventajas:

- Permiten reutilizar código.
- Mejorar la eficiencia.
- Aumentar su legibilidad.
- Reducir los costes de mantenimiento de los programas.

Ejemplo: Usar una función que implemente la validación de un formulario y utilizarla en todos los formularios que se programen.



3.1. FUNCIONES

3.1.1 Formas de declarar una función (I)

Declaración de función (formal)

```
function nombreFuncion( [params]){  
    <cuerpo función>  
}
```

```
nombreFuncion(); //llamada a la función
```

Ejemplo// Crear una función que reste dos valores

```
function restar( ){  
    console.log(6-4);  
}
```

```
restar(); //llamada a la función
```



Permite
HOISTING

3.1. FUNCIONES

3.1.1 Formas de declarar una función (II)



No permite
HOISTING

Expresión de función anónima

```
const/let nombreFuncion = function ([params]){  
    <cuerpo función>  
}  
nombreFuncion(); //llamada a la función
```

Expresión de función con nombre

```
const/let nombreFuncion = function nombreInterno([params]){  
    <cuerpo función>  
}  
nombreFuncion(); //llamada a la función  
nombreInterno(); //is not defined
```

Ejemplo// Crear una función que reste dos valores

```
const restar = function( ){  
    console.log(6-4);  
}  
restar(); //llamada a la función
```

3.1. FUNCIONES

3.1.2 Parámetros y argumentos en funciones

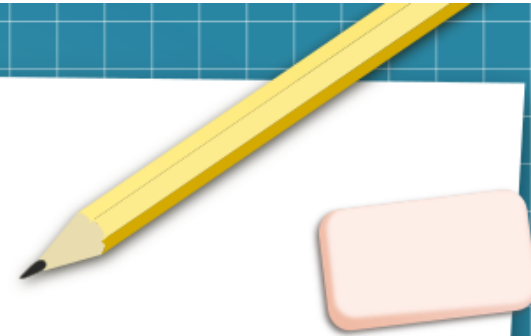
Los **parámetros** son la lista de variables que se indican al definir una función.

```
const/let nombre=function (a, b){  
    <cuerpo función>  
}
```

Los **argumentos** son los valores que se pasan a la función cuando se invoca.

```
const restar =function(a,b){  
    console.log(a-b);  
}
```

```
Restar(6,4);    //llamada a la función
```

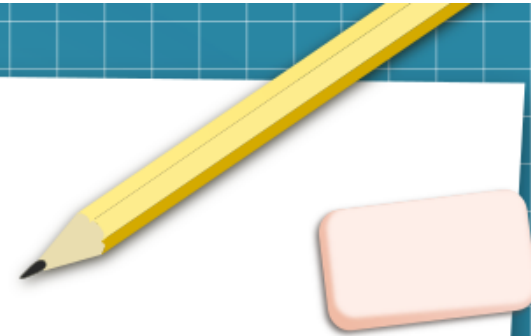


3.1. FUNCIONES

3.1.3 Parámetros por defecto

Los parámetros por defecto en la función permiten que los parámetros con nombre se inicien con valores predeterminados si no se pasa ningún valor o reciben *undefined*

```
function multiplicar (a, b=1){  
    console.log(a*b)  
}  
multiplicar(5, 6) //5*6=30  
multiplicar(6)   //6*1=6
```

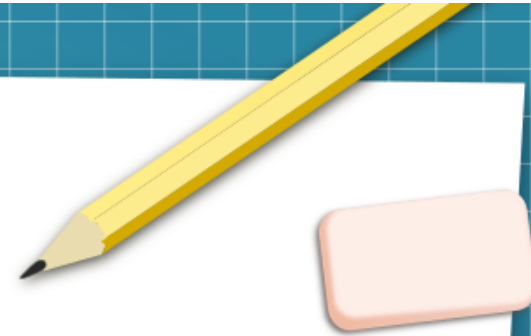


3.1. FUNCIONES

3.1.4 Return

La sentencia **return** finaliza la ejecución de la función y especifica un valor para ser devuelto a quien llama a la función. Su uso no es obligatorio.

```
const multiplicar=function(a,b) {  
    return (a*b)  
}  
  
const resultado=multiplicar(5,6);
```



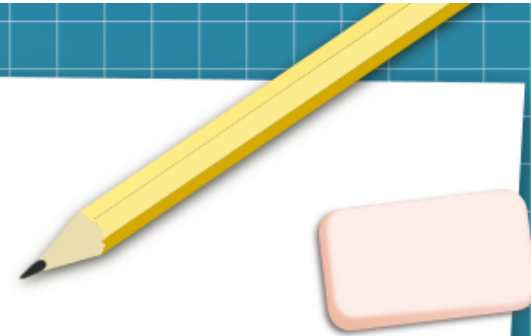
3.1. FUNCIONES

3.1.5 Ámbito (scope) de variables

Cómo ya se ha visto en la primera unidad, al hablar de ámbito de variables, se está haciendo referencia a aquellas partes del programa donde una variable o constante es “visible”, accesible.

Si una variable es accesible solo desde el interior de una función se dice que la variable es local a la función.

```
let mensaje="variable global";  
const mostrar=function() {  
    let mensaje="variable local";  
    console.log(mensaje); //variable local;  
}  
console.log(mensaje); // variable global;
```



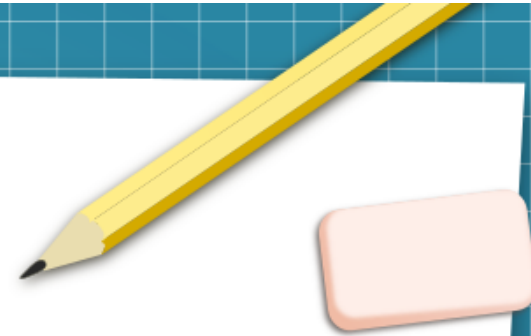
3.1. FUNCIONES

3.1.6 Funciones flecha o Arrow functions

Las funciones flechas son una alternativa abreviada a una función tradicional, pero es limitada y no se puede utilizar en todas las situaciones.

Limitaciones:

- No tiene sus propios enlaces a `this` o `super`.
- No es apta para los métodos `call`, `apply` y `bind` (métodos de función)
- No se puede utilizar como constructor.



3.1. FUNCIONES

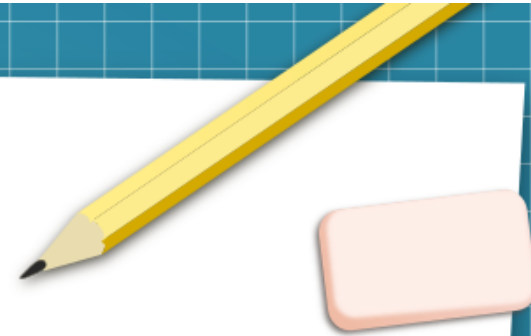
3.1.6 Funciones flecha o Arrow functions

Declaración de función **tradicional**:

```
const multiplicar=function(a,b) {  
    return (a*b)  
}
```

Declaración **Arrow function**: Elimina la palabra “function” y coloca la flecha entre el parámetro y la llave de apertura.

```
const multiplicar=(a,b)=>{  
    return (a*b)  
}
```



3.1. FUNCIONES

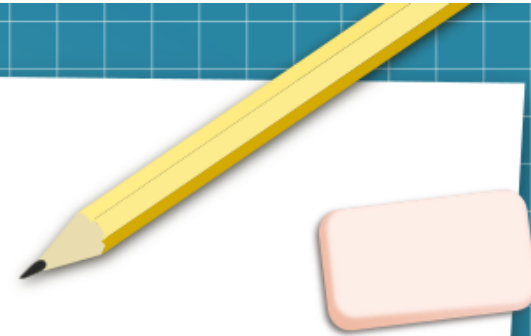
3.1.6 Funciones flecha o Arrow functions

Si tiene sólo una línea se pueden quitar las llaves y la palabra “return”

```
const multiplicar=(a,b)=> a*b;
```

Puede suprimir los paréntesis si solo hay un parámetro

```
const multiplicar=a=> a*5;
```



3.2. ARRAYS

3.2.1 Creación

Un array es una colección o agrupación de elementos en una misma variable, cada uno de ellos caracterizado por la posición que ocupa en el array. La posición cuenta desde 0 hasta longitud-1.

- Definición como una instancia del objeto Array.

```
const letras= new Array ("a", "b", "c"); //Array de 3 elementos
```

```
const letras =new Array(3) //Array vacío de tamaño 3
```

- Definición como array literal

```
const letras=["a", "b", "c"]; //Array de 3 elementos
```

```
const letras=[]; //Array vacío
```

```
const letras=["a", 5, true]; //Array mixto
```

Pueden ser mixtos

3.2. ARRAYS

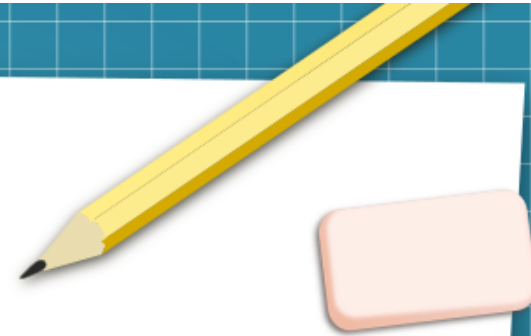
3.2.2 Acceso a elementos del array

- A través del índice:

```
const frutas = ["Manzana", "Banana", "Naranja", "Fresa"];  
console.log(frutas[1]); // "Banana"  
console.log(frutas[2]); // "Naranja"
```

- Método `at(índice)` → índice puede ser positivo (cuenta desde el principio del array) o negativo (cuenta desde el último elemento del array)

```
console.log(frutas.at(0)); // "Manzana"  
console.log(frutas.at(-1)); // "Fresa" (último elemento)
```



3.2. ARRAYS

3.2.3 Recorrido de un array (I)

- **For...in:** Variación del bucle for utilizada para recorrer arrays y propiedades enumerables de objetos.

```
for (const index in numeros) {  
  console.log(`${index}- ${numeros[index]}`);  
}
```

Opera sobre
índices

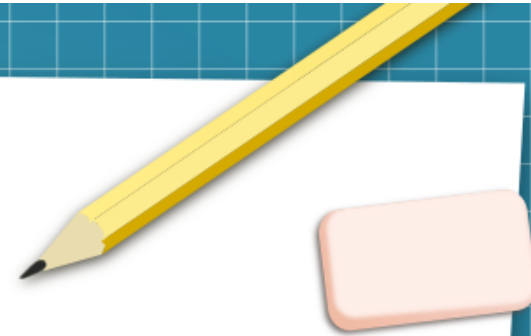
- **For...of:** Variación del bucle for para recorrer objetos iterables (arrays, cadenas, mapas, conjuntos...)

```
const numeros = [10, 20, 30];  
for (let numero of numeros) {  
  console.log(numero);  
}
```

Opera sobre
valores

3.2. ARRAYS

3.2.3 Recorrido de un array (II)



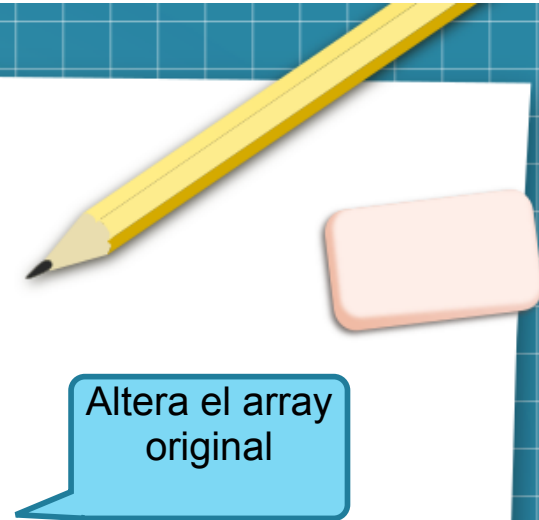
• **forEach:** Método de instancia del objeto Array. Ejecuta una función (callback) por cada elemento del array. La función callback recibe, al menos, 2 parámetros implícitos:

- **Elemento:** variable que representa el elemento del array en cada iteración.
- **Index:** índice del elemento accedido en cada iteración.

```
const numeros = [1, 2, 3, 4];  
numeros.forEach((num) => console.log(num * 2));
```


3.2. ARRAYS

3.2.4 Redimensionado de un array (I)



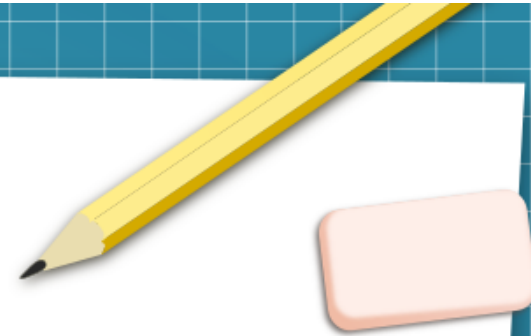
Formas de añadir/eliminar elementos al array:

En los extremos:

- Uso de **longitud** como índice: Añade un elemento al final
- Método **push(valor)** → Inserta el elemento valor al final del array.
- Método **unshift(valor1[,valor2,...,valorN])** → Inserta elementos delante del primer elemento del array.
- Método **pop()** → Extrae y devuelve el último elemento del array.
- Método **shift()** → Extrae y devuelve el primer elemento del array.

3.2. ARRAYS

3.2.4 Redimensionado de un array (II)



Formas de añadir/eliminar elementos al array:

En cualquier posición—→ **Método splice:**

- `splice(índice,N)`: Extrae y devuelve N elementos a partir del índice .
- `splice(índice,0,elemento1[,elemento2,...elementoN])`: Inserta los elementos a partir del índice.
- `splice(índice,N,elemento1[,elemento2,...elementoN])`: Elimina los N elementos a partir del índice e inserta en su lugar los elementos.
- `splice(índice)`: Extrae y devuelve todos los elementos desde el índice hasta el final.

3.2. ARRAYS

3.2.5 Hacer copias de un array

- **Método slice(índice_inicio, índice_fin):** Devuelve el array formado por los elementos desde el índice_inicio hasta el índice_fin (no incluido)
 - Si no se especifica índice_fin, devuelve el array desde índice inicio hasta el final.
 - Si no se especifican ninguno de los dos parámetros, se devuelve una copia del array.
- **Método concat(array):** concatena array al array que realiza la llamada al método.
- **Operador Spread[...]:** Devuelve una copia del array.

No altera el
array original

3.2. ARRAYS

3.2.6 Destructuring

Permite extraer elementos de arrays y objetos de forma posicional:

- Sintaxis Básica:

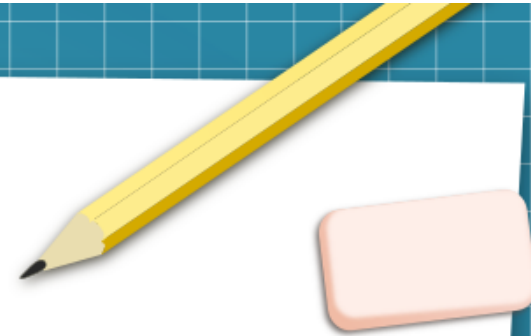
```
const [a, b, c, d] = [10, 20, 30, 40];
```

```
//a=10, b=20, c=30, d=40
```

- Saltar elementos:

```
const [,segundo,,cuarto] = [10, 20,30,40];
```

```
//segundo=20, cuarto=40
```

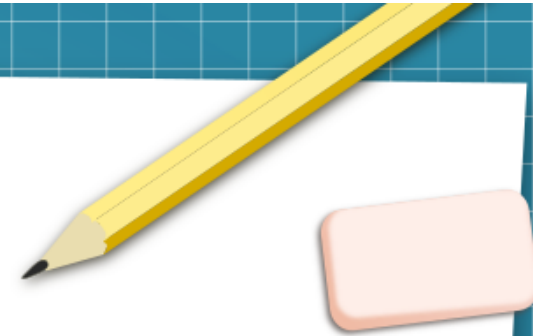


3.2. ARRAYS

3.2.7 Método map()

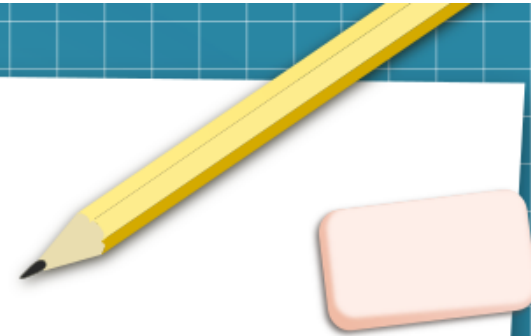
Crea un array aplicando una función a cada elemento del array original.

```
const numeros = [1, 2, 3, 4];  
// Multiplica cada número por 2  
const dobles = numeros.map(numero => numero * 2);  
console.log(dobles); // [2, 4, 6, 8]  
console.log(numeros); // [1, 2, 3, 4] (el array original no se modifica)
```



3.2. ARRAYS

3.2.8 Arrays bidimensionales

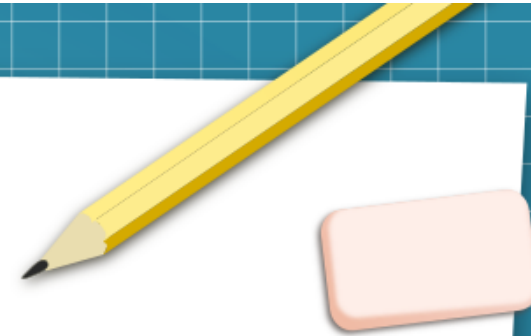


Cada elemento del array es un array.

- El acceso a un elemento requiere la especificación de dos índices.
- Su recorrido requiere la anidación de bucles.
- Como los arrays pueden ser heterogéneos, los elementos no tienen necesariamente que ser siempre otro array.

3.2. ARRAYS

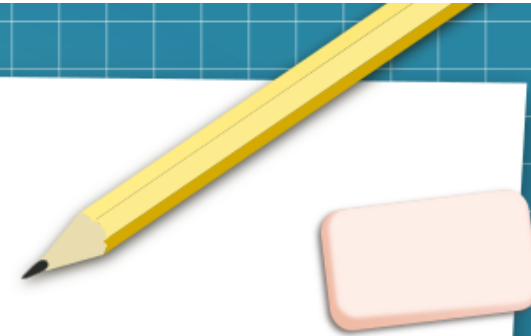
3.2.9 Métodos de Arrays (I)



Método	Descripción
indexOf(elemento)	Devuelve el índice de la primera posición donde se encuentra el elemento
lastIndexOf(elemento)	Devuelve el índice de la última posición donde se encuentra el elemento
includes(elemento)	Devuelve true si el elemento se encuentra en el array.
some(callback())	Devuelve true si algún elemento del array cumple la condición definida por la función de callback. Parámetros: elemento, índice y array.
findIndex(callback)	Devuelve el índice del primer elemento que cumple la condición definida por la función de callback. Parámetros: elemento, índice y array.
reduce(callback)	Reduce un array a un único valor aplicando una función a cada elemento del array utilizando un acumulador para mantener el resultado. Parámetros: acumulador, elemento, índice (opcional) y array (opcional).
filter(callback)	Devuelve un array formado por los elementos que cumplen el filtro especificado en la función de callback. Parámetros: elemento, índice y array.

3.2. ARRAYS

3.2.9 Métodos de Arrays (II)



Método	Descripción
find(callback)	Devuelve el primer elemento que cumple la condición definida por la función de callback. Parámetros: elemento, índice y array.
every(callback)	Devuelve true si todos los elementos del array cumplen la condición definida en la función de callback. Parámetros: elemento, índice y array.
sort([fncComparación])	Ordena los elementos del array alfabéticamente si no se le pasa una función. Para ordenar por otro criterio hay que definir la función de comparación con dos argumentos a y b (elementos del array) que devuelve: <ul style="list-style-type: none">- <0 si el orden debe ser a,b- 0 o NaN si el orden es equivalente.- >0 si el orden debe ser b,a.
reverse()	Invierte el orden de los elementos del array

3.3. CONJUNTOS

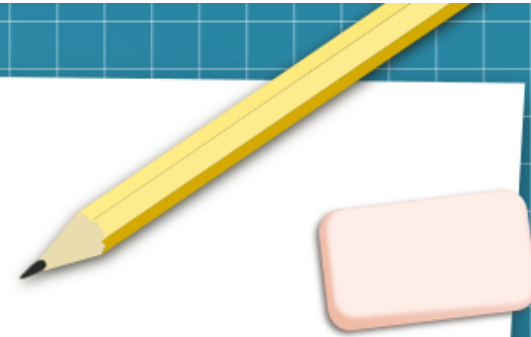
3.3.1 Creación de un conjunto

Los conjuntos o sets, son estructuras de datos muy parecidas a los arrays pero con la particularidad de que no permiten valores duplicados.

```
const conjunto = new Set();
```

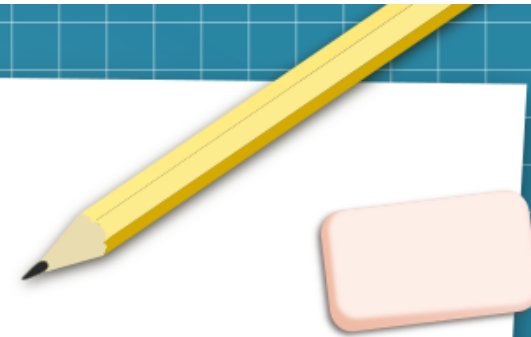
Para indicarle, desde su declaración, los elementos que lo componen inicialmente, es preciso pasarle un objeto de tipo iterable (array, map, string, set)

```
const set= new Set([1, "a", true]);
```



3.3. CONJUNTOS

3.3.2 Métodos de conjuntos



Método	Descripción
add(elemento)	Añade un elemento al conjunto. Si ya existe, no añade nada.
delete(elemento)	Elimina el elemento del conjunto
clear()	Vacía el conjunto.
forEach(callback)	Aplica la función a cada elemento del conjunto.
has(elemento)	Devuelve TRUE si encuentra el elemento dentro del conjunto.

3.4. OBJETOS LITERALES

3.4.1 Creación de objetos literales

Un objeto literal es un conjunto heterogéneo de información relacionada que se gestiona como un único elemento de datos.

Se definen como un conjunto de pares clave:valor (propiedades del objeto)

```
let objeto={  
    clave1:valor1,  
    clave2:valor2,  
    ...  
    claveN:valorN  
} ;
```

Clave: tiene que ser string.
Valor: cualquier tipo de datos.

3.4. OBJETOS LITERALES

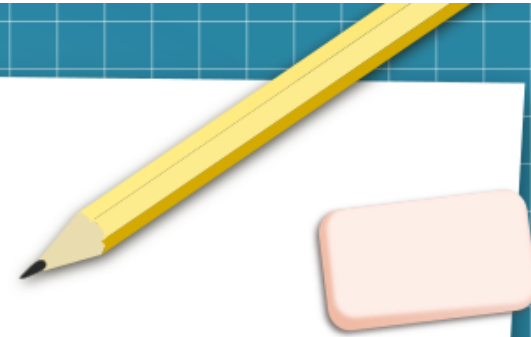
3.4.2 Acceso a propiedades

- Operador punto (.) : Objeto.propiedad;

```
let persona={  
    nombre:"Luis",  
    apellidos:"García López",  
    edad:25  
}  
console.log(persona.nombre);  
persona.edad=30;
```

- Corchetes[]: objeto["clave"];

```
console.log(persona["nombre"]);  
persona["edad"]=30;
```



3.4. OBJETOS LITERALES

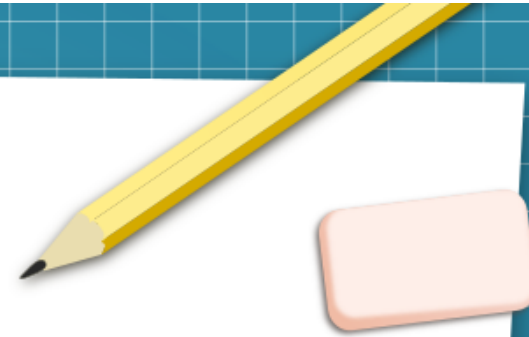
3.4.3 Agregar/Eliminar propiedades

- Se pueden agregar propiedades, simplemente haciendo referencia a ellas.

```
let persona={  
    nombre:"Luis",  
    apellidos:"García López",  
    edad:25  
}  
;  
persona.DNI="11111111A";
```

- Eliminar propiedades → delete objeto.propiedad

```
delete persona.DNI; //Elimina la propiedad DNI.
```

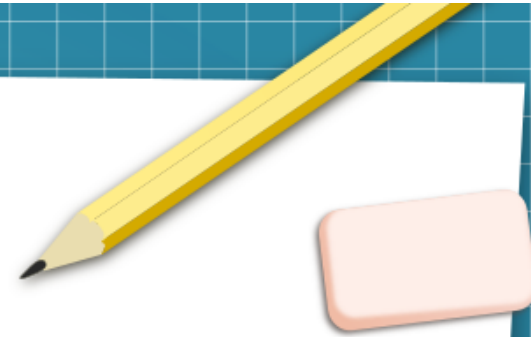


3.4. OBJETOS LITERALES

3.4.4 Destructuring

Podemos extraer valores de un objeto y asignarlos a variables utilizando destructuring. Para ello basta con hacer que las variables/constantes que reciben los valores tengan el mismo nombre que las propiedades del objeto.

```
let persona={
    nombre:"Luis",
    apellidos:"García López",
    edad:25
} ;
const { nombre , edad }=persona;
//nombre="Luis"
//edad=25
```



3.4. OBJETOS LITERALES

3.4.5 Protección de objetos

Método estático **freeze()** : Protege un objeto literal para que no se pueda modificar ni su estructura ni su contenido.

```
Object.freeze(persona);
```

Método estático **isFrozen()** devuelve true si el objeto está congelado.

```
Object.isFrozen(nombre_objeto);
```

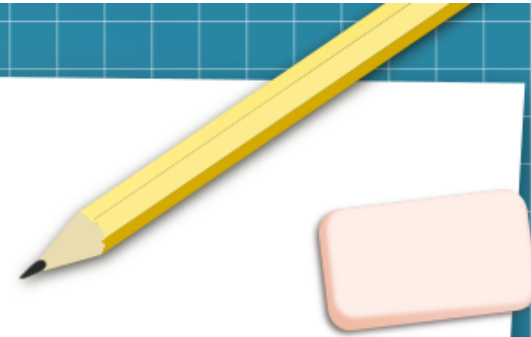
```
Object.isFrozen(persona);
```

Método **seal()**: Protege la estructura de un objeto literal pero deja el contenido abierto a cambios.

```
Object.seal(persona);
```

Método **isSealed()**: devuelve true si el objeto está protegido.

```
Object.isSealed(persona);
```



3.4. OBJETOS LITERALES

3.4.6 Comportamiento de objetos literales

Se puede establecer comportamiento para los objetos mediante la definición de métodos dentro del objeto.

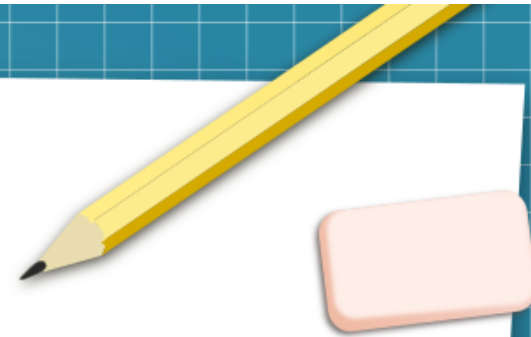
Para ello se define una propiedad cuya definición es una función.

```
let persona={  
  nombre:"Luis",  
  apellidos:"García López",  
  edad:25  
  cumplirannos:()=>this.edad+=1;  
};
```

Para hacer referencia a una propiedad del objeto desde un método es necesario utilizar **this**

3.4. OBJETOS LITERALES

3.4.7 Métodos de objetos (estáticos)



Método	Descripción
<code>assign(objeto1, objeto2,...,objetoN)</code>	Agrupar las propiedades de los objetos que se pasan como parámetro (eliminando las duplicadas) en un único objeto (objeto1) además de devolver un nuevo objeto.
<code>keys(objeto)</code>	Devuelve un array con el nombre de todas las propiedades del objeto (claves)
<code>values(objeto)</code>	Devuelve un array con los valores de las propiedades del objeto (valores)
<code>entries(objeto)</code>	Devuelve un array bidimensional con los pares clave-valor.

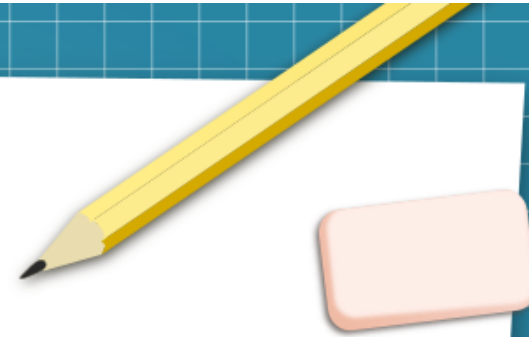
3.4. OBJETOS LITERALES

3.4.8 Anidamiento de objetos literales

Se puede definir una propiedad como un nuevo objeto.

En ese caso, el operador `.` nos da acceso a los niveles de profundidad del anidamiento.

```
let persona={
  nombre:"Luis",
  apellidos:"García López",
  edad:25
  direccion:{
    calle:"Calle Mayor",
    numero:4
  }
  cumplirannos:()=>this.edad+=1;
} ;
console.log(persona.direccion.calle);
```



3.4. OBJETOS LITERALES

3.4.9 Función de constructor

Un constructor de objetos permite definir una plantilla para crear múltiples objetos con propiedades y métodos específicos (NO ES UNA CLASE). Es la forma nativa de JS de implementar la POO. Sigue usándose actualmente.

Una función de constructor constructor es una función para crear objetos con propiedades.

```
function Persona(nombre, edad) {  
    this.nombre = nombre; // propiedad 'nombre'  
    this.edad = edad;      // propiedad 'edad'  
}  
  
const personal=new Persona("Juan",21);  
Const  persona2=new Persona("Pepe",30);
```

El nombre de la función debe empezar en mayúscula (convención)

El uso de new crea un nuevo objeto vacío y lo asigna a this.

Todos los objetos creados heredan de Object con todas sus propiedades y métodos de instancia.

3.4. OBJETOS LITERALES

3.4.10 Añadir propiedades al prototipo

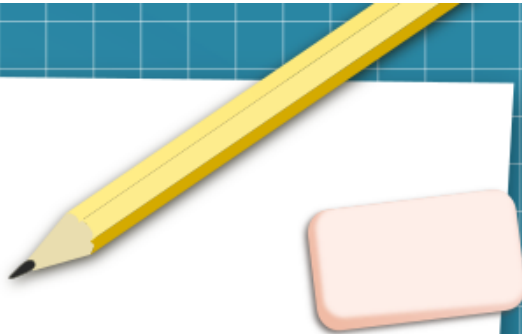
Crear métodos en un objeto:

- En la definición de la función de constructor:

```
function Persona(nombre, edad) {  
    this.nombre = nombre; // propiedad 'nombre'  
    this.edad = edad;      // propiedad 'edad'  
}
```

- En el prototipo:

```
Persona.prototype.saludar=function(){  
    console.log(`Hola, me llamo ${this.nombre}`)  
};  
  
const personal=new Persona("Juan",21);  
const persona2=new Persona("Pepe",30);  
personal.saludar();
```



Sería como añadirlo arriba en la definición del objeto

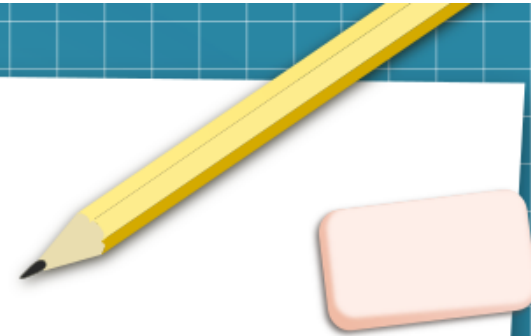
3.5. POO EN JAVASCRIPT

3.5.1 Introducción

La POO formalmente, necesita que el modelo de programación esté basado en clases, pero JavaScript no es un lenguaje basado en clases, sino basado en prototipos.

Con **ECMA6** (2015) se introduce una aproximación sintáctica para la creación de objetos más próxima a como lo hacen otros lenguajes incorporando una “sintáctica” a las funciones constructoras.

Útil para programadores que migran a JS desde otros lenguajes que sí son verdaderamente orientados a POO. Para ello se introduce la sintaxis de CLASES.



3.5. POO EN JAVASCRIPT

3.5.2 Definición de clases

Las clases se componen de:

- Propiedades:
- Definen las características.
- Métodos:
- Definen el comportamiento.
- Existe un método especial que es el **CONSTRUCTOR** para crear objetos de la clase.

Sintaxis general:

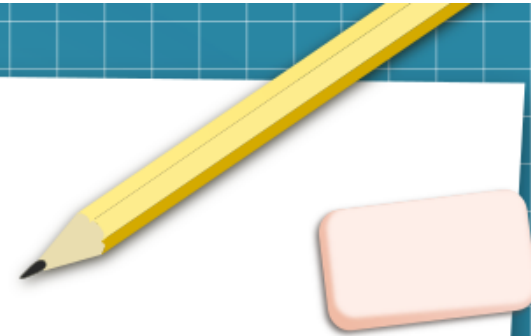
```
class nombre_clase{
  constructor(valor1,valor2...,valorN) {
    this.propiedad1=valor1;
    this.propiedad2=valor2;
    ...
    this.propiedadN=valorN
  }
  método() {
    cuerpo del método
  }
}
```

3.5. POO EN JAVASCRIPT

3.5.3 Propiedades (I)

Las propiedades se crean e inicializan en el constructor (modo tradicional) aunque a partir de ECMA 2022 se pueden definir fuera del constructor también.

```
Class Animal{  
  constructor (esp,raz,or){  
    this.especie=esp;  
    this.raza=raz;  
    this.origen=or;  
  }  
}
```



3.5. POO EN JAVASCRIPT

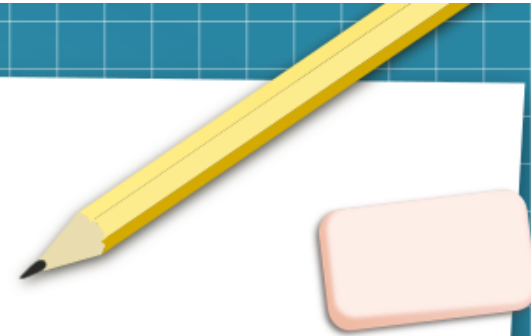
3.5.4 Elementos estáticos

Los elementos estáticos **se utilizan a través de la propia clase** (no se pueden utilizar a través de una instancia)

Se definen con *static*.

Válido tanto para propiedades como para métodos.

Se usan comúnmente para funciones auxiliares o valores que son comunes a todas las instancias de la clase.



3.5. POO EN JAVASCRIPT

3.5.5 Propiedades (II)

Las propiedades son públicas por defecto. Si queremos que una propiedad sea privada hay que definirla con # fuera del constructor:

```
Class Animal{  
  #especie;  
  #raza;  
  #origen;  
  constructor (esp,raz,or){  
    this.#especie=esp;  
    this.#raza=raz;  
    this.#origen=or;  
  }  
}
```

Si definimos propiedades como privadas hay que definir métodos públicos para su manipulación (getters y setters)

3.5. POO EN JAVASCRIPT

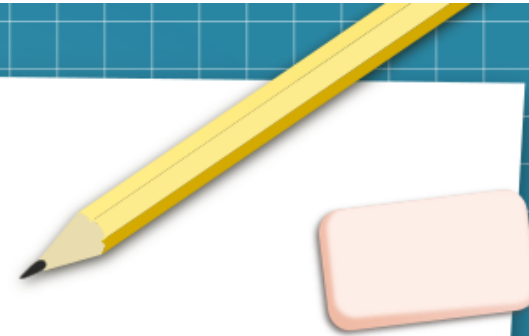
3.5.6 Métodos

Los métodos definen el comportamiento de la clase y se utilizan generalmente para manipular las propiedades.

Métodos especiales:

- **Constructor:** Permite instanciar objetos de una clase.
- **Getters y Setters:** Permiten acceder a propiedades privadas aunque podemos definirlos también para propiedades públicas (no hay restricción)

```
class Animal{  
    #especie;  
    #raza;  
    #origen;  
    constructor (esp,raz,or){  
        this.#especie=esp;  
        this.#raza=raz;  
        this.#origen=or;  
    }  
    get especie(){  
        return this.#especie;  
    }  
    set especie(valor){  
        this.#especie=valor;  
    }  
}
```



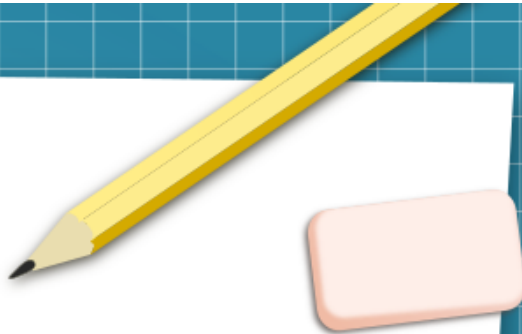
3.5. POO EN JAVASCRIPT

3.5.7 Herencia

Utiliza la palabra reservada *extends* .

Se utiliza *super()* para llamar al constructor de la clase padre.

Las propiedades privadas no pueden ser accedidas desde la clase hija. Para acceder a ellas es necesaria la definición de métodos públicos que las manipulen, lo cuales sí que pueden ser heredados.



```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
}

class Estudiante extends Persona {
  constructor(nombre, edad, carrera) {
    super(nombre, edad);
    this.carrera = carrera;
  }
}

const estudiante = new Estudiante("Ana",
22, "Medicina");
```

3.6. MÓDULOS

3.6.1 Introducción

A partir de ES2015 se introduce una característica nativa denominada **Módulos ES** que permita la importación y exportación de fragmentos de datos entre diferentes ficheros JavaScript, eliminando las desventajas que había hasta ahora y permitiendo trabajar de forma más flexible en el script.

```
// exportar un array
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep',
  'Oct', 'Nov', 'Dec'];

// exportar una constante
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// exportar una clase
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

3.7. PATRONES DE DISEÑO

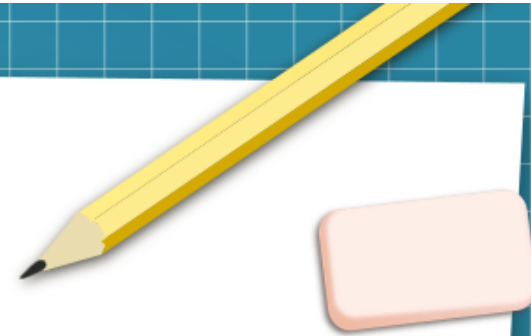
3.7.1 Introducción

Los patrones de diseño son soluciones probadas a problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación.

Ante un problema que nos podamos encontrar en el desarrollo de nuestra aplicación, si se puede solucionar usando un patrón, en lugar de reinventar la rueda.

Algunas ventajas de aplicar patrones de diseño:

- Simplificar los problemas.
- Estructurar mejor el código.
- Hacer el código más predecible



Desarrollo Web en el Entorno Cliente



This work is licensed under a Creative Commons
Attribution-ShareAlike 3.0 Unported License.
It makes use of the works of Mateus Machado Luna.

