

JavaScript Avanzado

Objetos en JS



Agenda de hoy

- Fundamentos de objetos
- objetos literales
 - propiedades - métodos
- funciones constructoras
 - .this
- clases JS
 - propiedades privadas vs públicas
 - getter y setter
 - instanciar objetos



fundamentos de objetos



fundamentos de objetos

En el mundo de la programación existe, desde más de treinta años, una rama denominada **Programación orientada a objetos** (POO).

A través de esta, se busca desarrollar un programa basado en el paradigma de objetos.

Los objetos son, básicamente, entidades que representan conceptos reales o abstractos en un formato digital.



fundamentos de objetos

Cada objeto tiene un **estado** y un **comportamiento**; el estado es definido por sus atributos mientras que, el comportamiento, se define por sus métodos.

Los objetos se relacionan entre sí a través de sus mensajes y sus métodos y, su uso en el desarrollo de software, permite clarificar de forma efectiva, aquellas tareas que definimos en cada línea de código que le da vida a nuestra aplicación.



fundamentos de objetos

entidad real

Podemos crear un objeto del tipo **Persona**. Este poseerá una serie de características que lo describa como tal, y que hasta puedan distinguirlo de otros objetos/entidades similares. Por ejemplo:

- nombre
- género
- edad
- profesión
- cabello
- tono de voz
- vestimenta



fundamentos de objetos

entidad abstracta

Aquí podemos pensar en, por ejemplo, una computadora. Si bien es algo físico, suele tratarse como una entidad abstracta por sus limitaciones o nicho específico. Sus características pueden ser:

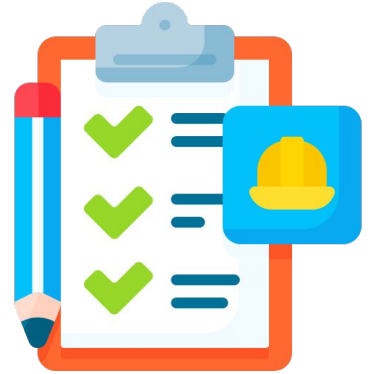
- marca
- modelo
- posee display
- posee teclado
- posee mouse pad o puntero
- sistema operativo
- batería



fundamentos de objetos

Cada característica (o *atributo*) que representa a estos objetos dentro del ecosistema de software, suele llamarse **propiedad**.

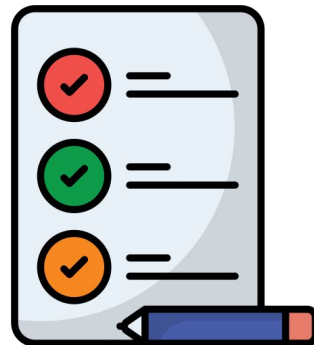
Cada comportamiento de estos objetos, es representado por un **método**.



fundamentos de objetos

Las propiedades que conforman un objeto, equivalen a lo que hasta ahora vinimos trabajando como variables o constantes. Mientras que, los métodos, son una referencia directa a lo que hasta ahora trabajamos como funciones.

La única diferencia de estas propiedades y métodos, es que siempre están directamente relacionados al objeto en el cual se crearon, y no pueden utilizarse de forma aislada a este.





fundamentos de objetos

Si bien la teoría de los objetos suele no ser tan clara para expresar la importancia de estos en el mundo de la programación, cuando comenzamos a plasmar la misma en herramientas integradas al lenguaje de programación con el cual trabajamos, nos baja cada uno de los conceptos a un plano mucho más comprensible.

Veamos a continuación cómo plasmar estos ejemplos.



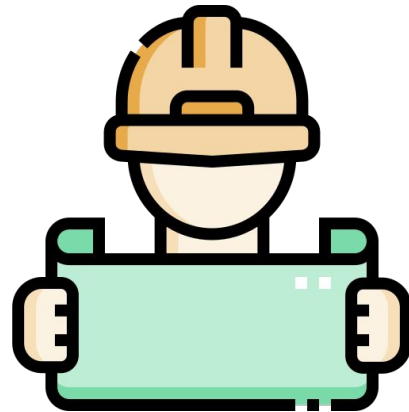
objeto literal



objeto literal

Para entender cómo trabajar con objetos en JS, arrancaremos viendo un ejemplo de objeto literal. Para ello, crearemos un ejemplo de objeto del tipo **Persona**.

Utilizaremos en este algunos atributos, o propiedades, para definirlo, y luego integraremos en él algunos métodos.



objeto literal

Aquí tenemos una representación rápida del objeto **Persona**, con algunos atributos (*propiedades*), que lo describen.

Cuando definimos un objeto literal, éste se debe crear a través de la palabra reservada **const**. El nombre del objeto literal suele definirse con su primera inicial en mayúsculas.



```
const Persona = {  
  nombreCompleto: '',  
  edad: 0,  
  profesion: '',  
  genero: ''  
}
```

objeto literal

En este ejemplo, dejamos el objeto literal sin ninguna propiedad configurada. Podremos definir sus valores en cualquier momento.

Cada propiedad, podrá almacenar cualquier tipo de datos de los que ya conocemos que posee **JavaScript**.



```
const Persona = {  
  nombreCompleto: '',  
  edad: 0,  
  profesion: '',  
  genero: ''  
}
```

objeto literal

Además de los datos del tipo **String** y **Number**, podemos agregar a sus propiedades, otros valores como ser un array de elementos, un objeto interno, un valor booleano, entre otros.



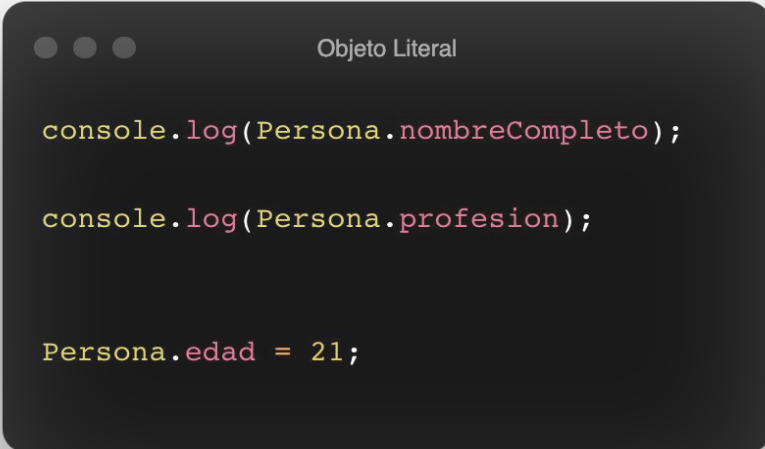
Objeto Literal

```
const Persona = {  
  nombreCompleto: 'Fer Moon',  
  edad: 47,  
  profesion: 'Profesor',  
  genero: 'Masculino'  
}
```

objeto literal

Para leer el valor de alguna de sus propiedades, simplemente escribimos **Objeto.propiedad**. Así obtendremos el valor almacenado.

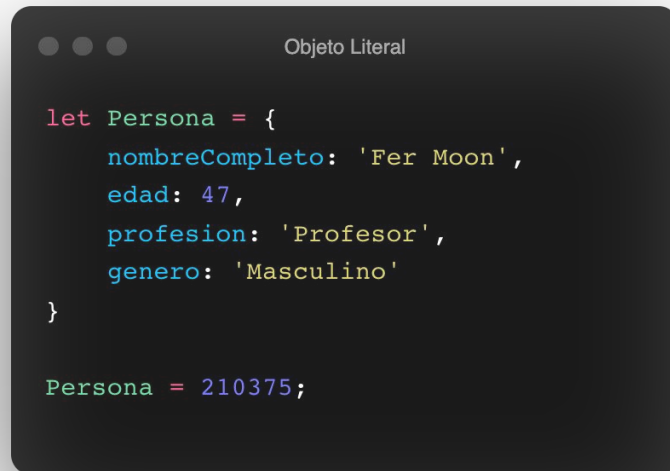
Para cambiarlo, usamos esta misma estructura sumando el operador de asignación igual, más el nuevo valor.



```
console.log(Persona.nombreCompleto);  
  
console.log(Persona.profesion);  
  
Persona.edad = 21;
```


objeto literal

Para definir un objeto literal, podemos utilizar **let**. El único problema que tenemos con esta palabra reservada es que, si el objeto literal es definido de forma global, quedamos expuestos a sobreescribirlo muy fácilmente.



```
Objeto Literal

let Persona = {
  nombreCompleto: 'Fer Moon',
  edad: 47,
  profesion: 'Profesor',
  genero: 'Masculino'
}

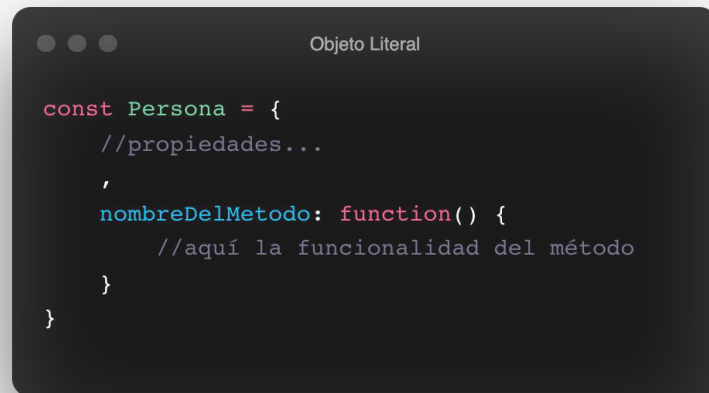
Persona = 210375;
```

métodos

métodos

Como hablamos anteriormente, los métodos nos permiten definir un comportamiento del objeto en cuestión.

Su estructura base, parte de una función anónima, pudiendo definir en su nombre, la acción que describa mejor a este.

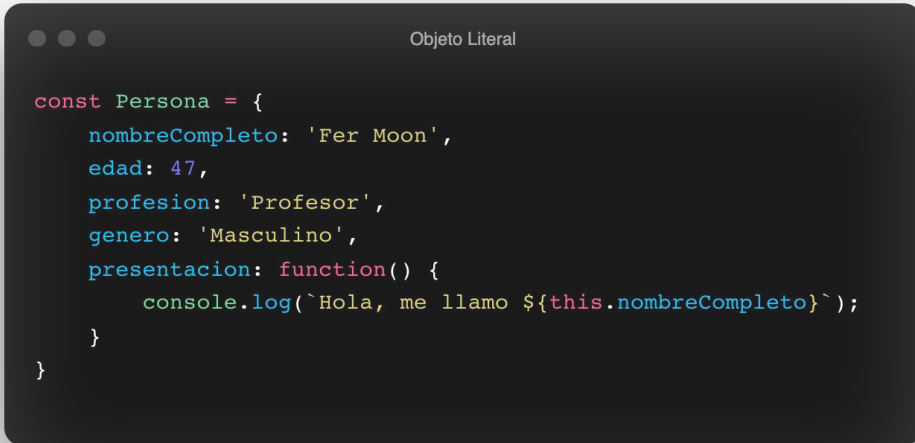


```
const Persona = {  
  //propiedades...  
  ,  
  nombreDelMetodo: function() {  
    //aquí la funcionalidad del método  
  }  
}
```

métodos

El método ejecuta una acción u operación predeterminada. En esta puede estar o no involucrado el uso de uno o más atributos (*propiedades*) del objeto.

Usualmente, los métodos utilizan un retorno implícito del resultado de la operación o tarea que resuelven.



```
const Persona = {  
  nombreCompleto: 'Fer Moon',  
  edad: 47,  
  profesion: 'Profesor',  
  genero: 'Masculino',  
  presentacion: function() {  
    console.log(`Hola, me llamo ${this.nombreCompleto}`);  
  }  
}
```

métodos

```
Objeto Literal

const Persona = {
  nombreCompleto: 'Fer Moon',
  edad: 47,
  //otras propiedades...
  ,
  presentacion: function() {
    console.log(`Hola, me llamo ${this.nombreCompleto}`);
  }
  miFechaDeNacimiento: function() {
    return Math.getFullYear() - this.edad
  }
}
```

Ejemplo de un método con
retorno implícito del
resultado de su operación.

métodos

Métodos

```
Persona.presentacion()  
//retorna 'Hola, me llamo Fer Moon'  
  
Persona.miFechaDeNacimiento()  
//retorna 47 como resultado
```

De igual forma que con las propiedades, la invocación de un método se realiza a partir del objeto literal que lo contiene.



 **ejemplo práctico** 



ejemplo práctico



Veamos, a continuación, un ejemplo práctico aplicando la construcción de un objeto literal aplicado a una entidad abstracta.

Para ello, crearemos un objeto **Producto**, con una serie de propiedades y métodos específicos, los cuales nos permitirán sacar el máximo provecho de este, dentro de una aplicación web.

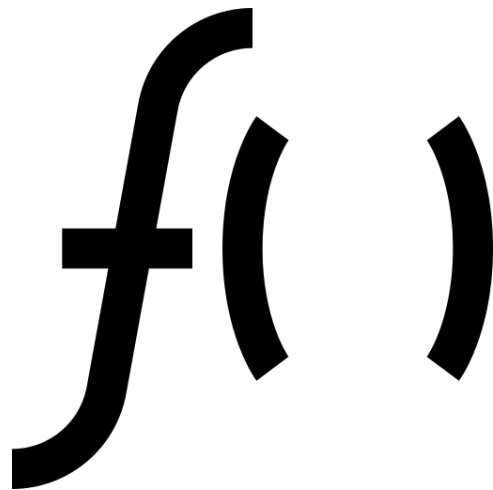
funciones constructoras

funciones constructoras

Una función constructora es una función especial de JavaScript que se utiliza para crear y inicializar objetos.

Algunas de sus características más importantes son:

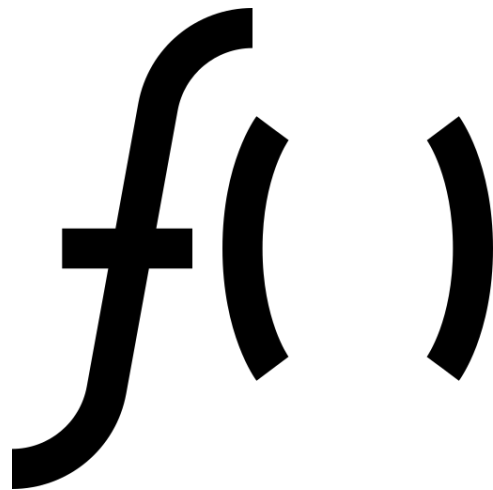
- ofician como una '*plantilla*' para crear objetos similares
- poseen un **constructor**, el cual le permite recibir parámetros
- **se deben instanciar** con otro nombre para utilizarse
- se invocan con la palabra clave **new**
- crean un nuevo objeto vacío
- sus propiedades públicas se definen anteponiendo **.this**



funciones constructoras

Su principal diferencia con los objetos literales, es que debemos instanciarla. Por ello es que una función constructora oficia como una plantilla.

Ninguna de sus propiedades suele estar configurada con valores estáticos, salvo raras excepciones. El constructor que integra, se utiliza para permitirle recibir parámetros en el momento en la cual se la instancia bajo un nuevo objeto.



funciones constructoras

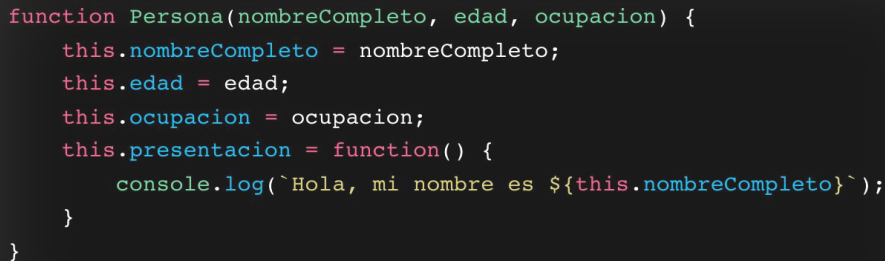
Ejemplo de una función constructora basada en el objeto literal **Persona**.

```
Función constructora

function Persona(nombreCompleto, edad, ocupacion) {
  this.nombreCompleto = nombreCompleto;
  this.edad = edad;
  this.ocupacion = ocupacion;
  this.presentacion = function() {
    console.log(`Hola, mi nombre es ${this.nombreCompleto}`);
  }
}
```

funciones constructoras

De igual forma que lo que sucede con objetos creados en otros lenguajes de programación más estrictos, se recomienda siempre que, cada función constructora, sea creada en un archivo **.JS** dedicado y que el código de dicha función constructora sea lo que este archivo contenga en su interior.



```
function Persona(nombreCompleto, edad, ocupacion) {  
  this.nombreCompleto = nombreCompleto;  
  this.edad = edad;  
  this.ocupacion = ocupacion;  
  this.presentacion = function() {  
    console.log(`Hola, mi nombre es ${this.nombreCompleto}`);  
  }  
}
```

funciones constructoras

Para utilizar la función constructora, debemos instanciarla utilizando la palabra reservada **new**. En el siguiente ejemplo, vemos cómo utilizar una única función constructora, para instanciar varios tipos de roles **Persona**.

Cada rol tiene sus propias características, que lo diferencian del resto.

```
Función constructora

const Profesor = new Persona('Fer Moon', 47, 'Profesor');

const Reguladora = new Persona('Laura Grisel', 50, 'Auditora de Regulaciones');

const Mecanico = new Persona('Nico Moon', 26, 'Técnico mecánico automotor');
```



this



this

La palabra reservada **this**, se antepone a cada una de las propiedades integradas en la función constructora, como también antes del nombre de cada método que esta función integre.

La misma, oficia como sinónimo de la propiedad, al momento de instanciar la función constructora.



this

De esta forma, podremos acceder a consultar el valor de cada una de las propiedades, por cada uno de los objetos instanciados a partir de esta función constructora.

```
Función constructora

const Profesor = new Persona('Fer Moon', 47, 'Profesor');
const Reguladora = new Persona('Laura Grisel', 50, 'Auditora de Regulaciones');
const Mecanico = new Persona('Nico Moon', 26, 'Técnico mecánico automotor');

Profesor.nombreCompleto; //devuelve 'Fer Moon'
Reguladora.edad;         //devuelve '50'
Mecanico.profesion;      //devuelve 'Técnico mecánico automotor'
```



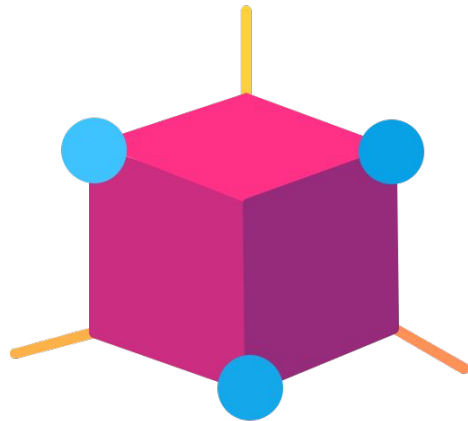
classes JS



Clases JS

Las clases JS son básicamente una función constructora, pero evolucionada hacia una sintaxis más parecida a cómo se definen las clases en otros lenguajes de programación.

Esta evolución llegó a JS a partir de la versión **EcmaScript 6**. Si bien su estructura para definirla cambia respecto a las funciones constructoras, técnicamente son lo mismo.



Clases JS

```
class Producto {
  constructor(nombre, importe, categoria) {
    this.nombre = nombre;
    this.importe = importe;
    this.categoria = categoria;
    this.stock = 0;
  }
  importeConIVA() {
    return this.importe * 1.21;
  }
  agregarStock(cantidad) {
    if (cantidad > 0) {
      this.stock += cantidad;
      return this.stock;
    }
  }
  descontarDeStock(cantidad) {
    let stockOriginal = this.stock;
    if (cantidad > 0) {
      this.stock -= cantidad;
      if (this.stock < 0) {
        console.warn(`No se acepta stock negativo: ${this.stock}`);
        this.stock = stockOriginal;
      }
    }
    return this.stock;
  }
}
```

La estructura de una Clase JS, también posee un **constructor**, utiliza la palabra reservada **this** en cada una de sus propiedades, e integra métodos asociados.

Lo bueno de ella es que es mucho más limpia la forma de declararla.



Clases JS

Se instancia de igual forma que una función constructora, y se utiliza también de igual forma.

También se recomienda crear cada clase JS en un archivo **.JS** independiente.

```
Clase JS

const nuevoProducto = new Producto('MacBook Air', 459000, 'Notebook');

nuevoProducto.nombre;
nuevoProducto.importeConIVA();
```

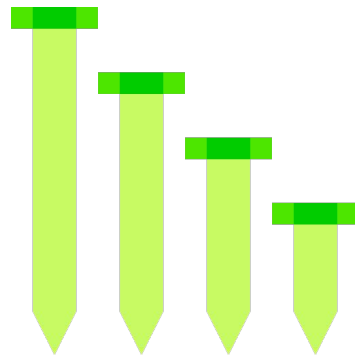


miembros estáticos en una Clase JS



miembros estáticos en una Clase JS

En una clase JS podemos definir miembros estáticos. Estos son métodos y/o propiedades que residen en el objeto constructor, y no en cada uno de los objetos creados a partir de una instancia de este.



miembros estáticos en una Clase JS

Por ejemplo, volviendo a la clase **Producto**, si creamos esta para ser instanciada y luego distribuirla como una clase JS estándar que cualquier desarrollador pueda utilizarla, podremos agregarle un método estático con un Copyright, el cual será visible a través de la consola JS.



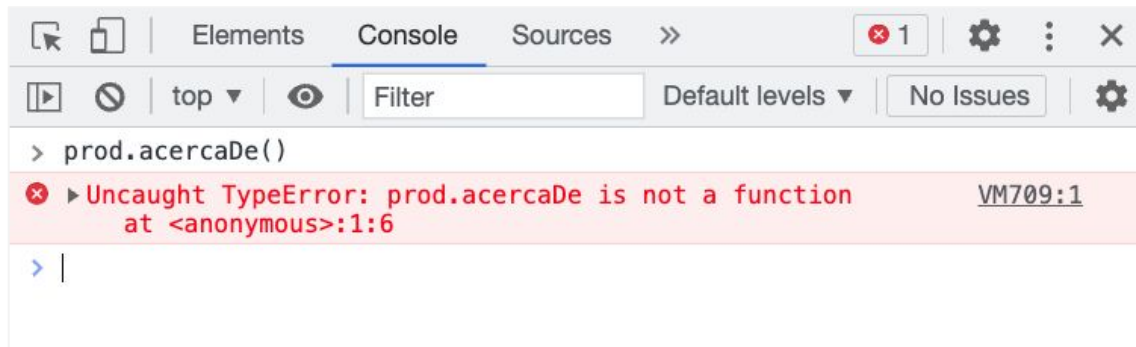
```
class Producto {  
  static acercaDe() {  
    console.log(`Copyright © 2023 - ISTEА.`);  
  }  
  constructor(nombre, importe, categoria) {  
    ...  
  }  
}
```


miembros estáticos en una Clase JS

```
Miembros estáticos en clases JS

const prod = new Producto('Macbook Air M1 13', 1250.00, 'Notebook');
```

Una vez instanciada la clase, el objeto en cuestión no hereda los miembros estáticos definidos en la clase.



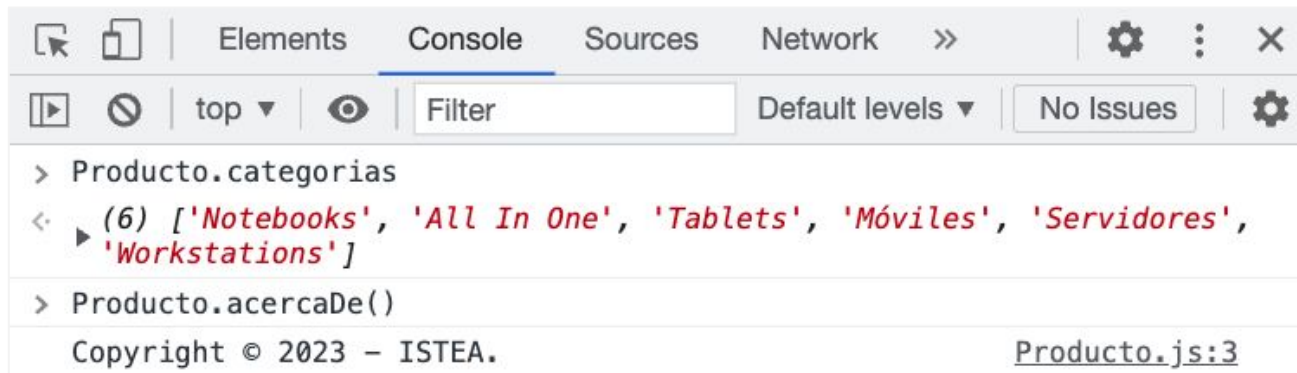
miembros estáticos en una Clase JS

Los miembros estáticos pueden ser tanto métodos como también propiedades. Aquí un ejemplo de una propiedad que contiene un array de elementos de forma interna.

```
Miembros estáticos en clases JS

class Producto {
  static acercaDe() {
    console.log(`Copyright © 2023 - ISTEА.`);
  }
  static categorias = ['Notebooks', 'All In One', 'Tablets',
                      'Móviles', 'Servidores', 'Workstations'];
  constructor(...) {
  }
}
```

miembros estáticos en una Clase JS



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following output:

```
> Producto.categorias
< (6) ['Notebooks', 'All In One', 'Tablets', 'Móviles', 'Servidores', 'Workstations']
> Producto.acercaDe()
Copyright © 2023 - ISTEА. Producto.js:3
```

The console interface includes a toolbar with icons for back, forward, and search, as well as tabs for Elements, Console, Sources, and Network. The Console tab is active, showing a list of messages. The first message is an expansion of the `Producto.categorias` array, which contains six elements: 'Notebooks', 'All In One', 'Tablets', 'Móviles', 'Servidores', and 'Workstations'. The second message is the return value of the `Producto.acercaDe()` method, which is the copyright notice 'Copyright © 2023 - ISTEА.'. The console also shows the file and line number `Producto.js:3` at the bottom right.

En la clase JS base encontraremos acceso a todos los miembros estáticos definidos dentro del constructor. Son los únicos métodos que podremos utilizar de forma directa.

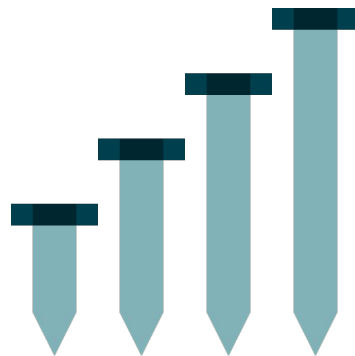
miembros privados en una Clase JS



miembros privados en una Clase JS

Hasta ahora, la estructura de propiedades definidas en una clase JS son accesibles de forma pública, y por lo tanto, instanciada la clase, cualquier propiedad puede ser modificada.

JS cuenta también con la posibilidad de definir miembros (*propiedades y/o métodos*) privados.



miembros privados en una Clase JS

Para esto, utilizaremos el caracter **#** (*hash - numeral*)
anteponiendo el mismo al nombre del miembro.

```
Miembros estáticos en clases JS

class Producto {
  static acercaDe() {
    console.log(`Copyright © 2023 - ISTEА.`);
  }
  #categorias = ['Notebooks', 'All In One', 'Tablets',
                'Móviles', 'Servidores', 'Workstations'];
  constructor(...) {
  }
}
```



miembros privados en una Clase JS

Esta propiedad no estará accesible de forma pública.

Para ello, debemos recurrir a un método, el cual nos permita ver su contenido.

```
Miembros privados en clases JS

class Producto {
  #categorias = ['Notebooks', 'All In One', 'Tablets',
                'Móviles', 'Servidores', 'Workstations'];
  constructor(...) {
  }
  getCategorias() {
    return this.#categorias;
  }
}
```



miembros privados en una Clase JS

Aquí es donde entran en juego el uso de **getter** y **setter**. Tanto en JS como en otros lenguajes de programación, se utilizan métodos denominados bajo estos términos, para obtener y modificar valores de propiedades de una clase JS.

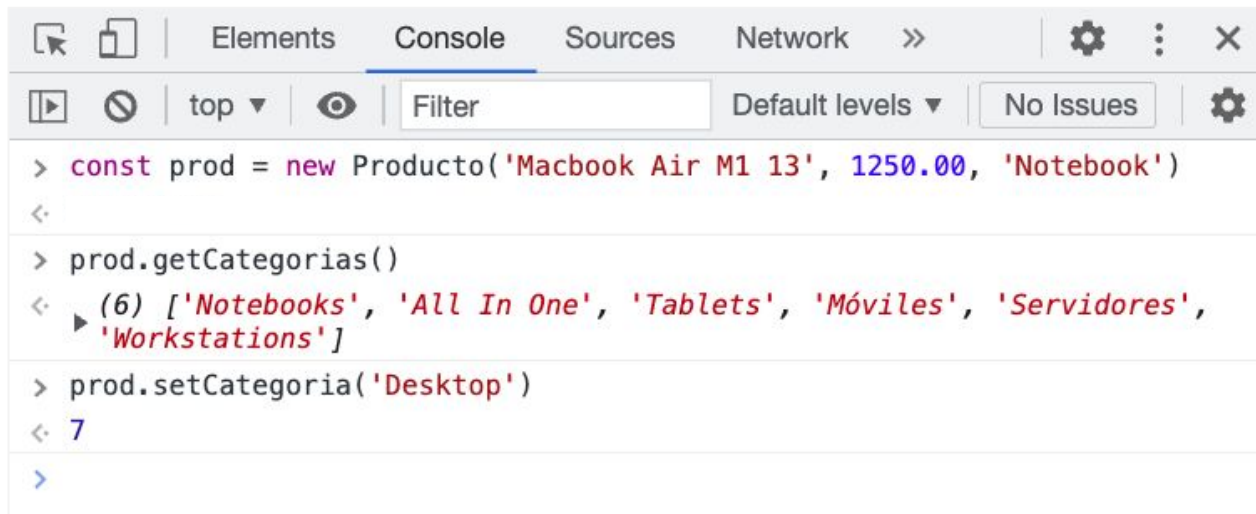
```
Miembros privados en clases JS

class Producto {
  #categorias = ['Notebooks', 'All In One', 'Tablets',
                'Móviles', 'Servidores', 'Workstations'];
  constructor(...) {
  }
  getCategorias() {
    return this.#categorias;
  }
  setCategoria(nueva) {
    return this.#categorias.push(nueva);
  }
}
```



miembros privados en una Clase JS

Veamos en acción a **getter** y **setter**. En este caso, accedemos a los datos de un array de elementos utilizando `getCategorias()`. Luego, con `setCategoria(nueva)`, podemos agregar un elemento al array en cuestión.



```
> const prod = new Producto('Macbook Air M1 13', 1250.00, 'Notebook')
<
> prod.getCategorias()
< (6) ['Notebooks', 'All In One', 'Tablets', 'Móviles', 'Servidores', 'Workstations']
> prod.setCategoria('Desktop')
< 7
>
```

¡Gracias!

educación IT