
TEMA 5.2 – OBJETOS Y CLASES EN JAVASCRIPT

La **Programación Orientada a Objetos (POO)** es uno de los paradigmas de programación más utilizados en las últimas décadas. Su popularidad se debe a que proporciona un enfoque **natural** y eficaz **para resolver problemas complejos**, especialmente aquellos que surgen en la programación estructurada tradicional. La POO cambia el foco **de cómo resolver un problema (enfoque procedimental)** a cuáles son los **elementos del problema y cómo interactúan entre ellos (enfoque orientado a objetos)**. La diferencia entre ambos paradigmas es:

- **Programación estructurada:** Se centra en dividir un problema en procedimientos o funciones que siguen una secuencia de **pasos** para llegar a una solución. En este paradigma, los datos y las funciones están separados.
- **Programación orientada a objetos:** En lugar de centrarse en los pasos para resolver un problema, la POO **organiza el código en torno a objetos que representan entidades en el mundo real**. Los objetos combinan datos (propiedades) y comportamientos (métodos), lo que hace que la interacción entre ellos resuelva el problema de manera más modular y organizada.

Tradicionalmente, la POO se implementa utilizando **clases**. Las clases **actúan como plantillas a partir de las cuales se crean los objetos**. Sin embargo, JavaScript no está basado en clases, sino en un sistema de **prototipos**.

- **Clases:** En lenguajes como java, una clase define las propiedades y métodos que tendrá un objeto. A partir de esa clase, se pueden crear múltiples instancias (objetos) con características similares.
- **Prototipos:** En lenguaje JavaScript, los prototipos son como **plantillas para crear nuevos objetos**. Cada objeto en JavaScript **hereda propiedades y métodos de su prototipo**, lo que permite la reutilización de código sin la necesidad de una estructura de clases formal.

Conocido este marco conceptual en el que moverse, es hora de empezar a escribir código, ya que gestionar estos objetos implica el conocimiento de importantísimos conceptos como es la distinción entre propiedades y métodos, que es un objeto genérico `Object`, etc.

1. Gestión de objetos:

El uso de propiedades y métodos de un objeto, aunque no se haya analizado formalmente, se ha practicado con mucha frecuencia a lo largo de esta asignatura. Si por ejemplo, recordamos como se consigue mostrar en consola el número de elementos de un array, se verá cómo se accede a las propiedades y métodos de un objeto.

Así que, de forma genérica, se puede acceder a las propiedades de un objeto utilizando la notación punto (.) o corchetes ([]).

```
let vector = [3,2,56,9];

console.log(typeof(vector));
console.log(typeof(console));

console.log(vector.length);
console.log(vector["length"]);

console.log("Mensaje 1");
console["log"]("Mensaje 2");
```

object	prueba.html:364
object	prueba.html:365
4	prueba.html:367
4	prueba.html:368
Mensaje 1	prueba.html:370
Mensaje 2	prueba.html:371

Por otra parte, `instanceOf` es un operador que ayuda a **comprobar el tipo de un objeto, es decir, el prototipo del que parte**. Para usarlo hay que preguntarle si el objeto es de un tipo concreto y devuelve `true` o `false`.

```
console.log(vector instanceof Array); //true
console.log(vector instanceof map); //false
```

true	prueba.html:373
false	prueba.html:374

1.1. Objeto genérico Object:

Los objetos son colecciones de pares **clave -valor**, donde cada **clave (propiedad)** tiene un **valor asociado**, que puede ser cualquier tipo de dato, incluso otro objeto o una función. Es **similar a un array pero en lugar de estar indexado por números, está indexado por nombres**.

Este objeto representa a los objetos literales (o instancias directas), que son los objetos más sencillos que pueden crearse en JavaScript. **No es necesario definirlos con una estructura completa, sino que puede construirse poco a poco a medida que va avanzando la ejecución.**

```
let notas = new Object();
notas.valores = [7, 5, 3, 2, 3, 9, 6];
notas.cantidad = notas.valores.length;
notas.suma = notas.valores.reduce((a, b) => a + b, 0);
notas.media = notas.suma / notas.cantidad;
notas.verMedia = function () {
  console.log(notas.media);
}
notas.verMedia();
console.log(notas);
```

5

[prueba.html:384](#)[prueba.html:387](#)

► {valores: Array(7), cantidad: 7, suma: 35, media: 5, verMedia: f}

Una presentación alternativa podría ser aquella en la que se utilice la **notación de objetos de JavaScript**:

```
let viaje = {
  origen : "Granada",
  destino : "El cairo",
  dias: 8,
  precio:750,
  mostrar:function(){
    console.log(`${viaje.origen} --> ${viaje.destino}`);
    console.log(`durante ${viaje.dias} dias: ${viaje.precio} EUR.`)
  }
}

viaje.mostrar();
console.log(viaje);
```

```
Granada --> El cairo                                prueba.html:397
durante 8 dias: 750 EUR.                             prueba.html:398
{origen: 'Granada', destino: 'El cairo', dias: 8, precio: 750, mostrar: f} prueba.html:403
```

La **notación JSON** (JavaScript Object Notation) es una representación de datos que está pensada para la transferencia de información entre sistemas. Es una versión simplificada de un objeto en JavaScript, que **no admite funciones**. JSON solo puede representar datos primitivos como cadenas, números, arrays y otros objetos, pero no funciones o métodos.

```
{
  "origen": "Granada",
  "destino": "El Cairo",
  "dias": 8,
  "precio": 750
}
```

1.2. Objeto this:

Es bastante común que un **método de un objeto necesite acceder a los datos almacenados en alguna de sus propiedades**, en el ejemplo anterior se podría reescribir el método `mostrar()` de este modo:

```
let viaje = {
  origen: "Granada",
  destino: "El cairo",
  dias: 8,
  precio: 750,
  mostrar: function () {
    console.log(`${this.origen} --> ${this.destino}`);
    console.log(`durante ${this.dias} dias: ${this.precio} EUR.`)
  }
}

viaje.mostrar();
console.log(viaje);
```

El objeto this es una referencia al contexto de ejecución actual. Es decir, **su valor cambia dependiendo de dónde y cómo se llame la función en la que se encuentra.**

Como se ve, se ha sustituido el objeto de referencia `viaje` por `this`. En tiempo de ejecución `this` valdrá `viaje`, aportando seguridad al código ya que se adapta automáticamente al contexto en el que se llama al método.

Por ejemplo, si tengo una variable `oferta` que apunta al mismo objeto que `viaje`:

```
let oferta = viaje;
```

Ahora mismo, tanto viaje como oferta hacen referencia al mismo objeto en memoria. Si ahora por ejemplo llamamos al método mostrar() usando la variable oferta, se adaptaría correctamente. Al contrario ocurriría si dentro de mostrar() estuviésemos usando viaje.origen en lugar de this.viaje, al ejecutar oferta.mostrar(), el código intentaría acceder a viaje.origen, lo que sería incorrecto.

```
let viaje = {
  origen: "Granada",
  destino: "El cairo",
  dias: 8,
  precio: 750,
  mostrar: function () {
    console.log(`${viaje.origen} --> ${viaje.destino}`);
    console.log(`durante ${viaje.dias} dias: ${viaje.precio} EUR.`)
  }
}

viaje.mostrar();
console.log(viaje);

let oferta = viaje;
viaje = null;
oferta.mostrar();
```

✖ ▶ Uncaught TypeError: Cannot read properties of null (reading 'origen')
 at Object.mostrar (prueba.html:397:38)
 at prueba.html:407:16

En cambio usando el objeto this en el método, si que se puede acceder correctamente a sus propiedades.

```
Granada --> El cairo
durante 8 dias: 750 EUR.
```

```
{origen: 'Granada', destino: 'El cairo', dias: 8, precio: 750, mostrar: f}
```

2. Operaciones sobre objetos:

Para crear y trabajar con estos objetos, JavaScript ofrece diversas **operaciones** que facilitan la construcción, manipulación y acceso a los datos que almacenan.

- **Constructores:** Son métodos especiales que se utilizan **para crear e inicializar objetos de manera personalizada**. Cada vez que se crea una instancia de un objeto utilizando

una clase o función constructora, el constructor se ejecuta para establecer los valores iniciales de ese objeto.

A partir de **ECMAScript 2015 (ES6)**, JavaScript introdujo el concepto de **clases**, lo que facilita mucho la definición de constructores y la creación de objetos. Sin embargo, aunque ahora tenemos la sintaxis de clases, **JavaScript sigue siendo un lenguaje basado en prototipos**. Las clases solo son una forma más conveniente de crear y gestionar objetos.

```
class Viaje {
  origen = "Granada";
  destino = "El cairo";
  dias = 8;
  precio = 750;
  constructor(or, des, di, pre) {
    this.origen = or;
    this.destino = des;
    this.dias = di;
    this.precio = pre;
  }

  mostrar() {
    console.log(`${this.origen} --> ${this.destino}`);
    console.log(`durante ${this.dias} dias: ${this.precio} EUR.`)
  }
}

let miViaje = new Viaje("Bcn", "Cracovia", 4, 49.99);
miViaje.mostrar();
```

Bcn --> Cracovia

[prueba.html:429](#)

durante 4 dias: 49.99 EUR.

[prueba.html:430](#)

- Solo puede haber un **método** llamado **constructor en cada clase**.
- Para invocar al constructor debo utilizar el operador **new**. Sin new no se crea una nueva instancia correctamente.
- El constructor **se ejecuta inicializando las propiedades del objeto** cuando se utiliza el operador new.
- Además, si la clase hereda de otra, se puede **ejecutar el constructor de la clase madre** invocando a **super()**. Esto es útil para asegurarse de que las propiedades definidas en la clase padre se inicializan correctamente.

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }
}

class Perro extends Animal {
  constructor(nombre, raza) {
    super(nombre);
    this.raza = raza;
  }
}

let miPerro = new Perro("Bobby", "Golden Retriever");
console.log(miPerro.nombre); // 'Bobby'
console.log(miPerro.raza); // 'Golden Retriever'
```

Bobby	prueba.html:452
Golden Retriever	prueba.html:453

- Aunque las clases introducidas en ES6 son la forma moderna de definir constructores, en objetos literales también puedes definir un método constructor dentro de un prototipo para lograr un comportamiento similar.

```
function Viaje(origen, destino, dias, precio) {
  this.origen = origen;
  this.destino = destino;
  this.dias = dias;
  this.precio = precio;
}

Viaje.prototype.mostrar = function () {
  console.log(`${this.origen} --> ${this.destino}`);
  console.log(`durante ${this.dias} días: ${this.precio} EUR.`);
};

let viaje1 = new Viaje("Barcelona", "París", 7, 500);
viaje1.mostrar();
```

- **Recorridos:** Para recorrer un objeto se recomienda utilizar el bucle for..in, pero con cuidado por se dan algunas circunstancias inesperadas:

```
let miViaje = new Viaje("Bcn", "Cracovia", 4, 49.99);
for (elemento in miViaje){
  console.log(elemento);
}
```

origen	prueba.html:436
destino	prueba.html:436
dias	prueba.html:436
precio	prueba.html:436

Además, también se ha de entender que este bucle itera sobre todas las propiedades enumerarles, incluidas las que pueda heredar del prototipo del objeto o de otro objeto:

```
class Transporte {
  constructor() {
    this.modo = "terrestre";
  }
}

class Viaje extends Transporte {
  constructor(origen, destino, dias, precio) {
    super();
    this.origen = origen;
    this.destino = destino;
    this.dias = dias;
    this.precio = precio;
  }
}

let miViaje = new Viaje("Bcn", "Cracovia", 4, 49.99);
for (let elemento in miViaje) {
  console.log(elemento);
}
```

modo	prueba.html:477
origen	prueba.html:477
destino	prueba.html:477
dias	prueba.html:477
precio	prueba.html:477

Este bucle lo que hace es iterar sobre las propiedades del objeto. Si se quiere prevenir este comportamiento para que acceda a esas propiedades del objeto en sí, y no a sus prototipos, se puede utilizar **getOwnPropertyNames()**, que devuelve un array con todas las propiedades propias del objeto:

```
let propiedades = Object.getOwnPropertyNames(miViaje);
for (let propiedad of propiedades) {
  console.log(propiedad);
}
```

- **Borrados:** La eliminación de propiedades sí que es una operación bastante simple. Tan solo hay que usar el operador **delete** sobre una propiedad de un objeto concreto.


```
let miViaje = new Viaje("Bcn", "Cracovia", 4, 49.99);

for (let elemento of Object.getOwnPropertyNames(miViaje)) {
  console.log(elemento);
}

delete miViaje.precio;
delete miViaje.dias;

console.log("Elimino las opriedades precio y días");

for(elemento of Object.getOwnPropertyNames(miViaje)){
  console.log(elemento);
}
```

Elimino las opriedades precio y días	prueba.html:484
modo	prueba.html:487
origen	prueba.html:487
destino	prueba.html:487

- **JSON**: Aunque anteriormente ya se ha comentado por encima en que consiste este formato de presentación basado en texto (.json). A continuación se va a detallar en profundidad. Aunque tiene una sintaxis similar a la de los **objetos literales de JavaScript**, **JSON** tiene algunas restricciones adicionales, como la imposibilidad de definir métodos y la necesidad de usar comillas dobles para las propiedades.

```
{
  "origen": "Granada",
  "destino": "El Cairo",
  "dias": 8,
  "precio": 750
}
```

- **Solo es posible definir propiedades, no métodos o funciones.** Solo pueden contener valores como números, cadenas, booleanos, arrays u otros objetos.
- Las propiedades deben ir entre comillas dobles. Aunque esto no es necesario en los objetos literales de JavaScript.

JavaScript incorpora un objeto llamado JSON que permite trabajar de forma muy eficiente con este tipo de cadenas. Proporciona dos métodos principales para trabajar con datos en formato JSON:

- **JSON.Stringify()** convierte un objeto JS a una cadena en forma JSON. Este método es útil cuando se desea enviar datos a través de la red, o cuando se quiere guardar datos de una manera legible como texto.

```
// Objeto JavaScript
let viaje = {
  origen: "Granada",
  destino: "El Cairo",
  dias: 8,
  precio: 750
};

console.log(typeof viaje);

// Convertir el objeto a una cadena JSON
let cadenaJSON = JSON.stringify(viaje);

console.log(cadenaJSON);
console.log(typeof cadenaJSON);
```

object	prueba.html:466
{"origen":"Granada","destino":"El Cairo","dias":8,"precio":750}	prueba.html:471
string	prueba.html:472

- **JSON.Parse()** devuelve el objeto equivalente en JS desde una cadena JSON. Es útil cuando recibes una cadena en formato JSON y se necesita convertirla en un formato JS para trabajar con ella.

```
// Cadena JSON
let cadenaJSON = '{"origen":"Granada","destino":"El Cairo","dias":8,"precio":750}';

console.log(typeof cadenaJSON);

// Convertir la cadena JSON en un objeto JavaScript
let objetoViaje = JSON.parse(cadenaJSON);

console.log(objetoViaje);
console.log(typeof objetoViaje);
```

string	prueba.html:463
▶ {origen: 'Granada', destino: 'El Cairo', dias: 8, precio: 750}	prueba.html:468
object	prueba.html:469

Es importante entender que: En JavaScript, podrías tener métodos en un objeto, pero cuando lo conviertes a JSON, los métodos se eliminarán.

3. Prototipos:

Como ya se avanzó, a diferencia de los lenguajes que usan el esquema de **clases** para la creación de objetos, JavaScript emplea un sistema basado en **prototipos**. En este modelo, todos los objetos en JavaScript están vinculados a un **prototipo**, que es un objeto del que heredan propiedades y métodos. Este es el sistema de herencia prototípica que permite a los objetos reutilizar propiedades y métodos sin copiarlos directamente (Property chain). Los prototipos son, en definitiva un mecanismo por el que los objetos de JavaScript heredan características entre sí.

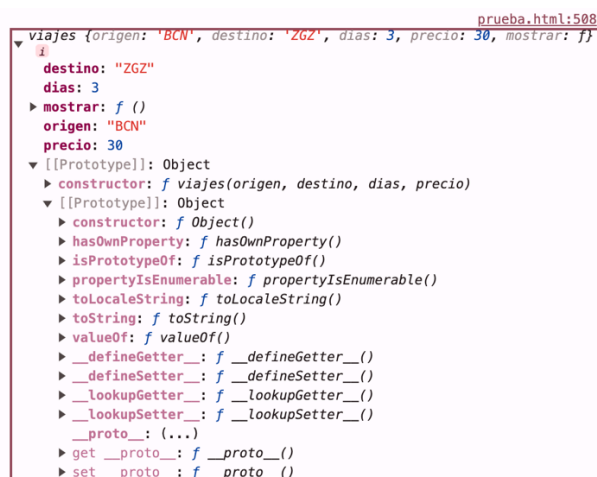
- Todos los objetos en JavaScript tienen una propiedad interna llamada **[[Prototype]]** (accesible mediante la propiedad `__proto__` o el método `Object.getPrototypeOf()`), que lo conecta con otro objeto, su **prototipo**.
 - Por ejemplo:

```
function viajes(origen, destino, dias, precio) {
  this.origen = origen,
  this.destino = destino,
  this.dias = dias,
  this.precio = precio,

  this.mostrar = function () {
    console.log(`${this.origen} --> ${this.destino}`);
    console.log(`durante ${this.dias} dias: ${this.precio} EUR.`);
  }
}

let viaje1 = new viajes("BCN", "ZGZ", 3, 30.00);
console.log(viaje1);
```

Si yo consulto en la salida de la consola y despliego el contenido de prototipo, veo todas las propiedades y métodos que tiene:



```
prueba.html:508
viajes {origen: 'BCN', destino: 'ZGZ', dias: 3, precio: 30, mostrar: f}
  destino: "ZGZ"
  dias: 3
  mostrar: f ()
  origen: "BCN"
  precio: 30
  [[Prototype]]: Object
    constructor: f viajes(origen, destino, dias, precio)
    [[Prototype]]: Object
      constructor: f Object()
      hasOwnProperty: f hasOwnProperty()
      isPrototypeOf: f isPrototypeOf()
      propertyIsEnumerable: f propertyIsEnumerable()
      toLocaleString: f toLocaleString()
      toString: f toString()
      valueOf: f valueOf()
      __defineGetter__: f __defineGetter__()
      __defineSetter__: f __defineSetter__()
      __lookupGetter__: f __lookupGetter__()
      __lookupSetter__: f __lookupSetter__()
      __proto__: (...)
      get __proto__: f __proto__()
      set __proto__: f __proto__()
```

El método **valueOf()** está diseñado para devolver el valor primitivo de un objeto. Por ejemplo, retorna el valor del objeto sobre el que se llama, pero que pasaría si el objeto Viaje reescribiera el método valueOf?

- El navegador comprobará inicialmente si el objeto viaje1 tiene un método valueOf() disponible en él.
- Si no lo tiene, el navegador comprobará si el objeto prototipado del objeto viaje1 tiene un método valueOf() disponible.
- Si tampoco lo tiene, entonces el navegador comprobará si Object() prototipo del objeto prototipado del constructor tiene un valueOf() disponible.

Además, se puede modificar cualquier prototipo operando con la propiedad prototype (sobre el constructor). Si se hace sobre la conocida clase viaje, se obtendría un prototipo estándar, puesto que nunca se ha modificado. Pero, se puede modificar y ver como dinámicamente asume la nueva situación.

```
let viaje1 = new viajes("BCN", "ZGZ", 3, 30.00);
console.log(viajes.prototype);

viajes.prototype.costeDiario = function () {
  return this.precio / this.dias;
}

viajes.prototype.descuento = "20%";
console.log(viajes.prototype)
```

► {}

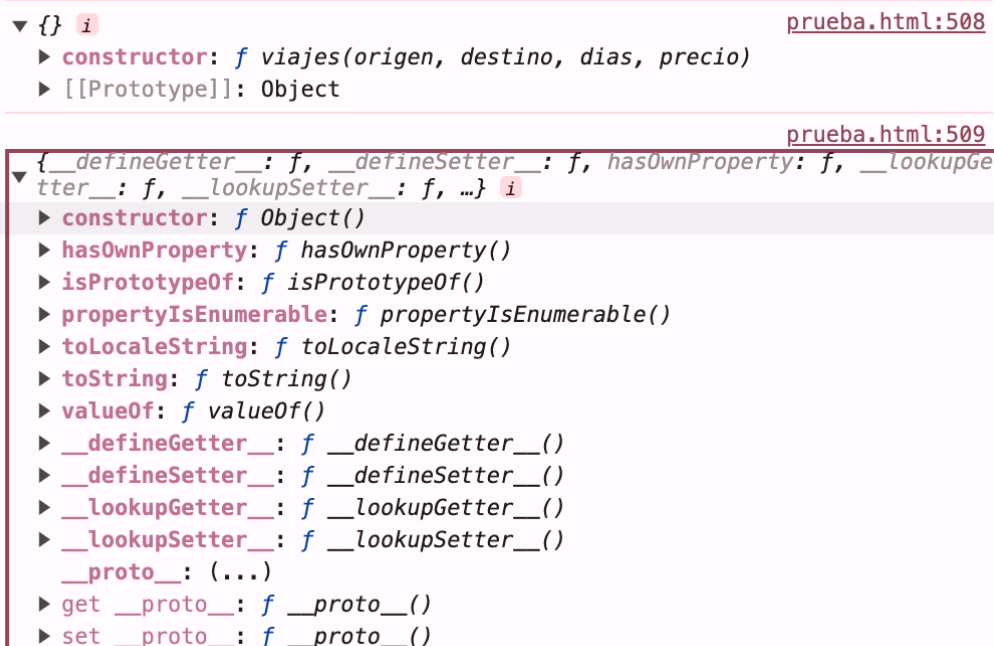
[prueba.html:508](#)

► {descuento: '20%', costeDiario: f}

[prueba.html:515](#)

Finalmente, recalcar una vez más que las propiedades y los métodos no se copian de un objeto a otro en la cadena del prototipo, sino que son accedidos subiendo por la cadena. Quizás, acceder de forma encadenada a la propiedad `__proto__` ayude a entender como conchen los navegadores quien es el objeto prototipo constructor del objeto:

```
let viaje1 = new viajes("BCN", "ZGZ", 3, 30.00);
console.log(viaje1.__proto__);
console.log(viaje1.__proto__.__proto__);
```



4. Objetos predefinidos en JavaScript:

Los objetos predefinidos del lenguaje son aquellos objetos que ya incorporan el lenguaje y que se pueden utilizar libremente con cualquier intérprete de JavaScript. Esto no es algo novedoso puesto que ya se ha trabajado con muchos de ellos.

- **String**: Cabe recordar que las cadenas se pueden especificar encerrándolas entre comillas simples ('), dobles (") o invertidas (`). También cabe recordar que el criterio de ordenación es el UNICODE. (TEMA 3)
- **Date**: Representan un momento dado en el tiempo que puede representarse en números formatos. Una fecha en JS se especifica como el número en milisegundos que han transcurrido desde el 1 de enero de 1970, UTC.

```
let fechaActual = new Date();
console.log("Fecha actual sin parámetros:", fechaActual);

let fechaCompleta = new Date(2024, 9, 7, 15, 30, 0, 0); // 7 de octubre de 2024, 15:30:00
console.log("Fecha completa con todos los parámetros:", fechaCompleta);

let fechaParcial1 = new Date(2024, 9); // 1 de octubre de 2024, 00:00:00
console.log("Fecha con solo año y mes:", fechaParcial1);

let fechaParcial2 = new Date(2024, 9, 15); // 15 de octubre de 2024, 00:00:00
console.log("Fecha con año, mes y día:", fechaParcial2);

let fechaDesdeMilisegundos = new Date(0); // 1 de enero de 1970
console.log("Fecha desde milisegundos (0):", fechaDesdeMilisegundos);
```

- **Array:** Ya visto en el tema anterior (TEMA 5.1)
- **Math:** Es un objeto predefinido que goza de mucha popularidad por la cantidad de operaciones matemáticas de cierta complejidad que resuelve. No es un objeto de función, no se puede editar y además todas sus propiedades y métodos son estáticos. (TEMA 3).
- **Boolean:** Es un objeto contenedor para un valor booleano, un valor lógico true o false. Por tanto no se deben confundir los valores booleanos primitivos true y false, con valores true y false del objeto Boolean.

```
let logico1= false;
let logico2= new Boolean(false);
console.log(logico1);
console.log(logico2);

if (logico1) {
  console.log("logico 1 entra");
}
if(logico2) {
  console.log("logico 2 entra");
}
```

false	prueba.html:532
► Boolean {false}	prueba.html:533
logico 2 entra	prueba.html:539

Este código muestra la diferencia entre valores primitivos y objetos en JavaScript, lo que ocurre es que en la primera condición se almacena el valor false y al evaluarse no se ejecuta el bloque if(logico1) porque ya entra en la condición. En la otra condición si que entra por que los **objetos** en JS siempre **se evalúan como verdaderos en una comparación booleana**, sin importar su valor interna.

- **Otros:**
 - **RegExp:** Las expresiones regulares es una utilidad que está presente en la mayoría de los lenguajes de programación. Una **expresión** es un **patrón** que se utiliza para **buscar coincidencias en las cadenas de texto**, de manera que puedan resolverse tareas frecuentes como la validación de datos. Se pueden construir tareas frecuentes como la validación de datos. Se puede construir de dos formas:

- Usando una expresión regular literal (las barras delimitan la expresión)
- Llamando a la función constructora del objeto RegExp.

```
let regex1 = /abc/; // Busca la cadena 'abc'|
let regex2 = new RegExp('abc'); // También busca 'abc'
```

La manera de comprobar si una cadena cumple con los **criterios del patrón establecido en la expresión regular es a través del método test()**, que recibe como parámetro la cadena a **comprobar**. Y devuelve true en caso de superar la **validación**, o false en caso contrario.

```
let regex = /abc/;
console.log(regex.test("abcde"));
console.log(regex.test("xyz")); |
```

true	prueba.html:546
false	prueba.html:547

Estos son algunos de los modificadores que ha creado JS para facilitar el uso de las sintaxis:

- **Modificador i:** Se indica para validar letras del alfabeto no se desea distinguir entre mayúsculas y minúsculas:

```
let regex = /abc/i;
console.log(regex.test("ABC")); // true (no distingue mayúsculas)
```

- **Modificar ^:** Fuerza a que la cadena empiece por el carácter inmediatamente posterior.

```
let regex = /^abc/;
console.log(regex.test("abcde")); // true (la cadena empieza con 'abc')
console.log(regex.test("deabc")); // false (la cadena no empieza con 'abc')
```

- **Modificador \$:** Fuerza a que la cadena termine por el carácter inmediatamente anterior:

```
let regex = /abc$/;
console.log(regex.test("deabc")); // true (la cadena termina con 'abc')
console.log(regex.test("abcde")); // false (la cadena no termina con 'abc')
```

- **Modificador . :** Representa un carácter cualquiera.

```
let regex = /a.c/;
console.log(regex.test("abc")); // true (coincide con 'a' seguido de cualquier carácter y luego 'c')
console.log(regex.test("a_c")); // true (cualquier carácter, incluyendo '_')
```

- **Modificador []:** Los símbolos de los corchetes establecen caracteres opcionales. La expresión la cumpliría cualquier cadena que contenga alguno de los elementos indicados entre corchetes.

```
let regex = /[abc]/;
console.log(regex.test("a")); // true (coincide 'a', 'b', o 'c')
console.log(regex.test("d")); // false (no está en el conjunto)
```

- **Modificador [^expresión]:** El carácter circunflejo como primer elemento de unos corchetes indica un carácter no permitido.

```
<script>
  let regex = /^[abc]/;
  console.log(regex.test("ab"));
  console.log(regex.test("gh"));
</script>
```

false

<!DOCTYPE html>.html:12

true

<!DOCTYPE html>.html:13

- **Modificador de cardinalidad:** Se trata de símbolos que permite configurar repeticiones de expresiones:

Métodos	Utilidad
Exp?	Coincide con 0 o 1 ocurrencia del carácter o grupo que lo precede.
Exp*	Coincide con 0 o más ocurrencias del carácter o grupo que lo precede.
Exp+	Coincide con 1 o más ocurrencias del carácter o grupo que lo precede.
Exp{n}	Coincide con exactamente n ocurrencias.
Exp{n,}	Coincide con al menos n ocurrencias.
Exp{m,n}	Coincide con entre m y n ocurrencias.

- Modificador `()`: Permiten agrupar expresiones, aumentando la complejidad del patrón.

```
let regex = /(abc)+/; // El patrón 'abc' debe repetirse al menos una vez
console.log(regex.test("abcabc")); // true
```

- Modificador `|`: Indica una opción es decir valida lo que está a su izquierda o a su derecha.

```
let regex = /abc|def/; // Coincide con 'abc' o 'def'
console.log(regex.test("abc")); // true
console.log(regex.test("def")); // true
```

- Modificador abreviados: Conjunto de símbolos a los que precede la barra invertida, que funciona muy bien en Unicode y permite escribir expresiones de una forma más ágil.

Símbolo	Utilidad
<code>\d</code>	Cualquier dígito numérico
<code>\D</code>	Cualquier carácter salvo los dígitos numéricos.
<code>\s</code>	Espacio en blanco.
<code>\S</code>	Cualquier carácter salvo el espacio en blanco.
<code>\w</code>	Cualquier carácter alfanumérico [a-zA-Z0-9]
<code>\W</code>	Cualquier carácter que no sea alfanumérico [^a-zA-Z0-9]
<code>\0</code>	Carácter nulo.
<code>\n</code>	Carácter de nueva línea.
<code>\t</code>	Carácter tabulador.
<code>\\</code>	El símbolo <code>\\</code>
<code>\"</code>	Comillas dobles.
<code>\'</code>	Comillas simples.
<code>\c</code>	Escapa el carácter c
<code>\ooo</code>	Carácter Unicode empleando la notación octal
<code>\xff</code>	Carácter ASCII empleando la notación hexadecimal.
<code>\uffff</code>	Carácter Unicode empleando la notación hexadecimal.

EJERCICIOS DE OBJETOS Y CLASES:

1. Crea una clase Producto con las propiedades nombre, precio y cantidad. Luego, crea una clase Inventario que maneje una lista de productos. El inventario debe tener métodos para **añadir productos, eliminar productos, y consultar el total del inventario (en términos de valor monetario)**. Usa el método *reduce()* para calcular el valor total del inventario.
2. Crea una clase Empleado que tenga las propiedades nombre y sueldo, y un método *detalles()*. Luego, crea dos clases derivadas: Gerente y Desarrollador. El Gerente debe tener una lista de empleados a su cargo. El Desarrollador debe tener una propiedad adicional llamada *lenguaje* (lenguaje de programación). Usa *super()* para invocar al constructor de la clase padre. Incluye un método en Gerente para **añadir empleados a su equipo** y un método para **mostrar los detalles del equipo**.
3. Crea un prototipo CuentaBancaria que tenga las propiedades saldo y un método *depositar()*. Luego, crea un prototipo derivado CuentaAhorro que herede de CuentaBancaria y tenga un método adicional *calcularInteres()* que aplique un interés del 5% sobre el saldo.
4. Crea una función *validarEmail()* que acepte un correo electrónico como argumento y use una expresión regular para verificar si es válido. Usa el método *test()* y explica la importancia de usar expresiones regulares en la validación de datos.
5. Crea un objeto *curso* que contenga un array de objetos *estudiante*, donde cada estudiante tenga nombre, nota y edad. Convierte el objeto en una cadena JSON usando *JSON.stringify()* y luego de nuevo en un objeto usando *JSON.parse()*. Modifica la nota de uno de los estudiantes.