# Los ciclos FOR

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Los **ciclos** nos permiten **repetir instrucciones** de manera sencilla. Podemos hacerlo una determinada **cantidad de veces** o mientras se cumpla una **condición**.

## Estructura básica

Consta de **3 partes** que definimos dentro de los paréntesis. En conjunto, nos permiten determinar de qué manera se van a realizar las **repeticiones** y definir las **instrucciones** que queremos que se lleven a cabo en cada una de ellas.

```
for (inicio; condición ; modificador) {
    //código que se ejecutará en cada repetición
}
```



## Estructura básica

En este ejemplo vamos a contar desde 1 hasta 5 inclusive.

```
for (let vuelta = 1; vuelta <= 5; vuelta++) {
  console.log('Dando la vuelta número ' + vuelta);
};
```

```
Dando la vuelta número 1
Dando la vuelta número 2
Dando la vuelta número 3
Dando la vuelta número 4
Dando la vuelta número 5
```

## Estructura básica

```
for (let vuelta = 1; vuelta <= 5; vuelta++) {
  console.log('Dando la vuelta número ' + vuelta);
};
```

**Inicio**

Antes de arrancar el ciclo, se establece el valor inicial de nuestro contador.



## Estructura básica

```
for (let vuelta = 1; vuelta <= 5; vuelta++) {
  console.log('Dando la vuelta número ' + vuelta);
};
```

**Condición**

Antes de ejecutar el código en cada vuelta, se pregunta si la condición resulta verdadera o falsa.

Si es **verdadera**, continúa con nuestras instrucciones.

Si es **falsa**, detiene el ciclo.

## Estructura básica

```
for (let vuelta = 1; vuelta <= 5; vuelta++) {
  console.log('Dando la vuelta número ' + vuelta);
};
```

**Modificador —incremento o decremento—**

Luego de ejecutar nuestras instrucciones, se modifica nuestro contador de la manera que hayamos especificado. En este caso se le suma 1, pero podemos hacer la cuenta que queramos.



## El ciclo for en acción

En cada ciclo se verifica si el valor de **vuelta** es menor o igual a 5. Si es así, se ejecuta el **console.log()** y se incrementa el valor de vuelta en 1.

Cuando vuelta deje de ser menor o igual a 5, se corta el ciclo.

| Iteración # | Valor de vuelta | ¿Vuelta <= 5 ? | Ejecutamos |
|---|---|---|---|
| 1 | 1 | true | ✓ |
| 2 | 2 | true | ✓ |
| 3 | 3 | true | ✓ |
| 4 | 4 | true | ✓ |
| 5 | 5 | true | ✓ |
| 6 | 6 | false | ✗ |



# Ciclos: while y do while

Como en muchos lenguajes de programación —y JavaScript no podría ser menos—, existe más de una forma de realizar las cosas. Varios casos pueden diferir en determinados aspectos y, por eso, el lenguaje nos provee de diversas estructuras para realizar acciones determinadas, que si bien comparten su función core difieren en pequeñas cuestiones, ya sean de lógica, de estructura o de código en sí.

En este caso veremos otras dos formas de representar y recrear acciones cíclicas: el **while loop** y el **do while**.

Si bien la estructura que más vamos a utilizar probablemente sea el ciclo **for**, existen casos y situaciones donde estas versiones pueden representar una mejora en nuestro código por la estructura que poseen.

Como pequeño aviso legal, durante las instancias evaluativas, no se pedirá ni evaluará el uso de la estructura While ni Do/While.

## Estructura básica while

Tiene una estructura similar a la de los condicionales **if/else**, palabra reservada + condición entre paréntesis. Sin embargo, el **while Loop** revalúa esa condición repetidas veces y **ejecuta su bloque de código** hasta que la condición **deja de ser verdadera**.

```
while (condicion) {
    //código que se ejecutará en cada repetición
    // Hace algo para que la condición eventualmente se deje de cumplir
}
```



## Estructura básica while

Tomando el ejemplo utilizado con el for, veamos como sería utilizando while.

```
let vuelta = 1
while(vuelta <= 5) {
  console.log('Dando la vuelta número ' + vuelta);
  vuelta++ //al final de cada vuelta sumara 1 a vuelta
};
```

```
Dando la vuelta número 1
Dando la vuelta número 2
Dando la vuelta número 3
Dando la vuelta número 4
Dando la vuelta número 5
```

Siguiente (→)

# Estructura básica while

Antes de ejecutar el código en cada vuelta, se pregunta si la condición resulta verdadera o falsa.

Si es **verdadera**, continúa con nuestras instrucciones.

Si es **falsa**, detiene el ciclo.

```
{}
let vuelta = 1
while(vuelta <= 5) {
  console.log('Dando la vuelta número ' + vuelta);
  vuelta++
};
```



# Estructura básica while

Es importante generar el contador al comenzar para evitar caer en lo que se conoce como **loop infinito**.

```
{}
let vuelta = 1
while(vuelta <= 5) {
  console.log('Dando la vuelta número ' + vuelta);
  vuelta++
};
```

El **loop infinito** sucede cuando nuestra condición es constantemente verdadera, lo que resulta en ejecutar nuestro código eternamente. Esto puede causar varios problemas, siendo el más importante que trabe todo nuestro programa.

> El ciclo **do while** es similar al **while**, pero se diferencia en que, sin importar la condición, la acción se realizará al menos una vez.



## Estructura básica do while

A diferencia del ciclo **while,** la condición en este caso se verifica al finalizar el bloque de código. Por lo tanto, sin importar lo que se resuelva, las acciones se realizarán al menos una vez.

```
{}
let vuelta = 5
do{
  console.log('Dando la vuelta número ' + vuelta);
  vuelta++ //Se suma 1 a vuelta por lo tanto vuelta = 6
} while(vuelta <= 5); //al vuelta ser 6 la condición retorna
false y se termina el bloque de código
```

Fuera de esto, el ciclo **do while** es idéntico en funcionalidad al ciclo **while**.

## Los ciclos en acción

**While**

En cada iteración se verifica si el valor de **vuelta** es menor o igual a 5. Si es así, se ejecuta el **console.log()** y se incrementa el valor de vuelta en 1.

Cuando vuelta deje de ser menor o igual a 5, se corta el ciclo.

| Iteración # | Valor de vuelta | ¿Vuelta <= 5 ? | Ejecutamos |
|---|---|---|---|
| 1 | 1 | true | ✓ |
| 2 | 2 | true | ✓ |
| 3 | 3 | true | ✓ |
| 4 | 4 | true | ✓ |
| 5 | 5 | true | ✓ |
| 6 | 6 | false | ✗ |

DigitalHouse>

## Los ciclos en acción

**Do While**

Por otro lado, en este caso se ejecuta el bloque de código al comenzar, y **luego** comienza a verificar que el valor de **vuelta** es menor o igual a 5.

Cuando vuelta deje de ser menor o igual a 5, se corta el ciclo.

| Iteración # | Valor de vuelta | ¿Vuelta <= 5 ? | Ejecutamos |
|---|---|---|---|
| 1 | 1 | no se verifica | ✓ |
| 2 | 2 | true | ✓ |
| 3 | 3 | true | ✓ |
| 4 | 4 | true | ✓ |
| 5 | 5 | true | ✓ |
| 6 | 6 | false | ✗ |

DigitalHouse>

# Introducción a arrays

Ya sabemos cómo generar distintos tipos de datos en distintas variables, ya sean **números, booleanos, o cadenas de caracteres (strings)**... Pero ¿cómo podemos agrupar mucha información en una sola variable? Para ello contamos con un tipo de dato un poco más estructurado llamado **array** —también conocido como lista o arreglo—, que no es más que una "colección de elementos".

# Los arrays

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Los **arrays** nos permiten generar una **colección de datos ordenados**.

## Estructura de un array

Utilizamos corchetes [] para indicar el **inicio** y el **fin** de un array. Y usamos comas , para separar sus elementos.

Dentro de un array, podemos almacenar la cantidad de elementos que queramos, sin importar el tipo de dato de cada uno. Es decir, podemos tener en un mismo array datos de tipo string, number, boolean y todos los demás.

```
{}  let miArray = ['Star Wars', true, 23];
```



## Posiciones dentro de un array

Cada dato de un array ocupa una posición numerada conocida como **índice**. La **primera posición** de un array es **siempre 0**.

```
{}  let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```


Para acceder a un elemento puntual de un array, nombramos al array y, **dentro de los corchetes**, escribimos el índice al cual queremos acceder.

```
{}  pelisFavoritas[2];
    // accedemos a la película Alien, el índice 2 del array
```

# Métodos de un array (Parte 1)

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Para JavaScript, los **arrays** son un **tipo especial de objetos**. Por esta razón disponemos de muchos **métodos** muy útiles a la hora de trabajar con la información que hay adentro.

Ya vimos antes que una **función** es un **bloque de código** que nos permite agrupar funcionalidad para usarla muchas veces. Cuando una **función pertenece a un objeto**, en este caso nuestro array, la llamamos **método**.



## .push()

Agrega uno o varios elementos al final del array.

- **Recibe** uno o más elementos como parámetros.
- **Retorna** la nueva longitud del array.

{}

```
let colores = ['Rojo','Naranja','Azul'];
colores.push('Violeta'); // retorna 4
console.log(colores); // ['Rojo','Naranja','Azul','Violeta']

colores.push('Gris','Oro');
console.log(colores);
// ['Rojo','Naranja','Azul','Violeta','Gris','Oro']
```

## .pop()

Elimina el último elemento de un array.

- **No recibe** parámetros.
- **Devuelve** el elemento eliminado.

```
let series = ['Mad Men','Breaking Bad','The Sopranos'];

// creamos una variable para guardar lo que devuelve .pop()
let ultimaSerie = series.pop();

console.log(series); // ['Mad men', 'Breaking Bad']
console.log(ultimaSerie); // ['The Sopranos']
```



## .shift()

Elimina el primer elemento de un array.

- **No recibe** parámetros.
- **Devuelve** el elemento eliminado.

```
let nombres = ['Frida','Diego','Sofía'];

// creamos una variable para guardar lo que devuelve .shift()
let primerNombre = nombres.shift();

console.log(nombres); // ['Diego', 'Sofía']
console.log(primerNombre); // ['Frida']
```

## .unshift()

Agrega uno o varios elementos al principio de un array.

- **Recibe** uno o más elementos como parámetros.
- **Retorna** la nueva longitud del array.

```
let marcas = ['Audi'];

marcas.unshift('Ford');
console.log(marcas); // ['Ford', 'Audi']

marcas.unshift('Ferrari','BMW');
console.log(marcas); // ['Ferrari','BMW','Ford', 'Audi']
```



## .join()

Une los elementos de un array utilizando el separador que le especifiquemos. Si no lo especificamos, utiliza comas.

- **Recibe** un separador (string), es opcional.
- **Retorna** un string con los elementos unidos.

```
let dias = ['Lunes','Martes','Jueves'];

let separadosPorComa = dias.join();
console.log(separadosPorComa); // 'Lunes,Martes,Jueves'

let separadosPorGuion = dias.join(' - ');
console.log(separadosPorGuion); // 'Lunes - Martes - Jueves'
```

## .indexOf()

Busca en el array el elemento que recibe como parámetro.

- **Recibe** un elemento a buscar en el array.
- **Retorna** el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna un -1.

```
{}
let frutas = ['Manzana','Pera','Frutilla'];
frutas.indexOf('Frutilla');
// Encontró lo que buscaba. Devuelve 2, el índice del elemento

frutas.indexOf('Banana');
// No encontró lo que buscaba. Devuelve -1
```



## .lastIndexOf()

Similar a `.indexOf()`, con la salvedad de que empieza buscando el elemento por el **final del array** (de atrás hacia adelante).

En caso de haber elementos repetidos, devuelve la posición del primero que encuentre (o sea el último si miramos desde el principio).

```
{}
let clubes = ['Racing','Boca','Lanús','Boca'];

clubes.lastIndexOf('Boca');
// Encontró lo que buscaba. Devuelve 3

clubes.lastIndexOf('River');
// No encontró lo que buscaba. Devuelve -1
```

## .includes()

También similar a .indexOf(), con la salvedad que retorna un booleano.

- **Recibe** un elemento a buscar en el array.
- **Retorna** *true* si encontró lo que buscábamos, *false* en caso contrario.

```
let frutas = ['Manzana','Pera','Frutilla'];

frutas.includes('Frutilla');
// Encontró lo que buscaba. Devuelve true

frutas.includes('Banana');
// No encontró lo que buscaba. Devuelve false
```

DigitalHouse>

## Longitud de un array

Otra propiedad útil de los arrays es su longitud, o cantidad de elementos. Podemos saber el número de elementos usando la propiedad length.

```
let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```

1 + 1 + 1 = 3

Para acceder al total de elementos de un **array**, nombramos al array y, **seguido de un punto** ., escribiremos la palabra **length**.

```
pelisFavoritas.length;
// Devuelve 3, el número de elementos del array
```

DigitalHouse>

# Un poco más sobre strings

Ahora que conocemos los **arrays** vemos que podemos definirlos como listas o colecciones de elementos, organizados por un índice que comienza con el 0. Si lo analizamos, los strings son algo similares. Veamos un poco más sobre ellos.

Antes de meternos a fondo con los métodos de **string**, recordemos que —en muchos sentidos— para JavaScript un string no es más que una colección de

caracteres. Esas cadenas de texto que parecen ser tan sencillas, pero que tienen un alto potencial.

En el siguiente video veremos que podemos hacer mucho más con un string que simplemente generar textos.

## Los strings en JavaScript

En muchos sentidos, para JavaScript, un **string** no es más que un **array de caracteres**. Al igual que en los arrays, la primera posición siempre será 0.

```
let nombre = 'Fran';
```

0123

Para acceder a un carácter puntual de un string, nombramos al string y, **dentro de los corchetes**, escribimos el **índice** al cual queremos acceder.

```
nombre[2];
// accedemos a la letra a, el índice 2 del string
```



## .length

Esta **propiedad** retorna la **cantidad total de caracteres** del string, incluidos los espacios.
Al ser una propiedad (veremos más sobre ellas en clases siguientes), solo debemos llamarla, sin necesidad de los paréntesis.

```
let miSerie = 'Mad Men';
miSerie.length; // devuelve 7

let arrayNombres = ['Bart', 'Lisa', 'Moe'];
arrayNombres.length; // devuelve 3

arrayNombres[0].length; // Corresponde a 'Bart', devuelve 4
```

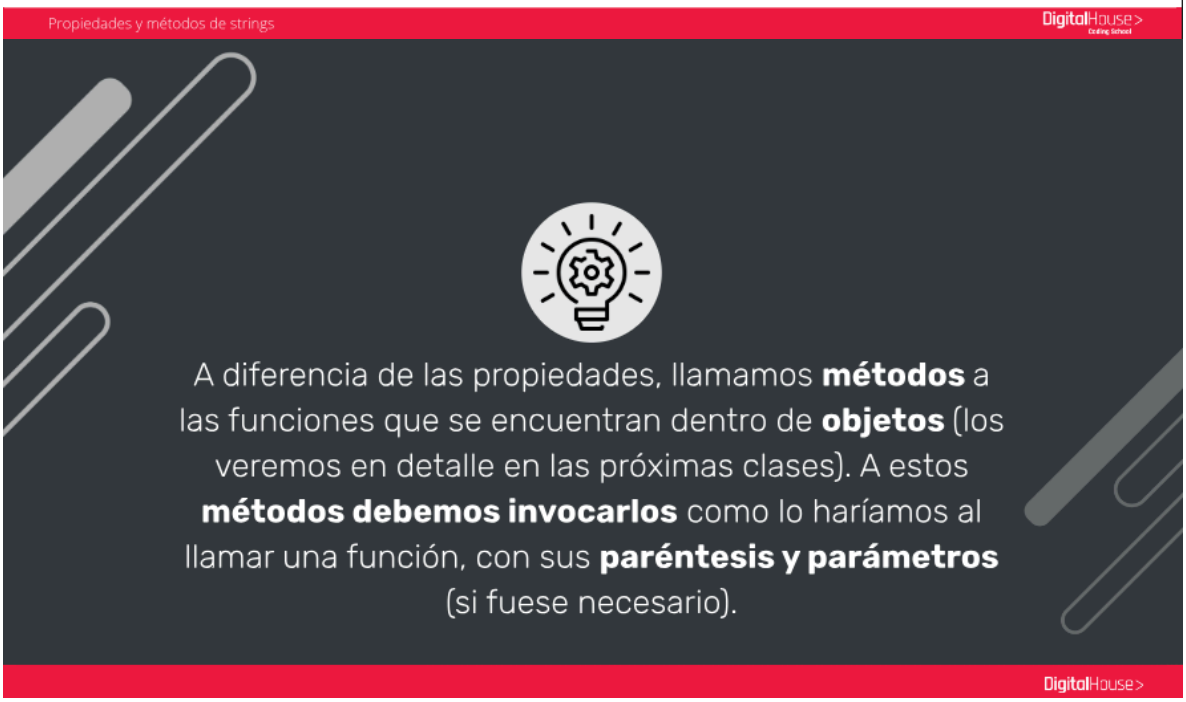
## .indexOf()

Busca, en el string, el string que recibe como parámetro.

- **Recibe** un elemento a buscar en el array.
- **Retorna** el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna un -1.

```
{}
let saludo = '¡Hola! Estamos programando';

saludo.indexOf('Estamos'); // devuelve 7
saludo.indexOf('vamos'); // devuelve -1, no lo encontró
saludo.indexOf('o'); // encuentra la letra 'o' que está en la posición 2, devuelve 2 y corta la ejecución
```



## .slice()

Corta el string y devuelve una parte del string donde se aplica.

- **Recibe 2** números como parámetros (pueden ser negativos):
  - El índice desde donde inicia el corte.
  - El índice hasta donde hacer el corte (es opcional).
- **Retorna** la parte correspondiente al corte.

```
{}
let frase = 'Breaking Bad Rules!';

frase.slice(9,12); // devuelve 'Bad'
frase.slice(13); // devuelve 'Rules!'
frase.slice(-10); // ¿Qué devuelve? ¡A investigar!
```



## .trim()

Elimina los espacios que estén al principio y al final de un string.

- **No recibe** parámetros.
- No quita los espacios del medio.

```
{}
let nombreCompleto = '   Homero Simpson   ';
nombreCompleto.trim(); // devuelve 'Homero Simpson'

let nombreCompleto = '   Homero   J.   Simpson   ';
nombreCompleto.trim(); // devuelve 'Homero   J.   Simpson'
```

## .replace()

Reemplaza una parte del string por otra.

- **Recibe** dos strings como parámetros:
  - El string que queremos buscar.
  - El string que usaremos de reemplazo.
- **Retorna** un nuevo string con el reemplazo.

```
{}
let frase = 'Aguante Python!';
frase.replace('Python', 'JS'); // devuelve 'Aguante JS!'
frase.replace('Py', 'JS'); // devuelve 'Aguante JSthon!'
```



## .split()

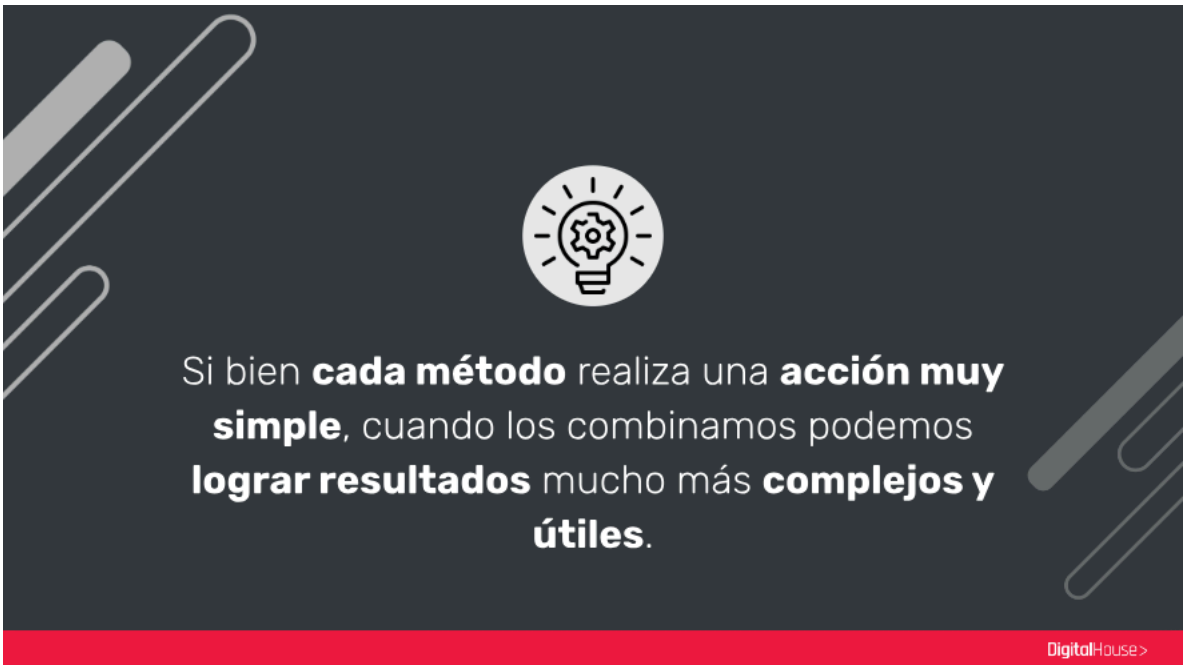
Divide un string en partes.

- **Recibe** un string que usará como separador de las partes.
- **Devuelve** un array con las partes del string.

```
{}
let cancion = 'And bingo was his name, oh!';

cancion.split(' ');
// devuelve ['And', 'bingo', 'was', 'his', 'name,' , 'oh!']

cancion.split(', ');
// devuelve ['And bingo was his name', 'oh!']
```




# Objetos literales

En JavaScript, un objeto literal es una entidad independiente con propiedades. A su vez, esas propiedades tienen valores.

El concepto de objetos puede compararse con entidades de la vida real. Por ejemplo, un país representaría un objeto literal con ciertas propiedades: su nombre, cantidad de habitantes, capital, etc. Del mismo modo, los objetos literales en JavaScript pueden tener propiedades que definan sus características.

Imaginemos entonces que queremos crear el modelo de un país en JavaScript. Mirá este video para aprender cómo hacerlo a través de un objeto literal.

# Objetos literales

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Podemos decir que los objetos literales son la **representación en código** de un **elemento de la vida real**.

# Estructura básica

Un **objeto** es una estructura de datos que puede contener **propiedades** y **métodos**.

Para crearlo usamos llave de apertura y de cierre {}.

```
let auto = {
    patente : 'AC 134 DD'
};
```

**PROPIEDAD**

Definimos el nombre de la **propiedad** del objeto.

**DOS PUNTOS**

**Separa** el nombre de la propiedad de su valor.

**VALOR**

Puede ser cualquier **tipo de dato** que conocemos.



# Propiedades de un objeto

Un objeto puede tener la cantidad de propiedades que queramos. Si hay más de una, las separamos con comas , .

Con la notación objeto.propiedad accedemos al valor de cada una de ellas.

```
let tenista = {
    nombre: 'Roger',
    apellido: 'Federer'
};

console.log(tenista.nombre) // Roger
console.log(tenista.apellido) // Federer
```

## Métodos de un objeto

Una propiedad puede almacenar cualquier tipo de dato. Si una propiedad almacena una **función**, diremos que es un **método** del objeto. Con una estructura similar a la de las funciones expresadas, vemos que se crean mediante el nombre del método, seguido de una función anónima.

```
{}
let tenista = {
    nombre: 'Roger',
    edad: 38,
    activo: true,
    saludar: function() {
        return '¡Hola! Me llamo Roger';
    }
};
```



## Ejecución de un método de un objeto

Para ejecutar un método de un objeto usamos la notación objeto.metodo(). Los paréntesis del final son los que hacen que el método se ejecute.

```
{}
let tenista = {
    nombre: 'Roger',
    apellido: 'Federer',
    saludar: function() {
        return '¡Hola! Me llamo Roger';
    }
};

console.log(tenista.saludar()); // ¡Hola! Me llamo Roger
```

# Trabajando dentro del objeto

La palabra reservada **this** hace referencia al objeto en sí donde estamos parados. Es decir, el objeto en sí donde escribimos la palabra.

Con la anotación this.propiedad accedemos al valor de cada propiedad interna de ese objeto.

{}

```
let tenista = {
     nombre: 'Roger',
     apellido: 'Federer',
     saludar: function() {
          return '¡Hola! Me llamo ' + this.nombre;
     }
};
console.log(tenista.saludar()); // ¡Hola! Me llamo Roger
```