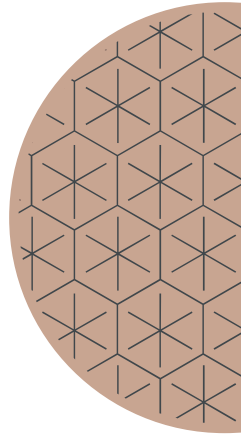




Características del lenguaje JavaScript

Características del lenguaje JavaScript

En esta unidad veremos los fundamentos de JavaScript. El objetivo es crear una base de conocimiento que será necesaria para cuando veamos a futuros conceptos más complejos de Javascript.



¿Cómo se escriben los comentarios en JavaScript?

En JavaScript, hay dos tipos de comentarios.

Un tipo es el comentario de una sola línea, y escribimos el comentario usando `//`. El texto que sigue después del `//` es tratado como un comentario y es ignorado por el compilador de JavaScript.

Otro tipo de comentario es el de varias líneas. Para ello se utilizan los caracteres de escape `/*` para abrir y `*/` para cerrar.

¿Cómo se escriben los comentarios en JavaScript?

//Comments in JavaScript

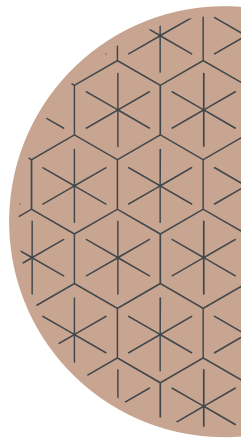
//This is a Single Line comment

*/**

Multi-line comment

Multi-line comments can also be used as single-line comments.

**/*





Conceptos básicos

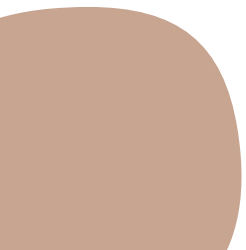
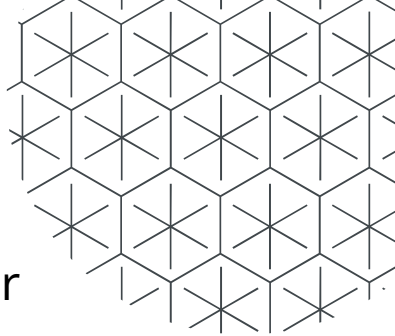


Identificadores

En JavaScript utilizamos identificadores para nombrar variables, propiedades y objetos, clases y funciones.

Hay reglas específicas que debemos seguir si queremos tener identificadores.

- El identificador de JavaScript debe comenzar con una letra, un guión bajo "_" o un signo de dólar "\$".
- El primer carácter no debe ser un número o dígito
- El siguiente carácter después del primero puede ser una letra, un dígito, un guión bajo o un signo de dólar, y cualquier combinación de ellos.



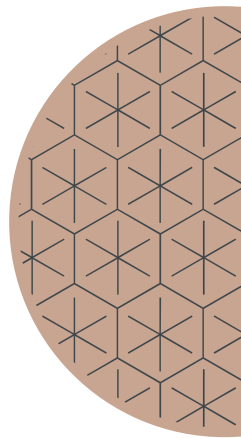
Statements

En JavaScript tenemos sentencias. Aquí hay un ejemplo con tres declaraciones:

```
let firstName = 'Rick ';
```

```
let lastName = 'Sekuloski';
```

```
alert(firstName + lastName);
```

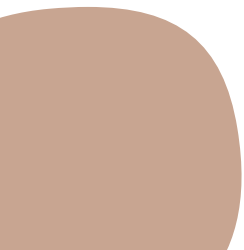


Statements

Aquí tengo tres líneas de código, y cada una de esas líneas está separada por un punto y coma (;). No es necesario en JavaScript.

Entonces, ¿qué es una sentencia de JavaScript? Las sentencias le dicen al programa de ordenador que haga algo.

Otra regla importante es que podemos agrupar dos o más sentencias en una línea siempre que las separe el punto y coma.



Sensibilidad a las mayúsculas y minúsculas

Los identificadores de JavaScript distinguen entre mayúsculas y minúsculas.

Esto significa que si nombramos una variable con una sola letra como 'x', podemos referirnos a esa variable en el programa como 'x'. Las variables como 'x' y la letra mayúscula 'X' son dos nombres de variables diferentes.

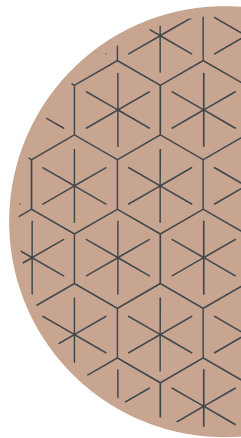
Tipos primitivos y objetos

Los tipos de JavaScript pueden pertenecer a dos categorías:

- Tipos primitivos
- Tipos objeto.

Los tipos primitivos de JavaScript pueden ser

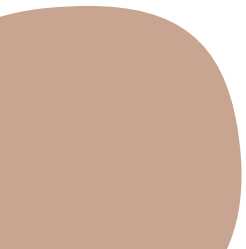
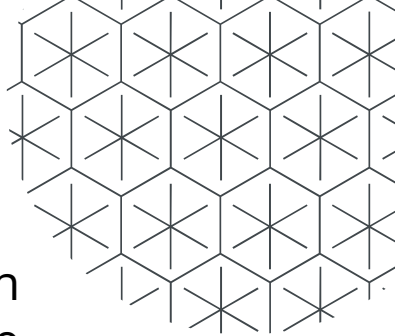
- Strings
- Números
- Valores booleanos
- etc.



Tipos primitivos y objetos

Cualquier valor que no sea un valor primitivo se considera un objeto. Un objeto es diferente a los valores primitivos porque cada objeto puede tener múltiples propiedades. Cada propiedad tiene un nombre y un valor.

La diferencia entre los tipos primitivos y los objetos es que los tipos primitivos se almacenan por valor, y los no primitivos (objetos) se almacenan por referencia.



Variables y asignación

Uno de los conceptos más importantes y fundamentales en JavaScript es la declaración de variables y la asignación de valores.

Una variable es un contenedor que existe para que podamos almacenar un valor. Las variables tienen nombres, y los nombres deben seguir las mismas reglas que hemos explicado sobre los identificadores. La parte más importante es **declarar** la variable antes de poder utilizarla.

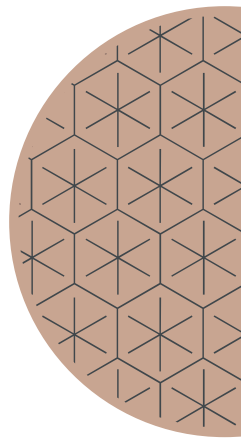
Inicialización de una variable

Una vez que hemos declarado una variable, podemos inicializarla con un valor. Podemos hacerlo escribiendo el nombre de la variable en el lado izquierdo, luego le sigue el signo de igualdad (=), y finalmente, proporcionamos el valor que queremos almacenar en esa variable. Por defecto el valor es undefined.

//Initializing variable

myName = 'Andy';

myAge = 33;

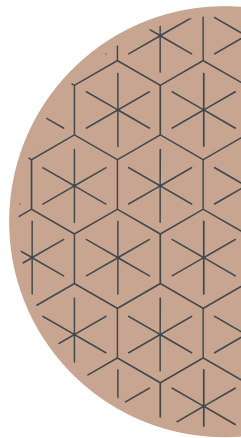


Inicialización de una variable

Ejercicio) ¿Cuál es la diferencia entre null y undefined?

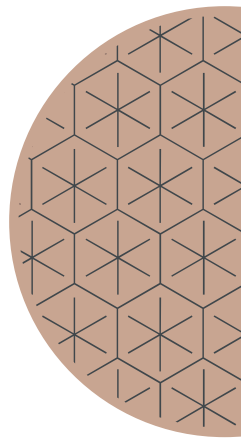
Cuando declaramos una variable pero no le asignamos un valor, el valor de esa variable es undefined por defecto. El segundo valor que indica una ausencia de valor de la variable es null.

La palabra clave null indica que "no hay valor" para cadenas, números y objetos. Tenga en cuenta que undefined no es una palabra clave reservada, pero null sí.



Inicialización de una variable

Es aconsejable no nombrar variables o constantes como `undefined`. Tanto `null` como `undefined` se convierten en valores booleanos falsos, y ambos se utilizan para indicar la ausencia de un valor. Rara vez tenemos que asignar a una variable que devuelva `null` o `undefined`, pero si se tuviera que hacer, utilizaría `null`.



Asignación de variables

Uno de los conceptos más importantes y fundamentales en JavaScript es la declaración de variables y la asignación de valores.

Una variable es un contenedor que existe para que podamos almacenar un valor. Las variables tienen nombres, y los nombres deben seguir las mismas reglas que hemos explicado sobre los identificadores.

Asignación de variables

Para las variables con nombres que son una composición de dos palabras utilizamos una convención llamada camel case.

let userNames;

Las variables también tienen un tipo de datos, y ese tipo de datos proviene de su valor actual. En la mayoría de los lenguajes de programación, el tipo de datos es fuertemente fijo y no puede ser cambiado después de su declaración, pero en JavaScript, este no es el caso; podemos establecer la variable a cualquier tipo que queramos.

Iniciación de variables

Una vez que hemos declarado una variable, podemos inicializarla con un valor.

Podemos hacerlo escribiendo el nombre de la variable en el lado izquierdo, luego le sigue el signo de igualdad (=), y finalmente, proporcionamos el valor que queremos almacenar en esa variable.

```
let myName = 'Andy';  
myAge = 33;
```

Var, Let, y Const

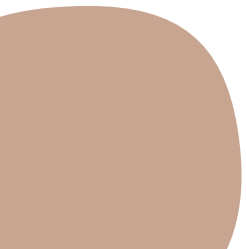
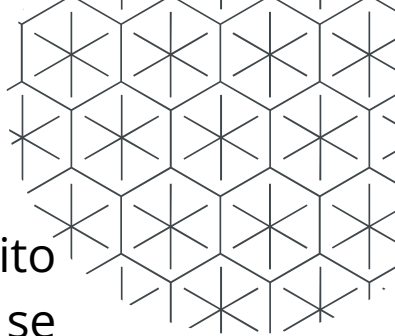
Antes de declarar el nombre de la variable, he utilizado la palabra clave "let", pero ¿qué significa esta palabra clave?

En las versiones de JavaScript anteriores a la ES6, utilizábamos otra palabra clave llamada "var" y todavía se puede utilizar. En ES2015 o ES6, tenemos dos nuevas formas de crear variables, y son utilizando las palabras clave "let" y "const".

Var, Let, y Const

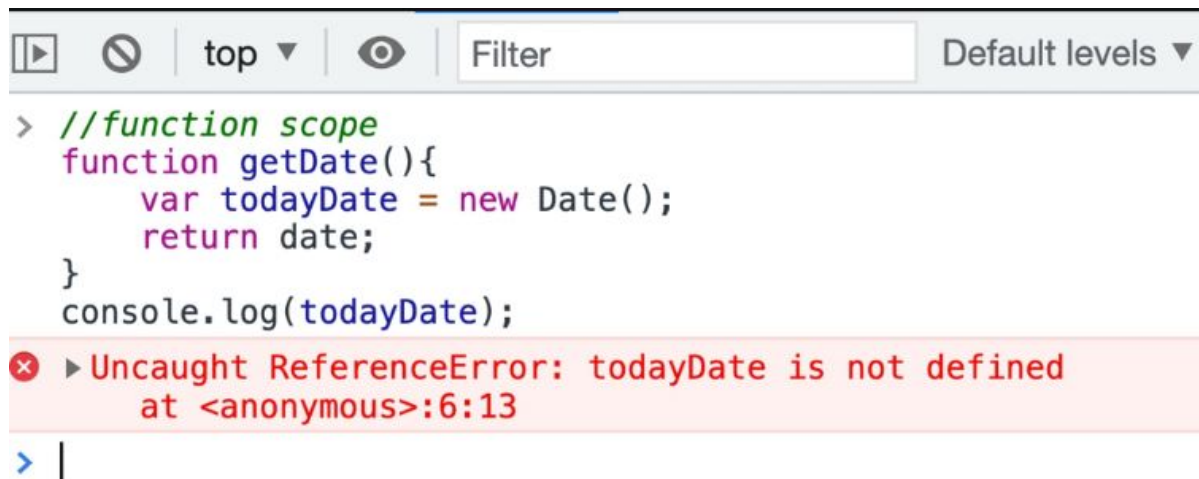
Para entender por qué usamos esas palabras clave, necesito explicarte un poco sobre el scope. El scope define dónde se puede acceder a las variables y funciones declaradas. Por lo tanto, no podemos acceder a algo que no está en el scope actual. En JavaScript, hay dos scopes:

- Scope global
- Scope función
- Scope bloque



Var, Let, y Const

Todavía no hemos hablado de las funciones, pero veamos el siguiente ejemplo:

A screenshot of a web browser's developer console. The top toolbar shows icons for running, pausing, and zooming, along with a 'Filter' input and 'Default levels' dropdown. The console log shows a JavaScript function definition for 'getDate' using 'var' to declare 'todayDate'. Below the code, a red error message is displayed: 'Uncaught ReferenceError: todayDate is not defined at <anonymous>:6:13'. The error message is preceded by a red 'x' icon. The console prompt '> |' is visible at the bottom.

```
> //function scope
function getDate(){
    var todayDate = new Date();
    return date;
}
console.log(todayDate);
```

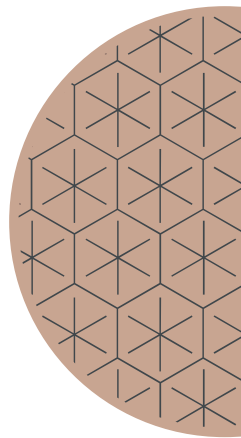
✖ ▶ Uncaught ReferenceError: todayDate is not defined
at <anonymous>:6:13

```
> |
```

Var, Let, y Const

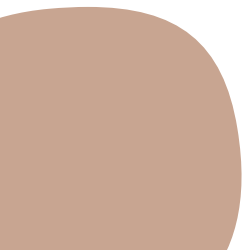
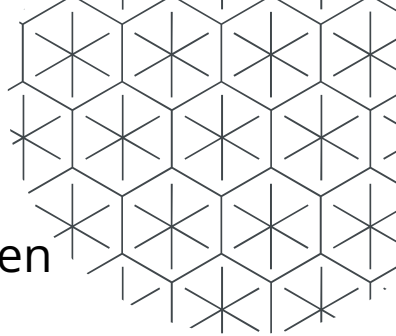
Cuando tenemos una variable declarada fuera del cuerpo de la función, esta variable pertenece a un scope global. Las variables de scope global son parte del objeto global, y podemos acceder a ese objeto usando la palabra clave global this.

Otro punto importante a entender es que puedes declarar la misma variable múltiples veces usando var, por lo que re-declarar y re-inicializar no es un problema con var, pero esto será un problema si usamos let y const.



Var, Let, y Const

Ejercicio) ¿Para qué sirve el comando delete en Javascript?



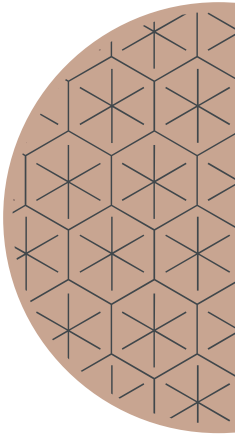
Let

Una forma más moderna de declarar variables es utilizar la palabra clave "**let**". A diferencia de la palabra clave `var`, `let` permite declarar variables que tienen un scope de bloque.

Se recomienda asignar un valor al declarar el nombre de la variable si es posible. Hay ejemplos en los que no necesitamos hacer esto, y también los cubriremos. Al igual que con `var`, la palabra clave `let` nos permite declarar una variable sin un valor inicial, y si intentamos acceder a esa variable, tendremos `undefined`.

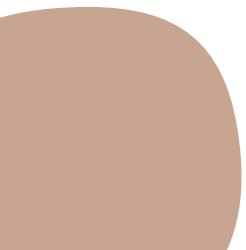
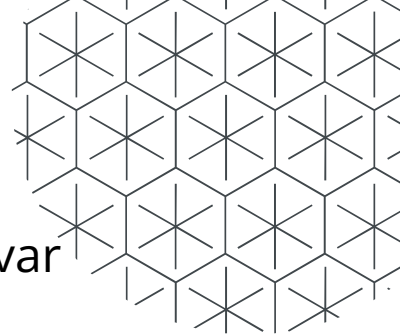
Let

Las variables declaradas con las palabras clave "let" y "const" tienen scope de bloque. Esto significa que son visibles y accesibles dentro del bloque donde se definen. Pero podemos declararlas fuera en el scope global, pero a diferencia de las palabras clave var, las let no crean propiedades del objeto global de la ventana.



Let

Ejercicio) Entonces, ¿cuál es la diferencia entre la variable var y let?



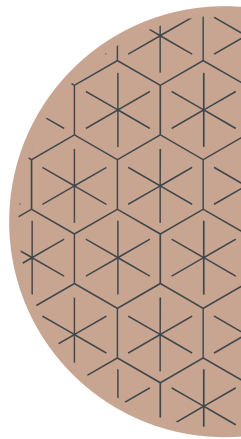
Let

Se parecen bastante, ¿verdad? Veamos el siguiente código, y con suerte, esto te ayudará a entender por qué las variables let son de scope de bloque.

```
function first() {  
  var x = 1; // declare and initialize variable x to 1  
  {  
    var x = 2; // same variable, just we assigned new value to x!  
    console.log(x); // Now x is: 2  
  }  
  console.log(x); // We print the last value assigned to x 2  
}
```

Let

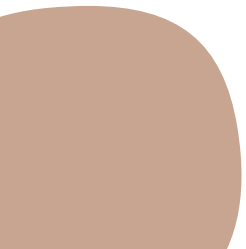
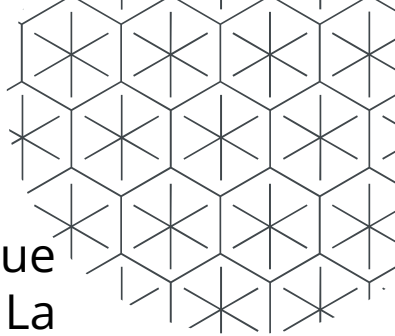
```
function second() {  
  let x = 1; // declare and initialize variable x to 1  
  {  
    let x = 2; // same variable, just we assigned new value to x!  
    console.log(x); // Now x is: 2  
  }  
  console.log(x); // We print the last value assigned to x 1  
}
```



Let

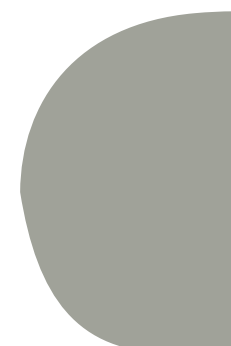

Con la palabra clave `let`, la variable creada dentro del bloque se trata como una variable diferente con un nuevo valor. La variable del bloque es diferente a la variable fuera del bloque. Por lo tanto, tenemos dos resultados diferentes.

Otra característica interesante es que no podemos volver a declarar la misma variable porque arrojará un error de sintaxis.



Let

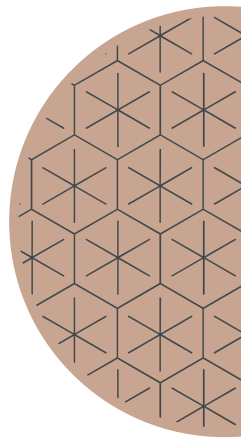
Es importante saber que no podemos leer o escribir en las variables `let` a menos que estén completamente inicializadas; si no se especifica ningún valor durante la declaración, entonces la variable igual a `var` tendrá un valor indefinido. Por lo tanto, si intentamos acceder a ella lanzará un `ReferenceError`.



Const

Si necesitas declarar una variable constante, podemos usar la palabra clave `const` en lugar de `let` o `var`. Ten cuidado cuando uses esta palabra clave `const` porque debes inicializar la constante durante su declaración.

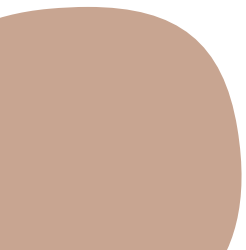
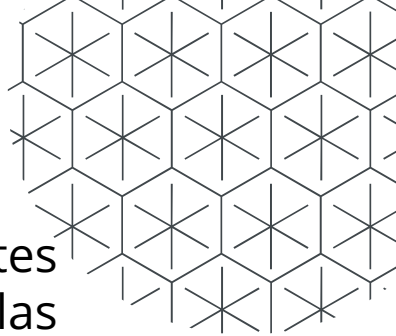
Como su nombre indica, las constantes no pueden cambiar sus valores, y si intentamos modificar el valor, obtendremos un `TypeError`.



Const

Es una buena práctica escribir los nombres de las constantes con letras mayúsculas. De esta manera, puedes distinguirlas fácilmente de las otras variables en tu código.

El scope de la constante es el mismo que el de las variables `let`. Recibirás un error de sintaxis si utilizas el mismo nombre más de una vez dentro del mismo ámbito, así que no hay que volver a declarar las variables `const`. Otro punto importante es que las variables de JavaScript pueden tener un valor de cualquier tipo.



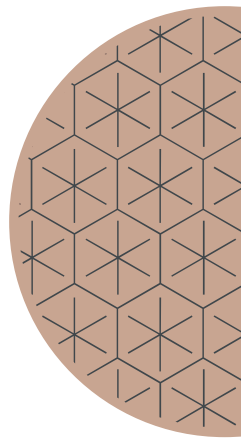
Const

Es importante saber que no podemos leer o escribir en las variables `let` a menos que estén completamente inicializadas; si no se especifica ningún valor durante la declaración, entonces la variable igual a `var` tendrá un valor indefinido. Por lo tanto, si intentamos acceder a ella lanzará un `ReferenceError`.

Literales número

El tipo numérico principal de JavaScript es un Number. Los números de JavaScript pueden ser tan grandes como $\pm 1,7976931348623157 \times 10^{308}$ y tan pequeños como $\pm 5 \times 10^{-324}$.

Todos los números son números de coma flotante de doble precisión. En resumen, esto significa que puedes utilizar 2 o 2,0 sin pensar en obtener un error.



Literales String

Las cadenas no son más que una secuencia de texto encerrada entre comillas. Las comillas pueden ser simples o dobles, pero recuerda que debemos utilizar las mismas comillas al principio y al final de cada texto.

"Hello World"

'Hello World'

`Hello World`

Literales String

Hay un problema con las cadenas. ¿Qué pasa si queremos usar una comilla dentro de una cadena? Para ello, tendremos que escaparla con barra invertida.

Ejercicio) Escapar en la consola del explorador una comilla.

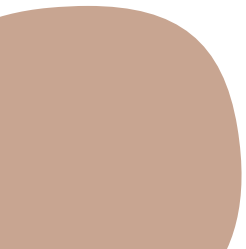
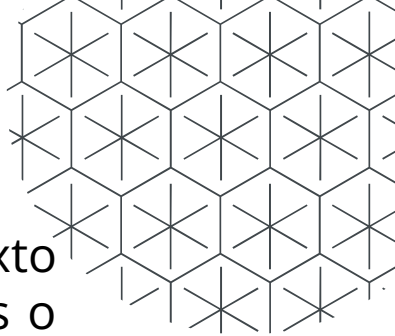
Literales String

Las cadenas no son más que una secuencia de texto encerrada entre comillas. Las comillas pueden ser simples o dobles, pero recuerda que debemos utilizar las mismas comillas al principio y al final de cada texto.

"Hello World"

'Hello World'

`Hello World`



Literales String

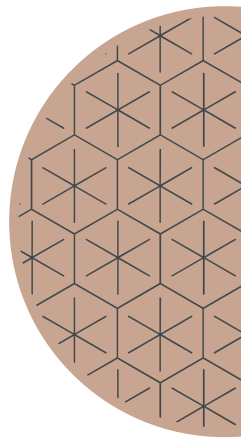
Hay un problema con las cadenas. ¿Qué pasa si queremos usar una comilla dentro de una cadena? Para ello, tendremos que escaparla con barra invertida.

Ejercicio) Escapar en la consola del explorador una comilla.

Literales String

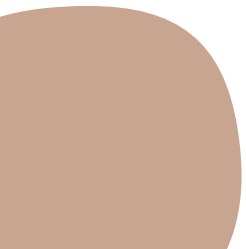
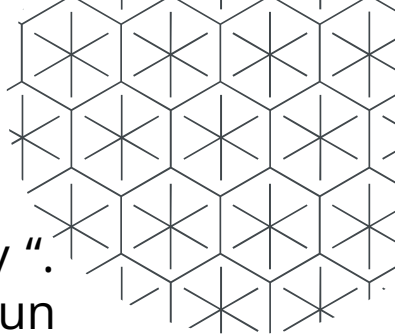
Desde ES6, podemos utilizar backtick que es una cadena de plantilla y soporta la interpolación de cadenas de varias líneas. Porque estamos usando `` backticks.

Usando los puntos suspensivos, podemos fácilmente estirar la cadena en múltiples líneas. Otra característica impresionante es que podemos usar múltiples comillas simples o dobles dentro de la cadena con los puntos suspensivos, y no necesitamos usar el carácter de escape.



Literales String

Ejercicio) Escapar en la consola del explorador usando `` y ``. Concatenar el valor de una variable usando `` y ``. Crear un string que contenga varias líneas usando `` y ``.



Operadores Aritméticos

Al igual que en matemáticas, podemos realizar operaciones matemáticas sencillas en JavaScript. JavaScript utiliza los operadores habituales:

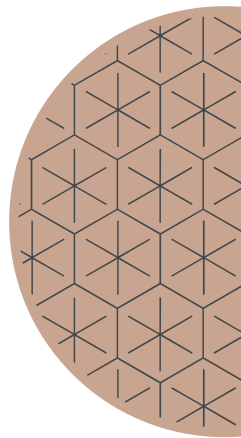
- Suma (+)
- Sustracción (-)
- Multiplicación (*) y División (/)
- ++ incremento (añadir 1)
- -- decremento (restar 1)
- Módulo %
- Exponente **

Literales booleanos

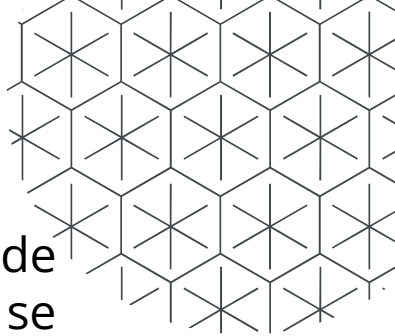
El tipo booleano sólo puede tener uno de estos dos valores:

- Verdadero
- Falso

En JavaScript, tenemos palabras reservadas, y `true` y `false` están reservados como valores booleanos.



Operadores lógicos y de comparación



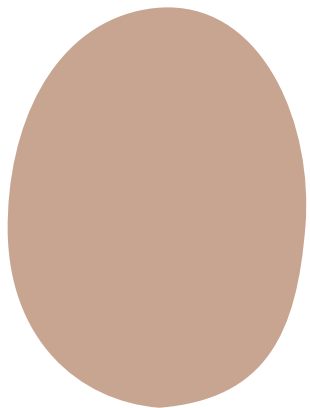
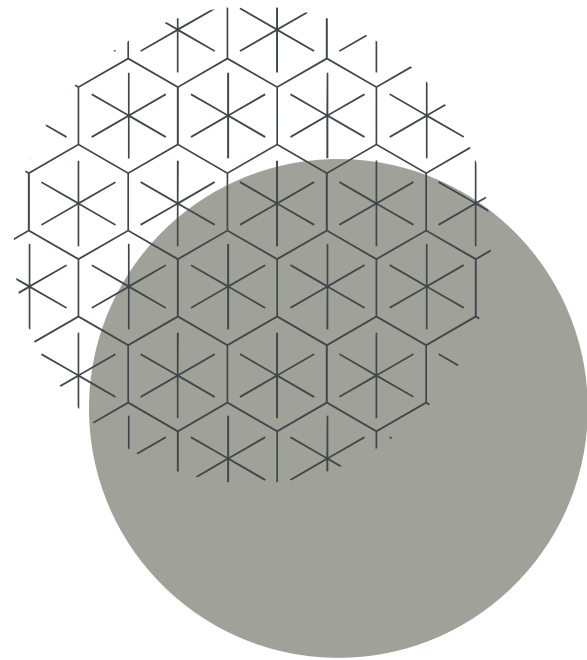
En JavaScript, tenemos operadores llamados operadores de comparación y operadores lógicos. Estos operadores se utilizan para probar valores verdaderos o falsos.

Son los operadores que ya conoces:
==, !=, >, >=, etc.

También están los operador and &&, or || y not !



Objetos



Objetos

Si has aprendido otros lenguajes de programación, entonces tendrás conocimientos sobre los objetos. Sin embargo, los objetos en JavaScript son diferentes, y son uno de los tipos de datos más fundamentales junto con los arrays.

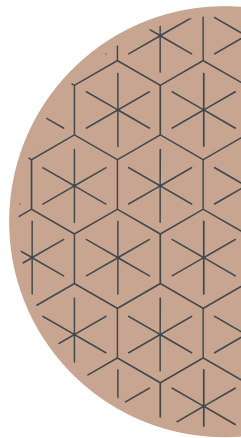
En JavaScript, cada objeto consiste en pares nombre:valor llamados propiedades. Los objetos son tipos de datos no primitivos.

Objetos

Ejemplo de objeto:

```
let user = {  
  userName: 'Rick',  
  age: 33  
};
```

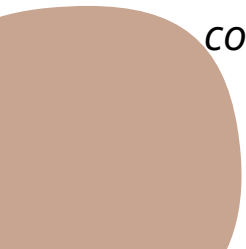
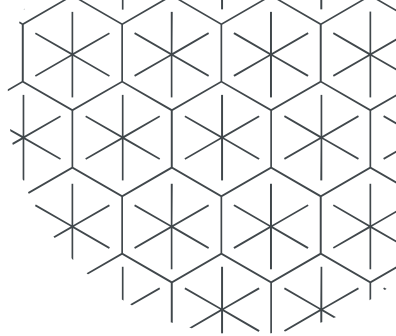
En el ejemplo anterior, en la variable "usuario" almacenamos un objeto con dos propiedades. Así es como creamos un objeto JavaScript con un literal de objeto.



Objetos

```
//Objects
let user = {
  userName: 'Rick',
  age: 33
};
//para acceder a los datos usamos la notación punto y los guardamos en variables
let userName = user.userName;
let userAge = user.age;

//mostramos sus valores
console.log(userName);
console.log(userAge);
```



Crear objetos con new

Hay otra forma de crear un objeto utilizando la palabra clave "new". Este método requerirá más escritura, pero al final, el resultado será el mismo

Crear objetos con new



```
> //Create an object using the new keyword
const newUser = new Object();
newUser.name = 'Rick Sekuloski';
newUser.student = false;
newUser.occupation = 'Web Developer & Programmer';
newUser.students = 20000;
console.log(newUser);
```

▼ {name: "Rick Sekuloski", student: false, occupation: "Web Developer & Programmer", students: 20000} ⓘ

- name: "Rick Sekuloski"
- occupation: "Web Developer & Programmer"
- student: false
- students: 20000

▶ [[Prototype]]: Object


undefined

Crear objetos con new

La primera forma de declarar un objeto como literal es más utilizada entre los desarrolladores. También podemos tener un objeto vacío, un objeto sin propiedades name:value o pares key:value, y esto es una práctica de codificación aceptable.

```
//empty obj  
let studentObj = {};
```

Crear objetos con new



```

> //car object
  let carObj1 = {
    name: 'Ford',
    model: 'Mustang',
    color: 'blue-white',
  }
  let carObj2 = carObj1;
  carObj2.model = 'Fiesta';
  console.log('Car Object1', carObj1);
  console.log('Car Object1', carObj2);

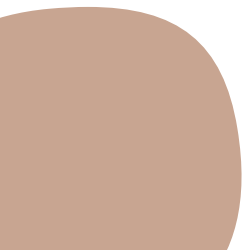
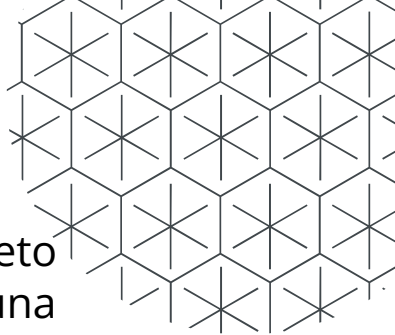
Car Object1 ▶ {name: "Ford", model: "Fiesta", color: "blue-white"}
Car Object1 ▶ {name: "Ford", model: "Fiesta", color: "blue-white"}
< undefined

```

Crear objetos con new

Aquí tenemos dos variables que apuntan al mismo objeto almacenado en algún lugar de la memoria y que sólo mantienen una referencia. Por lo tanto, cambiar el modelo del coche con el segundo objeto también afecta al primer objeto.

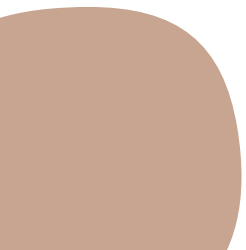
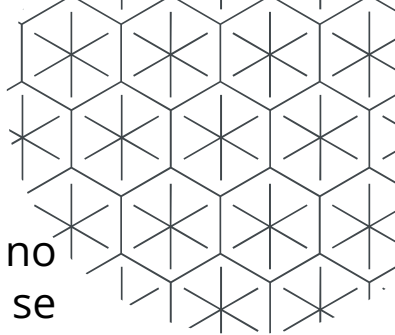
No importa qué objeto estemos usando, todavía estamos cambiando el valor del objeto almacenado en la memoria, y ambos objetos tienen una referencia a ese objeto. Por eso el valor del modelo se cambia de 'Mustang' a 'Fiesta', y cuando imprimimos las dos variables del objeto, obtenemos exactamente el mismo resultado.



Crear objetos con new

Si recuerdas, anteriormente dije que los objetos o valores no primitivos se pasan por referencia, y los tipos de datos primitivos se pasan por valor.

Si te has fijado, el objeto tiene una coma al final de la última propiedad. Esta coma final es muy útil cuando añadimos una propiedad adicional en nuestro objeto, de esta manera siempre estamos seguros de que podemos añadir una nueva propiedad sin causar un error de sintaxis.

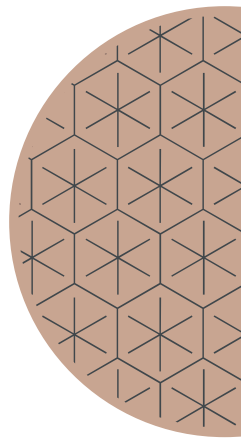


Creación de objetos con `Object.create()`

El método `Object.create()` crea un nuevo objeto utilizando su primer argumento como el prototipo del objeto recién creado.

Crear objetos con new

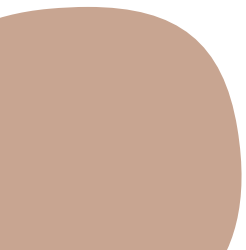
```
const person = {  
  isHuman: false,  
  myDetails: function() {  
    console.log(My full name is ${this.name} ${this.lastName}. Am I human?  
    ${this.isHuman});  
  }  
};  
const rick = Object.create(person);  
rick.name = 'Rick'; rick.lastName = 'Sekuloski';  
rick.isHuman = true;  
rick.myDetails();
```



Crear objetos con new

También se puede crear un nuevo objeto que no herede nada, lo cual es malo porque aún queremos utilizar algunos de los métodos básicos que ofrece `Object.prototype`.

```
const ob4 = Object.create(Object.prototype);
```



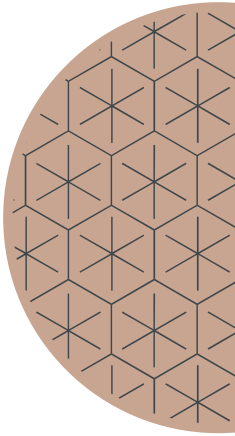
Primitivos pasados como valor

Cuando tenemos un tipo primitivo asignado a una variable, entonces esa variable contiene el valor primitivo real, no la referencia a la memoria sino la ubicación exacta a la memoria.

Esto es totalmente opuesto a los objetos, ya que las variables de los objetos sólo contienen una referencia al lugar donde se almacena el objeto en la memoria, pero no el valor real.

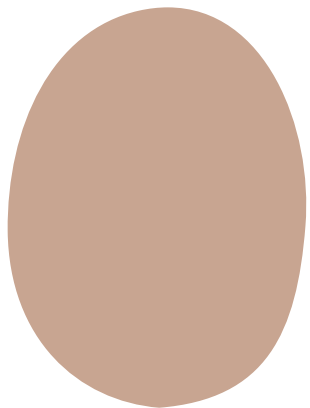
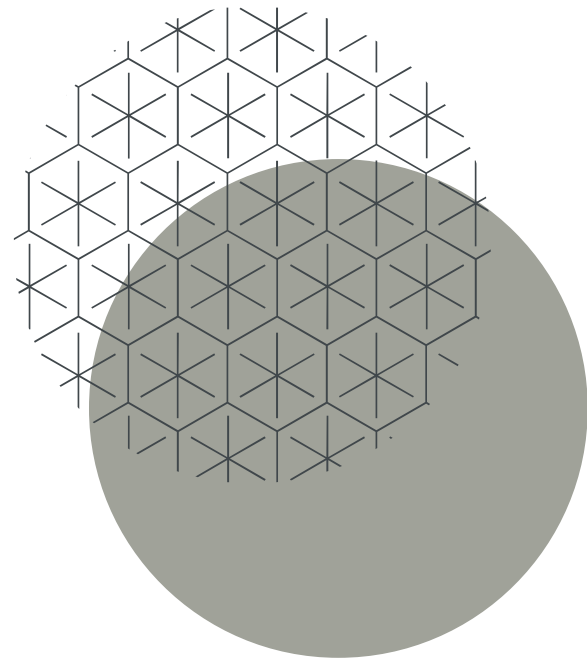
Crear objetos con new

```
let a = 5;  
let b = 'name';  
let c = null;  
c = a;  
a = 10;  
console.log('Value of a:',a);  
console.log('Value of c:',c);
```





Arrays



Arrays

Los arrays en JavaScript son el segundo tipo de datos más importante. En Javascript usamos los arrays para almacenar multitud de valores dentro de una misma variable.

Los arrays como las variables **no** son tipados.

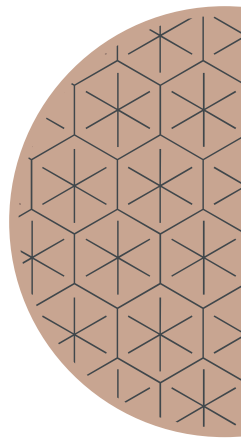
```
const myArray = [1,2,3,4,5,6];  
const namesArray = ['Tom','Andy','Jason','Chris','Loki','Luk'];  
const mixedArray = [1,true,'Andy',{x:1,y:2}];
```

Arrays

Ejercicio) ¿Qué pasa si ponemos dos comas seguidas al crear un array? ¿Qué tamaño tendrá?

Crear array usando new Array()

Otra forma de crear un array es utilizar la palabra clave new. Esto utilizará el constructor Array, y podemos llamar o invocar este constructor de tres maneras diferentes.



Crear array usando new Array()

//empty array

let array1 = new Array();

console.log(array1.length);//0

//single argument constructor

let array2 = new Array('Tom');

console.log(array2.length);//1

//multiple argument constructor

let array3= new Array('Tom','Jerry')

console.log(array3.length);//2

Operador Spread

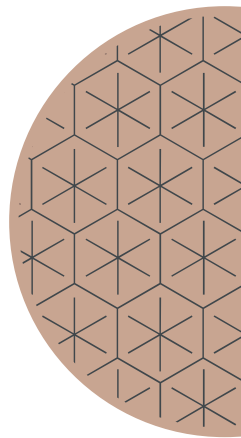
Desde ES2015 o ES6, podemos utilizar el 'operador spread' para crear un nuevo array o incluso para copiar un array a otro array. La sintaxis del operador de propagación es de tres puntos ...

```
const firstArray = [1,2,3,4,5];  
console.log('The elements from the first array '+ firstArray);  
console.log('The length of the first is: '+ firstArray.length);  
//Use the spread operator to create new array  
const secondArray = [...firstArray];  
console.log('The elements of the second array'+ secondArray);  
console.log('The length of second array is: '+ secondArray.length);
```


Acceso elementos array

Para acceder a los elementos del array, necesitas usar los índices. Los índices de los arrays están basados en el cero.

```
//multiple argument constructor  
let array3= new Array('Tom','Jerry')  
console.log(array3.length);//2  
console.log(array3[0]);//Tom
```

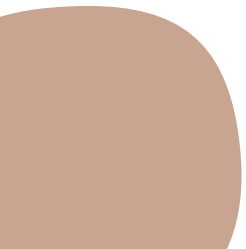




Sentencias programación

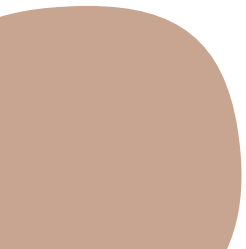
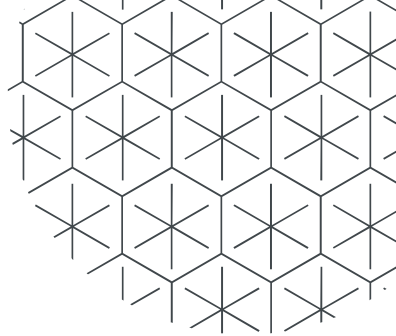
Sentencias Condicionales

Como su nombre indica, las sentencias condicionales necesitan tener una condición para funcionar. En función de la condición, podemos omitir o ejecutar un conjunto de sentencias. Estas sentencias también se conocen como ramas. En JavaScript, tenemos declaraciones de rama if/else y switch.



Sentencias Condicionales

```
age = 18;  
if(age <= 14){  
    console.log('You are in the children group');  
}  
else if( age >= 15 && age <= 24){  
    console.log('You are in youth group');  
}  
else if( age >= 25 && age <= 64){  
    console.log('You are in Adults group');  
}  
else{  
    console.log('You are in Senior group');  
}
```

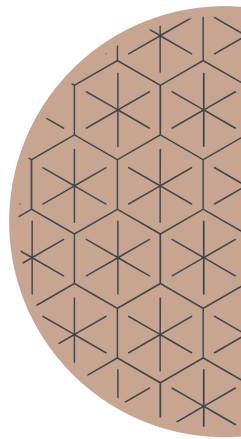


Condicional ternario

El operador ternario es como una versión abreviada de la sentencia if. La sintaxis de este operador:

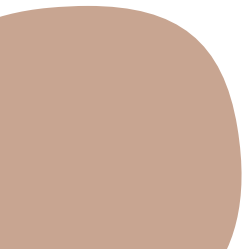
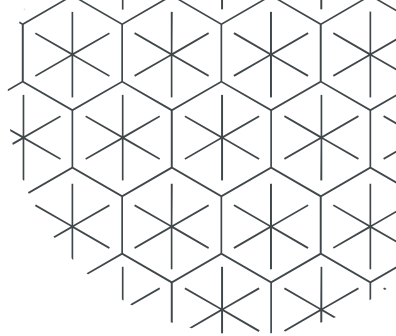
condición ? expresiónIfTrue : expresiónIfFalse

El primer operando es la condición, una expresión cuyo valor se utiliza como condición, y el segundo operando después son las sentencias que se ejecutan si es cierto o falso.



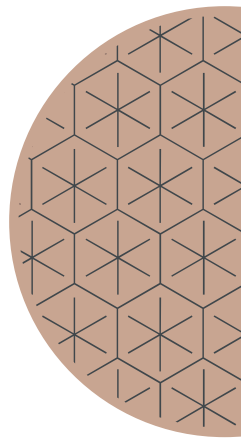
Condicional ternario

```
//Conditional (ternary) operator  
let age = 26;  
let whichBevarage = (age >= 21) ? "Beer" : "Juice";  
console.log(whichBevarage); // "Beer"
```



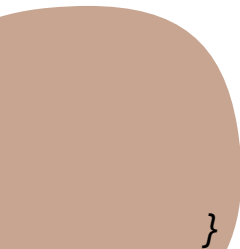
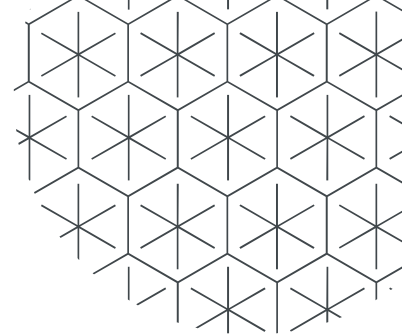
Sentencias Switch

Hasta ahora, hemos cubierto las sentencias if, if-else, y else-if para permitir que el programa ejecute un bloque diferente de sentencias basado en la condición dada o en múltiples condiciones como tenemos en la sentencia else-if. Sin embargo, JavaScript tiene una sentencia adicional que es muy popular, y se llama sentencia switch.



Sentencias Switch

```
let x = 6;  
switch(x){  
  case 1: // if x === 1  
    //Execute the statements for case #1  
    break; // Stop  
  case 2: // if x === 2  
    //Execute the statements for case #2  
    break; // Stop  
  case 3: // if x === 3  
    //Execute the statements for case #3  
    break; // Stop  
  case 4: // if x === 4  
    //Execute the statements for case #4  
    break; // Stop  
  default:  
    //Execute the statements  
    break;  
}
```



Operador asignación

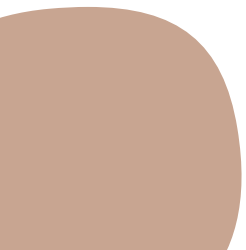
Hasta ahora hemos utilizado el operador de asignación. Por ejemplo, utilizamos este operador cuando asignamos un valor a una variable. El operador de asignación es este (=). También existen otros:

Operator	Description
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=	divide and assign

Bucle While

Mientras se cumple una condición (o es verdadera), el bucle `while` ejecuta repetidamente una o varias sentencias.

```
//While loop  
let i = 0;  
while (i < 10) {  
  i++;  
  console.log(i);  
}
```



Bucle do While

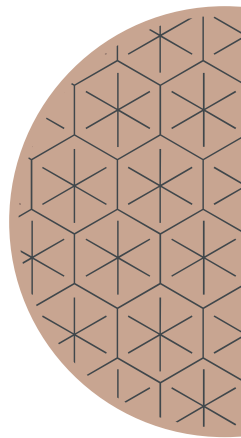
El bucle while nunca ejecutará su código si la condición es falsa, pero hay situaciones en las que necesitamos que ese código se ejecute al menos una vez, y para ello podemos utilizar el bucle do/while.

```
let i = 11;  
do{  
  console.log('The value of i:', i);  
}while(i < 10);
```

Bucle for

Uno de los bucles más utilizados y favoritos entre los desarrolladores es el bucle for. El bucle for tiene 3 declaraciones separadas.

```
For (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```

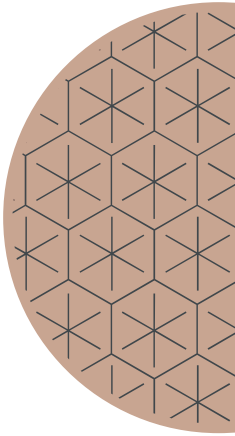


Bucle for

Statement 1: establece una variable antes de que comience el bucle. En la mayoría de los lenguajes de programación, esta variable es 'i' y siempre comienza desde cero.

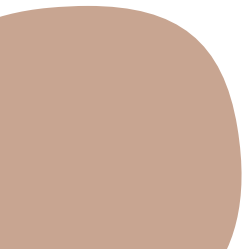
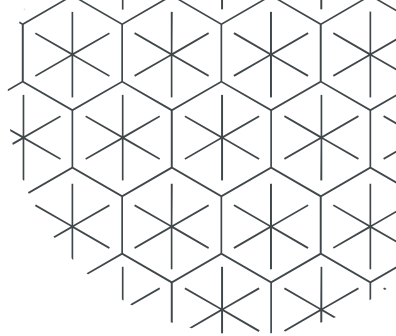
Statement 2: define la condición para que el bucle se ejecute. El bucle debe ser verdadero.

Statement 3: aumenta el valor de i (i++) cada vez que el bloque de código en el cuerpo del bucle se ha ejecutado.



Bucle for

```
let numbersArray = [1,2,3,4,5,6,7,8,9,10];  
for(let i = 0; i < numbersArray.length; i++){  
  console.log(numbersArray[i]);  
}
```

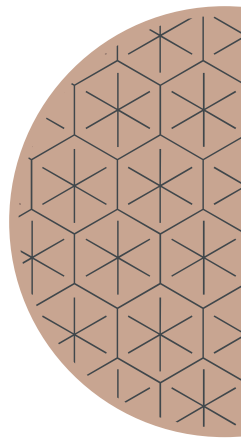


Bucle for con objetos

Previamente hemos visto los objetos y ahora veremos cómo acceder a ambos pares clave/valor con el bucle for/of.

Los objetos no son lo mismo que los arrays, y no podemos recorrer los objetos igual que los arrays porque nos dará un error. Para recorrer como un objeto, podemos utilizar los siguientes métodos:

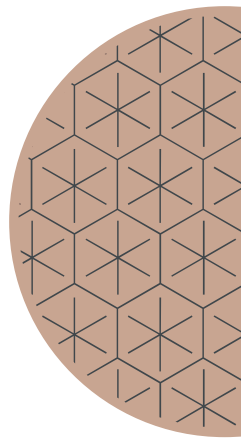
- `Object.keys();`
- `Object.entries();`



Object.keys() - for/of

Object.keys() es un método que devuelve un array con los nombres de las propiedades.

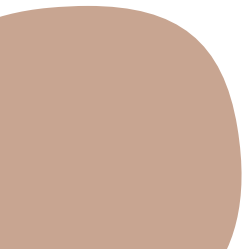
```
let propertyNames = "";  
for(let names of Object.keys(testObject)) {  
  propertyNames += names;  
}  
console.log(propertyNames); // => "abcd"
```



Object.entries() – for/of

Con el método `Object.keys()`; estamos obteniendo los nombres de las propiedades de un objeto, pero qué pasa si necesitamos los dos pares clave/valor de ese objeto.

Si queremos algo así, tenemos que utilizar `Object.entries()`. Este método nos dará acceso a ambos pares de claves y valores.



Object.entries() – for/of

```
let testObject1 = {  
  a: 1,  
  b: 2,  
  c: 3,  
  d: 4,  
}  
for(let [k,v] of Object.entries(testObject1)) {  
  console.log('Key: '+k);  
  console.log('Value: '+v);  
}
```

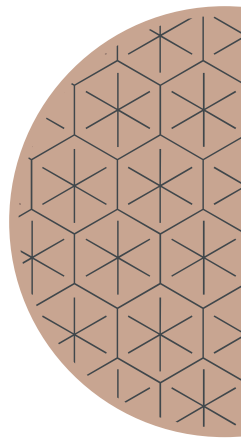
Bucle for/in

La sentencia for/in realiza un bucle a través de las propiedades de un objeto.

```
for (variable in object) {  
statement  
}
```

Repasemos la sintaxis rápidamente:

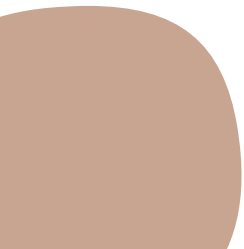
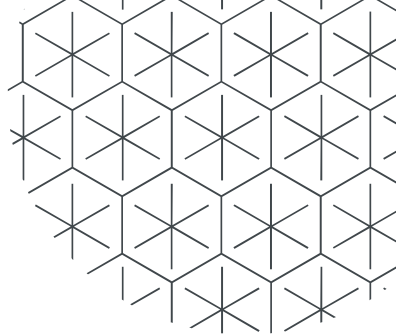
- **Variable:** se asigna un nombre de propiedad diferente a la variable en cada iteración.
- **Objeto:** objeto cuyas propiedades son iteradas.



Bucle for/in

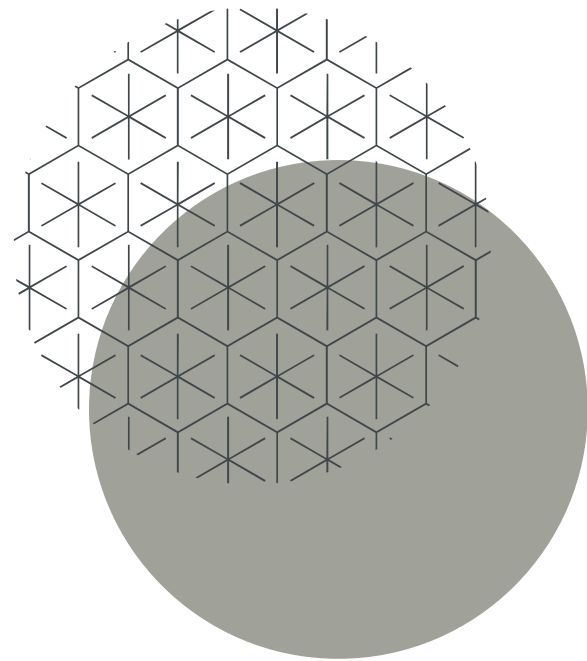
```
let objPerson = { firstName: 'Denzel', lastName: 'Washington', age: 67 }
```

```
for (const key in objPerson){  
  console.log(Key: ${key}, value: ${objPerson[key]})  
}
```





Funciones



Funciones

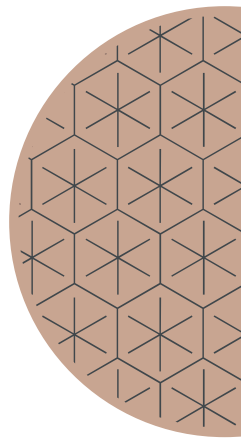
En JavaScript, hay diferentes maneras de declarar una función, y vamos a repasarlas. La forma más común de declarar la función es utilizando la palabra clave `function` seguida de paréntesis, donde escribimos los nombres de los parámetros.

Finalmente, el cuerpo de la función es donde podemos tener declaraciones simples o múltiples. El cuerpo de la función está entre las llaves, pero si tenemos una sola línea de código, como ya sabes, podemos omitirlas.

Funciones

```
function name (parameter1,...parameter2) {  
  //function body  
}  
function total (x, y) {  
  return (x + y);  
}
```

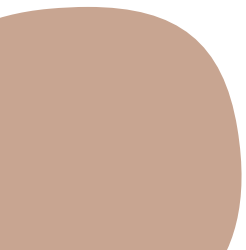
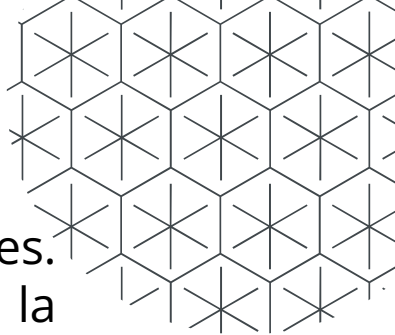
El nombre se conoce como identificador de la función. Dentro de los paréntesis, proporcionamos los parámetros. En nuestro caso, tenemos dos, pero pueden ser muchos siempre que estén separados por una coma.



Funciones

Los tipos de los parámetros tampoco son importantes. Finalmente, en el cuerpo, devolvemos su suma utilizando la sentencia return. Las sentencias en el interior son el cuerpo de la función, y se ejecutan sólo cuando la función es llamada o invocada.

La sentencia return en el cuerpo de la función da el valor que la función devolverá, y, en nuestro caso, será la suma de los valores de los dos parámetros x e y.



Expresión función

En JavaScript, existe otra sintaxis para crear una función, y se llama expresión de función.

```
let greeting = function(name,lastName){  
  return `Nice to see you again ${name} ${lastName}`;  
}
```

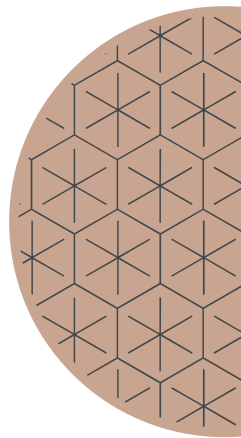
De esta manera podemos crear una función anónima que no tiene nombre de función. La expresión de función se almacena en una variable y puede ser invocada/llamada usando ese nombre de variable.

Expresión función

//invoke function expression

let theMessage = greeting('Luke', 'Perry');

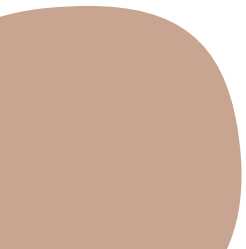
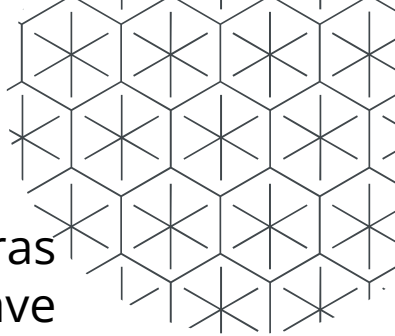
console.log(theMessage);



Función flecha

Al principio de este capítulo, dije que hay diferentes maneras de declarar una función, una de ellas es usar la palabra clave `function`, y la segunda es usar el operador de flecha `'=>'`.

Esta es una nueva feature de ES6 y significa una sintaxis abreviada para la declaración de funciones. El uso de una función de flecha viene con ventajas y algunas desventajas también. La expresión de función de flecha es una alternativa a la expresión de función tradicional.



Función flecha

// Traditional Function

```
function addFun (a){  
  return a + 100;  
}
```

// Arrow Function

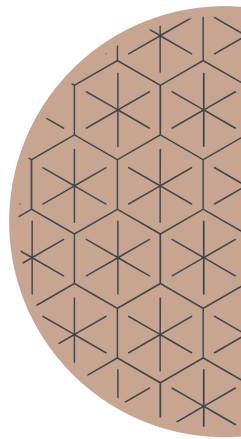
```
let addFun = a => a + 100;
```

Función flecha

Las funciones de flecha son mucho más limpias y cortas, pero repasemos su sintaxis.

Primero proporcionamos las variables de los parámetros en el lado izquierdo del operador flecha, y en el lado derecho del operador, devolvemos el valor de la función.

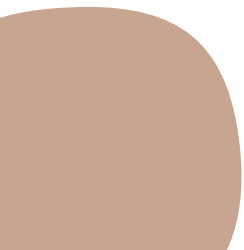
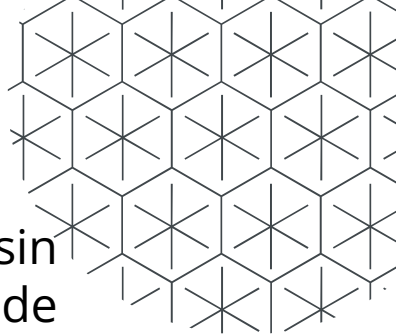
Si hay un único parámetro 'a' como tenemos en el ejemplo anterior, no necesitamos encerrar ese parámetro entre paréntesis como hacíamos en la declaración de función tradicional '(a)'.



Función flecha

Sabemos que una función tradicional puede ser sin parámetros y podemos hacer lo mismo con la función de flecha:

```
let noParams= () => 'No Parameter function';  
let storeResult = noParams();  
console.log(storeResult)
```



Función flecha

Si el cuerpo requiere varias líneas de código, entonces tenemos que utilizar las llaves más la declaración de retorno porque la función de flecha no sabe qué o cuándo queremos que la función regrese.

Función flecha

// Traditional Function

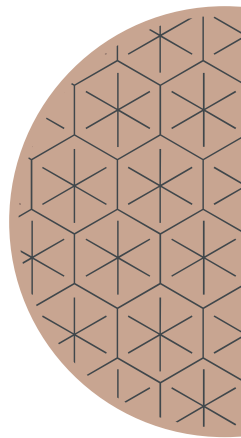
```
function additionFn (x, y){  
  let result = x + y;  
  return 'This is the result :'+ result;  
}
```

```
let store1 = additionFn(3,4);  
console.log(store1);
```

// Arrow Function

```
let additionFn1 = (x, y) => {  
  let result = x + y;  
  return 'This is the result :'+ result;  
}
```

```
let store2 = additionFn1(3,4);  
console.log(store2);
```



Función flecha en arrays

Hasta ahora para iterar sobre arrays hemos utilizado algunas de las siguientes sentencias como: bucles for, for/in y for/of, pero ya que hemos cubierto las funciones de flecha, podemos utilizar el método `forEach` para iterar tanto por los elementos como por los índices del array.

```
console.log('Printing the elements of the array, its index positions and the entire array');  
carBrands.forEach((element,index,array) => {  
    console.log(element , index, array);  
});
```

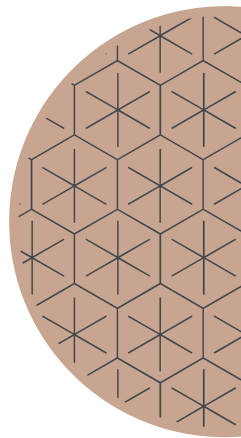
Pasar argumentos a funciones

Hasta ahora, hemos aprendido mucho sobre las funciones. Sabemos cómo definir/declarar una función con parámetros específicos e invocar la misma función en nuestro programa.

Sea la siguiente función:

Pasar argumentos a funciones

```
let a = 5;  
let b = 7;  
function findMax(x,z){  
  if(x > z){  
    return x;  
  }  
  else{  
    return z;  
  }  
}  
let biggerOne = findMax(a,b);  
console.log(biggerOne);
```



Pasar argumentos a funciones

Ejercicio) ¿Qué pasará si paso más argumentos que los que la función tiene?

Ejercicio) ¿Qué pasará si paso menos argumentos que los que la función tiene?

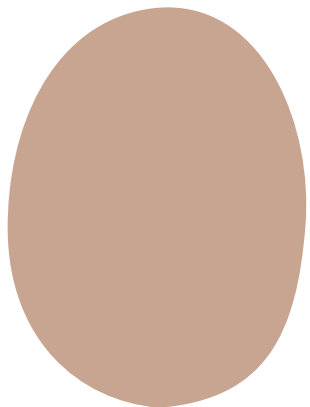
Ejercicio) ¿Qué pasará si paso como parámetro un valor null o undefined?

Funciones con valores por defecto

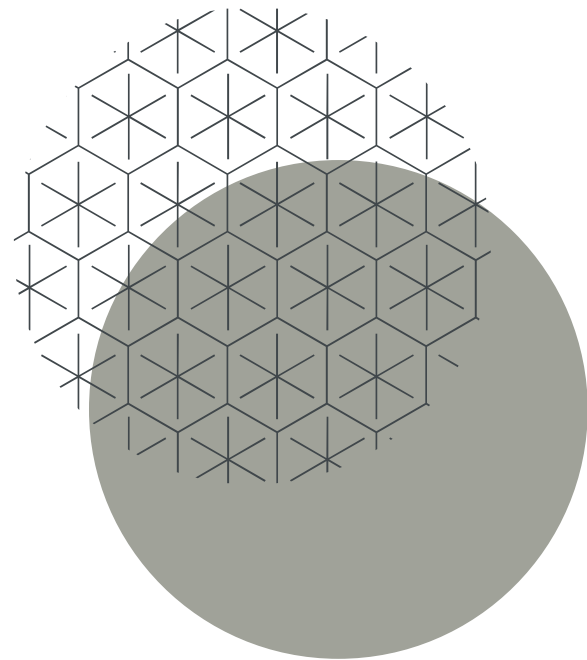
Es muy sencillo proporcionar argumentos por defecto en la declaración de la función.

```
function multiplyFn(a, b = 1) {  
  return a * b;  
}
```

```
let result1 = multiplyFn(5,2);  
console.log(result1);// output: 10  
let result2 = multiplyFn(5);  
console.log(result2);// output: 5
```

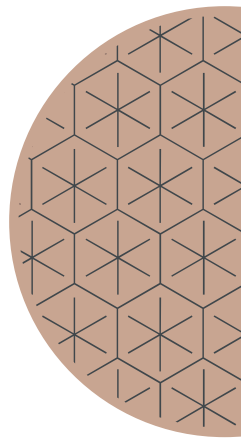


POO



POO - Clases

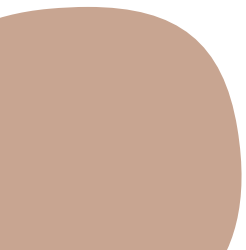
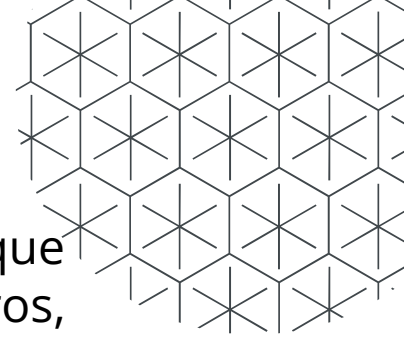
En esta sección, veremos en detalle por qué llamamos a JavaScript un lenguaje orientado a objetos. Pero, ¿qué es la programación orientada a objetos? Si has aprendido un lenguaje de programación como Java, entonces sabes que Java es un lenguaje que utiliza Objetos. Además, en Java, puedes crear clases, y puedes instanciar objetos de esas clases.



POO - Clases

Pero Java no es un lenguaje orientado a objetos puro porque también tiene tipos de datos primitivos como enteros, booleanos, long, etc. pero utiliza las principales características orientadas a objetos como:

- Herencia
- Encapsulación
- Polimorfismo
- Asociación
- Agregación



POO - Clases

Para algunos desarrolladores, entender JavaScript es más complejo que otros lenguajes. Como ya sabes los objetos de JavaScript, son un poco diferentes en comparación con los objetos que probablemente has visto en otra programación orientada a objetos como C++ o Java.

Es confuso cómo puedes tener métodos o clases o incluso herencia en JavaScript. La sintaxis y la definición de las clases son similares a las de Java, pero lo que sucede bajo es bastante diferente.

Classes

//Classes in JavaScript

class Person {

constructor(firstName, lastName, age, position, country){

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.position = position;

this.country = country;

}

printDetails() {

console.log(Hi! My details are:

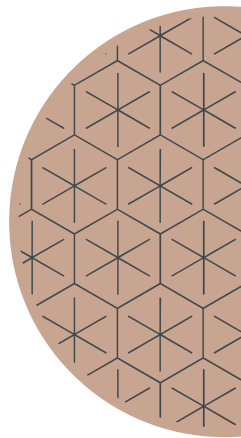
Name: \${this.firstName},

Name: \${this.lastName},

);

}

}



Clases

En la clase, tenemos un constructor (), y esta palabra clave declara el cuerpo de la función constructora Person. Las clases de JavaScript no son más que azúcar sintáctico para una función constructora y un objeto prototipo. Una cosa importante a mencionar es cómo podemos instanciar un objeto desde una clase.

```
const samuel = new Person('Samuel','L Jackson',59, 'Actor','America');  
samuel.printDetails();
```

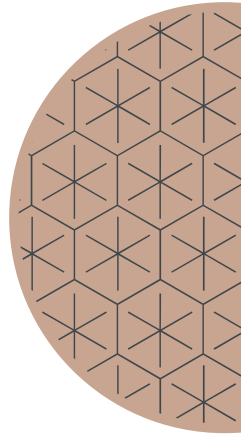
Clases

Desde ES6; se recomienda utilizar la sintaxis de las clases en lugar de las clases basadas en prototipos de JavaScript.

Para usarlas tenemos que recordar que hay que utilizar la palabra clave `class` seguida del nombre de la clase, que debe comenzar con una letra mayúscula. La clase tiene su cuerpo, y eso se marca con los corchetes. Dentro de la clase, tenemos la palabra clave `constructor`, que es la misma ya que estamos definiendo la función constructora.

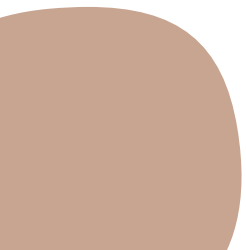
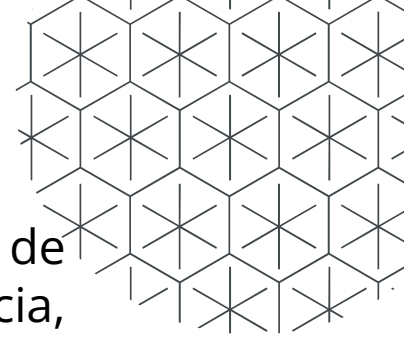
Clases

En otros ejemplos, encontrarás que no necesitamos definir el constructor en absoluto. Esto significa que no tenemos variables que necesiten ser inicializadas y con valores, pero porque omitamos la palabra clave constructor no significa que no habrá un constructor para esta clase. Una función constructora vacía será creada implícitamente para nosotros en el fondo.



Herencia

Uno de los temas más candentes en cualquier lenguaje de programación OOP es la herencia. Para hablar de herencia, necesitamos tener una clase padre, y queremos que los hijos de esa clase obtengan toda la funcionalidad y propiedades que la clase padre tiene para ofrecer, pero lo que es más importante es que los hijos de esa clase puedan añadir sus propias funcionalidades y propiedades nuevas.



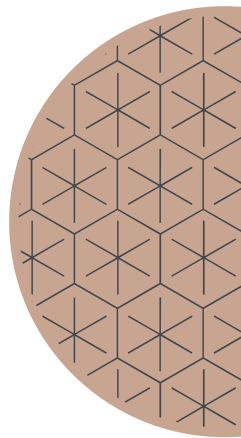
Herencia

Ahora vamos a ver cómo vamos a lograr la herencia en JavaScript a través de ejemplos.

Digamos que tenemos una clase llamada Animal. Dentro del constructor de la clase, estableceremos e inicializaremos dos propiedades del animal (nombre y velocidad). También tendremos dos métodos que son muy "genéricos", y que todo animal puede hacer.

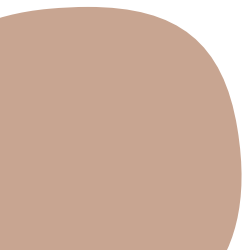
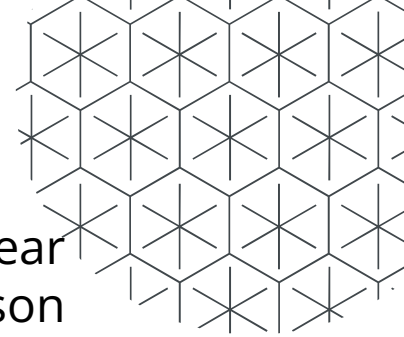
Herencia

```
class Animal {  
  constructor(name) {  
    this.speed = 0;  
    this.name = name;  
  }  
  startRunning(speed) {  
    this.speed = speed;  
    return The ${this.name} runs with speed of ${this.speed}km per hour.;  
  }  
  stopRunning() {  
    this.speed = 0;  
    console.log(The ${this.name} stopped running and now sits still.);  
  }  
}
```



Herencia

Después de declarar esta clase básica Animal, podemos crear otra clase, por ejemplo, Perro. Los perros también son animales, y deben estar basados en la clase Animal, lo que significa que pueden acceder a los métodos y propiedades genéricas de los animales, pero también los perros pueden hacer algo que otros animales no pueden. Para heredar propiedades de su padre, esta es la sintaxis que debemos utilizar



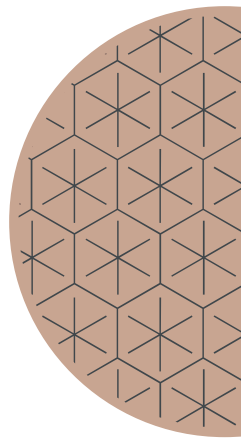
Herencia

```
class Dog extends Animal {  
  bark() {  
    return The ${this.name} barks!;  
  }  
}  
let dog = new Dog("Dog");  
console.log(dog.startRunning(10));  
console.log(dog.bark()); //The Dog barks!
```

Herencia

En este ejemplo, el objeto que instanciamos de la clase Dog tiene acceso tanto a los métodos de Dog como a los de Animal, como startRunning y stopRunning.

Bajo el capó, la palabra clave extends funciona utilizando el prototipo, por lo que establece el prototipo `Dog.prototype.[[Prototype]]` a `Animal.prototype`. Así, si el objeto solicita un método que no se encuentra en `Dog.prototype`, empezará a buscar en el de `Animal.prototype`.



Propiedades y métodos Static

En JavaScript, también podemos asignar un método a la función de la clase, no a su prototipo. Estos métodos se llaman métodos estáticos. Para declarar un método estático, debemos utilizar la palabra clave `static`. Los métodos estáticos se utilizan para implementar funciones que pertenecen a la clase pero no al objeto.

Propiedades y métodos Static

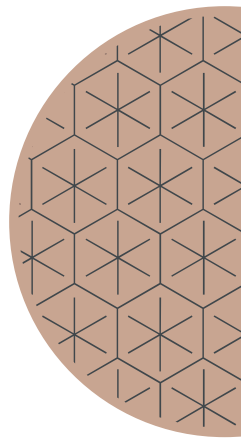
```
class User{  
  constructor(name){  
    this.name = name;  
  }  
  static staticMethod(){  
    console.log(this.name);  
  }  
}
```

```
User.staticMethod();
```

Overriding métodos

Recuerda de la sección de herencia que cuando las instancias de perro no podían encontrar el método de la clase Dog, empezaban a buscar en la superclase padre para ese método en particular. En nuestro caso, esta era la clase Animal.

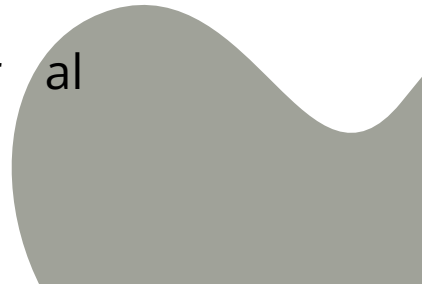
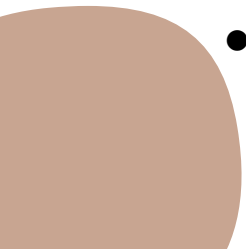
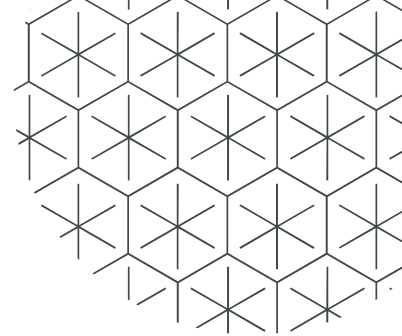
Así que, si añadimos otro método en la clase Dog con el mismo nombre que en la clase Animal, el proceso se llama override de métodos. ¿Por qué? El nombre se explica por sí mismo, significa que tenemos un método con el mismo nombre en la clase padre y en la clase hija.



Overriding métodos

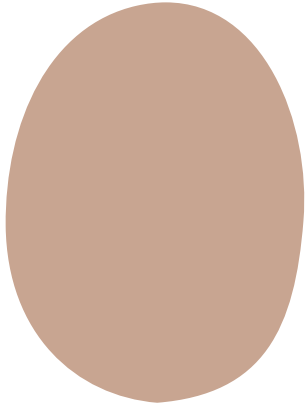
```
class Dog extends Animal {  
  stopRunning (){//this method will be used by the instances of the Dog class  
    super.stopRunning();  
    console.log('The dog is licking its paws');  
  }  
  bark() {  
    return `The ${this.name} barks!`;  
  }  
}
```

- `super (...)`, podemos usarlo para llamar al constructor padre (necesitamos usar esta palabra clave dentro del constructor hijo)
- `super.method (...)`, podemos usarlo para llamar al método declarado en el padre.





Modo strícto



Modo stricto

En JavaScript, tenemos algo llamado modo estricto, y fue introducido en ES5. Este modo es la forma de optar por la versión restringida de JavaScript, y el único propósito es indicar que nuestro código debe ser ejecutado en modo estricto.

Con este modo estricto, podemos detectar fácilmente los errores ocultos y depurar nuestro código fácilmente.

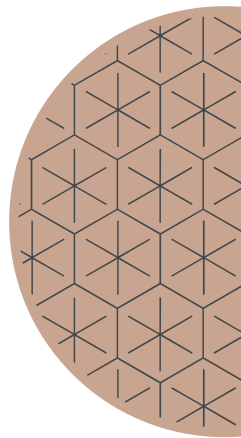
Modo stricto

Bien, para habilitar este modo en JavaScript, todo lo que tenemos que hacer es escribir esta única línea de código en la parte superior del archivo:

```
'use strict';
```

Se pueden utilizar las comillas simples o dobles sin ningún problema. Si queremos utilizar este modo en un entorno Node.js, entonces tenemos que escribir esto en la terminal

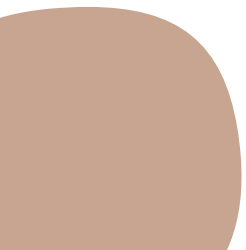
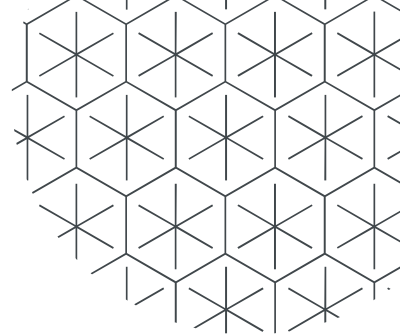
```
node --use-strict
```



Modo estricto

Beneficios de usar el modo estricto:

- El modo estricto nos asegurará que estamos escribiendo un código "seguro" y libre de errores
- El modo estricto resolverá los errores lanzando nuevos errores
- El modo estricto puede ser ejecutado más rápido que el código no estricto
- El modo estricto evitará los errores
- El modo estricto desactiva las características inusuales de JavaScript



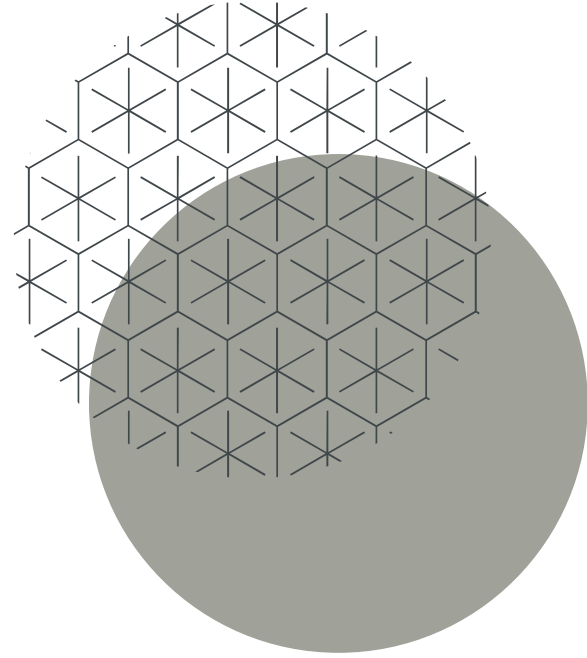
Modo stricto

```
> a = 10;  
  let addTwo = function(){  
    'use strict';  
    b = 15;  
    return a + b;  
  }  
  let result = addTwo();  
  console.log(result);
```

✖ ▶ Uncaught ReferenceError: b is not defined
 at addTwo (<anonymous>:4:7)
 at <anonymous>:7:14



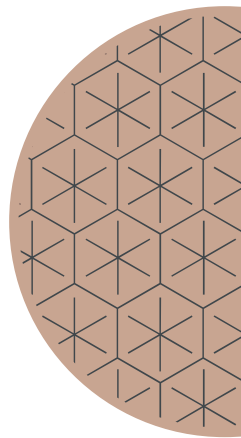
This



¿Qué es this?

Cuando no usamos el modo estricto entonces 'this' hace referencia al objeto global si llamamos a una función como esta

```
//function 'this' reference'  
function message() {  
  console.log(this === window); // true  
}  
message(); // Output: true
```



¿Qué es this?

En el ejemplo anterior, llamar a la función `message()` es lo mismo que usar `window.message()`.

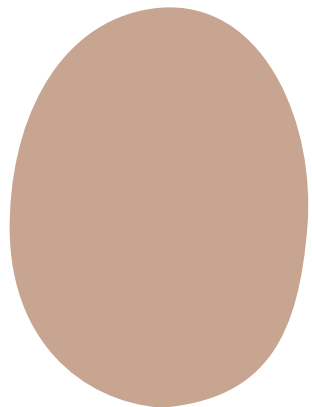
Cuando utilizamos el modo estricto, esta palabra clave se establece como indefinida.

```
//function using the strict mode  
'use strict';  
function newMessage() {  
  console.log(this === undefined);  
}  
newMessage(); //true
```

¿Qué es this?

```
//method invocation
let animal = {
  type: 'Mammals',
  getType: function () {
    return this.type;
  }
}
console.log(animal.getType()); // Mammals
```

En este ejemplo, este objeto en el método `getType()` hace referencia al objeto `animal`.



FIN

