

# Clase 14:

# Scope y Objetos

# Índice

## Funciones Arrow:

- Concepto.
- Estructura básica.
- Ejemplo en código

## Objetos Literales:

- Concepto.
- Estructura básica.
- Ejemplo en código



# Scope

## IMPORTANT

El scope o ámbito se refiere al alcance que tiene una variable, es decir, desde dónde podemos acceder a ella.

En JavaScript los scopes son definidos principalmente por las funciones.

# Scope local

En el momento en que declaramos una variable dentro de una función, esta pasa a tener alcance local. Es decir, esa variable vive únicamente dentro de esa función.

Si quisiéramos hacer uso de la variable por fuera de la función, no vamos a poder, dado que fuera del **scope** donde fue declarada, esa variable no existe.



```
function saludar() {
```

```
    // todo el código que escribamos dentro
```

```
    // de nuestra función, tiene scope local
```

```
}
```

```
// No podremos acceder desde afuera a ese scope
```



# Scope Local - Ejemplo

{código}

```
function hola() {  
  let saludo = 'Hola ¿qué tal?';  
  return saludo;  
}
```

```
console.log(saludo);
```

Definimos la variable saludo dentro de la función hola(), por lo tanto su **scope** es **local**.

Solo dentro de esta función podemos acceder a ella.

# Scope Local - Ejemplo

{código}

```
function hola() {  
    let saludo = 'Hola ¿qué tal?';  
    return saludo;  
}
```

```
console.log(saludo);  
// saludo is not defined
```

Al querer hacer uso de la variable saludo por fuera de la función, JavaScript no la encuentra y nos devuelve el siguiente error:

Uncaught ReferenceError:  
saludo is not defined

# Scope global

En el momento en que declaramos una variable **fuera** de cualquier función, la misma pasa a tener **alcance global**.

Es decir, podemos hacer uso de ella desde cualquier lugar del código en el que nos encontremos, inclusive dentro de una función, y acceder a su valor.



```
// todo el código que escribamos fuera  
// de las funciones es global  
function miFuncion() {  
    // Desde adentro de las funciones  
    // Tenemos acceso a las variables globales  
}
```





# Scope Global - Ejemplo

{código}

```
let saludo = 'Hola ¿qué tal?';
```

```
function hola() {  
    return saludo;  
}
```

```
console.log(saludo);
```

Declaramos la variable saludo por fuera de nuestra función, por lo tanto, su scope es global.

Podemos hacer uso de ella desde cualquier lugar del código.

# Scope Global - Ejemplo

{código}

```
let saludo = 'Hola ¿qué tal?';
```

```
function hola() {  
    return saludo;  
}
```

```
console.log(saludo);  
// 'Hola ¿qué tal?'
```

Dentro de la función hola() llamamos a la variable **saludo**.

Su alcance es **global**, por lo tanto, JavaScript sabe a qué variable nos estamos refiriendo y ejecuta la función con éxito.

# Veamos un Ejemplo en VSC!



# Funciones Tipo Flecha

# Las Arrow Functions

## IMPORTANT

Son una forma de crear funciones incorporadas a partir de ES6 (ECMAScript versión 6).

Una de sus ventajas es que son más concisas que las funciones clásicas creadas con la palabra reservada `function`.

Lo que en la forma clásica llevaba 3 líneas de código, con las arrow functions lo podés resolver en una sola.

# Estructura básica

Pensemos en una función simple que podríamos programar de la manera habitual: una suma de dos números.



```
function sumar (a, b) { return a + b }
```

Ahora veamos la versión reducida de esa misma función, al transformarla en una función arrow.



```
let sumar = (a, b) => a + b;
```

Notan algo raro? Falta algo?



# Nombre de una función arrow

Las funciones arrow son siempre anónimas. Es decir, que no tienen nombre como las funciones normales.



```
(a, b) => a + b;
```

Si queremos nombrarlas, es necesario escribirlas como una función expresada. Es decir, asignarla como valor de una variable.



```
let sumar = (a, b) => a + b;
```

De ahora en más podremos llamar a nuestra función por su nuevo nombre.

# Parámetros de una función arrow

Usamos paréntesis para indicar los parámetros. Si nuestra función no recibe parámetros, debemos escribirlos igual.



```
let sumar = (a, b) => a + b;
```

Una particularidad de este tipo de funciones es que si recibe un único parámetro, podemos prescindir de los paréntesis.



```
let doble = a => a * 2;
```



# La flecha de una función arrow

La usamos para indicarle a JavaScript que vamos a escribir una función –reemplaza a la palabra reservada **function**



```
let sumar = (a, b) => a + b;
```

Lo que está a la izquierda de la flecha será la entrada de la función –los parámetros– y lo que está a la derecha, la lógica –y el posible retorno–.

## IMPORTANT

Las funciones arrow reciben su nombre por el operador `=>`.

Si lo miramos con un poco de imaginación, se parece a una flecha. En inglés suele llamarse fat arrow (flecha gorda) para diferenciarlo de la flecha simple `->`.

# Cuerpo de una función arrow

Como ya vimos, si la función tiene una sola línea de código, y esta misma es la que hay que retornar, no hacen falta las llaves ni la palabra reservada return.



```
let sumar = (a, b) => a + b;
```

De lo contrario, vamos a necesitar utilizar ambas. Eso normalmente pasa cuando tenemos más de una línea de código en nuestra función.



```
let esMultiplo = (a, b) => {  
  let resto = a % b; // Obtenemos el resto de la div.  
  return resto == 0; // Si el resto es 0, es múltiplo  
};
```

# {código}

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
  let fecha = new Date();  
  return fecha.getHours() + ':' +  
  fecha.getMinutes();  
}
```

Función arrow **sin**  
**parámetros.**

Requiere de los paréntesis  
para iniciarse.

Al tener una sola línea de  
código, y que esa misma sea  
la que queremos retornar, el  
return queda implícito.

# {código}

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
  let fecha = new Date();  
  return fecha.getHours() + ':' +  
  fecha.getMinutes();  
}
```

Función arrow con un **único** **parámetro**  
–no necesitamos los paréntesis para indicarlo– y con un return implícito.



# {código}

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
  let fecha = new Date();  
  return fecha.getHours() + ':' +  
  fecha.getMinutes();  
}
```

Función arrow con dos  
parámetros.

Necesita de los paréntesis y  
tiene un return implícito.

# {código}

```
let saludo = () => 'Hola Mundo!';
```

```
let dobleDe = numero => numero * 2;
```

```
let suma = (a, b) => a + b;
```

```
let horaActual = () => {  
  let fecha = new Date();  
  return fecha.getHours() + ':' +  
    fecha.getMinutes();  
}
```

Función arrow **sin** parámetros y con un **return** explícito.

En este caso hacemos uso de las llaves y del return, ya que la lógica de esta función se desarrolla en más de una línea de código.

# Veamos un Ejemplo en VSC!





# Objetos Literales



## IMPORTANT

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 casa = {  
  direccion : 'Rioja altura 2000'  
};
```

## 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 cantante = {  
  nombre: 'Harry',  
  apellido: 'Styles'  
};
```

```
console.log(cantante.nombre) // Harry  
console.log(cantante.apellido) // Styles
```

# 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 cantante = {  
  nombre: 'Harry',  
  edad: 32,  
  activoCantando: true,  
  saludar: function() {  
    return '¡Hola! Me llamo Harry';  
  }  
};
```

# 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 cantante = {  
  nombre: 'Harry',  
  apellido: 'Styles',  
  saludar: function() {  
    return '¡Hola! Me llamo Harry';  
  }  
};  
  
console.log(cantante.saludar());  
// ¡Hola! Me llamo Harry
```



# 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 cantante = {  
  nombre: 'Harry',  
  apellido: 'Styles',  
  saludar: function() {  
    return '¡Hola! Me llamo ' + this.nombre;  
  }  
};  
  
console.log(cantante.saludar()); // ¡Hola! Me llamo Harry
```

# Veamos un Ejemplo en VSC!







**Momento de  
poner a prueba  
lo aprendido!**