

Clase 13:

Funciones

Índice

Funciones:

- Concepto.
- Estructura básica.
- Ejemplo en código

Tipos de Funciones:

- Expresadas y declaradas

La invocación

- Scope: Global y Local.
- Conceptos y ejemplos



Funciones

Funciones

IMPORTANT

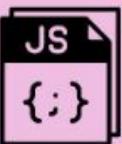
Una función es un **bloque de código** que nos permite **agrupar una funcionalidad** para usarla todas las veces que necesitemos.

Normalmente, realiza una **tarea específica** y **retorna** un valor como resultado.

Estructura básica - Funcion

Palabra reservada

Usamos la palabra function para indicarle a JavaScript que vamos a escribir una función.

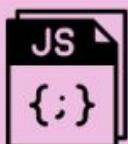


```
function sumar (a, b) {  
    return a + b;  
}
```

Estructura básica - Funcion

Nombre de la función

Definimos un nombre para referirnos a nuestra función al momento de querer **invocarla**.



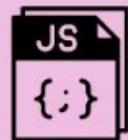
```
function sumar (a, b) {  
    return a + b;  
}
```

Estructura básica - Funcion

Parámetros

Escribimos los paréntesis y, dentro de ellos, los parámetros de la función. Si hay más de uno, los sepáramos usando comas ,

Si la función no lleva parámetros, igual debemos escribir los paréntesis sin nada adentro () .



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

Estructura básica - Funcion

Parámetros (cont.)

Dentro de nuestra función podremos acceder a los parámetros como si fueran variables.

Es decir, con solo escribir los nombres de los parámetros, podremos trabajar con ellos.

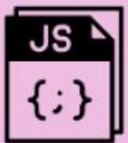


```
function sumar (a, b) {  
    return a + b;  
}
```

Estructura básica - Funcion

Cuerpo

Entre las llaves de apertura y de cierre escribimos la lógica de nuestra función, es decir, el código que queremos que se ejecute cada vez que la invoquemos.



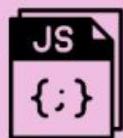
```
function sumar (a, b) {  
    return a + b;  
}
```

Estructura básica - Funcion

El retorno

Es muy común, a la hora de escribir una función, que queramos devolver al exterior el resultado del proceso que estamos ejecutando.

Para eso utilizamos la palabra reservada `return` seguida de lo que queramos retornar.



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

Tipos de Funciones

Funciones

Tipos de Funciones

Función Expresada:
Es aquella que se asigna como valor a una variable

```
let sumar = function(){...}
```

En ambos casos usamos la palabra reservada “function” ya que así le indicamos a javascript que vamos a hacer una función

Función Declarada:
Es aquella que recibe un nombre formal y no se asigna como valor a una variable

```
function sumar(){...}
```

Funciones

¿Cuál es la diferencia entre ambas?

Tipos de Funciones

Función Expresada:

Se carga únicamente cuando el interprete alcanza la línea de código donde se encuentra la función

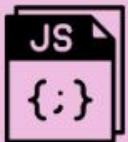
Función Declarada:

Se carga antes de que cualquier código sea ejecutado

Funciones expresadas

Son aquellas que se asignan como valor de una variable. En este caso, la función en sí no tiene nombre, es una **función anónima**.

Para invocarla podremos usar el nombre de la variable que declaremos.

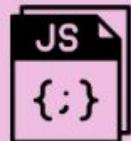


```
let triplicar = function (numero) {  
    return numero * 3;  
}
```

Funciones declaradas

Son aquellas que se declaran usando la **estructura básica**. Pueden recibir un **nombre**, escrito a continuación de la palabra reservada **function**, a través del cual podremos invocarla.

Las funciones con nombre son funciones nombradas.



```
function saludar(nombre) {  
    return 'Hola' + nombre;  
}
```

La invocación

Funciones

La invocación

IMPORTANT

Podemos imaginar las funciones como si fueran máquinas.
Durante la declaración nos ocupamos de construir la máquina y durante la invocación la ponemos a funcionar.

Funciones

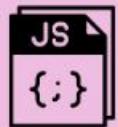
La invocación

IMPORTANT

Invocar una función significa llamarla para que se ejecute. Cuando invocas una función, le dices a tu programa que ejecute el conjunto de instrucciones que contiene esa función.

Invocando una función

Antes de poder invocar una función, necesitamos que haya sido declarada. Entonces, vamos a declarar una función:



```
function hacerHelado() {  
    return '🍦';  
}
```

La forma de invocar –también se puede decir llamar o ejecutar– una función es escribiendo su nombre seguido de apertura y cierre de paréntesis.

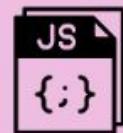


```
hacerHelado(); // Retornará '🍦'
```

Invocando una función

Si la función tiene parámetros, se los podemos pasar dentro de los paréntesis cuando la invocamos.

Es importante respetar el orden porque JavaScript asignará los valores en el orden en que lleguen.

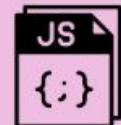


```
function saludar(nombre, apellido) {  
    return 'Hola ' + nombre + ' ' + apellido;  
}  
  
saludar('Robertito', 'Rodríguez');  
// retornará 'Hola Robertito Rodríguez'
```

Invocando una función

También es importante tener en cuenta que, cuando tenemos parámetros en nuestra función, JavaScript va a esperar que se los indiquemos al ejecutarla.

En este caso, al no haber recibido el argumento que necesitaba, JavaScript le asigna el tipo de dato **undefined** a los parámetros *nombre* y *apellido*.



```
function saludar(nombre, apellido) {  
    return 'Hola ' + nombre + ' ' + apellido;  
}  
  
saludar(); // // retorna 'Hola undefined undefined'
```

Invocando una función

Para casos como el anterior podemos definir **valores por defecto**. Si agregamos un igual = luego de un parámetro, podremos especificar su valor en caso de que no llegue ninguno.



```
function saludar(nombre = 'visitante',
  apellido = 'anónimo') {
  return 'Hola ' + nombre + ' ' + apellido;
}

saludar(); // retornará 'Hola visitante anónimo'
```

Guardando los resultados

En caso de querer guardar lo que retorna una función, será necesario almacenarlo en una variable.



```
function hacerHelados(cantidad) {  
    return '🍦 '.repeat(cantidad);  
}  
  
let misHelados = hacerHelados(3);  
console.log(misHelados); // Mostrará en consola '🍦🍦🍦'
```



IMPORTANT

El método repeat en JavaScript es un método de las cadenas de texto (strings) que permite crear una nueva cadena repitiendo la cadena original un número determinado de veces.

Es útil cuando necesitas duplicar una cadena varias veces de manera sencilla.

Aclaración:



Llamamos parámetros a las variables que escribimos cuando definimos la función.

Llamamos argumentos a los valores que enviamos cuando invocamos la función.

Veamos mas sobre las funciones en vsc!



Scope



IMPORTANT

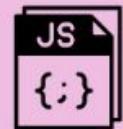
El scope o ámbito se refiere al alcance que tiene una variable, es decir, desde dónde podemos acceder a ella.

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 **fuerá** 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

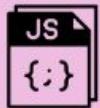
Son una forma de crear funciones incorporadas a partir de ES6 (ECMAScript versión 6).

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.

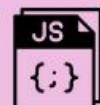
Estructura básica

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.

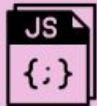


```
let sumar = (a, b) => a + b;
```

Notan algo raro? Falta algo?

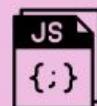
Nombre de una función arrow

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.

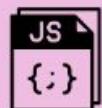


```
let sumar = (a, b) => a + b;
```

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

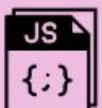
Parámetros de una función arrow

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!';
```

Función arrow sin parámetros.

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

Requiere de los paréntesis para iniciarse.

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

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

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

{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;
```

Función arrow con dos parámetros.

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

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;
```

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

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

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

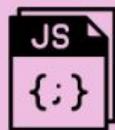
IMPORTANT

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 , .

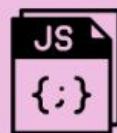
Con la notación objeto.propiedad accedemos al valor de cada una de ellas.



```
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

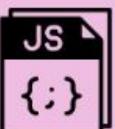
Para ejecutar un método de un objeto usamos la notación `objeto.metodo()`. Los paréntesis del final son los que hacen que el método se ejecute.



```
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!