

REPASO FUNCIONES

“

Una función es un **bloque de código** que podemos invocar todas las veces que necesitemos.

Puede realizar una **tarea específica** y **retornar** un valor.

Nos permite **agrupar** el **código** que vayamos a **usar muchas veces**.



A thick, solid green diagonal stripe runs from the top right corner towards the bottom left, separating the white background on the left from the solid green background on the right.

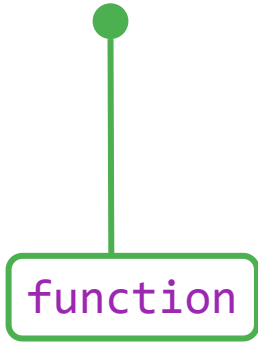
1.

DECLARACIÓN Y ESTRUCTURA

ESTRUCTURA BÁSICA

Palabra reservada

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

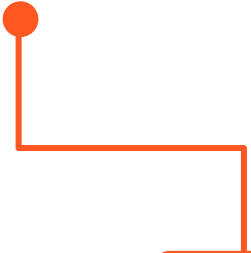


```
function sumar (a,b) {  
    return a + b;  
}
```

ESTRUCTURA BÁSICA

Nombre

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




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

ESTRUCTURA BÁSICA

Parámetros

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

Si la función no lleva parámetros, escribimos los paréntesis sin nada adentro ().

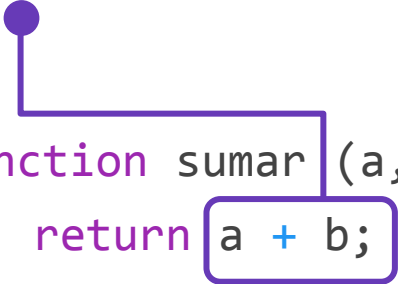


```
function sumar (a, b) {  
    return a + b;  
}
```

ESTRUCTURA BÁSICA

Parámetros

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.



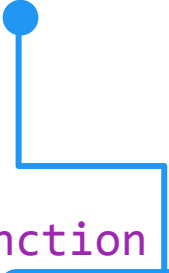
```
function sumar(a, b) {  
    return a + b;  
}
```

The diagram illustrates the function definition. A purple dot is connected by a line to a box that highlights the parameters `a` and `b` in the function signature `sumar(a, b)`. Another box highlights the expression `a + b` in the `return` statement, showing how the parameters are used within the function body.

ESTRUCTURA BÁSICA

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.



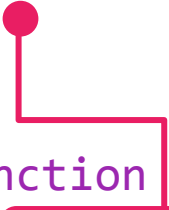
```
function sumar (a,b) {  
    return a + b;  
}
```


ESTRUCTURA BÁSICA

El retorno

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

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



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

FUNCIONES DECLARADAS

Son aquellas que se declaran usando la **estructura básica**.
Reciben un **nombre formal** a través del cual la invocaremos.

```
{}  
  
function hacerHelado(cantidad) {  
    return '🍦'.repeat(cantidad)  
}
```



Se cargan **antes** de que cualquier código sea ejecutado.

FUNCIONES EXPRESADAS

Son aquellas que se **asignan como valor** a una variable. El nombre de la función será el **nombre** de la **variable** que declaremos.

```
let hacerSushi = function(cantidad) {  
    return '🍣'.repeat(cantidad)  
}
```



Se cargan cuando el intérprete **alcanza la línea de código** donde se **encuentra la función**.

A thick, solid green diagonal stripe runs from the top right corner towards the bottom left, separating the white background on the left from the solid green background on the right.

2.

INVOCACIÓN

INVOCANDO UNA FUNCIÓN

La forma de **invocar** (ejecutar) una función es escribiendo su nombre seguido de apertura y cierre de paréntesis.

```
nombreFuncion();
```

En caso de querer ver o guardar el dato que **retorna**, será necesario almacenarlo en una variable, o hacer un *console.log* de la ejecución.

```
let resultado = nombreFuncion();  
console.log(nombreFuncion());
```

INVOCANDO UNA FUNCIÓN

Si la función espera argumentos, podemos pasárselos dentro del paréntesis.

Es importante respetar el orden si hay más de un parámetro ya que Javascript los asignará en el orden que lleguen.

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

{}

INVOCANDO UNA FUNCIÓN

También es importante tener en cuenta que cuando tenemos parámetros en nuestra función, Javascript va a esperar que los pasemos como argumentos al ejecutarla.

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



Al no haber recibido el argumento que necesitaba, Javascript le asigna el tipo de dato **undefined** a las variables nombre y apellido.

INVOCANDO UNA FUNCIÓN

Para este tipo de casos Javascript nos permite definir los **valores por defecto**.

Si agregamos un igual `=` luego del parámetro, podremos especificar su valor en caso de que no llegue ninguno.

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

{ }

Los **parámetros** son las **variables** que escribimos cuando **definimos** la función.

Los **argumentos** son los **valores** que enviamos cuando **invocamos** la función.



A thick, solid green diagonal stripe runs from the top right corner towards the bottom left, separating the white background from a solid green area on the right.

3.

SCOPE

“

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

Los scopes **son definidos** principalmente **por las funciones**.

Es fundamental dominarlo cuando trabajamos con ellas.



SCOPE LOCAL

En el momento en que declaramos una variable **dentro** de una función, la misma 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 para **Javascript** esa variable **no existe**.

```
function miFuncion() {  
    // todo el código que escribamos dentro  
    // de nuestra función, tiene scope local  
}
```

{ }

```
{ 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**.

Sólo dentro de esta función podemos acceder a ella.

```
{ código }
```

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

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

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
var miVariable;
function miFuncion() {
    // Tenemos acceso a las variables globales
}
```

```
{ código }
```

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

```
function hola() {  
    return 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.


```
{ código }
```

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

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

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

Su alcance es **global**, por lo tanto, Javascript sabe a qué variable me estoy refiriendo y ejecuta la función con éxito.