

# Las funciones

# Índice

- 01 [Funciones](#)
- 02 [Arrow functions](#)
- 03 [Callbacks](#)



01

# Funciones

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

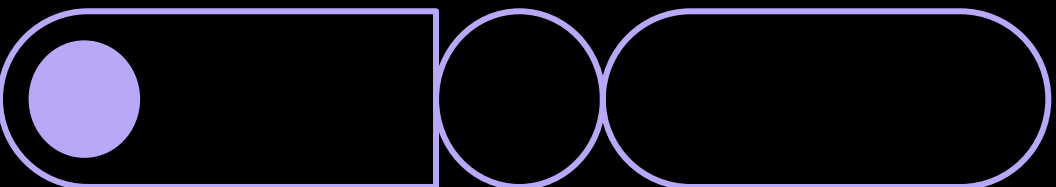
Normalmente realiza una **tarea específica** y **retorna** un valor como resultado.



Encotrá más información: [aquí](#)



# 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 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

## Parámetros

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

## 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;  
}
```



# 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 ejecutando.

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

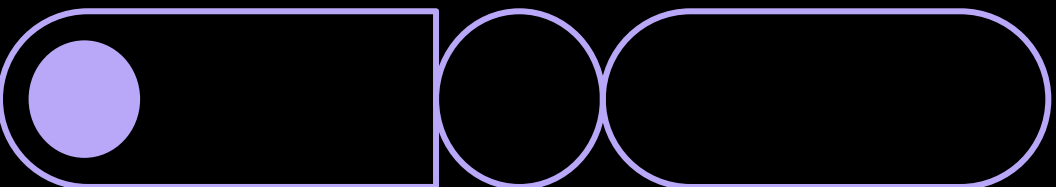


# Funciones expresadas

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

# Invocación





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**.



# Invocando una función

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

```
{ } function hacerHelado(cantidad) {  
    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, ya que 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.

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

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.



# Invocando una función

Para casos como el anterior podemos definir **valores por defecto**.  
Si agregamos un igual `=` luego 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'
```

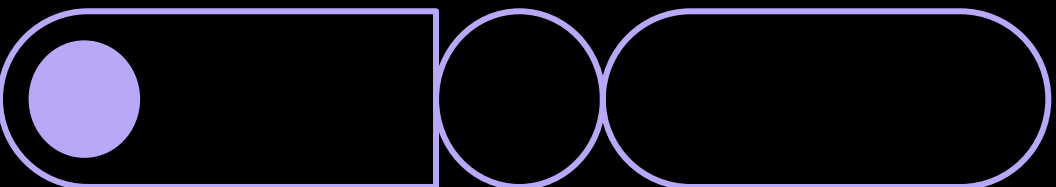


# Guardando los resultados

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 '🍦🍦🍦'
```

# Scope





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.




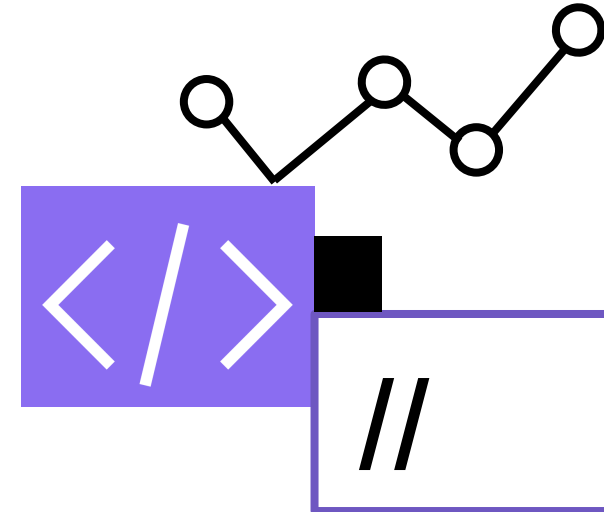
# Scope Local

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
```





{código}

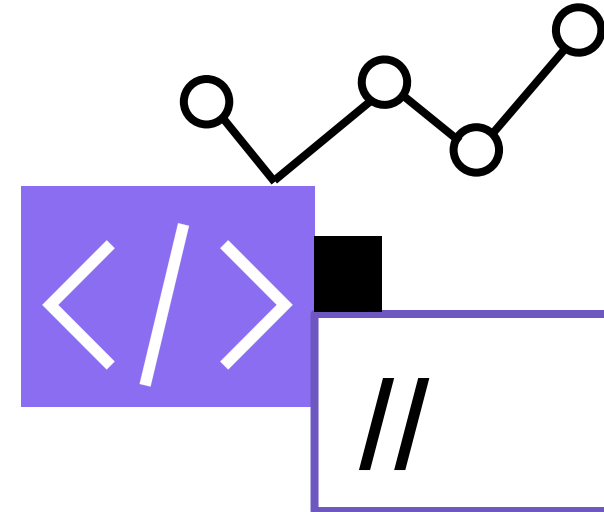
```
function hola() {  
  let saludo = 'Hola ¿qué tal?';  
  return saludo;  
}
```

```
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.





{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**



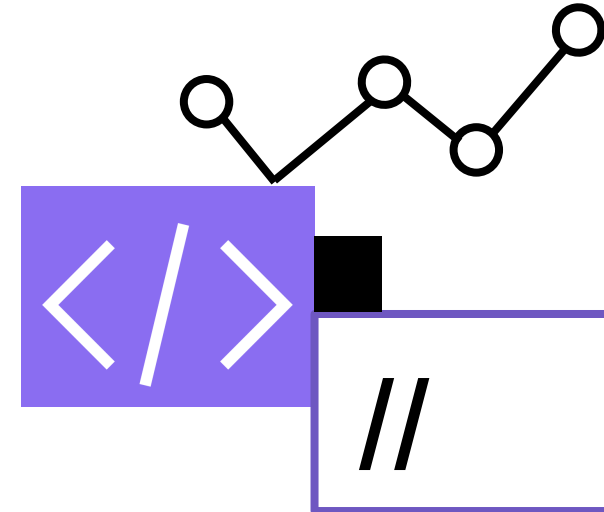
# Scope **global**

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  
  }  
}
```





## {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.



# {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 me estoy refiriendo y ejecuta la función con éxito.

02

# Arrow functions

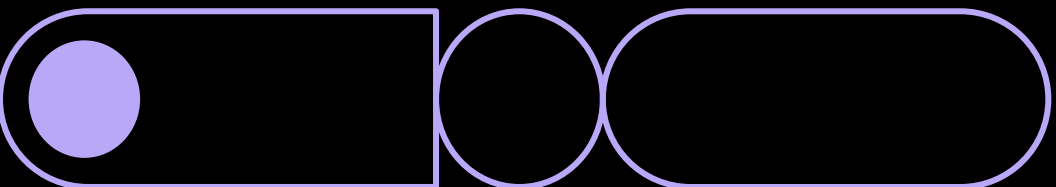
Las **Arrow functions**  
o funciones flecha  
nos permiten  
escribir funciones  
con una sintaxis  
más compacta.



Encotrá más  
información: [aquí](#)



# Declaración y estructura





# 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;
```





# Nombre de una arrow function

Las arrow functions **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 arrow function

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 arrow function

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).

# ¿Por qué se llama **arrow function**?

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 arrow function

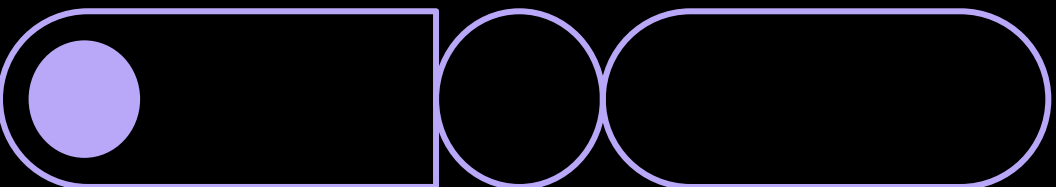
Como ya vimos, si la función tiene una sola línea de código, y esta misma es la que hay que **retornar**, no hacen falta las llaves ni la palabra reservada return.

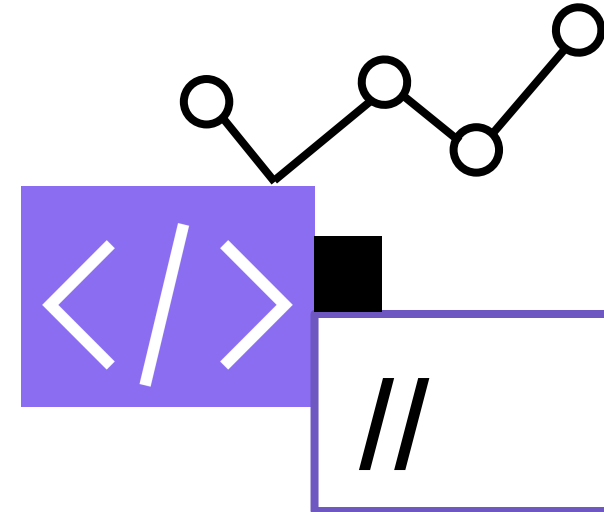
```
{ } let sumar = (a, b) => a + b;
```

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  
};
```

# Ejemplos





# {código}

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

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

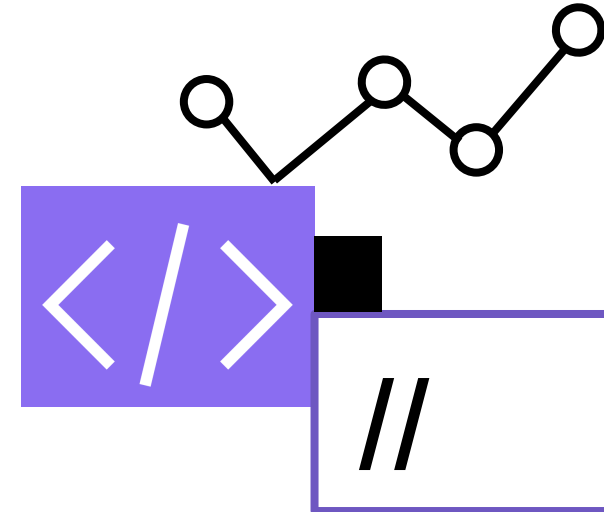
```
let suma = (a, b) => a + b;
```

```
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 esta misma sea la que queremos retornar, **el return queda implícito**.



## {código}

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

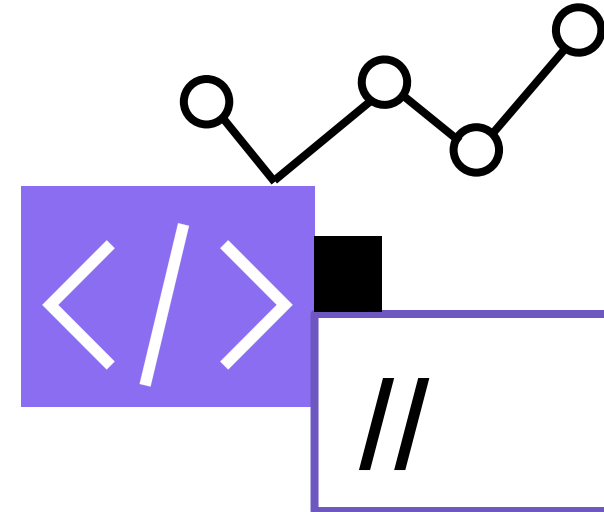
```
let dobleDe = numero => numero * 2;
```

```
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 **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.

03

# Callbacks

Un **callback** es una **función** que se pasa como parámetro de otra función.

La función que lo recibe es quien se encarga de **ejecutarla** cuando sea necesario.



Encotrá más información: [aquí](#)





# El callback **anónimo**

En este caso, la función que pasamos como **callback** no tiene nombre. Es decir, es una **función anónima**.

Como las **funciones anónimas** no pueden ser llamadas por su nombre, debemos escribirla dentro de la función que se encargará de llamar al callback.

```
{ }  
setTimeout( function(){  
    console.log('Hola Mundo!')  
} , 1000)
```



# El callback **definido**

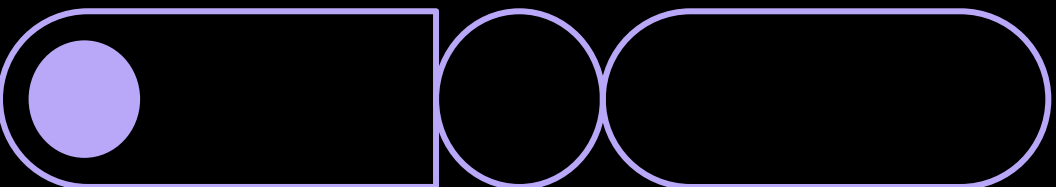
La función que pasamos como **callback** puede ser una función previamente **definida**. Al momento de pasarla como parámetro de otra función, nos referiremos a la misma por su nombre.

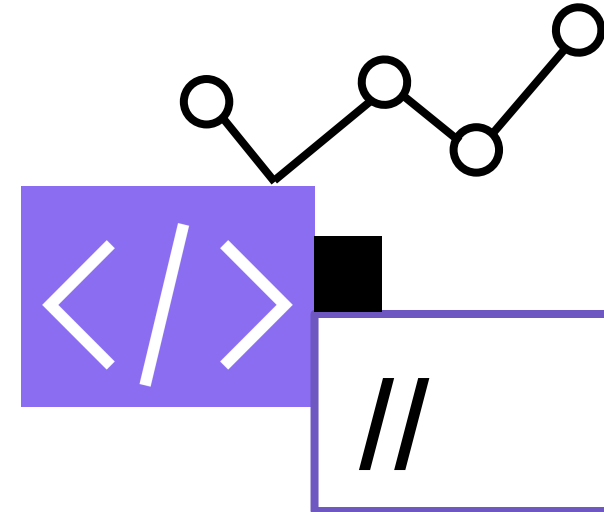
```
{ }  
let miCallback = () => console.log('Hola mundo!');  
setTimeout(miCallback, 1000);
```



Al escribir una función como parámetro lo hacemos sin los paréntesis para **evitar que se ejecute**. Será la función que la recibe quien se encargue de ejecutarla.

# Ejemplos





# {código}

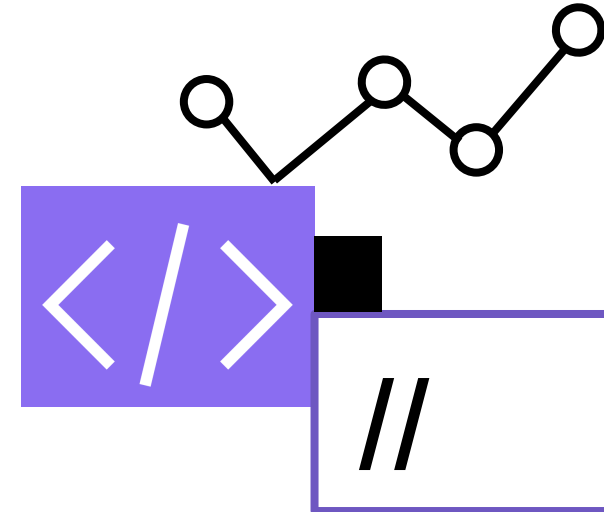
```
function nombreCompleto(nombre, apellido) {  
    return nombre + ' ' + apellido;  
};
```

```
function saludar(nombre, apellido, callback)  
{  
    return '¡Hola ' + callback(nombre,  
apellido) + '!';  
};  
saludar('Juanito', 'Sánchez',  
nombreCompleto);
```

Definimos la función **nombreCompleto()**.

Esta se encarga de unir con un espacio un nombre y un apellido.

Nos **devuelve** un **string**.



# {código}

```
function nombreCompleto(nombre, apellido) {  
  return nombre + ' ' + apellido;  
};
```

```
function saludar(nombre, apellido, callback) {  
  return '¡Hola ' + callback(nombre,  
apellido) + '!';  
};
```

```
saludar('Juanito', 'Sánchez', nombreCompleto);
```

Definimos la función **saludar()**.

Esta recibe un nombre, un apellido y un **callback** como parámetros.

Este último será la función que vamos a querer ejecutar internamente.





# {código}

```
function nombreCompleto(nombre, apellido) {  
    return nombre + ' ' + apellido;  
};  
  
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre,  
apellido) + '!';  
};  
  
saludar('Juanito', 'Sánchez', nombreCompleto);
```

Lo que queremos devolver es un string completo.

La primera parte la tenemos en el return: '¡Hola (...) !'.

El resto (...) vendrá de lo que nos retorne el **callback** en el momento en el que se ejecute.



# {código}

```
function nombreCompleto(nombre, apellido) {  
  return nombre + ' ' + apellido;  
};  
  
function saludar(nombre, apellido, callback) {  
  return '¡Hola ' + callback(nombre,  
  apellido) + '!';  
};  
  
saludar('Juanito', 'Sánchez', nombreCompleto);
```

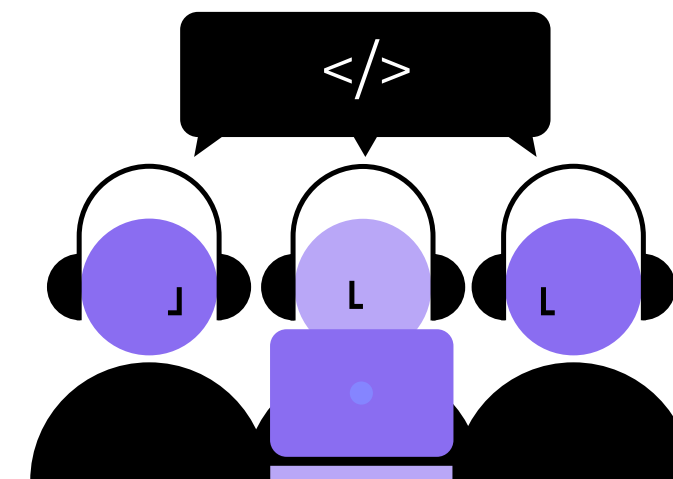
Ejecutamos la función **saludar**, le pasamos como parámetros un nombre, un apellido y la función **nombreCompleto()**.

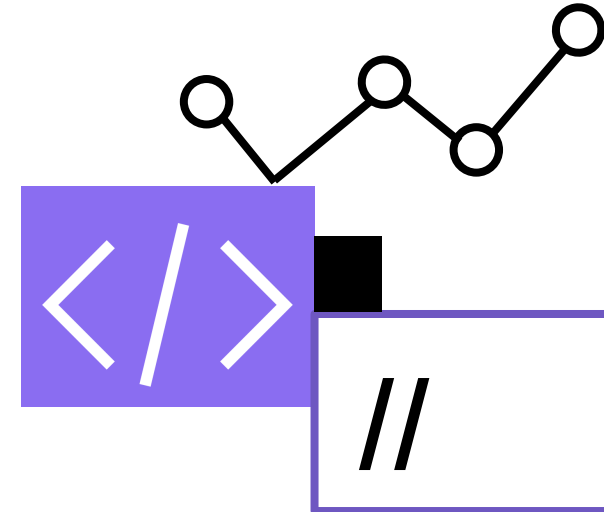
**Primero** se ejecutará el **callback**, que va a devolver el nombre completo y **luego** se ejecutará la función **saludar()**, que va a devolver el saludo completo.

La función **saludar()** ¿solo funciona si le pasamos como **callback** la función **nombre()**?

**¡No!**

Podemos pasarle cualquier otra función que devuelva un **string**, ya que en la estructura interna de **saludar()** definimos que opere con ese tipo de dato.





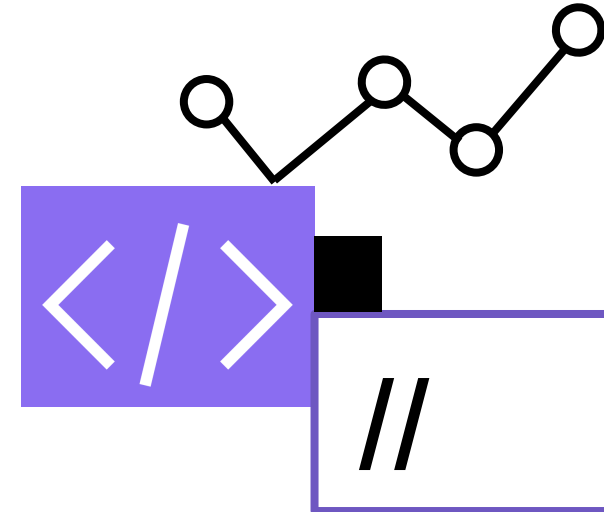
## {código}

```
function iniciales(nombre, apellido) {  
    return nombre[0] + apellido[0];  
};
```

```
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre,  
    apellido) + '!';  
};
```

```
saludar('Juanito', 'Sánchez', iniciales);
```

Podríamos definir otra función que se encargue de devolvernos las iniciales del nombre y el apellido que nos pasen.



## {código}

```
function iniciales(nombre, apellido) {  
    return nombre[0] + apellido[0];  
};  
  
function saludar(nombre, apellido, callback) {  
    return '¡Hola ' + callback(nombre,  
    apellido) + '!';  
};  
  
saludar('Juanito', 'Sánchez', iniciales);
```

Esta vez cuando ejecutamos la función `saludar()`, le pasamos la función **`iniciales()`** como callback.

Nuevamente se ejecutará el callback, esta vez va a devolver las iniciales y luego se ejecutará la función **`saludar()`**, que va a devolver el saludo completo.

¡Muchas gracias!