# DWC

# (Desarrollo Web en entorno cliente)

# Índice


1.- Las funciones ..... 1
2.- Declaración de funciones ..... 1
2.1.- Variables locales ..... 2
2.2.- Variables externas (globales) ..... 2
2.3.- Parámetros ..... 4
2.4.- Valores predeterminados ..... 5
2.5.- Parámetros predeterminados alternativos ..... 6
2.6.- Devolver un valor ..... 7
2.7.- Nombrar una función ..... 9
2.8.- Parametros variables: arguments, rest y spread ..... 11
2.9.- Recomendaciones ..... 14
2.10.- Resumen declaración de funciones ..... 15
3.- Expresiones de funciones ..... 17
3.1- Paso de funciones como parámetros ..... 19
3.2- Expresión de función vs declaración de función ..... 21
3.3.- Resumen de expresión de función ..... 24
4.- Fat arrow (“funciones de flecha”) ..... 25
4.1.- Funciones Fat Arrow multilínea ..... 26
4.2.- Resumen de funciones Fat Arrow ..... 27

# 1.- Las funciones

Muy a menudo necesitamos realizar una acción similar en muchos lugares del script.

Por ejemplo, debemos mostrar un mensaje atractivo cuando un visitante inicia sesión, cierra sesión y tal vez en otro lugar.

Las funciones son los principales "bloques de construcción" del programa. Permiten que el código se llame muchas veces sin repetición.

Ya hemos visto ejemplos de funciones integradas, como `alert(message)`, `prompt(message, default)`y `confirm(question)`. Pero también podemos crear funciones propias.

# 2.- Declaración de funciones

Para crear una función podemos usar una *declaración de función* .

Se parece a esto:

```
function showMessage() {
  alert( 'Hello everyone!' );
}
```

La palabra clave `function` va primero, luego va el *nombre de la función* , luego una lista de *parámetros* entre paréntesis (separados por comas, vacíos en el ejemplo anterior) y finalmente el código de la función, también llamado "el cuerpo de la función", entre paréntesis.

```
function name(parameters) {
  ...body...
}
```

Nuestra nueva función puede ser llamada por su nombre: `showMessage()`.

Por ejemplo:

```
function showMessage() {
  alert( 'Hello everyone!' );
}
showMessage();
showMessage();
```

La llamada showMessage() ejecuta el código de la función. Aquí veremos el mensaje dos veces.

Este ejemplo demuestra claramente uno de los propósitos principales de las funciones: evitar la duplicación de código.

Si alguna vez necesitamos cambiar el mensaje o la forma en que se muestra, es suficiente modificar el código en un lugar: la función que lo genera.

## 2.1.- Variables locales

Una variable declarada dentro de una función solo es visible dentro de esa función.

Por ejemplo:

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable

  alert( message );
}

showMessage(); // Hello, I'm JavaScript!

alert( message ); // <-- Error! The variable is local to the function
```

## 2.2.- Variables externas (globales)

Las variables declaradas fuera de cualquier función, como la externa userName en el siguiente ejemplo, se denominan *globales* .

Las variables globales son visibles desde cualquier función (a menos que estén duplicadas por las locales).

Es una buena práctica minimizar el uso de variables globales. El código moderno tiene pocos o no globales. La mayoría de las variables residen en sus funciones. A veces, sin embargo, pueden ser útiles para almacenar datos a nivel de proyecto.

Una función también puede acceder a una variable externa, por ejemplo:

```
let userName = 'John';
```

```
function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}
```

`showMessage(); // Hello, John`

La función tiene acceso completo a la variable externa. Puede modificarlo también.

Por ejemplo:

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable

  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function
```

La variable externa solo se usa si no hay una local.

Si se declara una variable con el mismo nombre dentro de la función, entonces *ignora a* la externa. Por ejemplo, en el siguiente código, la función usa el local `userName`.

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declare a local variable
```

```
  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// the function will create and use its own userName
showMessage();

alert( userName ); // John, unchanged, the function did not access the outer variable
```

## 2.3.- Parámetros

Podemos pasar datos arbitrarios a funciones usando parámetros (también llamados *argumentos de función* ).

En el siguiente ejemplo, la función tiene dos parámetros: `from` y `text`.

```
function showMessage(from, text) { // arguments: from, text
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

Cuando la función se llama en líneas `(*)`y `(**)`, los valores dados se copian en variables locales `from`y `text`.

Aquí hay un ejemplo más: tenemos una variable `from` y la pasamos a la función. La función cambia `from`, pero el cambio no se ve afuera, porque una función siempre obtiene una copia del valor:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // make "from" look nicer
  alert( from + ': ' + text );
}
let from = "Ann";
```

```
showMessage(from, "Hello"); // *Ann*: Hello

// the value of "from" is the same, the function modified a local copy
alert( from ); // Ann
```

## 2.4.- Valores predeterminados

Si no se proporciona un parámetro, su valor se convierte en undefined.

Por ejemplo, la función mencionada showMessage(from, text) se puede llamar con un solo argumento:

```
showMessage("Ann");
```

Eso no es un error. Tal llamada saldría "Ann: undefined". No hay text, así que se supone que text === undefined.

Si queremos utilizar un "valor predeterminado" text en este caso, podemos especificarlo después del simbolo = en el argumento de la cabecera:

```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```

Ahora, si text no se pasa el parámetro, obtendrá el valor "no text given"

Aquí "no text given" es una cadena, pero puede ser una expresión más compleja, que solo se evalúa y se asigna si falta el parámetro.

Esto también es posible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() only executed if no text given
  // its result becomes the value of text
}
```

**Evaluación de parámetros por defecto**

En JavaScript, se evalúa un parámetro predeterminado cada vez que se llama a la función sin el parámetro respectivo.

En el ejemplo anterior, `anotherFunction()` se llama cada vez que `showMessage()` se llama sin el parámetro `text`.

## 2.5.- Parámetros predeterminados alternativos

A veces tiene sentido establecer valores predeterminados para parámetros que no están en la declaración de la función, sino en una etapa posterior, durante su ejecución.

Para verificar un parámetro omitido, podemos compararlo con `undefined`:

```
function showMessage(text) {
  if (text === undefined) {
    text = 'empty message';
  }

  alert(text);
}

showMessage(); // empty message
```

... O podríamos usar el operador `||`:

```
// if text parameter is omitted or "" is passed, set it to 'empty'
function showMessage(text) {
  text = text || 'empty';
  ...
}
```

Los motores JavaScript modernos admiten el operador de fusión nulo `??` , es mejor cuando los valores falsos, como `0`, se consideran regulares:

```
// if there's no "count" parameter, show "unknown"
function showCount(count) {
  alert(count ?? "unknown");
```

```
}

showCount(0); // 0
showCount(null); // unknown
showCount(); // unknown
```

## 2.6.- Devolver un valor

Una función puede devolver un valor al código de llamada como resultado.

El ejemplo más simple sería una función que suma dos valores:

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

let result = sum(1, 2);
alert( result ); // 3
```

La directiva `return` puede estar en cualquier lugar de la función. Cuando la ejecución lo alcanza, la función se detiene y el valor se devuelve al código de llamada (asignado `result` anteriormente).

Puede haber muchas ocurrencias `return` en una sola función. Por ejemplo:

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
```

```
  alert( 'Access denied' );
}
```

Es posible usar `return` sin un valor. Eso hace que la función salga inmediatamente.
Por ejemplo:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```

En el código anterior, si `checkAge(age)` devuelve `false`, `showMovie` no se ejecutara el `alert`.

**Una función con un return `vacío` o sin él devuelve `undefined`**

Si una función no devuelve un valor, es lo mismo que si devuelve `undefined`:

```
function doNothing() { /* empty */ }

alert( doNothing() === undefined ); // true
```

Un vacío returntambién es lo mismo que return undefined:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

**Nunca se debe agregar una nueva línea entre `return` y el valor**

Para una expresión larga en `return`, puede ser tentador ponerlo en una línea separada, como esta:

```
return
    (some + long + expression + or + whatever * f(a) + f(b))
```

Eso no funciona, porque JavaScript asume un punto y coma después `return`.

Eso funcionará igual que:

```
return;
    (some + long + expression + or + whatever * f(a) + f(b))
```

Entonces, efectivamente se convierte en un return vacío.

Si queremos que la expresión devuelta se ajuste a través de varias líneas, debemos comenzarla en la misma línea que `return`. O al menos ponga los paréntesis de apertura de la siguiente manera:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

Y funcionará tal como lo esperamos.

## 2.7.- Nombrar una función

Las funciones son acciones. Entonces su nombre suele ser un verbo. Debe ser breve, lo más preciso posible y describir lo que hace la función, para que alguien que lea el código obtenga una indicación de lo que hace la función.

Es una práctica generalizada comenzar una función con un prefijo verbal que describe vagamente la acción. Debe haber un acuerdo dentro del equipo sobre el significado de los prefijos.

Por ejemplo, las funciones que comienzan `"show"` generalmente muestran algo.

Función que comienza con ...

- `"get…"` - devolver un valor
- `"calc…"` - calcular algo
- `"create…"` - crear algo
- `"check…"` - verifica algo y devuelve un booleano, etc.

Ejemplos de tales nombres:

- showMessage(..) // shows a message
- getAge(..) // returns the age (gets it somehow)
- calcSum(..) // calculates a sum and returns the result
- createForm(..) // creates a form (and usually returns it)
- checkPermission(..) // checks a permission, returns true/false

Con los prefijos en su lugar, un vistazo al nombre de una función permite comprender qué tipo de trabajo realiza y qué tipo de valor devuelve.

**Una función: una acción**

Una función debe hacer exactamente lo que sugiere su nombre, no más.

Dos acciones independientes por lo general merecen dos funciones, incluso si generalmente se llaman juntas (en ese caso, podemos hacer una tercera función que llame a esas dos).

Algunos ejemplos de romper esta regla:

- getAge- Sería malo si se muestra un alert con la edad (solo debería aparecer ).
- createForm - sería malo si modifica el documento, agregando un formulario (solo debería crearlo y devolverlo).
- checkPermission- sería malo si muestra el mensaje access granted/denied (solo debe realizar la verificación y devolver el resultado).

Estos ejemplos asumen significados comunes de prefijos. El programador y su equipo son libres de acordar otros significados, pero generalmente no son muy diferentes. En cualquier caso, debe tener una comprensión firme de lo que significa un prefijo, lo que una función con prefijo puede y no puede hacer. Todas las funciones con el mismo prefijo deben obedecer las reglas. Y el equipo debe compartir el conocimiento.

**Nombres de funciones ultracortos**

Las funciones que se usan con *mucha frecuencia a* veces tienen nombres ultracortos.

Por ejemplo, el marco jQuery define una función con $. La biblioteca Lodash tiene su función principal llamada _.

Estas son excepciones. En general, los nombres de las funciones deben ser concisos y descriptivos.

## 2.8.- Parametros variables: arguments, rest y spread

A una función en JavaScript podemos llamarla con un numero diferente de argumentos en cada llamada. Para poder identificar los parámetros con los que se llama a la función utilizamos el “array” **arguments** (realmente es un iterable similar a un array) que se crea cuando se llama a la función

```
<script>
function miFuncion(){
  console.log("Numero de parametros: "+ arguments.length);
  for (i=0;i<arguments.length;i++)
    console.log("Argumento " + (i+1) + ": " + arguments[i]);
}
miFuncion("hola",1,2,"adiós");
miFuncion(1,2,3,4,5,6,"a","b","c","d");
</script>
```

| | |
|---|---|
| Numero de parametros: 4 | parametros.html:3 |
| Argumento 1: hola | parametros.html:5 |
| Argumento 2: 1 | parametros.html:5 |
| Argumento 3: 2 | parametros.html:5 |
| Argumento 4: adiós | parametros.html:5 |
| Numero de parametros: 10 | parametros.html:3 |
| Argumento 1: 1 | parametros.html:5 |
| Argumento 2: 2 | parametros.html:5 |
| Argumento 3: 3 | parametros.html:5 |
| Argumento 4: 4 | parametros.html:5 |
| Argumento 5: 5 | parametros.html:5 |
| Argumento 6: 6 | parametros.html:5 |
| Argumento 7: a | parametros.html:5 |
| Argumento 8: b | parametros.html:5 |
| Argumento 9: c | parametros.html:5 |
| Argumento 10: d | parametros.html:5 |

Esta forma de trabajar podemos decir que es la forma más antigua de Javascript.

En el EmacScript 6 se añaden nuevos métodos para trabajar con Arrays y los conceptos de Rest y Spread.

### Parámetros Rest

La sintaxis de los parámetros rest nos permiten representar un número indefinido de argumentos como un array.

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));
// expected output: 6

console.log(sum(1, 2, 3, 4));
// expected output: 10
```

El último parámetro de una función se puede prefijar con ..., lo que hará que todos los argumentos restantes (suministrados por el usuario) se coloquen dentro de un array de javascript "estándar".

Sólo el último parámetro puede ser un "parámetro rest".

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}
```

```
myFun("one", "two", "three", "four", "five", "six");

// Console Output:
// a, one
// b, two
// manyMoreArgs, [three, four, five, six]
```

**Diferencia entre los parámetros rest y el objeto arguments**

Hay dos diferencias principales entre los parámetros rest y el objeto arguments:

1. los parámetros rest son sólo aquellos a los que no se les ha asignado un nombre, mientras que el objeto arguments contiene todos los argumentos que se le han pasado a la función;
2. el objeto arguments no es un array real, mientras que los parámetros rest son instancias de Array , lo que significa que lo los métodos como sort, map, forEach o pop pueden aplicarse directamente;

## Sintaxis Spread

La sintaxis extendida o spread syntax permite a un elemento iterable tal como un array desestructurarse en valores individuales asociados a un nombre, es decir en variables independientes.

Es la operación inversa a los parámetros Rest.

```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];

console.log(sum(...numbers));
// expected output: 6
```

## 2.9.- Recomendaciones

Las funciones deben ser cortas y hacer exactamente una cosa. Si esa cosa es grande, tal vez valga la pena dividir la función en algunas funciones más pequeñas. A veces, seguir esta regla puede no ser tan fácil, pero definitivamente es algo bueno.

Una función separada no solo es más fácil de probar y depurar, ¡su existencia es un gran comentario!

Por ejemplo, compare las dos funciones a `showPrimes(n)` continuación. Cada uno genera números primos hasta `n`.

La primera variante usa una etiqueta:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // a prime
  }
}
```

La segunda variante usa una función adicional `isPrime(n)` para comprobar si un número es primo:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);  // a prime
  }
}

function isPrime(n) {
```

```
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

La segunda variante es más fácil de entender, ¿no? En lugar de la pieza de código, vemos un nombre de la acción ( `isPrime`). A veces las personas se refieren a dicho código como *autodescriptivo* .

Por lo tanto, las funciones se pueden crear incluso si no tenemos la intención de reutilizarlas. Estructuran el código y lo hacen legible.

## 2.10.- Resumen declaración de funciones

Una declaración de función se ve así:

```
function name(parameters, delimited, by, comma) {
  /* code */
}
```

- Los valores pasados a una función como parámetros se copian a sus variables locales.
- Una función puede acceder a variables externas. Pero funciona solo de adentro hacia afuera. El código fuera de la función no ve sus variables locales.
- Una función puede devolver un valor. Si no lo hace, entonces su resultado es `undefined`.

Para que el código sea limpio y fácil de entender, se recomienda utilizar principalmente variables y parámetros locales en la función, no variables externas.

Siempre es más fácil comprender una función que obtiene parámetros, trabaja con ellos y devuelve un resultado que una función que no obtiene parámetros, pero modifica las variables externas como un efecto secundario.

Nombre de las funciones:

- Un nombre debe describir claramente lo que hace la función. Cuando vemos una llamada a la función en el código, un buen nombre nos da al instante una comprensión de lo que hace y devuelve.
- Una función es una acción, por lo que los nombres de las funciones suelen ser verbales.
- Existen muchos prefijos de función conocida como `create…`, `show…`, `get…`, `check…`y así sucesivamente. Úsalos para insinuar qué hace una función.

Las funciones son los principales bloques de construcción de los scripts. Ahora hemos cubierto los conceptos básicos, por lo que en realidad podemos comenzar a crearlos y usarlos.

## 3.- Expresiones de funciones

En JavaScript, una función es un tipo especial de valor.

La sintaxis que usamos antes se llama *Declaración de función* :

```
function sayHi() {
  alert( "Hello" );
}
```

Hay otra sintaxis para crear una función que se llama ***Expresión de función***.

Un ejemplo sería:

```
let sayHi = function() {
  alert( "Hello" );
};
```

Aquí, la función se crea y se asigna a la variable explícitamente, como cualquier otro valor. No importa cómo se defina la función, es solo un valor almacenado en la variable `sayHi`.

El significado de estos ejemplos de código es el mismo: "crear una función y ponerla en la variable `sayHi`".

Incluso podemos imprimir ese valor usando `alert`:

```
function sayHi() {
  alert( "Hello" );
}

alert( sayHi ); // shows the function code
```

La última línea no ejecuta la función, porque no hay paréntesis después `sayHi`. Hay lenguajes de programación donde cualquier mención del nombre de una función provoca su ejecución, pero JavaScript no es así.

En JavaScript, una función es un valor, por lo que podemos tratarlo como un valor. El código anterior muestra su representación de cadena, que es el código fuente.

Seguramente, una función es un valor especial, en el sentido de que podemos llamarla así `sayHi()`.

Pero sigue siendo un valor. Entonces podemos trabajar con él como con otros tipos de valores.

Podemos copiar una función a otra variable:

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    //     this still works too (why wouldn't it)
```

Esto es lo que sucede arriba en detalle:

1. La declaración de funcion (1) crea la función y la coloca en la variable nombrada sayHi.
2. La línea (2) copia en la variable func.la función No hay paréntesis después de sayHi. Si lo hubiera, a continuación, func = sayHi()escribiría *el resultado de la llamada* sayHi() a func, no *la función* sayHi en sí.
3. Ahora la función se puede llamar como ambos sayHi()y func().

También podríamos haber utilizado una expresión de función para declarar sayHi, en la primera línea:

```
let sayHi = function() {
  alert( "Hello" );
};

let func = sayHi;
// ...
```

Todo funcionaría igual.

**¿Por qué hay un punto y coma al final?**

Quizás os preguntéis por qué la expresión de función tiene un punto y coma ; al final, pero la declaración de función no:

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

La respuesta es simple:

- No hay necesidad de poner ; al final de los bloques de código y las estructuras sintácticas que los utilizan if { ... }, for { }, function f { }etc.
- Se utiliza una expresión de función dentro de la declaración: let sayHi = ...; como un valor. No es un bloque de código, sino más bien una asignación. El punto y coma ; se recomienda al final de las declaraciones, sin importar cuál sea el valor. Entonces, el punto y coma aquí no está relacionado con la Expresión de Función en sí misma, solo termina la declaración. Es decir, es como una instrucción de asignación y como tal lleva punto y coma al final.

## 3.1- Paso de funciones como parámetros

Veamos más ejemplos de pasar funciones como valores y usar expresiones de función.

Escribiremos una función ask(question, yes, no) con tres parámetros:

**question**

Texto de la pregunta

**yes**

Función para ejecutar si la respuesta es "Sí"

**no**

Función para ejecutar si la respuesta es "No"

La función debe preguntar `question` y, dependiendo de la respuesta del usuario, llamar `yes()` o `no()`:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  alert( "You agreed." );
}
function showCancel() {
  alert( "You canceled the execution." );
}
// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

En la práctica, tales funciones son bastante útiles.

**Los argumentos `showOky` `showCancel` de `ask` se denominan *funciones de devolución de llamada* o simplemente *devoluciones de llamada*.**

La idea es que pasemos una función y esperemos que sea "devuelta" más tarde si es necesario. En nuestro caso, `showOk` se convierte en la devolución de llamada para la respuesta "sí" y `showCancel` para la respuesta "no".
Podemos pasar la definición de la función en la llamada para escribir la misma función mucho más corta:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
```

```
  function() { alert("You canceled the execution."); }
);
```

Aquí, las funciones se declaran dentro de la llamada ask(...). No tienen nombre, y por eso se llaman ***anónimas*** . Dichas funciones no son accesibles fuera de ask (porque no están asignadas a variables), pero eso es justo lo que queremos aquí. Tal código aparece en nuestros scripts de forma muy natural, está en el espíritu de JavaScript.

## 3.2- Expresión de función vs declaración de función

Formulemos las diferencias clave entre las declaraciones de funciones y las expresiones.

Primero, la sintaxis: cómo diferenciarlos en el código.

***Declaración de función****:* una función, declarada como una declaración separada, en el flujo del código principal.

```
// Function Declaration
function sum(a, b) {
  return a + b;
}
```

***Expresión de función****:* una función, creada dentro de una expresión o dentro de otra construcción de sintaxis. Aquí, la función se crea en el lado derecho de la "expresión de asignación" =:

```
// Function Expression
let sum = function(a, b) {
  return a + b;
};
```

La diferencia más sutil es *cuando* el motor de JavaScript crea una función.

**Una expresión de función se crea cuando la ejecución la alcanza y solo se puede utilizar desde ese momento.**

Una vez que el flujo de ejecución pasa al lado derecho de la asignación `let sum = function…`, la función se crea y se puede usar (asignada, llamada, etc.) desde ese momento.
Las declaraciones de funciones son diferentes.

**Una declaración de función se puede llamar antes de lo que se define.**
Por ejemplo, una declaración de función global es visible en todo el script, sin importar dónde se encuentre.
Eso se debe a algoritmos internos. Cuando JavaScript se prepara para ejecutar el script, primero busca declaraciones de funciones globales y crea las funciones. Podemos considerarlo como una "etapa de inicialización".
Y después de procesar todas las declaraciones de funciones, se ejecuta el código. Entonces tiene acceso a estas funciones.
Por ejemplo, esto funciona:

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

La declaración de funciones `sayHi` se crea cuando JavaScript se está preparando para iniciar el script y es visible en todas partes.
Si fuera una expresión de función, entonces no funcionaría:

```
sayHi("John"); // error!

let sayHi = function(name) {  // (*) no magic any more
  alert( `Hello, ${name}` );
};
```

Las expresiones de función se crean cuando la ejecución las alcanza. Eso sucedería solo en la línea `(*)`. Demasiado tarde.

Otra característica especial de las declaraciones de funciones es su alcance de bloque.

**En modo estricto, cuando una declaración de función está dentro de un bloque de código, es visible en todas partes dentro de ese bloque. Pero no fuera de eso.**

Por ejemplo, imaginemos que necesitamos declarar una función welcome() dependiendo de la variable age que obtengamos durante el tiempo de ejecución. Y luego planeamos usarlo algún tiempo después.

Si usamos la declaración de funciones, no funcionará como se esperaba:

```
let age = prompt("What is your age?", 18);

// conditionally declare a function
if (age < 18) {

  function welcome() {
    alert("Hello!");
  }

} else {

  function welcome() {
    alert("Greetings!");
  }

}

// ...use it later
welcome(); // Error: welcome is not defined
```

Esto se debe a que una declaración de función solo es visible dentro del bloque de código en el que reside.

**¿Cuándo elegir la declaración de función versus la expresión de función?**

Como regla general, cuando necesitamos declarar una función, lo primero que debemos considerar es la sintaxis de la declaración de funciones. Da

más libertad en cómo organizar nuestro código, porque podemos llamar a tales funciones antes de que sean declaradas.

Eso también es mejor para la legibilidad, ya que es más fácil buscar `function f(…) {…}` en el código que `let f = function(…) {…};`. Las declaraciones de funciones son más "llamativas".

.Pero si una declaración de función no nos conviene por alguna razón, o si necesitamos una declaración condicional (acabamos de ver un ejemplo), entonces se debe utilizar la expresión de función.

Más adelante cuando manipulemos el DOM veremos casos en los que utilizaremos las funciones anónimas para asignar funciones a los eventos de los objetos creados.

## 3.3.- Resumen de expresión de función

- Las funciones son valores. Se pueden asignar, copiar o declarar en cualquier lugar del código.
- Si la función se declara como una declaración separada en el flujo del código principal, eso se llama una "declaración de función".
- Si la función se crea como parte de una expresión, se llama "expresión de función".
- Las declaraciones de funciones se procesan antes de ejecutar el bloque de código. Son visibles en todas partes en el bloque.
- Las expresiones de función se crean cuando el flujo de ejecución las alcanza.

En la mayoría de los casos cuando necesitamos declarar una función, es preferible una declaración de función, ya que es visible antes de la declaración misma. Eso nos da más flexibilidad en la organización del código, y generalmente es más legible.

Por lo tanto, deberíamos usar una expresión de función solo cuando una declaración de función no sea adecuada para la tarea.

Hemos visto un par de ejemplos de eso, y veremos más en el futuro.

## 4.- Fat arrow (“funciones de flecha”)

Hay otra sintaxis muy simple y concisa para crear funciones, que a menudo es mejor que las expresiones de funciones.

Es una característica incluida en el ES6 y que permite simplificar mucho la declaración y uso de las funciones. Esta sintaxis viene asociada a la ejecución de funciones llamadas de **callbak** que generalmente se asocia a las operaciones para manipular los Arrays que veremos más adelante

Se llaman "fat arrow", porque se ve así:

```
let func = (arg1, arg2, ...argN) => expression
```

Esto crea una función `func` que acepta argumentos `arg1..argN`, luego evalúa el lado derecho con `expression` y devuelve su resultado.

En otras palabras, es la versión más corta de:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Veamos un ejemplo concreto:

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

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

Como se puede ver, `(a, b) => a + b` significa una función que acepta dos argumentos nombrados `a` y `b`. Tras la ejecución, evalúa la expresión `a + b` y devuelve el resultado.

Hay que tener en cuenta en la sintaxis de las Fat Arrow:

- ***Si solo tenemos un argumento***, se pueden omitir los paréntesis alrededor de los parámetros, lo que lo hace aún más corto.
  Por ejemplo:

```
let double = n => n * 2;
// roughly the same as: let double = function(n) { return n * 2 }
alert( double(3) ); // 6
```

- **Si no hay argumentos**, los paréntesis estarán vacíos (pero deberían estar presentes):

```
let sayHi = () => alert("Hello!");

sayHi();
```

Las funciones de flecha se pueden usar de la misma manera que las expresiones de función.
Por ejemplo, para crear dinámicamente una función:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");

welcome();
```

Las funciones fat arrow pueden parecer desconocidas y poco legibles al principio, pero eso cambia rápidamente a medida que los ojos se acostumbran a la estructura.
Son muy convenientes para acciones simples de una línea, cuando sólo necesitamos operaciones básicas o estamos en encadenamiento de funciones.

## 4.1.- Funciones Fat Arrow multilínea

Los ejemplos anteriores tomaron argumentos de la izquierda `=>` y evaluaron la expresión del lado derecho con ellos.

A veces necesitamos algo un poco más complejo, como múltiples expresiones o declaraciones. También es posible, ***pero debemos encerrarlos entre llaves***. Además ***tendremos que utilizar un return*** con el valor que devuelva la función. Por ejemplo:

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit "return"
};

alert( sum(1, 2) ); // 3
```

*Las funciones fat arrow tienen otras características interesantes como pueden ser el uso de* ***this*** *que veremos cuando trabajemos con objetos.*

## 4.2.- Resumen de funciones Fat Arrow

Las funciones de flecha son útiles para líneas simples. Vienen en dos formatos:

1. Sin llaves: `(...args) => expression` el lado derecho es una expresión: la función lo evalúa y devuelve el resultado.
2. Con llaves: los `(...args) => { body }` nos permiten escribir varias declaraciones dentro de la función, pero necesitamos un explícito `return` para devolver algo.