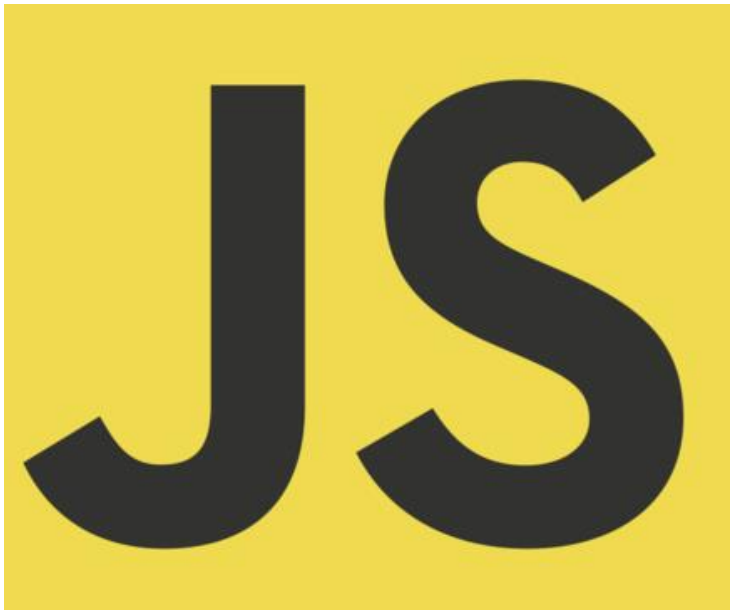


Funciones

Creación y sintaxis

```
function nombreFunción ([parámetro1]....[, parámetroN] ) {
  // instrucciones
   [ return valor; ]
}
```

- Se declaran mediante la palabra reservada **function** seguida de un nombre de función y unos paréntesis que pueden contener o no, uno o más argumentos o parámetros, separados (en el caso de ser más de uno) por comas.
- El código que ejecutará la función en el momento en que se llame. Este código va encerrado entre llaves a continuación de la declaración.
- La última instrucción de una función debe ser (en caso de existir) la de retornar el valor al programa principal desde donde fue llamada. No es obligatorio que las funciones devuelvan un valor

Invocar a una función

```
function saludar(){
     alert(“hola”);
}
saludar();
```

saludar; => permite guardar la función dentro de una variable.

```
dihola = saludar;
dihola()
```

Funciones anónimas

- No tienen nombre
- Muy empleadas al trabajar con eventos
- Sintaxis básica:

```
function (){}
```

Ejemplo de función anónima que suma dos valores:

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

- Las funciones anónimas pueden ser asignadas una variable

```
var suma = function (x,y) {return x+y;}
```

- Invocamos a la función como sigue:

```
var resultado = suma (2, 5);
alert(resultado);
```

- La variable suma puede usarse como si fuera una función
- Las funciones anónimas se invocan usando el nombre de la variable que las almacena

Funciones anónimas

- Otra forma de definir una función anónima es empleando el constructor del objeto **function**

```
var funcionSuma = new function
(“x”,”y”,”return x+y;”);
var resultado = funcionSuma(5,2);
```

Funciones anónimas autoinvocadas

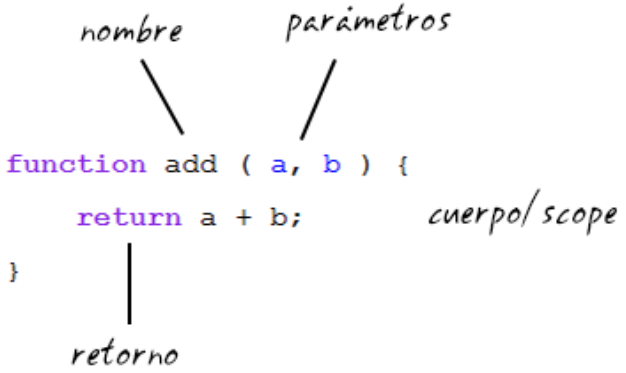
- Para llamar a una función, hay varios métodos
  - Emplear un evento
  - Llamar directamente a la función en el código
  - **Autoinvocación**
- Para que una función se ejecute automáticamente sin ser llamada, se añade () al final y se coloca la definición también entre paréntesis. Ejemplo, sea la siguiente función anónima:

```
function (){ alert (“Hola”);}
(function (){alert (“Hola”);} () )
```

- En el siguiente ejemplo, se autoinvoca a una función que permite calcular la suma de 2 números. En este caso se inicializan los argumentos con los valores deseados:
- `(function (x = 2 ,y = 5){return x+y;} () )`

Funciones Flecha (arrow functions)

- Aparecen en ES6 (ECMAScript 2015)
- Tienen una sintaxis más simple que una función convencional y generan un código más limpio
- **Son siempre anónimas**
- No pueden emplearse como constructores
- Una función tradicional se define de la siguiente forma:



Funciones Flecha

- La función flecha equivalente a la función anterior:

```
var suma = ( x, y ) => x + y;
```

- A la hora de invocarla, empleamos la variable, por ejemplo:

```
console.log( suma( 2, 3 ) );
```

- Utilizan una sintaxis declarativa
  - Elimina la palabra reservada *function* y se especifican los parámetros mediante paréntesis.
  - Elimina las llaves que delimitan el ámbito *abriéndolo* con una flecha.
  - Se puede eliminar la palabra reservada *return*.
- Para más información sobre las funciones flecha:
  - https://www.w3schools.com/js/js_scope.asp
  - https://www.espai.es/blog/2018/01/funciones-flecha-en-javascript/
  - http://www.etnassoft.com/2016/06/22/las-funciones-flecha-en-javascript-parte-1/

Parámetros

- ***Los parámetros son nombres que se pasan entre paréntesis a la función***
- ***Una función puede tener o no parámetros***
- ***Los argumentos son los valores que se pasan a esa función***
- ***En JS, no se indica el tipo de los parámetros pasados***
- ***No se comprueba el número de los argumentos definidos***
- ***Los parámetros se inicializan automáticamente en el momento de llamar a la función con los valores que se le pasan en la llamada.***
- Al realizar la llamada a la función se escribirán los valores (o variables que tiene dichos valores) en el mismo orden en el que figuran en la definición.
- ***El número y posición de los argumentos en la definición y en la llamada a la función debe ser el mismo.***

```
function devolverMayor(a,b) {
   if (a>b) {
    return a;
  } else {
    return b;
   }
}
devolverMayor(5,9);
```

Parámetros por defecto

- Una función puede ser llamada con **menos** argumentos de los que ha sido definida y falta alguno de los parámetros

```
function suma (x,y){return x+y;}
var resultado = suma (2);
```

- La variable resultado tendrá el valor NaN. (internamente se realizara la operación 2+NaN, que devuelve NaN)
- Por ello es buena idea comprobar los parámetros de la siguiente forma:

```
function suma (x,y){
  if (y === undefined) y = 0;
  return x+y;
}
```

- De esta forma nos aseguramos que, si no hay un segundo argumento, y tomará el valor 0.
- Desde ES6 es posible:

```
function suma (a=1, b=1) {
  return a+b;
}
suma();//2
```

Parámetros por exceso

- Una función puede ser llamada con **más** argumentos de los que ha sido definida y hay más parámetros de los que espera
- Los valores recibidos pueden capturarse a través de un objeto incluido en la función llamado **arguments**. (**OJO!, El objeto arguments no se encuentra disponible en las funciones flecha**)
- Arguments es un array y podemos emplear sus métodos y propiedades (en este caso con la propiedad length, accedemos al número de argumentos pasados)

```
funcion valores () {
        alert (arguments.length);
}
valores ( 2, 1, 0, “hola” );
//Se visualizará el valor 4
```

- Al ser arguments un array, podremos recorrerlo para recuperar los parámetros

```
funcion valores () {
        for (var i=0; i< arguments.length; i++ ){
                alert (“Argumento ”+ i+ “=“+ arguments[i]);
        }
}
valores ( 2, 1, 0 );
//Argumento 0=2
//Argumento 1=1
//Argumento 2=0
//Argumento 3=hola
```

Parámetros de tipo Rest

- Los parámetros de tipo Rest se usan cuando nuestra función va a recibir un número indeterminado de parámetros.
- Mientras que un parámetro clásico es un valor primitivo (string, booleano, etc), un parámetro de tipo Rest es un array
- Dentro de ese array están los parámetros clásicos

```
function suma(...numeros){
  var resultado = 0;
  numeros.forEach(function (numero) {
    resultado += numero;
  });
  return resultado;
}
```

Parámetros de tipo Rest

- Si usamos un parámetro de tipo Rest no podemos poner ningún otro después, pero sí podríamos ponerlos antes.

```
function suma(numero,...numeros){
  var resultado = 0;
  resultado += numero * 10;
  numeros.forEach(function (numero) {
    resultado += numero;
  });
  return resultado;
}
function suma(...numeros, numero){
  var resultado = 0;
  resultado += numero * 10;
  numeros.forEach(function (numero) {
    resultado += numero;
  });
  return resultado;
}
suma(5,1,2,3,4);
```

La primera función funciona bien. La segunda falla ya al declararla y arroja un error de sintaxis que indica como que despues de un Rest no puede haber otro parámetro.

Parámetros de tipo Spread

- Los parámetros spread convierten un array a una serie de parámetros.
- La sintaxis es igual a la de los parámetros Rest

```
var parametros = [1, 8, 3, 7];
function suma(a,b,c,d){
     return a+b+c+d;
}
suma(...parametros); // DEVUELVE 19
```

Explicación parámetros Rest y Spread:

https://www.youtube.com/watch?v=xoLsbVGJIyE

Ámbito de las variables

- Las **variables locales** definidas dentro de una función, solo están accesibles dentro de esa función. Se pueden declarar variables con el mismo nombre locales a funciones. La variable local desaparece cuando finaliza la función.
- Una variable declarada mediante la palabra reservada **var** es local al lugar donde se define, siendo global en caso contrario.
- El alcance de una **variable global**, se limita al documento actual que está cargado en la ventana del navegador o en un **frame**.
- Cuando se inicializa una variable como variable global, todas las instrucciones del script (incluidas las instrucciones que están dentro de las funciones) tendrán acceso directo al valor de esa variable.
- Todas las instrucciones podrán leer y modificar el valor de esa variable global.
- Al cerrar una página, todas las variables definidas en esa página se eliminarán de la memoria para siempre. Si es necesario que el valor de una variable persista de una página a otra, hay que utilizar técnicas que permitan almacenar esa variable (cookies, etc.).
- Reutilizar el nombre de una variable global como local es uno de los ***bugs*** más difíciles de encontrar en el código ya que la variable local, en momentos puntuales, ocultará el valor de la variable global, sin avisarnos de ello.
- No reutilizar un nombre de variable global como local en una función
- No declarar una variable global dentro de una función

Variables automáticamente globales

- Cuando asignamos un valor a una variable que no ha sido declarada antes, esta se convierte en global. Supongamos la siguiente función.

```
function mostrarMensaje(){
      mensaje= “Esto es un mensaje”
      alert (mensaje);
}
mostrarMensaje();
```

- La variable mensaje será por tanto global y estará accesible desde cualquier punto de la página.
- Para evitar esto, podemos utilizar **use strict** al principio de nuestro código

```
"use strict";
saludo = "hola";
```

- Este código producirá un error ya que la variable saludo no está definida.

Variables globales y locales con el mismo nombre

- Sea el código siguiente

```
var variable = “fuera”;
function ambito (){
  var variable = “Dentro”;
  alert (variable);
}
alert (variable); //devuelve fuera
ambito(); //devuelve dentro
alert (variable); // devuelve fuera
```

Variables globales y locales con el mismo nombre

- Sea el código siguiente

```
var variable = “fuera”;
function ambito (){
   variable = “Dentro”; //se modifica el valor de la
   //variable global
   alert (variable);
}
alert (variable); //devuelve fuera
ambito(); //devuelve dentro
alert (variable); // devuelve dentro
```

Hoisting

- En Javascript, cuando se define una variable en el interior de una función, el intérprete interno pasa a ubicarla al comienzo de su contexto (*la eleva*).

```
x = 5;
alert(x);
var x; // Declaración de x
alert(x);
```

- Las variables declaradas con let o const no son elevadas
- Para evitar errores y como norma general, es importante declarar todas las variables al principio del ámbito en el que se usan
- https://developer.mozilla.org/es/docs/Glossary/Hoisting

Funciones anidadas (closures)

- Se puede anidar una función dentro de otra función.
- La función anidada (interna) es privada a su función contenedora (externa).

```
function inicia() {
  var nombre = "Mozilla"; // 'nombre' es una variable local creada por la función 'inicia'
  function muestraNombre()
{// 'muestraNombre' es una función interna (un closure)
    alert(nombre); // dentro de esta función usamos una variable declarada en la función padre
  }
  muestraNombre();
}
inicia();
```

- La función **inicia()** crea una variable local llamada nombre, a continuación, define una función denominada muestraNombre().
- **muestraNombre()** es una función interna (un closure) definida dentro de inicia(), y sólo está disponible en el cuerpo de esa función.
- muestraNombre() no tiene ninguna variable propia, lo que hace es reutilizar la variable nombre declarada en la función externa.

Funciones anidadas

El siguiente código nos ofrece el mismo resultado:
Lo que lo hace diferente (e interesante) es que la función externa nos ha devuelto la función interna muestraNombre() antes de ejecutarla.

```
function creaFunc() {
  var nombre = "Mozilla";
  function muestraNombre() {
    alert(nombre);
  }
  return muestraNombre;
}

var miFunc = creaFunc();
miFunc();
```

miFunc es un closure o cierre. Un closure es un tipo especial de objeto que combina dos cosas: una función, y el entorno en que se creó esa función. El entorno está formado por las variables locales que estaban dentro del alcance en el momento que se creó el closure. En este caso, miFunc es un closure que incorpora tanto la función muestraNombre como el string "Mozilla" que existían cuando se creó el closure.

Funciones predefinidas del lenguaje

Propiedades globales de JavaScript	
Propiedad	**Descripción**
Infinity	Un valor numérico que representa el infinito positivo/negativo.
NaN	Valor que no es numérico "Not a Number".
undefined()	Indica que a esa variable no le ha sido asignado un valor.

Métodos globales de JavaScript	
Método	**Descripción**
decodeURI()	Decodifica los caracteres especiales de una URL excepto: , / ? : @ & = + $ #
decodeURIComponent()	Decodifica todos los caracteres especiales de una URL.
encodeURI()	Codifica los caracteres especiales de una URL excepto: , / ? : @ & = + $ #
encodeURIComponent()	Codifica todos los caracteres especiales de una URL.
escape()	Codifica caracteres especiales en una cadena, excepto: * @ - _ + . /
eval()	Evalúa una cadena y la ejecuta si contiene código u operaciones.
isFinite()	Determina si un valor es un número finito válido.
isNaN()	Determina cuándo un valor no es un número.
Number()	Convierte el valor de un objeto a un número.
parseFloat()	Convierte una cadena a un número real.
parseInt()	Convierte una cadena a un entero.
unescape()	Decodifica caracteres especiales en una cadena, excepto: * @ - _ + . /

eval

Algunos ejemplos:

```
eval("x=50;y=30;document.write(x*y)"); //1500
document.write("<br />" + eval("8+6")); //14
document.write("<br />" + eval(x+30));  //80
```

Veamos otro ejemplo:

```
var string1 = "foo"; var string2 = "bar";
var funcName = string1 + string2;
function foobar(){
alert( 'Hello World' );
}
eval( funcName + '()' );  // Hello World
```

Inconvenientes:

- Resulta complicado de leer. La mezcla entre referencias a variables y código válido entrecomillado da lugar a largas cadenas difíciles de seguir:
- Eval compromete la seguridad de la aplicación ya que proporciona un protagonismo excesivo a la cadena suministrada. Validar el *eval code* resulta por lo general complejo.