



Anexo: Sintaxis básica de Javascript

- C.F.G.S. DAW
- 6 horas semanales
- Mes aprox. de impartición: Oct
- iPasen - cjaedia071@g.educaand.es

Carmelo José Jaén Díaz

Índice



Objetivo

Glosario

Operador ternario

Tipos de datos

Operadores de tipos de datos

Funciones

Ámbito de variables

OBJETIVO



- Comprender la sintaxis básica de JavaScript para poder realizar pequeños scripts funcionales.
- Entender las reglas básicas del lenguaje JavaScript.
- Conocer la utilización de las sentencias básicas de JavaScript.
- Descubrir elementos propios de JavaScript como los eventos.

GLOSARIO



Case-sensitive. Expresión informática que se aplica a los textos en los que tiene relevancia escribir un carácter en mayúsculas o minúsculas y que significa "sensible a mayúsculas y minúsculas".

Consola de JavaScript. Herramienta del programador para comunicar su programa JavaScript con el exterior. Se invoca pulsando generalmente la tecla F12.

ES6 o ES2015 (ECMAScript). Especificación de lenguaje JavaScript publicada por ECMA International. Los navegadores utilizan una implementación de ECMAScript y acceso al DOM para manipular las páginas web.

GMT (Greenwich Mean Time). Estándar de tiempo que se refería al tiempo solar medio en el Real Observatorio de Greenwich y que dejó de ser utilizado por la comunidad científica en 1972, cuando se reemplazó por el UTC.

Listener. Código responsable de controlar los eventos. Están a la escucha (de ahí su nombre) y, cuando ocurre un evento, se ejecuta el código que el programador haya implementado.

Operador ternario. Operador que toma tres argumentos. La ventaja que ofrece es que puede reducir varias líneas de código en una sola.

GLOSARIO



UTC (Coordinated Universal Time). Principal estándar de tiempo, que casi siempre es sinónimo de GMT, y se obtiene a través del tiempo atómico internacional.

OPERADOR TERNARIO



El operador ternario es un operador que permite realizar una sentencia condicional (es decir, un if-then-else) en una sola línea.

Un operador ternario está formado tres argumentos (de ahí su nombre); o más concretamente, por cinco partes:

- La condición que se puede o no cumplir.
- Un signo de interrogación.
- Valor si la condición es cierta.
- Dos puntos.
- Valor si la condición es falsa.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Operador ternario</title>
  </head>

  <body>
    <script>
      // (CONDICION) ? RESULTADO_CIERTO : RESULTADO_FALSO
      var edad = 22;
      var puedeVotar = (edad > 18) ? "Puede votar" : "No
      puede votar";
      alert (puedeVotar);

    </script>
  </body>
</html>
```

TIPOS DE DATOS.

Números en notación científica.



Se utiliza para representar números muy grandes o muy pequeños. Su sintaxis es la siguiente:

```
var numeroGrande = 12e10;
```

- La *e* hace referencia al exponente
- En este caso el número, 12×10^{10} .

```
var numeroPequeño = 12e-10;
```

- En este caso el número, $12 \times 10^{-10} = 12 \times 1/10^{10}$.

TIPOS DE DATOS.

Arrays.



Un array o vector, es un conjunto de valores que se almacenan en una especie de estructura. Los array se declaran de la siguiente forma:

```
var lista = [];
```

- Los valores que se almacenan en el array irán dentro de los corchetes separados por comas “,”.
- Los valores pueden ser de tipo entero, decimal, char, string o cadena, booleano o incluso otro array (se estudiará más adelante).

- Para referirme al primer valor del array anterior se emplea *lista[0]*. Recuerda que en JavaScript el primer valor de un array hace referencia a la posición 0.

```
// ARRAY
```

```
var lista = [1, "Hola", 6, "Caracola"];  
window.alert(lista[0]); //Muestra 1  
window.alert(lista[1]); //Muestra Hola  
window.alert(lista[2]); //Muestra 6
```


OPERADORES DE TIPOS DE DATOS.

Typeof.



El operador *typeof* se sitúa delante de cualquier variable y nos devuelve de qué tipo es esa variable.

```
var animal = {nombre: "Lola", tipo: "Hamster", raza: "Ruso", edad: 1};
```

- Los valores que se almacenan en el objeto irán dentro de las llaves separados por comas “,”.
- Los valores pueden ser de tipo numérico, string o cadena, booleano o incluso funciones.

```
var cadena = "Carmelo";  
var cierto = true;  
var lista = [1, "Hola", 6, "Caracola"];
```

```
// OPERADOR typeof  
alert (typeof cadena); //Devuelve string  
alert (typeof cierto); //Devuelve boolean  
alert (typeof lista); //Tanto arrays como  
objetos devuelve object
```

Esto es debido a que en JavaScript los arrays en realidad son un tipo de objeto.

OPERADORES DE TIPOS DE DATOS.

Instanceof.



El operador *instanceof* se sitúa delante de cualquier objeto y nos devuelve *true* si es una instancia de otro especificado. Es decir, verifica si un objeto se corresponde con el tipo de dato indicado a continuación.

En realidad, *instanceof* comprueba si un objeto en su cadena de prototipos, contiene la propiedad prototype de un constructor. Pero esto son palabras mayores por el momento. Veremos qué son los prototipos y qué son los constructores más adelante.

//INSTANCEOF: nos devuelve true si un objeto es una instancia de otro especificado

```
var animales = ["perro", "gato",  
"hamster"];
```

```
window.alert (animales instanceof Object);  
//Devuelve true  
window.alert (animales instanceof Array);  
//Devuelve true  
window.alert (animales instanceof String);  
//Devuelve false
```

TIPOS DE DATOS.

Undefined.



Este tipo de dato nos sirve para controlar los tipos de datos que no están definidos. Por ejemplo si se declara una variable y no se define al utilizar el operador *typeof* nos devolverá:

```
var lista;  
window.alert(typeof lista); //Muestra undefined
```

RECUERDA

- El tipo de dato que finaliza un array es *undefined*.
- Los pares de un objeto que no se han asignado también son undefined (no son null).

```
var coche = { make: "Ford", model: "Mustang", year: 1969};  
window.alert(coche.color); // Muestra undefined
```

TIPOS DE DATOS.

Null.



Este tipo de dato nos sirve para mantener el tipo de objeto de la variable pero con un valor vacío. Por ejemplo si se declara una variable y se define con un valor vacío al utilizar el operador *typeof* nos devolverá:

```
var coche = "";  
window.alert(typeof coche); //Muestra string
```

```
var coche = null;  
window.alert(typeof coche); //Muestra object
```

```
var coche;  
window.alert(typeof coche); //Muestra undefined
```

TIPOS DE DATOS.

Diferencia entre Null y Undefined.



Este tipo de dato nos sirve para mantener el tipo de objeto de la variable pero con un valor vacío. Por ejemplo si se declara una variable y se define con un valor vacío al utilizar el operador *typeof* nos devolverá:

//NULOS vs NO DEFINIDOS

```
typeof undefined; //Devuelve undefined
```

```
typeof null; //Devuelve objeto
```

```
null === undefined; //Devuelve false
```

```
null == undefined; //Devuelve true
```

En el último caso como solo se compara el valor y no el tipo de dato, a efectos de JavaScript son lo mismo ya que no contienen nada.

FUNCIONES.

Introducción.



Las funciones son uno de los elementos más importantes de la programación estructurada.

Una función es un grupo de sentencias o instrucciones que realizan una tarea concreta, por ejemplo calcular un valor en base a unas operaciones matemáticas. Para utilizar una función, en primer lugar hay que definirla en algún lugar de nuestro programa y posteriormente, llamarla.

Una función debe seguir unas normas para ser definida, es decir, debemos escribirla utilizando los mismos elementos. Su estructura habitual es:

```
function <nombre de la funcion> (<parametros opcionales separados por comas>) {  
    <instrucciones>  
}
```

FUNCIONES.

Introducción.



Sin embargo no es la única estructura puesto que las funciones tienen multitud de variantes:

- Podemos escribir funciones sin parámetros.

Funciones que no necesitan (no se le pasan) datos externos para realizar la tarea para la que fueron concebidas.

- Las funciones pueden devolver un dato o no.

Para devolver un dato se hace uso del comando *return*.

- Podemos escribir una función sin paréntesis y guardarla en una variable.
- Podemos escribir una función sin nombre (función anónima).
- Podemos auto-invocar una función, esto es sin tener que llamarla.

FUNCIONES.

Función sin parámetros.



Normalmente para asignar la salida o argumento de una función a una variable, escribiríamos:

FUNCIÓN SIN PARÁMETROS

```
function saludo () {  
    alert("¡Hola!");  
}
```

//Los paréntesis () invocan a la función

```
var resultado_saludo = saludo();
```

```
alert (resultado_saludo);    //Devuelve undefined ya que la función no  
                             devuelve nada
```


FUNCIONES.

Función sin parámetros.



Podemos escribir una función sin paréntesis y guardarla en una variable.

FUNCIÓN SIN PARÁMETROS

```
function saludo () {  
    alert("¡Hola!");  
}
```

//Si se escribe en nombre de la función sin paréntesis, ésta no se invoca pero JS tampoco lo interpreta como un error.

```
saludo;
```

//Sin paréntesis se puede guardar la función en una variable

```
var resultado_saludo2 = saludo;
```

```
alert(resultado_saludo2); //El alert muestra el contenido de la función
```

FUNCIONES.

Función con parámetros.



Normalmente para asignar la salida o argumento de una función a una variable, escribiríamos:

FUNCIÓN CON PARÁMETROS

```
function producto (a, b){  
    return a * b;  
}
```

`producto(3, 4);` *//Los paréntesis () invocan a la función aunque no muestra nada*

//Para mostrar la salida de la función tenemos que asignarla a una variable y mostrar la variable

```
var resultado_producto = producto(3, 4);  
alert (resultado_producto);
```

//Podemos introducir en el alert directamente la función

```
alert (producto(5,6));
```

FUNCIONES.

Función con parámetros.



Podemos escribir una función sin paréntesis y guardarla en una variable.

FUNCIÓN CON PARÁMETROS

```
function producto (a, b){  
    return a * b;  
}
```

//Sin paréntesis se puede guardar la función en una variable

```
var resultado_producto = producto;  
alert(resultado_producto); //El alert muestra el contenido de la función
```

FUNCIONES.

Funciones anónimas



Como podéis imaginar, las funciones anónimas no tienen nombre. Esto quiere decir que no es necesario acompañar la palabra *function* a un nombre de función.

RECUERDA

Ya sabes que normalmente, cuando definimos una función, le asignamos un nombre y cero o más parámetros entre paréntesis; luego, para llamar a esa función indicamos el nombre y los parámetros (si los hubiera) entre paréntesis y esa función se ejecuta.

El uso de las funciones anónimas va mucho más allá: podemos asignarlas a variables, utilizarlas como parámetros de otra función, llamar a la función automáticamente en el momento que se define, etc.

FUNCIONES.

Funciones anónimas



Por tanto, a diferencia de la estructura de una función que vimos anteriormente, la función anónima se define:

```
function (<parametros opcionales separados por comas>) {  
    <instrucciones>  
}
```

Por ejemplo, la función anterior *producto* quedaría:

```
function (a, b){  
    return a * b;  
}
```

FUNCIONES.

Funciones anónimas



Podemos asignar dicha función a una variable tal que así:

```
//Ahora la variable multiplicación contiene el código de la función  
var multiplicacion = function (a, b){ return a * b; }  
  
//Para llamar a la función, escribiríamos (aunque no mostraría nada)  
multiplicación (3, 5);  
//Para ello, debemos asignarla a una variable o mostrarla con alert();  
var resultado = multiplicacion (3, 5);  
alert(resultado);
```

La finalidad de esto, es poder usar la variable multiplicacion como si fuera una función.

Las funciones anónimas se invocan usando el nombre de la variable que las almacena.

FUNCIONES.

Constructor *function*.



Aunque será algo que estudiaremos en el RA 3, cuando veamos objetos, podemos definir una función anónima como un objeto *function*. De hecho, si hacemos *typeof* **producto**, obtenemos *function*.

```
var producto = new Function ("a", "b", "return a*b;");  
    var resultado2 = producto(5,7);  
    alert (resultado2);
```

```
var multiplicacion = function (a, b){ return a * b; }
```

```
//Para llamar a la función, escribiríamos (aunque no mostraría nada)  
    multiplicación (3, 5);  
//Para ello, debemos asignarla a una variable o mostrarla con alert();  
    var resultado = multiplicacion (3, 5);  
    alert(resultado);
```

FUNCIONES.

Funciones anónimas autoinvocadas.



Anteriormente habíamos visto que las funciones podían invocarse mediante eventos (clicando un botón) o en el código (asignándoselas a una variable). Ahora veremos cómo invocarlas automáticamente.

Para que una función se ejecute automáticamente sin ser llamada, al final de la función, esto es después de la última llave tenemos que incluir un (). Y para que se ejecute como una expresión debemos introducirla entre dos paréntesis.

//FUNCIONES ANÓNIMAS AUTOINVOCADAS

```
(function () { alert ("¡Hola!");})();
```

Este tipo de funciones se utilizan mucho cuando se trabajan con eventos.

FUNCIONES.

Parámetros y argumentos.



Los **parámetros** son los nombres que aparecen en la definición de una función. Por su parte, los **argumentos** son los valores que le pasamos (y que, por tanto, recibe) una función cuando la invocamos.

Javascript es un lenguaje muy permisivo en cuanto los tipos de datos (tiene lo que se llama tipificación débil y dinámica). Eso quiere decir que no tenemos que indicar de qué tipo van a ser las variables que declaramos. En el caso de los parámetros y argumentos ocurre exactamente lo mismo: **no indicamos ni tipo ni número de parámetros y tampoco comprobamos que se correspondan.**

Aunque parezca increíble, Javascript permite que el número de parámetros de una función sea diferente del número de argumentos que se le pasan. En la mayoría de lenguajes de programación esto sería un problema. Pero en Javascript tenemos maneras de gestionar parámetros por exceso y por defecto.

FUNCIONES.

Parámetros por defecto.



Ocurre cuando llamamos a una función con menos argumentos de los declarados. Los valores que faltan no están definidos.

```
function suma (a, b){  
    return a + b;  
}  
  
var resultado = suma (4);  
alert (resultado);  
//Devuelve NaN ya que JS no  
entiende cómo gestionar la función  
suma con un solo argumento
```

```
function suma (a, b){  
    if (b === undefined)  
        b = 0;  
    return a + b;  
}  
  
var resultado = suma (4);  
alert (resultado);  
//Para agregar control de errores,  
comprobamos si el parámetro se ha pasado  
como argumento. En caso negativo, se le  
asigna un valor por defecto.
```

FUNCIONES.

Parámetros por exceso.



Ocurre cuando llamamos a una función con más argumentos de los que ha sido declarada. Los valores que nos llegan pueden capturarse a través de un objeto (incluido en la función) llamado arguments.

El objeto arguments es de tipo array.

```
function valores () {  
    alert ("El número de argumentos es "+arguments.length);  
    for (var i=0; i < arguments.length; i++){  
        alert ("Argumento "+(i+1)+"=" +arguments[i]);  
    }  
}
```

`valores (4, 6, 8, 2, 7, 5);`

//Mediante el uso de este objeto podemos controlar la gestión de argumentos por exceso.

FUNCIONES.

Ámbito de variables.



El ámbito de variables, en inglés scope, tanto en Javascript como en cualquier otro lenguaje de programación, define la zona del programa en la que podemos utilizar la variable.

En Javascript podemos hablar de dos tipos de ámbito de variables:

- Ámbito de variables global.
- Ámbito de variables local.

¿Sabías que la diferencia entre una variable global y local en Javascript es simplemente el uso de la palabra reservada “var”? Si declaras una variable en cualquier parte del programa sin usar “var” automáticamente se convierte en una variable global. ¡Así que ten cuidado! ¡El uso de las variables globales no es muy recomendado!

FUNCIONES.

Ámbito de variables.



Pero además, se pueden dar otros casos con los ámbitos de las variables que conviene conocer, como:

- Variables automáticamente globales (que aunque se definan dentro de un bloque de código, son globales).
- Variables globales y locales con el mismo nombre.
- Variables globales redefinidas en una función.

A partir de la versión ES6 (EcmaScript2015), podemos utilizar “let” para declarar variables. “let” tiene limitado el ámbito al bloque de ejecución, expresión o declaración, mientras que “var” limita el ámbito o bien a la función en la que se define o al ámbito global si está fuera de ella.

FUNCIONES.

Ámbito de variables *var*, *let* y *const*.



Con ES2015 (ES6) aparecieron muchas características nuevas y llamativas.

Una de las características que llegaron con ES2015 (ES6) es la adición de **let** y **const**, que se pueden utilizar para la declaración de variables. La pregunta es, ¿qué las hace diferentes del viejo **var** que hemos venido utilizando?

A continuación, se estudia el ámbito (alcance), uso y hoisting de **var**, **let** y **const**.

FUNCIONES.

Ámbito de variables *var*.



Antes de la llegada de ES6, las declaraciones **var** eran las que mandaban. Sin embargo, hay problemas asociados a las variables declaradas con **var**. Por eso fue necesario que surgieran nuevas formas de declarar variables. En primer lugar, vamos a entender más sobre **var** antes de discutir esos problemas.

ÁMBITO DE VAR

El ámbito, significa esencialmente dónde están disponibles estas variables para su uso. Las declaraciones **var** tienen un ámbito global o un ámbito de función/local, dependiendo de dónde sea declarada la función.

El ámbito es global cuando una variable **var** se declara fuera de una función. Esto significa que cualquier variable que se declare con **var** fuera de una función está disponible para su uso en toda la pantalla.

var tiene un ámbito local cuando se declara dentro de una función. Esto significa que está disponible y solo se puede acceder a ella dentro de esa función.

FUNCIONES.

Ámbito de variables *var*.



Si realizamos esto:

```
var saludar = "hey, hola";

function nuevaFuncion() {
    var hola = "hola";
}
```

Aquí, *saludar* tiene un ámbito global porque existe fuera de la función mientras que *hola* tiene un ámbito local.

Así que no podemos acceder a la variable *hola* fuera de la función.

```
var tester = "hey, hola";

function nuevaFuncion() {
    var hola = "hola";
}
```

```
console.log(hola); // error: hola is
not defined
```

Obtendremos un error que se debe a que *hola* no está disponible fuera de la función.

FUNCIONES.

Ámbito de variables *var*.



!!!LAS VARIABLES CON VAR SE PUEDEN VOLVER A DECLARAR Y MODIFICAR!!!

Esto significa que podemos hacer esto dentro del mismo ámbito y no obtendremos un error.

```
var saludar = "hey, hola";  
  
var saludar = "dice Hola tambien";
```

y esto también:

```
var saludar = "hey, hola";  
  
saludar = "dice Hola tambien"; //Obteniendo el mismo resultado
```

FUNCIONES.

Ámbito de variables *var*.



HOISTING DE VAR

Hoisting es un mecanismo de JavaScript en el que las variables y declaraciones de funciones se mueven a la parte superior de su ámbito antes de la ejecución del código. Esto significa que si hacemos esto:

```
console.log (saludar);  
var saludar = "dice hola"
```

se interpreta así:

```
var saludar;  
console.log(saludar); // saludar is undefined  
saludar = "dice hola"
```

Entonces las variables con **var** se elevan a la parte superior de su ámbito y se inicializan con un valor de undefined.

FUNCIONES.

Ámbito de variables *var*.



PROBLEMA CON VAR

Hay una debilidad que viene con *var*. Usaré el ejemplo de abajo para explicarlo:

```
var saludar = "hey, hola";  
var tiempos = 4;  
  
if (tiempos > 3) {  
    var saludar = "dice Hola tambien";  
}  
console.log(saludar) // "dice Hola  
tambien"
```

Por lo tanto, como *tiempos* > 3 devuelve *true*, *saludar* se redefine para "*dice Hola tambien*". Aunque esto no es un problema si quieres redefinir *saludar* a conciencia, se convierte en un problema cuando no te das cuenta de que la variable *saludar* ha sido definida antes.

Si has utilizado *saludar* en otras partes de tu código, puede que te sorprenda la salida que puedes obtener. Esto probablemente causará muchos errores en tu código. Por eso son necesarios *let* y *const*.

FUNCIONES.

Ámbito de variables *let*.



let es ahora preferible para la declaración de variables. No es una sorpresa, ya que es una mejora de las declaraciones con ***var***. También resuelve el problema con ***var*** que acabamos de cubrir. Consideremos por qué esto es así.

LET TIENE UN ÁMBITO DE BLOQUE

Un bloque es un trozo de código delimitado por {}. Un bloque vive entre llaves. Todo lo que está dentro de llaves es un bloque.

Así que una variable declarada en un bloque con ***let*** solo está disponible para su uso dentro de ese bloque.

Permíteme explicar esto con un ejemplo:

FUNCIONES.

Ámbito de variables *let*.



```
let saludar = "dice Hola";  
let tiempos = 4;  
  
if (tiempos > 3) {  
    let hola = "dice Hola tambien";  
    console.log(hola); // "dice Hola tambien"  
}  
  
console.log(hola) // hola is not defined
```

Vemos que el uso de *hola* fuera de su bloque (las llaves donde se definió) devuelve un error. Esto se debe a que las variables ***let*** tienen un alcance de bloque.

FUNCIONES.

Ámbito de variables *let*.



LET PUEDE MODIFICARSE PERO NO VOLVER A DECLARARSE.

Al igual que **var**, una variable declarada con **let** puede ser actualizada dentro de su ámbito. A diferencia de **var**, una variable **let** no puede ser re-declarada dentro de su ámbito. Así que mientras esto funciona:

```
let saludar = "dice Hola";  
  
saludar = "dice Hola tambien";
```

esto devolverá un error:

```
let saludar = "dice Hola";  
  
let saludar = "dice Hola tambien"; // error: Identifier 'saludar' has already been  
declared
```

FUNCIONES.

Ámbito de variables *let*.



Sin embargo, si la misma variable se define en diferentes ámbitos, no habrá ningún error:

```
let saludar = "dice Hola";
if (true) {
  let saludar = "dice Hola tambien";
  console.log(saludar); // "dice Hola tambien"
}

console.log(saludar); // "dice Hola"
```

¿POR QUÉ NO HAY NINGÚN ERROR? Esto se debe a que ambas instancias son tratadas como variables diferentes, ya que tienen ámbitos diferentes. Este hecho hace que **let** sea una mejor opción que **var**. Cuando se utiliza **let**, no hay que preocuparse de si se ha utilizado un nombre para una variable antes, puesto que una variable solo existe dentro de su ámbito. Además, como una variable no puede ser declarada más de una vez dentro de un ámbito, entonces el problema discutido anteriormente que ocurre con **var** no sucede.

FUNCIONES.

Ámbito de variables *let*.



HOISTING DE LET

Al igual que *var*, las declaraciones *let* se elevan a la parte superior. A diferencia de *var* que se inicializa como *undefined*, la palabra clave *let* no se inicializa. Así que si intentas usar una variable *let* antes de declararla, obtendrás un *Reference Error*.

FUNCIONES.

Ámbito de variables *const*.



Las variables declaradas con *const* mantienen valores constantes. Las declaraciones *const* tienen similitudes con las declaraciones *let*.

LAS DECLARACIONES CONST TIENEN UN ÁMBITO DE BLOQUE

Al igual que las declaraciones *let*, solamente se puede acceder a las declaraciones *const* dentro del bloque en el que fueron declaradas.

FUNCIONES.

Ámbito de variables *const*.



CONST NO PUEDE MODIFICARSE NI VOLVER A DECLARARSE

Esto significa que el valor de una variable declarada con **const** es el mismo dentro de su ámbito. No se puede actualizar ni volver a declarar. Así que si declaramos una variable con **const**, no podemos hacer esto:

```
const saludar = "dice Hola";
```

```
saludar = "dice Hola tambien";// error: Assignment to constant variable.
```

ni esto:

```
const saludar = "dice Hola";
```

```
const saludar = "dice Hola tambien";// error: Identifier 'saludar' has already been  
declared
```

Por lo tanto, toda declaración **const**, debe ser inicializada en el momento de la declaración.

FUNCIONES.

Ámbito de variables *const*.



Este comportamiento es algo diferente cuando se trata de objetos declarados con **const**. Mientras que un objeto **const** no se puede actualizar, las propiedades de este objeto sí se pueden actualizar. Como resultado, si declaramos un objeto **const** como este:

```
const saludar = {  
  mensaje: "dice Hola",  
  tiempos: 4  
}
```

podemos hacer esto:

```
saludar.mensaje = "dice Hola tambien";
```

mientras que no podemos hacer esto:

```
saludar = {  
  palabras: "Hola",  
  numero: "cinco"
```

Esto actualizará el valor de saludar.mensaje sin devolver errores.

```
} // error: Assignment to constant variable.
```

FUNCIONES.

Ámbito de variables *const*.



HOISTING DE CONST

Al igual que *let*, las declaraciones *const* se elevan a la parte superior, pero no se inicializan.

EN RESUMEN

En caso de que no hayas notado las diferencias, aquí están:

- Las declaraciones *var* tienen un ámbito global o un ámbito función/local, mientras que *let* y *const* tienen un ámbito de bloque.
- Las variables *var* pueden ser modificadas y re-declaradas dentro de su ámbito; las variables *let* pueden ser modificadas, pero no re-declaradas; las variables *const* no pueden ser modificadas ni re-declaradas.
- Todas ellas se elevan a la parte superior de su ámbito. Pero mientras que las variables *var* se inicializan con undefined, *let* y *const* no se inicializan.
- Mientras que *var* y *let* pueden ser declaradas sin ser inicializadas, *const* debe ser inicializada durante la declaración.