En Javascript cualquier expresión es también una sentencia.

**SENTENCIA DE BLOQUE**

La sentencia de bloque es el tipo de sentencia más básico y se utiliza para agrupar sentencias. El bloque se delimita entre un par de llaves:

{ sentencia\_1; sentencia\_2;. . . sentencia\_n; }

**Ejemplo**

Los bloques de sentencias son comúnmente utilizados para sentencias de control de flujo  (ej. if, for, while).

while (x < 10) {

x++;

}

En este caso {x++;} es el bloque de sentencias.

**Importante**: Javascript no tiene ámbito a nivel bloque en versiones anteriores a ECMAScript 6. Las variables introducidas dentro de un bloque pertenecen a la función o script que lo contiene y el efecto de declararlas persiste más allá del bloque mismo. En otras palabras, los bloques no introducen un nuevo ámbito. Si bien, los bloques "Standalone" son válidos, no deberían ser utilizados en Javascript ya que no se comportan como los bloques de C o Java. Por ejemplo:

var x = 1;

{

var x = 2;

}

console.log(x); // imprime 2

Este código imprime el número 2 dado que la sentencia de la variable x dentro del bloque está en el mismo ámbito que la variable x definida antes del bloque. En C o Java el equivalente de este código imprimiría 1.

A partir de ECMAScript 6, se introduce el ámbito a nivel bloque utilizando let para declarar las variables.

**SENTENCIAS CONDICIONALES**

Una sentencia condicional es un conjunto de comandos que se ejecutan si una condición es verdadera. JavaScript soporta dos sentencias condicionales: if...else y switch

**SENTENCIA if…else**

Se utiliza la sentencia if para comprobar si la condición lógica es verdadera. Se utiliza la opción else para ejecutar un sentencia si la condición es falsa. A continuación se muestra un ejemplo de if...else:

if (condición) {sentencia\_1;} else { sentencia\_2;}

La condición puede ser cualquier expresión que se evalúa a true o false. Consultar [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean" \l "Description) para una explicación de cómo se evalúa true y false. Si la condición es verdadera, se ejecuta sentencia\_1 ; si no, se ejecuta sentencia\_2 . La sentencia\_1 y la sentencia\_2 pueden ser cualquier sentencia, incluyendo sentencias anidadas en if.

También podéis componer sentencias más complejas usando else if para tener múltiples condiciones, como se muestra a continuación:

if (condición\_1) {

sentencia\_1;

} else if (condición\_2) {

  sentencia\_2;

} else if (condición\_n) {

  sentencia\_n;

} else {

  ultima\_sentencia;

}

En el caso de condiciones múltiples solamente la primera condición lógica que evalúa a verdadero va a ser ejecutada. Para ejecutar múltiples sentencias, agruparlas dentro de sentencias de bloque ({... }). En general, usar siempre sentencias de bloque es una buena práctica, sobre todo cuando se anidan sentencias if:

if (condición) {

ejecutar\_sentencia\_1\_si\_condición\_es\_verdadera;

ejecutar\_sentencia\_2\_si\_condición\_es\_verdadera;

} else {

ejecutar\_sentencia\_3\_si\_condición\_es\_falsa;

ejecutar\_sentencia\_4\_si\_condición\_es\_falsa;

}

Es aconsejable no usar asignación simple dentro de una expresión condicional porque dicha asignación puede ser confundida con el comparador de igualdad cuando se lee de pasada el código. Por ejemplo, no usar el siguiente código:

if (x = y) {

/\* sentencias aquí \*/

}

Si necesitas usar una asignación dentro de una expresión de condición, una práctica común es poner paréntesis adicionales alrededor de la asignación. Por ejemplo:

if ((x = y)) {

/\* sentencias aquí \*/

}

**Valores falsos:**

Los siguientes valores se evaluarán como falso (también conocidos como valores [Falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy" \o "Falsy: A falsy value is a value that translates to false when evaluated in a Boolean context.)):

* false
* undefined
* null
* 0
* NaN
* la cadena vacía ("")

El resto de valores, incluidos todos los objetos, son evaluados como verdadero cuando son pasados a una sentencia condicional.

No confundir los valores primitivos booleanos true y false con el verdadero y falso del objeto [Boolean](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Boolean" \o "El objeto Boolean es un objeto envoltorio para un valor booleano (boolean).). Por ejemplo:

var b = new Boolean(false);

if (b) // Esta condición se evalua a true

**Ejemplo**

En el siguiente ejemplo, la función comprobarDatos devuelve verdadero si el número de caracteres en un objeto Text es tres; en otro caso, muestra una alerta y devuelve falso.

function comprobarDatos() {

if (document.form1.threeChar.value.length == 3) {

return true;

} else {

alert("Introduce exactamente tres caracteres. " +

document.form1.threeChar.value + " no es válido.");

return false;

}

}

**Switch**

Una sentencia switch permite a un programa evaluar una expresión e intentar igualar el valor de dicha expresión a una etiqueta de caso (case). Si se encuentra una coincidencia, el programa ejecuta la sentencia asociada. Una sentencia switch se describe como se muestra a continuación:

switch (expresión) {

case etiqueta\_1:

sentencias\_1

[break;]

case etiqueta\_2:

sentencias\_2

[break;]

...

default:

sentencias\_por\_defecto

[break;]

}

El programa primero busca una cláusula case con una etiqueta que coincida con el valor de la expresión y, entonces, transfiere el control a esa cláusula, ejecutando las sentencias asociadas a ella. Si no se encuentran etiquetas coincidentes, el programa busca la cláusula opcional default y, si se encuentra, transfiere el control a esa cláusula, ejecutando las sentencias asociadas. Si no se encuentra la cláusula default, el programa continúa su ejecución por la siguiente sentencia al final del switch. Por convención, la cláusula por defecto es la última cláusula, aunque no es necesario que sea así.

La sentencia opcional break asociada con cada cláusula case asegura que el programa finaliza la sentencia switch una vez que la sentencia asociada a la etiqueta coincidente es ejecutada y continúa la ejecución por las sentencias siguientes a la sentencia switch. Si se omite la sentencia break, el programa continúa su ejecución por la siguiente sentencia que haya en la sentencia switch.

**Ejemplo**

En el siguiente ejemplo, si tipoFruta se evalúa como "Plátanos", el programa iguala el valor con el caso "Plátanos" y ejecuta las sentencias asociadas. Cuando se encuentra la sentencia break, el programa termina el switch y ejecuta las sentencias que le siguen. Si la sentencia break fuese omitida, la sentencia para el caso "Cerezas" también sería ejecutada.

switch (tipoFruta) {

case "Naranjas":

console.log("Naranjas cuestan 0,59€ el kilo.");

break;

case "Manzanas":

console.log("Manzanas cuestan 0,32€ el kilo.");

break;

case "Plátanos":

console.log("Plátanos cuestan 0,48€ el kilo.");

break;

case "Cerezas":

console.log("Cerezas cuestan 3,00€ el kilo.");

break;

case "Mangos":

console.log("Mangos cuestan 0,56€ el kilo.");

break;

case "Papayas":

console.log("Mangos y papayas cuestan 2,79€ el kilo.");

break;

default:

console.log("Disculpa, no tenemos el tipo de fruta " + fruittype + ".");

}

console.log("¿Te gustaría tomar algo?");

**SENTENCIAS DE MANEJO DE EXCEPCIONES**

Puedes lanzar excepciones usando la sentencia throw y manejarlas usando las sentencias try…catch.

* [throw statement](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Control_de_flujo_y_manejo_de_errores#throw_statement)
* [try...catch statement](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Control_de_flujo_y_manejo_de_errores#try_catch_statement)

**Tipos de excepciones**

Prácticamente cualquier objeto puede ser lanzado en JavaScript. Sin embargo, no todos los objetos lanzados son creados igual. Mientras que es bastante común para lanzar números o strings como errores, frecuentemente son más efectivos utilizar uno de los tipos de excepciones específicamente creados para este propósito:

* [Excepciones ECMAScript](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales#Objetos_fundamentales)
* [DOMException](https://developer.mozilla.org/es/docs/Web/API/DOMException) and [DOMError](https://developer.mozilla.org/es/docs/Web/API/DOMError" \o "La interfaz DOMError describe un objeto de error que contiene un nombre de error.)

**Sentencia throw**

Utiliza la sentencia throw  para lanzar una excepción. Cuando lanzas una excepción, se especifica la expresión que contiene el valor para ser lanzado:

throw expresión;

Puedes lanzar cualquier expresión, no solo expresiones de un tipo específico. En el siguiente código lanzamos varias excepciones de varios tipos:

throw "Error2"; // Tipo string

throw 42; // Tipo número

throw true; // Tipo booleano

throw {toString: function() { return "¡Soy un objeto!"; } };

**Nota:** Puedes especificar un objeto cuando lanzas una excepción. A continuación, puedes hacer referencia a las propiedades del objeto en un bloque catch. En el siguiente ejemplo se crea un objeto myUserException del tipo UserException y lo usa en la sentencia throw.

// Crear un tipo de objeto UserException

function UserException (aviso){

this.aviso=aviso;

this.nombre="UserException";

}

// Make the exception convert to a pretty string when used as a string

// (e.g. by the error console)

UserException.prototype.toString = function () {

return this.nombre + ': "' + this.aviso + '"';

}

// Create an instance of the object type and throw it

throw new UserException("Value too high");

**try…catch**

La sentencia try...catch marca un bloque de instrucciones a intentar que pueden causar alguna excepción, y declarar una o más respuestas en caso de que una excepción sea arrojada. Si una excepción es arrojada, la sentencia try...catch se encarga de atraparla.

La sentencia try...catch consiste en un bloque try, el cuál contiene una o más instrucciones, y ninguno o varios bloques catch, conteniendo sentencias que especifican qué hacer si una excepción es arrojada en un bloque try. Se desea que las instrucciones dentro del bloque try se ejecuten con éxito, en caso contrario caerán en el bloque catch para ser controladas.

* Si alguna instrucción dentro del bloque try (o en una función llamada dentro del bloque try) arroja una excepción, el control pasa inmediatamente al bloque catch.
* Si ninguna excepción es arrojada en el bloque try, el bloque catch es ignorado.
* Por último se ejecuta el bloque finally  después de que los bloques try y catch hayan sido ejecutados pero antes de las instrucciones que se encuentren después de la sentencia try...catch.

El siguiente ejemplo usa la sentencia try...catch. El ejemplo llama a una función que retorna el nombre de un mes desde un arreglo basado en un valor pasado como argumento a la función. Si el valor no corresponde con el número de un mes (entre 1 y 12), una excepción es arrojada con el valor "InvalidMonthNo" y las instrucciones en el bloque catch le asignarán a la variable monthName el valor de unknown.

function getMonthName (mo) {

mo = mo-1; // Ajusta el indice del arreglo para el arreglo de meses (1=Jan, 12=Dec)

var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul",

"Aug","Sep","Oct","Nov","Dec"];

if (months[mo] != null) {

return months[mo];

} else {

throw "InvalidMonthNo"; //Arroja la palabra "InvalidMonthNo" al ocurrir una excepción

}

}

try { // instrucciones a probar

monthName = getMonthName(myMonth); // La función puede arrojar una excepción

}

catch (e) {

monthName = "unknown";

logMyErrors(e); // Pasa el objeto de la excepción a un manejador de errores

}

**El bloque catch**

Un bloque catch es usado para manejar todas las excepciones que pueden ser generadas en el bloque try.

catch (catchID) {

instrucciones

}

El bloque catch especifica un identificador (catchID en la sintaxis anterior) que mantiene el valor especificado por la sentencia thrown; puedes usar este identificador para obtener información acerca de la excepción que fue arrojada. JavaScript crea este identificador cuando ha entrado en el bloque catch; el identificador dura mientras dure el bloque catch; después de que el bloque catch termine su ejecución, el identificador ya no está disponible.

Por ejemplo, el siguiente código arroja una excepción. Cuando la excepción ocurre, el control es transferido al bloque catch.

try {

throw "myException" // genera una excepción

}

catch (e) {

// instrucciones para manejar cualquier excepción generada

logMyErrors(e) // Pasa el objeto de excepción a un manejador de errores

}

**El bloque finally**

El bloque finally contiene instrucciones para ejecutar después de la ejecución del bloque try y el bloque catch pero antes de las instrucciones ubicadas después de la sentencia try...catch. El bloque finally se ejecuta cuando se haya arrojado o no una excepción. Si una excepción es arrojada, las instrucciones en el bloque finally se ejecutan incluso si no existe un bloque catch que maneje la excepción.

Se puede usar el bloque finally para hacer que tu script falle con gracia cuando una excepción ocurre; por ejemplo, puedes tener la necesidad de liberar un recurso que tu script tiene ocupado. El siguiente ejemplo abre un archivo y luego ejecuta instrucciones que usan el archivo (**JavaScript del lado del servidor permite acceder a archivos**). Si una excepción es arrojada mientras el archivo está abierto, el bloque finally cierra el archivo antes de que el script falle.

openMyFile();

try {

writeMyFile(theData); // Esto puede arrojar un error

} catch(e) {

handleError(e); // Si ocurre un error es manejado

} finally {

closeMyFile(); // Siempre cierra el recurso

}

Si el bloque finally retorna un valor, este valor se convierte en el valor de retorno de toda la sentencia try-catch-finally, independientemente de cualquier sentencia return en el bloque try y el bloque catch:

function f() {

try {

console.log(0);

throw "bogus";

} catch(e) {

console.log(1);

return true; // Esta sentencia de retorno es suspendida

// hasta que el bloque finally esté completo

console.log(2); // no alcanzable

} finally {

console.log(3);

return false; // sobreescribe la sentencia de retorno anterior

console.log(4); // no alcanzable

}

// "return false" es ejecutada ahora

console.log(5); // no alcanzable

}

f(); // console 0, 1, 3; retorna false

Sobreescribiendo los valores retornados por el bloque finally también aplica a excepciones arrojadas o relanzadas dentro de un bloque catch:

function f() {

try {

throw "bogus";

} catch(e) {

console.log('caught inner "bogus"');

throw e; // Esta sentencia throw es suspendida hasta que

// el bloque finally se termine de ejecutar

} finally {

return false; // Sobreescribe la sentencia throw anterior

}

// "return false" es ejecutado ahora

}

try {

f();

} catch(e) {

// Esta nunca es encontrada porque la sentencia throw dentro

// del bloque catch es sobrescrita por la sentencia return

// en el bloque finally

console.log('caught outer "bogus"');

}

// SALIDA

// atrapado dentro de "bogus"

**Sentencias try...catch anidadas**

Es posible anidar una o más sentencias try...catch. Si una sentencia try…catch interna no posee un bloque catch, la sentencia try...catch exterior verifica si el bloque exterior genera una coincidencia.

**Utilizando objetos de Error**

Dependiendo del tipo de error, es posible usar el 'name' (nombre) y el 'message' (mensaje) propiedades para obtener un mensaje más refinado. La propiedad 'name' provee la clase general del Error (por ejemplo, 'DOMException' or 'Error'), mientras que la propiedad 'message' por lo general provee un breve mensaje que puede ser obtenido convirtiendo el error de object a string.

Si estás arrojando tus propias excepciones, en orden para tomar ventaja de estas propiedades (Como si tu bloque catch no discrimina entre tus propias excepciones y las excepciones del sistema), puedes usar el constructor de Error. Por ejemplo:

function doSomethingErrorProne () {

if (ourCodeMakesAMistake()) {

throw (new Error('The message'));

} else {

doSomethingToGetAJavascriptError();

}

}

....

try {

doSomethingErrorProne();

}

catch (e) {

console.log(e.name); // logs 'Error'

console.log(e.message); // logs 'The message' o un error de JavaScript)

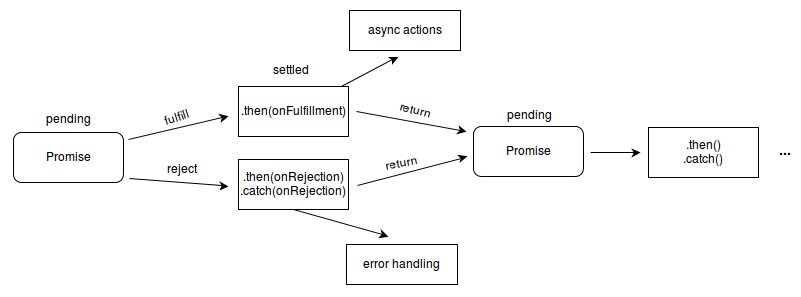
}

**Promises**

A partir de ECMAScript 6, Javascript gana soporte para los objetos Promise que le permiten controlar el flujo de operaciones diferidas y asíncronas.

Una Promise puede estar ubicada en estos estados:

* *pending*: Estado inicial, ni terminada exitosamente o rechazada.
* *fulfilled*: operación exitosa.
* *rejected*: operación fallida o rechazada.
* *settled*: la Promise ha sido exitosa o rechazada, pero no está pendiente.



**Cargando una imagen con XHR**

Un simple ejemplo del uso de Promise y [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) es cargar una imagen que está disponible en el repositorio [promise-test](https://github.com/mdn/promises-test/blob/gh-pages/index.html) de MDN GitHub. Puedes [verlo también en acción](http://mdn.github.io/promises-test/). Cada paso es comentado y permite que sigas la arquitectura de la Promise y XHR de cerca. Aquí está una versión sin comentar, mostrando el flujo de una Promise para que puedas tener una idea:

function imgLoad(url) {

return new Promise(function(resolve, reject) {

var request = new XMLHttpRequest();

request.open('GET', url);

request.responseType = 'blob';

request.onload = function() {

if (request.status === 200) {

resolve(request.response);

} else {

reject(Error('Image didn\'t load successfully; error code:'

+ request.statusText));

}

};

request.onerror = function() {

reject(Error('There was a network error.'));

};

request.send();

});

}

Para información más detallada, visitar [Promise](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise" \o "El objeto Promise (Promesa) es usado para computaciones asíncronas. Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca.) como página de referencia.