JavaScript

ECMA-262
ISO/IEC-16262

Referencias

- Standard ECMA-262
- Latest ECMA spec
- El Tutorial de JavaScript Moderno
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features
- Airbnb JavaScript Style Guide

INTRODUCCIÓN

Introducción

- El JavaScript es un lenguaje de secuencias de comandos multiplataforma e independiente de cualquier empresa, heredero de la sintaxis del C, que originalmente fue diseñada para ir inmerso dentro de las páginas web, ser interpretado por el navegador y hacer programable el HTML.
- Cuando JavaScript debutó en 1996, agregó interactividad a una web que, hasta entonces, estaba compuesta por documentos estáticos. La web se convirtió no solo en un lugar para leer cosas, sino para hacer cosas. La popularidad de JavaScript aumentó constantemente. Los desarrolladores que trabajaron con JavaScript escribieron herramientas para resolver los problemas que enfrentaban y las empaquetaron en paquetes reutilizables llamados bibliotecas, para que pudieran compartir sus soluciones con otros. Este ecosistema compartido de bibliotecas ayudó a dar forma al crecimiento de la web.
- Ahora, JavaScript es una parte esencial de la web, utilizado en el 95% de todos los sitios web, y la web es una parte esencial de la vida moderna. Los usuarios escriben artículos, administran sus presupuestos, transmiten música, ven películas y se comunican con otros a grandes distancias instantáneamente, con chat de texto, audio o video. La web nos permite hacer cosas que solían ser posibles solo en aplicaciones nativas instaladas localmente. Estos sitios web modernos, complejos e interactivos a menudo se denominan aplicaciones web .

Introducción

- Creado por Brendan Eich para el navegador Netscape Navigator 2.0, que iba a lanzarse en 1995. Inicialmente, se denomino LiveScript.
- Justo antes del lanzamiento, tras la alianza de Netscape con Sun Microsystems, se decidió cambiar el nombre por el de JavaScript. La razón del cambio de nombre fue exclusivamente por marketing, ya que Java era la palabra de moda en el mundo informático y de Internet de la época, no existe ninguna relación con el lenguaje Java de SUN salvo la común herencia de la sintaxis con el C.
- Al mismo tiempo, Microsoft lanzó JScript con su navegador Internet Explorer 3. JScript era una copia de JavaScript al que le cambiaron el nombre para evitar problemas legales.
- En 1997, Netscape decidió liberar el lenguaje y lo estandarizó a través del ECMA. El primer estándar se denominó ECMA-262, en el que se definió por primera vez el lenguaje ECMAScript, quedando el nombre de JavaScript como la implementación que realizó la empresa Netscape del estándar ECMAScript. La ISO adoptó el estándar ECMA-262 dando lugar al estándar ISO/IEC-16262.

Especificaciones oficiales

- ECMA ha publicado varios estándares relacionados con ECMAScript.
- En Junio de 1997 se publicó la primera edición del estándar ECMA-262.
- Un año después, en Junio de 1998 se realizaron pequeñas modificaciones para adaptarlo al estándar ISO/IEC-16262 y se creó la segunda edición.
- La tercera edición del estándar ECMA-262, publicada en Diciembre de 1999, es la versión que soportan todos los navegadores actuales.
- La cuarta versión de ECMA-262 no llego a publicarse.
- En junio de 2011, se publico la edición 5.1
- En junio de 2015 aparece la 6ª edición del estándar ECMAScript (ES6) con importantes cambios, cuyo nombre oficial es ECMAScript 2015, y solo la soportan las versiones mas modernas de los navegadores.
- A partir de la versión 6 cambia la política de versionado y se saca una revisión anual con pocos cambios: ECMAScript 2016 (ES7), ECMAScript 2017 (ES8), ECMAScript 2018 (ES9), ECMAScript 2019 (ES10), …
- Se puede consultar la especificación completa en:
  - http://www.ecma-international.org/publications/standards/Ecma-262.htm

Añadir JavaScript a una página

- Existen dos formas de insertar código JavaScript dentro de una página: escribiendo código en la misma (en ingles inline) o a través de un archivo externo utilizando la etiqueta script.
- El orden en el cual se incluye el código es importante:
  - Un código que depende de otro debe ser incluido después del que referencia.
  - El código es interpretado secuencialmente desde el inicio de la página.
  - Para mejorar el rendimiento de la página, el código JavaScript debe ser incluido al final del HTML.
  - Es conveniente, cuando se trabaja en un ambiente de producción con múltiples archivos JavaScript, que éstos sean combinados en un solo archivo minimizado.

Etiquetas

- Interno:

```
<script type="text/javascript">
   … Código JavaScript …
</script>
```

- Externo:

```
<script type="text/javascript" src="/js/codigo.js"></script>
```

- En línea:

```
<eti evento="javascript: … Código JavaScript …" …>
```

- Para los navegadores que no disponen de soporte completo de JavaScript o en los que ha sido bloqueado o inhabilitado por el usuario.

```
<noscript>
  … Aviso en HTML …
</noscript>
```

TIPOS, VARIABLES Y EXPRESIONES

Sintaxis

- Sensible a mayúsculas y minúsculas.
- Sentencias separadas por punto y coma (;), aunque opcional en algunas ocasiones es recomendable utilizarlo siempre.
- Formato libre en las sentencias, una sentencia puede utilizar varias líneas y una línea puede incluir varias sentencias.
- Los espacios en blanco se compactan a uno solo.
- Bloques marcados por llaves: { <sentencias> }
- Comentarios:
  - // hasta el final de la línea
  - /* … */ una o varias líneas

Separadores

( )	para contener listas de parámetros en los métodos o funciones, expresiones de control de flujo y establecer precedencias en la expresiones.
{ }	para definir bloques de código e inicializar objetos.
[ ]	para declarar y referencias tablas.
;	separar instrucciones.
,	separar identificadores en su declaración y concatenar sentencias dentro del for.
.	separador decimal y para hacer referencia a los atributos y métodos.

Identificadores

- Compuestos por letras, números, _, $.
- Sin límite en el número de caracteres.
- NO pueden comenzar con números.
- NO debe tener el mismo nombre que otro elemento del mismo ámbito.
- NO pueden ser palabras reservadas ni el nombre de un valor con nombre (true / false / null / undefined).
- Palabras reservadas:
  - break, case, catch, continue, debugger, default, delete, do, else, finally, for, function, if, in, instanceof, new, return, switch, this, throw, try, typeof, var, void, while, with
- Palabras reservadas para uso futuro:
  - *class, const*, enum, *export*, *extends, import, super*, implements, interface, *let*, package, private, protected, public, *static*, yield

Tipos de datos

- JavaScript divide los distintos tipos de datos en dos grupos:
  - Tipos del lenguaje:
    - Tipos primitivos: undefined, null, boolean, string, number
    - Tipos de referencia: Undefined, Null, Boolean, String, Number, Object.
  - Tipos de especificación (meta-valores): Reference, List, Completion, Property Descriptor, Property Identifier, Lexical Environment, Environment Record
- JavaScript define algunos objetos de forma nativa, por lo que pueden ser utilizados directamente por las aplicaciones sin tener que declararlos:
  - Global, Function, Array, Math, Date, Error, JSON y RegExp

Tipos de datos

- Tipo Undefined
  - Tipo con un solo valor llamado “undefined”.
  - Valor de las variables que no han sido aun declaradas o que no se les ha asignado un valor.
- Tipo Null
  - Tipo con un solo valor llamado “null”.
- Tipo Boolean
  - Sólo puede almacenar uno de los dos valores lógicos: “true” o “false”.
  - Los valores false, 0, null y undefined son considerados como false en las operaciones lógicas y el resto de los valores como true.

Tipo numérico

- Acepta Valores aritméticos enteros y reales. Si el número es real, se debe utilizar el punto (.) para separar la parte entera de la decimal.
- Formatos:
  - Decimal
  - Hexadecimal (0x????)
  - Coma flotante (?.??e??).
- JavaScript define valores especiales muy útiles cuando se trabaja con números.
  - Infinity, para representar números demasiado grandes (positivos y negativos) y con los que JavaScript no puede trabajar.
  - NaN (Not a Number), resultado de las operaciones matemáticas con variables no numéricas o la indeterminación en calculo numérico. isNaN() informa de que el valor no puede ser usado como numérico.
- JavaScript define constantes y operaciones matemáticas adicionales a través del objeto Math.

Tipo cadena

- Conjunto de caracteres encerrados entre comillas simples o dobles.
- Permite el anidamiento de una cadena entre comillas dobles dentro de una cadena entre comillas simples y viceversa.
- Las constantes de tipo cadena no pueden ocupar mas de una línea, para utilizar varias líneas requiere concatenación.
- El formato de las cadenas es UTF8 y pueden contener secuencias de escape:
  - \n, \r, \\, \", \', \t, \v, \e, \f
  - \[0-9]{1,3}
  - \x[0-9A-Fa-f]{2}
  - \u[0-9A-Fa-f]{4}

Template Strings (ES6)

- Interpolación: sustituye dentro de la cadena las variable por su valor:

```
let var1 = "JavaScript";
let var2 = "Templates";
console.log(`El ${var1} ya tiene ${var2}.`);  // El JavaScript ya tiene Templates.
```

- Se puede personalizar la interpolación con una función que recibe como un array los fragmentos cadena y un argumento por expresión:

```
frmt`El ${var1} ya tiene ${var2}.` // frmt(["El ", " ya tiene "], var1, var2);
```

- Las constantes cadenas pueden ser multilínea sin necesidad de concatenarlos con +.

```
let var1 = " El JavaScript
ya tiene
Templates ";
```

- Incorpora soporte extendido para el uso de Unicode en cadenas y expresiones regulares (caracteres en hebreo, árabe, cirílico, …).

Conversión entre tipos

- JavaScript es un lenguaje "no tipado", lo que significa que una misma variable puede guardar diferentes tipos de datos a lo largo de la ejecución de la aplicación.
- En JavaScript las conversiones entre tipos numéricos y cadenas se realiza de forma implícita.
- Los valores aritméticos se convierten automáticamente en cadenas en las operaciones de concatenación.
- En caso de ser necesario, para realizar las conversiones de forma explícita se utiliza:
  - El método toString() que permite convertir variables de cualquier tipo a variables de tipo cadena.
  - Las funciones parseInt() y parseFloat() convierten la cadena en un número entero o decimal respectivamente.
    - Si la cadena no contiene un valor aritmético o no empieza por un dígito devuelve el valor NaN.
    - La conversión numérica de una cadena se realiza carácter a carácter empezando por el de la primera posición hasta que encuentra un carácter que no es un dígito o termina la cadena.

Operadores

- Asignación
  Simple: <Destino> = <Expresión>
- Aritméticos
  - `+` : Suma, si el algún operando es una cadena se concatena.
  - `-` : Resta, cambio de signo cuando es unario.
  - `*` : Producto
  - `/` : División, el resultado es siempre real.
  - `%` : Resto de la división entera
- Acumulativos: `+=` , `-=` , `*=` , `/=` , `%=`
- Operadores relacionales
  - `>` : mayor
  - `>=` : mayor o igual
  - `<` : menor
  - `<=` : menor o igual
  - `==` : igual
  - `!=` : distinto
  - `===` : identidad (coincide el tipo)
  - `!==` : no identidad

Operadores

- Operadores lógicos (los operandos son booleanos)
  - `&&` : AND
  - `||` : OR
  - `!` : NOT
- Binarios
  - `>>` : Desplazamiento Derecha
  - `<<` : Desplazamiento Izquierda
  - `>>>` : Desplazamiento Derecha sin signo
  - `&` : AND binario
  - `|` : OR binario
  - `^` : XOR binario
  - `~` : complemento
- Acumulativos: `>>=`, `<<=`, `>>>=`, `&=`, `|=`, `^=`, `~=`
- Incremento/decremento:
  - `++<Variable>` : Se incrementa en 1 y se consulta.
  - `--< Variable >` : Se decrementa en 1 y se consulta.
  - `< Variable >++` : Se consulta y se incrementa en 1.
  - `< Variable >--` : Se consulta y se decrementa en 1.

Operadores

- Operador unario: **new**
  - Crea un nuevo objeto.
- Operador unario: **delete**
  - Elimina una propiedad de un objeto o quita un elemento de una matriz.
- Operador unario: **void**
  - Descarta el operando y devuelve no definido (undefined).
- Operador unario de tipo: **typeof**
  - Devuelve una cadena con el tipo de la variable, los valores devueltos son: "number", "string", "boolean", "object", "function" y "undefined".
- Operador binario de pertenencia a tipo: **instanceof**
  - Devuelve una true si es del tipo indicado.
- Operador binario de pertenencia: **in**
  - Comprueba la existencia de una propiedad en un objeto: property in object
- Operador condicional ternario:
  - <condición>**?**<expresión cuando verdadero>**:**<expresión cuando falso>

Precedencia de operadores

Operador	Descripción
. [] ()	Acceso a campos, indexación de matrices y llamadas a funciones
++ -- - ~ ! typeof new void delete	Operadores unarios, tipos de datos devueltos, creación de objetos, valores no definidos
* / %	Multiplicación, división, división módulo
+ - +	Adición, sustracción, concatenación de cadenas
<< >> >>>	Desplazamiento de bits
< <= > >=	Menor que, menor que o igual a, mayor que, mayor que o igual a
== != === !==	Igualdad, desigualdad, identidad, no identidad
&	AND de bits
^	XOR de bits
|	OR de bits
&&	AND lógico
||	OR lógico
?:	Condicional
= *OP*=	Asignación, asignación con operación
,	Evaluación múltiple

Variables

- No es necesario declararlas de forma explícita, pero sí recomendable, aunque con la directiva de prologo 'use strict' el navegador generara error si no se declara.
- Se declaran sin tipo, van asumiendo el tipo del valor contenido.
- Declaración:
  var <Identificador> *[= expresión], <Otro> [= expresión], …*;
- Ámbito:
  - Local:
    - Función en la que está declarada
    - Accesible desde la propia función y todos sus bloques anidados.
  - Global:
    - Declarada fuera de cualquier función.
    - Las variables no declaradas son siempre globales.
    - Accesibles desde cualquier punto.
  - Las variables locales prevalecen sobre las globales

Declaración de variables (ES6)

- let: ámbito de bloque, solo accesible en el bloque donde está declarada.

```
(function() {
   if(true) {
      let x = "variable local";
   }
   console.log(x); // error, "x" definida dentro del "if"
})();
```

- const: constante, se asina valor al declararla y ya no puede cambiar de valor.

```
const PI = 3.15;
PI = 3.14159; // error, es de sólo-lectura
```

- Las constantes numérica ahora se pueden expresar en binario y octal.

```
0b111110111 === 503
0o767 === 503
```

Destructuring (ES6)

- Asignar (repartir) los valores de un objeto o array en varias variables:

```
var tab = ["hola", "adiós"];
var [a, b] = tab;
console.log(a); // "hola"
console.log(b); // "adiós“
[ b, a ] = [ a, b ]

var obj = { nombre: "Pepito", apellido: "Grillo" };
var { nombre, apellido } = obj;
console.log(nombre); // "Pepito"
```


INSTRUCCIONES DE CONTROL

Bifurcación simple

```
if(<Condición>)
<Instrucción>; o {<Bloque de instrucciones>}
[
else
<Instrucción>; o {<Bloque de instrucciones>}
]
```

- Sin limite en los anidamientos.
- La parte else es opcional. Para evitar confusión es correcto crear bloques cuando existan anidamientos.

Bifurcación múltiple

```
switch(<Expresión>) {
case <Valor 1> :
            <Bloque de instrucciones>
            [break;]
case <Valor 2> :
            <Bloque de instrucciones>
            [break;]
    …
[default:
            <Bloque de instrucciones>]
}
```

- Se ejecutan las instrucciones en cascada hasta el final o hasta que encuentra un **break**.
- Sin límite en los anidamientos.
- La parte **default** es opcional, se ejecuta cuando no se cumplen las condiciones anteriores.

Bucles

```
while(<Condición>)
    ; o <Instrucción>; o {<Bloque de instrucciones>}

do
    <Instrucción>; o <Bloque de instrucciones>
while (<Condición>);
```

- while: Se ejecuta de 0 a n veces.
- do while: Se ejecuta de 1 a n veces.

Bucles

```
for([var] <Inicio>; <Final>; <Iteración>)
  ; o <Instrucción>; o {<Bloque de instrucciones>}
```

- Apartados (todos son opcionales):
  - <Inicio> : Se ejecutan antes de comenzar. Lista de expresiones separadas por comas. La palabra reservada **var** es opcional.
  - <Final> : Expresión condicional de finalización, se ejecuta mientras se cumpla la condición.
  - <Iteración> : Se ejecutan en cada iteración. Lista de expresiones separadas por comas.

```
for([var] <Variable> in <Objeto>)
  <Instrucción>; o {<Bloque de instrucciones>}
```

- A <Variable> se le asignan uno a uno todos los valores del índice o nombre de propiedad del <Objeto>.
- Para acceder a los valores contenido en el <Objeto>:

```
for(var cmp in obj)
    console.log(obj[cmp]);
```

(ES6)

```
for([var] <Variable> of <Objeto>)
  <Instrucción>; o {<Bloque de instrucciones>}
```

- A <Variable> se le asignan uno a uno todos los valores contenidos en <Objeto>.

Control de flujo

- **break**;
  - Termina la instrucción asociada y pasa a la siguiente.
- **continue;**
  - Detiene el ciclo actual de un bucle y salta a evaluar la condición.
- **return *[<Expresión>]*;**
  - Termina la ejecución de la función y, opcionalmente, devuelve el resultado de la misma;

Tratamiento de excepciones

```
try
    {<Bloque de instrucciones>}
catch (<Identificador>)
    {<Bloque de instrucciones>}
finally
    {<Bloque de instrucciones>}
```

- La sección catch solo se ejecuta en caso de error. El <Identificador> contiene un objeto de tipo Error con la información adicional sobre la excepción.
- La sección finally se ejecuta siempre, independientemente de si se produce o no la excepción.
- Las secciones catch y finally son opcionales pero al menos debe aparecer una.
- Se pueden anidar instrucciones try.
- El código de las secciones catch y finally no se encuentra vigilado por lo que es necesario envolverlo en una instrucción try si es susceptible de producir excepciones.
- Para lanzar excepciones se utiliza:
  `throw new Error(<Mensaje>);`

Otras instrucciones

- **with** (<objeto>)

  {<Bloque de instrucciones>}

  – Selector de objeto: No es necesario utilizar el punto para la selección de sus métodos y atributos.
- **debugger;**
  – Provoca un punto de interrupción cuando se ejecuta en un depurador, si un depurador no está presente o activo esta declaración no tiene ningún efecto observable.

FUNCIONES

Sintaxis

```
function Identificador ([<Lista de parámetros>]) {
    …
    [return [<Expresión>];]
}
```

- Una función es un objeto con conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente, permitiendo modularizar el código.
- Una función puede devolver un valor y participar en expresiones.
- No es obligatorio que las funciones tengan una instrucción de tipo return para devolver valores. Cuando una función no devuelve ningún valor o cuando en la instrucción return no se indica ningún valor, automáticamente se devuelve el valor undefined.
- Una función puede contener en su interior otras funciones anidadas.

Lista de parámetros

- Una función puede recibir uno o varios valores para realizar sus operaciones.
- La lista de parámetros es una lista de identificadores separados por comas.
- Al ser el JavaScript un lenguaje "no tipado", no es posible asegurar que los parámetros que se pasan a una función sean del tipo adecuado para las operaciones que realiza la función.
- El número de argumentos que se pasa a una función debería ser el mismo que el número de argumentos que ha indicado la función. No obstante, JavaScript no muestra ningún error si se pasan más o menos argumentos de los necesarios.
- Si se pasan menos parámetros que los definidos en la función, al resto de parámetros hasta completar el número correcto se les asigna el valor undefined.
- Si a una función se le pasan más argumentos que los parámetros que ha definido, a los argumentos sobrantes no se les asignan nombres.
- El orden de los argumentos es fundamental, ya que el primer dato que se indica en la llamada, será el primer valor que espera la función; el segundo valor indicado en la llamada, es el segundo valor que espera la función y así sucesivamente.
- Se puede utilizar un número ilimitado de argumentos.

```
function fn(p1, p2, p3) { … }
rslt  = fn('algo', var1, 3*pi);
```

Lista de parámetros variables

- Una función se puede definir sin parámetros pero utilizar los parámetros pasados.
- La propiedad de la función arguments es un objeto de tipo Arguments que se puede tratar como si fuera un Array.

```
function avg() {
  var rslt= 0;
  for(var i=0; i < arguments.length; i++) {
        rslt += arguments[i];
  }
  return arguments.length ? (rslt / arguments.length) : 0;
}

var variable1 = avg(1, 3, 5, 8);
var variable2 = avg(4, 6, 8, 1, 2, 3, 4, 5);
```

- La propiedad callee hace referencia a la función que se está ejecutando, accediendo a arguments.callee.length se puede obtener el número de parámetros con los que se ha definido la función.

Parámetros de funciones (ES6)

- Operador de propagación: Convierte un array o cadena en una lista de parámetros.

```
var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
```

- Valores por defecto: Se pueden definir valores por defecto a los parámetros en las funciones.

```
function(valor = "foo") {...};
```

- Resto de los parámetros: Convierte una lista de parámetros en un array.

```
function f (x, y, ...a) {
   return (x + y) * a.length
}
f(1, 2, "hello", true, 7) === 9
```

Tipo Function

- Las funciones son objetos de tipo Function y como tal se pueden almacenar en variables.

```
function suma(a, b) {
  return a + b;
}
var miFuncion = suma;
rslt = miFuncion(2, 2);
```

- La asignación se debe realizada solo con el identificador sin paréntesis para evitar que se evalúe.
- De igual forma se pueden pasar una función como valor de un argumento de otra función.

```
function calcula(fn, a, b) {
  return fn(a, b);
}

rslt = calcula(suma, 2, 2);
```


Funciones anónimas

- La asignación permite definir una función con una expresión en la que el nombre de la función es opcional, se conocen como funciones anónimas.

```
var miFuncion = function(a, b) { return a + b; }
```

- De igual forma, una función puede devolver otra función:

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

- Una función anónima autoejecutable consiste en crear una expresión de función e inmediatamente ejecutarla. Es muy útil para casos en que no se desea intervenir espacios de nombres globales, debido a que ninguna variable declarada dentro de la función es visible desde afuera.

```
(function(){
    var local= 'Valor interno';
})();
```

Función Arrow (ES6)

- Funciones anónimas.

```
data.forEach(elem => {
   console.log(elem);
   // …
});
var fn = (num1, num2) => num1 + num2;
pairs = evens.map(v => ({ even: v, odd: v + 1 }));
fn = () => {console.log("Error");};
```

- this: dentro de una función Arrow hace referencia al contenedor y no al contexto de la propia función.

```
bar : function() {
      document.addEventListener("click", (e) => this.foo());
   }
```

- equivale a (ES5):

```
bar : function() {
      document.addEventListener("click", function(e) {
         this.foo();
      }.bind(this));
   }
```


OBJETOS Y CLASES

Introducción

- El JavaScript utiliza objetos pero no es un lenguaje orientado a objetos, esta orientado a prototipos.
- No soporta clases, herencia, sobrecarga, …
- Técnicamente, un objeto de JavaScript es un array asociativo formado por las propiedades y los métodos del objeto.
- Un array asociativo es aquel en el que cada elemento no está asociado a su posición numérica dentro del array, sino que está asociado a otro valor específico, pares nombre/valor. En los arrays normales, los valores se asocian a índices que siempre son numéricos, mientras que en los arrays asociativos se asocian a claves que siempre son cadenas de texto.
- Las propiedades son variables globales del objeto y los métodos son variables globales del objeto pero de tipo Function.
- Los miembros son accesibles externamente mediante la notación punto (objeto.miembro) o mediante la notación array (objeto["miembro"]).
- Los miembros son accesibles internamente mediante la notación punto y la palabra clave this (this.miembro) .

Creación de Objetos

- Implementación directa

```
var elObjeto = new Object();
elObjeto.id = "99";
elObjeto.nombre = "Objeto de prueba";
elObjeto.muestraId = function() {
  alert("El ID del objeto es " + this.id);
}
elObjeto.ponNombre = function(nom) {
  this.nombre=nom.toUpperCase();
}
```

- Notación JSON

```
var elObjeto = {
  id : "99",
  nombre : "Objeto de prueba",
  muestraId : function() {
    alert("El ID del objeto es " + this.id);
  },
  ponNombre : function(nom) {
    this.nombre=nom.toUpperCase();
  }
}
```

Funciones constructoras

- Al contrario que en los lenguajes orientados a objetos, en JavaScript no existe el concepto de constructor. Por lo tanto, al definir un objeto no se incluyen uno o varios constructores. En realidad, JavaScript emula el funcionamiento de los constructores mediante el uso de funciones constructoras.

```
function MiClase(elId, elNombre) {
  this.id = elId;
  this.nombre = elNombre;
  this.muestraId = function() {
    alert("El ID del objeto es " + this.id);
  }
  this.ponNombre = function(nom) {
    this.nombre=nom.toUpperCase();
  }
}
var elObjeto = new MiClase("99", "Objeto de prueba");
```


Prototype

- Cada vez que se instancia un objeto con la función constructora, se definen tantas nuevas funciones como métodos incluya la función constructora.
- La penalización en el rendimiento y el consumo excesivo de recursos de esta técnica puede suponer un inconveniente en las aplicaciones profesionales realizadas con JavaScript.
- Todos los objetos de JavaScript incluyen una referencia interna a otro objeto llamado prototype o "prototipo". Cualquier propiedad o método que contenga el objeto prototipo, está presente de forma automática en el objeto original.
- Es como si cualquier objeto de JavaScript heredara de forma automática todas las propiedades y métodos de otro objeto llamado prototype. Cada tipo de objeto diferente hereda de un objeto prototype diferente.
- Dado que el prototype es el molde con el que se fabrica cada objeto de ese tipo. Si se modifica el molde o se le añaden nuevas características, todos los objetos fabricados con ese molde tendrán esas características.

Prototype

- En el prototype de un objeto sólo se deben añadir aquellos elementos comunes para todos los objetos. Normalmente se añaden los métodos y las constantes (propiedades cuyo valor no varía durante la ejecución de la aplicación). Las propiedades del objeto permanecen en la función constructora para que cada objeto diferente pueda tener un valor distinto en esas propiedades.

```
function MiClase(elId, elNombre) {
  this.id = elId;
  this.nombre = elNombre;
}
MiClase.prototype.muestraId = function() {
  alert("El ID del objeto es " + this.id);
}
MiClase.prototype.ponNombre = function(nom) {
  this.nombre=nom.toUpperCase();
}
```


Prototype

- La propiedad prototype también permite añadir y/o modificar las propiedades y métodos de los objetos predefinidos por JavaScript.
- Por lo tanto, es posible redefinir el comportamiento habitual de algunos métodos de los objetos nativos de JavaScript.
- Además, se pueden añadir propiedades o métodos completamente nuevos.
- Existen librerías de JavaScript formadas por un conjunto de utilidades que facilitan la programación de las aplicaciones y una de sus características habituales es el uso de la propiedad prototype para mejorar las funcionalidades básicas de JavaScript.

Palabra clave this

- En JavaScript, así como en la mayoría de los lenguajes de programación orientados a objetos, this es una palabra clave especial que hace referencia al objeto en donde el método está siendo invocado.
- El valor de this se determina utilizando una serie de simples reglas:
  1. Si la función es invocada utilizando Function.call o Function.apply, this tendrá el valor del primer argumento pasado al método. Si el argumento es nulo (null) o indefinido (undefined), this hará referencia el objeto global (el objeto window);
  2. Si la función a invocar es creada utilizando Function.bind, this será el primer argumento que es pasado a la función en el momento en que se la crea;
  3. Si la función es invocada como un método de un objeto, this referenciará a dicho objeto;
  4. De lo contrario, si la función es invocada como una función independiente, no unida a algún objeto, this referenciará al objeto global.

Métodos apply() y call()

- Los métodos del objeto Funciton apply() y call() permiten ejecutar una función como si fuera un método de otro objeto. La única diferencia entre los dos métodos es la forma en la que se pasan los argumentos a la función.
- El primer parámetro del método call() es el objeto sobre el que se va a ejecutar la función. El resto de parámetros del método call() son los parámetros que se pasan a la función.

```
function miFuncion(x) {
  return this.numero + x;
}
var elObjeto = new Object();
elObjeto.numero = 5;

var resultado = miFuncion.call(elObjeto, 4);
alert(resultado);
```

- El método apply() es idéntico al método call(), salvo que en este caso los parámetros se pasan como un array:

```
var resultado = miFuncion.apply(elObjeto, [4]);
alert(resultado);
```

Propiedades de objetos (ES6)

ES 2015

```
obj = { x, y }
obj = {
   foo (a, b) {
      …
   },
   bar (x, y) {
      …
   },
   *iter (x, y) {
      …
   }
}
```

Anteriormente:

```
obj = { x: x, y: y };
obj = {
   foo: function (a, b) {
      …
   },
   bar: function (x, y) {
      …
   },
   //  iter: sin equivalencia
   …
};
```


Clases (ES6)

- Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objetos que realizábamos en el estándar anterior, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como por ejemplo, herencia.

```
class LibroTecnico extends Libro {
   constructor(tematica, paginas) {
      super(tematica, paginas);
      this.capitulos = [];
      this.precio = "";
      // ...
   }
   metodo() {
      // ...
   }
}
```

Static Members (ES6)

```
class Rectangle extends Shape {
   …
   static defaultRectangle () {
      return new Rectangle("default", 0, 0, 100, 100)
   }
}
class Circle extends Shape {
   …
   static defaultCircle () {
      return new Circle("default", 0, 0, 100)
   }
}
var defRectangle = Rectangle.defaultRectangle()
var defCircle    = Circle.defaultCircle()
```


Getter/Setter (ES6)

```
class Rectangle {
   constructor (width, height) {
      this._width  = width;
      this._height = height;
   }
   set width  (width)  { this._width = width;            }
   get width  ()       { return this._width ;            }
   set height (height) { this._height = height;          }
   get height ()       { return this._height;            }
   get area   ()       { return this._width * this._height; }
}
var r = new Rectangle(50, 20)
r.area === 1000
```

mixin (ES6)

- Soporte para la herencia de estilo mixin mediante la ampliación de las expresiones que producen objetos de función.

```
var aggregation = (baseClass, ...mixins) => {
  let base = class _Combined extends baseClass {
    constructor (...args) {
      super(...args)
      mixins.forEach((mixin) => {
        mixin.prototype.initializer.call(this)
      })
    }
  }
  let copyProps = (target, source) => {
    Object.getOwnPropertyNames(source)
      .concat(Object.getOwnPropertySymbols(source))
      .forEach((prop) => {
      if (prop.match(/^(?:constructor|prototype|arguments|caller|name|bind|call|apply|toString|length)$/))
        return
      Object.defineProperty(target, prop, Object.getOwnPropertyDescriptor(source, prop))
    })
  }
  mixins.forEach((mixin) => {
    copyProps(base.prototype, mixin.prototype)
    copyProps(base, mixin)
  })
  return base
}
```


Módulos (ES6)

- Estructura el código en módulos similares a los espacios de nombres
- Los módulos siempre llevan 'use strict' de forma predeterminada.
- Cada fichero se comporta como un módulo y solo se carga la primera vez.
- Solo las partes marcadas como export pueden ser importadas en otros ficheros.
- Se puede llamar a las funciones desde los propios Scripts, sin tener que importarlos en el HTML (si usamos JavaScript en el navegador).

```
//File: lib/person.js
  export function hello(nombre) {
    return nombre;
  }
```

Y para importar en otro fichero:

```
//File: app.js
import { hello } from "lib/person.js";
var app = {   foo: function() { hello("Carlos");  }
}
export { app };
```

```
<script type="module">
import { app } from "./app.js";
```

Módulos: Exportaciones (ES6)

- Se puede etiquetar cualquier variable, contante, función o clase como exportada colocando precediéndola con la clausula 'export'.
  export oneFunc() { … }
- La etiqueta 'export default' permite la exportación predeterminada de una función o clase (exportación única o principal).
  export default MyClass { … }
- La instrucción 'export' permite dar la lista de variables, contantes, funciones o clases exportadas (separadas por comas).
  export { MY_CONST, MyClass, oneFunc, count as contador, …}
- La clausula as en la lista permite exportar con un nombre exterior diferente.
- La reexportación permite importar e inmediatamente exportarlas, para crear los ficheros index.js que aíslen de la complejidad interna de un directorio y sus sub directorios:
  export ... from ...

Módulos: Importaciones (ES6)

- Con * se importan todo (exportaciones) pero hay que asignarle un espacio de nombres con as, el acceso se realiza a través de dicho nombre:
  import * as ns from …
- Se puede especificar la lista de lo que se desea importar (sin espacio de nombres):
  import {MyClass, oneFunc as contador, … } from …
- La clausula as en la lista permite establecer nombres locales alternativos para evitar conflictos.
- La importación de la exportación predeterminada requiere un nombre local:
  import MyNameForDefault from …
- La clausula from indica la ubicación del fichero del módulo: es una ruta relativa al fichero sin extensión. Las rutas sin fichero utilizaran el fichero index.js de reexportación. Las rutas absolutas pueden estar asociadas a las bibliotecas o librerías en las herramientas de ensamblaje.
  import * as ns from './my-module'

Iteradores y Generadores (ES6)

- El patrón Iterador permite trabajar con colecciones por medio de abstracciones de alto nivel
- Un Iterador es un objeto que sabe como acceder a los elementos de una secuencia, uno cada vez, mientras que mantiene la referencia a su posición actual en la secuencia.
- En ES2015 las colecciones (arrays, maps, sets) son objetos iteradores.
- Los generadores permiten la implementación del patrón Iterador.
- Los Generadores son funciones que pueden ser detenidas y reanudadas en otro momento.
- Estas pausas en realidad ceden la ejecución al resto del programa, es decir no bloquean la ejecución.
- Los Generadores devuelven (generan) un objeto "Iterator" (iterador)

Generadores (ES6)

- Para crear una función generadora

```
function* myGenerator() {
  // ...
  yield value;
  // ...
}
```

- La instrucción yield devuelve el valor y queda a la espera de continuar cuando se solicite el siguiente valor.
- El método next() ejecuta el generador hasta el siguiente yield dentro del mismo y devuelve un objeto con el valor.

```
var iter = myGenerator();
// ...
rslt = iter.next();
// ...
```

- La nueva sintaxis del for permite recorrer el iterador completo:

```
for (let value of iter)
```

http://ecma-international.org/ecma-262/5.1/#sec-15

STANDARD BUILT-IN ECMASCRIPT OBJECTS

Objeto Global

- El objeto Global es parte del entorno léxico del programa en ejecución.
- Se encarga de encapsular los elementos no sintácticos del lenguaje:
  - contantes globales (propiedades valor)
    - NaN, Infinity, undefined
  - funciones globales (propiedades función)
    - eval(x), parseInt(string , radix), parseFloat(string), isNaN(number), isFinite(number)
    - decodeURI(uri), decodeURIComponent(uri), encodeURI(uri), encodeURIComponent(uri)
  - objetos globales (propiedades referencia)
    - Math, JSON
  - tipos predefinidos (propiedades funciones constructoras).
    - Object, Function, Array, String, Boolean, Number, Date, RegExp
    - Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError

Object

- **Object.create** (Función): Crea un objeto que tiene un prototipo especificado y contiene opcionalmente propiedades especificadas.
- **Object.defineProperties** (Función): Agrega una o varias propiedades a un objeto, o modifica atributos de propiedades existentes.
- **Object.defineProperty** (Función): Agrega una propiedad a un objeto o modifica atributos de una propiedad existente.
- **Object.seal** (Función): Impide la modificación de atributos de propiedades existentes e impide agregar nuevas propiedades.
- **Object.freeze** (Función): Impide la modificación de atributos y valores de propiedad existentes e impide agregar nuevas propiedades.
- **Object.preventExtensions** (Función): Impide la adición de nuevas propiedades a un objeto.
- **Object.isExtensible** (Función): Devuelve un valor que indica si se pueden agregar nuevas propiedades a un objeto.
- **Object.isFrozen** (Función): Devuelve true si no se pueden modificar atributos y valores de propiedad existentes en un objeto y no se pueden agregar nuevas propiedades al objeto.

Object

- **Object.isSealed** (Función): Devuelve true si no se pueden modificar atributos de propiedad existentes en un objeto y no se pueden agregar nuevas propiedades al objeto.
- **Object.keys** (Función): Devuelve los nombres de las propiedades y los métodos enumerables de un objeto.
- **Object.getPrototypeOf** (Función): Devuelve el prototipo de un objeto.
- **Object.getOwnPropertyDescriptor** (Función): Devuelve la definición de una propiedad de datos o de una propiedad de descriptor de acceso.
- **Object.getOwnPropertyNames** (Función): Devuelve los nombres de las propiedades y métodos de un objeto.
- **prototype** (Propiedad): Devuelve una referencia al prototipo correspondiente a una clase de objetos.
- **toLocaleString** (Método): Devuelve un objeto convertido en una cadena basándose en la configuración regional actual.
- **toString** (Método): Devuelve una representación en forma de cadena de un objeto.
- **valueOf** (Método): Devuelve el valor primitivo del objeto especificado.

Function

- **arguments** (Propiedad): Obtiene los argumentos del objeto Function que se está ejecutando actualmente.
- **caller** (Propiedad): Obtiene la función invocada por la función actual.
- **length** (Propiedad): Obtiene el número de argumentos definidos para una función.
- **apply** (Método): Llama a la función, sustituyendo el objeto especificado por el valor de this de la función, y la matriz especificada por los argumentos de la función.
- **bind** (Método): Para una función determinada, crea una función enlazada con el mismo cuerpo que la función original. En la función enlazada, el objeto this se resuelve en el objeto pasado. La función enlazada tiene los parámetros iniciales especificados.
- **call** (Método): Llama a un método de un objeto y sustituye el objeto actual por otro objeto.

Error

- **name** (Propiedad): Devuelve el nombre de un error.
- **message** (Propiedad): Devuelve una cadena con un mensaje de error.
  - *RangeError*: Este error se produce cuando se proporciona a una función un argumento que ha superado su intervalo permitido.
  - *ReferenceError*: Este error tiene lugar cuando se detecta una referencia no válida.
  - *SyntaxError*: Este error se produce cuando se analiza el texto de origen y su sintaxis no es correcta.
  - *TypeError*: Este error se produce cuando el tipo real de un operando no coincide con el tipo esperado.
  - *URIError*: Este error tiene lugar cuando se detecta un identificador uniforme de recursos (URI) no válido.

Array

- **length** (Propiedad): Devuelve un valor entero que supera en uno al elemento mayor definido en una matriz.
- **Array.isArray** (Función): : Devuelve un valor de tipo booleano que indica si un objeto es una matriz.
- **concat** (Método): Devuelve una matriz nueva que se compone de una combinación de dos matrices.
- **join** (Método): Devuelve un objeto String formado por todos los elementos de una matriz concatenados.
- **pop** (Método): Quita el último elemento de una matriz y lo devuelve.
- **push** (Método): Anexa nuevos elementos a una matriz y devuelve la nueva longitud de la matriz.
- **reverse** (Método): Devuelve un objeto Array con los elementos invertidos.
- **shift** (Método): Quita el primer elemento de una matriz y lo devuelve.
- **slice** (Método): Devuelve una sección de una matriz.
- **sort** (Método): Devuelve un objeto Array con los elementos ordenados.

Array

- **splice** (Método): Quita elementos de una matriz, inserta nuevos elementos en su lugar si procede y devuelve los elementos eliminados.
- **unshift** (Método): Inserta nuevos elementos al principio de una matriz.
- **indexOf** (Método): Devuelve el índice de la primera aparición de un valor de una matriz.
- **lastIndexOf** (Método): Devuelve el índice de la última aparición de un valor especificado de una matriz.
- **every** (Método): Comprueba si una función de devolución de llamada definida devuelve true para todos los elementos de una matriz.
- **some** (Método): Comprueba si una función de devolución de llamada definida devuelve true para cualquier elemento de una matriz.
- **forEach** (Método): Llama a una función de devolución de llamada definida para cada elemento de una matriz.
- **map** (Método): Llama a una función de devolución de llamada definida para cada elemento de una matriz y devuelve una matriz que contiene los resultados.

Array

- **filter** (Método): Llama a una función de devolución de llamada definida para cada elemento de una matriz y devuelve una matriz de aquellos valores para los que esa función devuelve true.
- **reduce** (Método): Acumula un solo resultado mediante la llamada a una función de devolución de llamada definida para todos los elementos de una matriz. El valor devuelto de la función de devolución de llamada es el resultado acumulado, y se proporciona como un argumento en la siguiente llamada a dicha función.
- **reduceRight** (Método): Acumula un solo resultado mediante la llamada a una función de devolución de llamada definida para todos los elementos de una matriz, en orden descendente. El valor devuelto de la función de devolución de llamada es el resultado acumulado, y se proporciona como un argumento en la siguiente llamada a dicha función.
- **toLocaleString** (Método): Devuelve una cadena con la configuración regional actual.
- **toString** (Método): Devuelve una representación de cadena de una matriz.
- **valueOf** (Método): Obtiene una referencia a la matriz.

Date

- **Date.now** (Función): Devuelve el número de milisegundos que hay entre el 1 de enero de 1970 y la fecha y hora actuales.
- **Date.parse** (Función): Analiza una cadena que contiene una fecha y devuelve el número de milisegundos transcurridos entre esa fecha y la medianoche del 1 de enero de 1970.
- **Date.UTC** (Función): Devuelve el número de milisegundos transcurrido entre la medianoche del 1 de enero de 1970 en el horario universal coordinado (UTC) (o GMT) y la fecha proporcionada.
- **toString** (Método): Devuelve una representación en forma de cadena de un objeto.
- **toDateString** (Método): Devuelve una fecha como un valor de cadena.
- **toTimeString** (Método): Devuelve una hora como un valor de cadena.
- **toLocaleString** (Método): Devuelve un objeto convertido en cadena usando la configuración regional actual.
- **toLocaleDateString** (Método): Devuelve una fecha como un valor de cadena apropiado para la configuración regional actual del entorno host.
- **toLocaleTimeString** (Método): Devuelve una hora como un valor de cadena apropiado para la configuración regional actual del entorno host.
- **toISOString** (método): Devuelve una fecha como un valor alfanumérico en formato ISO.
- **toJSON** (método): Se utiliza para transformar datos de un tipo de objeto antes de la serialización JSON.
- **valueOf** (Método): Devuelve el valor primitivo del objeto especificado.

Date

- **getTime** (Método): Devuelve el valor de tiempo en un objeto Date en milisegundos desde la medianoche del 1 de enero de 1970.
- **getFullYear**, **getUTCFullYear** (Método): Devuelve el valor de año usando la hora local o UTC.
- **getMonth**, **getUTCMonth** (Método): Devuelve el valor de mes usando la hora local o UTC.
- **getDate**, **getUTCDate** (Método): Devuelve el valor de día del mes usando la hora local o UTC.
- **getDay**, **getUTCDay** (Método): Devuelve el valor de día de la semana usando la hora local o UTC.
- **getHours**, **getUTCHours** (Método): Devuelve el valor de horas usando la hora local o UTC.
- **getMinutes**, **getUTCMinutes** (Método): Devuelve el valor de minutos usando la hora local o UTC.
- **getSeconds**, **getUTCSeconds** (Método): Devuelve el valor de segundos usando la hora local o UTC.
- **getMilliseconds** , **getUTCMilliseconds** (Método): Devuelve el valor de milisegundos usando la hora local o UTC.
- **getTimezoneOffset** (Método): Devuelve la diferencia en minutos entre la hora del equipo host y la hora universal coordinada (UTC).

Date

- **setTime** (Método): Establece el valor de fecha y hora en el objeto Date.
- **setMilliseconds** (Método): Establece el valor de milisegundos usando la hora local.
- **setUTCMilliseconds** (Método): Establece el valor de milisegundos usando la hora UTC.
- **setSeconds** (Método): Establece el valor de segundos usando la hora local.
- **setUTCSeconds** (Método): Establece el valor de segundos usando la hora UTC.
- **setMinutes** (Método): Establece el valor de minutos usando la hora local.
- **setUTCMinutes** (Método): Establece el valor de minutos usando la hora UTC.
- **setHours** (Método): Establece el valor de horas usando la hora local.
- **setUTCHours** (Método): Establece el valor de horas usando la hora UTC.
- **setDate** (Método): Establece el día del mes numérico usando la hora local.
- **setUTCDate** (Método): Establece el día numérico del mes usando la hora UTC.
- **setMonth** (Método): Establece el valor de mes usando la hora local.
- **setUTCMonth** (Método): Establece el valor de mes usando la hora UTC.
- **setFullYear** (Método): Establece el valor de año usando la hora local.
- **setUTCFullYear** (Método): Establece el valor de año usando la hora UTC.

Number

- **Number.MAX_VALUE**: El número más grande que se puede representar en JavaScript. Igual a aproximadamente 1,79E+308.
- **Number.MIN_VALUE**: El número más cercano a cero que se puede representar en JavaScript. Igual a aproximadamente 5,00E-324.
- **Number.NaN**: Un valor que no es un número.
- **Number.NEGATIVE_INFINITY**: Un valor inferior al número negativo más grande que se puede representar en JavaScript.
- **Number.POSITIVE_INFINITY**: Un valor superior al número más grande que se puede representar en JavaScript.

- **toExponential** (Método): Devuelve una cadena que contiene un número representado en notación exponencial.
- **toFixed** (Método): Devuelve una cadena que representa un número en notación de punto fijo.
- **toLocaleString** (Método): Devuelve un objeto convertido en una cadena basándose en la configuración regional actual.
- **toPrecision** (Método): Devuelve una cadena que contiene un número representado en notación exponencial o de punto fijo y que tiene un número especificado de dígitos.
- **toString** (Método): Devuelve una representación en forma de cadena de un objeto.
- **valueOf** (Método): Devuelve el valor primitivo del objeto especificado.

String

- **length** (Propiedad): Devuelve la longitud de un objeto String.
- **String.fromCharCode** (Función): Devuelve una cadena a partir de varios valores de caracteres Unicode.
- **charAt** (Método): Devuelve el carácter que se encuentra en el índice especificado.
- **charCodeAt** (Método): Devuelve la codificación Unicode del carácter que se especifique.
- **concat** (Método): Devuelve una cadena que contiene la concatenación de las dos cadenas proporcionadas.
- **indexOf** (Método): Devuelve la posición del carácter donde tiene lugar la primera repetición de una subcadena dentro de una cadena.
- **lastIndexOf** (Método): Devuelve la última repetición de una subcadena dentro de una cadena.
- **localeCompare** (Método): Devuelve un valor que indica si dos cadenas son equivalentes en la configuración regional actual.
- **match** (Método): Busca una cadena mediante un objeto Regular Expression proporcionado y devuelve los resultados como una matriz.
- **replace** (Método): Usa una expresión regular para reemplazar texto en una cadena y devuelve el resultado.
- **search** (Método): Devuelve la posición de la primera coincidencia de subcadena en una búsqueda de expresión regular.
- **slice** (Método): Devuelve una sección de una cadena.

String

- **split** (Método): Devuelve la matriz de cadenas resultante de la separación de una cadena en subcadenas.
- **substring** (Método): Devuelve la subcadena en la ubicación especificada dentro de un objeto String.
- **toLocaleLowerCase** (Método): Devuelve una cadena en la que todos los caracteres alfabéticos se convierten a minúsculas, según la configuración regional actual del entorno de host.
- **toLocaleString** (Método): Devuelve un objeto convertido en cadena usando la configuración regional actual.
- **toLocaleUpperCase** (Método): Devuelve una cadena en la que todos los caracteres alfabéticos se convierten a mayúsculas, según la configuración regional actual del entorno de host.
- **toLowerCase** (Método): Devuelve una cadena en la que todos los caracteres alfabéticos se convierten a minúsculas.
- **toString** (Método): Devuelve la cadena.
- **toUpperCase** (Método): Devuelve una cadena en la que todos los caracteres alfabéticos se convierten a mayúsculas.
- **trim** (Método): Devuelve una cadena donde se han quitado los caracteres de espacio en blanco iniciales y finales y los caracteres de terminador de línea.
- **valueOf** (Método): Devuelve la cadena.

RegEx

- Crea un objeto 'expresión regular' para encontrar texto de acuerdo a un patrón.
- Las cadenas delimitadas por / generan un objeto RegEx con el patrón contenido en la cadena:

```
var re = new RegExp("^\\d{1,8}[A-Z]$");
var re = /^\d{1,8}[A-Z]$/;
```

- Permiten los modificadores:
  - g: búsqueda global (no para tras la primera coincidencia)
  - i: ignorar mayúsculas o minúsculas
  - m: tratar caracteres de inicio y fin (^ y $) como múltiples líneas de texto
- Las expresiones regulares se utilizan con los métodos de RegExp (test y exec) y con los métodos de String (match, replace, search y split).

RegEx: Escapes de carácter

Carácter de escape	Descripción	Modelo	Coincidencias
\a	Coincide con un carácter de campana, \u0007.	\a	"\u0007" en "Error!" + '\u0007'
\b	En una clase de caracteres, coincide con un retroceso, \u0008.	[\b]{3,}	"\b\b\b\b" en "\b\b\b\b"
\t	Coincide con una tabulación, \u0009.	(\w+)\t	"artículo1\t", "artículo2\t" en "artículo1\tartículo2\t"
\r	Coincide con un retorno de carro, \u000D.(\r no es equivalente al carácter de nueva línea, \n.)	\r\n(\w+)	"\r\nEstas" en "\r\nEstas son\ndos líneas."
\v	Coincide con una tabulación vertical, \u000B.	[\v]{2,}	"\v\v\v" en "\v\v\v"
\f	Coincide con un avance de página, \u000C.	[\f]{2,}	"\f\f\f" en "\f\f\f"
\n	Coincide con una nueva línea, \u000A.	\r\n(\w+)	"\r\nEstas" en "\r\nEstas son\ndos líneas."
\e	Coincide con un escape, \u001B.	\e	"\x001B" en "\x001B"
\nnn	Usa la representación octal para especificar un carácter (nnn consta de tres dígitos como máximo).	\w\040\w	"a b", "c d" en "a bc d"
\xnn	Usa la representación hexadecimal para especificar un carácter (nn consta de exactamente dos dígitos).	\w\x20\w	"a b", "c d" en "a bc d"
\cX	Se corresponde con el carácter de control ASCII especificado por X, donde X es la letra del carácter de control.	\cC	"\x0003" en "\x0003" (Ctrl-C)
\unnnn	Se corresponde con un carácter Unicode usando la representación hexadecimal (exactamente cuatro dígitos, representados aquí por nnnn).	\w\u0020\w	"a b", "c d" en "a bc d"
\	Cuando va seguido de un carácter sin significado, coincide con ese carácter	\d+[\+-x*]\d+\d+[\+-x*]\d+	"2+2" y "3*9" en "(2+2) * 3*9"

RegEx: Clases de carácter

Clase de carácter	Descripción	Modelo	Coincidencias
[grupo_caracteres]	Coincide con cualquier carácter único de grupo_caracteres. De forma predeterminada, la coincidencia distingue entre mayúsculas y minúsculas.	[aeiou]	"a" en "casa" "a", "e" en "ave"
[^grupo_caracteres]	Negación: coincide con cualquier carácter individual que no esté en grupo_caracteres. De forma predeterminada, los caracteres de grupo_caracteres distinguen entre mayúsculas y minúsculas.	[^aei]	"r", "n", "o" en "reino"
[primero-último]	Intervalo de caracteres: coincide con cualquier carácter individual en el intervalo de primero a último.	[A-Z]	"A", "B" en "AB123"
.	Carácter comodín: coincide con cualquier carácter excepto con \n.	a.e	"ave" en "llave" "ate" en "yate"
\p{name}	Coincide con cualquier carácter único que pertenezca a la categoría general Unicode o al bloque con nombre especificado por nombre.	\p{Lu} \p{IsCyrillic}	"C", "L" en "City Lights" "Д", "Ж" in "ДЖem"
\P{name}	Coincide con cualquier carácter único que no pertenezca a la categoría general Unicode o al bloque con nombre especificado por nombre.	\P{Lu} \P{IsCyrillic}	"i", "t", "y" en "City" "e", "m" in "ДЖem"
\w	Coincide con cualquier carácter de una palabra.	\w	"I", "D", "A", "1", "3" en "ID A1.3"
\W	Coincide con cualquier carácter que no pertenezca a una palabra.	\W	" ". "." en "ID A1.3"
\s	Coincide con cualquier carácter que sea un espacio en blanco.	\w\s	"D " en "ID A1.3"
\S	Coincide con cualquier carácter que no sea un espacio en blanco.	\s\S	" _" en "int __ctr"
\d	Coincide con cualquier dígito decimal.	\d	"4" en "4 = IV"
\D	Coincide con cualquier dígito que no sea decimal.	\D	" ", "=", " ", "I", "V" en "4 = IV"

RegEx: Delimitadores

Aserción	Descripción	Modelo	Coincidencias
^	La coincidencia debe comenzar al principio de la cadena o de la línea.	^\d{3}	"901-" en "901-333-"
$	La coincidencia se debe producir al final de la cadena o antes de \n al final de la línea o de la cadena.	-\d{3}$	"-333" en "-901-333"
\A	La coincidencia se debe producir al principio de la cadena.	\A\d{3}	"901-" en "901-333-"
\Z	La coincidencia se debe producir al final de la cadena o antes de \n al final de la cadena.	-\d{3}\Z	"-333" en "-901-333"
\z	La coincidencia se debe producir al final de la cadena.	-\d{3}\z	"-333" en "-901-333"
\G	La coincidencia se debe producir en el punto en el que finalizó la coincidencia anterior.	\G\(\d\)	"(1)", "(3)", "(5(" en "(1)(3)(5)[7](9)"
\b	La coincidencia se debe producir en un límite entre un carácter \w (alfanumérico) y un carácter \W (no alfanumérico).	\b\w+\s\w+\b	"ellos ellos" en "ellos tema ellos ellos"
\B	La coincidencia no se debe producir en un límite \b.	\Bend\w*\b	"fin", "final" en "finalizar finalista finalizador finalizó"

RegEx: Construcciones de agrupamiento

Construcción de agrupamiento	Descripción	Modelo	Coincidencias
(subexpresión)	Captura la subexpresión coincidente y le asigna un número ordinal de base cero.	(\w)\1	"aa" en "aarón"
(?<name>subexpresión)	Captura la subexpresión coincidente en un grupo con nombre.	(?<double>\w)\k<double>	"aa" en "aarón"
(?<nombre1-nombre2>subexpresión)	Define una definición de grupo de equilibrio.	(((?'Open'\()[^\(\)]*)+((?'Close-Open'\))[^\(\)]*)+)*(?(Open)(?!))$	"((1-3)*(3-1))" en "3+2^((1-3)*(3-1))"
(?:subexpresión)	Define un grupo sin captura.	Write(?:Line)?	"WriteLine" en "Console.WriteLine()"
(?imnsx-imnsx:subexpresión)	Aplica o deshabilita las opciones especificadas dentro de subexpresión.	A\d{2}(?i:\w+)\b	"A12xl", "A12XL" en "A12xl A12XL a12xl"
(?=subexpresión)	Aserción de búsqueda anticipada positiva de ancho cero.	\w+(?=\.)	"es", "corría" y "hermoso" en "Él es. El perro corría. El sol está hermoso."
(?!subexpresión)	Aserción de búsqueda anticipada negativa de ancho cero.	\b(?!un)\w+\b	"seguro", "usado" en "aseguro seguro unidad usado"
(?<=subexpresión)	Aserción de búsqueda tardía positiva de ancho cero.	(?<=19)\d{2}\b	"99", "50", "05" en "1851 1999 1950 1905 2003"
(?<!subexpresión)	Aserción de búsqueda tardía negativa de ancho cero.	(?<!19)\d{2}\b	"51", "03" en "1851 1999 1950 1905 2003"
(?>subexpresión)	Subexpresión sin retroceso (o "expansiva").	[13579](?>A+B+)	"1ABB", "3ABB" y "5AB" en "1ABB 3ABBC 5AB 5AC"

RegEx: Cuantificadores

Cuantificador	Descripción	Modelo	Coincidencias
*	Coincide con el elemento anterior cero o más veces.	\d*.\d	".0", "19.9", "219.9"
+	Coincide con el elemento anterior una o más veces.	"be+"	"cai" en "caída", "be" en "bebé"
?	Coincide con el elemento anterior cero veces o una vez.	"rai?n"	"rata", "raicilla"
{n }	Coincide con el elemento anterior exactamente n veces.	",\d{3}"	",043" en "1,043.6", ",876", ",543", y ",210" en "9,876,543,210"
{n ,}	Coincide con el elemento anterior al menos n veces.	"\d{2,}"	"166", "29", "1930"
{n ,m}	Coincide con el elemento anterior al menos n veces, pero no más de m veces.	"\d{3,5}"	"166", "17668", "19302" en "193024"
*?	Coincide con el elemento anterior cero o más veces, pero el menor número de veces que sea posible.	\d*?.\d	".0", "19.9", "219.9"
+?	Coincide con el elemento anterior una o más veces, pero el menor número de veces que sea posible.	"be+?"	"cai" en "caída", "be" en "bebé"
??	Coincide con el elemento anterior cero o una vez, pero el menor número de veces que sea posible.	"rai??n"	"rata", "raicilla"
{n }?	Coincide con el elemento anterior exactamente n veces.	",\d{3}?"	",043" en "1,043.6", ",876", ",543", y ",210" en "9,876,543,210"
{n ,}?	Coincide con el elemento anterior al menos n veces, pero el menor número de veces posible.	"\d{2,}?"	"166", "29", "1930"
{n ,m}?	Coincide con el elemento anterior entre n y m veces, pero el menor número de veces posible.	"\d{3,5}?"	"166", "17668", "19302" en "193024"

RegEx: Construcciones de alternancia

Construcciones de alternancia	Descripción	Modelo	Coincidencias
|	Coincide con cualquier elemento separado por el carácter de barra vertical (|).	th(e|is|at)	"el", "este" en "este es el día. "
(?(expresión)sí|no)	Coincide con sí si expresión coincide; de lo contrario, coincide con la parte opcional no. expresión se interpreta como una aserción de ancho cero.	(?(A)A\d{2}\b|\b\d{3}\b)	"A10", "910" en "A10 C103 910"
(?(name)sí|no)	Coincide con sí si la captura con nombre nombre tiene una coincidencia; de lo contrario, coincide con la parte opcional no.	(?<quoted>")?(?(quoted).+?"|\S+\s)	Dogs.jpg, "Yiska playing.jpg" en "Dogs.jpg "Yiska playing.jpg""

RegEx: Sustituciones

Carácter	Descripción	Modelo	Modelo de reemplazo	Cadena de entrada	Cadena de resultado
$number	Sustituye la subcadena que coincide con el grupo número.	/b(/w+)(/s)(/w+)/b	$3$2$1	"one two"	"two one"
${name}	Sustituye la subcadena que coincide con el grupo nombre.	\b(?<word1>\w+)(\s)(?<word2>\w+)\b	${word2} ${word1}	"one two"	"two one"
$$	Sustituye un "$" literal.	\b(\d+)\s?USD	$$$1	"103 USD"	"$103"
$&	Sustituye una copia de toda la coincidencia.	(\$*(\d*(\.+\d+)?){1})	**$&	"$1.30"	"**$1.30**"
$`	Sustituye todo el texto de la cadena de entrada delante de la coincidencia.	B+	$`	"AABBCC"	"AAAACC"
$'	Sustituye todo el texto de la cadena de entrada detrás de la coincidencia.	B+	$'	"AABBCC"	"AACCCC"
$+	Sustituye el último grupo capturado.	B+(C+)	$+	"AABBCCDD"	AACCDD
$_	Sustituye toda la cadena de entrada.	B+	$_	"AABBCC"	"AAAABBCCCC"

JSON

- **JSON.parse** (Función): Convierte una cadena de la notación de objetos de JavaScript (JSON) en un objeto.
- **JSON.stringify** (Función): Convierte un valor de JavaScript en una cadena de la notación de objetos JavaScript (JSON).

Math

- **Math.E**: Constante matemática e. Es el número de Euler, base de los logaritmos naturales.
- **Math.LN2**: Logaritmo natural de 2.
- **Math.LN10**: Logaritmo natural de 10.
- **Math.LOG2E**: Logaritmo de base 2 de e.
- **Math.LOG10E**: Logaritmo de base 10 de e.
- **Math.PI**: Pi. Es la proporción entre la circunferencia de un círculo y su diámetro.
- **Math.SQRT1_2**: Raíz cuadrada de 0,5, o, de forma equivalente, uno dividido por la raíz cuadrada de 2.
- **Math.SQRT2**: Raíz cuadrada de 2.
- **Math.abs** (Función): Devuelve el valor absoluto de un número.
- **Math.acos** (Función): Devuelve el arco coseno de un número.
- **Math.asin** (Función): Devuelve el arcoseno de un número.
- **Math.atan** (Función): Devuelve el arco tangente de un número.
- **Math.atan2** (Función): Devuelve el ángulo, en radianes, desde el eje X a un punto representado por las coordenadas x e y proporcionadas.
- **Math.cos** (Función): Devuelve el coseno de un número.

Math

- **Math.ceil** (Función): Devuelve el entero más pequeño que sea mayor o igual que la expresión numérica proporcionada.
- **Math.exp** (Función): Devuelve e (base de los logaritmos naturales) elevado a una potencia.
- **Math.floor** (Función): Devuelve el entero más grande que sea menor o igual que la expresión numérica proporcionada.
- **Math.log** (Función): Devuelve el logaritmo natural de un número.
- **Math.max** (Función): Devuelve la mayor de dos expresiones numéricas proporcionadas.
- **Math.min** (Función): Devuelve el menor de dos números proporcionados.
- **Math.pow** (Función): Devuelve el valor de una expresión base elevada a una potencia especificada.
- **Math.random** (Función): Devuelve un número pseudoaleatorio entre 0 y 1.
- **Math.round** (Función): Devuelve una expresión numérica especificada redondeada al entero más cercano.
- **Math.sin** (Función): Devuelve el seno de un número.
- **Math.sqrt** (Función): Devuelve la raíz cuadrada de un número.
- **Math.tan** (Función): Devuelve la tangente de un número.

Nuevos Objetos (ES6)

- Symbol: Permite crear un identificador único.
- Promise: Proporciona un mecanismo para programar el trabajo de modo que se lleve a cabo en un valor que todavía no se calculó.
- Proxy: Habilita el comportamiento personalizado de un objeto.
- Reflect: Proporciona métodos para su uso en las operaciones que se interceptan.
- Intl.Collator: Proporciona comparaciones de cadenas de configuración regional.
- Intl.DateTimeFormat: Proporciona formato de fecha y hora específico de la configuración regional.
- Intl.NumberFormat: Proporciona formato de número específico de la configuración regional.

- Map: Lista de pares clave-valor.
- Set: Colección de valores únicos que pueden ser de cualquier tipo.
- WeakMap: Colección de pares clave-valor en los que cada clave es una referencia de objeto.
- WeakSet: Colección de objetos únicos.

Nuevos Objetos (ES6)

- ArrayBuffer: Representa un búfer sin formato de datos binarios, que se usa para almacenar datos de las diferentes matrices con tipo. No se puede leer directamente de Arraybuffer ni escribir directamente en Arraybuffer, pero se puede pasar a una matriz con tipo o un objeto DataView para interpretar el búfer sin formato según sea necesario.
- DataView: Se usa para leer y escribir diferentes tipos de datos binarios en cualquier ubicación de ArrayBuffer.
- Float32Array: Matriz con tipo de valores flotantes de 32 bits.
- Float64Array: Matriz con tipo de valores flotantes de 64 bits.
- Int8Array: Matriz con tipo de valores enteros de 8 bits.
- Int16Array: Matriz con tipo de valores enteros de 16 bits.
- Int32Array: Matriz con tipo de valores enteros de 32 bits.
- Uint8Array: Matriz con tipo de valores enteros sin signo de 8 bits.
- Uint8ClampedArray: Matriz con tipo de enteros sin signo de 8 bits con valores fijos.
- Uint16Array: Matriz con tipo de valores enteros sin signo de 16 bits.
- Uint32Array: Matriz con tipo de valores enteros sin signo de 32 bits.

Symbol (ES6)

- Por especificación, las claves (Keys) de un objeto deben ser solamente del tipo String o Symbol. El valor de “Symbol” representa un identificador único.

```
let id = Symbol();
let id = Symbol("id"); // es un symbol con la descripción "id"
```

- Se garantiza que los símbolos son únicos. Aunque declaremos varios Symbols con la misma descripción, éstos tendrán valores distintos. La descripción es solamente una etiqueta que no afecta nada más. Los Symbols no se auto convierten a String, hay que hacerlo explícitamente con .description o .toString()
- Los Symbols nos permiten crear claves “ocultas” en un objeto, a las cuales ninguna otra parte del código puede accesar ni sobrescribir.

```
let user = {
  [id]: 123 // no "id": 123
  name: "John",
}
user[id] = "Su id";
```


Promise Pattern (ES6)

- El Promise Pattern es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutaran a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón Callback:
  - Llamadas anidadas
  - Complejidad de código

o.m(1, 2, f(m1(3, f1(4,5,ff(8))) → o.m(1, 2).f().m1(3).f1(4 , 5).ff(8)

- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- El servicio $q es un servicio de AngularJS que contiene toda la funcionalidad de las promesas (está basado en la implementación Q de Kris Kowal).
- La librería JQuery incluye el objeto $.Deferred desde la versión 1.5.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión 6.

Objeto Promise (ES6)

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, …) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
  - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
  - Resuelta: Se ha podido obtener el resultado (Promise.resolve())
  - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (Promise.reject())
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

Crear promesas (ES6)

- El objeto Promise gestiona la creación de la promesa y los cambios de estados de la misma.

```
list() {
  return new Promise((resolve, reject) => {
   this.http.get(this.baseUrl).subscribe(
       data => resolve(data),
       err => reject(err)
       )
  });
 }
```

- Para crear promesas ya concluidas:
  - Promise.resolve: Crea una promesa nueva como resuelta cuyo resultado es igual que su argumento.
  - Promise.reject: Crea una promesa nueva como rechazada cuyo resultado es igual que el argumento pasado.

Invocar promesas (ES6)

- El objeto Promise creado expone los métodos:
  - then(fnResuelta, fnRechazada): Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
  - catch(fnRechazada): Recibe como parámetro la función a ejecutar en caso de que falle.

    list().then(calcular, ponError).then(guardar)
  - finally(fnResueltaOrRechazada): Recibe como parámetro la función a ejecutar tanto en caso de que se resuelva o se rechace (ES2018).
- Otras formas de crear e invocar promesas son:
  - Promise.all: Combina dos o más promesas y realiza la devolución solo cuando todas las promesas especificadas se completan o alguna sea rechaza.
  - Promise.race: Crea una nueva promesa que resolverá o rechazará con el mismo valor de resultado que la primera promesa que se va resolver o rechazar entre los argumentos pasados.

DETECCIÓN Y CORRECCIÓN DE ERRORES

Depuración

- La mayoría de los navegadores modernos ofrecen utilidades de desarrollo y depuración.
- Cada depurador ofrece:
  - Un editor multi-linea para experimentar con JavaScript;
  - Un inspector para revisar el código generado en la página;
  - Un visualizador de red o recursos, para examinar las peticiones que se realizan.
- Suelen suministrar un objeto console con los siguientes métodos:
  - console.log() para enviar y registrar mensajes generales.
  - console.dir() para registrar un objeto y visualizar sus propiedades.
  - console.warn() para registrar mensajes de alerta.
  - console.error() para registrar mensajes de error.
- Existen otros métodos para utilizar desde la consola, pero estos pueden variar según el navegador. La consola además provee la posibilidad de establecer puntos de interrupción y observar expresiones en el código con el fin de facilitar su depuración.

NUEVAS VERSIONES ECMASCRIPT

Introducción

- A partir de la versión 6 cambio la política de versionado y sacan una revisión anual en Junio con los pocos cambios hasta ese momento: ECMAScript 2016 (ES7), ECMAScript 2017 (ES8), ECMAScript 2018 (ES9), ECMAScript 2019 (ES10), ECMAScript 2020 (ES11), …
- Las nuevas versiones se van extendiendo progresivamente, los navegadores deben dar soporte pero los usuarios deben actualizar sus navegadores en caso de que sea posible, esto es un proceso lento que requiere su tiempo. En NodeJS es mas ágil.
- Las alternativas para empezar a utilizarlo son:
  - Transpiladores ("Transpiler": "Tanslator" y "Compiler"): Traducen o compilan un lenguaje de alto nivel a otro lenguaje de alto nivel, en este caso código ES6 a ES5. Los más conocidos y usados son Babel, TypeScript, Google Traceur, CoffeeScript.
  - Polyfill: Un fragmento de código o un plugin que permite tener las nuevas funcionalidades de HTML o ES en aquellos navegadores que nativamente no lo soportan. (core-js, polyfill.io, …)

EcmaScript 6

- http://www.ecma-international.org/ecma-262/6.0/
- http://es6-features.org/
- https://kangax.github.io/compat-table/es2016plus/
- https://babeljs.io/
- https://github.com/google/traceur-compiler
- https://www.typescriptlang.org/
- https://github.com/zloirock/core-js
- https://es.javascript.info/

EcmaScript 2016 (ES7)

- El operador exponencial (**):
  - x = Math.pow(3, 2);
  - x = 3 ** 2;
  - x = 3; x **= 2;
- Método Array.prototype.includes()
  - if(listado.indexOf(item) !== -1) // lo contiene
  - if(listado.includes(item)) // lo contiene

EcmaScript 2017 (ES8)

- Object.values(): devuelve un array con los valores correspondientes a las propiedades enumerables de un objeto.
- Object.entries(): devuelve una matriz de arrays [key, value] del objeto dado.
- Object.getOwnPropertyDescriptors(): devuelve todos los descriptores de propiedad propios de un objeto dado para su clonado.
  - value: El valor asociado con la propiedad (solo descriptores de datos).
  - writable: true si y solo si el valor asociado con la propiedad puede ser cambiado (solo descriptores de datos).
  - get: Un función que sirve como un getter para la propiedad, o undefined si no hay getter (solo descriptores de acceso).
  - set: Una función que sirve como un setter para la propiedad, o undefined si no hay setter (solo descriptores de acceso).
  - configurable: true si y solo si el tipo de este descriptor de propiedad puede ser cambiado y si la propiedad puede ser borrada de el objeto correspondiente.
  - enumerable: true si y solo si esta propiedad aparece durante la enumeración de las propiedad en el objeto correspondiente.
- str.padStart() y str.padEnd() rellenan la cadena actual, por el principio o por el final, con la cadena dada (repitiéndola si es necesaria) para que la cadena resultante alcance una longitud dada, si aun no la tiene.
- Ahora permite tener comas finales después del último parámetro de función.

asyc/await (ES8)

- La declaración de función asyc define una función asíncrona, que devuelve un objeto AsyncFunction. Una función asíncrona es una función que opera asincrónicamente a través del bucle de eventos, utilizando una promesa implícita para devolver su resultado. Pero la sintaxis y la estructura de su código usando funciones asíncronas se parece mucho más a las funciones síncronas estándar.
- El operador await se usa para esperar a una Promise y sólo dentro de una async function.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => { setTimeout(() => { resolve(x); }, 2000); });
}
async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}
```


Memoria compartida con acceso atómico (ES8)

- Cuando dos threads tienen acceso a un espacio de memoria compartido, es importante que las operaciones de escritura o lectura sean atómicas, es decir, que se hagan en bloque y aisladas de otros threads, para garantizar la validez de los datos.
- Esta funcionalidad está pensada para web workers, ya que es el único caso en el se puede implementar multithreading en Javascript, y de momento está orientada a muy bajo nivel: ArrayBuffers (arrays de datos binarios).
- El mecanismo consta de:
  - SharedArrayBuffer: Un nuevo constructor para crear ArrayBuffers de memoria compartida.
  - La clase Atomic, con métodos estáticos para acceder/manipular ese SharedArrayBuffer de forma atómica.

EcmaScript 2018 (ES9)

- ES6 incorporaba el spread operator y los parámetros rest para hacer asignaciones por destructuring con Arrays.

```
const primes = [2, 3, 5, 7, 11];
const [first, second, ...rest] = primes;
console.log(rest); // [5, 7, 11]
const primes2 = [first, second, ...rest];
console.log(primes2); // [2, 3, 5, 7, 11]
```

- ES9 permite hacer asignaciones por destructuring con objetos.

```
const person = { firstName: 'Peter', lastName: 'Parker', age: 26, }
const { age, ...name } = person;
console.log(name); // {firstName: "Peter", lastName: "Parker"}
const person2 = { age, ...name };
console.log(person2); // {age: 26, firstName: "Peter", lastName: "Parker"}
```

- promesa.finally: permite llamar siempre a una función de callback sin parámetros al completarse (resuelta o rechazada).
- For asíncrono: for await (const line of readLines(filePath)) {

EcmaScript 2018 (ES9)

- Mejoras en las expresiones regulares:
  - Named capture groups

```
let re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
let result = re.exec('2015-01-02');
// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02';
```

  - Comprobar si la expresión esta precedida de un determinado patrón sin incluirla en el resultado:

```
const regex = /(?<=€)([0-9])+/;
let result = regex.test('€2000'); // result === 2000
```

  - El indicado /s evita los problemas con las secuencias de escape del . como comodín:

```
/hola.mundo/.test('hola\tmundo'); //false
/hola.mundo/s.test('hola\tmundo'); //true
```

EcmaScript 2019 (ES10)

- Object.fromEntries(): transforma una lista de arrays [clave-valor] en un objeto, proceso contrario al método entries()

```
const arr = [["firstName", "Peter"], ["lastName", "Parker"], ["age", 26]];
Object.fromEntries(arr) // = { firstName: 'Peter', lastName: 'Parker', age: 26 }
```

- Array.prototype.flat(): permite aplanar un array multidimensional por niveles.

```
const arr = ['1', '2', ['3.1', '3.2'], '4', ['5.1', ['5.2.1', '5.2.2']], '6'];
arr.flat();      // ["1", "2", "3.1", "3.2", "4", "5.1", Array(2), "6"]
arr.flat(2);     // ["1", "2", "3.1", "3.2", "4", "5.1", "5.2.1", "5.2.2", "6"]
```

- Array.prototype.flatMap(): combina el método map(callback) con el flat() para transformar y aplanar un array de forma eficiente.

```
const arr = [[1, 2], [3, 4], [5, 6]];
arr.map(([a, b]) => [a, a * b]) // [[1, 2], [3, 12], [5, 30]]
   .flat() // [1, 2, 3, 12, 5, 30]
arr.flatMap(([a, b]) => [a, a * b]) // [1, 2, 3, 12, 5, 30]
```


EcmaScript 2019 (ES10)

- String.trimStart(), String.trimEnd(): eliminan los espacios en blanco al principio o del final de un string. Por consistencia con su proceso contrario padStart y padEnd, se ha mantenido la nomenclatura pero se añaden también como alias trimLeft y trimRight por compatibilidad con otros.
- Symbol.description: Obtiene la cadena asociada al símbolo.
- Function.prototype.toString() ahora devuelve la cadena con el código fuente sin eliminar espacios en blanco, comentarios y saltos de línea.
- La captura del error en una referencia al declarar el catch ahora es opcional:

```
try {
   :
} catch {    // catch(e)
   :
}
```

- JSON.stringify(), independiente del formato de entrada, debe devolver una cadena UTF-8 bien formada y JSON.parse() debe aceptar los símbolos de separador de línea (\u2028) y separador de párrafo (\u2029).

ECMAScript 2020 (ES11)

- Operadores:
  - Encadenamiento opcional, cortocircuitado (?.):
    - v?.p → v === null ? null : v.p
  - Selección de valores (||):
    - v || "default!" → v ? v : "default!"
  - Coalescencia nula (??):
    - v ?? "default!" → (v == null || v == undefined) ? "default!" : v
- El objeto globalthis establece una forma universal de acceder al objeto global (this) en JavaScript que anteriormente dependía del entorno, usando window o self en el navegador, o global en NodeJs.
- Ahora el orden de las propiedades está garantizado en el bucle for in o JSON.stringify.

ECMAScript 2020 (ES11)

- Aparece el BigInt para superar algunas limitaciones de (2^{53}) de ciertas APIS cuando los ids numéricos son muy grandes (los ids usados por Twitter), y cuyo valor superaba el valor de Number.MAX_SAFE_INTEGER, obligando a tener que tratarlos como strings. Se pueden crear números enteros más grandes añadiendo una "n" al final del entero o mediante la función BigInt.
- Strings.matchAll(): genera un iterador con todos los objetos coincidentes generados por una expresión regular global.
- Promise.allSettled: Ejecuta una colección de promesas y devuelve el resultado de cada una de ellas para su tratamiento como resuelta o rechazada.

ECMAScript 2020 (ES11)

- Importación dinámica de módulos como promesas:

```
import(`./my-modules/${ myModuleName }.js`)
  .then(module => {  module.doStuff(); })
  .catch(err => console.log(err));
(async () => {
  const module = await import (`./my-modules/${myModuleName }.js`);
  module2.doStuff();
})();
```

- El objeto import.meta expone el contenido especifico con los metadatos de módulo JavaScript.

```
<script type="module" src="my-module.js"></script>
console.log(import.meta); // { url: "file:///home/user/my-module.js" }
```

- Propagación (importación y exportación):

```
export * as ns from 'module'
```


ECMAScript 2021 (ES12)

- Separador numérico: permite separar los números con guiones bajo (_) sin afectar al valor (azúcar sintáctica): 100_000_000 === 100000000
- Operador de asignación lógica: se incorporan los operadores de asignación lógica &&= ||= y ??=
- Métodos, getters y setters privados en las clases: con el prefijo # en su nobre verán su alcance limitado a la clase donde están definidos.

```
#private(val) { … }
set #width  (width)  { this.#width = width; }
get width  ()      { return this.#width ; }
```

- String.replaceAll() reemplaza todas las ocurrencias de una cadena por otra cadena (replace() solo reemplazaba la primera ocurrencia).
- Promise.any(), cuyo argumento es un array de promesas, será gestionado capturando la respuesta de la primera que sea resuelta de forma satisfactoria, o un array de excepciones en caso de que todas sean rechazadas

```
Promise.any([promise1, promise2, promise3])
```

ECMAScript 2022 (ES13)

- Await a nivel superior: permite usar el await sin necesidad de estar contenido dentro de una función asíncrona (marcada con async).
- Object.hasOwn(): devuelve true si el objeto especificado tiene la propiedad indicada como propiedad propia. Si la propiedad es heredada, o no existe, el método devuelve false.
- Array.at(): recibe un valor numérico entero y devuelve el elemento en esa posición, permitiendo valores positivos y negativos. Los valores negativos contarán desde el último elemento del array.

```
[1,2,3,4,5].at(-1)   // devuelve 5
```

- error.cause: permite especificar que causó un error.

```
try {
  connectToDatabase();
} catch (err) {
  throw new Error("Connecting to database failed.", { cause: err });
}
```


ECMAScript 2022 (ES13)

- Propiedades y métodos privados: Se pueden crear atributos y métodos privados en la clases usando el prefijo # en el nombre.

```
class Usuario{
  #nombre;
  constructor(nombre) {
    this.#nombre = nombre;
  }
  #apellidos(apellido) {
    return this.#nombre + ' ' + apellido;
  }
  mostrar(apellido) {
    console.log(this.#apellidos(apellido));
  }
}

const usuario = new Usuario("Pepito");
console.log(usuario.#nombre);                 // Error
console.log(usuario.#apellidos("Grillo"));    // Error
console.log(usuario.mostrar("Grillo"));       // Pepito Grillo
```