

# PCW

---

## PROGRAMACIÓN DEL CLIENTE WEB

---

### Tema 04 - JavaScript



Dept. de Ciència de la Computació i Intel·ligència *artificial*  
Dpto. de Ciencia de la Computación e Inteligencia *artificial*



Universitat d'Alacant  
Universidad de Alicante

# JavaScript

# JavaScript

- ✓ Introducción
- ✓ Variables, constantes y tipos de datos
- ✓ Expresiones y operadores
- ✓ Literales
- ✓ Objetos
- ✓ Prototipado
- ✓ Control del flujo de ejecución
- ✓ Funciones

## Introducción

- JavaScript es un **lenguaje de programación interpretado**.
- Se define como **orientado a objetos**, basado en prototipos, imperativo, débilmente tipado y dinámico.
- Se utiliza principalmente **en el lado del cliente web**, permitiendo añadir mejoras y dinamismo a la interfaz de usuario.
- Tradicionalmente sólo servía para trabajar en el lado del cliente interactuando con el documento web HTML mediante el **Document Object Model (DOM)**.
- Hoy en día es ampliamente empleado para enviar y recibir información del servidor empleando otras tecnologías como **AJAX**.

# Introducción

## Introducción (y II)

- **Sintaxis similar a Java**, aunque no tienen nada que ver.
- Es sensible a mayúsculas y minúsculas (**case-sensitive**).
- Aunque no es obligatorio, **se recomienda** terminar las instrucciones con “;”
- Los comentarios se hacen igual que en C++:

```
// esto es un comentario de una línea  
/* esto es un  
   comentario multilínea  
*/
```

- Aunque no es obligatorio, **se recomienda utilizar el modo estricto**. Permite detectar errores en el código antes de que se produzcan en la ejecución. Para activar el modo estricto sólo es necesario incluir como primera línea de código la siguiente directiva:

```
'use strict';
```

# Variables, constantes y tipos de datos

# Variables, constantes y tipos de datos

## Variables

- Son nombres simbólicos, llamados identificadores, que **permiten guardar valores**.
- **Un nombre de variable debe empezar** con una letra (A-Z, a-z), el carácter “\_” o el carácter “\$”. Los siguientes caracteres pueden ser también números (0-9).
- Hay **palabras reservadas** que no se pueden utilizar como nombre de variable.
- Aunque las últimas revisiones de JavaScript permiten utilizar símbolos unicode como ñ, á, è, ü, etc, no se recomienda.
- Se recomienda utilizar el modo ***camelCase*** para nombrar variables: nuevoUsuario, numeroDeVisitas, etc.
- Se recomienda utilizar nombres descriptivos y concisos.



# Variables, constantes y tipos de datos

## Variables. Declaración.

Se pueden declarar de tres formas:

1. Simplemente asignando un valor a la variable: `x = 25;`

De esta forma se declara la variable como global. Aunque está permitido, **no se recomienda** utilizar esta forma.

2. Con la palabra clave **var**: `var x = 17;`

Permite declarar variables locales y globales. Tiene ámbito de función. Permite redeclaración de variables.

3. Con la palabra clave **let**: `let x = 3;`

Permite declarar variables locales con ámbito de bloque, es decir, sólo existirán dentro del bloque de código en el que se han declarado. No permite redeclaración de variables.

# Variables, constantes y tipos de datos

## Variables. Declaración.

- Cualquier variable declarada sin valor asignado, su valor es **undefined**.

```
var b;  
  
console.log(b); // El valor de b es undefined  
let x = 5, y;  
console.log(x); // El valor de x es 5  
console.log(y); // El valor de y es undefined
```

**undefined** se comporta como **false** en un contexto booleano:

```
var b;  
  
console.log( !b ); // La respuesta es true
```

# Variables, constantes y tipos de datos

## Variables. Ámbito.

El ámbito de una variable engloba la parte de código desde la que es accesible. Los tipos de ámbito son:

- **Global**. Cuando la variable se declara fuera de cualquier función. Es accesible desde cualquier parte del código.
- **Local**. Cuando la variable se declara dentro de una función. Sólo es accesible desde la función.
- **De bloque**. Cuando la variable se declara mediante `let` dentro de un bloque de código. Sólo es accesible dentro del bloque.

# Variables, constantes y tipos de datos

## Variables. Ámbito.

Ejemplo:

```
var b; // variable global

function foo(){
  var x = 5; // variable local. Ámbito de función.
  console.log( 'x:' + x ); // x:5

  for(let i = 1; i < x; i++){ // variable de bloque
    console.log('i: ' + i); // i:1 i:2 ...
  }
  console.log(i); // undefined
}
```

# Variables, constantes y tipos de datos

## Constantes.

Se declaran utilizando la palabra reservada `const`:

```
const x = 25;
```

Las constantes tienen ámbito de bloque, es decir sólo existirán en el bloque de código en el que se declaran y son de sólo lectura.

Se pueden declarar más de una constante en la misma declaración:

```
const nombre1 = valor1 [, nombre2 = valor2 [, ... [,  
    nombreN = valorN]]];
```

# Variables, constantes y tipos de datos

## Tipos de datos.

- JavaScript es un lenguaje **débilmente tipado**
- No es necesario declarar el tipo de variable.
- El tipo se determina automáticamente cuando se le asigna un valor a la variable.

```
var a = 18;      // a es de tipo Number  
var a = 'Hola'; // a es de tipo String  
var a = true;    // a es de tipo Boolean
```

# Variables, constantes y tipos de datos

## Tipos de datos.

- **Undefined.** Tipo de dato de variables declaradas a las que no se les ha asignado un valor, o de argumentos de funciones no existentes. Valor undefined.
- **Boolean.** Para almacenar valores lógicos: true, false.
- **Number.** Tipo de dato numérico de doble precisión en coma flotante (64 bits). Permite almacenar Integer, Float, Double y Bignum.
- **BigInt.** Para almacenar números enteros de longitud variable y mayores de  $2^{53} - 1$ , que es el máximo valor permitido por Number.
- **String.** Tipo de dato para almacenar una secuencia de caracteres que representan un texto.
- **Symbol.** Nuevo en ECMAScript 6. Permite crear datos anónimos.
- **Null.** Tipo de dato de un objeto vacío o no existente. Valor null.
- **Object.** Estructura que contiene datos e instrucciones para trabajar con los datos.

Los siete primeros tipos de datos son primitivos.

# Expresiones y operadores



# Expresiones y operadores

## Expresiones

- Una **expresión** es cualquier acción u operación que produce un resultado.

Ejemplo:

```
var diferencia = valor_entrada - valor_salida;
```

### Definición de expresión.

Una **expresión** en JavaScript es una secuencia ordenada de operandos y operadores, que es evaluada por el intérprete, produciendo un resultado.

# Expresiones y operadores

## Operadores

En JavaScript se tienen los siguientes tipos de operadores:

- Operadores de **asignación**
- Operadores de **comparación**
- Operadores **aritméticos**
- Operadores **bit a bit**
- Operadores **lógicos**
- Operadores de **cadenas de caracteres**
- Operador **condicional** (ternario)
- Operador **coma**
- Operadores **unarios**

# Expresiones y operadores

## Operadores. Asignación.

Nombre del operador	Abreviatura	Significado
Operador de asignación	$x = y$	$x = y$
Asignación de adición	$x += y$	$x = x + y$
Asignación de sustracción	$x -= y$	$x = x - y$
Asignación de multiplicación	$x *= y$	$x = x * y$
Asignación de división	$x /= y$	$x = x / y$
Asignación de resto (módulo)	$x %= y$	$x = x \% y$
Asignación de exponenciación	$x **= y$	$x = x ** y$

# Expresiones y operadores

## Operadores. Asignación.

Nombre del operador	Abreviatura	Significado
Asignación de desplazamiento de bit hacia la izquierda	$x \ll= y$	$x = x \ll y$
Asignación de desplazamiento de bit hacia la derecha	$x \gg= y$	$x = x \gg y$
Asignación de desplazamiento de bit hacia la derecha sin signo	$x \ggg= y$	$x = x \ggg y$
Asignación AND binaria	$x \&= y$	$x = y \& y$
Asignación XOR binaria	$x \^= y$	$x = y \^ y$
Asignación OR binaria	$x  = y$	$x = y   y$

# Expresiones y operadores

## Operadores. Asignación.

**Asignación desestructurada**. JavaScript permite desestructurar arrays y objetos, y asignar sus elementos directamente a variables.

**Ejemplo**:

```
// Desestructurando arrays:
var vector = [ -1, 'hola', 5, true]

var [a, b] = vector; // Resultado: a = -1 y b = 'hola'
var [a, , b] = vector; // Resultado: a = -1 y b = 5
var [, a, , b] = vector; // Resultado: a = 'hola' y b = true
var [a, b, c, d] = vector; // Resultado: a = -1, b = 'hola', c = 5 y d = true
var [a, b, c, d, e = 10, f] = vector;
// Resultado: a = -1, b = 'hola', c = 5, d = true, e = 10, f = undefined

// Desestructurando objetos:
var persona = {nombre:'Juan', apellidos:'Sin Miedo', dni:'12345678A', edad:23};

var {nombre:n, dni:d, edad: e} = persona;
console.log('Nombre: ' + n + ' - DNI: ' + d + ' - EDAD: ' + e);
// Resultado: Nombre: Juan - DNI: 12345678A - EDAD: 23
```

# Expresiones y operadores

## Operadores. Comparación.

Operador	Ejemplo	Descripción
Igualdad ( <b>==</b> )	<code>3 == var1</code> <code>'3' == var1</code> <code>3 == '3'</code>	Devuelve true si ambos operandos son iguales
Desigualdad ( <b>!=</b> )	<code>var3 != 4</code> <code>var2 != '3'</code>	Devuelve true si ambos operandos no son iguales
Estrictamente iguales ( <b>===</b> )	<code>3 === var1</code> <code>'3' === '3'</code>	Devuelve true si ambos operandos son iguales y tienen el mismo tipo
Estrictamente desiguales ( <b>!==</b> )	<code>3 !== var1</code> <code>'3' !== 3</code>	Devuelve true si ambos operandos no son iguales y/o no son del mismo tipo

## Expresiones y operadores

### Operadores. Comparación.

Operador	Ejemplo	Descripción
Mayor que ( <b>&gt;</b> )	var2 > var1 '12' > 2	Devuelve true si el operando de la izquierda es mayor que el de la derecha
Mayor o igual que ( <b>&gt;=</b> )	var2 >= var1 var1 >= 3	Devuelve true si el operando de la izquierda es mayor o igual que el de la derecha
Menor que ( <b>&lt;</b> )	var1 < var2 '2' < 12	Devuelve true si el operando de la izquierda es menor que el de la derecha
Menor o igual que ( <b>&lt;=</b> )	var1 <= var2 var2 <= 7	Devuelve true si el operando de la izquierda es menor o igual que el de la derecha

# Expresiones y operadores

## Operadores. Aritméticos.

Operador	Ejemplo	Descripción
Resto/Módulo ( <b>%</b> )	12 % 5 devuelve 2	Devuelve el resto de la división de los dos operandos
Incremento ( <b>++</b> )	++x (preincremento) x++ (postincremento)	Con preincremento, primero suma 1 a x y luego devuelve el nuevo valor de x. Con postincremento, primero devuelve el valor de x y luego le suma 1.
Decremento ( <b>--</b> )	--x (predecremento) x-- (postdecremento)	Con predecremento, primero resta 1 a x y luego devuelve el nuevo valor de x. Con postdecremento, primero devuelve el valor de x y luego le resta 1.



# Expresiones y operadores

## Operadores. Aritméticos.

Operador	Ejemplo	Descripción
Negación unaria ( <b>-</b> )	<code>- '3'</code> devuelve <code>-3</code> <code>-true</code> devuelve <code>-1</code>	Intenta convertir a número el operando y devuelve su forma negativa
Unario positivo ( <b>+</b> )	<code>+ '3'</code> devuelve <code>3</code> <code>+true</code> devuelve <code>1</code>	Intenta convertir a número el operando
Exponenciación ( <b>**</b> )	<code>2 ** 3</code> devuelve <code>8</code> <code>10 ** -1</code> devuelve <code>0.1</code>	Calcula la potencia de la base al valor del exponente

# Expresiones y operadores

## Operadores. A nivel de bit.

Operador	Ejemplo	Descripción
AND bit a bit ( <b>&amp;</b> )	a & b	Devuelve uno por cada posición de bit en la cual los correspondientes bits de ambos operandos tienen valor uno
OR bit a bit ( <b> </b> )	a   b	Devuelve uno por cada posición de bit en la cual al menos uno de los correspondientes bits de ambos operandos tiene valor uno
XOR bit a bit ( <b>^</b> )	a ^ b	Devuelve uno por cada posición de bit en la cual los correspondientes bits de ambos operandos son diferentes y cero cuando son iguales
NOT bit a bit ( <b>~</b> )	~ b	Invierte los bits del operando

# Expresiones y operadores

## Operadores. A nivel de bit.

Operador	Ejemplo	Descripción
Desplazamiento a izquierda ( <b>&lt;&lt;</b> )	<code>a &lt;&lt; b</code>	Desplaza <b>b</b> posiciones a la izquierda la representación binaria de <b>a</b> , el exceso de bits de la izquierda se descarta, rellenando con ceros por la derecha
Desplazamiento a derecha con propagación de signo ( <b>&gt;&gt;</b> )	<code>a &gt;&gt; b</code>	Desplaza <b>b</b> posiciones a la derecha la representación binaria de <b>a</b> , el exceso de bits de la derecha se descarta (el bit de signo no se desplaza)
Desplazamiento a derecha con relleno de ceros ( <b>&gt;&gt;&gt;</b> )	<code>a &gt;&gt;&gt; b</code>	Desplaza <b>b</b> posiciones a la derecha la representación binaria de <b>a</b> , el exceso de bits de la derecha se descarta, rellenando con ceros por la izquierda (el bit de signo se desplaza a la derecha)

# Expresiones y operadores

## Operadores. A nivel de bit.

### Importante:

- ✓ Los operandos son convertidos a enteros de 32 bits
- ✓ Se descartan los bits más relevantes de los números con más de 32 bits

### Ejemplo:

```
Antes:      111001101111101000000000000000001100000000000001
Después:      10100000000000000000001100000000000001
```

# Expresiones y operadores

## Operadores. Lógicos.

Operador	Ejemplo	Descripción
AND lógico ( <b>&amp;&amp;</b> )	expr1 && expr2	Devuelve true si ambos operandos son true. En caso contrario devuelve false.
OR lógico ( <b>  </b> )	expr1    expr2	Devuelve true si al menos uno de los dos operandos es true. Sólo devuelve false cuando ambos operandos son false.
NOT lógico ( <b>!</b> )	!expr	Devuelve false si su operando puede ser convertido a true. En caso contrario devuelve true.

# Expresiones y operadores

## Operadores. Cadenas de caracteres.

- Operador de **concatenación**: **+**

Ejemplo:

```
var nombre = 'Juan';  
var apellidos = 'García';  
apellidos += ' Pérez';  
console.log('Nombre: ' + nombre + ' ' + apellidos);  
// Nombre: Juan García Pérez
```

## Expresiones y operadores

### Operadores. Condicional ternario.

- Único operador en JavaScript que necesita tres operandos.
- Asigna uno de dos valores basándose en la condición.
- Operador **condicional ternario**: **?:**

Sintaxis: condición **?** valor1 **:** valor2

#### Ejemplo:

```
var estatura = (altura >= 180) ? 'alto' : 'bajo';
```

Es equivalente a:

```
var estatura;  
if(altura >= 180)  
    estatura = 'alto';  
else  
    estatura = 'bajo';
```

# Expresiones y operadores

## Operadores. Coma.

- Evalúa los operandos a ambos lados de la coma y devuelve el valor del último.
- Se suele utilizar dentro de los bucles for para actualizar más de una variable en cada iteración.

### Ejemplo:

```
for (var i = 0, j = 9; i <= j; i++, j--)  
    console.log('a[' + i + '][' + j + ']= ' + a[i][j]);
```

En el ejemplo anterior se está imprimiendo los elementos de la diagonal principal de una matriz y en cada iteración se están actualizando las dos variables que se utilizan como índices.



# Expresiones y operadores

## Operadores. Unarios.

Sólo necesitan un operando.

- **delete**. Permite eliminar un objeto, una propiedad de un objeto, o un elemento, con el índice especificado, de un Array.

### Ejemplo:

```
delete nombreObjeto;  
delete nombreObjeto.propiedad;  
delete nombreObjeto[indice];
```

Si la operación acaba con éxito, establece la propiedad o el elemento a undefined.

# Expresiones y operadores

## Operadores. Unarios.

- **typeof**. Devuelve una cadena de caracteres indicando el tipo del operando evaluado.

### Ejemplo:

```
var miFuncion = new Function('5 + 2');  
var forma = 'redonda';  
var largo = 1;  
var hoy = new Date();
```

```
console.log(typeof(miFuncion)); // devuelve: 'function'  
console.log(typeof(forma));     // devuelve: 'string'  
console.log(typeof(largo));     // devuelve: 'number'  
console.log(typeof(hoy));       // devuelve: 'object'  
console.log(typeof noExiste);   // devuelve: 'undefined'  
console.log(typeof 'Hola');     // devuelve: 'string'
```

Aunque los paréntesis son opcionales, se recomienda utilizarlos:

**typeof**(*expr*)

# Expresiones y operadores

## Operadores. Unarios.

- **void**. Evalúa la expresión que se le pasa y no devuelve ningún resultado. Se utiliza sobre todo cuando se crean enlaces como botones para evitar que recargue la página.

### Ejemplo:

```
<a href="javascript:void(0)"  
  onclick="alert('Hola mundo!!!');">Click</a>
```

# Expresiones y operadores

## Operadores. Precedencia de operadores.

Tipo de operador	Operador
miembro	<code>.</code> <code>[]</code>
llamar / crear instancia	<code>()</code> <code>new</code>
negación / incremento	<code>!</code> <code>~</code> <code>-</code> <code>+</code> <code>++</code> <code>--</code> <code>typeof</code> <code>void</code> <code>delete</code>
multiplicación / división	<code>*</code> <code>/</code> <code>%</code>
adición / sustracción	<code>+</code> <code>-</code>
desplazamiento binario	<code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>&gt;&gt;&gt;</code>
relación	<code>&lt;</code> <code>&lt;=</code> <code>=&gt;</code> <code>&gt;</code>
igualdad	<code>==</code> <code>!=</code> <code>===</code> <code>!==</code>

# Expresiones y operadores

## Operadores. Precedencia de operadores (continuación).

Tipo de operador	Operador
AND binario	&
XOR binario	^
OR binario	
AND lógico	&&
OR lógico	
condicional	? :
asignación	= += -= *= /= %= <<= >>= >>>= &= ^=  =
coma	,

# Literales

## Literales

- Se utilizan para representar valores en JavaScript
- Son valores fijos proporcionados *literalmente* en el código.
- Hay varios tipos de literales:
  - Array literals
  - Boolean literals
  - Floating-point literals
  - Integer literals
  - Object literals
  - RegExp literals
  - String literals

## Literales.

### ■ Array literals

- Se escribe como una lista de cero o más expresiones, cada una de las cuales representa un elemento de un array, encerrado en corchetes ([ ]).
- El array creado se inicializa con los valores indicados y su longitud es el número de elementos especificados.

#### Ejemplo:

```
var capitales = ['Madrid', 'París', 'Roma', 'Londres'];
```

No es necesario especificar todos los elementos del array, se pueden dejar “huecos”:

```
var nombres = ['Ana', 'Juan', , 'Luis', ' ', 'María'];
```



## Literales.

### ■ Boolean literals

- Tiene dos valores literales: **true**, **false**.
- No confundir los valores primitivos booleanos **true** y **false** con los valores true y false de un objeto booleano.

#### Ejemplo:

```
var empezado = true;  
var acabado  = false;
```

## Literales.

### ■ Integer literals

Se pueden expresar en base 10 (decimal), base 16 (hexadecimal), base 8 (octal) y base 2 (binario).

- Decimal: secuencia de dígitos (0-9). No debe empezar por 0.
- Octal: secuencia de dígitos (0-7) comenzando con 0 ó 0o (0O).
- Hexadecimal: secuencia de dígitos (0-9) y letras a-f y A-F comenzando con 0x ó 0X.
- Binario: secuencia de dígitos (0-1) comenzando con 0b ó 0B.

### Ejemplo:

0, 117 and -345 (decimal, base 10)

015, 0001 and -0o77 (octal, base 8)

0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" ó base 16)

0b11, 0b0011 and -0b11 (binario, base 2)

## Literales.

### ■ Floating-point literals

Para expresar números en coma flotante. Pueden tener las siguientes partes:

- Parte entera: Número entero decimal que puede ir precedido de signo (+ ó -).
- Un punto decimal (“.”).
- Parte decimal: Número entero decimal.
- Un exponente: Se representa con una letra “e” o “E”, seguida de un número entero que puede ir precedido de signo (+ ó -).

### Ejemplo:

```
3.1415926  
-.123456789  
-3.1E+12  
.1e-23
```

## Literales.

### ■ Object literals

- Lista de cero o más pares *propiedad:valor* encerrados entre llaves (`{}`).
- Para el nombre de *propiedad* se pueden utilizar literales numéricos y cualquier String literal, incluida la cadena vacía (`' '`).
- Para acceder al valor de las propiedades se utiliza el punto (`.`), o los corchetes (`[]`) encerrando el nombre de la propiedad entre comillas (`' '`).
- Si el nombre de la propiedad no es un identificador válido sólo se puede acceder utilizando los corchetes.
- Se pueden anidar objetos.

# Literales.

## ■ Object literals

### Ejemplo:

```
var persona = {nombre:'Juan',  
               apellidos:'García López',  
               'año de nacimiento':1987,  
               '':'blanco',  
               8:'No',  
               '!+':12.5,  
               edad:getEdad(1987),  
               'dirección':{calle:'Avda. de España',  
                             'nº':12,  
                             piso:'3º A'  
                           }  
};
```

```
console.log(persona.nombre); // "Juan"  
console.log(persona['año de nacimiento']); // 1987  
console.log(persona.edad); // 30  
console.log(persona['dirección']['nº']); // 12  
console.log(persona['']); // "blanco"
```

## Literales.

### ■ Object literals

#### **Formato JSON:** JavaScript Object Notation

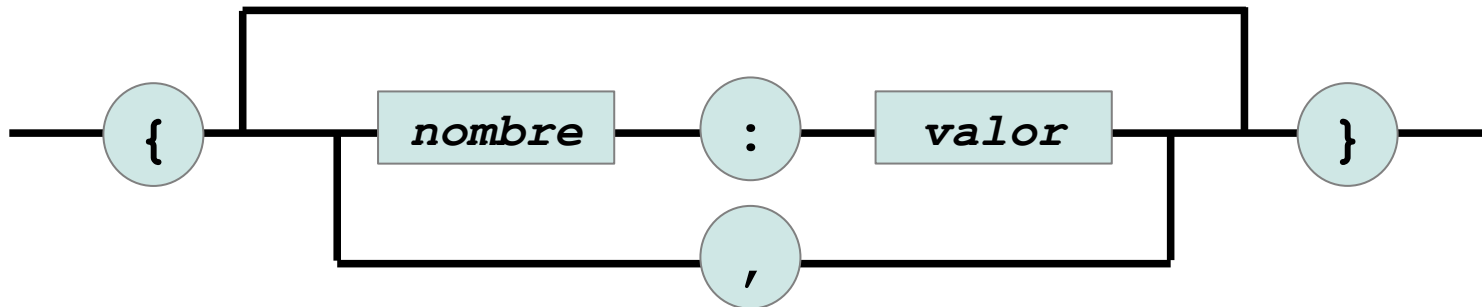
- ✓ Está basado en el formato *Object Literal*
- ✓ Se utiliza para intercambio de datos
- ✓ Es independiente del lenguaje
- ✓ Es auto-descriptivo y fácil de leer y entender
- ✓ Está constituido por dos estructuras:
  - Una colección de pares nombre/valor; conocido en algunos lenguajes como objeto, registro, estructura, diccionario, tabla hash, lista de claves, o vector asociativo.
  - Una lista ordenada de valores. En la mayoría de lenguajes, esto se implementa como vectores, listas o secuencias.

## Literales.

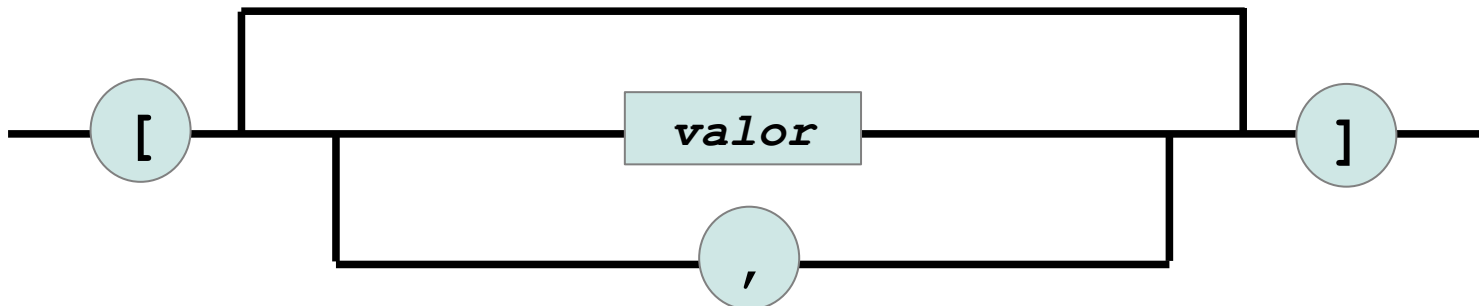
### ■ Object literals

**Formato JSON:** JavaScript Object Notation

- Objeto:



- Vector:



## Literales.

### ■ Object literals

#### Formato JSON: JavaScript Object Notation

- Objeto: (Ejemplo)

```
{ "nombre": "Juan", "apellidos": "Pérez Sánchez",  
  "dni": { "num": "21345678", "letra": "W" },  
  "dirección": { "calle": "Mayor", "número": 24, "planta": 3, "letra": "A" },  
  "población": "San Vicente del Raspeig",  
  "provincia": "Alicante", "cp": "03690",  
  "teléfonos": [ { "número": "966096120", "tipo": "casa" },  
                  { "número": "612350087", "tipo": "móvil" },  
                  { "número": "965125678", "tipo": "trabajo" } ]  
}
```

- Vector: (Ejemplo)

```
[ { "nombre": "Juan", "apellidos": "Pérez Sánchez", "dni": "21345678W" },  
  { "nombre": "Ana", "apellidos": "Gómez Carmona", "dni": "42054234A" },  
  { "nombre": "Pedro", "apellidos": "Abellán Soria", "dni": "23990123B" }  
]
```



## Literales.

### ■ Object literals

#### Formato JSON: JavaScript Object Notation

Para poder utilizar en JavaScript los datos de un texto en formato JSON, es necesario procesarlos y convertirlos en un objeto. Esto se puede hacer de dos maneras:

- Mediante la función **eval()**:

```
eval( '(' + texto_JSON + ')' )
```

#### Ejemplo:

```
var textoJSON = '{ "nombre": "Juan", "apellidos": "Pérez Sánchez",  
  "dni": { "num": "21345678", "letra": "W" }, "teléfonos": [ { "num": "966091234",  
    "tipo": "casa" }, { "num": "612347912", "tipo": "móvil" } ] }';  
  
var persona = eval( '(' + textoJSON + ')' );  
  
console.log( persona.nombre ); // Juan  
console.log( persona.dni.num + persona.dni.letra ); // 21345678W  
console.log( persona.teléfonos[1].tipo + ': ' + persona.teléfonos[1].num );  
// móvil: 612347912
```

# Literales.

## ■ Object literals

### Formato JSON: JavaScript Object Notation

- Mediante el objeto **JSON nativo**. Todos los navegadores modernos lo incorporan y es accesible mediante ***window.JSON***. Para navegadores antiguos es necesario comprobar si está soportado.
  - ✓ **JSON.parse**(*texto\_JSON*): Convierte un texto en formato JSON a objeto o valor JavaScript.

#### Ejemplo:

```
var textoJSON = '{ "nombre": "Juan", "apellidos":"Pérez Sánchez",
    "dni":{"num":"21345678","letra":"W"},"teléfonos":[{"num":"966091234",
    "tipo":"casa"}, {"num":"612347912","tipo":"móvil"}] }';
var persona;

if(window.JSON) // Comprueba si soporta JSON nativo
    persona = window.JSON.parse( textoJSON );
console.log( persona.nombre ); // Juan
console.log( persona.dni.num + persona.dni.letra ); // 21345678W
console.log( persona.teléfonos[1].tipo + ':' + persona.teléfonos[1].num );
// móvil:612347912
```

## Literales.

### ■ Object literals

#### Formato JSON: JavaScript Object Notation

- Mediante el objeto JSON nativo.
  - ✓ **JSON.stringify**(*valor*): Convierte y devuelve un valor JavaScript a una cadena de texto en formato JSON.

#### Ejemplo:

```
var objeto= { 'nombre': 'Juan', 'apellidos':'Pérez Sánchez',
              'dni':{'num':'21345678','letra':'W'},
              'telefonos':[{'num':'966091234','tipo':'casa'},
                           {'num':'612347912','tipo':'móvil'}] };

var textoJSON = window.JSON.stringify( objeto );

console.log( textoJSON);
// Resultado: '{"nombre":"Juan","apellidos":"Pérez
Sánchez","dni":{"num":"21345678","letra":"W"},"telefonos":[{"num":"966091234",
"tipo":"casa"}, {"num":"612347912","tipo":"móvil"}]}'
```

## Literales.

### ■ RegExp literals

- Son patrones encerrados entre barras (//)
- Se utilizan para buscar combinaciones de caracteres en strings.
- Una explicación de cómo construir expresiones regulares se puede encontrar en:

[https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions)

### Ejemplo:

```
var re = /ab+c/;  
var re2 = /^[A-Za-z][0-9]{2,3}$/;
```

# Literales.

## ■ String literals

- Está formado por cero o más caracteres encerrados en comillas dobles (") o simples (').
- Debe estar delimitado por comillas del mismo tipo.
- Se pueden utilizar caracteres especiales:

\n	Salto de línea
\r	Retorno de carro
\t	Tabulador horizontal
\uXXXX	Caracteres Unicode expresados los dígitos hexadecimales XXXX

```
console.log('primera línea \n segunda línea');  
console.log('\u00A9 \t 10');
```

## Literales.

### ■ String literals

- Se pueden *escapar* caracteres especiales utilizando la contrabarra (\):

```
console.log('Así se \'escapan\' las comillas');  
// resultado: Así se 'escapan' las comillas  
  
// lo mismo se hubiera conseguido sin escapar:  
console.log("Así se 'escapan' las comillas");  
  
// escapando la contrabarra:  
console.log('Path fichero: C:\\temp\\');
```

## Literales.

### ■ String literals

#### Template literals

- Un *template literal* es un nuevo tipo de string literal que puede ser multilínea e interpolar expresiones.
- Un *template literal* se escribe encerrándolo entre *backticks* (```).
- Las expresiones a interpolar dentro del *template literal* van entre `${` y `}`.
- Un *template literal* siempre produce como resultado strings.

## Literales.

### ■ String literals

#### Template literals

Ejemplo:

```
const nombre = 'María';  
console.log(`Hola ${nombre}!  
¿Cómo estás  
hoy?` );  
  
// Resultado:  
// Hola María!  
// ¿Cómo estás  
// hoy?
```



# Objetos

## Objetos

Es un tipo de variable mejorada que permite organizar el código de forma más clara mediante la encapsulación de **propiedades** y **métodos**.

- Propiedades: valores asociados al objeto. Para acceder a una propiedad de un objeto:

*nombre\_del\_objeto.nombre\_de\_la\_propiedad*

- Métodos: acciones que se pueden realizar sobre el objeto. Para invocar un método de un objeto:

*nombre\_del\_objeto.nombre\_del\_método*

**Casi todo en JavaScript es un objeto:**  
***String, Date, Number, Array, Function, ...***

# Objetos

## Ejemplo:

```
// Se define una variable mensaje con el valor "¡¡Hola Mundo!!"  
// La variable mensaje es, a su vez, un objeto de tipo String.  
var mensaje = '¡¡Hola Mundo!!';  
  
// Uso de la propiedad length del objeto mensaje  
var x = mensaje.length;  
  
console.log('El valor de x es: ' + x);  
// Resultado: "El valor de x es: 14"  
  
// Uso del método toUpperCase() del objeto mensaje  
var mensajeMAY = mensaje.toUpperCase();  
  
console.log('El mensaje en mayúsculas es: ' + mensajeMAY);  
// Resultado: "El mensaje en mayúsculas es: ¡¡HOLA MUNDO!!"
```

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

Al comenzar la ejecución de código JavaScript se dispone una serie de objetos predefinidos.

### ■ **Objeto global** (*global object*)

Es el primer objeto disponible. Es único y se crea antes de que el programa entre al contexto de ejecución.

#### Algunas propiedades y métodos útiles

- **Propiedades:**

- **Infinity**. Representa el valor  $+\infty$ .
- **NaN**. Representa un valor no numérico (**N**ot **a** **N**umber).
- **undefined**. Representa un valor no definido.
- **JSON**. Representa el objeto JSON.
- **Math**. Representa el objeto Math.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Objeto global (*global object*)

#### Algunas propiedades y métodos útiles

- Métodos:

- **eval**(*x*). Recibe una expresión como argumento y la evalúa devolviendo el resultado.
- **isFinite**(*número*). Devuelve **true** si *número* es finito, o **false** en caso contrario.
- **isNaN**(*x*). Devuelve **true** si *x* no es un número, o **false** en caso contrario.
- **parseFloat**(*texto*). Devuelve el valor tipo Number equivalente a *texto*, si éste es un número, o NaN en caso contrario.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Objeto global (*global object*)

#### Algunas propiedades y métodos útiles

- Métodos (*y III*):
  - **parseInt**(*texto*, *base*). Devuelve el valor tipo Number equivalente a *texto* en la *base* especificada, si éste es un número, o NaN en caso contrario. Los valores de *base* pueden ser 8 para octal, 10 para decimal, 16 para hexadecimal, etc. Si no se proporciona el argumento *base*, o es 0, se asume que:
    - ✓ si *texto* empieza por 0x, se convierte a hexadecimal;
    - ✓ si *texto* empieza por 0, se convierte a octal;
    - ✓ en cualquier otro caso, se convierte a decimal.

Se recomienda especificar la *base* al utilizar este método para producir el mismo resultado en todos los navegadores

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Objeto global (*global object*)

#### Algunas propiedades y métodos útiles

- Métodos (*y IV*):

- Manejo de direcciones URL (URI)

Los identificadores uniformes de recursos (Uniform Resource Identifiers, URI) son cadenas de texto que identifican recursos como páginas web o ficheros.

- ✓ **encodeURIComponent**(*uri*). Devuelve una nueva versión del texto *uri*, codificado en UTF-16, en la que se han añadido las secuencias de escape necesarias para los caracteres especiales, excepto: , / ? : @ & = + \$ #
- ✓ **encodeURIComponent**(*componenteURI*). Hace lo mismo que encodeURIComponent() pero añade secuencias de escape para todos los caracteres especiales.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Objeto global (*global object*)

#### Algunas propiedades y métodos útiles

- Métodos (y V):
  - Funciones de manejo de direcciones URL (URI)
    - ✓ **decodeURI**(*uriCodificado*). Devuelve una nueva versión del texto *uriCodificado*, en la que se han eliminado las secuencias de escape añadidas.
    - ✓ **decodeURIComponent**(*componenteURICodificado*). Hace lo mismo que `decodeURI()` pero sobre un texto que representa un componente URI codificado.



# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Objetos fundamentales

Son los objetos básicos en JavaScript sobre los que se construyen el resto de objetos.

- **Object**(*[valor]*). Es el constructor de objetos intrínseco (su uso se verá más adelante).
- **Function**(*a1, a2, ..., an, cuerpo*). Es el constructor para funciones. *a1 - an* son los argumentos y *cuerpo* es el código ejecutable de la función.

```
var sumar = new Function('a','b','return a + b;');
```

- **Boolean**(*valor*). Es el constructor de objetos Boolean. El *valor* que se le pasa se convierte a booleano: **true** o **false**. No hay que confundir con los valores primitivos booleanos **true** y **false**.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Objetos fundamentales

- **Symbol**(*[descripción]*). Nuevo en ECMAScript 6. Permite crear datos únicos anónimos. No se pueden crear símbolos con **new**.
- **Error**(*mensaje*). Permite crear nuevos objetos de tipo Error con *mensaje* como descripción asociada.

```
try{  
    throw = new Error('Se ha producido un error chungo!!!');  
} catch (e) {  
    console.log(e.name + ': ' + e.message);  
}
```

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

Permiten crear objetos de tipo número y fecha en JavaScript.

- **Number**(*[valor]*). Crea un objeto numérico con formato en coma flotante de doble precisión (64 bits) con el valor que se le pasa. Si no se pasa *valor*, el objeto Number se inicializa a 0.

Propiedades:

- ✓ **EPSILON**. Devuelve la diferencia entre 1 y el valor más pequeño mayor que 1 que se puede representar como Number.
- ✓ **MAX\_VALUE**. El número positivo más grande que se puede representar.
- ✓ **MIN\_VALUE**. El número positivo más pequeño que se puede representar, es decir, el número positivo más cercano a 0 (pero sin ser 0).
- ✓ **MAX\_SAFE\_INTEGER**. Máximo entero seguro en JavaScript:  $(2^{53}-1)$ .
- ✓ **MIN\_SAFE\_INTEGER**. Mínimo entero seguro en JavaScript:  $-(2^{53}-1)$ .
- ✓ **NaN**. Valor especial de no es número (**Not a Number**).
- ✓ **NEGATIVE\_INFINITY**. Infinito negativo. Valor devuelto cuando hay *overflow*.
- ✓ **POSITIVE\_INFINITY**. Infinito. Valor devuelto cuando hay *overflow*.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- **Number**([*valor*]).

Métodos:

- ✓ **isNaN**(*valor*). Devuelve true si el *valor* pasado no es un número. false en caso contrario.
- ✓ **isFinite**(*valor*). Devuelve true si el *valor* pasado es un número finito. false en caso contrario.
- ✓ **isInteger**(*valor*). Devuelve true si el *valor* pasado es un número entero. false en caso contrario.
- ✓ **isSafeInteger**(*valor*). Devuelve true si el *valor* pasado es un número entero seguro (entre  $-(2^{53}-1)$  y  $(2^{53}-1)$ ). false en caso contrario.
- ✓ **parseFloat**(*valor*). Mismo método que `parseFloat()` del objeto global.
- ✓ **parseInt**(*valor*). Mismo método que `parseInt()` del objeto global.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- **Number**(*[valor]*).

Métodos (y II):

- ✓ **toExponential**(*[núm\_decimales]*). Devuelve un String que representa el valor numérico en notación exponencial en base 10. *núm\_decimales* permite indicar el número de decimales a mostrar (por defecto 0).
- ✓ **toFixed**(*[núm\_decimales]*). Devuelve un String que representa el valor numérico en notación decimal de punto fijo. *núm\_decimales* permite indicar el número de decimales a mostrar (por defecto 0).
- ✓ **toLocaleString**( ). Devuelve un String que representa el valor numérico según la especificación ECMA-402, en función de la ubicación del host.
- ✓ **toString**(*[base]*). Devuelve un String que representa el valor numérico en notación exponencial en la *base* indicada. Los valores permitidos van de 2 a 36, siendo el valor por defecto 10.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- **Number**(*[valor]*).

Métodos (y III):

- ✓ **toPrecision**(*[precisión]*). Devuelve un String que representa el valor numérico en notación exponencial en base 10 utilizando tantos dígitos significativos como indique *precisión*.
- ✓ **valueOf**(). Devuelve el valor del objeto Number.

- **Math**. Proporciona propiedades y métodos para funciones y constantes matemáticas.

Propiedades:

- ✓ **E**. Valor del número *e*, aproximadamente 2.7182818284590452354
- ✓ **LN10**. Logaritmo natural de 10: aprox. 2.302585092994046
- ✓ **LN2**. Logaritmo natural de 2: aprox. 0.6931471805599453

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- Math.

Propiedades (y //).

- ✓ LOG10E. Logaritmo en base 10 del número  $e$ , aprox. 0.4342944819032518
- ✓ LN2E. Logaritmo en base 2 del número  $e$ , aprox. 1.4426950408889634
- ✓ PI. Número  $\pi$ : aprox. 3.1415926535897932
- ✓ SQRT1\_2. Valor de la raíz cuadrada de  $\frac{1}{2}$ , aprox. 0.7071067811865476
- ✓ SQRT2. Valor de la raíz cuadrada de 2, aprox. 1.4142135623730951

Métodos. Algunos de los métodos más utilizados son los siguientes:

- ✓ abs(valor). Devuelve el valor absoluto de *valor*.
- ✓ acos(valor). Devuelve el arcocoseno de *valor*.
- ✓ asin(valor). Devuelve el arcoseno de *valor*.
- ✓ atan(valor). Devuelve el arcotangente de *valor*.
- ✓ cbrt(valor). Devuelve la raíz cúbica de *valor*.
- ✓ ceil(valor). Devuelve el entero inmediatamente superior o igual a *valor*.
- ✓ cos(valor). Devuelve el coseno de *valor*.
- ✓ exp(valor). Devuelve el número  $e$  elevado a *valor* ( $e^{\text{valor}}$ ).

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- Math.

Métodos (y //).

- ✓ `floor(valor)`. Devuelve el entero inmediatamente anterior o igual a *valor*.
- ✓ `hypot([x[, y[, ...]]])`. Devuelve la raíz cuadrada de la suma de los cuadrados de los argumentos.
- ✓ `log(valor)`. Devuelve el logaritmo natural (en base *e*) de *valor*.
- ✓ `log10(valor)`. Devuelve el logaritmo en base 10 de *valor*.
- ✓ `log2(valor)`. Devuelve el logaritmo en base 2 de *valor*.
- ✓ `max([x[, y[, ...]]])`. Devuelve el máximo de los argumentos que se le pasan.
- ✓ `min([x[, y[, ...]]])`. Devuelve el mínimo de los argumentos que se le pasan.
- ✓ `pow(x, y)`. Devuelve el valor del número  $x^y$ .
- ✓ `random()`. Devuelve un número pseudo-aleatorio en el rango  $[0, 1)$ .
- ✓ `round(valor)`. Devuelve el entero más cercano a *valor*.
- ✓ `sqrt(valor)`. Devuelve la raíz cuadrada de *valor*.
- ✓ `tan(valor)`. Devuelve la tangente de *valor*.
- ✓ `trunc(valor)`. Devuelve la parte entera de *valor*.



# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- **Date**(*año*, *mes* [, *día* [, *hora* [, *minuto* [, *segundo* [, *milisegundo* ] ] ] ]]). Permite crear e inicializar un objeto **Date** al valor que se le indica con los argumentos. *mes* empieza en 0.

**Date**(*value*). Permite crear una fecha en base al número de milisegundos que se le pasa como argumento. El número de milisegundos se mide desde el 1/01/1970 00:00:00 UTC.

**Date**(*fechaComoString*). Permite crear una fecha en base a la fecha proporcionada como **String** y que debe estar en un formato reconocido.

Formato reconocido: **AAAA-MM-DDT**HH:mm:ss.ms**+HH:mm**

**AAAA**: año; **MM**:mes; **DD**:día; **T**: separador de fecha y hora (opcional); **HH**: hora; **mm**: minuto; **ss**:segundo; **ms**:milisegundo; **+HH:mm**: huso horario expresado en horas y minutos, que puede ser positivo (+) o negativo (-)

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

- **Date()**

Se puede utilizar de **Date()** como función (no como objeto) sin pasarle argumentos para obtener un **String** que representa la fecha/hora actual.

#### Métodos.

Algunos de los métodos más utilizados:

- ✓ **now()**. Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970 a las 00:00:00 UTC.
- ✓ **parse(*fechaComoString*)**. Devuelve el número de milisegundos de la fecha que se le pasa como argumento (en el formato reconocido) desde el 1 de enero de 1970 a las 00:00:00 UTC.
- ✓ **UTC(*año*, *mes* [, *día* [, *hora* [, *minuto* [, *segundo* [, *milisegundo* ]]]])**. Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970 a las 00:00:00 UTC.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Números y fechas

#### ○ Date

##### Métodos (y //)

- ✓ `getTime()/setTime()`. Devuelve/establece el número de milisegundos transcurridos desde el 1 de enero de 1970.
- ✓ `getDate()/setDate()`. Devuelve/establece el día del mes (1 - 31).
- ✓ `getMonth()/setMonth()`. Devuelve/establece el mes (0 - enero, 1 - febrero, 2 - marzo, ..., 11 - diciembre).
- ✓ `getFullYear()/setFullYear()`. Devuelve/establece el año con cuatro dígitos.
- ✓ `getDay()`. Devuelve el día de la semana (0 - domingo, 1 - lunes, ...).
- ✓ `getHours()/setHours()`. Devuelve/establece la hora (00 - 23).
- ✓ `getMinutes()/setMinutes()`. Devuelve/establece los minutos (00 - 59).
- ✓ `getSeconds()/setSeconds()`. Devuelve/establece los segundos (00 - 59).
- ✓ `toJSON()`. Devuelve la fecha en formato: `aaaa-mm-ddThh:mm:ssZ`

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **String**(*valor*). Es el constructor para *strings* o secuencias de caracteres.

#### Propiedades.

- ✓ **length**. Devuelve la cantidad de caracteres en el texto del objeto String.

#### Métodos.

- ✓ **fromCharCode**(*num1*[, ...[, *numN*]]). Devuelve un string que contiene los caracteres correspondientes a la secuencia de valores Unicode que se le pasan como argumento.
- ✓ **fromCodePoint**(*num1*[, ...[, *numN*]]). Funciona igual que **fromCharCode**() con la diferencia de que los parámetros pueden ser códigos de 4 bytes en lugar de 2.
- ✓ **charAt**(*pos*). Devuelve el carácter que se encuentra en la posición indicada por *pos*. La primera posición es 0.
- ✓ **charCodeAt**(*pos*). Devuelve un entero no negativo que representa el código del elemento de la cadena que se encuentra en la posición indicada .

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **String**(*valor*).

Métodos (y II).

- ✓ **codePointAt**(*pos*). Igual que **charCodeAt**() pero el entero es de 4 bytes.
- ✓ **concat**(*string2*[, *string3*, ..., *stringN*]). Combina el texto de uno o más strings y devuelve un nuevo string como resultado.
- ✓ **endsWith**(*cadenaABuscar*[, *pos*]). Devuelve true si *cadenaABuscar* coincide con los elementos del objeto String a partir de la posición *pos* menos la longitud de *cadenaABuscar*. El valor por defecto de *pos* es la longitud del texto del objeto String.
- ✓ **includes**(*cadenaABuscar*[, *pos*]). Devuelve true si *cadenaABuscar* aparece una o más veces como subcadena del texto del Objeto String a partir de la posición *pos*. El valor por defecto de *pos* es 0.
- ✓ **indexOf**(*cadenaABuscar*[, *pos*]). Devuelve la primera posición mayor o igual a *pos* en la que aparece *cadenaABuscar* dentro del texto del objeto String. Si no la encuentra devuelve -1. Valor por defecto de *pos*: 0.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **String**(*valor*).

Métodos (y III).

- ✓ **lastIndexOf**(*cadenaABuscar*[, *pos*]). Devuelve la última posición menor o igual a *pos* en la que aparece *cadenaABuscar* dentro del texto del objeto String. Si no la encuentra devuelve -1. En este caso el valor por defecto de *pos* es la longitud del texto del objeto String.
- ✓ **localCompare**(*cadenaAComparar*[, *locales*[, *opciones*]). Devuelve -1, 0, ó 1, si el texto del objeto String es anterior, igual o posterior, respectivamente, a *cadenaAComparar* teniendo en cuenta el orden alfabético y las opciones de comparación.
- ✓ **match**(*exprRegular*). Devuelve un vector con las subcadenas de texto, dentro del texto del objeto String, que coinciden con el patrón indicado por la expresión regular *exprRegular* (más adelante se verá cómo construir expresiones regulares).

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **String**(*valor*).

Métodos (y IV).

- ✓ **padEnd**(*máxLongitud*[, *textoRelleno*]). Devuelve un nuevo objeto String con el texto de relleno indicado, añadido al final del texto del objeto String hasta la longitud máxima indicada.
- ✓ **padStart**(*máxLongitud*[, *textoRelleno*]). Devuelve un nuevo objeto String con el texto de relleno indicado, añadido al inicio del texto del objeto String hasta la longitud máxima indicada.
- ✓ **repeat**(*número*). Devuelve un nuevo objeto String cuyo texto es el texto del objeto String repetido *número* veces.
- ✓ **replace**(*textoABuscar*|*expReg*, *nuevoTexto*|*función*). Devuelve un nuevo objeto String con alguna o todas las ocurrencias de *textoABuscar*, en el texto del objeto String, sustituidas por *nuevoTexto* o por el resultado de la función aplicada a cada ocurrencia de *textoABuscar*. Para sustituir todas las ocurrencias es necesario utilizar una expresión regular como primer argumento.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **String**(*valor*).

Métodos (y V).

- ✓ **replaceAll**(*textoASustituir*, *nuevoTexto*). Devuelve un nuevo objeto String con todas las ocurrencias de *textoASustituir*, en el texto del objeto String, sustituidas por *nuevoTexto*.
- ✓ **search**(*expReg*). Devuelve la posición de la primera subcadena del texto del objeto String que coincide con la expresión regular *expReg*.
- ✓ **slice**(*inicio*[, *fin*]). Devuelve la subcadena del texto del objeto String que empieza en la posición *inicio*. Si se proporciona el parámetro *fin*, la subcadena devuelta es la que se encuentra entre las posiciones *inicio* y *fin* - 1. La posición del primer carácter es 0. Si *fin* es negativo la posición final será la longitud del texto del String menos el valor absoluto de *fin*.
- ✓ **split**(*separador*[, *límite*]). Devuelve un array de strings, resultado de separar el texto del objeto String por el separador indicado. *límite* permite indicar el número máximo de divisiones a realizar.



# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **String**(*valor*).

Métodos (y VI).

- ✓ **startsWith**(*cadenaABuscar*[, *pos*]). Devuelve true si el texto del objeto String empieza con *cadenaABuscar*. *posición* permite indicar la posición inicial en la que empezar la comparación (por defecto 0).
- ✓ **substring**(*inicio*[, *fin*]). Funciona igual que **slice**() con la única diferencia de que no se pueden indicar valores negativos para *fin*.
- ✓ **toLowerCase**(). Devuelve el texto del objeto String en minúsculas.
- ✓ **toUpperCase**(). Devuelve el texto del objeto String en mayúsculas.
- ✓ **trim**(), **trimLeft**(), **trimRight**(). Devuelve el texto del objeto String, en el que se han eliminado los espacios en blanco a derecha e izquierda; a izquierda; o a derecha, respectivamente.
- ✓ **valueOf**(). Devuelve el texto del objeto String como una cadena de texto.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **RegExp**(*patrón*[, *flags*]). Constructor de objetos que representan expresiones regulares. Los argumentos tienen el siguiente significado:
  - *patrón*. Texto para especificar la expresión regular.
  - *flags*. Los modificadores de la expresión regular. Por ejemplo: **g** indica búsqueda global e **i** indica no distinguir entre mayúsculas y minúsculas.

#### Propiedades.

- ✓ **flags**. Devuelve una cadena que representa la configuración de los modificadores para el objeto RegExp.
- ✓ **global**. Devuelve true si el flag de búsqueda global (**g**) está aplicado al objeto RegExp. Devuelve false en caso contrario.
- ✓ **ignoreCase**. Devuelve true si el flag para ignorar mayúsculas/minúsculas (**i**) está aplicado al objeto RegExp. Devuelve false en caso contrario.
- ✓ **lastIndex**. Devuelve/establece la posición en la que se empezará la siguiente búsqueda por el patrón.
- ✓ **source**. Devuelve el patrón de la expresión regular.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Procesamiento de texto

- **RegExp**(*patrón*[, *flags*]).

#### Métodos.

- ✓ **exec**(*cadena*). Ejecuta una búsqueda para un patrón en la cadena de texto que se le pasa a partir de la posición indicada por `lastIndex`. Devuelve un array con las subcadenas coincidentes.
- ✓ **test**(*cadena*). Devuelve `true` si en *cadena* hay una subcadena que coincida con el patrón a partir de la posición indicada por `lastIndex`. `false` en caso contrario.
- ✓ **toString**(). Devuelve una cadena de texto que representa la expresión regular y los flags del objeto `RegExp`.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. El objeto Array es un objeto global que se utiliza para la construcción de arrays.

#### Creación de arrays

Se pueden crear de dos formas:

- Primera forma: Mediante el constructor.

```
// crear un array vacío  
var a = new Array();  
// crear un array vacío especificando el número de elementos  
var a = new Array(número_de_elementos);  
// crear un array inicializándolo con elementos  
var a = new Array(elem1,elem2[,...,elemN]);
```

- Segunda forma:

```
var a = [elem1,elem2[,...,elemN]];
```

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array.**

#### Creación de arrays

La forma más adecuada y recomendable es la segunda, puesto que la primera forma podría experimentar problemas de ambigüedad, tal y como se puede ver en el siguiente ejemplo:

```
function Array(){
    this.is = 'SPARTA';
}

var a = new Array(); // primera forma de crear arrays
var b = []; // segunda forma de crear arrays

alert(a.is); // 'SPARTA'
alert(b.is); // undefined

a.push('Woa'); // TypeError: push no es un método de a
b.push('Woa'); // 1 (OK)
```

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- Array.

#### Acceso a los elementos

A los elementos de un array se accede utilizando corchetes e indicando el índice (posición de elemento en el array, siendo 0 la del primero) del elemento a acceder: [índice].

Ejemplo:

```
var diasSemana = ['Lunes', 'Martes', 'Miércoles', 'Jueves',  
                  'Viernes', 'Sábado', 'Domingo'];  
  
console.log(diasSemana[2]); // Miércoles  
console.log(diasSemana[0]); // Lunes  
console.log(diasSemana[diasSemana.length - 1]); // Domingo
```

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array.**

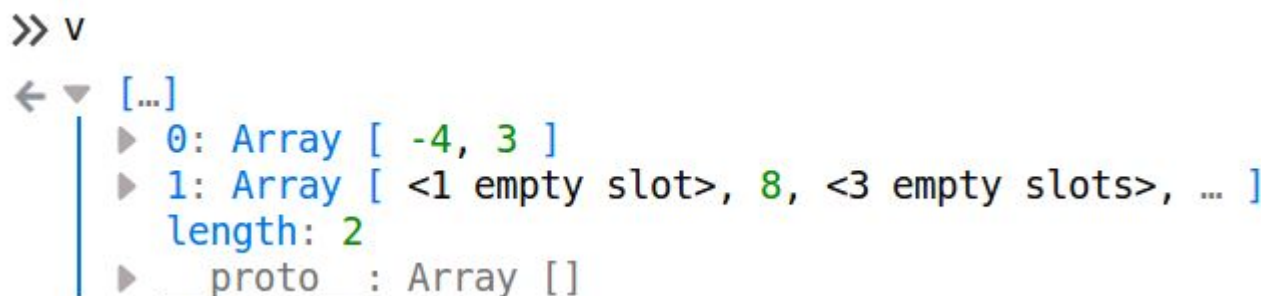
#### Arrays multidimensionales

JavaScript permite crear arrays multidimensionales, que no son más que arrays de arrays.

Ejemplo:

```
var v = [ [], [] ]; // v es un array bidimensional de 2 filas,  
                  // siendo cada fila, a su vez, un array
```

```
v[0][0] = -4;  
v[0][1] = 3;  
v[1][1] = 8;  
v[1][5] = 5;  
console.log(v);
```



```
>>> v  
← [...]   
  0: Array [-4, 3]  
  1: Array [ <1 empty slot>, 8, <3 empty slots>, ... ]  
    length: 2  
  __proto__: Array []
```

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. Algunas propiedades y métodos útiles

#### Propiedades.

- ✓ **length**. Devuelve el número de elementos del array.

#### Métodos.

- ✓ **from**(*objeto*). Devuelve un nuevo array a partir de un *objeto* que sea tipo array o iterable.
- ✓ **isArray**(*objeto*). Devuelve true si *objeto* es de tipo array.
- ✓ **of**(*elem0*[, *elem1*[, ...[, *elemN*]]]). Devuelve un nuevo array creado con los elementos pasados como argumento.
- ✓ **concat**(*valor0*[, *valor1*[, ...[, *valorN*]]]). Devuelve un nuevo array en el que une el actual con los arrays/elementos pasados como argumento.
- ✓ **copyWithin**(*posDestino*[, *posInicial*[, *posFinal*]]]). Copia los elementos del array que se encuentran entre las posiciones [*posInicial*, *posFinal*-1] a la posición indicada por *posDestino*, sobrescribiendo los elementos y no aumentando la longitud del array. El valor por defecto de *posInicial* es 0 y el de *posFinal* es la longitud del array.



# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. Algunas propiedades y métodos útiles

Métodos (y //).

- ✓ **every**(*callback*[, *thisArg*]). Para cada elemento del array evalúa la función *callback* y si para todos ellos esta función devuelve true, *every*() devuelve true. *thisArg* es un valor que se puede pasar a la función *callback* y al que se podrá hacer referencia mediante *this* dentro de la función. La función *callback* toma tres argumentos en este orden: elemento del array, índice del elemento en el array y, por último, el propio array.
- ✓ **fill**(*valor*[, *inicio*[, *fin*]]). Rellena los elementos de un array con el *valor* que se le pasa. *inicio* y *fin* permiten indicar el rango de elementos a rellenar.
- ✓ **filter**(*callback*[, *thisArg*]). Devuelve un nuevo array con todos los elementos para los que al ser evaluados por la función *callback*, ésta devuelve true.
- ✓ **find**(*callback*[, *thisArg*]). Devuelve el primer elemento para el que la función *callback* devuelve true. En caso contrario devuelve undefined.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. Algunas propiedades y métodos útiles

Métodos (y III).

- ✓ **findIndex**(*callback[, thisArg]*). Devuelve el índice del primer elemento para el que la función *callback* devuelve true.
- ✓ **forEach**(*callback[, thisArg]*). Para cada elemento del array se ejecuta la función *callback*. *thisArg* es un valor que se puede pasar a la función *callback* y al que se podrá hacer referencia mediante *this* dentro de la función. La función *callback* toma tres argumentos en este orden: elemento del array, índice del elemento en el array y, por último, el propio array.
- ✓ **includes**(*elementoABuscar[, posInicial]*). Devuelve true si encuentra *elementoABuscar* a partir de la posición *posInicial*, si ésta se indica.
- ✓ **indexOf**(*elementoABuscar[, posInicial]*). Devuelve la posición en la que se encuentra *elementoABuscar* a partir de la posición *posInicial*, si ésta se indica. Si no lo encuentra devuelve -1.
- ✓ **join**(*[separador]*). Devuelve todos los elementos del array unidos en un string. Se puede especificar un separador de tipo string para cada par de elementos del array. Si no se especifica separador se utiliza la coma (",").

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. Algunas propiedades y métodos útiles

Métodos (y IV).

- ✓ `lastIndexOf(elementoABuscar[, posInicial])`. Devuelve la última posición en la que se encuentra *elementoABuscar* a partir de la posición *posInicial*, si ésta se indica. La búsqueda se hace de atrás hacia delante. Si no lo encuentra devuelve -1.
- ✓ `map(callback[, thisArg])`. Devuelve un nuevo array cuyos elementos son los resultados de aplicar la función a cada uno de los elementos originales del array.
- ✓ `pop()`. Devuelve y elimina del array el último elemento del mismo.
- ✓ `push(elem0[, elem1[, ...[, elemN]]])`. Añade uno o más elementos al final del array y devuelve la nueva longitud del array.
- ✓ `reduce(callback[, valorInicial])`. Aplica la función *callback* a un acumulador y a cada elemento del array, de izquierda a derecha, devolviendo el resultado. Se puede asignar un valor inicial distinto de 0 al acumulador mediante *valorInicial*. *callback* toma cuatro argumentos: valor del acumulador, valor actual, índice del elemento y el propio array.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. Algunas propiedades y métodos útiles

Métodos (y V).

- ✓ **reduceRight**(*callback*[, *valorInicial*]). Aplica la función *callback* a un acumulador y a cada elemento del array, de derecha a izquierda, devolviendo el resultado. Se puede asignar un valor inicial distinto de 0 al acumulador mediante *valorInicial*. *callback* toma cuatro argumentos: valor del acumulador, valor actual, índice del elemento y el propio array.
- ✓ **reverse**(). Invierte el orden de los elementos en el array: el primer elemento pasa a ser el último y el último el primero, etc.
- ✓ **shift**(). Devuelve y elimina el primer elemento del array.
- ✓ **slice**(*inicio*[, *fin*]). Devuelve un array que contiene los elementos del array original que se encuentran en las posiciones desde *inicio* a *fin-1*, si se especifica *fin*. Si no se especifica *fin*, va hasta el final.
- ✓ **some**(*callback*[, *thisArg*]). Devuelve true si al menos para uno de los elementos del array la función *callback* devuelve true. La función *callback* toma tres argumentos: el elemento, el índice del elemento y el propio array.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones indexadas

- **Array**. Algunas propiedades y métodos útiles

Métodos (y VI).

- ✓ **sort**(*[funciónComparación]*). Ordena el vector. Si no se pasa función de ordenación, ordena el elemento de menor a mayor. Si se pasa función de ordenación, esta función recibe dos argumentos a, b. Para que a esté delante de b, la función debe devolver un número negativo o cero. Para que b esté delante de a, la función tiene que devolver un valor positivo.
- ✓ **splice**(*posInicial*[, *cantidadABorrar*, [, *elem01*[, ...]]]). Cambia el contenido del array eliminando a partir de *posInicial* la cantidad indicada (*cantidadABorrar*) de elementos existentes y/o añade los nuevos (*elem01*, ...) que se le pasan.
- ✓ **toString**(). Devuelve un string formado por los elementos del array separados por coma (",").
- ✓ **unshift**(*[elem01*[, *elem02*[, ...]]]). Añade uno o más elementos al inicio del array y devuelve la nueva longitud del array.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones con clave

- **Map**. Son colecciones de pares clave/valor en las que el valor de la clave no se puede repetir. Los objetos Map se crean de la siguiente manera:

```
var a = new Map([iterable]);
```

El argumento opcional que se le puede pasar es un objeto iterable como, por ejemplo, un array.

Ejemplo:

```
var a = new Map( [ [ 'n', 5 ], [ 'id', 'AB0012' ] ] );
```

Propiedades.

- ✓ **size**. Devuelve el número de pares clave/valor del objeto Map.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones con clave

- Map.

- Métodos.

- ✓ `clear()`. Elimina todos los pares clave/valor del objeto Map.
- ✓ `delete(clave)`. Elimina el par con clave indicada del objeto Map.
- ✓ `forEach(callback[, thisArg])`. Ejecuta la función *callback* para cada par clave/valor del objeto Map. La función *callback* toma tres argumentos: el valor, la clave y el propio objeto Map.
- ✓ `get(clave)`. Devuelve el valor del par clave/valor con la clave indicada.
- ✓ `has(clave)`. Devuelve true si el objeto Map tiene un par clave/valor con la clave indicada. false en caso contrario.
- ✓ `set(clave, valor)`. Añade un nuevo par clave/valor al objeto Map, o modifica el valor del par con la clave indicada si ya existiera. Devuelve el objeto Map resultante.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones con clave

- **Set**. Son colecciones de valores únicos de cualquier tipo. Los objetos Set se crean de la siguiente manera:

```
var a = new Set([iterable]);
```

El argumento opcional que se le puede pasar es un objeto iterable como, por ejemplo, un array.

Ejemplo:

```
var a = new Set( [ 'Juan', 'Ana', 'José' ] );
```

Propiedades.

- ✓ **size**. Devuelve el número de valores almacenados en el objeto Set.



# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Colecciones con clave

- Set.

- Métodos.

- ✓ `add(valor)`. Añade un nuevo elemento con el valor indicado al objeto Set. Devuelve el objeto Set resultante.
  - ✓ `clear()`. Elimina todos los elementos del objeto Set.
  - ✓ `delete(valor)`. Elimina el elemento con el valor indicado.
  - ✓ `forEach(callback[, thisArg])`. Ejecuta la función *callback* para cada elemento del objeto Set. La función *callback* toma tres argumentos: el valor, el valor y el propio objeto Set.
  - ✓ `has(valor)`. Devuelve true si el objeto Set tiene un elemento con el valor indicado. false en caso contrario.

# Objetos

## Objetos predefinidos (*built-in objects*) en JavaScript

### ■ Datos estructurados

- **JSON**. Permite trabajar con texto en formato JSON (**J**ava**S**cript **O**bject **N**otation).

Métodos.

- ✓ `parse(texto)`. Recibe un texto en formato JSON y construye y devuelve el objeto o valor JavaScript descrito por el texto.
- ✓ `stringify(valor)`. Convierte y devuelve un valor JavaScript a una cadena de texto en formato JSON.

## Objetos

# Cómo crear un objeto en JavaScript

# Objetos

## Cómo crear un objeto JavaScript

En JavaScript existen tres posibilidades para crear un objeto:

- 1ª Forma: definiendo y creando una instancia directa de un objeto. Se utiliza la instrucción **new Object()** y se añaden las propiedades y los métodos mediante lo que se conoce como *notación de puntos*.

Ejemplo:

```
// Se utiliza la instrucción new Object() para crear el nuevo objeto:
var persona = new Object()
// Se utiliza la notación de puntos para añadir propiedades y métodos:
persona.nombre = "Juan"; // nueva propiedad
persona.dni = "12345678A"; // nueva propiedad
persona.fNac = "21-04-1992"; // nueva propiedad
persona.calcularEdad = function(){ // nuevo método: devuelve la edad
    var vFecha = this.fNac.split("-");
    var f = new Date(parseInt(vFecha[2],10), parseInt(vFecha[1],10)-1,
        parseInt(vFecha[0],10));
    return ((new Date()).getTime()-f.getTime()) / (365*24*60*60*1000);
}
```

# Objetos

## Cómo crear un objeto JavaScript

- 2ª Forma: utilizar ***Object literals***. En este caso, la palabra reservada **this** hace referencia al objeto que se está creando.

Ejemplo:

```
// Definición del objeto persona del ejemplo anterior:
var persona = {
  nombre : 'Juan',
  apellidos : 'Sin Miedo',
  dni : '12345678A',
  fNac : '21-04-1992',
  calcularEdad : function(){ // Devuelve los años que tiene
    var vFecha = this.fNac.split('-');
    var f = new Date(parseInt(vFecha[2]), parseInt(vFecha[1])-1,
                      parseInt(vFecha[0]));
    return ((new Date()).getTime()-f.getTime())/(365*24*60*60*1000);
  }
}; // Fin de la definición y creación del objeto
```

# Objetos

## Cómo crear un objeto JavaScript

- 2ª Forma (y //): **Object literals**. Los métodos de los objetos también se pueden declarar como propiedades.

Ejemplo:

```
// Definición del objeto persona del ejemplo anterior:
var persona = {

    ...

    calcularEdad(){ // Devuelve los años que tiene
        var vFecha = this.fNac.split('-');
        var f = new Date(parseInt(vFecha[2]), parseInt(vFecha[1])-1,
                           parseInt(vFecha[0]));
        return ((new Date()).getTime()-f.getTime())/(365*24*60*60*1000);
    }
}; // Fin de la definición y creación del objeto
```

# Objetos

## Cómo crear un objeto JavaScript

- 3ª Forma: usando una función definida por el usuario. Se crea una función con la **definición del objeto**. Esta función alberga las propiedades y métodos.

### Ejemplo:

```
// Se utiliza una función como definición del objeto Persona
function Persona(nombre, dni, fNac){
    this.nombre = nombre; // Se definen las propiedades ...
    this.dni     = dni;
    this.fNac    = fNac;
    this.calcularEdad = function(){ // Se añaden los métodos ...
        var vFecha = this.fNac.split("-");
        var f = new Date(parseInt(vFecha[2],10),parseInt(vFecha[1],10)-1,
                           parseInt(vFecha[0],10));
        return ((new Date()).getTime() - f.getTime()) / (1000*60*60*24*365);
    }
} // Fin definición del objeto
// Se crean objetos de tipo Persona
var amigo1 = new Persona("Juan", "12345678A", "21-04-1992");
var amigo2 = new Persona("Pedro", "87654321Z", "18-10-1990");
```

# Objetos

## Cómo crear un objeto JavaScript

Se pueden **añadir**, **eliminar** y **modificar** métodos y propiedades de un **objeto ya creado**.

### Ejemplo:

```
// Utilizamos los mismos objetos del ejemplo anterior ...
var amigo1 = new Persona("Juan", "Sin Miedo", "12345678A", "21-04-1992");
var amigo2 = new Persona("Pedro", "Medario", "87654321Z", "18-10-1990");

delete amigo1.dni; // Se elimina la propiedad dni del primer objeto
amigo2.apellido = "Gómez"; // Se añade una nueva propiedad al segundo objeto
amigo1.getNombre = function(){ return this.nombre; }; // Se añade un nuevo método
amigo2.calcularEdad = function(){ return 15; }; // Se modifica un método existente
amigo1.nombre = "Antonio"; // Se modifica la propiedad nombre del primer objeto
// Resultado:
console.log( amigo1.dni ); // undefined
console.log( amigo1.apellido ); // undefined
console.log( amigo1.getNombre() ); // Antonio
console.log( amigo2.dni ); // 87654321B
console.log( amigo2.apellido ); // Gómez
console.log( amigo2.calcularEdad() ); // 15
console.log( amigo2.getNombre() ); // ERROR: amigo2.getNombre() no es una función
```



# Objetos

## Cómo crear un objeto JavaScript

Se pueden listar las propiedades de un objeto, sin necesidad de saber cuántas tiene, mediante el operador **in**.

### Ejemplo:

```
// Definición del objeto persona del ejemplo anterior:
var persona = {
  nombre : "Juan",
  dni : "12345678A",
  fNac : "21-04-1992",
  calcularEdad : function(){ // Devuelve los años que tiene
    var vFecha = this.fNac.split("-");
    var f = new Date(parseInt(vFecha[2],10), parseInt(vFecha[1],10)-1,
                      parseInt(vFecha[0],10));
    return ((new Date()).getTime()-f.getTime()) / (1000*60*60*24*365);
  }
}; // Fin de la definición y creación del objeto
// Se listan las propiedades del objeto persona, con su valor:
for(var p in persona) {
  console.log("Propiedad:valor=> " + p + ":" + persona[p]);
}
```

# Objetos

## Cómo crear un objeto JavaScript

A la hora de declarar los métodos del objeto se puede utilizar una **sintaxis** más **reducida**.

Ejemplo:

```
// Definición del objeto persona del ejemplo anterior:
var persona = {
  nombre : "Juan",
  dni : "12345678A",
  fNac : "21-04-1992",
  calcularEdad() { // Devuelve los años que tiene
    var vFecha = this.fNac.split("-"),
        f = new Date(parseInt(vFecha[2],10), parseInt(vFecha[1],10)-1,
                      parseInt(vFecha[0],10));
    return ((new Date()).getTime()-f.getTime()) / (1000*60*60*24*365);
  }
}; // Fin de la definición y creación del objeto
```

# Prototipado

## Prototipado

- Mecanismo que permite añadir nuevas propiedades y métodos a TODOS los objetos de un mismo tipo, es decir, a TODAS las instancias de una misma clase: a las ya creadas y a las que se creen posteriormente.
- Se utiliza la palabra reservada **prototype** entre el nombre de la clase y el nombre de la propiedad o método y es necesario asignar un valor por defecto.

```
nombreClase.prototype.nombrePropiedad = valor;  
nombreClase.prototype.nombreMétodo = function(){  
    ...  
};
```

# Prototipado

## Ejemplo:

```
// Se crea la CLASE Persona:
function Persona(nombre, dni, fNac)
{ // Se definen las propiedades ...
  this.nombre    = nombre;
  this.dni       = dni;
  this.fNac      = fNac;
  // Se añaden los métodos ...
  this.calcularEdad = function(){ // Devuelve los años que tiene
    var vFecha = this.fNac.split("-");
    var f = new Date(parseInt(vFecha[2]),parseInt(vFecha[1])-1,parseInt(vFecha[0]));
    return ((new Date()).getTime()-f.getTime()) / (1000*60*60*24*365);
  }
} // Fin definición de la clase
// Se crean los objetos
var amigo1 = new Persona('Juan', '12345678A', '21-04-1992');
var amigo2 = new Persona('Pedro', '87654321Z', '18-10-1990');
// Se crea una nueva propiedad para todos los objetos de la clase Persona:
Persona.prototype.telefono = '900100099'; // Todos tienen el mismo valor de telefono
amigo1.telefono = '966123456'; // Se cambia el valor de telefono para amigo1
// Se crea un nuevo método para todos los objetos de la clase Persona:
Persona.prototype.getTelefono = function(){ return this.telefono };
amigo1.getTelefono(); // Resultado: "966123456"
amigo2.getTelefono(); // Resultado: "900100099"
```

## Prototipado

- Algunos objetos nativos de JavaScript admiten **prototipado**, por lo que se puede ampliar su funcionalidad. Se trata de aquellos objetos que se pueden crear con **new**: *Image*, *String*, *Date*, *Array*, *Number*.

### Ejemplo:

```
// Código extender el objeto String con un método que escribe el texto al revés
String.prototype.alReves = function(){
    var reves = "";
    for (var i = this.length - 1; i >= 0; i--){
        reves += this.charAt(i);
    }
    return reves;
}

var cadena = "Hola mundo!!!";
document.write( "<h3>" + cadena.alReves() + "</h3>");
// Resultado: !!!odnum aloH

var cadena2 = "Otro ejemplo de cadena";
document.write( "<h3>" + cadena2.alReves() + "</h3>");
// Resultado: anedac ed olpmeje ortO
```

## Prototipado

- Declarando los métodos con ***prototype***, éstos sólo se crean una vez en memoria para todas las instancias de la clase, en lugar de crearlos para cada instancia de la misma, consiguiendo un ahorro significativo de memoria.

### Del ejemplo anterior:

```
// Se crea la definición del objeto Persona:
function Persona(nombre, apellidos, dni, fNac)
{ // Se definen las propiedades ...
    ...
} // Fin definición del objeto

// Se añaden los métodos con prototype fuera de la definición ...
Persona.prototype.calcularEdad = function(){
    // Calcula y devuelve la edad
    ...
}
```

# Control del flujo de ejecución



## Control del flujo de ejecución

- Por defecto, en JavaScript las sentencias se ejecutan de forma secuencial.
- El orden en que se ejecutan las instrucciones se llama **flujo de ejecución**.
- El flujo de ejecución se puede alterar mediante ***sentencias de control*** de dos tipos:
  - Sentencias de decisión o condicionales
    - `if ... [else ...]`
    - `switch`
  - Sentencias de bucle
    - Determinadas: `for`, `for ... in`, `for ... of`
    - Indeterminadas: `while`, `do ... while`

## Control del flujo de ejecución

- **Sentencias de decisión**. Permiten ejecutar conjuntos de instrucciones en función del cumplimiento, o no, de una condición.

- Sentencia **if ... [else ...]**

Sintaxis:

### Forma 1:

```
if ( condición ) {  
    // se cumple  
    ...  
    sentencias  
    ...  
}
```

Los siguientes valores se evalúan como falso en JavaScript: **false**, **undefined**, **null**, **0**, **NaN**, cadena vacía ('').

### Forma 2:

```
if ( condición ) {  
    // se cumple  
    ...  
    sentencias  
    ...  
} else {  
    // no se cumple  
    ...  
    sentencias  
    ...  
}
```

### Forma 3:

```
if ( condición ) {  
    ...  
    sentencias  
    ...  
} else if ( condición ) {  
    ...  
    sentencias  
    ...  
} else {  
    ...  
    sentencias  
    ...  
}
```

## Control del flujo de ejecución

### ■ Sentencias de decisión.

- Sentencia **switch**. Permite evaluar una expresión y ejecutar un conjunto de instrucciones u otro en función del resultado.

```
switch (expresión) {  
  case resultado_1:  
    sentencias_a_ejecutar  
    [break;]  
  case resultado_2:  
    sentencias_a_ejecutar  
    [break;]  
  ...  
  default:  
    sentencias_a_ejecutar  
    [break;]  
}
```

- Se ejecutará el conjunto de sentencias asociado al *resultado\_x* que coincida con el resultado producido al evaluar *expresión*.
- Si el resultado producido al evaluar *expresión* no coincide con ningún *resultado\_x*, se ejecutará el conjunto de sentencias asociado al caso *default*, siempre y cuando éste se haya añadido a la estructura *switch*.
- **break** hace que el flujo de ejecución salga de la estructura switch.

## Control del flujo de ejecución

- **Sentencias de bucle**. Permiten repetir un grupo de sentencias un número determinado o indeterminado de veces.
  - Sentencias de bucle **determinadas**. Repiten un número determinado de veces un grupo de sentencias.
  - Sentencias de bucle **indeterminadas**. Repiten un número indeterminado de veces un grupo de sentencias. El número de veces que se repiten va en función del cumplimiento de una condición.

# Control del flujo de ejecución

## ■ Sentencias de bucle.

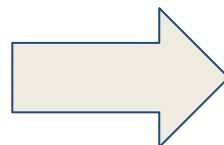
- Sentencias de bucle **determinadas.**

```
for ([expresión_inicial]; [condición]; [expresión_final]) {  
    sentencias_a_ejecutar  
}
```

### Ejemplo:

```
var i;  
for (i = 0; i < 5; i++) {  
    console.log('Número ' + i);  
}
```

```
// Resultado:  
// Número 0  
// Número 1  
// Número 2  
// Número 3  
// Número 4
```



### Formas equivalentes de escribir el bucle **for**

```
var i = 0;  
for (; i < 5; i++) {  
    console.log('Número ' + i);  
}
```

```
for ( var i = 0; i < 5; i++) {  
    console.log('Número ' + i);  
}
```

# Control del flujo de ejecución

## ■ Sentencias de bucle.

- Sentencias de bucle **determinadas.**

```
for (variable in objeto) {  
    sentencias_a_ejecutar  
}
```

Esta estructura **for** permite iterar por las propiedades enumerables de un objeto, elementos de un array, etc.

### Ejemplo:

```
var obj = {a:1, b:2, c:3};  
for (var prop in obj) {  
    console.log(`obj.${prop} = ${obj[prop]}`);  
}  
  
// Resultado:  
// obj.a = 1  
// obj.b = 2  
// obj.c = 3
```

```
var a = [4, -1, 3];  
for (let i in a) {  
    console.log(`${i}: ${a[i]}`);  
}  
  
// Resultado:  
// 0: 4  
// 1: -1  
// 2: 3
```

# Control del flujo de ejecución

## ■ Sentencias de bucle.

- Sentencias de bucle **determinadas.**

```
for (variable of iterable) {  
    sentencias_a_ejecutar  
}
```

Esta estructura **for** permite iterar sobre objetos iterables como Array, Map, Set, String, etc.

### Ejemplo:

```
var v = ['rojo', 'verde', 'azul'];  
for (let e of v) {  
    console.log(e);  
}
```

// Resultado:  
// rojo  
// verde  
// azul

## Control del flujo de ejecución

### ■ Sentencias de bucle.

- Sentencias de bucle **indeterminadas**.

#### Forma 1:

```
while (condición) {  
    sentencias_a_ejecutar  
}
```

#### Forma 2:

```
do {  
    sentencias_a_ejecutar  
} while (condición);
```

**Nota:** Aunque ambas formas son equivalentes, hay una diferencia importante entre ellas. En la primera forma las sentencias podrían no llegar a ejecutarse si la condición es falsa. Sin embargo, en la segunda forma las sentencias se ejecutarán como mínimo una vez antes de evaluar la condición.



# Control del flujo de ejecución

## ■ Sentencias de bucle.

- Sentencias de bucle **indeterminadas**.

### Ejemplo:

#### Forma 1:

```
var i = 10;  
while (i < 5) {  
    console.log('Número ' + i);  
    i++;  
}  
console.log('Fin');
```

```
// Resultado:  
// Fin
```

#### Forma 2:

```
var i = 10;  
do {  
    console.log('Número ' + i);  
    i++;  
} while (i < 5);  
console.log('Fin');
```

```
// Resultado:  
// Número 10  
// Fin
```

# Control del flujo de ejecución

## ■ Sentencias de bucle.

### ○ Sentencias **break** y **continue**.

- **break**. Permite salir no sólo de una estructura switch, sino también de una estructura de bucle.

Ejemplo:

```
var i = 0, n, num = 17;

while(i < 10) {
  n = Math.floor( 100 * Math.random() ) + 1;
  if( n % num == 0) break;
  i++;
}
if(i < 10)
  console.log(`Múltiplo de ${num}: ${n}`);
else
  console.log(`No se ha encontrado múltiplo de ${num}`);
```

## Control del flujo de ejecución

### ■ Sentencias de bucle.

#### ○ Sentencias **break** y **continue**.

- **continue**. Permite romper una iteración en un bucle y pasar a la siguiente, evitando así la ejecución del código que quedara pendiente en la iteración.

Ejemplo:

```
var i = 0;

while(i < 5) {
  if( i == 3) continue;
  console.log(`Iteración ${i}`);
  i++;
}
```

```
// Resultado:
// Iteración 0
// Iteración 1
// Iteración 2
// Iteración 4
```

# Funciones

## Funciones

- Son una de las herramientas fundamentales para la creación de bloques de código en JavaScript.
- Es un conjunto de instrucciones que realizan una tarea concreta o calculan un valor.
- Para poder utilizar una función hay que definirla previamente.
- La declaración de una función se hace mediante:
  - La palabra reservada **function**;
  - seguida del **nombre de la función**;
  - a continuación, una lista de **cero ó más parámetros** encerrados entre paréntesis y separados por comas;
  - por último, las **instrucciones** JavaScript que definen la función, encerradas **entre llaves { }**.

# Funciones

## Declaración de una función. Sintaxis.

```
function nombre_función( [param1[,param2[...paramN]]] ) {  
    ...  
    conjunto_de_instrucciones  
    ...  
}
```

Las funciones pueden devolver un valor mediante la instrucción **return**.

Ejemplo:

```
function calcularArea( base, altura ) {  
    let area = (base * altura) / 2;  
    return area;  
}
```

# Funciones

## Expresión de función

JavaScript permite crear funciones como expresiones.

- Estas funciones pueden ser anónimas.
- Son convenientes cuando se pasa una función como argumento a otra función.
- Permite crear funciones en base a una condición.

Ejemplo:

```
var area;

switch(figura){
  case 0: area = function(x){ return x * x; };
    break;
  case 1: area = function(b,h){ return b * h; };
    break;
  case 2: area = function(b,h){ return (b * h) / 2; };
    break;
}
```

# Funciones

## Ámbito de una función

- En JavaScript el ámbito de una función es el mismo que el de una variable.
- Las variables definidas dentro de una función no son accesibles desde fuera de la función.
- Una función puede acceder a todas las variables y funciones definidas dentro de su ámbito.

### Ejemplo:

```
var n1 = 10, n2 = 4;

function sumar(a,b){
  return a + b;
}

function operar(){
  var n1 = 3, n2 = 1;
  console.log(sumar(n1,n2));
}
```

```
operar(); // Resultado: 4

console.log(n1); // Resultado: 10
console.log(n2); // Resultado: 4
```



# Funciones

## Funciones anidadas y cierres (clausuras)

- En JavaScript se puede anidar una función dentro de otra.
- La función **anidada** (interna) es privada a su función contenedora (externa). Sólo es accesible desde la función contenedora.
- La función interna forma un **cierre**: la función interna puede utilizar los argumentos y variables de la función externa, mientras que la función externa no puede utilizar los argumentos y las variables de la función interna.

### Ejemplo:

```
function externa(x) {  
  function interna(y) {  
    return x + y;  
  }  
  return interna;  
}
```

```
fn_interna = externa(3); // fn_interna sería  
// una función que suma 3 a lo que se le pase  
  
resultado = fn_interna(5); // Resultado: 8  
externa(3)(4); // Resultado: 7
```

# Funciones

## Objeto arguments

- Permite acceder a los argumentos de una función como si fuera un array.
- Se puede saber el número total de argumentos de una función mediante **arguments.length**.

Ejemplo:

```
function listarArgs(separador) {  
    var res = '', // inicializa lista a devolver  
        i;  
    // itera por los argumentos  
    for (i = 1; i < arguments.length; i++) {  
        res += arguments[i] + separador;  
    }  
    return res;  
}  
  
listarArgs(' : ', 'rojo', 'verde', 'azul');  
// Resultado: "rojo : verde : azul : "
```

# Funciones

## Parámetros por defecto y parámetros REST

- A partir de la versión ES6, se pueden especificar valores por defecto para los parámetros.

Ejemplo:

```
function calcularImpuesto(precio, tasa = 0.21) {  
  return precio * (1 + tasa);  
}
```

- A partir de la versión ES6, se pueden especificar un número indefinido de argumentos en forma de array.

Ejemplo:

```
function aplicarTasa(tasa, ...valores) { // se utiliza "..."  
  return valores.map(function(x){return x * (1 + tasa);});  
}  
  
aplicarTasa(.21,23,14,17,20.5,16.25);  
// Resultado:  
// Array [ 27.83, 16.939999999999998, 20.57, 24.805, 19.662499999999998 ]
```

# Funciones

## Expresión de función flecha (=>)

- Tiene una sintaxis más corta y enlaza léxicamente con el valor **this**.

Sintaxis:

```
([param1[, param2]]) => {  
  statements  
}
```

```
param => expression
```

Ejemplo:

```
var v = [4, -3, 5, 6, 2, 7];  
v.forEach((e,idx) => {  
  if( e % 2 == 0)  
    console.log(`Elemento par en posición ${idx}: ${e}`);  
});
```

# Funciones

## Funciones asíncronas (*async functions*)

- JavaScript permite definir funciones asíncronas mediante el modificador **async**.
- Mediante la palabra reservada **await** se puede llamar a una función desde la función asíncrona y hacer que ésta espere hasta que la función llamada devuelva el resultado.

### Ejemplo:

```
async function getDatos(url) {  
  let v;  
  try {  
    v = await tareaAsincrona(url);  
  } catch(e) {  
    v = await falloTareaAsincrona(url);  
  }  
  return procesarDatosDevueltos(v);  
}
```

# Funciones

## Funciones *callback*

- JavaScript permite pasar funciones como argumentos a otras funciones. Estas funciones se llaman ***callback***.
- Una función *callback* se puede invocar añadiéndole "()" y los parámetros dentro.

### Ejemplo:

```
function sacarMensaje(texto){  
    console.log(texto);  
}  
  
function llamarFuncion(funcionALLamar, params){  
    funcionALLamar(params);  
}  
  
function main(){  
    llamarFuncion(sacarMensaje, 'Hola mundo!!!');  
}  
  
main(); // Resultado: "Hola mundo!!!"
```