



UD4: Estructuras definidas por el usuario

Desarrollo Web en Entorno Cliente

Índice

- [Arrays](#)
- [Funciones](#)
- [Objetos: Creación y uso](#)
- [Objetos: Otros aspectos](#)



Arrays

[Indice](#)

UD4: Estructuras definidas por el usuario

Array, vector o matriz

- Es la estructura más usada en la mayoría de los lenguajes
 - Conjunto de elementos ordenados en filas.
 - Permite acceder a los elementos de manera aleatoria (directa), usando un **índice** (indexados).
 - Si el índice es una cadena hablamos de un **array asociativo**
 - **Matriz:** Array bidimensional (usamos varios índices para acceder a los elementos).

Objeto Array

- Cada elemento es referencia por la posición que ocupa dentro del array
 - La posición se llama índice y es correlativa.
- En JS podemos tener distintos tipos de datos almacenados en cada posición del array.
- La indexación numérica siempre es “base cero”

Creación de Arrays indexados

```
/* Array vacío */  
var miArray= new Array();  
  
/* Array con número fijo de elementos */  
var miArray=new Array(40);  
  
/* Array inicializado al instanciar */  
var diasLaborables=new  
Array("Lunes","Martes","Miercoles","Jueves","Viernes");  
  
/* Definición de array de forma literal */  
var diasFestivos=["Sabado","Domingo"];
```

Creación de Arrays asociativos

```
var traducciones={  
  "Lunes": "Monday",  
  "Martes": "Tuesday",  
  "Miércoles": "Wednesday",  
  "Jueves": "Thursday",  
  "Viernes": "Friday",  
  "Sabado": "Saturday",  
  "Domingo": "Sunday"};
```

Acceso al Array

- Array indexado

```
var lunes=diasLaborables[0];
```

- Array asociativo

```
var traduccionLunes=diasLaborables["Lunes"];
```

```
var traduccionLunes=diasLaborables.Lunes;
```


Recorriendo un Array-I (bucle for)

- **Array indexado**

```
for(i=0;i<diasLaborables.length;i++)  
  console.log(diasLaborables[i]);
```

Recorriendo un Array-II (bucle for..in)

- Array indexado

```
for(var indice in diasLaborables)
    console.log("Indice "+indice+" Valor:"+diasLaborables[indice]);
```

- **Array asociativo**

```
for(var clave in traducciones)
    console.log("Clave "+clave+" Valor:"+traducciones[clave]);
```

Recorriendo un Array-III (bucle for..of)

- Array indexado

```
for(var dia of diasLaborables)  
    console.log(dia);
```

Recorriendo un Array-III (método forEach())

- `forEach(functionCallback)`
 - Ejecuta la función indicada una vez por cada elemento del array.
- **Array indexado**

```
diasLaborables.forEach( function(valor, indice) {  
    console.log("En el índice " + indice + " hay este valor: " + valor);  
});
```

Objeto Array: Propiedades y métodos

- Propiedades

length	Longitud del array
---------------	--------------------

- Métodos

concat(array2)	Concatena con array2 y devuelve una copia de los arrays unidos.
join(separador)	Une todos los elementos del array separados por separador
reverse()	Invierte el orden de los elementos del array

Objeto Array: Métodos

<code>reverse()</code>	Invierte el orden de los elementos del array
<code>slice([inicio[,fin]])</code>	Devuelve una copia de una parte del array empezando por inicio y acabando en fin
<code>sort()</code>	Ordena los elementos de un array
<code>toString()</code>	Convierte el array a cadena y devuelve el resultado
<code>splice(i,n,e1,e2...)</code>	Cambia el contenido de un array eliminando o añadiendo contenido. i indica a partir de donde se modifica el contenido. n indica el número de elementos a eliminar. En caso de ser 0, e1, e2... indica los elementos a añadir.

Objeto Array: Métodos

pop()	Elimina el último elemento del array y devuelve dicho elemento
push(elemento)	Añade elementos al final del array y devuelve el nuevo tamaño
shift()	Elimina el primer elemento del array y lo devuelve
unshift(elemento)	Añade un elemento al comienzo del array, devolviendo el nuevo tamaño

Borrado de elementos en un Array

- Podemos borrar un elemento:
 - Asignando el valor del elemento a borrar a null o cadena vacía.
 - Mediante el operador delete
 - Uso: `delete array[i]`

```
delete diasLaborables[0];
```

```
> diasLaborables  
< [undefined × 1, "Martes", "Miercoles", "Jueves", "Viernes"]
```


Borrado de elementos en un Array (2)

- Al borrar un elemento con los métodos anteriores no se reduce la longitud del array .
- Podemos eliminar un elemento o una secuencia de elementos ajustando el número de elementos usando el método `splice(pos,cantidad)`:

```
diasLaborables.splice(0,2);
```

```
diasLaborables
```

```
["Miercoles", "Jueves", "Viernes"]
```

Ordenando elementos en un array

- Función **sort([funcionComparacion])**
 - La función recibe un parámetro opcional que es el criterio que se va a usar para ordenar.
 - Dicha función recibe dos parámetros (a,b)
 - Para ordenar el array se irá llamando a dicha función sucesivamente.
 - En función de lo que devuelva la función se determina el resultado de la comparación entre a y b
 - Negativo: $a < b$
 - Cero: $a = b$
 - Positivo: $a > b$

Ordenando elementos: Ejemplo

- Función que usa sort() por defecto:

```
function (a,b){return a-b}
```

- Al comparar 40 y 100:
 - Se devuelve 40-100 (negativo)
 - Se determina que 40 es menor que 100.
- Al comparar 20 y 10:
 - Se devuelve 20-1 (positivo)
 - Se determina que 20 es mayor que 10.

Ordenando elementos: Ejemplo (2)

¿Cuál es el resultado de ejecutar este script?

```
function ordena(a,b){  
    return b-a;  
}  
  
var array=[4,2,6,3];  
  
array.sort(ordena);
```

Ejercicios propuestos: Ordenación

- Ordena un array numérico de manera aleatorio usando **sort()**.
- Tenemos un array de cadenas donde cada posición es el nombre y apellidos de un alumno.
 - Ordena usando **sort()** el array por los apellidos de los alumnos.

Ejercicio propuesto: Arrays

- Define un array países que contenga un listado de nombre de países.
- Implementa funciones que permitan:
 - Mostrar todos los elementos del array separados por un salto de línea (consola)
 - Mostrar los elementos del array en sentido inverso separados por un salto de línea
 - Mostrar los elementos del array alfabéticamente separados por un salto de línea
 - Añadir un elemento al comienzo del array
 - Añadir un elemento al final del array
 - Borrar un elemento al comienzo del array (indicar cuál es)
 - Borrar un elemento al final del array (indicar cuál es)
 - Mostrar la posición en la que se encuentra un elemento

No usar variables globales!

Arrays multidimensionales

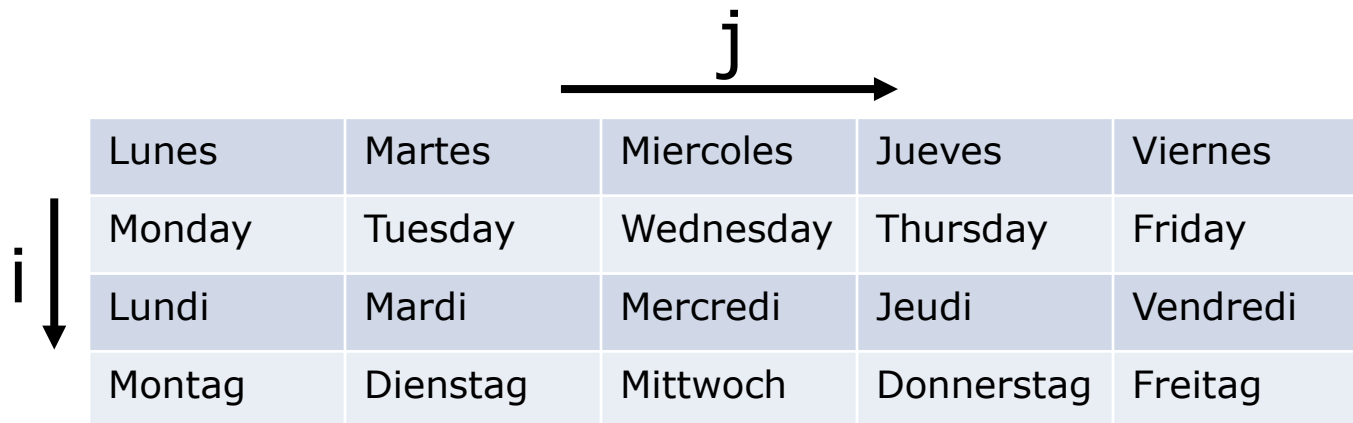
- Los arrays bidimensionales no existen de manera nativa en JS
 - Podemos crear un array que en sus posiciones contengan otros arrays.
 - Podemos entender los arrays bidimensionales como *arrays de arrays*.
 - Acceso: `nombre[indice1][indice2]`

Arrays multidimensionales: Ejemplo

```
var diasLaborables=new Array();

diasLaborables[0]=new Array("Lunes","Martes","Miercoles","Jueves","Viernes");
diasLaborables[1]=new Array("Monday","Tuesday","Wednesday","Thursday","Friday");
diasLaborables[2]=new Array("Lundi","Mardi","Mercredi","Jeudi","Vendredi");
diasLaborables[3]=new Array("Montag","Dienstag","Mittwoch","Donnerstag","Freitag");

console.log("La semana empieza en "+diasLaborables[0][0]);
console.log("Week ends on "+diasLaborables[1][4]);
```



	Lunes	Martes	Miercoles	Jueves	Viernes
	Monday	Tuesday	Wednesday	Thursday	Friday
	Lundi	Mardi	Mercredi	Jeudi	Vendredi
	Montag	Dienstag	Mittwoch	Donnerstag	Freitag

Arrays multidimensionales: Recorrido

```
for(i=0;i<diasLaborables.length;i++)  
    for(j=0;j<diasLaborables[i].length;j++)  
        console.log(" "+diasLaborables[i][j]);
```

Depurando tablas

- **console.table(array)**
 - Muestra en forma tabular un array bidimensional

```
> console.table(notas)
```

VM1004:1

(index)	0	1	2	3	4	5
0	2	8	4	3	9	1
1	6	6	5	4	6	0
2	6	5	8	2	8	3
3	9	8	1	9	3	2

► Array(4)

Ejercicio propuesto: Arrays bidimensionales-I

- Queremos almacenar usando una tabla los resultados obtenidos en las elecciones en Villaconejos, teniendo en cuenta que:
 - Ha habido 5 sedes para votar (Ayuntamiento, Polideportivo, Instituto, Mercado y Colegio)
 - Se han presentado 4 partidos (Puede que Villaconejos (PV), Obreros de Villaconejos (OV), Villaconejos Por el Si (VpSI), Unión Progreso y Villaconejos (UPV).
- Se solicitarán al usuario los votos por sede y partido.
 - Para simularlo, en su lugar se pueden generar aleatoriamente los votos correspondientes a cada partido (entre 5 y 10 votos).

Ejercicio propuesto:

Arrays bidimensionales-II

- A continuación se mostrarán por consola:
 - Una tabla con todos los colegios electorales y partidos, así como sus votos asociados.
 - Calcular el número total de votos por partido y por sede
 - Indicar de mayor a menor los votos recibidos

Map y Set

- **Map** es un diccionario clave-valor donde cualquier tipo puede ser usado como clave.
 - Es la mayor diferencia con los arrays asociativos, donde las claves solo pueden ser cadenas de texto.
- **Set** permite almacenar valores únicos de cualquier tipo, con la única limitación de que no pueden estar duplicados.



Objeto Map: Métodos (principales)

Map([conjunto])	Constructor. Acepta un conjunto de pares-valor
set(key,value)	Añade nueva pareja clave-valor
get(key)	Obtiene el valor asociado a una clave
delete(key)	Borra una pareja clave-valor mediante la clave
has(key)	Comprueba si hay determinada clave en el mapa

Objeto Map: Propiedades y métodos

- Propiedades

size	Número de valores en el mapa
-------------	------------------------------

- Métodos

values()	Devuelve los valores del mapa
keys()	Devuelve las claves del mapa
entries()	Devuelve un conjunto de matrices [key,value]
clear()	Elimina todos los valores del mapa



Objeto Map: Ejemplo

```
var map = new Map();  
  
map.set( 'uno', 1 );  
map.set( 'dos', 2 );  
map.set( 'tres', 3 );  
  
console.log( "map.get('dos') =", map.get('dos') );  
  
map.delete('tres');  
  
console.log( "map.has('tres') =", map.has('tres') );
```




Objeto Map: Ejemplo-II

- Las claves pueden ser de cualquier tipo:

```
var obj = {  
  a: 1,  
  b: 2  
};  
  
function mult(a, b) {  
  return a * b;  
}  
  
var map = new Map();  
  
map.set( false, 0 );  
map.set( obj, "es un objeto" );  
map.set( mult, "Función mult()" );  
  
console.log( map.get(mult) );
```



Objeto Map: Recorrido

```
var myMap = new Map( [ [0, "cero"], [1, "uno"], [2, "dos"] ] );  
  
for (var [key, value] of myMap) {  
    console.log(key + " = " + value);  
}
```



Objeto Set: Métodos (principales)

add(value)	Añade un nuevo valor
delete(value)	Borra un valor
delete(key)	Borra una pareja clave-valor mediante la clave
has(key)	Comprueba si hay determinada clave en el mapa

Objeto Set: Propiedades y métodos

- Propiedades

size	Número de valores en el mapa
-------------	------------------------------

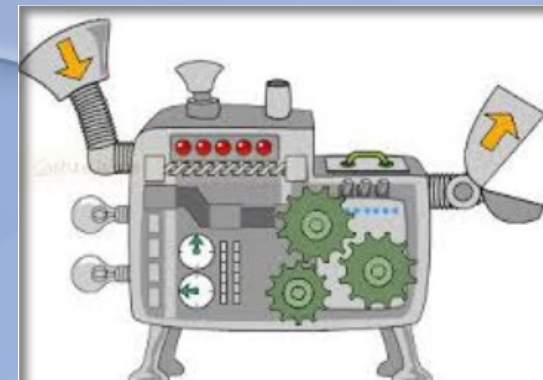
- Métodos

values()	Devuelve los valores del mapa
keys()	Devuelve las claves del mapa
entries()	Devuelve un conjunto de matrices [key,value]
clear()	Elimina todos los valores del mapa



Objeto Set: Ejemplo

```
var obj1 = {  
  a: 1,  
  b: 2  
};  
  
var obj2 = {  
  a: 1,  
  b: 2  
};  
  
var map = new Set();  
  
map.add( obj1 );  
map.add( obj2 );  
  
console.log( 'map.has(obj1) = ', map.has(obj1) );  
console.log( 'map.has(obj2) = ', map.has(obj2) );
```



Funciones

[Indice](#)

UD4: Estructuras definidas por el usuario

Funciones

- Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y se pueden reutilizar de manera sencilla.
- Facilitan mucho la organización, y por consiguiente el mantenimiento y depuración de los programas.
- Se suele diferenciar:
 - **Procedimientos:** Sólo ejecutan acciones
 - No existen en JS.
 - **Funciones:** Ejecutan acciones y devuelven valores

Funciones simples

- Primero declaramos la función y luego la utilizamos (llamada o invocación):

```
/* Definición */  
function nombreFuncion(){  
    sentencias;  
}  
  
/* Llamada o invocación */  
nombreFuncion();
```


Funciones simples: Ejemplo

```
function sumayMuestra(){  
    var resultado = numero1 + numero2;  
    alert("El resultado es "+ resultado);  
}
```

Definición

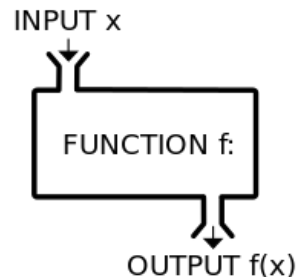
```
var resultado;  
var numero1=3;  
var numero2=5;  
  
sumayMuestra();  
  
numero1=5;  
numero2=6;  
sumayMuestra();
```

Llamada o
invocación

E/S de datos en funciones

- **Argumentos/parámetros**
 - Permiten especificar las **entradas** de la función
 - Ejemplo (suma): Los sumandos.
- **Retorno**
 - Especifica el **valor que devuelve** la función.
 - Ejemplo (suma): El resultado de la suma.

```
/* Definición */  
function nombreFuncion(argumento1, argumento2){  
    sentencias;  
    return valor;  
}
```



Argumentos: Ejemplo

```
function sumayMuestra(primerNumero, segundoNumero){  
    var resultado = primerNumero + segundoNumero;  
    alert("El resultado es: "+ resultado);  
}
```

Definición

```
//Declaración de las variables  
var numero1=3;  
var numero2=5;  
  
//Llamada a la función  
sumayMuestra(numero1, numero2);
```

Valor de retorno: Ejemplo

```
function suma(primerNumero,segundoNumero){  
    var resultado = primerNumero + segundoNumero;  
    return resultado;  
}
```

Definición

```
//Declaración de las variables  
var numero1=3;  
var numero2=5;  
  
//Llamada a la función  
var resultado=suma(numero1,numero2);  
alert(resultado);
```

Funciones y sobrecarga

- **Sobrecarga:** Podemos nombrar con un mismo identificador diferentes variables u operaciones.
 - Podemos tener dos o más funciones con mismo nombre y distinto comportamiento (que reciben distinto número de parámetros)
- En JS no existe la sobrecarga, **pero** podemos llamar a una función con cualquier número de parámetros
 - Es decir, podemos declarar una función con un número de parámetros pero invocarla con cualquier otro juego de parámetros que se quiera o necesite.

Funciones y sobrecarga - II

- En caso de no coincidir los parámetros no se considera un error del lenguaje, sino que el intérprete se intentará adaptar:
 - Si faltan parámetros, su valor será “undefined”
 - Si sobran parámetros, podemos acceder a través de la variable **arguments**
 - Es un array que siempre está disponible dentro de una función y contiene todos los parámetros que se le han pasado a la función.

Sobrecarga - Ejemplo

```
function concatena(p1,p2,p3){  
    alert(p1+" "+p2+" "+p3);  
}
```

p3 → undefined

```
concatena("Hola","Caracola");
```

```
function concatena (){  
    var salida="";  
    for (var i=0;i<arguments.length;i++)  
        salida+=arguments[i]+" ";  
    alert(salida);  
}
```

```
concatena("Hola","Caracola", "Como vas");
```

Estableciendo valores por defecto

```
function miProfesion(profesion){  
    var p=profesion || "Fontanero";  
    console.log(p);  
  
}  
  
miProfesion("Informático");  
miProfesion();
```


Ámbito de las variables

- El ámbito (*scope*) de una variable es la zona del programa donde se puede usar la variable.
- Tradicionalmente se definen dos ámbitos:
 - **Ámbito local:** La variable está definida en el ámbito de una función (usando **var**) y sólo puede usarse en la misma.
 - **Ámbito global:** La variable está definida a nivel de programa y puede usarse en cualquier parte del mismo (incluidas funciones).
 - Dentro de una función, podemos definir una variable global si omitimos la palabra **var**.

Si existen dos variables de mismo nombre en ámbitos distintos prevalece la más local.

Variables locales y globales

- El alcance de una variable **global** se limita al documento cargado en la ventana del navegador.
- Si la página se cierra todas las variables definidas en ella se eliminan
 - Para que el valor de la variable persista se usan técnicas como las cookies.



Ámbito de las variables: Ejemplo

¿Cuál es correcto?

```
function creaMensaje(){  
  var mensaje="Mensaje de prueba";  
}  
  
creaMensaje();  
alert(mensaje);
```

```
function creaMensaje(){  
  var mensaje="Mensaje de prueba";  
  alert(mensaje);  
}  
  
creaMensaje();
```

Ámbito de las funciones: Ejemplo

¿Es correcto?

```
function creaMensaje(){  
    mensaje="Mensaje de prueba";  
}  
  
creaMensaje();  
alert(mensaje);
```

Ámbito de las funciones: Ejemplo

¿Cuál es la salida de los siguientes scripts?

```
var mensaje="gana la de fuera";

function muestraMensaje(){
var mensaje="gana la de dentro";
alert(mensaje);
}

alert(mensaje);
muestraMensaje();
alert(mensaje);
```

```
var mensaje="gana la de fuera";

function muestraMensaje(){
mensaje="gana la de dentro";
alert(mensaje);
}

alert(mensaje);
muestraMensaje();
alert(mensaje);
```

Modo estricto (use strict)

- Se especifica mediante la cadena 'use strict'
- Podemos usarlo:
 - A nivel global (comienzo de un script)
 - A nivel de función (comienzo de una función).
- Se nos advertirá de un error en las variables declaradas sin usar **var**.

```
"use strict";
```

```
numero=5;
```

numero is not defined

```
function imprime(){  
    console.log(numero);  
}
```

```
function imprime(){  
    "use strict";  
    nombre="Pepe";  
    console.log(nombre);  
}
```

nombre is not defined



Ámbito de visibilidad: **let**

- **let** nos permite declarar una variable de alcance local.
 - Su alcance se limita al bloque, declaración o expresión donde se está usando.
 - Es decir, al contexto (entre llaves) en el que se ha definido.

var define una variable global o local en una función sin importar el ámbito de bloque.



Ejemplo: var vs let

```
function varTest() {  
  var x = 31;  
  if (true) {  
    var x = 71; // misma variable!  
    console.log(x); // 71  
  }  
  console.log(x); // 71  
}
```

```
function letTest() {  
  let x = 31;  
  if (true) {  
    let x = 71; // Variable distinta  
    console.log(x); // 71  
  }  
  console.log(x); // 31  
}
```


Funciones predefinidas

- Podemos usarlas a nivel global en cualquier parte del código JS

decodeURI()	Decodifica los caracteres especiales de una URL salvo: ,/?:@&=+\$#
decodeURIComponent()	Decodifica todos los caracteres especiales de una URL
encodeURI()	Codifica los caracteres especiales de una URL salvo: ,/?:@&=+\$#
encodeURIComponent()	Codifica todos los caracteres especiales de una URL
escape()	Codifica caracteres especiales en una cadena salvo: *@-_.+./
unescape()	Decodifica caracteres especiales en una cadena salvo: *@-_.+./

Funciones predefinidas

isNaN()	Determina cuando un valor no es número (contiene NaN)
Number(objeto)	Convierte el valor de un objeto a un número
parseFloat(cadena)	Convierte una cadena a un real
parseInt(cadena)	Convierte una cadena a un entero
isFinite()	Determina si un valor es un número finito válido
eval()	Evalúa una cadena y la ejecuta si contiene código u operaciones

Conversión de cadenas

- `parseInt(cadena)` ó `parseFloat(cadena)`
 - Obtiene un entero o flotante a partir de una cadena.

```
var numero=parseInt("30");  
alert(numero);
```

30

```
var numero2=parseInt("40px");  
alert(numero);
```

40

```
var numero3=parseInt("5pepe");  
alert(numero);
```

5

```
var numero4=parseInt("8+2");  
alert(numero);
```

8

```
var numero=parseFloat("3,2");  
alert(numero);
```

3

```
var numero2=parseFloat("3.2");  
alert(numero);
```

3.2

eval(cadena)

- **eval()** recibe una cadena:
 - Si contiene una expresión, se evalúa.
 - Si contiene una o más sentencias, se ejecutan.

```
eval("4*5");
```

20

```
eval("20/0");
```

Infinity

```
eval("var a=2;var b=4;a+b");
```

6

Problemas de eval()

- Puede comprometer la seguridad de la aplicación o ejecutar código malicioso

```
eval("while(true) contador++;");
```

- El código dentro del `eval()` resulta difícil de leer
- Evaluar el código del `eval()` es más lento que evaluar una cadena normal.

Por lo general se considera una mala práctica y debemos evitarlo en la medida de lo posible

Funciones anónimas

- Son funciones que se definen sin utilizar un identificador
 - Facilitan la programación, pero pueden complicar la lectura y depuración del código.

```
setTimeout(function (){pasos++;},1000);
```

Es una práctica muy extendida en JS pero requiere una cierta experiencia para depurar.

Funciones anónimas

- Podemos introducir una función anónima en una variable para luego usarla.

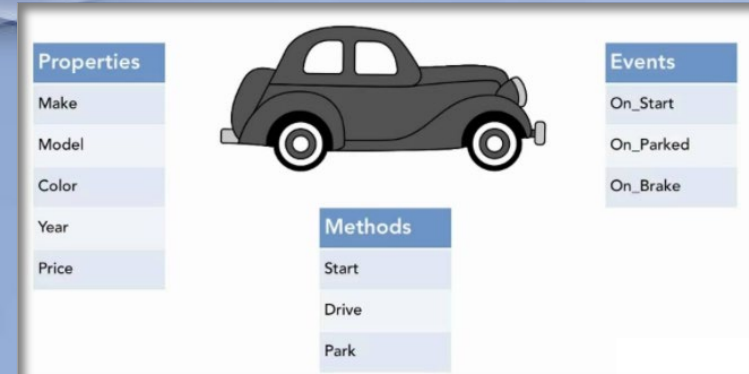
```
var avanza=function(nombre){console.log("Avanza "+nombre)};

avanza("Pepe");
avanza("Luis");
```

Closures (*cláusulas*)

- Es una porción de código que se evalúa en un entorno de ejecución que contiene variables de otro entorno de ejecución
 - Se crean para todas las funciones definidas dentro de otra función
 - Su uso es limitado y se reserva a ciertas técnicas (ejemplo: Simular propiedades privadas)

```
var x = "estoy fuera";  
function funcionExterna() {  
    var x = "estoy dentro";  
    function funcionAnidada() { alert(x); }  
    funcionAnidada();  
}  
  
funcionExterna();
```

Objetos: Creación y uso

[Indice](#)

UD4: Estructuras definidas por el usuario

Objetos y clases

- Un **objeto** encapsula un conjunto de datos relacionados entre sí de modo que los puedo tratar de manera conjunta.
- Habitualmente en un objeto distinguimos:
 - Estado: Contenido de las variables que lo forman (**propiedades**).
 - Comportamiento: Acciones (funciones) que puedo realizar con él (**métodos**).
- Una **clase** es una estructura que agrupa objetos con propiedades y métodos comunes

Objetos, clases y Javascript

- JS no permitía crear clases propiamente dichas como en lenguajes Orientados a Objetos como Java o C++
- Ya se pueden crear clases con la palabra reservada *class*, pero su uso es muy reciente y puede dar problemas.

Objetos, clases y Javascript

- Hasta hace relativamente poco se podían simular unos elementos muy parecidos a las clases que se denominan **pseudoclases**.
 - Técnicamente, un objeto es un array asociativo formado por las propiedades y los métodos del objeto.

Objetos en Javascript

- Los objetos pueden crearse en JS de tres maneras:
 - Objetos literales
 - Funciones constructoras.
 - Prototipos
 - Clases

Objetos literales

- Asigno a una variable un literal de objeto.
- Si no vamos a volver a usar el objeto, no necesito asignarle un nombre.
- Pero Javascript debe interpretar el objeto cada vez que aparezca una referencia al mismo código (lento).

```
var perro={  
  nombre:"Fosi",  
  raza:"Callejero",  
  edad:5,  
  esTravieso:true  
}  
  
alert(perro.raza);  
alert(perro["edad"]);
```

```
var perro={  
  "nombre":"Fosi",  
  "raza":"Callejero",  
  "edad":5,  
  "esTravieso" :true  
}  
  
alert(perro.raza);  
alert(perro["edad"]);
```

```
var perro=new Object();  
perro.nombre="Fosi";  
perro.raza="Callejero";  
perro.edad=5;  
perro.esTravieso=true;  
  
alert(perro.raza);  
alert(perro["edad"]);
```

EQUIVALENTES

Funciones constructoras

- En JS no existe el concepto de constructor
- Emulamos el comportamiento de los constructores mediante el uso de funciones.
 - Definimos el perfil del objeto con una función (constructora).
 - El acceso a las propiedades se realiza de manera implícita usando la palabra clave **this**

```
function Perro(nombre, raza, edad, esTravieso){  
  this.nombre=nombre;  
  this.raza=raza;  
  this.edad=edad;  
  this.esTravieso=esTravieso;  
}
```

Funciones constructoras

- Una vez que hemos creado la función podemos crear nuevos objetos usando **new**.
- Acceso a las propiedades:
 - **objeto.propiedad**
 - `objeto["propiedad"]`

```
var fosi=new Perro("Fosi", "callejero",5,true);  
var reina=new Perro("Reina", "husky",2,false);  
  
console.log(reina.edad);  
console.log(reina["raza"]);
```


Funciones constructoras-Métodos

- Se invocan con la sintaxis `objeto.metodo(parametros)`
- Dos opciones:
 - A. Se definen como cualquier otra función y se asocian a un nombre de método en la función constructora
 - B. Se definen en la función constructora con una función anónima.
- Desde los métodos puedo acceder a las propiedades usando `this`.
 - Pueden recibir parámetros.

Métodos: Ejemplo (A)

```
function Perro(nombre,raza, edad, esTravieso){  
  this.nombre=nombre;  
  this.raza=raza;  
  this.edad=edad;  
  this.esTravieso=esTravieso;  
  this.ladra=guau;  
  this.llama=ven;  
}  
  
function guau(){  
  alert("Guauuuuu!");  
}  
  
function ven(nombreAmo){  
  alert(this.nombre + ": Soy "+nombreAmo+" y te ordeno que vengas!");  
}  
  
var fosi=new Perro("Fosi", "callejero",5,true);  
fosi.ladra();  
fosi.llama("Marcial");
```

Métodos: Ejemplo (B)

```
function Perro(nombre,raza, edad, esTravieso){
  this.nombre=nombre;
  this.raza=raza;
  this.edad=edad;
  this.esTravieso=esTravieso;

  this.ladra=function(){
    console.log("Guauuuuu!");
  };

  this.llama=function(nombreAmo){
    console.log(this.nombre + ": Soy "+nombreAmo+" y te ordeno que vengas!");
  };
}

var fosi=new Perro("Fosi", "callejero",5,true);
fosi.ladra();
fosi.llama("Marcial");
```

Getters y setters

- Por defecto, en JS todas las propiedades son públicas.
- Podemos acceder a las mismas invocando **nombreObjeto.nombrePropiedad**
 - **Una convención** para indicar que no deberíamos acceder a las mismas es **añadir como prefijo un subrayado** (_) al nombre
- Se recomienda definir *getters* y *setters* para el acceso a las propiedades
 - Fomenta el encapsulamiento de los objetos, facilitando depuración y mantenimiento.

Getters y setters: Ejemplo

```
function Alumno(nombre, edad){  
  this._nombre = nombre;  
  this._edad = edad;  
  this._numMatricula = null;  
  
  this.getNombre = getNombre;  
  this.setNombre = setNombre;  
}  
  
function getNombre(){  
  return this._nombre;  
}  
  
function setNombre(nombre){  
  this._nombre=nombre;  
}  
  
/*... Getters y setters del resto de propiedades */
```

Funciones constructoras-Inconveniente

- Las funciones se crean de nuevo en cada instancia.
- Esto penaliza el rendimiento

```
Perro {nombre: "Fosi", raza: "callejero", edad: 5, esTravieso: true, ladra: f, ...} ⓘ  
  edad: 5  
  esTravieso: true  
  ▶ ladra: f ()  
  ▶ llama: f (nombreAmo)  
    nombre: "Fosi"  
    raza: "callejero"  
  ▶ __proto__: Object  
▼ Perro {nombre: "Lana", raza: "Pastor Alemán", edad: 2, esTravieso: false, ladra: f, ...} ⓘ  
  edad: 2  
  esTravieso: false  
  ▶ ladra: f ()  
  ▶ llama: f (nombreAmo)  
    nombre: "Lana"  
    raza: "Pastor Alemán"  
  ▶ __proto__: Object
```

Prototipos

- Todos los objetos de JS incluyen una referencia interna a otro objeto llamado *prototype*.
- Cualquier propiedad o método que contenga el prototipo está presente en el objeto original de forma automática.
 - Es la “plantilla” con el que se fabrican objetos de ese tipo.
 - Si modificamos dicha plantilla, todos los objetos creados a partir de la misma tendrán esas características.
- Para evitar el problema del rendimiento, podemos añadir los métodos directamente al prototipo.

Prototipos: Ejemplo

```
function Perro(nombre,raza, edad, esTravieso){  
  this.nombre=nombre;  
  this.raza=raza;  
  this.edad=edad;  
  this.esTravieso=esTravieso;  
}  
  
Perro.prototype.ladra=function(){  
  console.log("Guauuuuu!");  
}  
  
Perro.prototype.llama=function(nombreAmo){  
  console.log(this.nombre + ": Soy "+nombreAmo+" y te ordeno que  
vengas!");  
};
```

Los métodos se añaden al prototipo (común para todas las clases)

Prototipos y clases predefinidas

- Podemos usar los prototipos para modificar y/o añadir propiedades y métodos de objetos predefinidos.

```
String.prototype.toArray = function(){  
    return this.split('');  
}
```

Clases

- Hasta hace muy poca palabra *class* estaba reservada para su uso en futuras versiones de Javascript.
- En la actualidad *class* está en funcionamiento al igual que en otros lenguajes de programación.

Clases

- No obstante, su uso es muy reciente y podemos toparnos con problemas.
 - Por ejemplo, seguimos sin poder encapsular parámetros y métodos para hacerlos privados.
- Y si comparamos las instancias creadas mediante *class* con las creadas mediante prototipos veremos que no hay diferencia.

Clases: Ejemplo

```
class Perro {  
    constructor(nombre,raza,edad,esTravieso){  
        this.nombre=nombre;  
        this.raza=raza;  
        this.edad=edad;  
        this.esTravieso=esTravieso;  
    }  
  
    ladra(){  
        console.log("Guauuuuu!");  
    }  
  
    llama(nombreAmo){  
        console.log(this.nombre + ": Soy " + nombreAmo  
            + " y te ordeno que vengas!");  
    }  
}
```

Clases: Setters y Getters

- En Javascript siguen sin poder definirse propiedades o métodos privados.
- Por lo que, como convención aceptada en todo el mundo, aquellos elementos que se consideran privados de la clase se les da un nombre que empiece por guión bajo
 - `_propiedad`

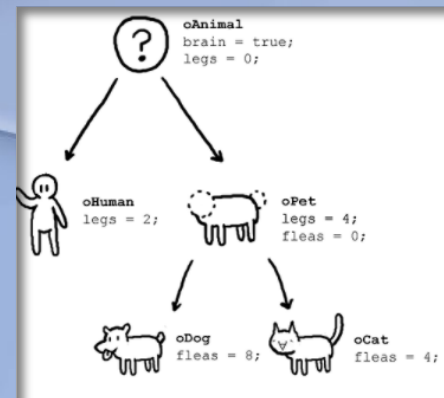
Clases: Setters y Getters

- Con el nuevo formato de clases de Javascript se implementan los métodos *Get* y *Set*, pero no dejan de ser una mera simulación, ya que podremos seguir accediendo a los atributos sin ningún problema.

Clases: Setters y Getters Ejemplo

```
class Perro {  
    constructor(nombre,raza,edad,esTravieso){  
        this._nombre=nombre;  
        this._raza=raza;  
        this._edad=edad;  
        this._esTravieso=esTravieso;  
    }  
  
    get nombre(){  
        return this._nombre;  
    }  
  
    set nombre(name){  
        this._nombre = name;  
    }  
}
```

Y esto se repetiría para cada una de las propiedades.



Objetos: Otros aspectos

[Indice](#)

UD4: Estructuras definidas por el usuario

Herencia

- Podemos hacer que una clase (hija) herede la estructura y el comportamiento de otra clase (padre).
 - `ClaseHija.prototype=new ClasePadre();`
 - `ClaseHija.prototype.constructor=ClaseHija;`
- Llamadas desde la clase hija a métodos de la clase padre:
 - `ClasePadre.call(this,argumentos);`

Herencia: Ejemplo

Constructor sin parámetros

```
function Animal () {}  
  
Animal.prototype.crece = function () {  
  console.log("Me hago grande");  
};
```

```
function Felino () {  
}
```

```
Felino.prototype = new Animal();  
Felino.prototype.constructor = Felino;
```

```
Felino.prototype.maulla = function () {  
  console.log('Miauuuu');  
};
```

```
var isidoro = new Felino();  
isidoro.crece();  
isidoro.maulla();
```

Herencia: Ejemplo

Constructor con parámetros

```
function Animal (edad) {  
  this.edad = edad;  
}  
  
Animal.prototype.crece = function () {  
  this.edad = this.edad + 1;  
};
```

```
function Felino (edad) {  
  Animal.call(this, edad);  
}  
  
Felino.prototype = new Animal();  
Felino.prototype.constructor = Felino;  
  
Felino.prototype.maula = function () {  
  console.log('Miauuuu');  
};
```

```
var isidoro = new Felino(3);  
isidoro.crece(); //edad=4  
isidoro.maula();
```

Herencia: Ejemplo

Redefiniendo métodos

```
function Animal (edad) {  
    this.edad = edad;  
}  
  
Animal.prototype.crece = function () {  
    this.edad = this.edad + 1;  
};
```

```
function Felino (edad) {  
    Animal.call(this,edad);  
}  
  
Felino.prototype = new Animal();  
Felino.prototype.constructor = Felino;
```

```
Felino.prototype.crece = function () {  
    this.edad = this.edad + 2;  
};
```

```
var isidoro = new Felino(3);  
isidoro.crece(); //edad=5  
isidoro.maula();
```

Herencia: Ejemplo

Llamando métodos del padre (una vez redefinidos)

```
function Animal (edad) {  
    this.edad = edad;  
}  
  
Animal.prototype.crece = function () {  
    this.edad = this.edad + 1;  
};
```

```
function Felino (edad) {  
    Animal.call(this,edad);  
}  
  
Felino.prototype = new Animal();  
Felino.prototype.constructor = Felino;  
  
Felino.prototype.crece = function () {  
    Animal.prototype.crece.call(this);  
    this.edad = this.edad + 2;  
};
```

```
var isidoro = new Felino(3);  
isidoro.crece(); //edad=6  
isidoro.maula();
```

Excepciones

- Mecanismo muy similar al de otros lenguajes de programación.
 - **try** encierra el bloque de código JS donde se controlan las excepciones.
 - Debe ir seguido por un bloque **catch** o un bloque **finally**
 - Cuando se produce una excepción en **try**, se ejecutan las instrucciones contenidas dentro de **catch**
 - Después de **catch**, podemos definir un bloque con **finally**, cuyo código se ejecuta independientemente de la excepción ocurrida en **try**

Excepciones: Ejemplo

```
try{  
    resultado=5/a;  
}  
catch(excepcion){  
    console.log(excepcion);  
}
```

ReferenceError: a is not defined
at index.js:4

Más excepciones

- También podemos lanzar excepciones manualmente mediante **throw**

```
try{
    if(typeof a === "undefined" || isNaN(a)){
        throw new Error('La variable "a" no es un número');
    }
    var resultado = 5/a;
}
catch(excepcion){
    console.error(excepcion);
}

finally{
    console.warn("Se ejecuta si hubo excepción");
}
```


Agrupando por paquetes/namespaces-I

- En Javascript no existe una funcionalidad para crear paquetes
- Crear todas nuestras clases, variables y funciones en un único espacio de nombres puede ser problemático:
 - No podemos repetir nombres
 - Se dificulta la organización del código en proyectos grandes.

Agrupando por paquetes/namespaces-II

- Para simular los paquetes podemos crear un objeto global único y hacer que todas las variables y funciones pertenezcan al mismo.

```
var miProyecto={};  
  
miProyecto.variable1="Una variable";  
miProyecto.variable2="Otra variable";  
miProyecto.variable3={ uno:1, dos:2};  
miProyecto.hazAlgo=function(){  
    console.log("Hago algo");  
}
```

```
var miProyecto={  
    miProyecto.variable1="Una variable";  
    miProyecto.variable2="Otra variable";  
    miProyecto.variable3={ uno:1, dos:2};  
    miProyecto.hazAlgo=function(){  
        console.log("Hago algo");  
    }  
};
```

Módulos anidados

- Podemos anidar un módulo dentro de otro usando composición de objetos.

```
var miProyecto={};

miProyecto.interfazUsuario={
    /* Lógica del módulo */
};

miProyecto.gestorEventos={
    /* Lógica del módulo */
};

miProyecto.manejadoresAJAX={
    /* Lógica del módulo */
};
```

Módulos anidados-II

- Pero la práctica anterior puede dar problemas si tenemos los módulos en ficheros distintos..

Al redefinir el objeto de nuevo estamos borrando el contenido definido en interfaz.js

```
var miProyecto={};

miProyecto.interfazUsuario={
  /* Lógica del módulo */
};
```

interfaz.js

```
var miProyecto={};

miProyecto.gestorEventos={
  /* Lógica del módulo */
};
```

eventos.js

Módulos anidados-III

- Podemos comprobar si existen los objetos padre antes de usarlos

```
var miProyecto= miProyecto || {};
```

JSON (JavaScript Object Notation)

- Es un formato sencillo para el intercambio de información.
- Permite representar estructuras de datos (**arrays**) y objetos (**arrays asociativos**) en forma de texto.
- Es una alternativa al XML para intercambio de datos, al ser más fácil de leer/escribir y mucho más conciso.

Tipos de datos en JSON

- Tipos de datos simples
 - Números, cadenas, booleanos, null.
- Array
 - Cero o más valores de cualquier tipo entre corchetes
- Objetos
 - Colecciones no ordenadas de pares <nombre>:<valor> puestas entre llaves
 - El nombre va entre comillas

```
{ "departamento":8,  
  "nombredepto":"Ventas",  
  "director": "juan rodriguez",  
  "empleados":[{ "nombre":"Pedro","apellido":"Fernandez"},  
                { "nombre":"Jacinto","apellido":"Benavente"} ] }
```