



# Funciones y objetos

UD1: Introducción a JS

Javier G. Pisano ([javiergpi@educastur.org](mailto:javiergpi@educastur.org))



# Al acabar la lección...

- Habrás recordado el concepto de **clase/objeto**
- Conocerás las **propiedades y métodos** más interesantes de los siguientes objetos predefinidos de JavaScript:
  - String
  - Math
  - Number
  - Date
  - Array



# Al acabar la lección... (II)

- Recordarás lo que ya sabes sobre funciones
- Aprenderás cosas más avanzadas sobre funciones
  - Valores por defecto
  - Sobrecarga
  - Funciones anónimas
  - Funciones flecha
- Sabrás cómo se **declaran** las clases en JS, con sus propiedades y métodos
- Habrás recordado el concepto de **herencia** entre clases y sabrás cómo se hace en JS
- Podrás modularizar tus aplicaciones.



# Indice

- Objetos predefinidos
- Funciones
- Objetos y clases



# Objetos predefinidos

[Indice](#)

# Objetos

- 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
    - A dichas variables las llamamos **propiedades**
  - **Comportamiento**:
    - Acciones (funciones) que puedo realizar con él.
    - A las funciones asociadas a un objeto las llamamos **métodos**.

Clase: Agrupa un conjunto de objetos con estado y comportamiento común

# En general...

- Creación de un objeto (una instancia)
  - `var nombreObjeto=new NombreClase`
- Acceso a propiedades:
  - `nombreObjeto.propiedad`
- Acceso a un método de un objeto:
  - `nombreObjeto.método([parametros])`

# Creación de String

- Sintaxis tradicional

```
var miCadena="texto de la cadena";
```

- Creación alternativa (sintaxis de objeto)

```
var miCadena=new String("texto de la cadena");
```

Independientemente de la sintaxis usada podemos acceder a los métodos y propiedades



# Propiedades y métodos de String

Propiedades	
<b>length</b>	Longitud de la cadena
Métodos	
<b>charAt(pos)</b>	Devuelve el carácter ubicado en <b>pos</b>
<b>charCodeAt(pos)</b>	Devuelve el Unicode del carácter ubicado en <b>pos</b>
<b>fromCharCode(code)</b>	Convierte valores Unicode a caracteres
<b>indexOf(car)</b>	Devuelve la posición de la primera ocurrencia del carácter buscado por <b>car</b>
<b>lastIndexOf(car)</b>	Devuelve la posición de la última ocurrencia del carácter buscado por <b>car</b>

# Más métodos de String

Métodos	
<b>replace(c1,c2)</b>	Busca la subcadena <b>c1</b> y la reemplaza con <b>c2</b>
<b>search(c)</b>	Busca la subcadena <b>c</b> y devuelve la posición donde se encontró
<b>slice(inicio,fin)</b>	Extrae y devuelve la subcadena entre los índices <b>inicio</b> y <b>fin</b> .
<b>split(separador)</b>	Devuelve un array de subcadenas. El parámetro especifica el carácter a usar para la separación de la cadena.
<b>substr(inicio,[lon])</b>	Devuelve la subcadena que comienza en <b>inicio</b>
<b>toLowerCase()</b>	Convierte la cadena a minúsculas
<b>toUpperCase()</b>	Convierte la cadena a mayúsculas

# String (Ejemplos)

```
var cadena="El parapente es un deporte de riesgo";  
console.log("La longitud de la cadena es "+cadena.length);  
console.log(cadena.toLowerCase());  
console.log(cadena.charAt(3));  
console.log(cadena.indexOf("pente"));  
console.log(cadena.slice(3,16));
```



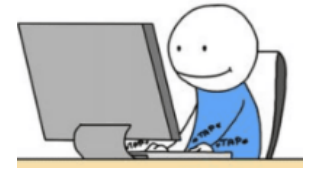
La longitud de la cadena es 36

el parapente es un deporte de riesgo

p

7

parapente es



# UD1 ACT2 Ejercicio 1

- Crea un programa que pida al usuario su nombre y apellidos y muestre:
  - El tamaño del nombre más los apellidos (sin contar espacios).
  - La cadena en minúsculas y en mayúsculas.
  - Que divida el nombre y los apellidos y los muestre en 3 líneas, donde ponga  
Nombre:  
Apellido 1:  
Apellido 2:
- Una propuesta de nombre de usuario, compuesto por el nombre, la inicial del primer apellido y la inicial del segundo: *ej. Para Javier Gonzalez Pisano sería javiergp*. Implementa esta funcionalidad mediante una función anónima.
- Una propuesta de nombre de usuario compuesto por las dos primeras letras del nombre y de los dos apellidos: *ej. jagopi*. Implementa esta funcionalidad mediante una función flecha.

# Objeto Math

- Permite realizar operaciones matemáticas
- Es una clase **estática**
  - No tiene constructor: no creamos instancias de objetos de tipo Math

```
var x = Math.PI; // Devuelve el número PI  
var y = Math.sqrt(16); // Calcula la raíz cuadrada de 16
```

# Propiedades y métodos de String

Propiedades	
PI	Número PI (aprox. 3.14)
SQRT2	Raíz cuadrada de 2 (aprox. 1.41)
Algunos Métodos	
ceil(x)	Número <b>x</b> redondeado al alza al siguiente entero
floor(x)	Número <b>x</b> redondeado a la baja al anterior entero
round(x)	Redondea <b>x</b> al entero más próximo
random()	Devuelve un numero aleatorio entre 0 y 1
pow(x,y)	Devuelve el resultado de <b>x</b> elevado a <b>y</b>
sqrt(x)	Raíz cuadrada de <b>x</b>
max(x,y,z...n) min(x,y,z...n)	Máximo/mínimo de los números que se pasan como parámetros

# Objeto Number

## Propiedades

<b>MAX_VALUE</b>	Número más alto posible
<b>MIN_VALUE</b>	Número más bajo posible
<b>NEGATIVE_INFINITY</b>	Infinito negativo (en caso de overflow)
<b>POSITIVE_INFINITY</b>	Infinito positivo (en caso de overflow)

## Algunos Métodos

<b>toFixed(n)</b>	Devuelve el número usando <b>n</b> dígitos decimales. Si no se especifica <b>n</b> por defecto se devuelve el número entero.
<b>toString(B)</b>	Representación como cadena en base B

# Conversión entre String y Number

- De String a Number:

```
let n = Number(s);
```

- De Number a String:

```
let s = num.toString();
```



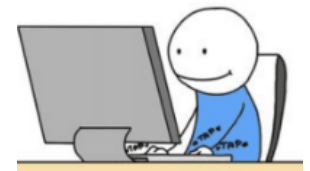
# Clase Date

- Se usa para trabajar con fechas y horas
- Constructores:

<b>Date()</b>	Crea un objeto <b>Date</b> con la fecha actual
<b>Date(cadena)</b>	Crea un objeto <b>Date</b> a partir de la información de cadena
<b>Date(a,m,d,h,m,s,ms)</b>	Crea un objeto <b>Date</b> a partir del año, mes, día, hora, minuto, segundo y milisegundo.
<b>Date(a,m,d)</b>	Crea un objeto <b>Date</b> a partir del año, mes y día.

# Métodos de Date

<b>setHours(), getHours()</b>	Cambia/devuelve la hora (0-23)
<b>setMilliseconds()/getMilliseconds()</b>	Cambia/devuelve los milisegundos (0-9999)
<b>setMinutes()/getMinutes()</b>	Cambia/devuelve los segundos (0-59)
<b>setMonth(), getMonth()</b>	Cambia/devuelve el mes (0-11)
<b>setSeconds()/getSeconds()</b>	Cambia/devuelve los segundos (0-59)
<b>setDate()/getDate()</b>	Cambia/devuelve el día del mes (1-31)
<b>getDay()</b>	Devuelve el día de la semana (0-6)
<b>getFullYear()</b>	Devuelve el año (4 dígitos)
<b>getTime()</b>	Devuelve los milisegundos desde el 1/01/1970
<b>toString(), toLocaleString(), toGMTString()...</b>	Métodos para mostrar la fecha como cadena usando diferentes formatos.



# UD1 ACT2 Ejercicio 2

- Crea la siguiente función

Función <code>infoCumple()</code>	
Recibe	dia: Día de tu cumpleaños mes: Mes de tu cumpleaños
Devuelve	Objeto con dos propiedades: <b>días</b> : Días que faltan para tu cumpleaños <b>diaSemana</b> : Día de la semana de tu cumpleaños

Pruébalo con fechas límite (hoy, mañana...)

- **AMPLIACIÓN**: Pedir fecha de nacimiento en lugar de cumpleaños

# Array, vector o matriz

- Es la estructura más usada en la mayoría de los lenguajes
  - Cada elemento se referencia por la posición que ocupa dentro del array
    - La posición se llama **índice** y es correlativa.
    - La indexación numérica siempre es “base cero”
- Cuando definimos un array en JavaScript realmente estamos definiendo un objeto de la clase **Array**

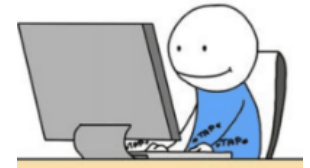
# Arrays

- Es una colección de elementos que pueden ser **del mismo o distinto tipo**

```
var dias = ["Lunes", "Martes", "Miércoles",  
"Jueves", "Viernes", "Sábado", "Domingo"];
```

- Podemos acceder a los elementos del array a través de un índice
  - Las posiciones de un array comienzan a contarse en 0 y no en 1

```
var diaSeleccionado = dias[0]; // diaSeleccionado = "Lunes"  
var otroDia = dias[5]; // otroDia = "Sábado"
```



# UD1 ACT2 Ejercicio 3

- Crea la siguiente función

Función <code>numerosCorrectos()</code>	
Recibe	Un array de números que se espera que estén ordenados. Ejemplo: [1,2,3] ó [1,3,4,5,6,7,8,9]
Devuelve	<p>Un número que es el índice del primer elemento fuera de orden dentro de dicho array.</p> <p>Por ejemplo, si recibe el array [1,2,3,7,4] debería devolver el número 3 (es la posición donde hay un elemento fuera de orden)</p> <p>Si todos los elementos están en orden se devuelve un -1</p>

- Prueba la función con varios casos de prueba diversos. Recuerda probar posiciones límites

# Creación de Arrays indexados

- Podemos inicializarlo de distintas maneras:

```
var mascotas=[];
```

```
var mascotas=new Array();
```

```
var mascotas=["Pepi","Luci","Bom"];
```

```
var mascotas=new Array("Pepi","Luci","Bom");
```

```
var mascotas=new Array(10);  
/* Se inicializan los elementos a null*/
```

- En JS podemos tener **distintos tipos de datos** almacenados en cada posición del array

```
var mezcla=new Array("Pepi",2,true);
```

# Arrays asociativos

- En los **arrays asociativos** el índice es una cadena

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

DEFINICIÓN

¡OJO! El último valor no lleva coma

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

Acceso (opción 1)

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

Acceso (opción 2)



# Objeto Array: Propiedades y métodos

Propiedades	
<code>length</code>	Longitud del array
Algunos Métodos	
<code>concat(array2)</code>	Concatena con <b>array2</b> y devuelve una copia de los arrays unidos.
<code>join(separador)</code>	Une todos los elementos del array separados por <b>separador</b>
<code>reverse()</code>	Invierte el orden de los elementos del array
<code>toString()</code>	Convierte el array a cadena y devuelve el resultado
<code>sort()</code>	Ordena los elementos de un array

# Objeto Array: Más métodos

Algunos Métodos	
<b>slice([inicio[,fin]])</b>	Devuelve una copia de una parte del array empezando por <b>inicio</b> y acabando en <b>fin</b>
<b>splice(i,n,e1,e2...)</b>	Cambia el contenido de un array eliminando o añadiendo contenido. <b>i</b> indica a partir de donde se modifica el contenido. <b>n</b> indica el número de elementos a eliminar. En caso de ser 0, <b>e1, e2...</b> indica los elementos a añadir.
<b>pop()</b>	Elimina el último elemento del array y devuelve dicho elemento
<b>push(elemento)</b>	Añade elementos al final del array y devuelve el nuevo tamaño
<b>shift()</b>	Elimina el primer elemento del array y lo devuelve
<b>unshift(elemento)</b>	Añade un elemento al comienzo del array, devolviendo el nuevo tamaño

# Borrado de elementos en un Array

- Podemos borrar un elemento **sin reducir la longitud del array**:

- Asignándole el valor null o cadena vacía.

- Mediante el operador delete

- Uso: delete array[i]

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

```
> diasLaborables
```

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

- 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"]
```

# Recorriendo un array. Bucle for...in

- Se utiliza para recorrer un array y hacer que el índice se actualice automáticamente
  - Funciona aunque cambie el número de elementos del array

```
var dias=["Lunes", "Martes", "Miércoles", "Jueves", "Viernes",  
"Sabado", "Domingo"];  
  
for(i in dias)  
    console.log(dias[i]);
```

# Recorriendo un Array-I

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

Bucle for  
(sólo para índices  
numéricos)

Bucle for..in

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

Índices numéricos

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

Array asociativo

# Recorriendo un Array-II

ES6

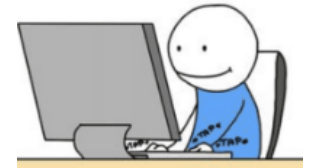
```
for(var dia of diasLaborables)
  console.log(dia);
```

for(variable of array)

forEach(función Callback)

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

Ejecuta la función indicada una vez por cada elemento del array.



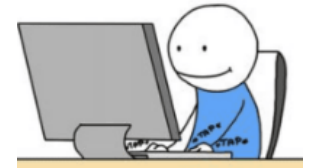
# UD1 ACT2 Ejercicio 4

- Crea la siguiente función

Función <code>tiposEnArray()</code>	
Recibe	Un array de elementos de distintos tipos  Ejemplo: <code>tiposEnArray[true,1,"Casa",function(){}]</code>
Devuelve	Otro array con el mismo número de elementos del array que recibe que contiene los tipos de cada elemento del array original  En el caso del ejemplo: <code>[boolean, number, string, function]</code>

- Prueba la función con varios casos de prueba diversos.

# UD1 ACT2 Ejercicio 5



- Define un array **países** que contenga un listado de países
- Implementa funciones que permitan:
  - Mostrar todos los elementos del array separados por un salto de línea usando `for...of(console)`
  - Mostrar los elementos del array en sentido inverso separados por un salto de línea usando `foreach`
  - Mostrar los elementos del array alfabéticamente separados por un salto de línea usando una sola sentencia
  - 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)

¡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: Ver el contenido

- Recorrido con bucles:

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

- Depuración por consola: **console.table(array)**

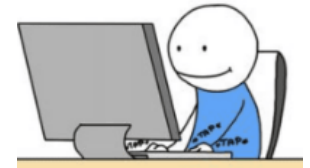
> console.table(diasLaborables)

VM431:1

(index)	0	1	2	3	4
0	'Lunes'	'Martes'	'Miercoles'	'Jueves'	'Viernes'
1	'Monday'	'Tuesday'	'Wednesday'	'Thursday'	'Friday'
2	'Lundi'	'Mardi'	'Mercredi'	'Jeudi'	'Vendredi'
3	'Montag'	'Dienstag'	'Mittwoch'	'Donnerstag'	'Freitag'

► Array(4)

# UD1 ACT2 Ejercicio 6



## Elecciones en Villaconejos



# UD1 ACT2 Ejercicio 6 (Ampliación)



> console.table(resultados)

VM252:1

(index)	0	1	2	3	4	5
0	' '	'Ayuntamiento'	'Polideportivo'	'Instituto'	'Mercado'	'Colegio'
1	'PV'	8	10	8	10	7
2	'OV'	10	8	7	9	6
3	'VpSI'	5	7	6	9	9
4	'UPV'	9	7	5	9	10

► Array(5)

Calcular el número total de votos por partido y por sede

# 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
  - Con cualquier tipo nos referimos no sólo a cadenas, números... sino incluso objetos o funciones
- **Set** permite almacenar valores **únicos** de cualquier tipo, es decir no pueden estar duplicados

# Objeto Map: Propiedades y métodos

Propiedades	
<b>size</b>	Número de valores en el mapa
Algunos Métodos	
<b>Map([conjunto])</b>	Constructor. Acepta un conjunto de pares-valor
<b>set(key,value)</b>	Añade nueva pareja clave-valor
<b>get(key)</b>	Obtiene el valor asociado a una clave
<b>delete(key)</b>	Borra una pareja clave-valor mediante la clave
<b>has(key)</b>	Comprueba si hay determinada clave en el mapa
<b>values()</b>	Devuelve los valores del mapa
<b>keys()</b>	Devuelve las claves del mapa
<b>entries()</b>	Devuelve un conjunto de matrices [key,value]
<b>clear()</b>	Elimina todos los valores del mapa

# Objeto Map: Ejemplo

## Definición

```
let mapa = new Map();
mapa.set('1', 'str1'); // un string como clave
mapa.set(1, 'num1'); // un número como clave
mapa.set(true, 'bool1'); // un booleano como clave

// Map mantiene el tipo de dato en las claves, por lo
que estas dos son diferentes:
console.log( mapa.get(1) ); // 'num1'
console.log( mapa.get('1') ); // 'str1'
console.log( mapa.size ); // 3
```

## Recorrido

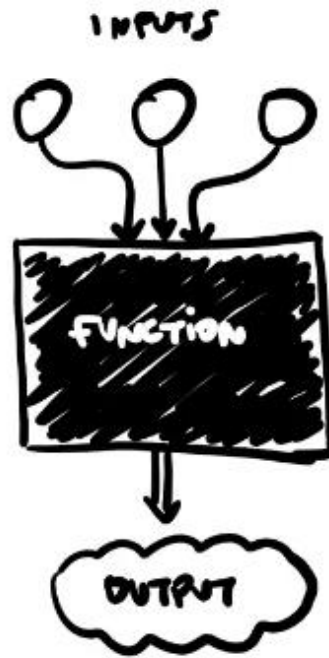
<pre>for(var [clave, valor] of mapa) {</pre>	1 = str1
<pre>    console.log(clave + " = " +valor);</pre>	1 = num1
<pre>}</pre>	true = bool1



# Objeto Set: Propiedades y métodos

ES6

Propiedades	
<b>size</b>	Número de valores en el mapa
Algunos Métodos	
<b>add(value)</b>	Añade un nuevo valor
<b>delete(value)</b>	Borra un valor
<b>delete(key)</b>	Borra una pareja clave-valor mediante la clave
<b>has(key)</b>	Comprueba si hay determinada clave en el mapa
<b>values()</b>	Devuelve los valores del mapa
<b>keys()</b>	Devuelve las claves del mapa
<b>entries()</b>	Devuelve un conjunto de matrices [key,value]
<b>clear()</b>	Elimina todos los valores del mapa



# Funciones

[Indice](#)

# Funciones

- Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y se pueden reutilizar
- Facilitan mucho la organización, y por tanto el **mantenimiento y depuración** de los programas.

```
/* Definición */
```

```
function nombreFuncion(){  
  sentencias;  
}
```

DEFINICIÓN

```
/* Llamada o invocación */  
nombreFuncion();
```

LLAMADA Ó INVOCACIÓN

# Funciones simples: Ejemplo (malo)

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

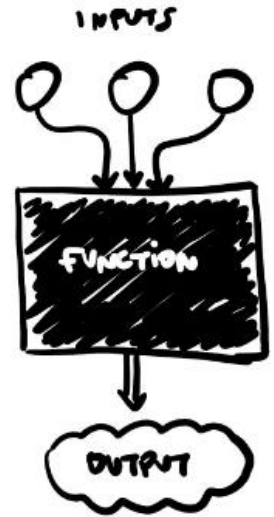
Definición

```
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 (regular)

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

Definición

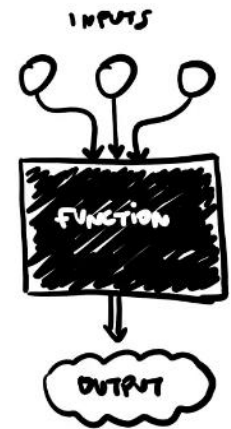
```
//Declaración de las variables  
var numero1=3;  
var numero2=5;  
  
//Llamada a la función  
sumayMuestra(numero1, numero2);
```

# Valor de retorno: Ejemplo (bueno)

```
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);
```



# Ámbito de las variables: 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);
```



# Funciones y sobrecarga (I)

- **Sobrecarga:** Podemos nombrar con un mismo identificador diferentes variables u operaciones.
  - Podemos tener dos o más funciones con mismo nombre y distinto comportamiento (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);  
}
```

```
concatena("Felipe","Juan");
```

Felipe Juan undefined

p3 → undefined

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

Felipe Juan Froilán

# Parámetros por defecto

ES5  
classic

```
function saludar(nombre, titulo, saludo){  
  var t=titulo || 'D.';  
  var s=saludo || 'Hola' + t  
  console.log(s + ' ' + nombre);  
};  
saludar('Jordi'); // Hola D. Jordi
```

ES6

```
function saludar(nombre, titulo = 'D.', saludo = 'Hola' + titulo){  
  console.log(saludo + ' ' + nombre);  
};  
saludar('Jordi'); // Hola D. Jordi
```

# Funciones anónimas

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

```
let double = function(n) { return n * 2 }
```

- A partir de ES6 se extendió el uso de la **función flecha** o **función arrow**:

```
let double = n => n * 2;
```

- La llamada para usar la función sigue siendo igual:

```
alert( double(3) ); // 6
```

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

# Funciones flecha

- Notación alternativa muy usada para definir funciones

```
function sumar(x,y){  
    return x+y  
}  
const sumar = (x,y)=>x+y;
```

EQUIVALE A

```
const sumar = (x,y)=>x+y;
```

*Parámetros que recibe la función*

*Cuerpo de la función*  
Si solo tiene una línea no hay que poner llaves y se asume un return

# Funciones flecha

Si la función solo tiene una sentencia  
(que es el return)

- Sin function
- Sin llaves
- Sin return

```
() => sentencia; //función sin parámetros  
unParam=> sentencia; //función con un parámetro  
(param1, param2,paramN) => sentencia; //más parámetros
```

Si la función tiene varias  
sentencias

- Sin function
- Con llaves
- Con return

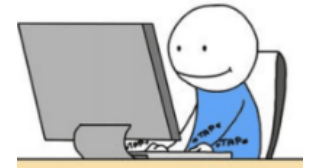
```
() => {  sentencias; }  
unParam=> {  sentencias; }  
(p1, p2,pN) => {  sentencias; }
```

# Ejemplo funciones flecha

```
//Sintaxis convencional 1 ("estilo Java")  
function arraysConcatenados(array1, array2) {  
    return array1.concat(array2);  
}  
  
//Sintaxis convencional 2 ("estilo JavaScript")  
const arraysConcatenados= function(array1, array2) {  
    return array1.concat(array2);  
}  
  
// Sintaxis con función flecha ("estilo JavaScript Moderno")  
const arraysConcatenados= (array1, array2) => array1.concat(array2);  
  
//A las tres se les invoca de la misma forma  
console.log(arraysConcatenados([1,2],[3,4,5]));
```

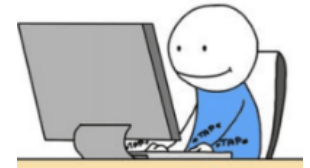


# UD1 ACT2 Ejercicio 7



- Desarrolla un script con una función **crearPersona** que pueda recibir los siguientes datos:
  - Nombre y apellidos
  - Nombre, apellidos y edad
    - De no recibir la edad esta tendrá por defecto el valor 0.
  - Nombre, apellidos, edad y un número indeterminado de formas de contacto (números de teléfono, email, etc.)
- Posteriormente deberá mostrar por consola un texto en el que se indique toda la información del usuario.
  - Usa `forEach` para recorrer las formas de contacto.
    - Prueba a usar una función anónima y una función arrow dentro de `forEach`

# UD1 ACT2 Ejercicio 7

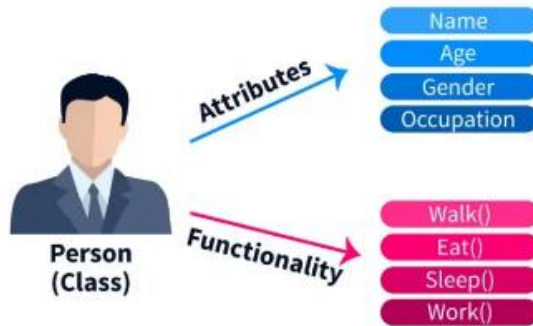


```
crearPersona("Jordi", "Hurtado");  
crearPersona("Jordi", "Hurtado", 123);
```

```
crearPersona("Jordi", "Hurtado", undefined, 222222, "jordi@tv.es");
```

A green arrow originates from the 'undefined' parameter in the function call above and points to the 'Edad:0' field in the object below.

```
Nombre:Jordi  
Apellidos:Hurtado  
Edad:0  
Contactos: 222222 jordi@tv.es
```



# Objetos y clases

[Indice](#)

# Objetos, clases y JavaScript

- JS no permitía crear clases hasta su versión ECMAScript 2015, donde se empieza a usar la palabra reservada *class*
  - *Antes se usaba function para crear Prototypes*

```
class NombreClase {  
  constructor(parametro1 [,parametro 2...]) {  
    this.propiedad1 = parametro1;  
    [this.propiedad2 = parametro2;  
  }  
}
```

DECLARACIÓN

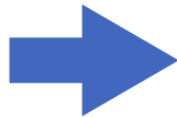
```
let nombreObjeto = new NombreClase (argumentos);
```

INSTANCIA

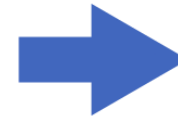
Primero es necesario declarar la clase y luego acceder a ella para evitar los errores de referencia.

# Organización

Funciones



Clases



Ficheros (módulos)



# Clases: Setters y getters

- JS no permite definir propiedades o métodos privados
  - Como convención, el nombre de los elementos privados de la clase comienza por guión bajo
    - `_propiedad`
  - El nombre de los setter y getters es el mismo pero en mayúsculas
  - Si pusiésemos el mismo nombre al método que a la propiedad se entraría en un bucle
- Esto es una mera simulación, ya que podremos seguir accediendo a los atributos sin ningún problema

# Clases: 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

```
let miPerro=new Perro("Toby", "Dálmata", 7, true);  
console.log (miPerro.Nombre); //lo permite el get
```

# 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!");  
    }  
}
```





# Utilidad

- Para generar los getters y setters de manera más rápida puedes instalar la extensión



JavaScript (ES6) code snippets v1.8.0

charalampos karypidis | 6.820.638 | ★★★★★ (31)

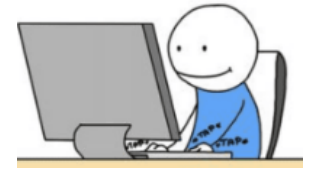
Code snippets for JavaScript in ES6 syntax

Instalar



Trigger	Content
con→	adds default constructor in the class <code>constructor() {}</code>
met→	creates a method inside a class <code>add() {}</code>
pge→	creates a getter property <code>get propertyName() {return value;}</code>
pse→	creates a setter property <code>set propertyName(value) {}</code>

# UD1 ACT2 Ejercicio 8



## Persona, Ancestro y Sucesor



# Métodos estáticos

- Pertenecen a la clase, no a los objetos
- Se pueden usar sin instanciar ningún objeto

```
static nombreMetodo (parametros) { //código }
```

# Herencia

- Podemos hacer que una clase (hija) herede la estructura y el comportamiento de otra clase (padre).
  - **class ClaseHijo extends ClasePadre**
- Para hacer referencia a atributos del padre:
  - **super(atributo)**
- Llamadas desde la clase hija a métodos de la clase padre:
  - **super.metodo()**

# Ejemplo de Herencia

```
class Animal {
    constructor(nombre) {
        this._nombre = nombre;
    }
    hablar() {
        console.log(this._nombre + ' hace un ruido.');
```

```
class Perro extends Animal {
    hablar() {
        super.hablar();
        console.log(this._nombre + ' ladra.');
```

# Modularización

- Una **buena práctica** es la **modularización**
  - Dividir programas en pequeños **módulos independientes** que pueden ser importados
- En un principio, y de forma nativa, la forma más extendida era incluir varias etiquetas `<script>` desde nuestra página HTML
  - Varios ficheros JS separados, cada uno para una finalidad concreta.
    - Poco modular
    - Lento, sobrecarga al cliente con múltiples peticiones extra

# Módulos

- Tener en cuenta:
  - Los módulos utilizan automáticamente modo `strict mode`
  - Los módulos se ejecutan una única vez aunque se haga referencia a ellos en varias etiquetas `<script>`.
  - El fichero se carga en diferido (como si tuviera la palabra `defer`)
  - Las características de un módulo no están disponibles a nivel global: solamente se puede acceder a las funciones importadas en el script en el que se importan

*Para utilizar módulos es necesario ejecutar los ficheros desde un servidor; si lo haces localmente obtendrás un error de CORS (Cross-Origin Request Blocked) debido a requisitos de seguridad de JS*

# Módulos: ejemplo

- En el html indicar que el script es un módulo:

```
<script src="ModulosImportExport.js" type="module"></script>
```

- En el archivo donde está el código que queremos reutilizar (*MisFunciones.js*):

```
export const cadenaMayuscula = str => str.toUpperCase();
```

- En el archivo donde lo queremos usar (*ModulosImportExport.js*)

```
import { cadenaMayuscula } from "../MisFunciones.js";  
                                //Hay que poner ./ en la ruta  
const saludo = cadenaMayuscula("¡Hola, caracola!");  
console.log(saludo);
```



# Más módulos

- Puedo exportar funciones, constantes...

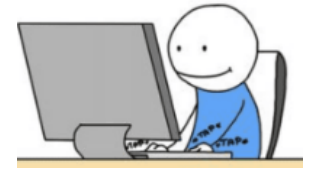
```
export function add(x,y){  
    return x+y  
}  
export const puntos=[10, 20, 30]  
export default puntos
```

En la etiqueta script del html indicamos que vamos a usar módulos con el atributo `type="module"`

- Puedo importar uno a uno – desestructurando con {}, o en general (necesitamos un **export default**)

```
import {add, puntos} from './add.js'  
import porDefecto from './add.js'  
  
console.log(add(10,20)); //30  
console.log(puntos); //[10, 20, 30]  
console.log (porDefecto); //[10, 20, 30]
```

# UD1 ACT2 Ejercicio 9



## Clase Familia



# Objetos literales

Notación abreviada  
“Línea directa” con JSON

- Se crean con las llaves {}
- Se entienden como un conjunto de variables de cualquier tipo declaradas como **clave: valor** sin necesidad de crear una clase

```
// Esto es un objeto vacío  
const objeto = {};
```

```
const jugador = {  
  nombre: "Manz",  
  vidas: 99,  
  potencia: 10,  
};
```

- Acceso a propiedades con punto o corchete

```
// Notación con puntos (preferida)  
console.log(jugador.nombre); // Muestra "Manz"  
// Notación con corchetes  
console.log(jugador["vidas"]); // Muestra 99
```

# ¿y qué pasa con los métodos?

```
const usuario = {  
  nombre: "Manolito García",  
  edad: 30,  
  nacimiento: {  
    pais: "España",  
    ciudad: "Oviedo"  
  },  
  amigos: ["Menganito", "Antoñito"],  
  activo: true,  


sendMail: function () {  
      return "Enviando email..."  
    }  
  },  
}


```



```
sendMail () {  
  return "Enviando email..."  
}
```

EQUIVALENTE:

```
console.log(usuario);  
console.log(usuario.nombre);  
console.log(usuario.nacimiento.ciudad);  
console.log(usuario.amigos)  
console.log(usuario.sendMail); //devuelve el código  
console.log(usuario.sendMail()); //ejecuta la función
```

## ¿Te suena?

# Shorthand property names

- Podemos crear un objeto a partir de otras constantes/variables
  - Existe una notación abreviada donde sólo es necesario poner su nombre y se crea la propiedad

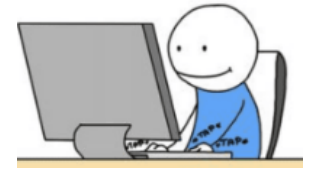
```
const nombre="portatil"  
const precio =3000;  
  
const nuevoProducto = {  
  nombre: nombre,  
  precio: precio  
}
```



```
const nombre="portatil"  
const precio =3000;  
  
const nuevoProducto = {  
  nombre  
  precio  
}
```

- Lo mismo se aplica a los Arrays

# UD1 ACT2 Ejercicio 10



## Libros y biblioteca

