Introducción a Javascript

Manuel Molino Milla

27 de septiembre de 2017

Índice

1. Introducción

1.1. ¿Qué es JavaScript?

- Es el lenguaje de programación en la web.
- Los navegadores ejecutan código JavaScript. Hoy en día las páginas web contiene un alto contenido de éste lenguaje. Hablamos de JavaScript en el lado del cliente.
- Bases de datos *NoSQL* como *CoucheDB* o *MongoDB* usan JavaScript en sus script o consultas.
- Encontramos JavaScript en el lado del servidor en proyectos como *Node.js*

1.2. Características de JavaScript

- Es un lenguaje de programación interpretado.
- Es orientado a objetos.
- Basado en prototipos.
- Débilmente tipado.
- Dinámico.

1.3. Historia de JavaScript

- Fue desarrollado originalmente por *Brendan Eich* de *Netscape* con el nombre de *Mocha*.
- Posteriormente fue renombrado a *LiveScript* y posteriormente pasó a ser JavaScript
- En 1997 se propuso que JavaScript fuera adoptado como estándar *ECMA* con el nombre de *ECMAScript*, posteriormente fue un estándar *ISO*
- Hoy en día el último estándar es de junio de 2015 denominado *ECMAScript 6*

1.4. Ejecución de JavaScript

- Podemos ejecutar en un navegador y utilizar la consola del mismo. Ejemplo en *Firefox* en *menu* tenemos la opción de *desarrollador* y en éste la *consola web*.
- Podemos usar un intérprete como puede ser *node*.

1.5. Instalación Node.js

1.6. Ejecución de código

console.log('hola mundo');

1.7. Diferencias con otros lenguajes

- JavaScript no soporta clases como otros lenguajes como Java
- En cambio en JavaScript la mayoría de datos son objetos o se acceden a través de objetos.
- Incluso las funciones son objetos por lo que hace de ellas funciones de primera clase.
- En JavaScript se pueden crear objetos en cualquier momento y añadir o quitar propiedades de éstos.
- Creando objetos:

```
//creando un objeto sin propiedades
//posteriormente se incluyen
var objeto1 ={};
objeto1.propiedad  = "soy un objeto";
objeto1.indice = 1;
console.log(objeto1);
//creando un objeto con propiedades
//posteriormente quitamos alguna de ellas
var objeto2 = {
  nombre: 'soy otro objeto',
  indice: 2
}
console.log(objeto2);
delete objeto2.indice;//borrando objetos
console.log(objeto2);
```

2. Tipos de datos en JavaScript

2.1. Introducción

- Aunque JavaScript no usa clase, tiene dos tipos de datos: datos primitivos y referencias.
- Tipos primitivos que almacena en memoria dicho datos simple.
- Y las referencias que almacena las posiciones de memoria donde está localizado el objeto.

2.2. Tipos primitivos en JavaScript

boolean con valores *true* o *false*

number cualquier entero o número en coma flotante.

string un caracter o un conjunto de caracteres delimitados por comillas simples o dobles.

null con un único valor null.

undefined con un único valor undefined, es el caso en que una variable no se le ha asignado valor alguno.

En *ECMAScript 6* aparece un nuevo tipo denominado *symbol*

2.3. Number

- Representa tanto enteros de 32 bits y reales de 64 bits.

  var entero = 555;
  var real = 600.2;

Tenemos valores especiales como:

- Number.MAX_VALUE
- Number.MIN_VALUE
- Number.POSITIVE_INFINITY
- Number.NEGATIVE_INFINITY
- Tenemos métodos como *isInfinite()* para veriificar estos dos últimos valores.
- NaN (*Not a Number*)
- Y tenemos el método isNan() para comprobar si es un *NaN*. Ejemplo isNaN("elephant"); que devuelve *true*

Tenemos la biblioteca Math para operaciones matemáticas:

- Math.E *(número e)*
- Math.abs(-900) *valor absoluto de de -900*
- Math.pow(2,3) *2 elevado a tres.*

Podemos convertir *string* a enteros o float:

- simplemente con "12"+3 realiza el casting a *string.*

- parseInt("230",10); (el 10 es la base);
- Interesante comprobar que es un número:

```
var underterminedValue = "elephant";
if (isNaN(parseInt(underterminedValue,2)))
 {
   console.log("handle not a number case");
  }
else
 {
   console.log("handle number case");
 }
```

2.4. string

Es una secuencia de caracteres Unicode. Se pueden crear con comillas dobles o comillas simples.
Una característica es que podemos usar *wrapper* para envolver este tipo de dato primitivo:

```
var s = new String("dummy"); //Creates a String object
console.log(s); //"dummy"
console.log(typeof s); //"object"
var nonObject = "1" + "2"; //Create a String primitive
console.log(typeof nonObject); //"string"
var objString = new String("1" + "2"); //Creates a String object
console.log(typeof objString); //"object"
//Helper functions
console.log("Hello".length); //5
console.log("Hello".charAt(0)); //"H"
console.log("Hello".charAt(1)); //"e"
console.log("Hello".indexOf("e")); //1
console.log("Hello".lastIndexOf("l")); //3
console.log("Hello".startsWith("H")); //true
console.log("Hello".endsWith("o")); //true
console.log("Hello".includes("X")); //false
var splitStringByWords = "Hello World".split(" ");
console.log(splitStringByWords); //["Hello", "World"]
var splitStringByChars = "Hello World".split("");
console.log(splitStringByChars); //["H", "e", "l", "l", "o", " ",
"W", "o", "r", "l", "d"]
console.log("lowercasestring".toUpperCase()); //"LOWERCASESTRING"
console.log("UPPPERCASESTRING".toLowerCase());
//"upppercasestring"
console.log(" There are no spaces in the end ".trim());
//"There are no spaces in the end"
```

2.5. undefined

Aparece cuando a una variable no asignamos valor:

var xl;
console.log(typeof xl);
undefined

2.6. boolean

Representados por dos valores *true* y *false*. Debemos tener en cuenta las dos reglas siguientes:

- false, 0, un string vacío, NaN, null y undefined se representa como *false*
- El resto es true.

2.7. Identificando datos primitivos

La mejor formqa de identificar datos primitivos es usar el operador *typeof*:

```
console.log(typeof "Nicholas"); //"string"
console.log(typeof 10); //"number"
console.log(typeof 5.1); //"number"
console.log(typeof true); //"boolean"
console.log(typeof undefined); //"undefined"
```

3. Funciones en JavaScript

3.1. Introducción

- Las funciones son objetos en JavaScript
- Existen dos formas de declararlas:

  1. Como declaración:

     ```
     function add(num1, num2) {
       return num1 + num2;
     }
     ```

  2. Como expresión:

     ```
     var add = function(num1, num2) {
       return num1 + num2;
     };
     ```

Existe una diferencia exencial entre ambas declaraciones: el ámbito de la declaración, las cuales son elevadas al contexto superior del ámbito de la declaración. Es decir podemos declararlas en cualquier lugar y usarla sin problema alguno de visibilidad, cosa que no ocurre al delcararlas como epresión deben ser usadas posteriormente a la misma.
A estas dos formas podemos añadir la forma funcional:

```
var add = (num1, num2) =>
  num1 + num2;
```

3.2. Características de las funciones en JavaScript

- Como son objetos pueden asignarse a variables:

```
function sayHi() {
  console.log("Hi!");
}
sayHi(); // outputs "Hi!"
var sayHi2 = sayHi;
sayHi2();// outputs "Hi!"
```

- Se puede pasar cualquier número de argumentos, incluso posteriormente llamarla con el número de argumentos que queramos.

```
function sum() {
  var result = 0, i = 0, len = arguments.length;
    while (i < len) {
      result += arguments[i];
      i++;
   }
  return result;
}
console.log(sum(1, 2)); // devuelve 3
console.log(sum(3, 4, 5, 6)); devuelve 18
console.log(sum(50)); //devuelve 50
console.log(sum()); //devuelve 0
```

- Debido a lo anteriormente indicado no existe sobrecarga de funciones como ocurre en otros lenguajes. Se aplicaría la última definición:

```
function sayMessage(message) {
  console.log(message);
}
function sayMessage() {
  console.log("Default message");
}
sayMessage("Hello!"); // outputs "Default message"
```

3.3. Parámetros por defecto

Es buena praxis dar valores por defecto a los parámetros pasados a una función:

```
function sum(a,b){
  a = a || 0;
  b = b || 0;
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9)); //9
```

3.4. Métodos en JavaScript

En un objeto una propiedad puede ser una función, en este caso hablamos de *métodos*.

```
var person = {
  name: "Nicholas",
  sayName: function() {
    console.log(person.name);
  }
};
person.sayName(); // outputs "Nicholas"
```

Existe un fuerte acoplamiento entre el método *sayName* y el objeto mediante *person.name*, de entrada si cambiamos en una refactorización del código el nombre de la variable -en este caso *person*- debemos acordarnos de cambiar también *person.name*, también dificulta el uso de la función para otros objetos. JavaScript tiene una solución para esto, *this* representa al objeto que es llamado. Por lo que el código lo podemos cambiar por:

```
var person = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name);
  }
};
person.sayName();// outputs "Nicholas"
```

En otra refactorización, teniendo en cuenta que las funciones son objeto (referencias), podemos asignarlas a cualquier objeto, por lo que podemos desacoplarla completamente del objeto *person*, quedando el código:

```
function sayNameForAll() {
  console.log(this.name);
}
var person1 = {
  name: "Nicholas",
  sayName: sayNameForAll
};
var person2 = {
  name: "Greg",
  sayName: sayNameForAll
};
var name = "Michael";
person1.sayName();// outputs "Nicholas"
person2.sayName();// outputs "Greg"
sayNameForAll(); // outputs "Michael"
```

En el último caso la llamada *sayNameForAll()* origina *Michael* porque se define una variable global (var name = "Michael";) y al ser global es propiedad del objeto global.

3.5. Funciones como valores

Las funciones pueden ser asignadas a varibles.

```
function square(x) { return x*x; }
var s = square;// Now s refers to the same function that square
    does
square(4);// => 16
s(4);// => 16
```

3.6. Funciones y espacio de nombres

- En un programa JavaScript un nombre identifica un elemento y puede representar una variable, una función, una propiedad, ...
- En un programa debemos minimizar y estructurar los nombres utilizados
- *Espacio de nombres*: conjunto de nombres visibles en un lugar del programa:

  Espacio de nombres global: nombres visibles en el ámbito global

  Espacio de nombres locales: nombres visibles en algún ámbito local
- Variables declaradas dentro de una función son visibles dentro de la función, incluso en funciones anidadas.
- Dichas variables no son visibles fuera de la función.
- Variables declaradas fuera de la función son visibles.
- A veces es útil declarar una función para actuar de forma temporal en un espacion de nombres y no contaminar el espacio de nombres globlal.

```
-GLOBAL SCOPE-------------------------------------------|
var g =0;                                                |
function foo(a) { ------------------------|              |
   var b = 1;                             |              |
   //code                                 |              |
   function bar() { ------|               |              |
      // ...              |               |              |
                          |ScopeBar       | ScopeFoo     |
   }                ------|               |              |
   // code                |               |              |
   var c = 2;             |               |              |
}-----------------------------------------|              |
foo(); //WORKS                                           |
bar(); //FAILS                                           |
---------------------------------------------------------|
```

Un uso corrrecto de los espacio de nombres hará que nuestro código sea mas correcto y facilita la detección de errores.

```
var a = 1;
//Lets introduce a function -scope
//1. Add a named function foo() into the global scope
function foo() {
  var a = 2;
  console.log( a ); // 2
}
//2. Now call the named function foo()
foo();
console.log( a ); // 1
```

La función *foo* se crea como ámbito global y posteriormente se ejecuta. Este código es un poco enfarragoso. Quedaría mejor:

```
var a = 1;
//Lets introduce a function -scope
//1. Add a named function foo() into the global scope
(function foo() {
  var a = 2;
  console.log( a ); // 2
})(); //<---this function executes immediately
console.log( a ); // 1
```

Este patrón se denomina *IIFE* (Immediately Invoked Function Expression.)

```
(function foo(){ /* code */ })();
```

3.6.1. Uso de let en ECMAScript 6

JavaScript es un lenguaje bastante flexible, sobre todo con el uso y declaración de variables, que pueden contener cualquier tipo de dato y mutar el tipo sin ningún problema, además de poder ser declaradas de diferentes formas y pasar a ser globales o locales según como se haga (lo que puede llevar a errores básicos de programación por no declarar correctamente una variable).
Además, las únicas clausuras para las variables se crean solo y exclusivamente con funciones, como no pasa en otros lenguajes como por ejemplo Java:

```
public static void main(String[] args) {
    int a = 1;
    int b = 2;

    {
        int c = 3;
    }

    System.out.println(a +" "+ b +" "+ c);
    // Error de compilacion: c cannot be resolved to a variable
}
```

En cambio en JavaScript:

```
var a = 1;
var b = 2;

{
    var c = 3;
}

console.info(a, b, c); // 1 2 3
```

Y esto se puede trasladar a bucles o condiciones, donde podemos declarar variables temporales y serán visibles desde fuera de esa sentencia.

Por suerte, esta nueva versión de JavaScript incorpora let para declarar variables y controlar mejor su propagación. En principio es bastante simple de entender, simplemente evita que una variable sea visible cuando no debe como ocurre con var.

```
let a = 1;
let b = 2;

{
    let c = 3;
}

console.info(a, b, c); // ReferenceError: c is not defined
```

Como vemos, declarar con var algunas variables temporales podría alterar la ejecución del código cuando realmente esas variables no se usan para ningún fin, más allá de poder hacer un bucle

.

```
var i = "i";
var temporal = "temporal";

for(let i = 1; i != false; i = false){
    let temporal = 10;
}
console.log(i, temporal); // i temporal


for(var i = 1; i != false; i = false){
    var temporal = 10;
}
console.log(i, temporal); // false 10
```

3.6.2. Constantes en JavaScript con const

Las constantes en JavaScript siempre se las ha echado en falta y nos hemos visto obligados a crear pseudo-constantes simplemente estableciendo el nombre de la variable en mayúscula (var PSEUDO_CONSTANTE) y dando por hecho que nadie la modificará posteriormente, algo que se podía hacer sin problemas en JavaScript, y ahora como debe de ocurrir en una constante, ya no variará el propio valor por mucho que se intente modificar.

```
// x: constante
const x = 20;
console.log(x); // 20

x = 1;
console.log(x); // 20
// No cambia el valor de x
```

3.7. Clousures

- Función que encapsula un conjunto de definiciones locales.
- Que solo son accesibles a través del objeto interfaz retornado por dicha función
- Las variables y funciones locales no se pueden acceder ni ver desde el exterior de una función.
- *¡OJO!* Un cierre no instancia sus variables hasta que no se ejecuta (invocar función).

```
function cierre(...){
  //variables internas
  var variableLocal;
  ......
  //funciones internas
  function funcionLocal(..) {...};
  ......
  //interfaz exterior
  return parametrosRetorno
```

- La interfaz de un cierre con el exterior es el parámetro de retorno de la función.
- Suele ser un objeto JavaScript que da acceso a las variables y funciones locales.
- **Las variables locales de un cierre siguen existiendo si existen referencias a ellas.**
- Aunque la función que las engloba haya finalizado su ejecución.
- Un cierre permite crear un objeto independiente cada vez que se invoca *(factoría de objetos)*.
- Cada objeto tiene sus propias variables y funciones internas, y su propio interfaz de acceso.

```
var scope = "global scope"; // A global variable
function checkscope() {
  var scope = "local scope"; // A local variable
  function f() { return scope; } // Return the value in scope here
  return f();
}
checkscope(); // => "local scope"
```

De otra manera:

```
var scope = "global scope"; // A global variable
function checkscope() {
  var scope = "local scope"; // A local variable
  function f() { return scope; } // Return the value in scope here
  return f;
}
checkscope()(); // => "local scope"
```

Este último ejemplo es interesante porque:

- Movemos un par de paréntesis desde el interior a fuera.
- Antes se devolvía el valor de la funcion *f()* ahora la funcion *f*
- Invocamos al valor de la función fuera del ámbito de la función con el segundo paréntesis de *checkscope()()*
- El valor que devuelve f() sigue existiendo aunque previamente se haya ejecutado la llamada a la clousure *checkscope()*

Otro ejemplo:

```
var uniqueInteger = (function() { // Define and invoke
  var counter = 0; // Private state of function below
  return function() { return counter++; };
})();
console.log(uniqueInteger()); //print 0
console.log(uniqueInteger()); //print 1
console.log(uniqueInteger()); //print 2
```

Podemos devolver en la interfaz de retorno mas de un único valor:

```
function counter() {
  var n = 0;
  return {
    count: function() { return n++; },
    reset: function() { n = 0; }
  };
}
var c = counter(), d = counter(); //Create two counters
console.log(c.count()); //=>0
console.log(d.count()); //=>0 they count independently
c.reset(); // reset() and count() methods share state
console.log(c.count()); // => 0: because we reset c
console.log(d.count()); // => => 1: d was not reset
```

Podemos usar los cierres para crear *getters y setters*

```
function counter(n) { // Function argument n is the private
    variable
  return {
  // Property getter method returns and increments private counter
      var.
    get count() { return n++; },
  // Property setter doesn't allow the value of n to decrease
    set count(m) {
      if (m >= n) n = m;
      else throw Error("count can only be set to a larger value");
    }
  };
}
var c = counter(1000);
console.log(c.count); // => 1000
console.log(c.count); // => 1001
c.count = 2000;
console.log(c.count); // => 2000
c.count = 2000 // => Error!*/
```

Los *cierres* se usan para crear objetos:

```
function contador(inic){
  var _cont = inic; //variable interna

  function contador() { return _cont };
  function incr() { return ++_cont };

  return { contador: contador,
           incr:     incr
         };
}

var contador_1 = contador(0);
var contador_2 = contador(7);

console.log(contador_1.contador()); // => 0
contador_1.incr();
console.log(contador_1.contador()); // => 1


console.log(contador_2.contador()); // => 7
contador_2.incr();
console.log(contador_2.contador()); // => 8
```

- La función *contador* permite crear objetos *contador*
- El parámetro de esta función inicializa el contador (variable _cont)
- _cont utiliza el convenio de comenzar por "_" las definiciones no visibles fuera (muy habitual)

Un programa semejante pero usando un patrón diferente:

```
function contador(inic){
  var _cont = inic; //variable interna

  return { contador: function() { return _cont },
           incr:     function() { return ++_cont}
         };
}

var contador_1 = contador(0);
var contador_2 = contador(7);

console.log(contador_1.contador()); // => 0
contador_1.incr();
console.log(contador_1.contador()); // => 1


console.log(contador_2.contador()); // => 7
contador_2.incr();
console.log(contador_2.contador()); // => 8
```

- Hay menos código pero menos legible.
- En el interfaz los objetos de retorno se construyen.

4. Ejercicios

4.1. Ejercicio 1

Realiza un programa en JavaScript que usando *clousures* y dado una cadena de texto:

- Nos diga si la cadena es un número o no.
- La cadena al revés.
- El número de caracteres que tiene la cadena.
- La cadena en mayúscula.
- Si es o no un palíndromo.

4.2. Ejercicio 2

Realiza un programa en JavaScript que usando *clousures* y dado un NIF/DNI

- Nos diga si es correcto o no el formato DNI/NIF (que contenga 8 números u 8 números mas una letra)
- Nos diga si hemos introducido un DNI o NIF
- En el caso de un DNI nos de la letra
- En el caso de un NIF nos valide la letra.

5. Referencias en JavaScript

5.1. Introducción

- Una referencia de tipos son objetos.
- Un objetos es una lista no ordenada de propiedades consistente en un nombre y un valor.
- Cuando el valor de una propiedad es una función se denomina método.

5.2. Creando objetos

- Podemos crear objetos creandolos o instanciándolos.
- Cuando los creamos usamos *new* con un *constructor*
- Un *constructor* es una función cualquiera.
- Por convención se usa el constructor con la primera letra en mayúscula.
- *var object = new Object();*
- Cuando se crea un objeto se almacena en memoria una referencia que apunta al objeto en sí.
- JavaScript utiliza un recolector de basura para gestionar la memoria.
- Pero es buena práctica *deferenciar objetos* haciéndolos *null*

5.3. Tipo de objetos

En cuanto los objetos pueden ser:

- Function
- Array
- Date
- RegExp
- Error

Podemos crearlas:

```
var items = new Array();
var now = new Date();
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

5.4. Objetos como literales

Podemos crear objetos usando literales, sin necesidad de usar el operador *new*

```
var book = {
  name: "The Principles of Object-Oriented JavaScript",
  year: 2014
};
```

Este código es equivalente a:

```
var book = new Object();
book.name = "The Principles of Object-Oriented JavaScript";
book.year = 2014;
```

Ejemplo usando objetos *Array*:

```
var colors = [ "red", "blue", "green" ];
//equivalente a:
var colors = new Array("red", "blue", "green")
```

En el caso de las funciones:

```
function reflect(value) {
  return value;
}
// es igual a
var reflect = new Function("value", "return value;");
```

Para el caso de expresiones regulares:

```
var numbers = /\d+/g;
//equivalente a:
var numbers = new RegExp("\\d+", "g");
```

5.5. Arrays

5.5.1. Bucle forEach

Podemos iterar sobre un array de distinta forma:

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++)
  console.log(colors[i]);
```

Y de otra manera:

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
```

La función pasada a *forEach()* es ejecutada para cada item del array

5.5.2. Métodos de los arrays

Método **concat()**

```
var myArray = new Array("33", "44", "55");
myArray = myArray.concat("3", "2", "1");
console.log(myArray);
// ["33", "44", "55", "3", "2", "1"]
```

Método **join()**

```
var myArray = new Array('Red','Blue','Yellow');
var list = myArray.join(" ~ ");
console.log(list);
//"Red ~ Blue ~ Yellow"
```

Método **pop()**

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop();
// myArray = ["1", "2"], last = "3"
```

Método **push()**

```
var myArray = new Array("1", "2");
myArray.push("3");
// myArray = ["1", "2", "3"]
```

Método **shift()**

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift();
// myArray = ["2", "3"], first = "1"
```

Método **unshift()**

```
var myArray = new Array ("1", "2", "3");
myArray.unshift("4", "5");
// myArray = ["4", "5", "1", "2", "3"]
```

Método **reverse()**

```
var myArray = new Array ("1", "2", "3");
myArray.reverse();
// transposes the array so that myArray = [ "3", "2", "1" ]
```

Método **sort()**

```
var myArray = new Array("A", "C", "B");
myArray.sort();
// sorts the array so that myArray = [ "A","B","C" ]
```

5.6. Propiedades de los objetos

- Podemos añadir propiedades a los objetos en cualquier momento.
- Creamos el objeto *var objeto = new Object();*
- Añadimos propiedades: *objeto.nombre ='mi nombre';*

- Son pares clave/valor que se almacena en el objeto.
- Podemos acceder a los valores usando la notación de punto.
- Usando la notación de corchetes.
- Ejemplo:

```
var array = [];
array.push(12345);
//equivalente
var array = [];
array["push"](12345);
```

Otro ejemplo:

```
var x = {titulo: 'Avatar', director: 'James Cameron'}
x.titulo; => 'Avatar'
x["titulo"]; => 'Avatar'
```

5.7. Constructores

- Un constructor es una simple función que nos sirve para crear objetos usando el operador *new*.
- Ya hemos vistos constructores tales como *Object* , *Array* , *y Function*
- La ventaja de usar un constructor es que todos los objetos tienen las mismas propiedades.
- El constructor es una simple función pero por convenio su primera letra va en mayúscula.

Ejemplo:

```
function Person() {
// intentionally empty
}
//creamos objetos:
var person1 = new Person();
var person2 = new Person();
//podemos omitir el parentesis:
var person1 = new Person;
var person2 = new Person;
```

Introduciendo propiedades y métodos:

```
function Person(name) {
  this.name = name;
  this.sayName = function() {
    console.log(this.name);
  };
}
```

5.8. Prototipos

- JavaScript es un lenguaje basado en *prototipos*
- La herencia en JavaScript se basa en prototipos.
- Todo objeto JavaScript posee un prototipo del cual hereda sus propiedades y métodos.
- El prototipo es un objeto como otro cualquiera al que se pueden añadir o quitar propiedades y métodos.
- **Una modificación del prototipo trascenderá a todos los objetos asociados.**
- Si borramos un método, este ya no se podrá invocar sobre los objetos enlazados.
- Si añadimos un método este se podrá invocar sobre todos los objetos asociados al prototipo.
- Todos los objetos creados como literales pertenecen *Object.prototype*
- Podemos saberlo con *Object.getPrototypeOf(obj)*

```
var objeto = {nombre : 'manuel'};
Object.getPrototypeOf(objeto); //=> {}
//podemos incluirir una propiedad a todos los objetos
Object.prototype.x=7;
objeto.x; //=>7
var otro = {};
otro.x; //=>7
delete Object.prototype.x;; //borra la propiedad x del Object.
    prototype
objeto.x; //=>undefined NO mantiene la propiedad
```

Ejemplo usando un constructor (recuerda el constructor se escribe en mayúscula)

```
function Persona(nombre, apellidos){
  this.nombre    = nombre;
  this.apellidos = apellidos;
}

Persona.prototype = {
  nombrar: function() { return this.nombre +
                        ' '+ this.apellidos; }
}

Persona.prototype.nombrar_oficialmente = function(){
  return this.apellidos + ', '+ this.nombre;
}

var persona_1 = new Persona("manuel", "garcia garcia");
var persona_2 = new Persona("luisa", "morales moral");

console.log(persona_1.nombrar()); //=>manuel garcia garcia
console.log(persona_2.nombrar()); //=>luisa morales moral
console.log(persona_1.nombrar_oficialmente()); //=>garcia garcia,
    manuel
console.log(persona_2.nombrar_oficialmente()); //=>morales moral,
    luisa
```

5.9. Herencia en JavaScript

Usando *prototipos* podemos hacer herencia:

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}

var persona = new Person();
persona.cry(); //Crying

function Child() {}
Child.prototype = new Person();
var aChild = new Child();
aChild.cry();//Crying

console.log(aChild instanceof Child); //true
console.log(aChild instanceof Person); //true
console.log(aChild instanceof Object); //true

Child.prototype.bebe = function(){
  console.log('Soy un bebe');
}
aChild.bebe(); //soy un bebe
// persona.bebe(); no es propiedad de Person

Person.prototype.esHumano = function(){
  console.log('Soy un humano');
}

persona.esHumano(); //Soy un humano
aChild.esHumano();  //Soy un humano
```

5.10. Serialización de objeto en JavaScript

- La serialización de objetos es el proceso de convertir un objeto en un *string* el cuál posteriormente podrá ser restaurado.
- Para serializar un objeto usamos **JSON.stringify()**
- Para restaurar el estado usamos **JSON.parse()**
- La serialización es un formato de intercambio de datos:
  - Almacenar datos en un fichero.
  - Enviar datos a través de una línea de comunicación.
  - Paso de parámetros en interfaces *REST*.

```
o = {x:1, y:{z:[false,null,""]}}; // Define a test object
s = JSON.stringify(o);// s is '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s);// p is a deep copy of o
```

- JavaScript no serializa todos los tipos de datos.

- Pueden serializarse y restaurarse: Objectos, arrays, strings, números finitos, true, false, y null
- NaN , Infinity , y -Infinity son serializados a null.
- Function, RegExp, y Error (objecto) y el valora undefined no pueden ser serializados.
- JSON.stringify() serializar las propiedades susceptibles de ser serializadas, si el valor de una propiedad no es serializable se omite en la salida de este método.

5.11. Ejercicios

Realiza los ejercicios anteriores propuestos (Ejercicio 1 y Ejercicio 2) y añade una nueva función que serializa los objetos, y esta vez usando constructores y prototipos.