JavaScript - The Good Parts (Resumen)

Carolina Jiménez Gómez

Julio-Agosto 2016

1 Contexto

Se toma de referencia el libro ***JavaScript: The Good Parts***[4] y las páginas web ***Everything you wanted to know about JavaScript scope***[3], ***Advanced Javascript: Objects, Arrays, and Array-Like Objects***[1], ***A Beginner's Guide to Currying in Functional JavaScript***[2] entre otras muchas lecturas/búsquedas para la realización de éste informe/resumen sobre JavaScript.

1.1 ¿Qué es?

JavaScript es un lenguaje de programación interpretado, es orientado a objetos, basado en prototipos, imperativo, debilmente tipado y dinámico.

1.2 ¿Dónde se utiliza?

- Páginas web del lado del cliente para hacerlas dinámicas
- Páginas web del lado del servidor
- Aplicaciones de escritorio
- Documentos PDF

1.3 ¿Dónde nace?

JavaScript fue diseñado para añadir interactividad a páginas HTML.

El lenguaje fue inventado por Brendan Eich de Netscape (con Navigator 2.0).

Por el desarrollo por parte de Microsoft de un dialecto compatible del lenguaje JavaScript el cual llamaron JScript, se hizo una estandarización del lenguaje, el cual recibe el nombre de ECMAScript.[1].

[1] https://es.wikipedia.org/wiki/ECMAScript

ECMAScript es el estandar, y JavaScript es una implementación de éste estandar.
Otras implementaciones populares son ActionScript, el lenguaje que se usaba para programar Flash.

1.4 ¿Por qué JS?

- Javascript es el lenguaje de la web
- Un mismo lenguaje para cliente y servidor (node.js y js)
- Compatible con todos los navegadores
- Porque es un lenguaje fácil de aprender

2 Números:

Sólo existe un tipo: flotante de 64 bits. 1 y 1.0 son iguales. NaN es un valor numérico que es el resultado de una operación que no produce un resultado normal. Existe una función para detectar un NaN: isNaN(number) Ejempplo:

```
0/0 = NaN

Infinity/Infinity = NaN

isNaN(0/0) = true

Infinity = Number.POSITIVE_INFINITY

-Infinity = Number.NEGATIVE_INFINITY
```

2.1 numero.toFixed(digitos):

Convierte un número a un string con x dígitos decimales (por defecto es cero). *digitos* debe ser un número entre 0 y 20.

```
Math.PI.toFixed(); // "3"
Math.PI.toFixed(5); // "3.14159"
```

2.2 numero.toPrecision(precision):

Hace lo mismo que *toFixed* con la diferencia que *precision* toma valores entre 1 y 21.

2.3 numero.toString(base):

Convierte un número a string. *base* debe ir entre 2 y 36, y por defecto es 10.

```
(3).toString(); // "3"

(3).toString(2); // "11"

(3).toString(16); // 3
```

3 Falsy values:

Existen diferentes valores que se denominan **falsos** en JS, es decir, aquellos que en un condicional retornarían falso.

- false
- null
- undefined
- ”
- 0
- NaN

```
if (false) console.log("Esto no se cumple");
else console.log("Esto si se cumple");

if (null) console.log("Esto no se cumple");
else console.log("Esto si se cumple");

if (undefined) console.log("Esto no se cumple");
else console.log("Esto si se cumple");

if ('') console.log("Esto no se cumple");
else console.log("Esto si se cumple");

if (0) console.log("Esto no se cumple");
else console.log("Esto si se cumple");

if (NaN) console.log("Esto no se cumple");
else console.log("Esto si se cumple");
```

Todos los resultados del ejemplo anterior son ”Esto si se cumple”, ya que son valores que retornan falso.

Cualquier otro valor retornaría verdadero, hasta la cadena ”false”.

4 Strings

Se puede utilizar comillas simples o dobles para definir un string, En JS no existe el tipo caracter.

Los strings tienen una propiedad para saber su longitud: *length*.

Ejemplo:

```
"siete".length === 5
"".length === 0
```

Se pueden hacer concatenaciones de strings:

```
'H' + 'o' + 'l' + 'a' === 'Hola'
'H' + 'o' + 'l' + 'a' !== 'hola'
'H' + 'o' + 'l' + 'a' === "Hola"
'H' + 'o' + "l" + 'a' === 'Hola'
```

Todos los anteriormente mostrados retornan true.

Existen diferentes métodos de los strings, como por ejemplo pasar todo a mayúsculas:

```
'jointdev'.toUpperCase() === 'JOINTDEV'
```

4.1 string.charAt(pos):

Retorna el caracter en la posición *pos*. Si *pos* es menor a cero o mayor al tamaño del string, retorna una cadena vacía.

```
"Caro".charAt(3); // "o"
"Caro".charAt(4); // ""
```

4.2 string.charCodeAt(pos):

Hace lo mismo que *charAt* pero en vez de retornar un string, retorna el código ascii de ese caracter en esa posición *pos*.

4.3 string.indexOf(letra,pos):

Busca una *letra* a partir de una posición *pos*. Si no se define *pos*, por defecto es cero. Retorna la posición de la primer ocurrencia que encuentre. Si no encuentra *letra* en el string, retorna -1.

```
var txt = "Mississippi";

var p = txt.indexOf("ss"); // p = 2
p = txt.indexOf("ss",3); // p = 5
p = txt.indexOf("ss",6); // p = -1
```

4.4 string.lastIndexOf(letra,pos):

Hace lo mismo que *indexOf* pero inicia su recorrido de búsqueda del último elemento del string al primero.

```
var txt = "Mississippi";

var p = txt.lastIndexOf("ss"); // p = 5
```

4.5 string.slice(inicio,fin):

Crea un nuevo string copiando una porción de otro string. Si *inicio* es negativo, se le suma el tamaño del string. *fin* es opcional, por defecto es el tamaño del string.

```
var txt = "Carolina";

var a = txt.slice(2,5); // a = "rol"
```

4.6 string.split(separador,limite):

Crea un arreglo de strings partiendo el *string* en pedazos. *limite* es la cantidad de piezas que van a ser partidas, éste valor es opcional.

```
var c = "|a|b|c|".split('|');

// c = [ "", "a", "b", "c", "" ]

var d = "|a|b|c|".split('|',3);

// d = [ "", "a", "b" ]
```

4.7 string.toLowerCase():

Convierte un string a minúsculas.

```
var a = "JOINTDEV".toLowerCase(); // a = "jointdev"

var a = "JointDev".toLowerCase(); // a = "jointdev"
```

4.8 string.toUpperCase():

Convierte un string a mayúculas.

```
var a = "jointdev".toUpperCase(); // a = "JOINTDEV"

var a = "JointDev".toUpperCase(); // a = "JOINTDEV"
```

5 Objetos:

En JS todos los valores son OBJETOS, exceptuando los *números*, *strings*, booleanos(*true, false*), *null*, y *undefined*. [2]

Un *objeto* es un contenedor de propiedades, caracterizados por un *nombre* y un *valor*, donde el nombre puede ser cualquier string (incluido el ""), y su valor puede ser cualquiera, excluyendo *undefined*.

Para crear un objeto, vale con abrir y cerrar llaves .

```
var obj = {};
```

Un objeto puede contener más objetos, o inclusive funciones.

```
var jointdev = {nombre: "JointDeveloper",
           ciudad: "Pereira",
           integrantes: [
               {nombre: "Pepita",
                apellido: "Perez"
               },
               {nombre: "Juanito",
                apellido: "Alimaña"
               },
            ]
           };

var saludar = {
                saludo: function(nombre){
                    console.log("Hola "+nombre);
                }

              };

saludar.saludo("JointDevelopers!");
var integrantes = jointdev.integrantes;
var integrante1 = integrantes[0]; //Pepita Perez
```

[2] Los valores descritos son *object-like, ya que tienen métodos, pero son inmutables*

En los ejemplos anteriores podemos ver el poder de los objetos de javascript para almacenar casi cualquier cosa. En el primer objeto *jointdev* tenemos almacenado todos los datos de JointDeveloper, con cada uno de sus integrantes y practicamente cualquier información adicional que queramos agregar de ellos. En el segundo objeto *saludar* tenemos almacenados una función, a la cual se puede hacer uso con el correcto llamado.

5.1 Actualización:

También se pueden agregar cosas a los objetos que ya han sido creados, de esta manera:

```
obj.eventos = ["jquery","html+css","javascript"];
```

Hemos agregado un array de strings al nombre *eventos* de *jointdev*.

También podemos modificar los objetos, así:

```
integrante1.nombre = "Pepita María";
```

integrante1 devuelve el objeto del array, donde está contenida la información de Pepita.

5.2 Referencias:

Se debe tener mucho cuidado a la hora de declarar objetos e igualarlos a alguno ya existente, ya que no se hace una copia si no que ambos objetos referencian el mismo, si modifico uno, el otro también cambiará.

```
var obj = jointdev;

jointdev.nombre; //JointDeveloper

obj.nombre = "jointdev";

obj.nombre; //jointdev
jointdev.nombre; //jointdev
```

5.3 Prototipos:

Todos los objetos de JS están vinculados a un objeto prototipo del cual pueden heredar sus propiedades: Object.prototype.

5.4 Creación de prototipos:

La forma de crear objetos prototipos es de la siguiente manera:

```
function Persona(nombre, apellido, semestre) {
    this.nombre = nombre;
    this.apellido = apellido;
    this.semestre = semestre;

    this.muestraInfo = function (){
        console.log("Nombre : "+ nombre);
        console.log("Apellido : "+ apellido);
        console.log("Semestre : "+ semestre);
    }
}
```

Persona se convierte en nuestra función constructora del objeto.
Se pueden crear objetos a partir del prototipo *Persona*, y éste herdará todos sus atributos y métodos.

```
var Carolina = new Persona("Carolina","Jiménez",7);

var Pepe = new Persona("Pepe","Perez",1);

Carolina.muestraInfo();
Pepe.muestraInfo();

console.log(Carolina instanceof Persona); // Significa que Carolina heredó de Persona

console.log(Persona instanceof Object);
```

En el ejemplo anterior vemos el funcionamiento de la herencia en JS y la forma de acceder a los atributos y métodos de los objetos creados.

Éste método no es recomendado en el libro [4], ya que si a la hora de crear el objeto no se pone el prefijo *new*, entonces *this* no estará ligado al nuevo objeto, si no, al objeto global.

5.5 Añadiendo atributos y métodos:

Se pueden añadir atributos y métodos tanto al objeto creado como al objeto prototipo.

Para agregar un atributo o método a un único objeto que ha heredado del objeto prototipo se hace de la siguiente manera:

```
Carolina.edad = 20;
```

```
Carolina.edad; // 20
Pepe.edad; // undefined
```

De esta manera sólo se ve afectado el objeto **Carolina**, y no todos los objetos que heredan de *Persona*.

Para que todos los objetos tengan los mismos atributos y métodos se debe modificar su padre, por así decirlo, ya que sus hijos heredan de él.

```
Persona.prototype.saluda = function (){
    console.log("Hola, me llamo " + this.nombre);
};

Carolina.saluda();
Pepe.saluda();
```

También se pueden modificar los objetos estándar de JS como son arrays, Date, RegExp, function, etc.

5.6 Reflejo:

Otra forma de crear objetos es especificando cuál va a ser su prototipo, de la siguiente manera:

```
var Persona = {nombre: "",
               apellido: "",
               edad: 0
              };

var Carolina = Object.create(Persona);

Carolina.edad; // 0
Carolina.edad = 20;
Carolina.edad; // 20
Persona.edad; // 0
Persona.edad = 1;
Carolina.edad; // 20
Carolina.semestre; // undefined
Persona.semestre = 0;
Carolina.semestre; // 0
```

De esta forma, heredamos todos los métodos y variables del objeto *Persona* al objeto *Carolina*. En el ejemplo anterior se puede ver que al modificar *Carolina* no se modifica *Persona*, pero si agregamos un nuevo atributo a *Persona*, automáticamente los objetos que lo utilizan como prototipo adquieren este mismo atributo.

Por el momento, si vemos el objeto ***Persona***, vemos que sus tributos son: *nombre, apellido, edad, semestre*, ya que así lo hemos ido definiendo, pero si miramos el objeto ***Carolina***, veremos que solo aparece con el tributo de *edad*, pues es el único que hemos fijado directamente al objeto; sin embargo, si accedemos a *Carolina.nombre* o a algún otro atributo que posea *Persona*, éste nos devolverá el atributo que tiene el objeto prototipo. Aquí se puede observar como se encuentran vinculados los objetos; a esto se le llama ***prototype chain***.

Si quisieramos saber si un objeto tiene su propio atributo definido, debemos hacer uso del método *hasOwnProperty* que ofrece JavaScript.

```
Carolina.hasOwnProperty('semestre'); // false
Persona.hasOwnProperty('semestre'); // true
Carolina.hasOwnProperty('edad'); // true
Persona.hasOwnProperty('edad'); // true
```

5.7 Enumeración:

Se puede recorrer los atributos que contiene un objeto de la siguiente manera:

```
for (var atr in Persona) {
    console.log(atr + " : " + Persona[atr]);
}
```

También podráimos filtrar resultados:

```
for (var atr in Carolina) {
    if (Carolina.hasOwnProperty(atr)) {
        console.log(atr + " : " + Carolina[atr]);
    }
    else {
        console.log("No tiene a " + atr);
    }
}
```

5.8 Removiendo atributos:

Existe un método para remover un atributo de un objeto, éste es *delete*.

```
delete Carolina.edad;
Carolina.edad; // 1
Carolina.hasOwnProperty('edad'); // false
Persona.edad; // 1

Carolina.semestre; // 0
Persona.semestre; // 0
delete Persona.semestre;
Persona.semestre; // undefined
Carolina.semestre; // undefined
```

5.9 Evitando las variables globales:

Para evitar la mala interacción con otras aplicaciones, widgets y librerías, es recomendable utilizar una única variable que contenga toda la aplicación, de esta forma:

```
var MYAPP = MYAPP || {}; // Crearla de forma segura, por si ya existe

MYAPP.Persona = {nombre: "",
                 apellido: "",
                 edad: 0
                };
MYAPP.Carolina = Object.create(Persona);
```

Y de ésta manera se haría el llamado a cada uno de los objetos:

```
MYAPP.Persona;
MYAPP.Carolina;
MYAPP.Persona.nombre; // ""
```

El problema de las variables globales en JS es por el uso que se tiene con los espacios de nombre (*namespace*), ya que todas las variables globales comparten el mismo espacio de nombre, lo que puede producir fallas.[3]

6 Funciones:

Como ya habíamos mencionado, todo en JS son objetos, inclusive las funciones. Ésta es la razón por la cual podemos almacenar funciones en variables, objetos y arreglos. Se pueden enviar funciones como argumentos a otras funciones, y una función puede retornar otra. Además pueden contener métodos.

Existen dos formas de declarar una función: **declarativas** y de **expresión**.

```
function f(){} // declarativa

var fun = function f(){}; // de expresión
```

La diferencia entre ambas radica en que la primera forma parte de un programa global, y son evaluadas antes que cualquier otra expresión. La segunda es una expresión que forma parte de un punto concreto, y es evaluada en ese punto en concreto.

```
console.log( suma( 3, 5 ) ); // 8

function suma( x, y ){
  return x + y;
}
```

[3] https://www.kenneth-truyers.net/2013/04/27/javascript-namespaces-and-modules/

En el ejemplo anterior no importa que el llamado a la función se haga antes de su declaración, ya que la función tiene preferencia sobre las expresiones que le precedan.

```
alert( suma( 3, 5 ) ); // ErrorType: add is not defined

var suma = function ( x, y ){
  return x + y;
}
```

Las funciones están vinculadas a *Function.prototype*, el cual a sí mismo está vinculado a *Object.prototype*.

A parte de los parámetros definidos por una función, éste recibe siempre *this* y *arguments*.
No se muestra mensaje de error si la cantidad de parámetros enviados a una función no coincide con las esperadas. Si se envían más parámetros de los esperados, éstos se ignoran; si por el contrario, se envían menos parámetros de los esperados, el valor *undefined* será sustituido en estos valores.

```
var miFuncion = function(a,b){
  console.log(a,b,arguments[0],arguments[1],arguments[2]);
}

miFuncion(1,2); // 1 2 1 2 undefined
miFuncion(1,2,3); // 1 2 1 2 3
```

6.1 Scopes:

Scopes se refiere al alcance de las variables en cada función o en todo el código. Pueden ser definidas localmente o globalmente.

```
//variable global
var nombre = "Carolina";

var miFuncion = function(){
    //variable local
    var nombre = "Carolina";
}
```

Las variables locales no pueden ser alcanzadas por la globales.

```
var miFuncion = function () {
  var nombre2 = 'Pepe';
  console.log(nombre2);
};

console.log(nombre2);
```

En el ejemplo anterior, cuando trato de acceder a *nombre* el intérprete me dice que está indefinido, pero si llamo directamente a la función, éste me mostrará en pantalla el valor de la variable *nombre*.

6.2 This:

This hace referencia al objeto que hace el llamado, y éste valor cambia dependiendo del scope al que estamos sometidos.

```
var miFuncion = function () {
  console.log("Retorna : " , this);
};
miFuncion();

var obj = {};
obj.metodo1 = function () {
  console.log("Retorna : ", this);
};
obj.metodo1();
```

El problema se presenta cuando queremos hacer referencia al mismo objeto del padre en las funciones hijas, pero cuando cambiamos de scope se pierde dicha referencia:

```
obj.metodo2 = function () {
  console.log("Retorna : ",this);
  setTimeout(function () {
    console.log("El otro retorna : ",this);
  }, 1000);
};
obj.metodo2();
```

Esto sucede porque el que hizo el llamado a la función deja de ser *obj*, por ende, cuando imprimimos *this* no hace referencia a él, si no, de nuevo a *window*.

Para referenciar al mismo objeto dentro de la misma función padre, guardamos *this* en una variable, y problema solucionado.

```
obj.metodo3 = function () {
  console.log("Retorna : ",this);
  var that = this;
  setTimeout(function () {
    console.log("El otro retorna : ",that);
  }, 1000);
};
obj.metodo3();
```

6.3 Closures:

Cuando hablamos de *Closures* nos referimos al alcance que tienen las funciones internas con respecto a sus propias variables locales y a las que están por fuera de ellas.

Ejemplo:

```
// Scope A
var x = 5;

var miFuncion = function () {

  // Scope B
  var nombre = 'Caro';

  //Desde aquí podemos acceder a la variable 'x'
  //y a la variable 'nombre'

  var otraFuncion = function () {
    // Scope C
    //Desde aquí podemos acceder a variables
    //definidas dentro de ésta función y a
    //la variable definida por fuera 'nombre' y 'x'
  };

};
```

Podríamos decir que todos los "hijos" pueden acceder a las variables de sus "padres", pero no en sentido contrario.

Veamos otro ejemplo:

```
var scope1 = function () {
  var nombre = 'JointDeveloper';

  var scope2 = function () {
    console.log('Quiero ser un@ ' + nombre + '!');
  };

  console.log(nombre);
  scope2(); //Llamado a la función
};

scope1();//Llamado a la función
```

Otro ejemplo:

```
var Saludar = function (nombre) {
  var decir = 'Hola, ' + nombre;
  return function () {
    console.log(decir);
  };
};

var HolaJointDevs = Saludar("JointDeveloper");
HolaJointDevs();
```

Como podemos ver en el ejemplo anterior, **Saludar** retorna una función, es por esto que la almacenamos en una variable para después hacer el llamado con **HolaJointDevs**.

Entonces, **¿para qué sirven los *Closures*?**

Los Closures son utilizados para el encapsulamiento, ya que en JS no existe otra forma de hacer funciones, objetos o variables privados.

```
(function () {
  var miFuncion = function () {
    console.log("Hola JointDevs!!");
  };
})();

miFuncion(); // Llamado a la función
```

En el ejemplo anterior, utilizamos una "envoltura" para la declaración de *miFuncion*, pero a la hora de hacer el llamado, nos muestra el error: ***ReferenceError: miFuncion is not defined***, esto es porque es una función privada, únicamente podríamos hacer el llamado dentro de la envoltura.

```
(function () {
  var miFuncion = function () {
    console.log("Hola JointDevs!!");
  };

  miFuncion(); // Llamado a la función

})();
```

Entonces, ¿cómo manejar funciones privadas y públicas?

```
var ppal = (function () {
```

```
  var x = 10; // Privado
  var nombre = "JointDeveloper"; // Privado

  var saludar = function (nombre) { // Privado
    console.log("Hola "+nombre)
  };

  var getX = function () { // Público
    return x;
  };

  var getInfo = function () { // Público
    saludar(nombre);
  };

  return {
    muestraX: getX,
    muestraInfo: getInfo
  }

})();

var todo = ppal; // Almacenamos el objeto en una variable
todo.x; // undefined
todo.muestraX(); // 10
todo.muestraInfo(); // Hola JointDeveloper
```

De esta forma, podremos acceder únicamente a las funciones que retornamos de *ppal: muestraX* y *muestraInfo*.

Nótese que ppal retorna el llamado de una función, no la función como tal.

6.4 Excepciones:

```
var add = function (a,b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
        throw {
            name: 'TypeError',
            message: 'add need numbers'
        };
    }

    return a + b;
}

var try_it = function () {
```

```
    try {
        add("seven");
    } catch (e) {
        console.log(e.name + ' : ' + e.message);
    }
};

try_it(); // TypeError : add need numbers
```

6.5 function.apply(thisArg,argArray):

El método *apply* invoca una función *function* pasándole el objeto que va a estar ligado a *this* y un arreglo de argumentso *argArray* opcional.

6.6 Método *call*:

Es el mismo que *apply*, pero en vez de recibir un arreglo de argumentos, los recibe tal cual.

6.7 Cascadas:

Se pueden hacer llamados consecutivos de métodos al mismo objeto si cada uno de éstos métodos retornan *this*.

```
var Calculo = (function(){

    var res = 0;

    var suma = function(){

        for(var i = 0; i < arguments.length; i++){
            if(typeof arguments[i] === 'number') res += arguments[i];
        }
        return this;
    };

    var resta = function(){

        for(var i = 0; i < arguments.length; i++){
            if(typeof arguments[i] === 'number') res -= arguments[i];
        }
        return this;
    };

    var getN = function(){ console.log(res); };
```

```
    return {
        suma: suma,
        resta: resta,
        getN: getN
    }
})();

Calculo.getN(); // 0
Calculo.suma(10);
Calculo.getN(); // 10
Calculo.suma(10,10).resta(10);
Calculo.getN(); // 20
```

Se le llama **cascada** al verse de esta forma:

```
Calculo
       .suma(10)
       .suma(10,10)
       .getN(); // 20
```

6.8 Curry:

Es una forma de construir funciones, nos permite crear nuevas funciones combinando una función y un argumento.

```
var saludo = function(mensaje){
    return function(nombre){
        console.log(mensaje + " " + nombre);
    };
};

var saludar1 = saludo("Hola");
var saludar2 = saludo("Hola")("JointDev"); // Hola JointDev
saludar1("JointDev"); // Hola JointDev
saludar1("Jaime"); // Hola Jaime
```

La función ***saludo*** retorna una función la cual espera recibir su propio argumento; entonces, ***saludar1*** es una variable con la función que retornó ***saludo*** con el mensaje **Hola**, y proseguimos con la invocación de dicha función con el argumento **JointDev**.

Podemos tener una functión que retorne otra, y esta a su vez retorne otra, y así sucesivamente. Nosotros elegimos la manera como manejar dicho retorno de funciones.

```
var saludoEstructurado = function(mensaje){
```

```
    return function(separador){

        return function(finalizador){

            return function(nombre){
                console.log(mensaje + separador + nombre + finalizador);
            };

        };

    };
};

saludo1 = saludoEstructurado("Hola")(": ")(".");
destinatario = saludo1("JointDevs"); // Hola: JointDevs.

saludoEstructurado("Hola")(": ")(".")("Jaime"); // Hola Jaime

saludo2 = saludoEstructurado("Hola")(": ");
destinatario2 = saludo2(".")("JointDevs"); // Hola: JointDevs.
```

Otro ejemplo de utilidad de éste método es el siguiente:

```
var suma = function(){
    var args1 = arguments;
    var suma = 0;

    for (var i = 0; i < args1.length; i++) {
        if (typeof args1[i] == 'number') suma += args1[i];
    }
    console.log("suma : " + suma);

    return function (){
        var args2 = arguments;
        for (var i = 0; i < args2.length; i++) {
            if (typeof args2[i] == 'number') suma += args2[i];
        }
        console.log("suma : " + suma);
        return suma;
    };
};

var suma1 = suma(1,2,3); // 6
var suma2 = suma1(4,5,6); // 6 + 15 = 21
```

Nótese que el resultado se va guardando puesto que hacemos referencia a la misma variable *suma*.

7 Herencia:

La herencia es un mecanismo para la reutilización y extensibilidad del código. En lenguajes orientadas a objetos, los objetos son instancias de una clase, y una clase puede heredar de otra. JS es un lenguaje prototipado, lo que significa que los objetos pueden heredar directamente de otros objetos.

En JS existen muchas formas de herencia, como algunas vistas en las secciones pasadas con la creación de nuevos objetos y funciones constructoras; aquí mostraremos la *más apropiada*[4].

```
var persona = function(especificaciones) {
    var that = {}; // Creamos nuestro objeto que le vamos a insertar toda la información

    that.getName = function () {
        var nombre = especificaciones.nombre;
        return typeof nombre === 'string' ? nombre : "";
    };

    that.getAge = function () {
        var edad = especificaciones.edad;
        return typeof edad === 'number' ? edad : "";
    };

    return that;
};

var Carolina = persona({
    edad: 20,
    nombre: "Carolina"
});
```

Nótese que no importa el orden que se tenga en el objeto *especificaciones*.

Una vez tengamos nuestro objeto tal y como lo queremos, podemos empezar a crear nuevos objetos que hereden de él, únicamente agregando o cambiando atributos que queramos que tenga este nuevo objeto.

```
var estudiante = function (especificaciones) {
    var that = persona(especificaciones); // Un estudiante es una persona

    that.getSemester = function () {
        var semestre = especificaciones.semestre;
```

[4]Todo depende siempre del uso que se necesite del código, por ejemplo, ésta forma de crear objetos no tiene el problema del *new* y hace muy buen uso de la privacidad del objeto como tal.

```
        return typeof semestre === 'number' ? semestre : "Semestre indefinido";
    };

    that.getCareer = function () {
        var carrera = especificaciones.carrera;
        return typeof carrera === 'string' ? carrera : "Carrera indefinida";
    };

    return that; // Objeto con los nuevos métodos
};

var e1 = estudiante({
    nombre: "Pepe",
    edad: 18,
    semestre: 1,
    carrera: "Ingeniería en Sistemas y Computación"
});

e1.getName(); // Pepe
e1.getAge(); // 18
e1.getSemester(); // 1
e1.getCareer(); // "Ingeniería en Sistemas y Computación"
```

8 Arreglos:

Los arreglos en JS tienen el mismo funcionamiento que en otros lenguajes de programación.
Éstos heredan de *Array.prototype*, y es por esto que adquieren diferentes métodos.

Un arreglo puede contener cualquier tipo de valor, inclusive una mezcla de ellos:

```
var miArreglo = [
    1,"JointDev",
    function (a,b){ return a+b ; },
    {nombre: "Carolina", apellido: "Jiménez"}
    ];
```

Se acceden a los elementos de forma *Array[i]*, donde *i = 0 , 1, 2, ... , tamaño del arreglo -1* .

8.1 Tamaño del Arreglo:

Para conocer el tamaño de un arreglo *A*, se accede a la propiedad *length*.
Si almacenamos un elemento en una posición que no está definida en el arreglo, el tamaño de éste incrementará para contener este nuevo elemento.

```
var miArreglo = []; // Arreglo vacío

miArreglo.length; // 0

miArreglo[10] = 1;

miArreglo.length; // 11
```

Se puede acortar el tamaño de un arreglo de la siguiente manera:

```
var miArreglo = [1,2,3,4]; // Arreglo con 4 elementos

miArreglo.length = 2; // miArreglo = [1,2]
```

Se pueden agregar elementos al final del arreglo de la siguiente manera:

```
miArreglo[miArreglo.length] = 10; // miArreglo = [1,2,10]

miArreglo.push(20); // miArreglo = [1,2,10,20]
```

Para eliminar elementos de un arreglo se hace uso de *delete*:

```
delete miArreglo[0]; // miArreglo [ <1 empty slot>, 2, 10, 20 ];

miArreglo[0]; // undefined
```

Ésto deja un hueco en la posición donde estaba el elemento. Para solucionarlo, se utiliza otro método: *splice*:

```
miArreglo.splice(0,1); // miArreglo = [ 2, 10, 20 ]
```

Lo que hace es que remueve x elementos desde la posición i sin dejar huecos: Array.splice(i,x);

8.2 Matrices:

Los arreglos en JS son de una sola dimensión, así que si queremos crear una matriz, debemos inicializarla como un arreglo de arreglos.

```
var matriz = [
                [0],
                [1]
            ];

matriz[0][0]; // 0
```

Podemos crear un método que nos retorne una matriz inicializada con cierto valor:

```
Array.matriz = function (m, n, valor) {
    var A, res = [];
    for (var i = 0; i < m; i++) {
        A = [];
        for (var j = 0; j < n; j++) {
            A[j] = valor;
        }
        res[i] = A;
    }
    return res;
};

matriz = Array.matriz(2,2,0); // Matriz de 2

matriz[0][1]; // 0
```

8.3 array.concat(item...):

Concatena elementos a un arrego.

```
var a = [1,2,3];

var b = a.concat(4); // b = [1,2,3,4]

var c = b.concat(5,6,7,8); // c = [1,2,3,4,5,6,7,8]
```

8.4 array.join(separador):

Convierte un array en un string con un separador, por defecto es ','.

```
// a = [1,2,3]

var b = a.join(''); // "123"

var c = a.join(); // "1,2,3"
```

8.5 array.pop():

Remueve y retorna el último elemento en el arreglo. Si el arreglo está vacío, retorna *undefined*.

```
var b = a.pop(); // 3

// a = [1,2]
```

8.6 array.push(item...):

Agrega elementos al final de un arreglo. Retorna el tamaño del arreglo.

```
var matriz = [];

var a = [1,2,3];

var b = [4,5,6];

matriz.push(a); // 1

matriz.push(b); // 2

// matriz = [ [1,2,3] , [4,5,6] ]

matriz[0][1]; // 2
```

8.7 array.reverse():

Reversa el arreglo, retornándolo.

```
var a = [1,2,3];

var b = a.reverse();

// a = [3,2,1], b = [3,2,1]
```

8.8 array.slice(inicio,fin):

Hace una copia superficial de un pedazo de un arreglo. *fin* es opcional, por defecto es el tamaño del arreglo.

```
var a = [1,2,3];

var b = a.slice(0,1); // b = [1]
var c = a.slice(1); // c = [2,3]
var d = a.slice(1,2); // d = [2]
```

8.9 array.splice(inicio,cantidad,elemento):

Remueve elementos de un arreglo reemplazándolos con nuevos elementos. *cantidad* es la cantidad de elementos que van a ser reemplazados comenzando desde *inicio*, *elemento* son los nuevos elementos a insertar. Retorna los elementos removidos.

```
var matriz = [ [1,2,3] , [4,5,6] ];

var a = matriz.splice(0,1,[7,8,9]); // a = [1,2,3]

// matriz = [ [7,8,9], [4,5,6] ]
```

8.10 array.unshift(item...):

Agrega elementos a un arreglo pero posicionándolos al comienzo de éste.; "empuja" el resto de elementos a la derecha. Retorna el nuevo tamaño del arreglo.

```
var a = [1,2,3];

var b = a.unshift(4,5); // b = 5

// a  = [4,5,1,2,3]
```

9 Comparadores de igualdad

Usar SIEMPRE === y !== en lugar de == y !=, ya que los primeros hacen una comparación más profunda.

```
var x = 5;

x == 8 ; // false
x == 5; // true
x == '5'; // true

x === 5; // true
x === '5'; // false

x != 8; // true
x != "5"; // flase

x !== 8; // true
x !== "5"; // true
x !== 5; // true
```

References

[1] Advanced javascript: Objects, arrays, and array-like objects.

[2] A beginner's guide to currying in functional javascript.

[3] Everything you wanted to know about javascript scope.

[4] Douglas Crockford. *JavaScript: The Good Parts*. Copyright 2008 Yahoo! Inc.