1 Objetos en JavaScript

Los objetos son estructuras de información capaces de contener propiedades y métodos. **ECMA-262** define objeto como "una colección sin ordenar de propiedades que contienen un valor primitivo, un objeto o una función".

Todo objeto se define mediante una **clase**, que puede definirse como "la receta del objeto". Cuando un programa utiliza una clase para crear un objeto, el objeto resultante se denomina instancia de la clase.

- **Objetos nativos**

  Son cualquier objeto proporcionado por una implementación de JavaScript independiente del entorno anfitrión. Son los siguientes: **object, function, array, string, boolean, number, date, regexp, error, evalerror, rangeerror, referenceerror, syntaxerror, typeerror, urierror**.

- **Objetos incorporados**

  Son cualquier objeto proporcionado por una implementación de JavaScript, independientemente del entorno anfitrión, que está presente al iniciarse la ejecución de un programa de JavaScript. Son los siguientes: **global, math**.

- **Objetos anfitrión**

  Cualquier objeto que no sea nativo se considera anfitrión. Todos los objetos BOM y DOM se consideran objetos anfitrión y se analizarán en un apartado posterior.

1.1 Sumario

1.2 Objetos nativos en Javascript

ECMA-262 define los objetos nativos como "cualquier objeto proporcionado por una implementación de ECMAScript independiente del entorno anfitrión". Así, tenemos las siguientes clases: **Object**, **Function**, **Array**, **String**, **Boolean**, **Number**, **Date** y **RegExp**.

Algunos ya los vimos y, ahora, vamos a echar ahora una ojeada a algunos más de estos objetos nativos de JavaScript, ésto es, aquellos que JavaScript nos da, listos para su utilización en nuestra aplicación.

1.2.1 Clase Array

En ECMAScript, al contrario que en Java, existe una clase **Array**. Podemos crear un objeto **Array** del siguiente modo:

```
var aValores = new Array()
```

Otro modo de definirlo sería:

```
var aValores = [];
```

Sabiendo con antelación el número de elementos del array:

```
var aValores = new Array(10)
```

Para acceder a cada una de las posiciones del array, lo haremos utilizando corchetes, tanto para escribir:

```
var aColores = new Array();
var aColores[0] = "rojo";
var aColores[1] = "verde";
var aColores[2] = "azul";
```

como para leer el dato:

```
console.log(aColores[1]); //verde
```

Si queremos crear el array ya con una cantidad de valores conocidos:

```
var aColores = new Array("rojo", "verde", "azul");
```

Y, claro está, también:

```
var aColores = ["rojo", "verde", "azul"];
```

Para descubrir el tamaño total del array, lo haremos con su propiedad *length*:

```
console.log(aColores.length); //3
```

Los arrays pueden tener un total de 4.294.967.295 elementos (¿sufientes?).

Si queremos iterar en los elementos del array lo podemos hacer de varias formas:

- Utilizando la sentencia **for** con su propiedad **length**:

```
var aColores = ["rojo", "verde", "azul"];
for (let i=0; i<aColores.length; i++) {
    console.log(aColores[i]);
}
```

- Utilizando la sentencia **for...in**:

```
var aColores = ["rojo", "verde", "azul"];
for (let i in aColores) {
    console.log(aColores[i]);
}
```

Podemos crear una cadena a partir del array, utilizando para ello el método **join()** y, como parámetro, el separador que nos interese:

```
var aColores = ["rojo", "verde", "azul"];
sColores = aColores.join(","); //rojo, verde, azul
```

También el objeto **String** cuenta con el método **split()** para convertirse en array.

```
sColores = "rojo, verde, azul";
var aColores = sColores.split(",");
```

El objeto **Array** también tiene el método **concat()**, que nos permite añadir, al final, el contenido de un array en otro. Y el método **slice()**, para extraer varios elementos del array.

Curiosa es la utilidad de **slice()** para clonar un array:

```
let arrayOriginal = [1,2,3,4,5];
let arrayCopia = arrayOriginal.slice();
```

Para añadir y eliminar elementos del final del array, existen los métodos **push()** y **pop()**.

Para manipular el "primer elemento", en vez de el último, también posee la clase **Array** el método **shift()**, que lo elimina y, **unshift()** que añade un elemento en esa primera posición.

Para manipular el orden de los elementos, la clase **Array** tiene dos métodos. El método **reverse()**, que invierte la posición de todos los elementos. Y, el método **sort()** que los organiza en modo ascendente como si fuesen "cadenas de texto".

Un ejemplo interesante con el método **sort()** es el que nos permite ordenar un array de números enteros:

```
const numeros = [10, 30, 40, 33, 24, 67, 32, 12, 77, 54];
const numOrdenados = numeros.sort(function(a,b) {
    return a - b;
});
console.log(numOrdenados);
//[ 10, 12, 24, 30, 32, 33, 40, 54, 67, 77 ]
```

Sin duda, el método más complicado de la clase **Array** es **splice()**, que nos permite manipular elementos "en el medio" del array:

```
//Eliminar los dos primeros elementos
unArray.splice(0,2);
//Añadir esas cadenas en la posición 2
unArray.splice(2, 0, "rojo", "verde");
//Eliminar un elemento en la posición 2 y añadir dos cadenas ahí
unArray.splice(2, 1, "rojo", "verde");
```

1.2.1.1 Método *filter()*

El método **filter()** crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.

Veamos un ejemplo donde se filtran los elementos de un array de *strings* por el número de caracteres que tiene.

- Utilizando *callback* en modo ***Traditional function***:

```
var aColores = ["rojo", "verde", "azul", "violeta"];
var result = aColores.filter(function (color) {
    return color.length > 4;
});
//filter(callback(currentValue))
console.log(result); //["verde","violeta"]
```

- Utilizando *callback* en modo ***Arrow function***:

```
var aColores = ["rojo", "verde", "azul", "violeta"];
var result = aColores.filter(color => color.length > 4);
console.log(result); //["verde","violeta"]
```

No hay ningún beneficio por utilizar uno u otro modo de llamar a la función de *callback* pero, si no se está acostumbrado a emplear la notación de "funciones flecha", es mejor que sigas empleando la llamada tradicional.

1.2.1.2 Método *map()*

El método **map()** crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. Veamos un ejemplo donde se multiplica por 2 cada uno de los elementos de un array.

```
var numeros = [1, 5, 10, 15];
var doublesNumeros = numeros.map(function(num) {
      return num * 2;
 });
//map(callback(currentValue));
 console.log(numeros); //[1, 5, 10, 15]
 console.log(doublesNumeros); //[2, 10, 20, 30]
```

Si utilizamos el método **map()** sobre un objeto y estos objetos tienen múltiples propiedades con objetos en su interior tendremos que realizar la siguiente modificación:

```
const rockets = [
    { country:'Russia', launches:32 },
    { country:'US', launches:23 },
    { country:'China', launches:16 },
    { country:'Europe(ESA)', launches:7 },
    { country:'India', launches:4 },
    { country:'Japan', launches:3 }
];

const launchOptimistic = rockets.map(function(elem) {
  return {
    ...elem,
    launches: elem.launches+10,
  }
});

console.log(launchOptimistic);
```

1.2.1.3 Método *reduce()*

El método **reduce()** ejecuta una función reductora sobre cada elemento de un array, devolviendo como resultado un único valor. En el siguiente ejemplo se suman todos los elementos de un array devolviendo un único valor que es esa suma.

```
var numeros = [1, 5, 10, 15];
var total = numeros.reduce(function(a, b) {
    return a + b; }, 0);
//El último parámetro "0" será el valor inicial del acumulador
//reduce(callback(acumulador, valorActual), valorInicial);
console.log(total); // 31
```

1.2.2 Clase *String*

Una cadena **String** consta de uno o más caracteres de texto, rodeados de comillas simples o dobles; da igual cuales usemos ya que se considerará una cadena de todas formas, pero en algunos casos resulta más cómodo el uso de unas u otras.

Por ejemplo, si queremos meter el siguiente texto dentro de una cadena de JavaScript:

```
<input  type="checkbox"  name="coche"  />Audi
```

podremos emplear las comillas dobles o simples:

```
//Un modo:
var cadena = '<input  type="checkbox"  name="coche"  />Audi';
//Otro modo:
var cadena = "<input  type='checkbox'  name='coche'  />Audi";
```

Si queremos emplear comillas dobles al principio y fin de la cadena, y que en el contenido aparezcan también comillas dobles, tendríamos que escaparlas con **\"** , por ejemplo:

```
var cadena = "<input  type=\"checkbox\"  name=\"coche\"  />Audi";
```

Cuando estamos hablando de cadenas muy largas, podríamos concatenarlas con **+=**, por ejemplo:

```
var  nuevoDocumento  =  "";
nuevoDocumento  +=  "<!DOCTYPE  html>";
nuevoDocumento  +=  "<html>"  ;
nuevoDocumento  +=  "<head>";
nuevoDocumento  +=  '<meta  http-equiv="content-type"';
nuevoDocumento  +=  'content="text/html;charset=utf-8">';
?
```

Si queremos concatenar el contenido de una variable dentro de una cadena de texto emplearemos el símbolo **+**.

```
var nombreEquipo  =  prompt("Introduce el nombre de tu equipo:","");
var  mensaje= "El  "  +  nombreEquipo +  "  ha sido el campeón de la Copa del Rey!";
alert(mensaje);
```

O utilizaríamos las comillas inclinadas.

```
var nombreEquipo  =  prompt("Introduce el nombre de tu equipo:","");
var mensaje= `El ${nombreEquipo} ha sido el campeón de la Copa del Rey!`;
alert(mensaje);
```

- **Caracteres especiales o caracteres de escape**

  La forma en la que se crean las cadenas en JavaScript, hace que cuando tengamos que emplear ciertos caracteres especiales en una cadena de texto, tengamos que escaparlos, empleando el símbolo \ seguido del carácter.

  Vemos aquí un listado de los caracteres especiales, o de escape:

Símbolos	Explicación
\"	Comillas dobles
\?	Comilla simple
\\	Barra inclinada
\b	Retroceso
\t	Tabulador
\n	Nueva Línea
\r	Salto de Línea
\f	Avance de página

- **Propiedades y métodos del objeto String.**

  Para crear un objeto String lo podremos hacer de la siguiente forma:

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

//O también se podría hacer:
var miCadena = "texto de la cadena";

//Es decir, cada vez que tengamos una cadena de texto,
//en realidad es un objeto String que tiene propiedades y métodos:
//cadena.propiedad;
//cadena.metodo( [parámetros] );
```

Propiedades del objeto String

Propiedad	Descripción
length	Contiene la longitud de una cadena.

Métodos del objeto String

Método	Descripción
charAt()	Devuelve el carácter especificado por la posición que se indica entre paréntesis.
charCodeAt()	Devuelve el Unicode del carácter especificado por la posición que se indica entre paréntesis.
concat()	Une una o más cadenas y devuelve el resultado de esa unión.
fromCharCode()	Convierte valores Unicode a caracteres.
indexOf()	Devuelve la posición de la primera ocurrencia del carácter buscado en la cadena.
lastIndexOf()	Devuelve la posición de la última ocurrencia del carácter buscado en la cadena.
match()	Busca una coincidencia entre una expresión regular y una cadena y devuelve las coincidencias o null si no ha encontrado nada.
normalize()	Convertir a Unicode.

Método	Descripción
replace()	Busca una subcadena en la cadena y la reemplaza por la nueva cadena especificada.
search()	Busca una subcadena en la cadena y devuelve la posición dónde se encontró.
slice()	Extrae una parte de la cadena y devuelve una nueva cadena.
split()	Divide una cadena en un array de subcadenas.
substr()	Extrae los caracteres de una cadena, comenzando en una determinada posición y con el número de caracteres indicado.
substring()	Extrae los caracteres de una cadena entre dos índices especificados.
toLowerCase()	Convierte una cadena en minúsculas.
toUpperCase()	Convierte una cadena en mayúsculas.

Ejemplos de uso:

```
var cadena="El parapente es un deporte de riesgo medio";

document.write("Longitud de la cadena: "+ cadena.length + "<br/>");
document.write(cadena.toLowerCase()+ "<br/>");
document.write(cadena.charAt(3)+ "<br/>");
document.write(cadena.indexOf('pente')+ "<br/>");
document.write(cadena.substring(3,16)+ "<br/>");
```

Ejemplo de eliminación de acentos (método normalize()):

```
let texto = "camión lápiz pingüino";
const patronTildes = /[\u0300-\u036f]/g;
//normalize("NFD") convierte, por ejemplo, 'ó' = 'o' + '´'
//replace(patronTildes,'') elimina todo tipo de tildes
let textoSinTildes = texto.normalize("NFD").replace(patronTildes,"");
console.log(texto);
console.log(textoSinTildes);
```

1.2.3 Clase Boolean

El objeto **Boolean** se utiliza para convertir un valor no Booleano, a un valor Booleano (*true* o *false*).

Propiedades del objeto Boolean:

Propiedad	Descripción
Constructor	Devuelve la función que creó el objeto Boolean.
Prototype	Te permitirá añadir propiedades y métodos a un objeto.

Métodos del objeto Boolean:

Método	Descripción
toString()	Convierte un valor Boolean a una cadena y devuelve el resultado.
valueOf()	Devuelve el valor primitivo de un objeto Boolean.

Algunos ejemplos de uso:

```
let bool = new Boolean(1);
document.write(bool.toString());
document.write(bool.valueOf());
```

1.2.4 Clase Number

El objeto **Number** se usa muy raramente, ya que para la mayor parte de los casos, JavaScript satisface las necesidades del día a día con los valores numéricos que almacenamos en variables. Pero el objeto *Number* contiene alguna información y capacidades muy interesantes para programadores más serios.

Lo primero, es que el objeto *Number* contiene propiedades que nos indican el rango de números soportados en el lenguaje. El número más alto es $1.79E^{+308}$; el número más bajo es $2.22E^{-308}$. Cualquier número mayor que el número más alto, será considerado como infinito positivo, y si es más pequeño que el número más bajo, será considerado infinito negativo.

Los números y sus valores están definidos internamente en JavaScript, como valores de doble precisión y de 64 bits.

El objeto *Number*, es un objeto envoltorio para valores numéricos primitivos.

Los objetos *Number* son creados con **new Number()**.

Propiedades del objeto Number:

Propiedad	Descripción
constructor	Devuelve la función que creó el objeto Number.
MAX_VALUE	Devuelve el número más alto disponible en JavaScript.
MIN_VALUE	Devuelve el número más pequeño disponible en JavaScript.
NEGATIVE_INFINITY	Representa a infinito negativo (se devuelve en caso de overflow).
POSITIVE_INFINITY	Representa a infinito positivo (se devuelve en caso de overflow).
prototype	Permite añadir nuestras propias propiedades y métodos a un objeto.

Métodos del objeto Number:

Método	Descripción
toExponential(x)	Convierte un número a su notación exponencial.
toFixed(x)	Formatea un número con x digitos decimales después del punto decimal.
toPrecision(x)	Formatea un número a la longitud x.
toString()	Convierte un objeto Number en una cadena. ? Si se pone 2 como parámetro se mostrará el número en binario. ? Si se pone 8 como parámetro se mostrará el número en octal. ? Si se pone 16 como parámetro se mostrará el número en hexadecimal.
valueOf()	Devuelve el valor primitivo de un objeto Number.

Algunos ejemplos de uso:

```
let num = new Number(13.3714);
document.write(num.toPrecision(3)+"<br />");
document.write(num.toFixed(1)+"<br />");
document.write(num.toString(2)+"<br />");
document.write(num.toString(8)+"<br />");
document.write(num.toString(16)+"<br />");
document.write(Number.MIN_VALUE);
document.write(Number.MAX_VALUE);
```

1.2.5 Clase Date

El objeto **Date** se utiliza para trabajar con fechas y horas. Hay 4 formas de instanciar (crear un objeto de tipo *Date*):

```
let d = new Date();
let d = new Date(milisegundos);
let d = new Date(cadena de Fecha);
let d = new Date(año, mes, [día, [horas, [minutos, [segundos, [milisegundos]]]]]);
// (el mes comienza en 0, Enero sería 0, Febrero 1, etc.)
```

Propiedades del objeto Date:

Propiedad	Descripción
constructor	Devuelve la función que creó el objeto Date.
prototype	Te permitirá añadir propiedades y métodos a un objeto.

Algunos métodos del objeto Date:

Método	Descripción

Método	Descripción
getDate()	Devuelve el día del mes (de 1-31).
getDay()	Devuelve el día de la semana (de 0-6).
getFullYear()	Devuelve el año (4 dígitos).
getHours()	Devuelve la hora (de 0-23).
getMilliseconds()	Devuelve los milisegundos (de 0-999).
getMinutes()	Devuelve los minutos (de 0-59).
getMonth()	Devuelve el mes (de 0-11).
getSeconds()	Devuelve los segundos (de 0-59).
getTime()	Devuelve los milisegundos desde media noche del 1 de Enero de 1970.
getTimezoneOffset()	Devuelve la diferencia de tiempo entre GMT y la hora local, en minutos.
getUTCDate()	Devuelve el día del mes en base a la hora UTC (de 1-31).
getUTCDay()	Devuelve el día de la semana en base a la hora UTC (de 0-6).
getUTCFullYear()	Devuelve el año en base a la hora UTC (4 dígitos).
setDate()	Ajusta el día del mes del objeto (de 1-31).
setFullYear()	Ajusta el año del objeto (4 dígitos).
setHours(2)	Ajusta la hora del objeto (de 0-23).
UTC()	Devuelve el número de milisegundos desde el 1 de enero de 1970 hasta la fecha pasada como parámetro pasado en formato largo.

Algunos ejemplos de uso:

```
let d = new Date();
document.write(d.getFullYear());
document.write(d.getMonth());
document.write(d.getUTCDay());
//Cálculo de diferencia de fechas
let d2 = new Date(2011,5,28,22,58,00);
d2.setMonth(0);
d.setFullYear(2020);
//Las diferencias entre fechas es en milisegundos
```

1.3 Objetos incorporados en JavaScript

ECMA-262 define los objetos incorporado como "cualquier objeto proporcionado por una implementación de ECMAScript, independientemente del entorno anfitrión, que está presente al iniciarse la ejecución de un programa de ECMAScript". Esto significa que no es necesario que el programador cree una instancia explícita del objeto incorporado, por que dicha instancia ya existe. Sólo existen dos objetos incorporados: **Global** y **Math** que, por definición, ya son objetos nativos.

Con el objeto **Global** no nos vamos a parar pues, básicamente, sirve para que, funciones como **isNaN()**, **parseInt()**,... que parecen funciones independientes, son en realidad métodos del objeto **Global**.

1.3.1 Clase Math

Ya vimos anteriormente algunas funciones que nos permitían convertir cadenas a diferentes formatos numéricos (**parseInt()**, **parseFloat()**). A parte de esas funciones, disponemos de un objeto **Math** en JavaScript, que nos permite realizar operaciones matemáticas. El objeto **Math** no es un constructor (no nos permitirá por lo tanto crear o instanciar nuevos objetos que sean de tipo Math), por lo que, para llamar a sus propiedades y métodos, lo haremos anteponiendo **Math** a la propiedad o el método. Por ejemplo:

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

Propiedades del objeto Math:

Propiedad	Descripción

Propiedad	Descripción
E	Devuelve el número Euler (aprox. 2.718).
LN2	Devuelve el logaritmo neperiano de 2 (aprox. 0.693).
LN10	Devuelve el logaritmo neperiano de 10 (aprox. 2.302).
LOG2E	Devuelve el logaritmo base 2 de E (aprox. 1.442).
LOG10E	Devuelve el logaritmo base 10 de E (aprox. 0.434).
PI	Devuelve el número PI (aprox. 3.14159).
SQRT2	Devuelve la raíz cuadrada de 2 (aprox. 1.414).

Métodos del objeto Math:

Método	Descripción
abs(x)	Devuelve el valor absoluto de x.
acos(x)	Devuelve el arcocoseno de x, en radianes.
asin(x)	Devuelve el arcoseno de x, en radianes.
atan(x)	Devuelve el arcotangente de x, en radianes con un valor entre -PI/2 y PI/2.
atan2(y,x)	Devuelve el arcotangente del cociente de sus argumentos.
ceil(x)	Devuelve el número x redondeado al alta hacia el siguiente entero.
cos(x)	Devuelve el coseno de x (x está en radianes).
floor(x)	Devuelve el número x redondeado a la baja hacia el anterior entero.
log(x)	Devuelve el logaritmo neperiando (base E) de x.
max(x,y,z,...,n)	Devuelve el número más alto de los que se pasan como parámetros.
min(x,y,z,...,n)	Devuelve el número más bajo de los que se pasan como parámetros.
pow(x,y)	Devuelve el resultado de x elevado a y.
random()	Devuelve un número al azar entre 0 y 1.
round(x)	Redondea x al entero más próximo.
sin(x)	Devuelve el seno de x (x está en radianes).
sqrt(x)	Devuelve la raíz cuadrada de x.
tan(x)	Devuelve la tangente de un ángulo.

Ejemplos de uso:

```
document.write(Math.cos(3) + "<br />");
document.write(Math.asin(0) + "<br />");
document.write(Math.max(0,150,30,20,38) + "<br />");
document.write(Math.pow(7,2) + "<br />");
document.write(Math.round(0.49) + "<br />");
```

Podemos ver un ejemplo donde se utilizan métodos del objeto Math en el que se generan colores de modo aleatorio. Cada vez que se hace *click* en el elemento **h1** de la página, cambia de color.

- Código HTML

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8" />
<title>JavaScript</title>
</head>
<body>
<h1 id="miH1">Este título cambia de color</h1>

<script src="prueba1.js"></script>
</body>
</html>
```

- Código JavaScript

```
//Generar un número aleatorio de 0 a val
function randomVal(val) {
return Math.floor(Math.random() * val);
}

//Generar un color rgb aleatorio
function randomColor() {
return (
"rgb(" +
randomVal(255) +
"," +
randomVal(255) +
"," +
randomVal(255) +
")"
);
}

//Función que cambia de color el elemento html que se pasa por parámetro
function cambiaColor(id) {
let nColor = randomColor();
console.log(nColor);
document.getElementById(id).style.color = nColor;
}

//Función que manda cambiar de color
//Como cambiaColor(id) se configura como escuchador
//no se le puede enviar parámetro directamente
function mandaCambiarColor() {
    cambiaColor("miH1");
}

//Cuando ya se encuentre toda la página cargada
window.addEventListener("load", function (event) {
let textoH1 = document.getElementById("miH1");
//Al hacer click en el título
textoH1.addEventListener("click", mandaCambiarColor);
});
```

1.4 Definir clases y objetos

La posibilidad de utilizar objetos predefinidos es sólo una parte de cualquier lenguaje orientado a objetos. Las verdaderas prestaciones se obtienen al crear clases y objetos propios para usos específicos.

- **La clase**

  En JavaScript las clases son funciones, para comprobarlo veremos luego un ejemplo donde definimos una clase "**Coche**" utilizando la expresión **function** y la expresión **class**.

- **El objeto**

  Para crear un nuevo objeto se utiliza la declaración **new**, asignando el resultado (que es de tipo **obj**) a una variable para tener acceso más tarde. En el siguiente ejemplo se define, como ya se dijo, una clase llamada "**Coche**" y una instancia llamada "**coche1**".

- **El constructor**

  El constructor es llamado en el momento de la creación de la instancia (el momento en que se crea la instancia del objeto). *El constructor es un método de la clase*. En JavaScript no hay necesidad de definir explícitamente un método constructor. Cada acción declarada en la clase es ejecutada en el momento de la creación de la instancia.
  El constructor se usa para establecer las propiedades del objeto o para llamar a los métodos para preparar el objeto para su uso.

- **Las propiedades**

  Las propiedades son variables contenidas en la clase, cada instancia del objeto tiene dichas propiedades. Las propiedades deben establecerse a la propiedad prototipo de la clase, para que la herencia funcione correctamente.

Para trabajar con propiedades dentro de la clase se utiliza la palabra reservada **this**, que se refiere al objeto actual. El acceso (lectura o escritura) a una propiedad desde fuera de la clase se hace con la sintaxis: **NombreDeLaInstancia.Propiedad** (desde dentro de la clase la sintaxis es **this.Propiedad**).
En el siguiente ejemplo definimos varias propiedades de la clase **Coche**.

- **Los métodos**

Los métodos siguen la misma lógica que las propiedades, la diferencia es que son funciones y se definen como funciones. Llamar a un método es similar a acceder a una propiedad, pero se agrega **()** al final del nombre del método, pero, posiblemente vaya acompañado de argumentos.
En el siguiente ejemplo se define y utiliza el método **mostrarColor()** para la clase **Coche**, en él se define una clase **Coche** con varias propiedades y un método. Luego se crea un objeto coche1 a partir de una instancia de ella.

Hasta ES5 el modo de definir objetos es utilizando la expresión **function**, luego, con la llegada de ES6 en JavaScript se introduce **class** que permite tener una expresión específica para la definición de objetos y un modo mucho más sencillo y dinámico de implementar conceptos como la Herencia, la Abstracción y el Polimorfismo. Así, aunque en el siguiente punto se hace una pequeña referencia a la definición de objetos con **function** se recomienda emplear **class** y nos pararemos más en este modo de crear Clases de Objetos y sus posteriores instancias.

1.4.1 Definimos la clase "Coche" con la expresión *function*

Veamos como se puede definir una sencilla clase de objetos con **function** y el posterior instanciado de un objeto en concreto. Se trata de un ejemplo muy básico, en donde con cuatro propiedades y un método, sin pararnos ya a ver herencia, *getters* ni *setters*, esas características veremos como se implementan con la expresión **class**. Hasta la llegada de ES6 JavaScript era un lenguaje basado en prototipos que no contiene ninguna declaración de clase, como se encuentra, por ejemplo, en C++ o Java. Esto es a veces confuso para los programadores acostumbrados a los lenguajes con una declaración de clase. En su lugar, JavaScript utiliza funciones como clases.

```
function Coche(sColor, iPuertas, iKmpl) {
this.color = sColor;        //Color coche
this.puertas = iPuertas;        //Número de puertas coche
this.kmpl = iKmpl;        //Kms recorridos por litro
this.conductores = new Array(); //Lista conductores coche
}

Coche.prototype.mostrarColor = function() {
alert(this.color);
}

//Inicializar un coche de prueba
let coche1 = new Coche("rojo", 4, 20);
coche1.conductores = ["Andrea", "Ana", "Miguel"];
//Leemos una propiedad
        console.log(`El coche es de color ${coche1.color}`);
        console.log(`Lo conducen ${coche1.conductores.join(", ")}`);
//Llamamos al método
coche1.mostrarColor();
```

1.4.2 Definimos la clase ?Coche? con la expresión *class*

Las clases de javascript, introducidas en ECMAScript 2015 (ES6), son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript. La sintaxis de las clases no introduce un nuevo modelo de herencia orientada a objetos en JavaScript. Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y definir la herencia.

Las clases son "funciones especiales", teniendo la sintaxis de una clase dos componentes: expresiones de clases y declaraciones de clases.

Para declarar una clase, se utiliza la palabra reservada **class** y un nombre para la clase, que en el siguiente ejemplo será "Coche".

Una importante diferencia entre las declaraciones de funciones y las declaraciones de clases es que las declaraciones de funciones son alojadas (*hoisted*) y las declaraciones de clases no lo son. Es decir, en primer lugar se necesita declarar la clase y luego acceder a ella, de otro modo el ejemplo de código siguiente arrojará un **ReferenceError**:

```
class Coche {
constructor (sColor, iPuertas, iKmpl) {
  this.color = sColor; //Color coche
  this.puertas = iPuertas; //Número de puertas coche
  this.kmpl = iKmpl;        //Kms recorridos por litro consumido
```

```
  this.conductores = new Array(); //Lista conductores coche
}
//Método
mostrarColor() {
  alert(this.color);
}
}

//Inicializar un coche
let coche1 = new Coche("rojo", 4, 20);
coche1.conductores = ["Andrea", "Ana", "Miguel"];
//Leemos una propiedad
console.log(`El coche es de color ${coche1.color}`);
console.log(`Lo conducen : ${coche1.conductores.join(", ")}`);
//Llamamos al método
coche1.mostrarColor();
```

1.4.3 Anular referencias a objetos

En JavaScript no se puede acceder a la representación física del objeto, sólo a **referencias** del mismo. Saber también que JavaScript cuenta con una rutina de recolección de elementos sin utilizar, por lo que no es necesario destruir específicamente los objetos para liberar memoria. Cuando no quedan referencias a un objeto, se dice que se han anulado las referencias al mismo. Luego, al ejecutar el recolector de elementos sin utilizar, todos los objetos con referencias anuladas se destruyen.

Pueden anularse las referencias a objetos de forma manual, si se establecen todas en **null**. Es interesante realizar esta acción con objetos sin utilizar para así liberar memoria.

```
let miObjeto = new Object;
//Al terminar de trabajar con el objeto
miObjeto = null;
```

1.4.4 Los *getters* y los *setters*

Desde ES2015, tenemos la posibilidad de usar getters y setters para definir propiedades en nuestros objetos.

Una función que obtiene un valor de una propiedad se llama **getter** y una que establece el valor de una propiedad se llama **setter**.

Veamos un ejemplo muy sencillo de la utilidad de estas funciones (fuente):

```
//Creamos una clase Persona
class Persona {
constructor (sNombre, sApellido) {
this.nombre = sNombre;
this.apellido = sApellido;
}

// Con get creamos propiedad "nombreCompleto"
//con propiedades del propio objeto
get nombreCompleto() {
return `${this.nombre} ${this.apellido}`;
}

// Con set modificamos alguna propiedad del objeto
//utilizando propiedades ya existentes
set nombreYapellido(nom) {
let palabras = nom.split(/\s/);
this.nombre = palabras[0] || '';
this.apellido = palabras[1] || '';
}
}

//Probamos el funcionamiento
let persona1 = new Persona;
persona1.nombreYapellido = 'Camilo Sánchez';
console.log(persona1.nombre); //Camilo
console.log(persona1.apellido); //Sánchez

console.log('------------------');
```

```
let persona2 = new Persona('Andrea', 'Torres');
console.log(persona2.nombreCompleto); //Andrea Torres
```

1.4.5 De/a objeto a/de *string* JSON

Para pasar un objeto a formato texto lo haremos empleando los objetos **JSON**. El modo de hacerlo será utilizando el método **JSON.stringify()**, y que añadiremos al código anterior del siguiente modo:

```
//Pasar un objeto a 'Formato texto (JSON)'
console.log(JSON.stringify(persona1));
//{"nombre":"Camilo","apellido":"Sanchez"}
```

También podemos aprovechar el método **JSON.parse()** de los objetos JSON para importar un string JSON y pasar su contenido a un objeto determinado. Podemos ver un ejemplo:

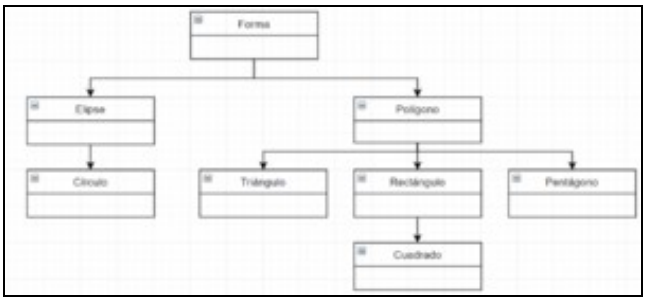
```
//String con formato JSON con los datos a importar
let jsonAna = '{"nombre":"Ana","apellido":"Torres"}';
//Lo pasamos a Objeto con el método parse()
let temporal = JSON.parse(jsonAna);
//Escribimos esos datos importados en una instancia del objeto Persona
let personaAna = new Persona(temporal.nombre, temporal.apellido);

//Comprobamos lo importado
console.log(personaAna);
console.log(personaAna.nombreCompleto);
```

1.4.6 Herencia

La herencia es una manera de crear una clase como una versión especializada de una o más clases (JavaScript sólo permite herencia simple). La clase "especializada" comúnmente se llama hija o secundaria, y la otra clase se le llama padre o primaria. En JavaScript la herencia se logra mediante la asignación de una instancia de la clase primaria a la clase secundaria, y luego se hace la especialización.

Veamos un ejemplo donde se definen formas geométricas.

> Existen dos tipos de formas: las elipses (con forma redonda) y los polígonos (con un número concreto de lados). Los círculos son un tipo de elipse con un centro; los triángulos, rectángulos y pentágonos son tipos de polígonos con distintos lados. Un cuadrado es un rectángulo con sus cuatro lados iguales. De este modo podemos definir una relación de herencia perfecta.
> El diagrama UML podría ser el siguiente:



UML Polígonos

> En este ejemplo, sólo nos centraremos en la creación de la clase **Poligono** y de las subclases **Triangulo** y **Rectangulo**.
> La clase **Poligono** tendrá la propiedad lados y el método **getArea()**.

```
//Definición de la clase Poligono
class Poligono {
constructor(iLados) {
this.lados = iLados; }
//Métodos
getArea() { return 0; }
}
```

> Vemos que el método **getArea()** devuelve **0** porque es simplemente un marcador de posición de los métodos de las posteriores subclases que se vayan a crear.

Ahora creamos la subclase **Triangulo**. Esta clase tiene tres lados, de forma que esta clase tiene que reemplazar la propiedad **lados** de la clase **Poligono** y establecerla en **3**. También es necesario reemplazar **getArea()** para que utilice la fórmula del área del triángulo, que es "1/2 x base x altura".
El método obtendrá esos dos datos de las dos propiedades **base** y **altura**.
Así, el código de la clase Triangulo será el siguiente:

```
//Definición de la clase Triangulo subclase de Poligono
class Triangulo extends Poligono {
constructor(iBase, iAltura, iLados) {
super(iLados);
this.lados = 3;
this.base = iBase;
this.altura = iAltura; }
getArea() { return 0.5 * this.base * this.altura; }
}
```

De igual modo, definimos el código de la clase Rectangulo:

```
//Definición de la clase Rectangulo
class Rectangulo extends Poligono {
    constructor (iLargo, iAncho, iLados) {
        super (iLados);
        this.lados = 4;
        this.largo = iLargo;
        this.ancho = iAncho; }
    getArea() { return this.largo * this.ancho; }
}
```

Por fin, comprobamos que el código está bien escrito:

```
var triangulo1 = new Triangulo(12, 4);
var rectangulo1 = new Rectangulo(22, 10);

console.log(triangulo1.getArea());
console.log(triangulo1.lados);
console.log(rectangulo1.getArea());
console.log(rectangulo1.lados);
```

1.4.7 Ejemplo bolas en movimiento

Código

Volver