Introducción a JS

Clases Core

Módulo: **Desarrollo Web en entorno cliente**

CFGS Desarrollo de Aplicaciones Web

Además de los tipos primitivos que ya hemos visto, JS tiene unas clases, llamadas Core. Las más usadas y que ya hemos visto por encima son: Object , Number , Array y String. Todas heredan de Object.

A continuación se darán algunos tips que se consideran importantes como introducción de las clases Oject, Number y Arrary. El apartado de String será tratado en profundidad en la siguiente unidad.

Object

Un objeto es una colección de variables y funciones agrupadas de manera estructural. A las variables definidas dentro de un objeto se las denomina **propiedades** (pares clave-valor), y las funciones, **métodos**.

```
var album = {
    title: 'Brave enough',
    singer: 'Lindsey Stirling',
    year: 2016,
    formats: ["mp3", "CD audio", "Disco de vinilo"],
    price: 15,
    currency: 'Euros',
    published: true
};
```

Se accede a las propiedades a través de la notación punto o con la notación array:

```
    album.titulo; // Brave enough
    album['year']; // 2016
```

También se pueden modificar:

```
album.price = 16;
album['publicado'] = false;
```

¿Se pueden usar variables para acceder a las propiedades de un objeto? Solo en el caso de usar la notación array, pero no con la notación punto.

```
var property =  "singer";
album[property]; // "LindseyStirling"
album.property ; // undefined
```

El objeto contiene funciones que llamaremos métodos:

```
var album = {
    price: 15,
    listenDisk: function () {
        console.log("He oído el disco de Lindsey Stirling");
    }
};

album.leer(); // Devuelve: "He oído el disco de Lindsey Stirling"
```

Para crear un objeto podemos hacerlo con la notación de llaves {...} o creando una nueva instancia de clase:

```
var miObjeto = { propiedad: "valor" };
var miObjeto = new Object({ propiedad: "valor" });
```

Anidación

Un objeto puede tener propiedades y estas propiedades tener en su interior más propiedades. Sería una representación en forma de árbol:

```
var album = {
    title: "Brave enough",
    singer: {
        name: "Lindsey Stirling",
        age: 29,
        contact: {
            facebook: "https://www.facebook.com/lindseystirlingmusic",
            twitter: "@LindseyStirling"
        }
    },
    media: {
        youtube: "https://www.youtube.com/user/lindseystomp",
        spotify: "https://open.spotify.com/artist/378dH6EszOLFShpRzAQkVM"
    }
};

// Podemos acceder con notación punto, array, o mixto.
album.singer.name; // "Lindsey Stirling"
album['singer']['age']; // 29
album['media'].spotify; // "https://open.spotify.com/artist/378dH6EszOLFShpRzAQkVM"
album.singer['contact'].twitter; // "@LindseyStirling
```

Igualdad entre objetos

Para que dos objetos sean iguales al compararlos, deben tener la **misma referencia**:

- hemos de usar el operador de identidad estricta ===
- dos objetos con el mismo contenido no serán iguales a menos que compartan la referencia

```
var coche1 = { marca: "Ford", modelo: "Focus" };
var coche2 = { marca: "Ford", modelo: "Focus"};

coche1 === coche2; // Devuelve false, no comparten referencia
coche1.modelo === coche2.modelo; // Devuelve true porque el valor es el mismo.

var coche3 = coche1;
coche1 === coche3; // Devuelve true, comparten referencia
```

Number

Es la clase del tipo primitivo number . Se codifican en formato de coma flotante con doble precisión (8 bytes = 64 bits) y podemos representar números enteros, decimales, hexadecimales, y en coma flotante.

La clase Number incluye los números **Infinity** y -Infinity para representar números muy grandes:

```
1/0 = Infinity
-1/0 = -Infinity
1e1000 = Infinity
-1e1000 = -Infinity
```

Cuando un número supera el límite de los números en punto flotante (1.797693134862315E+308) se devuelve Infinity .

También disponemos del valor **NaN** (*Not A Number*) para indicar que un determinado valor no representa un número:

```
"a"/15 = NaN
```

Para crear un número podemos hacerlo con la forma primitiva o con la clase Number. Por comodidad se usa la primitiva:

```
var myNumber = 6;
var myNumber = new Number(6);
```

Array

Es una colección de datos que pueden ser números, strings, objetos, otros arrays, etc. Se puede crear de dos formas con el literal [...] o creando una nueva instancia de la clase Array.

```
var myArray = [];
var myArray = new Array();
var myArray = [1, 2, 3, 4]; // Array de números
var myArray = ["Hola", "que", "tal"]; // Array de Strings
var myArray = [ {property: "valor1" }, { property: "valor2" }]; // Array de objetos
var myArray = [[2, 4], [3, 6]]; // Matriz
var myArray = [1, true, [3,2], "Hola", {key: "valor"}]; // Array mixto
```

Se puede acceder a los elementos del array a través de su índice y la propiedad lenght nos da su longitud.

```
var myArray = ["uno", "dos", "tres"];
myArray[1]; // Devuelve: "dos"
myArray.length; // Devuelve 3
```

Si accedemos a una posición que no existe en el array, nos devuelve undefined.

```
myArray[8]; // undefined
```

Métodos

Array es una clase de JS, por tanto los objetos creados a partir de esta clase heredan todos los métodos de la clase padre. Los más utilizados son:

```
var myArray = [3, 6, 1, 4];
myArray.sort(); // Devuelve un nuevo array con los valores ordenados: [1, 3, 4, 6]
myArray.pop(); // Devuelve el último elemento del array y lo saca. Devuelve 6 y miArray queda [1,
myArray.push(2); // Inserta un nuevo elemento en el array, devuelve la nueva longitud del array y
myArray.reverse(); // Invierte el array, [2,4,3,1]
```

Otro método muy útil es **join**() sirve para crear un string con los elementos del array uniéndolos con el "separador" que le pasemos como parámetro a la función. Es muy usado para imprimir strings, sobre todo a la hora de crear templates. Ejemplo:

```
var value = 3;
var template = [
    "<li>",
    valor
    "</li>"
].join("");
console.log(template); // Devuelve: "<li>3</li>"
```

Lo cual es mucho más eficiente en términos de procesamiento, que realizar lo siguiente, sobre todo si estas uniones se realizan dentro de bucles.

```
var value = 3;
var template = "<li>" + valor + "</li>";
```

Si queremos aplicar una misma función a todos los elementos de un array podemos utilizar el método **map**. Imaginemos el siguiente array de números [2, 4, 6, 8] y queremos conocer la raíz cuadrada de cada uno de los elementos podríamos hacerlo así:

```
var myArray = [2, 4, 6 ,8];
var raices = myArray.map(Math.sqrt);  // raices: [ 1.4142135623730951, 2, 2.449489742783178, 2.8284271247461903 ]
```

O algo más específico:

```
var myArray = [2, 4, 6, 8];
var results = myArray.map(function(element) {
    return element *= 2;
});  // resultados: [ 4, 8, 12, 16 ]
```

Otra función interesante de los arrays es la función **filter** . Nos permite "filtrar" los elementos de un array dada una condición sin necesidad de crear un bucle para iterarlo. Por ejemplo, dado un array con los números del 1 al 15, obtener un array con los números que son divisibles por 3:

```
var myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
var result = myArray.filter(function(element) {
    return element % 3 === 0;
});  // resultados: [ 3, 6, 9, 12, 15 ]
```

Si queremos obtener una parte del array, podemos emplear la función **slice** pasándole por parámetro el índice a partir del que queremos cortar y el final. Si no se indica el parámetro de fin, se hará el "corte" hasta el final del array, si no, se hará hasta la posición indicada y si se pasa un número negativo, contará desde el final del array hacia atrás.

El método devuelve un nuevo array sin transformar sobre el que se está invocando la función.

```
var myArray = [4, 8, 15, 16, 23, 42];
myArray.slice(2); // [15, 16, 23, 42]
myArray.slice(2, 4); // [15, 16] (la posición de fin no es inclusiva)
myArray.slice(2, -1); // [15, 16, 23]
```