{0}

Autores: Algunos

```
console.log("Hello World!!");
```

JavaScript (JS) es el lenguaje de programación desarrollado por Netscape para la web[1]. Se trata de un lenguaje de script, generalmente interpretado por el navegador del usuario, esto es, se ejecuta en el *cliente*[2]. Hablamos del cliente refiriéndonos a la máquina del usuario final que visitará el sitio web, y del servidor para referirnos a la máquina o el conjunto de ellas que lo almacenan y proporcionan. Sin embargo, también se puede utilizar JavaScript como lenguaje de servidor gracias a Node.js.

JavaScript tiene una sintaxis similar a la de C++ o Java, y puede funcionar como un lenguaje procedimental o como uno orientado a objetos. Además, es capaz de responder a eventos generados mediante la interacción del usuario, lo que es generalmente una de sus principales funciones.

Nota : Pese al nombre, JavaScript no deriva de Java, los dos lenguajes apenas se parecen, exceptuando el aspecto de la sintaxis. De hecho, se llamó *Mocha* y *LiveScript* antes de obtener su nombre definitivo.

Para ejecutar tu propio código JavaScript, pulsa F12 o Ctrl-Mayus-C en tu navegador. En las herramientas de desarrollador tendrás una consola JS interactiva disponible para experimentar. Es importante notar que cada navegador implementa JavaScript de una manera ligeramente distinta, luego puede haber peculiaridades en la forma de aprovechar algunos recursos del navegador. El estándar del lenguaje se denomina ECMAScript.

Sintaxis y tipos

Declaraciones, variables y funciones

JavaScript no es fuertemente tipado, por lo que declaramos las variables simplemente con la palabra reservada `var`. Declaramos una constante con `const` y una función con `var ... = function()` o `function ...()`. También podemos declarar variables simplemente con una asignación, pero entonces serán globales, lo que es equivalente a declararlas fuera de cualquier función (las variables y funciones globales se convierten en atributos del objeto `window`).

```
var una = 1;
const PI = 3.141592;
```

[1] Netscape and Sun announce JavaScript. Press Release
[2] About JavaScript - Mozilla Developer Network

```
var perimetro_circulo = function(radio) {
    // Esta variable no estaba declarada antes,
    // así que es global
    tau = 2 * PI;

    return tau * radio;
};

// Otra forma de declarar funciones
function area_circulo(radio) {
    return PI * radio * radio;
}
```

Nota : JavaScript acepta ciertos símbolos a la hora de nombrar variables. Un nombre de variable no puede empezar por un número, pero sí que puede comenzar y contener símbolos como $, _ o incluso

$$\pi$$

[3].

Colecciones y objetos

En JS utilizamos los corchetes para representar arrays. Estos pueden contener objetos de distinta naturaleza, por ejemplo, el siguiente array es completamente válido:

```
var válido = [20, "Hola", /^[a-z]/];
```

Los objetos son uno de los pilares de JS. En este lenguaje, los objetos existen, no se crean como instancias de una clase, por lo que podemos declarar nuevos con la siguiente sintaxis:

```
var un_objeto = {
    dato: "valor",
    otro: 3,
    una_funcion: function() {
        return Math.random();
    }
};

// Acceso a datos:
```

[3] Valid JavaScript variable names - Mathias Bynens

```
un_objeto.otro;      // --> 3
un_objeto["dato"];  // --> "valor"
un_objeto.una_funcion();
```

La forma en que se define el objeto es conocida como *JavaScript Object Notation* (JSON). Un objeto puede funcionar como un diccionario, de la forma `{ clave: "valor", clave2: "otro" }`. Solemos llamar *propiedad* a cada una de las parejas *clave-valor* de un objeto.

Tipos

JavaScript tiene algunos tipos básicos: `Number`, `String`, `Boolean`. El resto de objetos (los de `Date`, `RegExp`...) serán de tipo `Object`, y las funciones y constructores son de tipo `Function`. Se puede comprobar el tipo de un objeto mediante `typeof`:

```
typeof 5; // --> Number
typeof []; // --> Object
typeof Math.pow; // --> Function
```

Además, se utilizan algunas palabras para identificar variables nulas y no inicializadas: `null` y `undefined`, respectivamente. Para indicar resultados de operaciones matemáticas no válidas se usa `NaN` (o `Infinity`, en caso de divisiones por 0 o números demasiado grandes):

```
var sin_inicializar;
sin_inicializar; // --> undefined

Math.log(-1); // --> NaN
Math.pow(2, 10000); // --> Infinity
```

Control de flujo

Disponemos de las mismas estructuras que en el resto de lenguajes similares:

- if-else
- for
- for-in
- while
- do-while
- switch-case

La estructura *for-in* es algo diferente de sus equivalentes en C++ y Java, y además de arrays nos permite recorrer las propiedades de cualquier objeto:

```
var recetas = [
    {
        nombre: "Paella",
        ingrediente_principal: "arroz"
    },
    {
        nombre: "Tortilla",
        ingrediente_principal: "huevo"
    },
    {
        nombre: "Bocata",
        ingrediente_principal: "pan"
    }
];

for (var i in recetas) {
    // i referencia al índice!
    console.log("Receta nº " + i);

    for (var dato in recetas[i]) {
        // dato referencia la clave
        console.log(dato + ": " + recetas[i][dato]);
    }
}
```

Tenemos dos comparaciones de igualdad, una que convierte tipos implícitamente para comparar el contenido, y otra que es estricta con los tipos:

```
const POLO = 28;

// Comparación de igualdad:
POLO == "28"; // --> true

// Comparación de igualdad y tipo:
POLO === "28"; // --> false ("28" es String)
```

Por último, también podemos usar los operadores lógicos `||` y `&&` para control de flujo, algo muy utilizado para escribir código compatible con todos los navegadores:

```
var nodo = document.getElementById("identificador");
```

```
// Cada navegador aceptará innerText o textContent
var texto = nodo.innerText || nodo.textContent;
```

El Document Object Model

Acceso a los nodos HTML

Desde luego, si nuestro objetivo es modificar contenido en un sitio web e interactuar con el usuario, es necesario algo más que todo esto. Para esta tarea disponemos del *Document Object Model* o DOM, que básicamente se encarga de proveer la información de la página web de una forma que sea consistente con el lenguaje.

Así, tendremos disponibles una serie de objetos, partiendo de `window`, que nos proporcionarán acceso, entre otros, a lo siguiente:

- `window`: Es el padre de todos los objetos del documento. Es indistinto acceder a `window.objeto` y a `objeto`. Si declaramos una variable fuera de una función queda como objeto hijo de `window`.
- `title`: Título de la página.
- `location`
  - `location.href`: URL de la página.
  - `location.hash`: *hash* (la parte del URL que viene después del símbolo `#`)
- `document`: Acceso a los nodos HTML del documento.
  - `document.getElementByID(identificador)`: Obtiene un nodo mediante el identificador HTML (atributo `id`).
  - `document.querySelector(selector)`: Obtiene un nodo mediante un selector estilo CSS.
  - `document.createElement(tipo)`: Crea un nuevo nodo HTML del tipo especificado.

Nota : El DOM sólo está disponible cuando se ejecuta JavaScript en una página HTML en el cliente. No existe en otros intérpretes de JS como Rhino ni en la parte de servidor como en node.js.

Eventos

Además, para responder a interacciones del usuario disponemos de eventos, es decir, podemos asociar funciones a acciones de tipo *click*, *key press*, *drag*, etc. de forma que la página web responda ante ellas. Por ejemplo:

```
<!-- Nodo HTML: -->
<input id="mi_boton" type="button" value="Púlsame">

var alerta_click = function() {
    // this referencia al nodo que responde al evento
    console.log("Has pulsado el botón " + this.id);
};

document.querySelector("#mi_boton").onclick = alerta_click;

// Otra forma:
document.querySelector("#mi_boton").addEventListener("click", alerta_click);
```

Mucho más sobre eventos en [4].

Orientación a objetos: Los prototipos

Construyendo objetos

JavaScript está orientado a objetos, literalmente. No existen las clases en este lenguaje, sino que podemos crear objetos a partir de otros que llamamos *prototipos*, que están asociados a funciones constructoras. Estas funciones son idénticas a cualquier otra, salvo por que no devuelven un valor, y en su lugar obtenemos un objeto al utilizarlas con `new`. Veamos un ejemplo:

```
// El constructor
var Cancion = function(nombre, artista) {
    this.nombre = nombre;
    this.artista = artista;
};

// Definimos un método en el prototipo
Cancion.prototype.reproducir = function() {
    console.log("Reproduciendo " + this.nombre + " de " + this.artista);
};

// Creamos un objeto con el constructor
var una_cancion = new Cancion("Clocks", "Coldplay");
una_cancion.reproducir(); // --> "Reproduciendo Clocks de Coldplay"
```

Mediante la función constructora `Cancion` hemos creado un prototipo vacío (en `Cancion.prototype`), al que después hemos añadido el método `reproducir`.

[4] Introduction to Events - Quirks Mode

Al utilizar el prototipo como plantilla, cada objeto que se cree mediante `new Cancion(...)` dispondrá del método. Aun así, también podemos añadir otros métodos y atributos a cualquier objeto independientemente del prototipo con que se haya creado.

Visibildad de datos

Otra peculiaridad de la orientación a objetos de JS es la falta de encapsulación. Un dato miembro de un objeto, declarado con `this.dato` es visible en cualquier ámbito:

```
var inception = new Cancion("Time", "Hans Zimmer");

// Podemos modificar los datos miembro, e incluso los métodos
inception.nombre = "Mombasa";
inception.reproducir = function() {
    console.log("Banda sonora épica");
};
```

Una alternativa para ocultar información es utilizar variables en el constructor, de forma que solo están visibles en el ámbito de la función constructora. El problema es que entonces no estarán visibles para los métodos que declaremos en el prototipo. Podemos sin embargo definir *getters* dentro del constructor, y utilizarlos en los métodos:

```
var Cancion = function(nombre, artista) {
    var name = nombre;
    var artist = artista;

    this.nombre = function() {
        return name;
    }

    this.artista = function() {
        return artist;
    }
};

Cancion.prototype.reproducir = function() {
    console.log("Reproduciendo " + this.nombre() + " de " + this.artista());
};
```

Ahora las variables `name` y `artist` no son visibles ni modificables fuera del constructor, pero se pueden consultar mediante los métodos `nombre()` y `artista()`

respectivamente. Esto funciona porque aunque la función constructora se termine de ejecutar, las variables declaradas no desaparecen, y permanecen disponibles en los métodos *getters*.

Nota : La diferencia entre declarar métodos en el prototipo y declararlos en el constructor (con `this.metodo`) es que los del prototipo se definen una vez y se aplican a todos los objetos, mientras que los del constructor se definen para cada objeto creado, lo cual es menos eficiente. Por eso es preferible declarar los métodos en el prototipo[5].

Herencia

Como era de esperar, la herencia en JavaScript también es bastante particular. No existe una forma directa de hacer que una función "herede" de otra, por lo que no podemos crear constructores a partir de constructores anteriores. Sin embargo, podemos escribir constructores cuyo prototipo asociado sea una instancia de uno ya existente[6]:

```
var Parent = function() {
    console.log("Creando objeto de la clase padre");
};
Parent.prototype.metodo = function() {
    return "Este método se heredará";
};


// Constructor nuevo
var Child = function() {
    console.log("Este es el constructor de la clase hija");
};

// Usamos el prototipo de la clase padre
Child.prototype = new Parent();
// ...pero asociándolo al constructor de la clase hija
Child.prototype.constructor = Child;

// Podemos añadir métodos
Child.prototype.nuevo_metodo = function() {
    return "Solo en la clase hija";
};

var partest = new Parent();
partest instanceof Parent; // --> true
```

[5] Efficient encapsulation of JavaScript objects
[6] Javascript Constructors and Prototypes - Toby Ho

```
partest instanceof Child; // --> false

var chitest = new Child();
chitest instanceof Parent; // --> true
chitest instanceof Child; // --> true
```

En este ejemplo hemos creado dos constructores, uno con prototipo propio, y otro al que asociamos un objeto creado a partir del primer prototipo. Comprobamos mediante `instanceof` que la herencia funciona como se esperaba, un objeto creado mediante `Child` es instancia de este constructor y del padre, `Parent`, y dispone de los métodos de ambos en su prototipo.

Conclusión

En este artículo hemos visto cómo funciona JavaScript y cómo interactúa con un sitio web, y hemos estudiado sus posibilidades desde el punto de vista de la orientación a objetos. Si queréis saber más sobre el lenguaje y aprender a programar en él, serán muy útiles las referencias a páginas web, y os puedo recomendar el libro [7], algo anticuado ya pero muy completo y conciso.

[7] JavaScript - Revisado y actualizado 2004, José Manuel Alarcón