Programación

Objetos avanzados

Undécima semana

Enero 2022

Cruz García, Iago

Introducción

Hasta ahora trabajamos con objetos nativos al lenguaje JavaScript, utilizando funciones para la declaración de un nuevo objeto y la palabra reservada **new** para instanciar o crear una variable de ese tipo de objeto. Esto era relativamente sencillo, pero si queríamos utilizar la herencia teníamos que pasar por un proceso engorroso, modificando los valores del prototipo de cada "hijo" con los valores del "padre".

Con ECMAScript6, el estándar actual del lenguaje, tenemos una nueva forma de crear clases y de heredar funciones y atributos mucho más directa, pero siempre es necesario reconocer y trabajar con las características antiguas, pues estas funciones solo están disponibles siempre que el navegador siga el estándar ECMAScript 6.

Como dato, el uso de **let** y **const** también son propios de este estándar.

Recomendación: A partir de ahora es necesario que a la hora de realizar ejercicios nos acostumbremos a buscar información en la documentación oficial. En caso de no poder resolver las dudas con la documentación ofrecida, el siguiente paso será preguntar las dudas en el foro de clase.

Documentación oficial de JavaScript

Foro de la asignatura

ECMAScript 6

ECMAScript 6 en W3CSchools

Clases

Hasta ahora hemos llamado "objeto" tanto a la creación en código del mismo como a la instanciación, cuando se convierte en una variable. Esto no es incorrecto, pero la nomenclatura más correcta es la siguiente:

- La declaración del objeto, su programación, creación de atributos y funciones, su creación desde 0, recibe el nombre de **Clase**.
- La instanciación de una **clase**, creando una variable de dicho tipo, es lo que llamaremos **Objeto.**

Podemos verlo tal que una **Clase** es la plantilla y el **objeto** es la creación física de la plantilla.

Vamos a ver cómo trabajaremos con los objetos ahora:

```
class Instrumento {
        constructor(nombre, tono, material) {
        this.nombre = nombre;
        this.tono = tono;
        this.material = material;
        }

        tocarMelodia() {
        let mensaje;
        mensaje = "La melodía del/la " + this.nombre +
        " es debido a su tono " + tono +
        " y su construcción de " + material;
        return mensaje;
        }
}
```

Vamos a desgranar este código y compararlo con la estructura que hemos visto hasta ahora.

- *class Instrumento{...}:* La declaración de una nueva clase. Antes utilizábamos "var Instrumento = function(){...}".
- *constructor (...){...}:* Realiza la misma función que el siguiente código:

```
var Instrumento  = function(nombre, tono, material){
    this.nombre = nombre;
    this.tono = tono;
    this.material =material;
}
```

En este caso, *constructor* se separa de la declaración de la función para facilitar su lectura en el código

- *tocarMelodia(){...}:* En esta ocasión, la única diferencia con la estructura anterior (function tocarMelodia(){...}) es la desaparición del uso de la palabra clave *function*, aunque la construcción será la misma, soportando parámetros de entrada y de salida.

Vemos que la construcción es muy parecida a la vista hasta el momento, pero haciendo mucho más sencilla la nomenclatura.

Herencia

Una pieza fundamental de la Programación Orientada a Objetos es la herencia como sabemos, pues permite generalizar atributos y funciones y que los compartan muchos objetos, evitando así reescribir código sin sentido.

Con la estructura anterior, necesitábamos hacer usos de los prototipos, accediendo a funciones "ocultas" de JavaScript para que se transmitiesen los valores de "padre" a "hijo", utilizando bucles y *prototype*.

La estructura mediante clases facilita esto una vez más, pues no hará falta recurrir al acceso del atributo *prototype*, si no que utilizaremos la palabra clave **extends**, que indica que el objeto B extiende los atributos y funciones del objeto A. Teniendo en cuenta la clase Instrumento creada antes:

```
class Guitarra extends Instrumento {
        tocarGuitarra() {
        let mensaje;
        mensaje = "La guitarra suena genial";
        return mensaje;
        }
}

var lesPaul = new Guitarra("Gibson", "Estándar", "Madera");
console.log (lesPaul.tocarGuitarra());
console.log (lesPaul.tocarMelodia());
```

Podemos corroborar que se hereda **todo** del "padre" Instrumento: Guitarra carece de constructor, pero lo heredó de Instrumento, no declaramos el

método "tocarMelodia" pero podemos usarlo y por supuesto los atributos "nombre", "tono" y "material" son accesibles.

Super

Un nuevo añadido que nos permite el uso de clases es el uso de la palabra reservada **super**, utilizada en el método constructor. Utilizaremos esta a la hora de recrear un objeto "hijo" que herede de un "padre" cuando nos interesen, además de los atributos heredados, otros nuevos. Usemos el ejemplo anterior de la guitarra:

```
class Guitarra extends Instrumento {

        constructor(nombre, tono, material, nCuerdas) {
        super(nombre, tono, material);
        this.nCuerdas = nCuerdas;
        }

        tocarGuitarra() {
        let mensaje;
        mensaje = "La guitarra suena genial gracias a las " + this.nCuerdas + "
cuerdas";
        return mensaje;
        }
}
```

Como vemos, **super** hace referencia al constructor del padre, por lo que necesita que indiquemos qué valores recibirá (de ahí los parámetros de entrada).

Getter y setter

Habíamos realizado antes las funciones de obtención y modificación de atributos como métodos normales. Ahora utilizarán una palabra clave, **get y set** respectivamente para diferenciarse y se accederá como atributos, en vez de funciones, tal que:

```
class Guitarra extends Instrumento {
        constructor(nombre, tono, material, nCuerdas) {
        super(nombre, tono, material);
        this.nCuerdas = nCuerdas;
        }

        get getNCuerdas() {
        return this.nCuerdas;
        }

        set setNCuerdas(_nCuerdas) {
        this.nCuerdas = _nCuerdas;
        }

        tocarGuitarra() {
        let mensaje;
        mensaje = "La guitarra suena genial gracias a las " + this.nCuerdas + "
cuerdas";
        return mensaje;
        }
}

var lesPaul = new Guitarra("Gibson", "Estándar", "Madera", 6);
console.log (lesPaul.tocarGuitarra());
console.log (lesPaul.tocarMelodia());
lesPaul.setNCuerdas = 12;
console.log(lesPaul.getNCuerdas);
```

Static

Por último vamos a ver el uso de una palabra clave menos utilizada pero muy interesante: **static**. Casi como su traducción del inglés, *estático/a*, índica que el método precedido por esta palabra será estático. Esto quiere decir que las variables internas de ese método se compartirán entre todos los objetos del mismo tipo, tal que:

```
class Guitarra extends Instrumento {
        constructor(nombre, tono, material, nCuerdas) {
        super(nombre, tono, material);
        this.nCuerdas = nCuerdas;
        }

        get getNCuerdas() {
        return this.nCuerdas;
        }

        set setNCuerdas(_nCuerdas) {
        this.nCuerdas = _nCuerdas;
        }

        static numeroDeGuitarrasCreadas() {
        if (!this.nGuitarras && this.nGuitarras !== 0) {
        this.nGuitarras = 0;
        } else {
        this.nGuitarras++;
        }
        return this.nGuitarras+1;//Para saltarnos el 0
        }

        tocarGuitarra() {
        let mensaje;
        mensaje = "La guitarra suena genial gracias a las " + this.nCuerdas + "
```

```
cuerdas";
        return mensaje;
        }
}

var lesPaul = new Guitarra("Gibson", "Estándar", "Madera", 6);
console.log (lesPaul.tocarGuitarra());
console.log (lesPaul.tocarMelodia());
lesPaul.setNCuerdas = 12;
console.log(lesPaul.getNCuerdas);
console.log(lesPaul.numeroDeGuitarrasCreadas());

var stratoCaster = new Guitarra ("Fender", "Estándar", Madera", 8);
console.log(lesPaul.numeroDeGuitarrasCreadas());
```

En la primera llamada, nos mostrará "1" y en la segunda "2".