JAVASCRIPT FRONTEND AVANZADO
CLASE 9 : OOP

MÉTODO CLÁSICO

HERENCIA

Recordemos entonces que desde el punto de vista de Javascript, consideramos que dos objetos son “instancias” de la misma “clase” si ambos comparten el mismo prototipo, no necesariamente habiendo sido creados por el mismo constructor. Esto implica que si quisiéramos acercarnos a recrear un modelo clásico de orientación a objetos usando el paradigma de prototipos, consideraríamos que la herencia implica únicamente conseguir que todos los objetos de una función constructora compartan el mismo prototipo. Por lo tanto, si queremos que una función constructora herede de otra y al mismo tiempo pueda compartir estas nuevas habilidades con sus propias instancias, tenemos que modificar su prototype de manera tal que creemos manualmente una cadena de herencia por prototipos. Observemos el siguiente ejemplo :

```
function Persona(nombre,edad){
   this.nombre = nombre
   this.edad = edad
   Persona.prototype.saludo = function(){
       console.log("Hola, mi nombre es "+this.nombre)
   }
}
```

Como ya sabemos, todos los objetos creados por el constructor new Persona van a tener dos propiedades propias únicas de cada instancia(nombre y edad) y una propiedad compartida en el prototipo de todas que es una función saludo. Nótese que no estamos compartiendo propiedades que sí corresponde que tengan valor único, es decir imaginemos por un momento que la propiedad nombre se encuentra en el prototype del constructor, esto haría que si llegaramos a asignar esa variable con un valor estático, todas las instancias van a tener el mismo nombre. Entonces :

```
function Persona(nombre,edad){
   this.edad = edad
```

```
    Persona.prototype.nombre = nombre
    Persona.prototype.saludo = function(){
        console.log("Hola, mi nombre es "+this.nombre)
    }
}

var persona_1 = new Persona("Juan",30)
var persona_2 = new Persona("Ana",30)
persona_1.saludo() //Ana
```

Teniendo esto en cuenta , podríamos considerar incluso componer nuestro prototype con lo que esté en el prototype del constructor del que queremos heredar, ya que evidentemente la herencia se transmite más fácilmente por prototipos de prototipos compartiendo todas aquellas propiedades que sean de igual valor para todas las instancias :

```
function Persona(nombre,edad){
    this.edad = edad
    Persona.prototype.nombre = nombre
    Persona.prototype.saludo = function(){
        console.log("Hola, mi nombre es "+this.nombre)
    }
}

function Empleado(sueldo){
    this.sueldo = sueldo
    Empleado.calcular = function(){
        //...
    }
}

Empleado.prototype = Persona.prototype
```

Entonces si crearamos nuevos empleados con el constructor new, podríamos observar que nuestros empleados pueden acceder a sus propios métodos(calcular) y a los métodos del constructor “padre” (saludo).

Si observamos en la consola de desarrollo, nos queda un prototipo conjunto, es decir que el objeto contiene todas las propiedades hijas y padre sin importarle el nivel de capa de prototipo de donde venía. Para corregir este tipo de diseño podríamos implementar un Object.create si estuviera disponible :

```
Empleado.prototype = Object.create(Persona.prototype)
Empleado.prototype.constructor = Empleado
//SubClass.prototype = Object.create(SuperClass.prototype)
```

Podemos notar también que volvemos a redefinir el constructor ya que de otra forma perdemos referencia del mismo, aunque esto no implica que tengamos problemas a futuro en nuestro código.

COMPOSICIÓN

Recordemos que nuestra herencia no involucra el hecho de implementar la misma interfaz de propiedades pero sí de métodos compartidos por lo que si además queremos poder las mismas propiedades que nuestro constructor padre podríamos hacer uso de la composición de clases por medio de los métodos apply ó call :

```
function Empleado(sueldo,nombre,edad){
    //Composición
   //Ejecutamos la funcion constructora Persona redefiniendole el contexto con el que actualmente estemos usando en nuestro operador new, al cual se le van a asignar las propiedades de la clase padre
   Persona.call(this,nombre,edad)

   this.sueldo = sueldo
   Empleado.calcular = function(){
       //...
   }
```

De esta manera volvemos a obtener las propiedades originales de nuestro constructor Persona pudiendo así conservar esa “plantilla” del método clásico.

CLASES

Las clases de javascript son introducidas en el ECMAScript 2015 y 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 a JavaScript. Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia [1].

Las clases son de hecho "funciones especiales", tal y como el caso de las expresiones de funciones y declaraciones de funciones, la sintaxis de la clase tiene dos componentes:

- expresiones de clases y
- declaraciones de clases.

DECLARACIONES DE CLASE

Una manera de definir una clase es mediante una *declaración de clase.* Para la declaración de una clase, es necesario el uso de la palabra reservada `class` y un nombre para la clase ("Poligono" en esté caso) :

```
class Poligono {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
}
```

EXPRESION DE CLASE

Una **expresión de clase** es otra manera de definir una clase. Las expresiones de clase pueden ser nombradas o anónimas. El nombre dado a la **expresión de clase**nombrada es local dentro del cuerpo de la misma:

```
var Poligono = class {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
};
```

El *cuerpo* de una **clase** es la parte que se encuentra entre las llaves {}. Este es el lugar donde se definen los **miembros de clase,** como los **métodos** o **constructores.**

El cuerpo de las *declaraciones de clase* y las *expresiones de clase* son ejecutadas en modo estricto.

CONSTRUCTOR

El método constructor es un método especial para crear e inicializar un objeto creado con una clase. Solo puede haber un método especial con el nombre "constructor" en una clase. Si esta contiene más de una ocurrencia del método **constructor,** se arrojará un *Error* SyntaxError.

Un **constructor** puede usar la palabra reservada **super** para llamar al **constructor** de una superclase.

```
class Poligono {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  get area() {
    return this.calcArea();
  }

  calcArea() {
    return this.height * this.width;
  }
}

const cuadrado = new Poligono(10, 10);

console.log(cuadrado.area);
```

Nótese cómo en este ejemplo se está haciendo uso de un getter. Getters y Setters se declaran de la misma forma que en el módulo anterior de prototipos donde reemplazamos la palabra función por get o set y el nombre de la misma era igual al nombre de la propiedad que representaba.

SUBCLASES

La palabra clave `extends` es usada en *declaraciones de clase* o *expresiones de clase* para crear una clase hija :

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hablar() {
    console.log(this.nombre + ' hace un ruido.');
  }
}

class Perro extends Animal {
  hablar() {
    console.log(this.nombre + ' ladra.');
  }
}
```

MODO ESTRICTO

El modo estricto de ECMAScript 5 es una forma de optar *explícitamente* por una variante restringida de JavaScript. El modo estricto no es sólo un subconjunto: *intencionalmente* tiene diferente semántica que código normal. Los navegadores que no soportan el modo estricto ejecutarán el código con un modo diferente de los que sí lo soportan, así que no hay que usar en el modo estricto sin hacer pruebas previas. Código en modo estricto y en no estricto pueden coexistir, de modo que código puede migrar a modo estricto incrementalmente.

El modo estricto hace varios cambios en la semántica normal de JavaScript. Primero, elimina algunos errores silenciosos de JavaScript haciendo que lancen excepciones. Segundo, corrige errores que hacen que sea difícil para los motores de JavaScript realizar optimizaciones: a veces, el código escrito en modo estricto puede correr más rápido que el que no es estricto. Tercero, el modo estricto prohíbe cierta sintaxis que es probable que sea definida en futuras versiones de ECMAScript [3].

El modo estricto se aplica a un *script por completo* o a *funciones individuales.* No se aplica a bloques entre corchetes {} ; intentar aplicarlo en tales contextos no hace nada. Código `eval`, código `Function`, atributos de manejadores de eventos, cadenas pasadas a

`setTimeout`, y similares son scripts enteros, de modo que invocar modo estricto en tales contextos funciona como se espera.

Para invocar el modo estricto en todo un script, escriba *exactamente* `"use strict";` (o `'use strict';`) antes de cualquier otra sentencia.

```
"use strict";
var v = "Hola! Soy un modo estricto para script!";
```

Esta sintaxis tiene un problema que ya ha afectado a cierta página bien conocida: no es posible concatenar ciegamente scripts no confictivos entre sí. Si concatena un script en modo estricto con otro que no lo es, la concatenación de ambos producirá código en modo estricto. Lo contrario también es cierto: código en modo no estricto mas código estricto produce código que no es estricto. Concatenar scripts no produce problemas si todos están en modo estricto (o si todos están en modo no estricto). El problema es mezclar scripts en modo estricto con scripts en modo no estricto. Por eso se recomienda habilitar el modo estricto a nivel de función solamente (al menos durante el periodo de transición de un programa).

Otra opción es envolver el contenido completo del script en una función y tener esa función externa en modo estricto. Así se elimina el problema de la concatenación, pero entonces usted tiene que hacerse cargo de exportar explícitamente las variables globales fuera del alcance de la función.

De forma similar, para invocar el modo estricto para una función, escriba *exactamente* `"use strict"` (o `'use strict';`) en el cuerpo de la función antes de cualquier otra sentencia.

```
function strict(){
  // sintaxis estricta a nivel función
  'use strict';
  function nested() { return "y yo también!"; }
  return "Hola, estoy en modo estricto  " + nested();
}
function notStrict() { return "Yo no soy estricto."; }
```

IIFE

IIFE (Expresión de función invocada inmediatamente ó Immediately Invoked Function Expression) es una función de JavaScript que se ejecuta tan pronto como se define. Es un patrón de diseño que también se conoce como Función anónima autoejecutable y contiene dos partes principales. La primera es la función anónima con alcance léxico incluido dentro

del operador de agrupamiento (). Esto evita el acceso a variables dentro del lenguaje IIFE y la contaminación del alcance global. La segunda parte está creando la expresión de función de ejecución inmediata (), a través de la cual el motor de JavaScript interpretará directamente la función.

La función se convierte en una expresión de función que se ejecuta inmediatamente. No se puede acceder a la variable dentro de la expresión desde afuera :

```
(función () {
     var aName = "Juan";
}) ();
// El nombre de la variable no es accesible desde el alcance
externo
aName // arroja "UnEught ReferenceError: aName no está definido"
```

Asignar el IIFE a una variable no lo almacena sino su resultado :

```
var result = (function () {
    var name = "Juan";
    return name;
})();
// Crea el resultado:
result; // "Juan"
```

En primer lugar, el modo estricto hace imposible crear variables globales por accidente. En JavaScript no estricto, si se escribe mal una variable en una asignación, se creará una nueva propiedad en el objeto globlal y el código continuará "trabajando" como si nada (aunque es posible que el código asi escrito falle en el futuro, en concreto, en JavaScript más moderno). En modo estricto, cualquier asignación que produzca variables globales por accidente lanzará un error:

```
"use strict";
                      // Asumiendo que exista una variable global
llamada mistypedVariable
mitipodeVariable = 17; // esta línea lanza un ReferenceError debido a
                      // una errata en el nombre de la variable
```

Por otro lado, el modo estricto lanza una excepción al intentar eliminar propiedades no eliminables (mientra que en código normal el intento no tendría ningún efecto):

```
"use strict";
```

```
delete Object.prototype; // lanza TypeError
```

Además, el modo estricto requiere que los nombres de los parámetros de una función sean únicos. En código normal, el último argumento repetido oculta argumentos anteriores con el mismo nombre. Estos argumentos permanecen disponibles a través de `arguments[i]`, de modo que no son completamente inaccesibles. Aun así, esta ocultación tiene poco sentido y es probablemente indeseable (pues puede ocultar por ejemplo un error al teclear una letra). Por lo tanto, en modo estricto, duplicar nombres de argumentos es un error de sintaxis.

```
function sum(a, a, c){ // !!! syntax error
  "use strict";
  return a + b + c; // erróneo si éste código se ejecuta
}
```

Optimizando un poco el código entonces podría quedarnos un archivo de tipo módulo de la siguiente forma :

```
(function(window){

   let personas = []

   class Persona {

       constructor(nombre,edad){
           personas.push(nombre)
           this.nombre = nombre
           this.edad = edad
           this.getNombre = function(){
               return nombre;
           }
       }

       getPersonas(){
           return personas;
       }

   }
```

```
   window.Persona = Persona
})(window)
```

De esta forma podemos tener acceso a un closure donde guardar variables privadas que nadie tenga acceso solo a través de la interfaz que nosotros definimos.

1. https://es.wikipedia.org/wiki/Single-page_application
2. https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes
3. https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Modo_estricto