Tema 4 Objetos Nativos en Javascript

Practica 04-05 Objetos ECMA2015 o ES6

Nombre		Curso	
Apellidos		Fecha	

Clases

Las clases de javascript, introducidas en ECMAScript 2015, 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 lidiar con la herencia.

Definiendo clases

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

Declaración de clases

Una manera de definir una clase es mediante una **declaración de clase**. Para declarar una clase, se utiliza la palabra reservada class y un nombre para la clase "Rectangulo".

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

Alojamiento

Una importante diferencia entre las **declaraciones de funciones** y las **declaraciones de clases** es que las **declaraciones de funciones** son alojadas y las **declaraciones de clases** no lo son. En primer lugar necesitas declarar tu clase y luego acceder a ella, de otro modo el ejemplo de código siguiente arrojará un ReferenceError:

```
const p = new Rectangle(); // ReferenceError

class Rectangle {}
```

Expresiones de clases

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.

```
// Anonima
let Rectangulo = class {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
};

console.log(Rectangulo.name);
// output: "Rectangulo"

// Nombrada
let Rectangulo = class Rectangulo2 {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
};
console.log(Rectangulo.name);
// output: "Rectangulo2"
```

Nota: Las **expresiones** de clase están sujetas a las mismas restricciones de elevación que se describen en la sección Class declarations.

Cuerpo de la clase y definición de métodos

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

Modo estricto

El cuerpo de las *declaraciones de clase* y las *expresiones de clase* son ejecutadas en modo estricto. En otras palabras, el código escrito aquí está sujeto a una sintaxis más estricta para aumentar el rendimiento, se arrojarán algunos errores silenciosos y algunas palabras clave están reservadas para versiones futuras de ECMAScript.

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 mas 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*

Métodos prototipo

la *palabra reservada* **get** para llamar al metodo de lectura de una propiedad
la *palabra reservada* **set** para llamar al metodo de escritura de una propiedad

```
class Rectangulo {
  constructor (alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
  // Getter
  get area() {
     return this.calcArea();
   }
  // Setter
  set area(value) {
            this.height = this.width = Math.sqrt(value);
}

  // Método
  calcArea () {
    return this.alto * this.ancho;
  }
}
const cuadrado = new Rectangulo(10, 10);
console.log(cuadrado.area); // 100

cuadrado.area=27
console.log(cuadrado.area); // 27
```

Métodos estáticos

La *palabra clave* static define un método estático para una clase. Los métodos estáticos son llamados sin instanciar su clase y **no** pueden ser llamados mediante una instancia de clase. Los métodos estáticos son a menudo usados para crear funciones de utilidad para una aplicación.

```
class Punto {
  constructor ( x , y ){
    this.x = x;
    this.y = y;
  }

  static distancia ( a , b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;

    return Math.sqrt ( dx * dx + dy * dy );
  }
}

const p1 = new Punto(5, 5);
const p2 = new Punto(10, 10);

console.log (Punto.distancia(p1, p2)); // 7.0710678118654755
```

"Boxing" con prototipos y métodos estáticos

Cuando un método estático o método del prototipo es llamado sin un valor para "this" (o con "this" como booleano, cadena, número, undefined o null), entonces el valor de "this" será **undefined** dentro de la funciona llamada.

Autoboxing no ocurrirá. El comportamiento será igual inclusive si se escribe el código en modo no estricto. **(NO PODEMOS SACAR LOS METODOS DEL CONTEXTO DEL OBJETO)**

```
class Animal {
  hablar() {    return this;   }
  static comer() {    return this;  }
}

let obj = new Animal();
obj.hablar(); // Animal {}
let hablar = obj.hablar;
hablar(); // undefined

Animal.comer() // class Animal
let comer = Animal.comer;
comer(); // undefined
```

Si se escribe el código del cuadro superior usando clases función tradicionales, entonces autoboxing ocurrirara porque tomará valor de "this" sobre la función que es llamada.

```
function Animal() { }

Animal.prototype.hablar = function(){  return this; }
Animal.comer = function() {   return this;   }

let obj = new Animal();
let hablar = obj.hablar;
hablar(); // global object

let hablar = Animal.hablar;
hablar(); // global object
```

Subclases con extends

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.');   }   }
```

También se pueden extender las clases tradicionales basadas en funciones:

```
function Animal (nombre) {
  this.nombre = nombre;
} // Objeto Tradicional

Animal.prototype.hablar = function () {
  console.log(this.nombre + 'hace un ruido.');
} //  añadimos la funcion hablar dinamicamente con  prototype

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

var p1 = new Animal('Mitzie');
var p = new Perro('Mitzie');
p1.hablar();    //  Mitzie hace un ruido
p.hablar();  //  Mitzie hace un ruido .  Mitzie  ladra
```

Fijarse que las clases no pueden extender objectos regulares (literales). Si se quiere heredar de un objecto regular, se debe user Object.setPrototypeOf(hijo.prototype, padre)::

```
var Animal = {
  hablar() {    console.log(this.nombre + ' hace ruido.');   },
  comer() {   console.log(this.nombre + ' se alimenta.');  }
};

class Perro {        constructor(nombre) {    this.nombre = nombre;   }
                      hablar() {    console.log(this.nombre + ' ladra.');   }
}

// Solo adjunta los métodos aún no definidos
Object.setPrototypeOf(Perro.prototype, Animal);

var d = new Perro('Mitzie');
d.hablar(); // Mitzie ladra.
d.comer(); // Mitzie se alimenta.
```

Especies

Quizás se quiera devolver objetos Array derivados de la clase array MyArray. **El patron *species* permite sobreescribir constructores por defecto.**

Por ejemplo, cuando se usan metodos del tipo map() que devuelven el constructor por defecto, se quiere que esos métodos devuelvan un objeto padre Array, en vez de MyArray. El símbolo **Symbol.species** permite hacer:

```
class MyArray extends Array {
  // Sobreescribe species sobre el constructor padre Array
  static get [Symbol.species]() { return Array; }
}

var a = new MyArray(1,2,3);
var mapped = a.map(x => x * x);

console.log(mapped instanceof MyArray); // false
console.log(mapped instanceof Array);   // true
```

Llamadas a súperclases con super

La palabra clave super es usada para llamar funciones del objeto padre.

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

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

class Leon extends Gato {

  hablar() {
    super.hablar();
    console.log(this.nombre + ' rugeeeee.');
  }
}
```

Mix-ins

Subclases abstractas or *mix-ins* son plantillas de clases. Una clase ECMAScript solo puede tener una clase padre, con lo cual la herencia multiple no es posible.

Mixin – en programación orientado a objetos es
una clase que contiene métodos para otras clases.

Algunos lenguajes permiten la herencia múltiple. JavaScript no admite la herencia múltiple, pero los mixins se pueden implementar copiando métodos en el prototipo.

Podemos usar mixins como una forma de expandir una clase agregando múltiples comportamientos, como el manejo de eventos que hemos visto anteriormente.

Los mixins pueden convertirse en un punto de conflicto si sobrescriben accidentalmente los métodos de clase existentes. Por lo tanto, generalmente debes planificar correctamente la definición de métodos de un mixin, para minimizar la probabilidad de que suceda.

Los mixins son un conjunto de funciones separados por comas y acotadas con { } al asignarlas les pones un nombre que podemos asignar a un objeto para que pueda hacer uso de las funciones que incluye.

```
let EjemplodeMixin = { funcion1() {} , funcion2() {} , funcion3() {} }
Class ObjetoQueTomaLasFuncionesDelMixin { }
```

Se asignan a un objeto con las siguiente sintasix

```
Object.assign(ObjetoQueTomaLasFuncionesDelMixin.prototype, EjemplodeMixin);

// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hola ${this.name}`);
    alert('Hola '+this.name);
  } ,
  sayBye() {
    alert(`Adiós ${this.name}`);
  }
,
  sayBuenastardes() {
    alert(`Buenas tardes ${this.name}`);
   }

};

// uso:
class User {
  constructor(name) {
    this.name = name;
  }
}
```


```
// copia los métodos
Object.assign(User.prototype, sayHiMixin);

// Ahora el User puede decir hola
new User("tío").sayHi(); // Hola tío!
new User("Colega").sayBye(); //
```

Ejercicio 1. Modifica el ejercicio anterior para usar las propiedades class y extend

Ejercicio 2. Crea una alumno con los siguientes valores metodos y propiedades

1. Propiedades: nombre, apellidos, id, NombreModulos, notasModulos (estos dos son dos arrays paralelos)
2. Metodos:
    1. set y get de las propiedades nombre, apellidos, id
    2. Mostrar modulos(id): muestra el nombre de los modulos : devuelve un string con los modulos
    3. nota(nombremodulo) :devuelve un string con la nota del modulo
    4. media(id) te muestra su media
    5. suspensas(id): te muestra las notas suspensas
    6. Aprobadas(id): te muestra las notas aprobadas

Ejercicio 3. Crea un objeto clase que sea un array de alumnos con los metodos (con notacion anonima)

1. matricular(alumno)
2. eliminar(alumnos)
3. numero(alumnos)
4. Mostrar(id)

Ejercicio 4. Crea dos clases que hereden de alumno y se llamen alumnofp (debe tener un atributo empresaFct un metodo que se llame hacerFCT(empresa)) y otra que herede de esta ultima y se llame alumnofpDual (debe tener un atributo empresaFctDual debe tener un metodo que se llame hacerFCTDual(EmpresaDual))

Ejercicio 5. Crea una clase mixin llamada matricular con un metodo matricula y asignala las tres clases anteriores.