

INSTITUTO TECNOLÓGICO SUPERIOR

DESARROLLO DE SOFTWARE

BASE DE DATOS

ESTUDIANTE: SILVA ORTEGA JUAN FRANCISCO

PRIMERO “B”

2018

**CLASES DE JS**

## Definiendo clases

Las clases son de hecho "[funciones](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions) especiales", tal y como el caso de las [expresiones de funciones](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) y [declaraciones de funciones](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function), la sintaxis de la clase tiene dos componentes:

* [expresiones de clases](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/class) y
* [declaraciones de clases](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class).

### **Declaración de clases**

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;

}

}

#### Izado (Hoisting)

Una importante diferencia entre las **declaraciones de funciones** y las **declaraciones de clases** es que las **declaraciones de funciones** son [izadas](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) y las **declaraciones de clases** no lo son. En primer lugar necesitas declarar tu clase y luego acceder a ella, de otra modo el ejemplo de código siguiente arrojará un [ReferenceError](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/ReferenceError):

var p = new Poligono(); // ReferenceError

class Poligono {}

### **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

var Poligono = class {

constructor(alto, ancho) {

this.alto = alto;

this.ancho = ancho;

}

};

// Nombrada

var Poligono = class Poligono {

constructor(alto, ancho) {

this.alto = alto;

this.ancho = ancho;

}

};

## Cuerpo de la clase y definición de métodos

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.**

### **Modo estricto**

El cuerpo de las declaraciones de clase y las expresiones de clase son ejecutadas en [modo estricto.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

### **Constructor**

El método [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/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](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/SyntaxError)

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

### **Métodos prototipo**

Vea también [métodos definidos](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions).

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);

### **Métodos estáticos**

La palabra clave [static](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static) define un método estático para una clase. Los métodos estáticos pueden ser llamados sin instanciar la clase ni una vez instanciada. 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));

### **"Boxing" con prototipos y métodos estáticos**

Cuando un métodos estático o del prototipo es llamado sin un objecto evaluado "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 inclose si se escribe el código en modo no estricto.

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

let obj = new Animal();

let speak = obj.speak;

speak(); // undefined

let eat = Animal.eat;

eat(); // 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.speak = function(){

return this;

}

Animal.eat = function() {

return this;

}

let obj = new Animal();

let speak = obj.speak;

speak(); // global object

let eat = Animal.eat;

eat(); // global object

## Subclases con extends

La palabra clave [extends](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/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;

}

Animal.prototype.hablar = function () {

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

}

class Perro extends Animal {

hablar() {

super.hablar();

console.log(this.nombre + ' ladra.');

}

}

var p = new Perro('Mitzie');

p.hablar();

Fijarse que las clases no pueden extender objectos regulares (literales). Si se quiere heredar de un objecto regular, se debe user [Object.setPrototypeOf()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/setPrototypeOf)::

var Animal = {

hablar() {

console.log(this.nombre + 'hace ruido.');

}

};

class Perro {

constructor(nombre) {

this.nombre = nombre;

}

hablar() {

console.log(this.nombre + ' ladra.');

}

}

Object.setPrototypeOf(Perro.prototype, Animal);

var d = new Perro('Mitzie');

d.hablar();

## Especies

Quizás se quiera devolver objectos [Array](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array) derivados de la clase array MyArray. El patron species permite sobreescribir constructores por defecto.

Por ejemplo, cuendo usando metodos del tipo [map()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map) que devuelve el constructor por defecto, se quiera que esos métodos devulvan un objeto padre Array, en cambio de MyArray. El símbolo [Symbol.species](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/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

## Objetos de subclase built-in

TBD

## Super class calls with super

La palabra clave [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) es usada para llamar funciones del objeto padre.

class Gato {

constructor(nombre) {

this.nombre = nombre;

}

hablar() {

console.log(this.nombre + ' hace ruido.');

}

}

class Leon extends Gato {

hablar() {

super.hablar();

console.log(this.nombre + ' maulla.');

}

}

## 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. La funcionalidad debe ser proporcionada por la clase padre.

Una función con una clase padre como entrada y una subclase extendiendo la clase padre como salida puede ser usado para implementar mix-ins en EMCAScript:

var calculatorMixin = Base => class extends Base {

calc() { }

};

var randomizerMixin = Base => class extends Base {

randomize() { }

};

Una clase que use este método puede ser escrita tal que así:

class Foo { }

class Bar extends calculatorMixin(randomizerMixin(Foo)) { }

<script>

class Human {

}

class Tutor{

constructor(nombre,last\_name){

this.nombre=nombre;

this.last\_name = last\_name;

}

full\_name(){

return this.nombre + " " + this.last\_name;

}

}

var tutor = new Tutor("JUAN" , "SILVA");

console.log ( full\_name());

</script>

**https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes**

https://www.youtube.com/watch?v=tzmmlAhSEK4