

# Javascript

Marzo, 2019



# Agenda

- Que és?
- Prototipos vs Extensión de clases
- Herencia vs delegación
- Funciones como componente de primera clase
- Alcances globales y localizados

OOP: es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulamiento.

OOP: es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulamiento.

**Pero no necesariamente tiene que implementarse mediante clases**

La programación por prototipos es una forma de programación orientada a objetos en el que el reuso del comportamiento es a través de objetos ya existentes.

**Javascript** es un lenguaje de programación que usualmente es clasificado como un lenguaje con un modelo de objetos basado en prototipos (prototype-based object model).

# Modelo de clases vs Modelo de prototipos

- Estático vs dinámico: la clase es estática mientras que el prototipo es dinámico, lo que permite tener múltiples ventajas:
  - Agregar comportamiento y estado durante la ejecución
  - Modificar todas las instancias del mismo prototipo a la vez
  - Crear nuevos prototipos de forma dinámica
  - Y más...

# Algunos Lenguajes Prototipados

75.10 - Técnicas de Diseño - FI.UBA

Lenguaje **Self**: desarrollado por David Ungar y Randall Smith

Y lo podemos ver implementado en lenguajes como:

Javascript

Pascual

Cecil

NewtonScript, Io, MOO, REBOL, Squeak, y otros

# Javascript - Un poco de historia..

Javascript surge ante la necesidad de hacer que el HTML tuviera una interacción con el usuario más completa y de disminuir la cantidad de pedidos al servidor.

En 1995, surgen dos opciones: Java Applets y LiveScript, creado por Brendan Eich que luego fue incluido en Netscape 2.0 bajo el nombre de 'JavaScript'. Los applets no tuvieron mucho éxito pero sí lo tuvo JavaScript.

### ¿Cómo hacemos que un objeto herede de otro objeto?

Cuando vemos que cierto comportamiento entre objetos se repite, que hacemos?

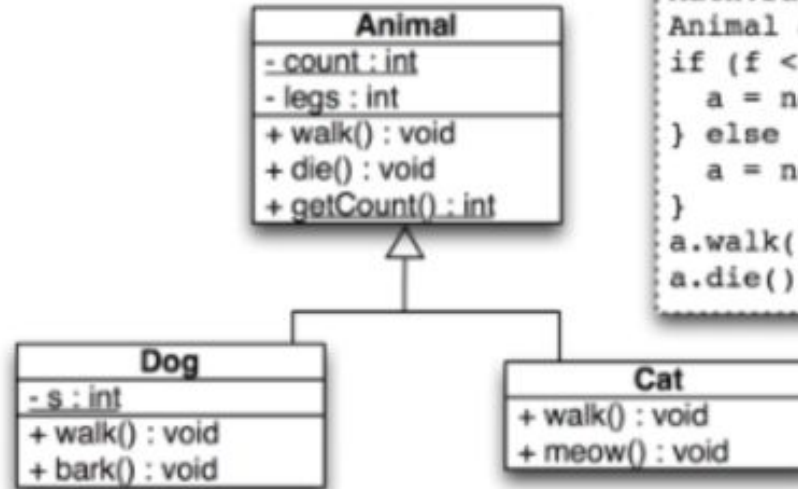
Aquí es donde entran los conceptos de clase y de prototipo, dependiendo de las herramientas que tengamos disponibles.



¿Cómo hacemos que un objeto herede de otro objeto?

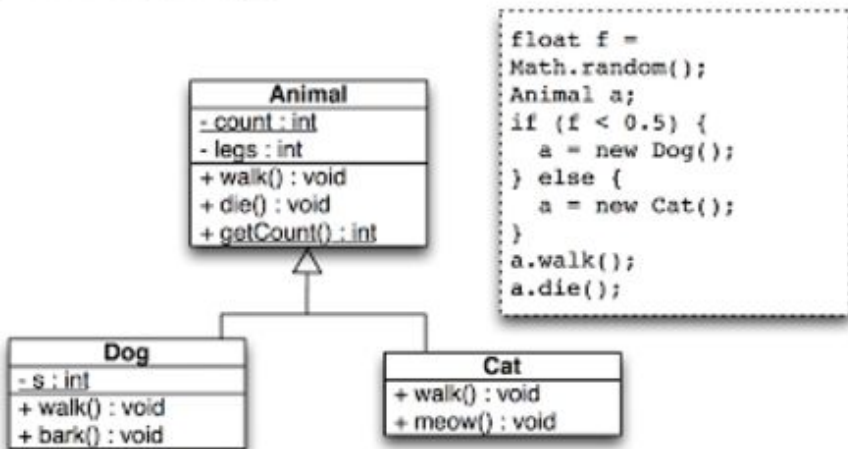
- Clases

**Late binding example**

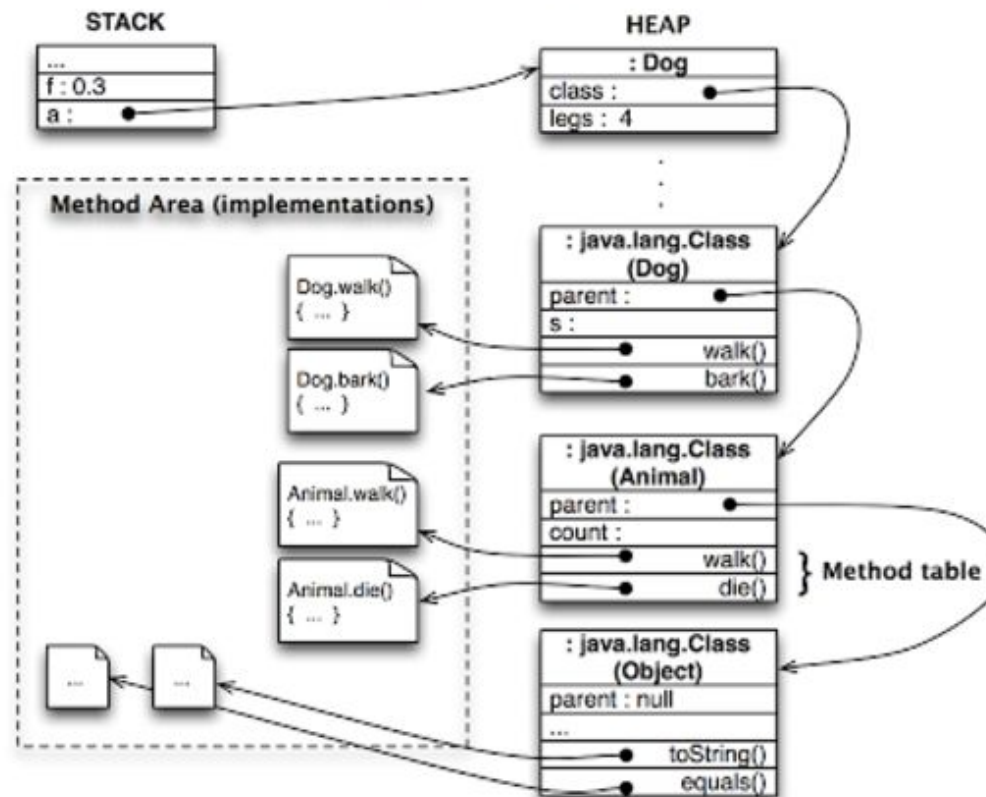


```
float f =
Math.random();
Animal a;
if (f < 0.5) {
    a = new Dog();
} else {
    a = new Cat();
}
a.walk();
a.die();
```

Late binding example



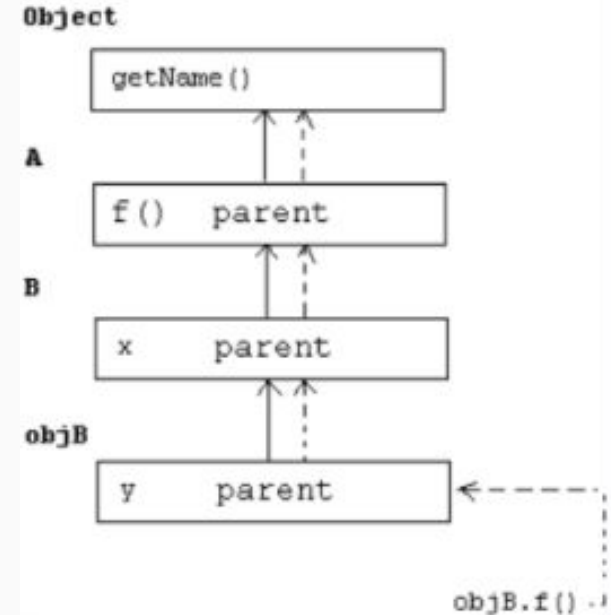
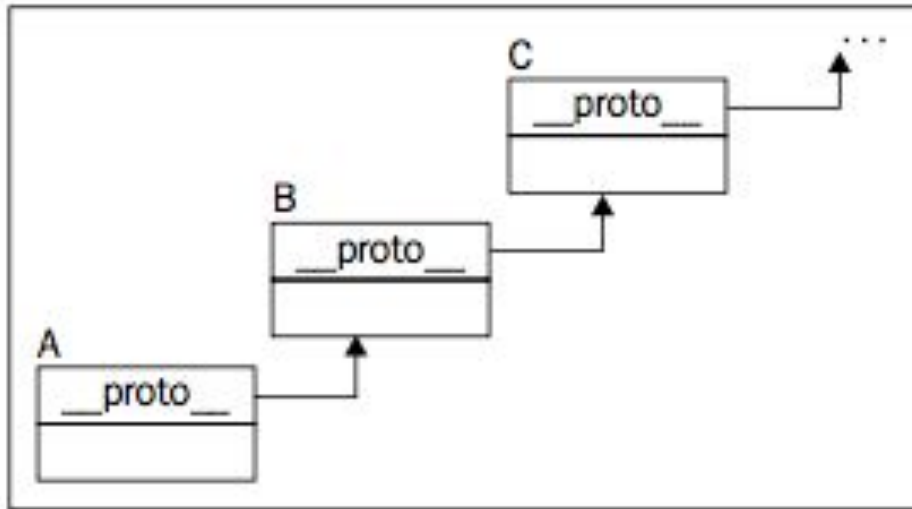
JVM Runtime memory layout



¿Cómo hacemos que un objeto herede de otro objeto?

- Prototipos.

Cada objeto creado tiene una referencia a su prototipo.  
El prototipo es otro objeto. Se genera una Prototype Chain



¿Cómo hacemos que un objeto herede de otro objeto?

Usando prototipos. Cada objeto creado tiene una referencia a su prototipo.

```
var shape = {};
```

¿Cómo hacemos que un objeto herede de otro objeto?

Usando prototipos. Cada objeto creado tiene una referencia a su prototipo.

```
var shape = {};
```

```
shape.constructor.prototype // Object{}
```

```
shape.constructor.prototype === Object.prototype; // true
```

```
Object.getPrototypeOf(shape) === Object.prototype; // true
```

```
function Shape() { }
```

```
Shape.prototype = {  
  name: "shape",  
  toString: function() {  
    return this.name;  
  }  
};
```

```
var instancia1 = new Shape();  
var instancia2 = new Shape();  
console.log(instancia1 == instancia2);  
Object.getPrototypeOf(instancia1) ==  
Object.getPrototypeOf(instancia2);
```

```
instancia1.name; // "shape"  
Shape.prototype.a = 1;  
instancia1.a; // 1
```

```
function Shape() { }
```

```
Shape.prototype = {  
  name: "shape",  
  toString: function() {  
    return this.name;  
  }  
};
```

```
instancia1.name; // "shape"  
Shape.prototype.a = 1;  
instancia1.a; // 1
```

```
var instancia1 = new Shape();  
var instancia2 = new Shape();  
console.log(instancia1 == instancia2); // false  
Object.getPrototypeOf(instancia1) ==  
Object.getPrototypeOf(instancia2); // true
```

```
function TwoDShape() {  
    this.name = "2D shape";  
};
```

```
function Triangle(side, height) {  
    this.name = "Triangle"; this.side = side; this.height = height;  
    this.getArea = function() { return this.side * this.height / 2; }  
};
```

```
TwoDShape.prototype = new Shape();  
Triangle.prototype = new TwoDShape();  
TwoDShape.prototype.constructor = TwoDShape;  
Triangle.prototype.constructor = Triangle;
```



```
var t = new Triangle(5,10);
```

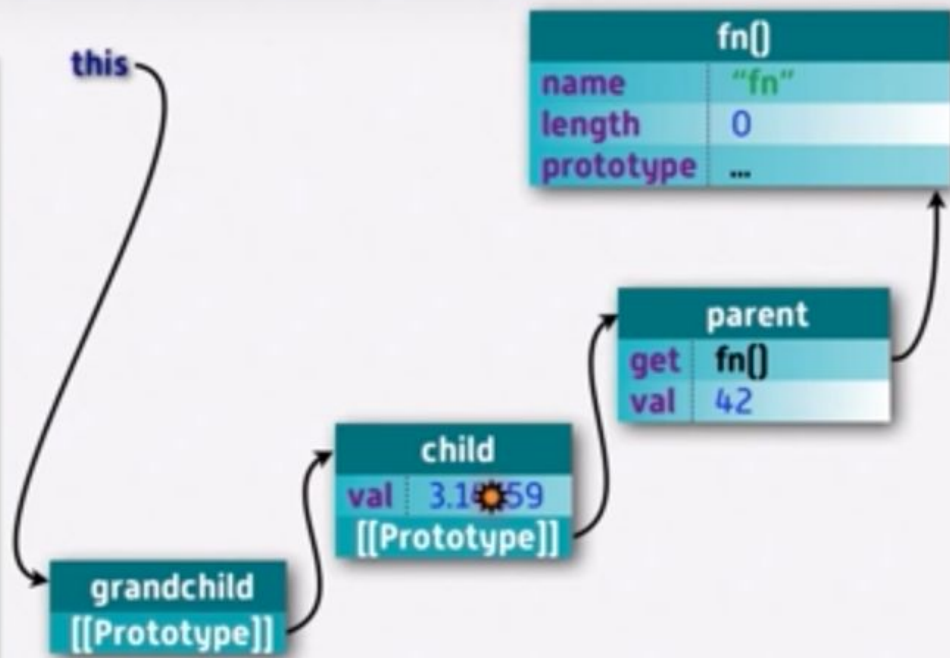
```
t.getArea(); // 25
```

```
t.toString(); // "Triangle"
```

```
Object.getPrototypeOf(t)
```

El método `Object.create()` crea un objeto nuevo, utilizando un objeto existente como el prototipo del nuevo objeto creado.

```
var parent = {  
  get: function fn() {  
    return this.val;  
  },  
  val: 42  
};  
  
var child = Object.create(parent);  
child.val = 3.14159;  
  
var grandchild = Object.create(child);  
  
parent.get();      // →42  
child.get();       // →3.14159  
grandchild.get();  // →3.14159
```



ECMAScript 2015 introduce sintaxis de CLASS en una manera de simplificar la sintaxis y hacerla más parecida a C++ o Java. Pero internamente el funcionamiento de los prototipos es el mismo.

```
class Shape {  
    constructor() {  
        this._name = "shape";  
    }  
    toString() { return this._name; };  
    get name() { return this._name; };  
    set name(newName) { this._name = newName; }  
}  
class TwoDShape extends Shape {  
    constructor() {  
        super();  
        this._name = "2D Shape";  
    }  
}  
  
class Triangle extends TwoDShape {  
    constructor() {  
        this._name = "Triangle";  
    }  
    toString() { return "I am a " +  
        this.name; };  
}  
  
var twoD = new TwoDShape();  
console.log(twoD.name);  
var triangle = new Triangle();  
console.log(triangle.toString());
```

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();
```

Leer este artículo para entender las distintas formas de herencia y cadena de prototipos:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance\\_and\\_the\\_prototype\\_chain#Different\\_ways\\_to\\_create\\_objects\\_and\\_the\\_resulting\\_prototype\\_chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#Different_ways_to_create_objects_and_the_resulting_prototype_chain)

Al igual que los lenguajes funcionales, las funciones se pueden tratar como valores de primera clase. Se puede hacer lo mismo que con otros tipos integrados.

- Se le pueden asignar un nombre
- Se pueden guardar en una variable
- Se pueden pasar por parámetro a otra función
- Se pueden retornar como respuesta de una función

```
var f = function(param) { return param; }  
f(f);  
f(f)(2);  
f( function() { return "123"; } )();
```

## ANONIMAS

```
function(){  
    alert("aaa");  
}
```

## SELF INVOKING FUNCTION

```
(  
    function(){  
        alert("aaa");  
    }  
)();
```

## CALLBACKS FUNCTIONS

```
function doSomethingAsync( inErrorCase, inSuccessCase ){  
    ...  
}  
doSomethingAsync( function(err){ alert(err);}, function(result){ ..... } );
```

## INNER (Private) FUNCTIONS

```
function a(param){  
  function b(theinput){  
    return theinput * 2;  
  }  
  return "result: " + b(param);  
}
```

b es una función privada, solo accesible desde a

## Literal Notation:

```
var a = function (param){  
  var b = function (theinput){  
    return theinput * 2;  
  }  
  return "result: " + b(param);  
}
```

## Beneficios de usar funciones privadas:

- El global space es menor, menos colisiones de nombres.
- Privacidad, se expone solo lo que se quiere exponer

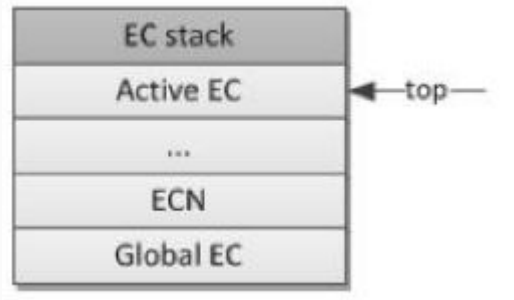


## Contexto de ejecución

Hay tres tipos de código con su contexto de ejecución asociado: global code, function code y eval code.

## Stack del contexto de ejecución

El modelo es un stack con el global context como ítem inicial, a medida que se generan nuevos se apilan en el stack. A medida que el de más arriba (el activo) deja el control al que lo llamó (el que le sigue en el stack) es sacado del stack.



## Estructura del Contexto de ejecución

Cada contexto es representado por un objeto simple, con propiedades que permiten el seguimiento del código asociado.

### thisValue:

1. Cuando un objeto es creado desde una función, this se refiere al objeto.
2. En el caso de una función, anidada o no, this se refiere al contexto global.

## Variable Object

Representación de los datos que definen un alcance.

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[ Variable object + all parent scopes ]
thisValue	Context object

Global VO	
foo	10
bar	<function>
<built-ins>	

## Scope Chain:

No hay scope de llaves, sino scope de funciones

## Lexical Scope:

El scope se crea cuando las funciones son definidas, no cuando son ejecutadas.

```
function f1(){  
  var a = 1;  
  return f2();  
}  
function f2(){  
  return a;  
}  
f1();  
// que retorna f1 ?
```

```
x = 0;  
function BigTest() {  
  this.x = 1;  
}  
BigTest();  
console.log(x); //¿qué imprime y por qué?
```

```
if (true){  
  var a = 1;  
}  
++a;  
// que retorna? es valido?
```

## Scope Chain:

No hay scope de llaves, sino scope de funciones

## Lexical Scope:

El scope se crea cuando las funciones son definidas, no cuando son ejecutadas.

```
function f1(){  
  var a = 1;  
  return f2();  
}  
function f2(){  
  return a;  
}  
f1();  
a is not defined
```

```
x = 0;  
function BigTest() {  
  this.x = 1;  
}  
var obj = new BigTest();  
console.log(x); ¿qué imprime y por qué?
```

```
if (true){  
  var a = 1;  
}  
++a;  
2
```

## Breaking the chain with closure:

### Closure #1:

```
function f(){  
    var b = "b";  
    return function(){  
        return b;  
    }  
}  
var n = f();  
n();  
b;
```

b solo existe dentro de f()  
y es accesible solo por f

### Closure #2:

```
var n;  
function f(){  
    var b = "b";  
    n = function(){  
        return b;  
    }  
}  
n();  
b;
```

### Closure #3:

```
function f(arg){  
    var n = function(){  
        return arg;  
    }  
    arg++;  
    return n;  
}  
var arg = 123;  
var m = f(arg);  
m(); //124  
m(); // ????  
arg = 125;  
m(); // ????
```

## Getter / Setter

```
var getValue, setValue;  
(function(){  
    var secret = 0;  
    getValue = function(){  
        return secret;  
    };  
    setValue = function(v){  
        secret = v;  
    };  
})();
```

```
function Foo(){  
    var privateData = {};  
    this.get = function(key){  
        return privateData[key];  
    }  
    this.set = function(key, value){  
        privateData[key] = value;  
        return this;  
    }  
}
```

Ejercicio:

Hacer una función que reciba un array de items, y retorne una función que la itere

Ejemplo:

```
var next = setupIterator(['a', 1, new Object(), 'b', [1,2,3]])
```

```
function setupIterator(x){  
    var i =0;  
    return function(){  
        return x[i++];  
    }  
}
```

```
var next = setupIterator(['a', 1, new Object(), 'b', [1,2,3]])
```

```
next(); // 'a'  
next(); // 1  
next(); // Object{}
```

Ejercicio:

Hacer una función que reciba un array de items, y retorne una función que la itere

Javascript es un lenguaje dinámico

Nos permite modificar las propiedades y métodos de objetos existentes en cualquier momento. (También agregar y quitar)

```
var hero = {}  
typeof hero.breed // undefined  
hero.breed = 'turtle';  
hero.name = 'Leonardo';  
hero.sayName = function(){ return this.name; }
```

Nos permite acceder como un mapa, esto es muy útil para trabajar dinámicamente con los objetos

```
hero["name"];  
hero["sayName"]  
  
hero.sayName();  
hero["sayName"]();  
delete hero.name;  
typeof hero.name // undefined
```



# Built In Objects

Podemos extender los objetos ya existentes en el lenguaje:

```
Array.prototype.inArray = function(needle) {  
  for (var i = 0, len = this.length; i < len; i++) {  
    if (this[i] === needle) {  
      return true;  
    }  
  }  
  return false;  
}
```

Ahora todos los arrays van a tener el nuevo método:

```
var a = ['red', 'green', 'blue'];  
a.inArray('red');  
true
```

```
  a.inArray('yellow');  
false
```

Cuidado con extender los prototipos de los Built In objects, que pasa si futuras versiones nativas incluyen ahora una función llamada inArray?

- <https://github.com/getify/You-Dont-Know-JS>
- Java Script Patterns, Stoyan Stefanov, O'Reilly, 2010
- <http://eloquentjavascript.net>
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- Pro JavaScript Design Patterns, Ross Harmes and Dustin Diaz, Apress, 2008 3
- JavaScript: The Good Parts, Douglas Crockford, O'Reilly, 2008.
- Object-Oriented JavaScript, Stoyan Stefanov, Packt Publishing, 2008.
- Test-Driven JavaScript Development, Christian Johansen, Addison Wesley, 2011
- Node, <https://nodejs.org/dist/latest-v9.x/docs/api/>
- Node, [http://eloquentjavascript.net/20\\_node.html](http://eloquentjavascript.net/20_node.html)
- ECMA Script 262 <https://www.ecma-international.org/ecma-262/9.0/index.html>