# PROTOTIPADO Y HERENCIA EN JAVASCRIPT

## CREACIÓN DE PSEUDOCLASES

```
function Animal (edad) {
  this.edad = edad || 0;
}

var iris = new Animal(4),
    rufo = new Animal(6);

console.log(iris instanceof Animal); // true
console.log(rufo.edad); // 6
```

Para crear una pseudoclase lo primero que debemos hacer es definir nuestro constructor, tarea que desempeñará una función que hará uso de la variable this como el objeto creado. Desde el momento en el que se defina la función podemos crear instancias de la clase para obtener nuevos objetos. El siguiente paso sería añadir métodos a nuestra clase que puedan ser utilizados por los objetos creados, y dado que en JavaScript todo es un objeto podríamos hacer algo así:

```
function Animal (edad) {
  this.edad = edad;
  this.crecer = function () {
    this.edad = this.edad + 1;
    return this.edad;
  }
}

var iris = new Animal(3);
console.log(iris.crecer()); // 4
```

Efectivamente este código funcionaría y nos da una forma de implementar métodos de instancia, pero surge un pequeño problema, y es que nuestro constructor está creando una función crecer para cada una de las instancias que creemos de la clase, por lo que si creamos un array con 1000 instancias de Animal se crearán 1000 funciones distintas (e independientes), algo que **en términos de eficiencia es tremendamente indeseable.**

Para esto, JavaScript nos provee de un pequeño pero práctico atributo que todos los objetos poseen: **prototype**. Este atributo a priori es un objeto vacío y simple pero que cuenta con unas características muy interesantes. Cuando una función se utiliza como constructor (que es lo que estábamos haciendo en los ejemplos anteriores) JavaScript establece como atributo prototype del objeto al mismo prototipo del constructor, de forma que los objetos creados pueden acceder a todos los atributos que se encuentren dentro de este objeto sin necesidad de duplicar información. Veamos cómo queda el código introduciendo este concepto:

```
function Animal (edad) {
  this.edad = edad;
}

Animal.prototype.crecer = function () {
  this.edad = this.edad + 1;
  return this.edad;
};

var iris = new Animal(3);
console.log(iris.crecer()); // 4
```

## HERENCIA

JavaScript implementa una herencia que nos permite asociar un objeto prototipo con una función constructora. De esta manera, el **nuevo objeto hereda todas las propiedades del objeto prototipo.**

Ahora que hemos visto como crear la clase animal, ¡vamos a hacer que los animales creados puedan hablar!. El problema es que no todos emiten el mismo sonido, así que deberíamos hacer clases distintas que hereden de la clase animal.

```
function Felino () {
}

Felino.prototype = new Animal();
//Otra forma de indicar el prototype
// Felino.prototype = Object.create(Animal.prototype);
Felino.prototype.constructor = Felino;

Felino.prototype.maullar = function () {
  console.log('meowwwww');
};

var iris = new Felino();
iris.crecer();//NaN
iris.maullar(); //'meowwwww'
```

¡Genial! Ahora iris puede maullar por ser un felino y crecer por ser un animal pero, ¿qué ha ocurrido aquí exactamente?

Cuando creamos nuestra clase Felino establecemos su prototipo como un nuevo objeto Animal de forma que el prototipo contendrá todos los métodos y atributos que tendría un animal corriente. Seguidamente sobrescribimos el constructor de Felino para que los objetos creados pertenezcan a esta clase y ya podemos añadir todos los métodos/atributos que queramos sin ningún problema.

*ACLARACIÓN*:

```
function Animal() {
// Expresiones
}
var felino = new Animal(); // Instanciando
// Herencia de Propiedades y Métodos a instanciar de Animal (Lo que ocurre internamente)
felino.__proto__ = Animal.prototype
```

- No es lo mismo __proto__ que prototype.
- Todos los objetos tienen __proto__ (Puntero de donde va heredad).
- Todas las funciones tienen una propiedad extra "prototype".
- Las instancias "mediante __proto__" apuntan/heredan de "Animal.prototype".

Si revisamos el código anterior para Iris.crecer() funcione correctamente y no arroje NaN, tenemos que pasarle los mismos parámetros que tiene la clase padre:

```
function Felino (edad) {
// Invoco constructor padre en constructor hijo (Utilizable para herencia Múltiple)
      Animal.call(this,edad);
}
//Luego podrás usarlo en la instancia para luego utilizar método heredado de Animal, algo así:
var iris = new Felino(4);
iris.crecer(); //5
iris.maullar(); //'meowwwww'
```

**OTRO EJEMPLO**

```
// Función constructora de Persona
function Persona(nombre){
 this.nombre=nombre;
}

// métodos
Persona.prototype.getNombre = function () {return this.nombre;};
Persona.prototype.respirar = function () {return 'respirando...';};

// Función constructora de Trabajador.
// Podríamos decir 'trabajador es una persona'

function Trabajador(oficioArg, empresaArg, nombreArg){

 Persona.call(this,nombreArg); //Llamamos al constructor de Persona
                               //y asignamos la misma propiedad
 this.oficio=oficioArg;
 this.empresa=empresaArg;
}
```

```
// Extendemos las propiedades y métodos de Persona hacia Trabajador,
// esto es, realizamos la herencia.
 // Y asignamos el constructor a la subclase.

Trabajador.prototype = new Persona();
Trabajador.prototype.constructor=Trabajador;

// Y ahora añadimos, y solamente después de realizar la herencia,
// un método a Trabajador para que nos devuelva en qué empresa trabaja

Trabajador.prototype.getEmpresa = function() {return this.empresa;}
```

Resultando:

```
 var p=new Persona('José Antonio');
 var t=new Trabajador('Programador','S.A.','José Antonio');

 alert(p.getNombre()); //José Antonio
 alert(t.getNombre()); //José Antonio
 alert(t.getEmpresa()); //S.A.
 alert(t.respirar()); //respirando... (método de la superclase Persona)
 alert(t instanceof Persona); //true
 alert(t instanceof Trabajador); //true
```

Viendo lo anterior, podemos hacer una función que nos facilite la extensión:

```
 function extiende(subClase, superClase){
  subClase.prototype = new superClase;
  subClase.prototype.constructor=subClase;
 }
```

Aplicándolo al ejemplo anterior:

```
 function Persona(nombre){
  this.nombre=nombre;
}

// métodos
Persona.prototype.getNombre = function () {return this.nombre;};
Persona.prototype.respirar = function () {return 'respirando...';};

function Trabajador(oficioArg, empresaArg, nombreArg) {
   Persona.call(this,nombreArg);
   this.oficio=oficioArg;
   this.empresa=empresaArg;
  }

extiende(Trabajador,Persona);

Trabajador.prototype.getEmpresa = function() {return this.empresa;}
```

## DETERMINANDO LA RELACIÓN ENTRE INSTANCIAS

La búsqueda de propiedades en JavaScript comienza en las propias propiedades del objeto, y si este nombre de propiedad no se encuentra, consulta las propiedades del objeto especial **__proto__**. Este proceso se realiza de manera recursiva.

La propiedad especial __proto__ se define cuando un objeto es construido: su valor corresponde con la propiedad prototype del constructor. Así, la expresión new Foo() crea un objeto con la propiedad __proto__ == Foo.prototype. En consecuencia, los cambios producidos en Foo.prototype alteran la búsqueda de propiedades de todos los objetos creados con new Foo().

Todo objeto tiene una propiedad __proto__, así como una propiedad prototype. Por lo tanto, los objetos pueden relacionarse a través de esta propiedad. Un ejemplo:

```
var t = new Trabajador('Programador','S.A.','Jose Antonio');
```

En este objeto, todas las siguientes sentencias se cumplen:

```
alert(t.__proto__ == Trabajador.prototype);
alert(t.__proto__.__proto__ == Persona.prototype);
alert(t.__proto__.__proto__.__proto__ == Object.prototype);
alert(t.__proto__.__proto__.__proto__.__proto__ == null);
```

De esta manera, podemos construir una función para saber si una variable es una instancia de un objeto.

```
function instanceOf(object, constructor) {
   while (object != null) {
      if (object == constructor.prototype)
         return true;
      if (typeof object == 'xml') {
        return constructor.prototype == XML.prototype;
      }
      object = object.__proto__;
   }
   return false;
}

instanceOf (t, Trabajador)      // true
instanceOf (t, Persona)     // true
instanceOf (t, Object)      // true
```

## HERENCIA BASADA EN CLASES VS PROTOTIPOS

| Basado en clases (Java) | Basado en prototipos (JavaScript) |
| --- | --- |
| Clase e instancia son dos entidades diferentes | Todos los objetos son instancias |
| Las clases se definen de manera explícita, y se instancias a través de su método constructor. | Las clases se definen y crean con las funciones constructoras. |
| Un objeto se instancia con el operador `new`. | Un objeto se instancia con el operador `new`. |
| La estructura de clases se crea utilizando la definición de clases. | La estructura de clases se crea asignando un objeto como prototipo. |
| La herencia de propiedades se realiza a través de la cadena de clases. | La herencia de propiedades se realiza a través de la cadena de prototipos. |
| La definición de clases especifica todas las propiedades de una instancia de una clase. No se pueden añadir propiedades en tiempo de ejecución. | La función constructora o el prototipo especifican unas propiedades iniciales. Se pueden añadir o eliminar estas propiedades en tiempo de ejecución, en un objeto concreto o a un conjunto de objetos. |