# Informe de Lenguajes de Programación: Seminario # 13

Laura Brito Guerrero
Sheyla Cruz Castro
Ariel Antonio Huerta Martín
Julio Daniel Gambe Alcorta
Pablo Antonio de Armas
Grupo 2

Universidad de La Habana
Facultad de Matemática y Computación

# 1. Modelo de objetos de Javascript

*Javascript* es un lenguaje de *scripting*, un lenguaje de programación típicamente interpretado. Está diseñado sobre un paradigma simple basado en objetos. Es un lenguaje *multiparadigma* que permite enfocar la solución de los problemas mediante - y entre otras opciones - la programación orientada a *Prototipos*.

### Propiedades

Un objeto de *Javascript* tiene propiedades asociadas a él. Una propiedad de un objeto puede ser explicada como una variable adjunta al objeto. Las propiedades de un objeto son básicamente lo mismo que las variables comunes de *Javascript*, excepto que su tiempo de vida está ligado al del objeto. Un objeto en *Javascript* es un *diccionario*, un mapa entre llaves y valores. Las propiedades de un objeto definen sus las características y se puede acceder a ellas con la notación:

*nombreObjeto.nombrePropiedad*

A continuación se mostrará un ejemplo en el que se asignan propiedades a un objeto:

```
var myCar = new Object();
myCar.mark = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;
```

Las propiedades no asignadas de un objeto son *undefined* (no *null*). Basándose en el ejemplo anterior:

```
myCar.color;  // undefined
```

También pueden ser accedidas mediante *corchetes* ([ ]), de ahí su relación con los *diccionarios*. Por ejemplo:

```
myCar["mark"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

El nombre de la propiedad de un objeto puede ser cualquier cadena válida de *Javascript*, o cualquier cosa que se pueda convertir en una cadena, incluyendo una cadena vacía. Sin embargo, cualquier nombre de propiedad que no sea un identificador válido de *Javascript*(puede ser el nombre de alguna propiedad que tenga un espacio o un guión, o que comience con un número) solo puede ser accedido utilizando la notación de *corchetes*. Ejemplo de ello es:

```
var myObject = new Object(),
  myChain = "myChain",
```

```
    myRandom = Math.random(),
    obj = new Object();
  myObject.type = "Syntax at point";
  myObject["Date_ Creation"] = "Chain at spaces and accent";
  myObject[myChain] = "String_ Value";
  myObject[myRandom] = "Random number_";
  myObject[obj] = "Object_";
  myObject[""] = "Within empty chain";

  console.log(myObject);
```

Todas las llaves con notación en *corchetes* son convertidas a *string* a menos que sean *Symbols*.
También se pueden acceder mediante el uso de un valor de cadena que se almacena en una variable:

```
var nameProperty = "mark";
myCar[nameProperty] = "Ford";
```

Las propiedades de un objeto y sus valores pueden ser creadas, cambiadas o eliminadas en tiempo de ejecución y la mayoría de ellas pueden ser enumeradas por medio de la sentencia cíclica: *for...in*. La siguiente función muestra las propiedades del objeto cuando se pasa como argumentos de la función el *objeto* y el *nombre del objeto*:

```
function lookProperties(obj, nameObj){
  var result = '';
  for(var i in obj){
    if(obj.hasOwnProperty(i)){
      result += '${nameObj}.${i} = ${obj[i]}\n';
    }
  }
  return result;
}
```

La sentencia *obj.hasOwnProperty* se usa para filtrar las propiedades de *obj*. Por lo tanto, la llamada a la función *lookProperties(myCar, "myCar")* retornaría:

```
console.log(lookProperties(myCar, "myCar"));
>myCar.mark = Ford
>myCar.model = Mustang
>myCar.year = 1969
```

Otras maneras de listar las propiedades de un objeto son:
→ *Object.keys(o)* : Devuelve un array con todos los nombres de propiedades ("keys") enumerables y propias (no de *la cadena de prototipos*) de un objeto *o*.
→ *Object.getOwnPropertyNames(o)*: Devuelve un array que contiene todos los nombres(enumerables o no) de las propiedades de un objeto *o*.

## Creando nuevos objetos

*Javascript* tiene un conjunto de objetos predefinidos. En *Javascript 1.2* y versiones posteriores, se puede crear un objeto usando una *función constructora* al invocar dicha función con el operador *new* como se muestra a continuación:

```
var arr = new Array();
```

Por ejemplo, se desea crear un tipo de objeto para modelar automóviles: *Car*, es decir, el *prototipo*(tema que hablaremos más adelante) de los objetos creados con la función definida a continuación es *Car*, por tanto, todos los objetos creados a partir de esta función tendrán como propiedades las que se definan en dicha función:

```
function Car(mark, model, year){
  this.mark = mark;
  this.model = model;
  this.year = year;
}
```

Analicemos que ocurre en la función anterior. Las funciones que van a ser utilizadas como constructores tienen una semántica especial asociada a ellas. Al llamar a la función se crea un objeto en el intérprete con una referencia asociado a él dentro de la función: *this*. Al aadirle propiedades a *this*, se le aaden propiedades al objeto "devuelto"por el operador *new*.

La creación de una instancia de *Car* sería:

```
var myCar = new Car("Eagle", "Talon TSi", 1993);
```

Entonces el valor de *myCar.mark* es el string *Eagle*, y así sucesivamente.

Además de la creación de objetos mediante una *función constructora*, se pueden crear utilizando un *inicializador de objeto*. El uso de los *inicializadores de objeto* se refiere a veces a cómo crear objetos con la notación literal. *Inicializador de objeto* es consistente con la terminología utilizada por C++. La sintaxis para un objeto usando un inicializador es:

```
var obj = {
  prop_1       : value_1, // prop_# can by one id
  2            : value_2, // or a number}
  // ...
  "property n": value_n // or a chain
};
```

donde *obj* es el nombre del objeto, $prop_i$ es un identificador, y cada $value_i$ es una expresión cuyo valor es asignado a la $prop_i$. Se tiene que *obj* y la asignación es opcional, si no se necesita hacer referencia a este objeto desde otro lugar, no es necesario asignarlo a una variable.
Los *inicializadores de objeto* son expresiones, y cada uno da como resultado un nuevo objeto donde la instrucción de creación sea ejecutada. Si se tienen *inicializadores de objetos* iénticos se crean objetos distintos que no se compararán entre sí como iguales.
Los objetos se crean con la llamada *new Object()*; esto es, los objetos hechos de expresiones literales

de objetos son instancias de *Object*.
El siguiente ejemplo crea *myMotorcycle* con tres propiedades.

```
var myMotorcycle = {color : "red", ring : 4, prop: {cylinder : 4,
                      size: 2.2}};
```

En *Javascript 1.1* y versiones anteriores, no se puede utilizar *inicializadores de objeto*. Se pueden crear objetos usando solo funciones constructoras o utilizando una función suministrada por algún otro objeto para ese propósito.

Los objetos también se pueden crear mediante el método *Object.create*. Este método puede ser muy útil, ya que te permite elegir el *prototipo* del objeto que deseas crear, sin tener que definir una función constructora.

```
//Propiedades y metodo de encapsulacion para Animal
var Animal = {
  tipo : 'Invertebrados', // valor por defecto de la propiedad
  mostrarTipo: function(){
      // mostrara el tipo de animal
      console.log(this.tipo);
    }
};

// Crear un nuevo objeto de tipo Animal llamado animal1
var animal1 = Object.create(Animal);
animal1.mostrarTipo();  // Output : Invertebrados

var pez = Object.create(Animal);
pez.tipo = 'Pescados';
pez.mostrarTipo(); // Output: Pescados
```

De la siguiente manera se puede modificar una instancia de *Car* para que sus propiedades sean *totalmente privadas*:

```
const Car = (function() {
   const carProps = new WeakMap();
   class Car {
    constructor(make, model) {
      this.make = make;
      this.model = model;
      this._userGears = ['P', 'N', 'R', 'D'];
      this._userGear = this._userGears[0];      }
        get userGear() { return this._userGear; }
    set userGear(value) {
      if(this._userGears.indexOf(value) < 0)
```

```
        throw new Error(`Invalid gear: ${value}`);
      this._userGear = vaule;      }
        shift(gear) { this.userGear = gear; }
  }
   return Car; })();
```

*Javascript* no incorpora de serie ningún mecanismo de visibilidad para los miembros de un objeto. La manera de encapsular es con clausuras. Todo es público por defecto. Pero es posible lograrlo si hacemos lo anterior. Básicamente la regla es:
→ usar la técnica de función constructora,
→ los métodos / variables privados son funciones o variables declaradas dentro de la función constructora,
→ los métodos / variables públicas se asignan a *this*,
→ guardar el valor de *this* en una variable privada(usualmente se usa *that* o *self*),
→ dadas las funciones privadas debes usar *self* para acceder a las variables públicas,
→ desde las funciones públicas puedes usar *self* para acceder a las variables públicas,
→ en ambos casos puedes acceder a las variables privadas directamente con su nombre.

En Javascript *las clases son funciones*. En *ES5*, se puede inicializar un *Car* de esta forma:

```
function Car(make, model) {
  this.make = make;
  this.model = model;
  this._userGears = ['P', 'N', 'R', 'D'];
  this._userGear = this.userGears[0];
}
```

Si se realiza en *ES6* las siguientes líneas:

```
class Es6Car {}
function Es5Car {}
> typeof Es6Car          // "function"
> typeof Es5Car          // "function"
```

Se observa que es exactamente lo mismo.

## Prototipo

Un *prototipo* es un objeto especial que describe su comportamiento y que sirve como base para crear unos nuevos. Son un mecanismo mediante el cual los objetos en *Javascript* heredan características entre sí. *Javascript* es un lenguaje basado en *prototipos*. Para proporcionar mecanismos de herencia los objetos pueden tener un *prototipo* (objeto *prototipo*) asociado, que actúa como una plantilla desde la que el objeto puede heredar métodos y propiedades. Un objeto *prototipo* puede tener, a su vez, otro objeto *prototipo* asociado. Esto es conocido como la *cadena de prototipos*, y es la razón por la que los objetos pueden tener métodos y propiedades disponibles que no han sido declarados por ellos mismos.
Para ser exactos, los métodos y propiedades son definidas en la propiedad *prototype*, que reside en la

función constructor del objeto, no en su instancia. *Javascript* establece un enlace entre la instancia del objeto y su *prototipo*, donde el objeto tendrá acceso a una serie de métodos y propiedades que se encuentran a lo largo de la *cadena de prototipos* asociada.
Desde *ECMAScript 2015* se puede acceder indirectamente al objeto *prototipo* de un objeto mediante *Object.getPrototypeOf(obj)*.
Las propiedades y métodos heredados se definen en la propiedad *prototipo*, es decir, los que empiezan con *Object.prototype*, y no los que empiezan solo con *Object*. El valor de la propiedad del *prototipo* es un objeto, que es básicamente un repositorio (bucket) para almacenar propiedades y métodos deseadas que sean heredados por los objetos más abajo en la cadena del *prototipo*. Así que *Object.prototype.watch()*, *Object.prototype.valueOf()*, etc, están disponibles para cualquier tipo de objeto que herede de *Object.prototype*, incluyendo nuevas instancias de objeto creadas desde el constructor.
*Object.is()*, *Object.keys()*, y otros miembros no definidos dentro del *prototipo* del repositorio (bucket) no son heredados por instancias de objeto o tipos de objetos que heredan de *Object.prototype*, sino que son métodos / propiedades disponibles solo en el propio constructor *Object*.
La propiedad *prototipo* es una de las partes más confusamente nombradas en *Javascript*, podría pensarse que apunta al objeto *prototipo* del objeto actual, pero no lo hace: es un objeto interno al que puede accederse mediante:

```
__proto__
```

(si el código se ejecuta desde Chrome, ya que el nombre de la propiedad depende de la implementación de *Javascript* que se utilice, en donde se ejecute el código(Chrome, Firefox, NodeJS)). En su lugar, el *prototipo* es una propiedad que contiene un objeto en el que se definen los miembros en el que se desea que se herede.

Si se quiere acceder a una propiedad o un método de un objeto, y este no existe, *Javascript* chequea si existe en el prototipo del objeto. Todas las instancias de una función comparten el mismo prototipo, si existe una propiedad en el prototipo, todas las instancias de la clase tienen acceso a esa propiedad.
Notar que definiendo un método o una propiedad en una instancia se puede sobreescribir la versión en el prototipo, recordar que *Javascript* primeramente chequea la instancia antes de chequear el prototipo. Observando el siguiente ejemplo:

```
// class Car as defined previously, with shift method
const car1 = new Car();
const car2 = new Car();
car1.shift === Car.prototype.shift;     // true
car1.shift('D');
car1.shift('d');                        // error
car1.userGear;                          // 'D'
car1.shift === car2.shift               // true
car1.shift = function(gear) { this.userGear = gear.toUpperCase(); }
car1.shift === Car.prototype.shift;     // false
car1.shift === car2.shift;              // false
car1.shift('d');
```

```
car1.userGear;                    // 'D'
```

Inicialmente, el objeto *car1* no tiene un método *shift*, pero cuando se llama *car1.shift('D')*, *Javascript* observa el prototipo para *car1* y busca un método con ese nombre. Cuando se reemplaza *shift*, *car1* y su prototipo tienen un método con ese nombre. Cuando se invoca *car1.shift('d')*, se realiza en el método de *car1* y no en su prototipo.

## Inheritance

*Javascript* es un poco confuso ya que es dinámico y no proporciona una implementación de *class* (aunque la palabra clave *class* se introduce en *ES2015*, pero es un azúcar sintáctico, *Javascript* sigue siendo basado en prototipos).
Cuando se trata de herencia, *Javascript* solo tiene una construcción: *objetos*. Cada objeto tiene una propiedad privada que contiene un enlace a otro objeto llamado *prototipo*. Ese objeto *prototipo* tiene un *prototipo* propio, y así sucesivamente hasta que se alcanza un objeto *null* como prototipo. Por definición *null* no tiene prototipo y actúa como el eslabón final en esta *cadena de prototipos*.
Al intentar acceder a una propiedad de un objeto, la propiedad no solo se buscará en el objeto sino también en el *prototipo* del objeto, el *prototipo* del *prototipo*, y así sucesivamente hasta que se encuentre una propiedad con un nombre coincidente o el final de la *cadena prototipo* se alcanza. Las propiedades heredadas se pueden encontrar en el objeto *prototype* del constructor.
*ECMAScript2015* introduce un nuevo set de palabras reservadas que implementan clases. Aunque estos constructores lucen más familiares para los desarrolladores de lenguajes basados en clases, aún así no son clases. *Javascript* sigue estando basado en prototipos. Los nuevos *keywords* incluyen *class*, *static*, *extends* y *super*.
Cuando se crea una instancia de una función, su *inheritance* funcionalmente es un prototipo de la función. A continuación se verá un ejemplo, donde se observa que *Car* es un tipo de *Vehicle*:

```
class Vehicle {
  constructor() {
    this.passengers = [];
    console.log("Vehicle created");   }
  addPassenger(p) {      this.passengers.push(p);    }
}
class Car extends Vehicle {
  constructor() {
    super();
    console.log("Car created");   }
  deployAirbags() {      console.log("BWOOSH!");    }
}
```

Esta sintaxis marca que *Car* es una subfunción de *Vehicle*. El llamado a *super()*, es una función especial en Javascript que invoca al constructor de la *superclase*. Esta es requerida en las subclases y se genera un error cuando se omite.
En el siguiente ejemplo:

```
const v = new Vehicle();
v.addPassenger("Frank");
v.addPassenger("Judy");
v.passengers;                    // ["Frank", "Judy"]
const c = new Car();
c.addPassenger("Alice");
c.addPassenger("Cameron");
c.passengers;                    // ["Alice", "Cameron"]
v.deployAirbags();               // error
c.deployAirbags();               // "BWOOSH!"
```

Notar que se puede llamar *deployAirbags* en *c*, pero no en *v*. En otras palabras, la herencia se emplea solamente de una manera. Las instancias de la clase *Car* pueden acceder a todos los métodos de la clase *Vehicle*, pero no en el otro sentido.

*Javascript* no tiene métodos en la forma que los lenguajes basados en clases los define. En *Javascript*, cualquier función pude añadirse a un objeto como una propiedad. Una función heredada se comporta como cualquier otra propiedad, viéndose afectada por el solapamiento de propiedades. Cuando una función heredada se ejecuta, el valor *this* apunta al objeto que hereda, no al prototipo en el que la función es una propiedad. Por ejemplo:

```
var o = {
  a : 2,
  m : function(b){
    return this.a + 1;
  }
};
console.log(o.m()); // 3
// cuando en este caso se llama a o.m, <this> se refiere a o

var p = Object.create(o);
//p es un objeto que hereda de o

p.a = 12; //crea una propiedad <a> en <p>
console.log(p.m()) // 13
//Cuando se llama a p.m(), <this> se refiere a p,
//De esta manera, cuando p hereda la funci\'on <m> de <o>,
//<this.a> significa <p.a>
```

### Duck Typing:

Este lenguaje tiene una característica conocida como *Duck Typing*, el cual significa: *Si camina como pato, grazna como pato y se comporta como pato, entonces es un pato*. Se suele asociar a los lenguajes dinámicos.
Como ejemplo se tiene:

```
class Ave(){}

function func(ave){
  ave.vuela();
}
paloma = new Ave();
func(paloma);

func({vuela: () => 1});
```

En *Javascript*, todos los objetos que tengan como propiedad *vuela*, incluso si no extienden la clase *Ave*, no darán excepción en el código anterior. Ambos llamados a la función *func* funcionan igual, porque los objetos tienen *forma de ave*, que solo significa tener una propiedad *vuela*.

## Mixins

Un *mixin* es una clase que contiene métodos que pueden ser utilizados por otras clases sin necesidad de heredar de ella. Proporciona métodos que implementan un determinado comportamiento, se utiliza para agregarle el comportamiento a otras clases(funciones). Ejemplo de ello se tiene:

```
a = new Array();
f = (prototype) => {
    prototype.functionality = () => {
        console.log('some fancy stuff');
    }
};

f2 = function(prototype) {
    prototype.functionality2 = function(){
        console.log('some more cheesy stuff');
    };
};

f(a);
a.functionality();
f2(a);
a.functionality2();
```

Aquí se puede observar que luego que *a* sea tomado de parámetro en las funciones *f* y *f2*, adopta dos comportamientos distintos: *funcionality* y *funcionality2*, lo cual puede dar una noción de *herencia múltiple*, ya que cumple con comportamientos de funciones distintas.

# 2. Implementación en python de la clase *JavascriptObject*

**Conceptos Importantes:**

**Decorador:** Son funciones (ó clase) que reciben como parámetro otra función (ó clase) y permiten que estas sean llamadas como propiedades de la clase en lugar de métodos.

La manera clásica de crear un objetos es:

- Definir la estructura del objeto usando una declaración de clase.
- Instanciar esa clase para crear un nuevo objeto.

En la herencia por prototipos, simplemente creamos el objeto, el cual luego es reusado por nuevos objetos gracias a la manera en que funciona la búsqueda por la cadena de prototipos.

Este objeto suele ser llamado prototype object (objeto prototipo). De ahí sale el nombre de herencia por prototipos.

Recreación del modelo Person/Author usando herencia por prototipos y la función clone():

```
// Clone function
function clone(object){
  function F(){}
  F.prototype = object;
  return new F;
}
// Person Prototype Object
  var Person = {
    name: 'default name',
    getName: function(){
    return this.name;
  }
}
var reader = clone(Person);
console.log(reader.getName()); // Print: "default name"
reader.name = 'Hans Christian Andersen';
console.log(reader.getName()); // Print: "Hans Christian Andersen"
```

Para crear Author, no hacemos una subclase de persona, sino un clon en el que pueden ser añadidos nuevos miembros o redefinir los existentes.

```
  // Author Prototype Object
  var Author = clone(Person);
  Author.books = [ ]; //valor x defecto
```

```
Author.getBooks = function(){
  return this.books;
}
```

Se ha creado un nuevo prototype object, el cual puede ser clonado ahora para crear nuevos objetos del tipo Author

```
author = clone(Author);
author.name = "Agatha Christie"
author.books = ["Asesinato en el Orient Express"];
author.getName();
author.getBooks();
```

### Herencia Clásica Mediante Cadena de Prototipos:

Definamos y creemos un objeto de tipo Person.

```
//Class Person
function Person(name){
  this.name =  name;
}
Person.prototype.getName = function(){
  return this.name;
}
var reader = new Person("James Bond");
reader.getName();
```

Definamos ahora un objeto de tipo Author que herede los miembros de Person.

```
//Class Author
function Author(name, books){
  Person.call(this, name);  // Llama al constructor de persona
  this.books = books;  //Atributo adicionado
}
Author.prototype = new Person(); //Para setear la cadena de prototipos
Author.prototype.constructor = Author; //Para setear Author
Author.prototype.getBooks = function(){ //Metodo adicionado
  return this.books;
}
var author = new Author("Agatha Christie", ["Asesinato en el Orient Exp
author.getName();
author.getBooks();
```

### Herencia Clásica Mediante Cadena de Prototipos (Función extend):

```
      //Extend function
      function extend(subClass, superClass){
        var F = function(){ };
        F.prototype = superClass.prototype;
        subClass.prototype = new F();
        subClass.prototype.constructor = subClass;
      }
      //Class Person
      function Person(name) {
        this.name = name;
      }
      Person.prototype.getName = function() {
        return this.name;
      }
      //Class Author
      function Author(name, books) {
        Person.call(this, name);
        this.books = books;
      }
      extend(Author, Person);
      Author.prototype.getBooks = function() {
        return this.books;
      }
```

Añadir la superClase como un atributo de la subClase permite llamar métodos directamente de la superClase, lo cual resulta útil cuando queremos redefinir un método de la superClase y a la vez seguir conservando acceso a la implementación de dicho método en la superClase.

```
      Author.prototype.getName = function(){
        var name = Author.superClass.getName.call(this);
        return name + ', author of ' + this.getBooks().join(', ');
      };
```

**Implementación:**

```
class JavaScriptObject(object):
    prototype = None
    def __init__(self):
        self.__proto__ = JavaScriptObject.prototype
    def __getattr__(self, item):
        try:
            return getattr(self.__proto__,item)
        except:
            print("undefined")
            return
```

```
    """
      Boolean que indica si el objeto tiene property
      como miembro propio de la instancia
    """

    def Property(self, property):
        return property in self.__dict__

    """
    Permite el comportamiento de similitud entre
    obj["attr"] = value y obj.attr = value
    """
    def __setitem__(self, key, value):
        setattr(self, key, value)

    def __getitem__(self, item):
        return getattr(self,item)


    """
    Funci n create de Object en Javascript que permite que el
    nuevo objeto tenga como prototipo el par metro pasado
    """
    @staticmethod
    def create(object):
        obj = JavaScriptObject()
        obj.__proto__ = object
        return obj

    def PrototypeOf(self,obj):
        if obj.__proto__ == None:
            return False
        return obj.__proto__ == self or self.PrototypeOf(obj.__proto__)

    def JavaScriptContructObject(function):
        cls = type(function.__name__, (JavaScriptObject,), {})
        cls.prototype = JavaScriptObject()
        def init(*args):
            args[0].__proto__ = cls.prototype
            function(*args)
        cls.__init__ = init
        return cls

    @JavaScriptContructObject
    def Person(self, name):
        self.name = name
```

```
    @JavaScriptContructObject
    def Author(self,edad):
        self.edad = edad
```

## 3. Javascript y sus características que lo hacen *orientado a objetos*

Entonces, dado que en *Javascript* existen las características anteriores que se pueden observar en los lenguajes orientados a objetos, pudiéramos decir que es orientado a objetos.