# JavaScript Object Notation

Instance 🡪 declaration of the specific class of the object. Constructor is a function with the same name as the class that is building or represents.

1. Abstraction principle: declaration of objects and properties without values.
2. Encapsulation: the objects have their own methods and properties by default.
3. Inheritance: the common properties of the derived classes should be defined in the super-class
4. Polymorphism: Is a consequence of the inheritance. The same method works different depending on the object (overload). The Extension or subtyping (polimorfismo de inclusión o de redefinición).

Objects have two kinds of properties: **attributes** and **methods**

## Objects creation with a function constructor

***var* perro = function()** // los constructores no funcionan con **=>**  
**{** //función constructor de nombre perro ***this*.tienePelo = true;** // atributo del objeto perro  
 ***this*.ladrar = *function*() {** //método del objeto perro ***console*.log("guau-guau");** //el método lleva incorporado un console.log **}**

***var* edad = 7;  
};** //este punto y coma ; es opcional

keyword ***this*** indicates that reference is being made to attribuetes and methods of the object itself. Properties with the prefix ***this*** are usable from the outside. Variables declared with the keyword ***var*** inside of the local scope of the constructor, only will be accessible from the constructor.

***var* pastorAleman = new perro();**

***console*.log("el perro hace: ");  
pastorAleman.ladrar();** //¡No se puede llamar al método dentro del console.log!

***console*.log(pastorAleman.edad);** //Undefined

## Constructor with arguments call

***var* dog = *function*(raza)** // atributo del objeto perro  
**{ *this*.raza = raza; };** //también funciona sin el punto y coma ;

***var* miPerro = new dog("bulldog");  
*console*.log("tengo un perro de raza " + miPerro.raza);**//En este caso sí que se puede llamar al atributo dentro del console

When declaring an object, its own properties must be defined with keyword ***this***. To add new properties in the later, use the following sintax: **objectName.prototype.property**

## Prototype

***var* perro = *function*() {};** //codificamos un constructor vacío  
***console*.log(*perro*.prototype);** //cualquier constructor tiene un prototype

**{…} object { ladrar: perro.prototype.ladrar(), … }  
constructor: function perro() \_\_proto\_\_: Object { … }**

***perro*.prototype.ladrar = *function*()** //adding a method to the constructor´s proto **{ *console*.log("guau-guau"); }** //método del objeto perro

//Instanciación de objetos a través del constructor perro ***var* pastorAleman = new perro(); *var* dogo = new perro();**

***console*.log ("el pastor alemán hace: "); pastorAleman.ladrar();  
*console*.log ("el dogo hace: "); dogo.ladrar();**

## Method overload

***var* dog = *function*() {};** //declarar la función constructor vacío de nombre dog

***dog*.prototype.colorDePelo = *function*()**//adding a method to the constructor´s proto **{ *document*.write("marrón.<br/>"); }**

***var* boxer = new dog(); *var* labrador = new dog();**

***document*.write("el boxer sin sobrecargar tiene el pelo de color ");  
boxer.colorDePelo();** //marrón

//Modificación (sobrecarga) del método colorDePelo para el objeto boxer  
***boxer*.colorDePelo = *function*()   
{ *document*.write("marrón y blanco.<br/>") }**

//llamada al método colorDePelo (sobrecargado) del objeto bóxer  
***document*.write("el boxer sobrecargado tiene el pelo de color ");  
boxer.colorDePelo();** //marrón y blanco

//llamada al método colorDePelo (no sobrecargado) del objeto labrador  
***document*.write("el labrador tiene el pelo de color ");  
labrador.colorDePelo();** //marrón. La sobrecarga del método solo afecta al objeto boxer

## Prototype extension

***dog*.prototype.velocidadMaxima = *function*()** //Añadir un método nuevo (sobrecarga) accesible a través del prototipo del constructor dog  
**{ *document*.write("40 kilómetros por hora.<br/>"); }**

//Llamada al método velocidadMaxima accesible a través del prototipo del constructor dog  
***document*.write("el labrador corre a ");  
labrador.velocidadMaxima();**

When calling a property like **.velocidadMaxima()**, JavaScript firstly search the own property (***this****)* in the constructor; if it is not declared, makes a second search in the extension or prototype

## Dot notation

The object name must be entered first to access anything **encapsulated** inside the object. Next you write a dot, then the item you want to access. This can be the name of a simple property, an item of an array property, or a call to one of the object's methods.

## Bracket notation

Bracket notation is useful for dynamic entry of data and for property names with spaces and several words.

***var* miCoche = new *Object*();** //🡪 miCoche is not a constructor!!

**miCoche.marca = "Opel"; miCoche["modelo"] = "Corsa"; miCoche.año = 2002;  
miCoche.conductores = {** // JSON del objeto **first : "Jaime",  
 second : "Sonsoles",  
 ["third"] : "Lucho"  
};** //Acceder a las propiedades del objeto

***console*.log(miCoche.conductores["second"]);  
*console*.log(miCoche["conductores"]["third"]);  
*console*.log(miCoche["conductores"].first);  
*console*.log(miCoche["marca"]);**

This looks very similar to how you access the items in an array, and it is basically the same thing — instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.

Property “conductores” represents an object written in JSON format notation. It is an object inside of an object. With the bracket notation it is possible to add properties dynamically:

***var* clave = "motor"; *var* valor = "1.200 cc gasolina";  
miCoche[clave] = valor;**//Dot notation sets “clave” (variable name) instead of “motor” (variable value) as the key

***function* mostrarPropiedadesObjeto(*objectName*) {  
*var* resultado ="";** //Abstraction

**for (*var* i in objectName)  
{   
if (objectName.hasOwnProperty(i))  
{resultado += "objectName." + i + " = " + objectName[i] + "\n";}**

**return resultado; }**

The **for……in statement** iterates over the [enumerable properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties) of an object. For each distinct property, statements can be executed.

***var* string1 = ""; *var* object1 = {a: 1, b: 2, c: 3};**

**for (*var* ownProperties in object1)   
{ string1 = string1 + object1[ownProperties]; }**

***console*.log(string1);** // expected output: "123"

***function* listarTodasLasPropiedaes(objectName) {**

***var* resultado = [];  
*var* properties;**

**for ( properties = objectName; properties !== null; properties = *Object*.getPrototypeOf(properties) )**

**{ resultado = resultado.concat(*Object*.getOwnPropertyNames(properties)) + "\n"; }**

**return resultado; }**

The ***Object*.getOwnPropertyNames()** method returns an array of all properties (including non-enumerable properties except for those which use Symbol) found directly upon a given object.

***console*.log(*Object*.getOwnPropertyNames(objectName));** //Array de propiedades ***console*.log(*Object*.getOwnPropertyNames(objectName).sort());** //ordena alfabeticamente

If you want only the enumerable properties, see ***Object*.keys()** or use a **for in** loop (although note that this will return enumerable properties not only found directly upon that object but also along the prototype chain for the object unless the latter is filtered with **hasOwnProperty()**.

***Object*.keys(objectName)** returns an array whose elements are strings corresponding to the enumerable properties found directly upon object. The ordering of the properties is the same as that given by looping over the properties of the object manually (coding order).

/\* When declaring an object, it is given its own predeterminated values --> the parameters\*/

***function* Persona(*elNombre, laEdad*) {**  ***this*.nombre = elNombre;  
 *this*.edad = laEdad;**

***this*.decirNombre = *function*() {  
 return *this*.nombre;  
 }**

**}**

***var* persona1 = new Persona("Jaime", 28);**

***console*.log( persona1.nombre );**//Malas prácticas (me paso los principios de la POO por el culo)

***console*.log( persona1.decirNombre() );** //forma correcta 🡪 Jaime

//Persona prototype

***Persona*.prototype.cambiarNombre = *function*(nuevoNombre)   
 { *this*.nombre = nuevoNombre; };  
 *Persona*.prototype.especie = “ser humano”**

**persona1.cambiarNombre( "Max Power" );  
*console*.log( persona1.decirNombre() );** //🡪Max Power

***var* persona2 = new Persona("David", 36);**

***console*.log( persona2.decirNombre() );** //🡪David **persona2.cambiarNombre( "Goliat" );  
 console.log( persona2.decirNombre() );** //🡪Goliat

**/\*+++++++++++++++ HERENCIA +++++++++++++++\*/**

//Yo soy el prototipo de mi hijo //Mi padre es mi prototipo

***function* Jedi(*nombre, edad, fuerza*) {** //Equivalente al extends **Persona.call(*this*, nombre, edad);**//llamo al constructor de persona ***this*.fuerza = fuerza;**

**}**

***var* jedi1 = new Jedi("Obi Wan", 45, 72);**

***console*.log( jedi1.fuerza );** //72 ***console*.log( jedi1.decirNombre() );**//Inherited as it was declared in superclass´ constructor (*this)*

***console*.log(jedi1.especie);** //undefined 🡪Not inherited as it is in the prototype

**//Yo (Persona) soy el prototipo de mi hijo (Jedi) //Mi padre es mi prototipo**  
***Jedi*.prototype = *Object*.create(*Persona*.prototype);**

When jedi1 was defined, the prototype of the Jedi was not instanced !!!

**jedi1.cambiarNombre( "Obi Juan ke no ve" );  
 console.log( jedi1.decirNombre() );**//decirNombre isUNDEFINED cuando definí jedi1 no estaba instanciado el prototipo del Jedi

***var* jedi2 = new Jedi("Anakin", 21, 69);**

***console*.log( jedi2.decirNombre() );** //🡪Anakin  
**jedi2.cambiarNombre("Darth Bader");  
*console*.log( jedi2.decirNombre() );** //🡪Darth Bader  
***console*.log(jedi2.color);** //🡪Blanco

**/\*+++++++++++++++ otra forma de HERENCIA +++++++++++++++\*/**

***function* Resistencia( *nombre*, *edad*, *poder* ) {  
 Persona.call( *this*, nombre, edad )  
 *this*.poder = poder;  
}**

***Resistencia*.prototype = new Persona;  
*Resistencia*.prototype.*constructor* = Resistencia;**

***var* miembro = new Resistencia ( "Vicente Fuentes", 28, 75 );**

***console*.log( miembro.decirNombre() );** //🡪Vicente Fuentes  
**miembro.cambiarNombre( "JL" );  
console.log( miembro.decirNombre() );** //🡪JL

**/\*+++++++++++++++ HERENCIA DE UN JSON +++++++++++++++\*/**

**borregoJSON = {  
 informado : "mass-mierda",  
 idolos : "famosos",  
 fe : "políticos, farmafia, etc-",  
 changeBorrego : *function*( *nombre* ) { *this*.nombre = nombre; }  
}**

***function* BorregoMatrix( *nombre*, *edad* ) {  
 Persona.call(*this*, nombre, edad);  
}**

***Object*.setPrototypeOf( *BorregoMatrix*.prototype, borregoJSON );**

***var* ciudadanoEjemplar = new BorregoMatrix( "sumiso", 53 );**

***console*.log( ciudadanoEjemplar.informado );** //🡪 mass-mierda  
***console*.log( ciudadanoEjemplar.decirNombre() );** //🡪 sumiso **ciudadanoEjemplar.changeBorrego( "siervo" );   
*console*.log( ciudadanoEjemplar.decirNombre() );** //🡪 sumiso

## Class expressions and class declaration

***class* ClassName {** // class body **};** //class declaration

***var* MyClass = *class* [derivedClassName] [extends] *[superClassName]* {** // class body **};**

A class expression has a similar syntax to a [class statement (declaration)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class). However, with class expressions, you are able to omit the class name ("binding identifier"), which you can't with class statements. The name given to a named class expression is local to the class's body. it can be retrieved through the class's[**.name**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name)property (not an instance's), though.

Additionally, class expressions allow you to redefine/re-declare classes and **don't throw** any type errors like [class declaration](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class). The **constructor()** property is optional. And, *typeof* the classes generated using this keyword will always be "function".

The **constructor(**[arguments]**) {…}** method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A constructor can use the **super** keyword to call the constructor of a parent class. If you do not specify a constructor method, a default constructor is used.

***class* Rectangle {  
 constructor( *height*, *width* ) {  
 *this*.name = "Rectangle";  
 *this*.height = height;  
 *this*.width = width;  
 }**

**perimetro() {** //Method ¡¡This is a function!!, getter and setter are not.  
 **return ( *this*.height\*2 + this.width\*2 );  
 }**

***get* perimeter() {** //Getter  
 **return *this*.perimetro();  
 }**

***get* diagonal() {**  //Getter  
 **return *Math*.sqrt( (*this*.height\*\*2) + (*this*.width\*\*2) ); }**

***get* sayPolygonName() {**  //Getter  
 **return ( "Hi, I am a " + *this*.name );  
 }**

***get* superficie() {**  //Getter  
 **return *this*.height \* *this*.width;  
 }**

**} // Close class declaration**

***var* miPoligono = new Rectangle(5, 8);**

***console*.log( miPoligono.sayPolygonName );**  //Hi, I am a Rectangle ***console*.log( miPoligono.superficie );**  //40 ***console*.log( miPoligono.diagonal );**  //9.434 ***console*.log( miPoligono.perimeter );** //26

**/\*+++++++++++++++ HERENCIA +++++++++++++++\*/**

***class* Square extends Rectangle {  
 constructor( *length* ) {   
 super( length, length );  
 *this*.name = "Square";  
 }  
 *set* newArea( *value* ) {  
 return *this*.height = *this*.width = *Math*.sqrt( value );  
 }  
}**

***var* segundoCuadrilatero = new Square(5, 5); //Square(5);** //is also valid  
***console*.log( segundoCuadrilatero.width );** //5 ***console*.log(segundoCuadrilatero.height);** //5  
**segundoCuadrilatero.newArea = 36;  
*console*.log( segundoCuadrilatero.width );** //6 ***console*.log( segundoCuadrilatero.height );** //6

If there is a constructor present in sub-class, it needs to first call **super()** before using ***this***

The **super** keyword is used to access and call functions on an object's parent. The **super.prop** and **super[expr]** expressions are valid in any method definition in both classes and objects literals. When used in a constructor, **super** keyword appears alone and must be used before the ***this*** keyword is used. **super([arguments]); // calls the parent constructor.**

The **static** keyword defines a static method for a class. Static methods are called without instantiating their class and **cannot** be called through a class instance; this means that, static method are made directly on the class and are not callable on instances of the class. Static methods are often used to create utility functions for applications. ***static* *methodName*(){}**

***class* Point {  
 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.hypot*(dx, dy); }  
}**

***const* p1 = new Point(5, 5); *const* p2 = new Point(9, 8);  
*console*.log(Point.distancia(p1, p2));** //5

Static methods are not directly accessible using the ***this*** keyword from non-static methods. You need to call them using the class name: CLASSNAME.STATIC\_METHOD\_NAME()  
or by calling the method as a property of the **constructor**: ***this***.**constructor**.STATIC\_METHOD\_NAME().

***class* StaticMethodCall {**

***static* staticMethod() {**

**return 'Static method has been called';}**

***static* anotherStaticMethod() {**

**return *this*.staticMethod() + ' from another static method';}**

**}**

**StaticMethodCall.staticMethod();** // Static method has been called

**StaticMethodCall.anotherStaticMethod();**

***class* Cat {**

**constructor( *name* ) {**

***this*.name = name;**

**}**

**speak() { *console*.log(*this*.name + ' makes a noise.'); }**

**}**

***class* Lion extends Cat {**

**speak() {**

***super*.speak();**

***console*.log(*this*.name + ' roars.');**

**}**

**}**

***var* otro = new Lion('Fuzzy');**

**otro.speak();** //Fuzzy makes a noise.

//Fuzzy roars.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor>

<https://developer.mozilla.org/es/docs/Web/JavaScript/Herencia_y_la_cadena_de_protipos>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties>

<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Modo_estricto>

# Date object

The JavaScript Date object supports a number of UTC (universal) methods, as well as local time methods. UTC, also known as Greenwich Mean Time (GMT), refers to the time as set by the World Time Standard. The local time is the time known to the computer where JavaScript is executed.

**new *Date*(); *console*.log(*Date*);** 🡪//function Date()

JavaScript Date objects can only be instantiated by calling JavaScript Date as a constructor: calling it as a regular function (i.e. without the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) operator) will return a string rather than a Date object; invoking JavaScript Date as a function (i.e., without the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) operator) will return a string representing the current date and time.

**new *Date*(***year*, *month*, *day*, *hours*, *minutes*, *seconds*, *milliseconds***);**

The value of Date.length is 7. This is the number of arguments handled by the constructor.

The argument month is 0-based. This means that January = 0 and December = 11. If no arguments are provided, the constructor creates a JavaScript Date object for the current date and time according to system settings. If at least two arguments are supplied, missing arguments are either set to 1 (if the day is missing) or 0 for all others.

## JavaScript Date instances: Date.prototype Methods

### Getter:

**getDate()**; returns the day of the month (1-31) for the specified date according to local time.  
**getDa()**; returns the day of the week (0-6) for the specified date according to local time.  
**getFullYear()**; returns the year (4 digits for 4-digit years) of the specified date according to local time.  
**getHours()**; returns the hour (0-23) in the specified date according to local time.  
**getMilliseconds()**; returns the milliseconds (0-999) in the specified date according to local time.  
**getMinutes()**; returns the minutes (0-59) in the specified date according to local time.  
**getMonth()**; returns the month (0-11) in the specified date according to local time.  
**getSeconds()**; returns the seconds (0-59) in the specified date according to local time.  
**getTime()**;returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC (negative for prior times).

All this methods also exists according to universal time: same name methods with **UTC** after **get**

**getTimezoneOffset()**; returns the time-zone offset in minutes for the current locale.  
**getYear(); r**eturns the year (usually 2-3 digits) in the specified date according to local time. Use [getFullYear()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/getFullYear) instead.

### Setter:

**setDate()**; sets the day of the month for a specified date according to local time.  
**setFullYear()**; sets the full year (e.g. 4 digits for 4-digit years) for a specified date according to local time.  
**setHours()**: sets the hours for a specified date according to local time.  
**setMilliseconds()**; sets the milliseconds for a specified date according to local time.  
**setMinutes()**; sets the minutes for a specified date according to local time.

**setMonth()**; sets the month for a specified date according to local time.  
**setSeconds()**; sets the seconds for a specified date according to local time.

**setTime()**; sets the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) object to the time represented by a number of milliseconds since.

All this methods also exists according to universal time: same name methods with **UTC** after **get**

**setFullYear()**;

### Conversion getter:

**toDateString()**; returns the "date" portion of the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) as a human-readable string.  
**toISOString()**; converts a date to a string following the ISO 8601 Extended Format.

**toJSON()**; returns a string representing the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) using [toISOString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toISOString). Intended for use by [JSON.stringify()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify).  
**toUTCString()**; returns a string representing the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) based on the GMT (UT) time zone.   
**toLocaleDateString()**; returns a string with a locality sensitive representation of the date portion of this date based on system settings.

**toLocaleFormat()**; converts a date to a string, using a format string.  
**toLocaleString()**; returns a string with a locality sensitive representation of this date. Overrides the [Object.prototype.toLocaleString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toLocaleString) method.

**toLocaleTimeString**; returns a string with a locality sensitive representation of the time portion of this date based on system settings.

**toSource()**; returns a string representing the source for an equivalent [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) object

**toTimeString()**; returns the "time" portion of the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) as a human-readable string.

**toUTCString()**; converts a date to a string using the UTC timezone.

**valueOf()**; returns the primitive value of a [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) object.

***class* MyDate extends *Date* {  
 constructor(*height, width*) {  
 *this*.name = "Rectangle";  
 *this*.height = height;  
 *this*.width = width;  
 }**

# HERENCIA

# Programación Funcional

No utiliza memoria, sino todo CPU. Útil para validación de formularios, juegos (tres en raya), etc.

Programación Funcional: **Input** 🡪return **Output**

**JSON**

var o = {

a: 7,

get b() { return this.a + 1; },

set c(x) { this.a = x / 2; }

};

Los getters y setters también pueden añadirse a un objeto en cualquier momento después de su creación utilizando el método Object.defineProperties. El primer parámetro de este método es el objeto sobre el que se quiere definir el getter o setter. El segundo parámetro es un objeto cuyo nombre de propiedad son los nombres getter o setter, y cuyos valores de propiedad son objetos para la definición de las funciones getter o setter. Aquí hay un ejemplo que define el mismo getter y utilizado getter en el ejemplo anterior:

var o = { a:0 }

Object.defineProperties(o, {

"b": { get: function () { return this.a + 1; } },

"c": { set: function (x) { this.a = x / 2; } }

});

o.c = 10 // Ejecuta el setter, que asigna 10/2 (5) a la propiedad 'a'

console.log(o.b) // Ejecuta el getter, que produce a + 1 o 6