



# MA: Clases ES6

Javier Ribal del Río

2025-12-13

## Table of contents

1	Programación orientada a objetos en JavaScript	2
2	Definición de una clase	2
2.1	Sintaxis básica . . . . .	2
3	Creación de instancias	2
4	Métodos de instancia	2
5	Métodos vs funciones flecha	3
6	Propiedades públicas	3
7	Getters y setters	4
8	Herencia (extends)	4
9	Sobrescritura de métodos	5
10	Uso de super en métodos	5
11	Métodos estáticos	5
12	Campos privados (#)	6
13	Clases y objetos literales	6
14	Clases prototípicas (pre-ES6)	7

### Contenido

Repaso intensivo de **clases en JavaScript (ES6+)**, orientado a usuarios con experiencia previa en programación y familiarizados con funciones, objetos y arrays.



# 1 Programación orientada a objetos en JavaScript

JavaScript es un lenguaje **basado en prototipos**, pero desde ES6 introduce la sintaxis `class`, que:

- Es **azúcar sintáctico** sobre el sistema de prototipos
  - Facilita la escritura y lectura de código OO
  - No convierte a JS en un lenguaje basado en clases clásicas
- 

## 2 Definición de una clase

### 2.1 Sintaxis básica

```
class Persona {  
  constructor(nombre, edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
  }  
}
```

- `class`: palabra clave
  - `constructor`: método especial de inicialización
  - `this`: referencia a la instancia actual
- 

## 3 Creación de instancias

```
const p1 = new Persona("Javier", 45);  
const p2 = new Persona("Ana", 32);
```

- `new` crea un nuevo objeto
  - Ejecuta automáticamente `constructor`
- 

## 4 Métodos de instancia

Los métodos se definen **sin function** y se comparten vía prototipo.



```
class Persona {  
  constructor(nombre, edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
  }  
  
  saludar() {  
    return `Hola, soy ${this.nombre}`;  
  }  
  
  cumple() {  
    this.edad++;  
  }  
}
```

Uso:

```
p1.saludar();  
p1.cumple();
```

---

## 5 Métodos vs funciones flecha

No usar arrow functions como métodos de clase (salvo casos concretos):

```
class MalEjemplo {  
  metodo = () => {  
    console.log(this);  
  };  
}
```

- Rompe el modelo prototípico
- Mayor consumo de memoria

---

## 6 Propiedades públicas

Las propiedades se suelen declarar en el constructor:

```
class Coche {  
  constructor(marca, km = 0) {  
    this.marca = marca;  
    this.km = km;  
  }  
}
```

Uso:



```
const c = new Coche("Toyota");  
c.km += 100;
```

---

## 7 Getters y setters

Permiten **acceder como propiedades** a lógica encapsulada.

```
class Rectangulo {  
  constructor(ancho, alto) {  
    this.ancho = ancho;  
    this.alto = alto;  
  }  
  
  get area() {  
    return this.ancho * this.alto;  
  }  
  
  set escala(factor) {  
    this.ancho *= factor;  
    this.alto *= factor;  
  }  
}
```

Uso:

```
const r = new Rectangulo(2, 3);  
r.area;      // 6  
r.escala = 2;  
r.area;      // 24
```

---

## 8 Herencia (extends)

JavaScript soporta herencia simple.

```
class Empleado extends Persona {  
  constructor(nombre, edad, salario) {  
    super(nombre, edad);  
    this.salario = salario;  
  }  
  
  salarioAnual() {  
    return this.salario * 12;  
  }  
}
```



- `extends`: herencia
  - `super()`: llama al constructor padre (obligatorio)
- 

## 9 Sobrescritura de métodos

```
class Empleado extends Persona {  
    saludar() {  
        return `Empleado: ${this.nombre}`;  
    }  
}
```

- Si el método existe en la clase hija, **sobrescribe** al del padre
- 

## 10 Uso de `super` en métodos

```
class Empleado extends Persona {  
    saludar() {  
        return super.saludar() + " (empleado)";  
    }  
}
```

Permite reutilizar lógica del padre.

---

## 11 Métodos estáticos

Pertenecen a la **clase**, no a las instancias.

```
class Utilidades {  
    static suma(a, b) {  
        return a + b;  
    }  
}
```

Uso:

```
Utilidades.suma(2, 3);
```

No accesible desde instancias:



```
new Utilidades().suma; // undefined
```

---

## 12 Campos privados (#)

Introducidos en ES2022.

```
class Cuenta {  
  #saldo = 0;  
  
  ingresar(cantidad) {  
    this.#saldo += cantidad;  
  }  
  
  getSaldo() {  
    return this.#saldo;  
  }  
}
```

- #saldo es **realmente privado**
  - No accesible fuera de la clase
- 

## 13 Clases y objetos literales

Esto:

```
const a = {  
  x: 1,  
  inc() { this.x++; }  
};
```

Es equivalente conceptualmente a **una instancia única**.

Usar `class` cuando:

- Hay múltiples instancias
  - Existe estado y comportamiento común
  - Se necesita herencia o abstracción
-



## 14 Clases prototípicas (pre-ES6)

JavaScript **siempre ha sido un lenguaje basado en prototipos**. Antes de ES6, la creación de objetos y herencia se realizaba mediante **funciones constructoras y el prototipo**.

En este repaso **no entraremos en detalle** en este modelo.

Basta con saber que:

- Las clases ES6 **no sustituyen** al modelo prototípico
- Son una **capa de abstracción** sobre él
- Todo el comportamiento sigue resolviéndose vía prototipos

Para profundizar:

- [https://developer.mozilla.org/es/docs/Web/JavaScript/Inheritance\\_and\\_the\\_prototype\\_chain](https://developer.mozilla.org/es/docs/Web/JavaScript/Inheritance_and_the_prototype_chain)

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

vs

```
class Persona {  
  constructor(nombre) {  
    this.nombre = nombre;  
  }  
  
  saludar() {  
    return this.nombre;  
  }  
}
```