

Desarrollo Web en Entorno Cliente

Tema 17



The title 'JS CLASSES' is presented within a horizontal rectangle divided into two color sections: red on the left and blue on the right. The letters 'JS' are white with a black outline and are positioned on the red background. The word 'CLASSES' is in a plain white sans-serif font and is positioned on the blue background.

JS

CLASSES

[clases]

Las clases en JavaScript fueron introducidas en el ECMAScript 2015, ES6.

La sintaxis de las clases no introduce un nuevo modelo de herencia orientada a objetos en JavaScript, son simplemente una mejora sintáctica (*syntactic sugar*) de la herencia basada en prototipos.

Las clases de JS proporcionan una sintaxis mucho más clara y simple para crear objetos y utilizar la herencia.

Para declarar una clase se utiliza la palabra reservada `class` seguida del nombre escogido para dicha clase, este nombre se escribe, por convención, en *Upper Camel Case*, y a continuación le sigue el cuerpo de la clase encerrado entre llaves `{}`.

Ejemplo:

```
class Animal {}
```

(clases)

Las clases también se pueden definir mediante expresiones de clase, y, al igual que las funciones, pueden tener nombre o ser anónimas.

El nombre que se le da a una expresión de clase es local dentro del cuerpo de la misma.

En el siguiente ejemplo se define una clase mediante una expresión de clase anónima:

```
const pato=class {};
```

```
pato  
class {}
```

En este ejemplo se define una clase mediante una expresión de clase con un nombre *Animal*:

```
const pato=class Animal{};
```

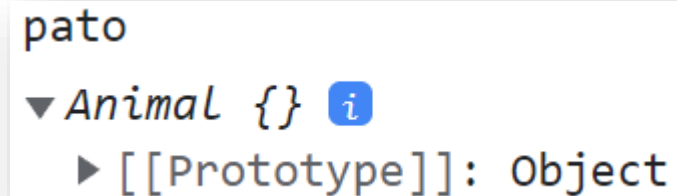
```
pato  
class Animal {}
```

[clases]

Para crear una instancia de la clase, esto es, un objeto, se utiliza la palabra reservada *new* seguida del nombre de la clase y paréntesis ().

En el siguiente ejemplo, se crea una instancia de la clase *Animal* llamado *pato*:

```
class Animal {};  
const pato=new Animal();
```



```
pato  
▼ Animal {} ⓘ  
  ► [[Prototype]]: Object
```

No obstante, a diferencia de las declaraciones de funciones, las declaraciones de clases no *gozan de Hoisting* por lo que se deben definir antes de poder ser utilizadas.

Cuando se crea una instancia se ejecuta el *constructor* definido en la clase, se ejecuta automáticamente cuando se ejecuta la palabra *new* seguido del nombre de la clase.

Este método crea e inicializa un objeto creado con una clase.

[constructor]

Para definir el constructor de una clase se utiliza la palabra reservada *constructor* seguido de paréntesis (), con los argumentos que se desean pasar al crear la instancia de la clase, y el cuerpo de la función, con sus instrucciones correspondientes, encerrado entre llaves {}.

Solo puede haber un método con el nombre *constructor* en una clase, si la clase contiene más de una ocurrencia del método constructor se produce un error de sintaxis (*Error SyntaxError*), y en el caso de que no se especifique un constructor se tendrá uno vacío de forma implícita.

Ejemplo:

```
class Animal{  
    constructor(nombre){  
        console.log("Ha nacido " + nombre);  
    }  
}
```

[constructor]

Recuerda que todas las funciones en JS devuelven algo, lo que se especifique a través de la sentencia *return* o si no se especifica dicha sentencia se devolverá *undefined*.

Ejemplo:

```
function prueba() {}  
console.log(typeof prueba());
```

undefined

Sin embargo, hay una excepción, en un constructor no se debe utilizar la palabra reservada *return*, puesto que siempre devuelve una nueva instancia de la clase al invocarse con la palabra *new*.

El constructor se utiliza para tareas de inicialización cuando se crea un nuevo objeto.

A diferencia de otros lenguajes de programación, JS no tiene destructor, el opuesto al constructor.

[propiedades]

En las clases se pueden definir propiedades, de forma análoga a como se hace en los objetos literales, que permitan guardar el estado de los instancias de la misma.

Hasta ES11 únicamente se podían definir en el interior del cuerpo del constructor, precediendo el nombre de la propiedad por la palabra reservada *this.*, con el fin de poder acceder al ámbito (*scope*) de la clase.

Ejemplo:

```
class Animal{  
  constructor(nombre){  
    this.nombre=nombre;  
    console.log("Ha nacido " + this.nombre);  
  }  
}
```

Es importante tener cuidado con la palabra reservada *this*, ya que en muchas situaciones puede pensarse que devolverá una referencia al elemento padre que la contiene, pero devolverá el objeto *Window* porque se encuentra fuera de una clase o dentro de una función con otro contexto.

this devuelve una referencia al elemento que lo invocó.

[propiedades]

Desde ECMAScript2020 (<https://262.ecma-international.org/11.0/>) se pueden declarar propiedades en la parte superior del cuerpo de la clase, justo después de la primera llave {.

De esta forma, ya no es necesario declarar las propiedades obligatoriamente dentro del constructor.

Ejemplo:

```
class Animal{  
  edad=0;  
  constructor(nombre){  
    this.nombre=nombre;  
    console.log("Ha nacido " + this.nombre);  
  }  
}
```

[propiedades]

Las propiedades no deben ir precedidas de *const* o de *var* o *let* y pueden o no ser inicializadas a un valor, incluido *undefined*.

Las propiedades tienen ámbito de la clase y pueden ser accedidas desde dentro de la clase, directamente, pero si se quiere acceder desde una función (método) de la clase, como el constructor ha de utilizarse la palabra reservada *this* como ya se ha comentado.

Ejemplo:

```
class Animal{  
  edad;  
  amigable=undefined;  
  constructor(nombre=undefined){  
    this.nombre=nombre;  
    console.log("Ha nacido " + this.nombre);  
  }  
}
```

[propiedades]

Estas propiedades existen en la clase, y se puede establecer de forma que todos los objetos tengan el mismo valor, o como que tengan valores diferentes dependiendo del objeto en cuestión, pasándole los valores específicos por parámetro en el constructor.

Para acceder a la propiedad, desde fuera del ámbito de la clase se utiliza el nombre de la instancia de la clase, esto es, el nombre del objeto, seguido de punto . y el nombre de la propiedad.

Las propiedades de la clase pueden ser accedidas y modificadas externamente, ya que por defecto son propiedades públicas.

Ejemplo:

```
class Animal{  
    edad=0;  
    amigable=undefined;  
    constructor(nombre=undefined){  
        this.nombre=nombre;  
        console.log("Ha nacido " + this.nombre);  
    }  
}  
  
const pato=new Animal("Donald");  
pato.edad=20;
```

[propiedades privadas]

A partir de ECMAScript2020 se pueden crear propiedades de clase privadas, aunque es muy probable que el navegador no soporte esta característica.

Para crear una propiedad de clase privada hay que añadir el carácter # justo antes del nombre de la propiedad.

Ejemplo:

```
class Animal{  
  #edad;  
  #amigable;  
  constructor(nombre=undefined){  
    this.nombre=nombre;  
    console.log("Ha nacido " + this.nombre);  
  }  
}  
const pato=new Animal("Donald");  
pato.#edad=20;
```

✖ Uncaught SyntaxError: Private field '#edad' must be declared in an enclosing class

[propiedades privadas]

Hay que tener especial cuidado de acceder correctamente a las propiedades mediante su nombre, ya que si accedemos por error a una propiedad mediante su nombre, la propiedad se creará en el objeto.

En el siguiente ejemplo accedemos erróneamente a la propiedad *edad* en vez de a *#edad*, no obtendremos un error de acceso a una propiedad privada, sino que crearemos una nueva propiedad en el objeto *pato*:

```
class Animal{  
  #edad;  
  #amigable;  
  constructor(nombre=undefined){  
    this.nombre=nombre;  
    console.log("Ha nacido " + this.nombre);  
  }  
}  
const pato=new Animal("Donald");  
pato.edad=20;  
console.log(pato);
```

```
▼ Animal {nombre: 'Donald', edad: 20, #edad: undefined,  
  edad: 20  
  nombre: "Donald"  
  #amigable: undefined  
  #edad: undefined  
  ► [[Prototype]]: Object
```

[métodos]

En las clases, al igual que sucedía en los objetos literales, se pueden definir métodos, que añadirán funcionalidad a la misma.

Los métodos de una clase definen su comportamiento.

Para definir un método dentro de una clase, se define igual que una función en JS, pero no se utiliza la palabra clave *function*.

Los métodos se definen dentro del cuerpo de la clase, y al igual que las funciones pueden recibir argumentos y pueden devolver valores.

Ejemplo:

```
class Animal{  
    constructor(){}  
    metodoSaludo(){console.log("Hola");}  
}
```

[métodos]

Los métodos de una clase se invocan mediante el nombre de la instancia de la clase, esto es, el nombre del objeto, seguido de punto . el nombre del método y paréntesis (), con los argumentos que se desean pasar al método si fuera el caso.

Ejemplo:

```
class Animal{  
    constructor(){}  
    metodoSaludo(){console.log("Hola");}  
}  
const pato=new Animal();  
pato.metodoSaludo();
```

Para acceder a las propiedades u otros métodos de la clase desde el cuerpo de un método hay que utilizar el nombre de la propiedad o método en particular, precedido de la palabra reservada *this* junto con el punto . , ya que dentro del cuerpo del método no se tiene acceso al ámbito (*scope*) de la clase.

[métodos]

Ejemplo:

```
class Animal{  
    constructor(nombre){this.nombre=nombre;}  
    metodoSaludo(){console.log("Hola " + this.nombre);}  
}  
const pato=new Animal("Donald");  
pato.metodoSaludo();
```

Hola Donald

[métodos privados]

También se pueden crear métodos privados, esto es métodos que no pueden invocarse desde fuera de la clase, al igual que las propiedades privadas se han definido a partir de ECMAScript2020 por lo que en la actualidad hay navegadores que no soportan esta característica.

Para crear un método de clase privado hay que añadir el carácter # justo antes del nombre del método.

Ejemplo:

```
class Animal{  
  constructor(nombre){this.nombre=nombre;}  
  #metodoSaludo(){console.log("Hola " + this.nombre);}  
}  
const pato=new Animal("Donald");  
pato.#metodoSaludo();
```

✖ Uncaught SyntaxError: Private field '#metodoSaludo' must be declared in [VM2146:6](#)
an enclosing class

[métodos privados]

Ejemplo:

```
class Animal{  
    constructor(nombre){  
        this.nombre=nombre;  
        this.#metodoSaludo();  
    }  
    #metodoSaludo(){console.log("Hola " + this.nombre);}  
}  
const pato=new Animal("Donald");
```

Hola Donald

[métodos setter]

Los *setters* son métodos de acceso a las propiedades definidas dentro de la clase, por lo que siempre son declarados públicos.

Un método *setter* (del Inglés *set*, establecer) permite asignar valores a una propiedad de la clase, y, por ello debe recibir el valor como argumento y, de igual forma, un método *setter* nunca retorna nada.

Mediante estos métodos y el uso de propiedades de clase privadas se consigue la encapsulación, ya que solo se permite dar acceso a ciertas propiedades de manera controlada a través de estos métodos.

Para crear un método *setter* se utiliza la palabra reservada *set* seguido del nombre elegido para el *set* junto con los paréntesis *()*, que contendrán el valor que se desea establecer en la propiedad y a continuación el cuerpo de la función entre llaves *{}*.

Ejemplo:

```
class Animal{  
    #nombre;  
    set setNombre(nombre){this.#nombre=nombre;}  
}
```

A veces se utiliza la terminología de propiedades computadas.

Las propiedades computadas son simplemente propiedades, pero contienen valores sobre los que se quiere realizar operaciones antes de asignarlos o recuperarlos en dichas propiedades.

[métodos setter]

El nombre elegido para el método *set* no puede ser igual al nombre de la propiedad que se establece porque se crearía un bucle infinito.

Se utiliza la convención de anteponer un guion bajo `_` delante del nombre de la propiedad y el nombre del setter directamente el nombre de la propiedad.

La forma de invocar un *setter* es igual que el uso de una propiedad, esto es, no se utilizan los paréntesis `()`, como en los métodos, en su lugar se utiliza un símbolo igual `=`, como si fuera una asignación, seguido del valor que se pasa como argumento.

Ejemplo:

```
class Animal{  
    #_nombre;  
    set nombre(nombre){this.#_nombre=nombre;}  
}  
const pato=new Animal();  
pato.nombre="Donald";
```

[métodos setter]

Observando el *prototype* del objeto puede apreciarse que aunque el setter parezca una propiedad realmente es un método *setter*.

Ejemplo:

```
class Animal{
  #_nombre;
  set nombre(nombre){this.#_nombre=nombre;}
}
const pato=new Animal();
console.log(pato);
```

```
▼ Animal {#_nombre: undefined} ⓘ
  #_nombre: undefined
  ▼ [[Prototype]]: Object
    ► constructor: class Animal
    ▼ set nombre: f nombre(nombre)
      length: 1
      name: "set nombre"
      arguments: (...)
      caller: (...)
      [[FunctionLocation]]: VM41:3
      ► [[Prototype]]: f ()
      ► [[Scopes]]: Scopes[2]
      ► [[Prototype]]: Object
```

[métodos getter]

Los *getters* son métodos de acceso a las propiedades definidas dentro de la clase, por lo que, al igual que los métodos *setters*, siempre son declarados públicos.

Un método *getter* (del Inglés *get*, obtener) permite obtener el valor ya asignado a una propiedad de la clase, y, por ello no debe recibir ningún argumento y, de igual forma, un método *getter* devolver siempre algo, el valor de la propiedad.

Mediante estos métodos y el uso de propiedades de clase privadas se consigue la encapsulación, ya que solo se permite dar acceso a ciertas propiedades de manera controlada a través de los *setters* y *getters*.

Para crear un método *getter* se utiliza la palabra reservada *get* seguido del nombre elegido para el *get* junto con los paréntesis *()*, sin ningún parámetro, y a continuación el cuerpo de la función entre llaves *{}*.

Ejemplo:

```
class Animal{  
    #nombre;  
    get getNombre(){return this.#nombre;}  
}
```

[métodos getter]

El nombre elegido para el método *get* , al igual que sucedía con *set*, no puede ser igual al nombre de la propiedad que se establece porque se crearía un bucle infinito.

También se utiliza la convención de anteponer un guion bajo `_` delante del nombre de la propiedad y el nombre del *getter* directamente el nombre de la propiedad.

La forma de invocar un *getter* es igual que el uso de una propiedad, esto es, no se utilizan los paréntesis `()`, como en los métodos, en su lugar se utiliza directamente el nombre de la propiedad.

Ejemplo:

```
class Animal{
    #_nombre;
    set nombre(nombre){this.#_nombre=nombre;}
    get nombre(){return this.#_nombre;}
}
const pato=new Animal();
pato.nombre="Donald";
console.Log(pato.nombre);
```

propiedades y métodos estáticos

Un método o propiedad estática es un método o propiedad que puede utilizarse directamente en la clase, sin necesidad de crear un objeto o una instancia de la misma.

Para crear un método o propiedad estática hay que anteponer al nombre del método o propiedad la palabra reservada *static*.

Ejemplo:

```
class Animal{  
    static _nombre='';  
    static set nombre(nombre){this._nombre=nombre;}  
    static get nombre(){return this._nombre;}  
}
```


propiedades y métodos estáticos

Considera el siguiente ejemplo:

```
class Animal{  
    static _nombre='';  
    static set nombre(nombre){this._nombre=nombre;}  
    static get nombre(){return this._nombre;}  
}  
Animal.nombre="Jorge";  
const pato= new Animal();  
pato.nombre;
```

undefined

Aunque podría pensarse que la ejecución de *pato.nombre* iba a devolver *Jorge* no ha sido así, esto se debe a que se está preguntando realmente por una propiedad de la clase llamada *nombre* y ésta no existe, y no por el método *getter* como podría pensarse, ya que al definirlo como estático no está en las instancias sino únicamente en la clase.

propiedades y métodos estáticos

Como puede apreciarse en el siguiente ejemplo, la instancia de la clase *Animal*, llamada *pato* está vacía.

Solo a través del *prototype* se accede a su constructor donde se observa que se guarda la propiedad estática *_nombre* y los métodos *getter* y *setter* estáticos.

```
class Animal{
  static _nombre='';
  static set nombre(nombre){this._nombre=nombre;}
  static get nombre(){return this._nombre;}
}
Animal.nombre="Jorge";
const pato= new Animal();
console.log(pato);
```

```
▼ Animal {} ⓘ
  ▼ [[Prototype]]: Object
    ▼ constructor: class Animal
      _nombre: "Jorge"
      length: 0
      name: "Animal"
      nombre: (...)
      ► prototype: {constructor: f}
        arguments: (...)
        caller: (...)
        ► get nombre: f nombre()
        ► set nombre: f nombre(nombre)
          [[FunctionLocation]]: VM814:1
        ► [[Prototype]]: f ()
        ► [[Scopes]]: Scopes[2]
        ► [[Prototype]]: Object
```

propiedades y métodos estáticos

Haciendo uso de los conocimientos expuestos en el tema anterior, esto es, recorriendo la cadena de prototipos, sí que podríamos acceder a la variable `_nombre`:

```
class Animal{
  static _nombre='';
  static set nombre(nombre){this._nombre=nombre;}
  static get nombre(){return this._nombre;}
}
Animal.nombre="Jorge";
const pato= new Animal();
console.log(pato.__proto__.constructor._nombre);
```

'Jorge'

```
▼ Animal {} ⓘ
  ▼ [[Prototype]]: Object
    ▼ constructor: class Animal
      _nombre: "Jorge"
      length: 0
      name: "Animal"
      nombre: (...)
      ► prototype: {constructor: f}
        arguments: (...)
        caller: (...)
        ► get nombre: f nombre()
        ► set nombre: f nombre(nombre)
          [[FunctionLocation]]: VM814:1
        ► [[Prototype]]: f ()
        ► [[Scopes]]: Scopes[2]
        ► [[Prototype]]: Object
```

propiedades y métodos estáticos

Considera los siguientes ejemplos:

```
class Animal{nombre='';}  
Animal.nombre="Jorge";  
console.log(Animal.nombre);
```

'Jorge'

```
class Animal{static nombre='';}  
Animal.nombre="Jorge";  
console.log(Animal.nombre);
```

'Jorge'

Aunque parece que el resultado y la funcionalidad es la misma, no es así:

```
class Animal{nombre='';}  
Animal.nombre="Jorge";  
const pato=new Animal();  
console.log(pato);
```

```
▼ Animal {nombre: ''} ⓘ  
  nombre: ""  
  ► [[Prototype]]: Object
```

```
class Animal{static nombre='';}  
Animal.nombre="Jorge";  
const pato=new Animal();  
console.log(pato);
```

```
▼ Animal {} ⓘ  
  ► [[Prototype]]: Object
```

propiedades y métodos estáticos

Las propiedades estáticas son útiles, por ejemplo, pueden usarse para saber cuántas instancias de la clase se han creado.

Ejemplo:

```
class Animal{
  static _instancias=0;
  constructor(){
    this._nombre='';
    Animal._instancias++;
  }
  set nombre(nombre){this._nombre=nombre;}
  get nombre(){return this._nombre;}
}
const pato=new Animal();
console.log(Animal._instancias);
```

1

propiedades y métodos estáticos

Con lo expuesto anteriormente, es fácil entender que los métodos estáticos no deben acceder a las propiedades que no sean estáticas, ya que trabajan solo con la clase y no con las instancias de la clase, los objetos.

Si un método estático trata de acceder a una propiedad no estática no obtendrá el valor de dicha propiedad y tampoco se genera un error, se obtendrá *undefined*.

Ejemplo:

```
class Animal{  
    nombre='Jorge';  
    static mostrarNombre(){console.log(this.nombre);}  
}  
Animal.mostrarNombre();
```

undefined

propiedades y métodos estáticos

JS permite crear propiedades y métodos estáticos fuera del cuerpo de una clase.

Para ello, asignamos un valor a una nueva propiedad de la clase. Esto al igual que sucedía con los objetos literales, creará una nueva propiedad dentro de la clase con carácter estático, por lo que no estará presente en sus instancias, los objetos de esa clase.

Ejemplo:

```
class Animal{}  
Animal.nombre='Jorge';  
Animal.mostrarNombre=function(){console.log(Animal.nombre);}  
Animal.mostrarNombre();
```

Jorge

Observa como no se hace uso de la palabra reservada *this* ya que no se trata de acceder a instancias sino a la propia clase.

propiedades y métodos estáticos

Por último, se puede utilizar métodos estáticos para conseguir una especie de “polimorfismo” sobre el constructor, mediante una sobrecarga de operadores.

Para ello, creamos un método estático que devuelva una nueva instancia de la clase y que será invocado cuando necesitemos crear una instancia pero los parámetros no se correspondan exactamente con los del constructor.

Ejemplo:

```
class Animal{  
  nombre;  
  edad;  
  static porObjeto({nombre,edad}){  
    const tempo=new Animal(nombre);  
    tempo.edad=edad  
    return tempo;}  
  constructor(nombre){this.nombre=nombre;}  
}  
const datos={nombre:'Jorge',edad:20}  
const pato=Animal.porObjeto(datos);  
console.log(pato);
```

```
▼ Animal {nombre: 'Jorge', edad: 20} ⓘ  
  edad: 20  
  nombre: "Jorge"  
  ► [[Prototype]]: Object
```


[herencia]

Para hacer uso de la herencia, esto es, heredar los métodos y las propiedades de una clase existente en otra, se utiliza la palabra reserva *extends* a continuación del nombre la clase hija, la subclase, en su declaración, y seguido de la palabra reserva *extends* el nombre de la clase padre, la superclase.

Ejemplo:

```
class Animal{
  _nombre;
  set nombre(nombre){this._nombre=nombre;}
  get nombre(){return this._nombre;}
}
class Ave extends Animal{
  volar(){console.log(this.nombre + (" volando"));}
}
const pato=new Ave();
pato.nombre="Donald";
pato.volar();
```

Donald volando

[herencia]

Si en el constructor de la subclase se quiere hacer uso de alguna propiedad o método de la subclase o de la superclase se debe invocar primero el constructor de la superclase.

Para ello se utiliza la palabra reservada *super* que siempre hace referencia a la superclase.

Ejemplo:

```
class Animal{
  _nombre;
  set nombre(nombre){this._nombre=nombre;}
  get nombre(){return this._nombre;}
}
class Ave extends Animal{
  constructor(nombre){
    super();
    this._nombre=nombre;}
}
const pato=new Ave("Donald");
console.log(pato.nombre);
```

JS solo utiliza herencia simple, por lo que la palabra *super*, no tiene ningún error en acceder a la superclase.

[herencia]

Las subclases pueden sobrescribir los métodos que heredan de la super clase.

Ejemplo:

Aunque pueden invocarse los métodos de la super clase a través de la palabra *super* seguido de punto . y el nombre del método a llamar.

Ejemplo:

```
class Animal{  
    hablar(){console.Log("Hola");}  
}  
class Ave extends Animal{  
    hablar(){console.Log("cuak");}  
}  
const pato=new Ave();  
pato.hablar();
```

cuak

```
class Ave extends Animal{  
    hablar(){  
        super.hablar();  
        console.Log("cuak");}}  
const pato=new Ave();  
pato.hablar();
```

Hola

cuak

Bibliografía y recursos online

- <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Classes/constructor>
- <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Classes>
- <https://lenguajejs.com/javascript/caracteristicas/clases-es6/>
- <https://www.arkaitzgarro.com/javascript/capitulo-9.html>
- <https://262.ecma-international.org/11.0/>
- <https://ricardogeek.com/que-hay-de-nuevo-en-ecmascript-2020/>