DWC

(Desarrollo Web en entorno cliente)

Índice

1.- ¿Qué es una Clase?

Aunque Javascript no soporta clases de forma nativa, en **ECMAScript 2015** se introduce la posibilidad de usar clases como en otros lenguajes, aunque internamente Javascript traduce estas clases al ***sistema basado en prototipos*** que usa en realidad. Para los programadores funciona a modo de **azúcar sintáctico**, es decir, sólo «endulza» la forma de trabajar para que sea más agradable para nosotros.

Una **clase** es una forma de organizar código de forma entendible con el objetivo de simplificar el funcionamiento de nuestro programa. Además, hay que tener en cuenta que las clases son «conceptos abstractos» de los que se pueden crear objetos de programación, cada uno con sus características concretas.

En Javascript se utiliza una sintaxis muy similar a otros lenguajes como, por ejemplo, Java. Declarar una clase es tan sencillo como escribir lo siguiente:

```
// Declaración de una clase
class Animal {}

// Crear o instanciar un objeto
const pato = new Animal();
```

El nombre elegido debería hacer referencia a la información que va a contener dicha clase. El objetivo de las clases es almacenar en ella todo lo que tenga relación (*en este ejemplo, con los animales*).
Luego creamos una variable donde hacemos un **new Animal()**. Estamos creando una variable **pato** (*un objeto*) que es de tipo **Animal**, y que contendrá todas las características definidas dentro de la clase **Animal** (*de momento, vacía*).
Una norma de estilo en el mundo de la programación es que las **clases** deben siempre **empezar en mayúsculas**. Esto nos ayudará a diferenciarlas sólo con leerlas.

2.- Elementos de una clase

Una clase tiene diferentes características que la forman, vamos a ir explicándolas todas detalladamente.

Propiedad: Variable que existe dentro de una clase. Puede ser pública o privada.
Propiedad pública: Propiedad a la que se puede acceder desde fuera de la clase.
Propiedad privada: Propiedad a la que no se puede acceder desde fuera de la clase.
Propiedad computada: Función para acceder a una propiedad con modificaciones (getter/setter).
Método: Función que existe dentro de una clase. Puede ser pública o privada.
Método público: Método que se puede ejecutar desde dentro y fuera de la clase.
Método privado: Método que sólo se puede ejecutar desde dentro de la clase.
Método estático: Método que se ejecuta directamente desde la clase, no desde la instancia.
Constructor: Método que se ejecuta automáticamente cuando se crea una instancia.

Como vemos, todas estas características se dividen en dos grupos: las **propiedades** (*a grandes rasgos, variables dentro de clases*) y los **métodos** (*a grandes rasgos, funciones dentro de clases*). Veamos cada una de ellas en detalle, pero empecemos por los **métodos**.

3.- Métodos

Hasta ahora habíamos visto que los **métodos** eran funciones que viven dentro de una variable, más concretamente de un objeto. Si añadimos un método a la clase **Animal**, al crear cualquier variable haciendo un **new Animal()**, tendrá automáticamente ese método disponible. Podemos crear varias variables de tipo **Animal** y serán totalmente independientes cada una:

```
// Declaración de clase
class Animal {
  // Métodos
  hablar() {
    return "Cuak";
  }
}

// Creación de una instancia u objeto
const pato = new Animal();
pato.hablar(); // 'Cuak'

const donald = new Animal();
donald.hablar(); // 'Cuak'
```

Observa que el método **hablar()**, que se encuentra dentro de la clase **Animal**, existe en las variables **pato** y **donald** porque realmente son de tipo **Animal**. Al igual que con las funciones, se le pueden pasar varios parámetros al método y trabajar con ellos como venimos haciendo normalmente con las funciones.

3.1.- Métodos estáticos

En el caso anterior, para usar un método de una clase, como por ejemplo **hablar()**, debemos crear el objeto basado en la clase haciendo un **new** de la clase. Lo que se denomina crear un objeto o una instancia de la clase. En algunos casos, nos puede interesar crear **métodos estáticos** en una clase porque para utilizarlos no hace falta crear ese objeto, sino que se pueden ejecutar directamente sobre la clase directamente:

```
class Animal {
  static despedirse() {
    return "Adiós";
  }

  hablar() {
    return "Cuak";
  }
}

Animal.despedirse(); // 'Adiós'
```

Como veremos más adelante, lo habitual suele ser utilizar métodos normales (*no estáticos*), porque normalmente nos suele interesar crear varios objetos y guardar información diferente en cada uno de ellos, y para eso tendríamos que instanciar un objeto.

Una de las limitaciones de los **métodos estáticos** es que en su interior sólo podremos hacer referencia a elementos que también sean estáticos. No podremos acceder a propiedades o métodos no estáticos, ya que necesitaríamos instanciar un objeto para hacerlo.

Los **métodos estáticos** se suelen utilizar para crear funciones de apoyo que realicen tareas concretas o genéricas, porque están relacionadas con la clase en general.

3.2.- El constructor

Se le llama **constructor** a un tipo especial de método de una clase, que se ejecuta automáticamente a la hora de hacer un **new** de dicha clase. Una clase **solo puede tener un constructor**, y en el caso de que no se especifique un constructor a una clase, tendrá uno vacío de forma implícita.

Veamos el ejemplo anterior, donde añadiremos un constructor a la clase:

```
// Declaración de clase
class Animal {
  // Método que se ejecuta al hacer un new
  constructor() {
    console.log("Ha nacido un pato.");
  }
  // Métodos
  hablar() {
    return "Cuak";
  }
}

// Creación de una instancia u objeto
const pato = new Animal(); // 'Ha nacido un pato'
```

El **constructor** es un mecanismo muy interesante y utilizado para tareas de inicialización o que quieres realizar tras haber creado el nuevo objeto. Otros

lenguajes de programación tienen concepto de **destructor** (*el opuesto al constructor*), sin embargo, en Javascript no existe este concepto.

Ojo: En un constructor no se puede utilizar nunca un **return**, puesto que al hacer un **new** se devuelve siempre el propio objeto creado.

4.- Propiedades

Las clases, siendo estructuras para guardar información, pueden guardar variables con su correspondiente información. Dicho concepto se denomina **propiedades** y en Javascript se realiza en el interior del constructor, precedido de la palabra clave **this** (*que hace referencia a «este» elemento, es decir, la clase*), como puedes ver en el siguiente ejemplo:

```
class Animal {
  constructor(n = "pato") {
    this.nombre = n;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos
const pato = new Animal();
pato.quienSoy(); // 'Hola, soy pato'

const donald = new Animal("Donald");
pato.quienSoy(); // 'Hola, soy Donald'
```

Desde **ECMAScript** se pueden declarar propiedades en la parte superior de la clase, justo después de abrir el **{** del **class**. De esta forma, ya no es necesario utilizar la palabra clave **this** ni declararlas obligatoriamente dentro del **constructor()**.

Como se puede ver, estas **propiedades** existen en la clase, y se puede establecer de forma que todos los objetos tengan el mismo valor, o como en el

ejemplo anterior, tengan valores diferentes dependiendo del objeto en cuestión, pasándole los valores específicos por parámetro.
Observa que, las propiedades de la clase podrán ser modificadas externamente, ya que por defecto son **propiedades públicas**:

```
const pato = new Animal("Donald");
pato.quienSoy(); // 'Hola, soy Donald'

pato.nombre = "Paco";
pato.quienSoy(); // 'Hola, soy Paco'
```

4.1.- Propiedades y métodos estáticos

A partir de la versión **ECMAScript** , se introduce la posibilidad de crear campos de clase privados (*los cuales aún cuentan con poco soporte*). Antiguamente todas las propiedades y métodos eran públicos por defecto, pero ahora también pueden ser privados. Para ello, solo hay que añadir el carácter # justo antes del nombre de la propiedad o método:

```
class Animal {
  #miSecreto = "Me gusta Internet Explorer";

  #decirSecreto() {
    return this.#miSecreto;
  }

  decirSacrilegio() {
    return this.#decirSecreto();
  }
}

const patitoFeo = new Animal();
patitoFeo.#decirSecreto();     // Error
patitoFeo.decirSacrilegio();   // OK
```

Estas propiedades o métodos precedidos de # son privados y sólo podrán ser llamados desde un método de clase, ya que si se hace desde fuera obtendremos un error similar al siguiente:
Uncaught SyntaxError: Private field '#decirSecreto' must be declared in an enclosing class

Sin embargo, si se llama a un método público como **decirSacrilegio()**, que a su vez llama a un método privado (*pero desde dentro de la clase*), todo funcionará correctamente sin error, ya que el método **#decirSecreto()** se está llamando desde dentro de la clase.

5.- Los ámbitos en una clase

Dentro de una clase tenemos dos tipos de ámbitos: **ámbito de método** y **ámbito de clase**:

En primer lugar, veamos el **ámbito dentro de un método**. Si declaramos variables o funciones dentro de un método con **var**, **let** o **const**, estos elementos existirán sólo en el método en cuestión. Además, no serán accesibles desde fuera del método:

```
class Clase {
  constructor() {
    const name = "Manz";
    console.log("Constructor: " + name);
  }

  metodo() {
    console.log("Método: " + name);
  }
}

const c = new Clase(); // 'Constructor: Manz'

c.name; // undefined
c.metodo(); // 'Método: '
```

Observa que la variable **name** solo se muestra cuando se hace referencia a ella dentro del **constructor()** que es donde se creó y donde existe.

En segundo lugar, tenemos el **ámbito de clase**. Podemos crear propiedades precedidas por **this.** (*desde dentro del constructor*) y desde la parte superior de la clase, lo que significa que estas propiedades tendrán alcance en toda la clase, tanto desde el constructor, como desde otros métodos del mismo:

```
class Clase {
  role = "Teacher";

  constructor() {
    this.name = "Manz";
    console.log("Constructor: " + this.name);
  }

  metodo() {
    console.log("Método: " + this.name);
  }
}

const c = new Clase(); // 'Constructor: Manz'

c.name;     // 'Manz'
c.metodo(); // 'Método: Manz'
c.role;     // 'Teacher'
```

Ojo, estas propiedades también pueden ser modificadas desde fuera de la clase, simplemente asignándole otro valor. Si quieres evitarlo, añade el # antes del nombre de la propiedad al declararla.

6.- This

Como te habrás fijado en ejemplos anteriores, hemos introducido la palabra clave **this**, que hace referencia al **elemento padre** que la contiene. Así pues, si escribimos **this.nombre** dentro de un método, estaremos haciendo referencia a la propiedad **nombre** que existe dentro de ese objeto. De la misma forma, si escribimos **this.hablar()** estaremos ejecutando el método **hablar()** de ese objeto.

Veamos el siguiente ejemplo, volviendo al símil de los animales:

```
class Animal {
  constructor(n = "pato") {
    this.nombre = n;
  }

  hablar() {
    return "Cuak";
  }
```

```
  quienSoy() {
    return "Hola, soy " + this.nombre + ". ~" + this.hablar();
  }
}

const pato = new Animal("Donald");

pato.quienSoy(); // 'Hola, soy Donald. ~Cuak'
```

Ten en cuenta que, si usas **this** en contextos concretos, como por ejemplo fuera de una clase te devolverá el objeto **Window**, que no es más que una referencia al objeto global de la pestaña actual donde nos encontramos y tenemos cargada la página web.

Es importante tener mucho cuidado con la palabra clave **this**, ya que en muchas situaciones creeremos 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. Asegúrate siempre de que **this** tiene el valor que realmente crees que tiene.

7.- Propiedades computadas

En algunos casos nos puede interesar utilizar lo que se llaman **propiedades computadas**. Las **propiedades computadas** son un tipo de propiedades a las que queremos realizarle ligeros cambios antes de guardarla o antes de obtenerla.

Imagina un caso en el que, tenemos una clase con 3 propiedades **A**, **B** y **C** que guardan valores específicos. Sin embargo, **B** y **C** guardan unos valores que se precalculan con unas fórmulas pero que parten del valor de la propiedad **A**. En lugar de guardar las 3 propiedades por separadas y tener que mantenerlas actualizadas, podemos simplemente crear una propiedad **A**, y una propiedad computada **B** y **C**, que obtendrán el valor de **A** y aplicarán la formula en cuestión para devolver el valor resultante.

Por ejemplo, en una clase **Circulo** podríamos tener una propiedad **radio** con un valor numérico y una propiedad computada **area** que devuelve ese valor numérico elevado por **2** y multiplicado por **π**, ya que el **área** de un círculo es **π · radio²**.

7.1.- Los getters

Los **getters** son la forma de definir propiedades computadas de lectura en una clase. Veamos un ejemplo sobre el ejemplo anterior de la clase **Animal**:

```
class Animal {
  constructor(n) {
    this._nombre = n;
  }

  get nombre() {
    return "Sr. " + this._nombre;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos
const pato = new Animal("Donald");

console.log(pato.nombre); // 'Sr. Donald'
pato.nombre = "Lucas"; // No se puede modificar un get
console.log(pato.nombre); // 'Sr. Donald'
```

Si observas los resultados de este último ejemplo, puedes comprobar que la diferencia al utilizar **getters** es que las propiedades con **get** no se pueden cambiar, son de sólo lectura.

7.2.- Los setters

De la misma forma que tenemos un **getter** para obtener información mediante **propiedades computadas**, también podemos tener un **setter**, que es el mismo concepto, pero en lugar de obtener información, para establecer información.

Si incluimos un **getter** y un **setter** a una propiedad en una clase, podremos modificarla directamente:

```
class Animal {
  constructor(n) {
    this.nombre = n;
  }

  get nombre() {
    return "Sr. " + this._nombre;
  }

  set nombre(n) {
    this._nombre = n.trim();
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos
const pato = new Animal("Donald");

console.log(pato.nombre); // 'Sr. Donald'
pato.nombre = "   Lucas  "; // '   Lucas  '
console.log(pato.nombre); // 'Sr. Lucas'
```

Observa que de la misma forma que con los **getters**, podemos realizar tareas sobre los parámetros del **setter** antes de guardarlos en la propiedad interna. Esto nos servirá para hacer modificaciones previas, como por ejemplo, en el ejemplo anterior, realizando un **trim()** para limpiar posibles espacios antes de guardar esa información.

En los ejemplos anteriores hemos utilizado el símbolo underscore(**-**) para la nomenclatura de la propiedad nombre ***this._nombre*** , eso permite diferenciar entre la propiedad y los métodos get y set

8.- Prototype en JavaScript

JavaScript es un lenguaje dinámico. Puede adjuntar nuevas propiedades a un objeto en cualquier momento, como se muestra a continuación.

```
function Student() {
    this.name = 'John';
    this.gender = 'Male';
}

var studObj1 = new Student();
studObj1.age = 15;
alert(studObj1.age); // 15

var studObj2 = new Student();
alert(studObj2.age); // undefined
```

Como puede ver en el ejemplo anterior, la propiedad age se adjunta a la instancia de studObj1. Sin embargo, la instancia de studObj2 no tendrá la propiedad de edad porque está definida solo en la instancia de studObj1.

Entonces, ¿qué hacer si queremos agregar nuevas propiedades en una etapa posterior a una función que se compartirá en todas las instancias? La respuesta es **Prototype** .

El prototype es un objeto que está asociado con todas las funciones y objetos de forma predeterminada en JavaScript, donde la propiedad del prototype de la función es accesible y modificable y la propiedad del prototype del objeto (también conocido como atributo) no es visible. Cada función incluye un objeto prototype por defecto.

El objeto prototype es un tipo especial de objeto enumerable al que se le pueden adjuntar propiedades adicionales que se compartirán entre todas las instancias de su función constructora. Entonces, usaremos la propiedad prototype de la función del ejemplo anterior para tener propiedades de edad en todos los objetos.

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}
```

```
Student.prototype.age = 15;

var studObj1 = new Student();
alert(studObj1.age); // 15

var studObj2 = new Student();
alert(studObj2.age); // 15
```

Cada objeto que se crea usando sintaxis literal o **sintaxis de constructor** con la palabra clave **new**, incluye una propiedad __**proto**__ que apunta al objeto prototipo de la función que creó este objeto.

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

var studObj = new Student();

console.log(Student.prototype); // object
console.log(studObj.prototype); // undefined
console.log(studObj.__proto__); // object

console.log(typeof Student.prototype); // object
console.log(typeof studObj.__proto__); // object

console.log(Student.prototype === studObj.__proto__ ); // true
```

Como se puede ver en el ejemplo anterior, se puede acceder a la propiedad del prototipo de la función usando <nombre-función> .prototype. Sin embargo, un objeto (instancia) no expone la propiedad del prototipo, sino que se accede a él utilizando __proto__.

Es más adecuado utilizar el método Object.getPrototypeOf(obj) en lugar de __proto__acceder al objeto prototipo.

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

var studObj = new Student();
```

```
Student.prototype.sayHi= function(){
    alert("Hi");
};

var studObj1 = new Student();
var proto = Object.getPrototypeOf(studObj1);  // returns Student's prototype object

alert(proto.constructor); // returns Student function
```

El objeto prototipo incluye las siguientes propiedades y métodos.

Propiedad	Descripción
constructor	Devuelve una función que creó la instancia.
__proto__	Esta es la propiedad invisible de un objeto. Devuelve el objeto prototipo de una función a la que se vincula.

Método	Descripción
hasOwnProperty ()	Devuelve un valor booleano que indica si un objeto contiene la propiedad especificada como una propiedad directa de ese objeto y no se hereda a través de la cadena de prototipos.
isPrototypeOf ()	Devuelve una indicación booleana si el objeto especificado está en la cadena de prototipo del objeto al que se llama este método.
propertyIsEnumerable ()	Devuelve un valor booleano que indica si la propiedad especificada es enumerable o no.
toLocaleString ()	Devuelve una cadena en formato local.
toString()	Devuelve cadena.
valueOf	Devuelve el valor primitivo del objeto especificado.

Uso de prototipo

El motor de JavaScript está utilizando el objeto prototipo en dos cosas:

1) para encontrar propiedades y métodos de un objeto

2) para implementar la herencia en JavaScript.

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}
```

```
Student.prototype.sayHi = function(){
   alert("Hi");
};

var studObj = new Student();
studObj.toString();
```

En el ejemplo anterior, el método toString () no está definido en Student, entonces, ¿cómo y desde dónde encuentra toString ()?

Aquí, el prototipo entra en escena. En primer lugar, el motor de JavaScript verifica si el método toString () está adjunto a studObj. (Es posible adjuntar una nueva función a una instancia en JavaScript).
Si no encuentra allí, entonces usa el __proto__enlace de studObj que apunta al objeto prototipo de la función Student. Si todavía no puede encontrarlo allí, sube en la jerarquía y verifica el objeto prototipo de la función Object porque todos los objetos se derivan de Object en JavaScript, y busca el método toString (). Por lo tanto, encuentra el método toString () en el objeto prototipo de la función Object y podemos llamar a studObj.toString ().

De esta manera, el prototipo es útil para **mantener solo una copia de funciones para todos los objetos (instancias)**.

A través del prototype podemos modificar los objetos incorporados a sistema, para ello sólo deberemos añadirle las funcionalidades desde el prototype y dese ese momento estarán disponibles para todas las instancias de ese tipo de objeto.

Vamos a definir un método nuevo para la clase Date que se llamara **fecha** y que le pasaremos una instancia de un objeto Date y nos devolverá la fecha en un string con un formato “dd/mm/aaaa”.
Para ello deberemos modificar el prototipo de la clase Date que ya existe para añadirle ese método a través de una función. En el momento que este definida ya estará disponible para todas las instancias de la clase Date a las que queramos aplicárselo.

```
<html>
<script>

"use strict"

Date.prototype.fecha=function(){
                        let dia=this.getDate()<10?"0"+this.getDate():this.getDate();
                        let mes=this.getMonth()<10?"0"+this.getMonth():this.getMonth();
                        return dia+"/" + mes +"/" + this.getFullYear()
                        }

let d=new Date();
alert (d.fecha());

</script>
</html>
```