Contenido

¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS?

La **Programación Orientada a Objetos** (*POO, o en inglés OOP*) es un estilo de programación muy utilizado, donde creas y utilizas estructuras de datos de una forma **muy similar a la vida real**, lo que facilita considerablemente la forma de planificar y preparar el código de tus programas o aplicaciones.

Una de las partes más complejas cuando estás empezando en el mundo de la programación (*o incluso cuando ya llevas tiempo*) es a la hora de **crear las estructuras de datos**. Con ejemplos sencillos, esto no es un problema, sin embargo, cuando los ejercicios se complican, una buena elección de una estructura de datos adecuada puede simplificar mucho el ejercicio, o complicarlo demasiado.

Problemas comunes

Cuando comenzamos a programar, nuestros ejemplos son bastante sencillos y faciles de controlar y modificar. Sin embargo, a medida que tenemos que programar cosas más complejas, todo se vuelve más complicado de organizar. Uno de los problemas más fáciles de observar, es que comenzamos a tener una gran cantidad de variables y funciones, que al estar inconexas en nuestro código, es fácil que al seguir añadiendo más variables y funciones, nuestro código se descontrole y se vuelva muy difícil de entender.

Por esa razón, necesitamos una forma de organizar las variables y constantes, las funciones y tenerlo todo bien agrupado, de modo que con el tiempo, sea sencillo de entender, modificar y ampliar. Esa agrupación, en programación, se denomina **Clase**.

Orientación a objetos

El concepto de **orientación a objetos** se ve muy claro cuando tenemos en nuestra mente el concepto de **Clase**. Todos los elementos relacionados con esa Clase los vamos a incluir en su interior. Por un lado, las variables y constantes que teníamos «sueltas» en nuestro programa, las agruparemos dentro de una clase, donde también incluiremos todas las funciones.

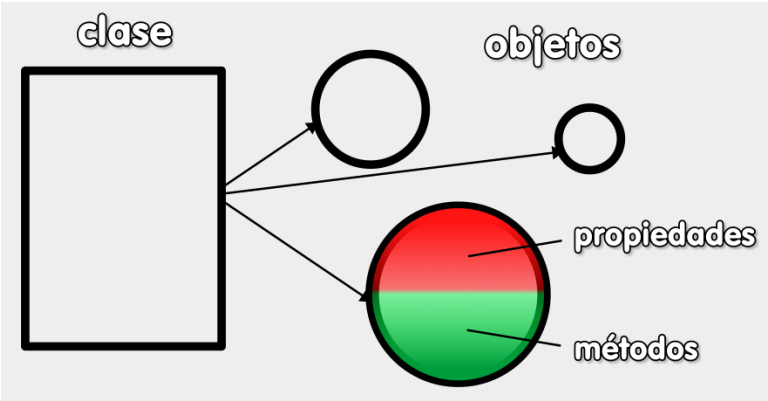
Las variables y constantes incluidas en una clase se denominan **propiedades**, y se utilizan para guardar información relacionada (*se suele denominar estado*). Por otro lado, las funciones incluidas en una clase se denominan **métodos** y se utilizan para realizar una acción relacionada con la clase.

Ejemplo

Piensa, por ejemplo, en el protagonista (*héroe*) de un juego. Tiene una serie de **variables** relacionadas con él (*vidas, fuerza, energía, etc...*), pero también tiene una serie de **funciones** relacionadas con él (*hablar, disparar, curar, etc...*). Todas ellas, las podríamos agrupar en una clase porque tienen relación con ese concepto de **personaje**:

```
Personaje
----------
- Vidas restantes (propiedad)     # Número de vidas que le quedan al personaje
- Fuerza (propiedad)              # Número que representa la fuerza del personaje
- Energía (propiedad)             # Número que representa la energía de la vida actual
- Velocidad (propiedad)           # Número que representa la velocidad actual del personaje
- Hablar (método)                 # Función que hará que el personaje diga algo
- Disparar (método)               # Función que hará que el personaje dispare con su arma
- Curar (método)                  # Función que hará que el personaje use un botiquín
```

Sin embargo, el concepto de **Clase** es un concepto abstracto. En el juego, por ejemplo, podríamos tener dos héroes que podemos elegir al principio. Ambos héroes tienen los mismos atributos y funciones, pero son dos personajes diferentes. Por esa razón, en la **programación orientada a objetos** se tiene un concepto llamado **Clase** y otro concepto llamado **Objeto**:



El primero de ellos, la **Clase** se refiere al concepto abstracto de personaje, mientras que el segundo de ellos, el **objeto** se refiere a un elemento **particular**. Por ejemplo, la **clase** podría ser **Personaje**, mientras que los **objetos** serían **Mario** y **Luigi**, ya que ambos se basan en un **Personaje**, pero tienen sus detalles particulares (*Mario podría tener más vida, o Luigi más energía, diferentes velocidades, etc...*).

Similitud con la vida real

Además de proveernos una forma de agrupar y organizar nuestro código y crear nuevos elementos basados en ellas sin repetirnos, las **clases** nos ofrecen una forma **similar a la vida real** de crear estructuras de datos, que de otra forma podría ser mucho más complejo.

Si por ejemplo, necesitáramos añadir una variable que indique la velocidad que tiene el personaje, podríamos añadir una propiedad denominada **velocidad** que contenga un **5**. Luego, podríamos incluir un método denominado **correr** que cambie esa propiedad **velocidad** a **10**, y un método denominado **caminar** que la vuelva a cambiar a **5**.

Como ves, se trata de una forma que se asemeja bastante al mundo real, y puede ser mucho más sencillo para nosotros crear estructuras de datos para nuestros programas

porque sólo tenemos que pensar en el elemento en la vida real, e imitarlo al programarlo.

Todo esto puede complicarse bastante, pero una vez sentadas estas bases, ya podemos comenzar a ver un poco de código para entender como funciona la **Programación orientada a objetos** en Javascript.

Una vez dominamos las bases de la programación y nuestro código va creciendo cada vez más, comprobaremos que las **variables y funciones** no suelen ser suficiente como para que nuestro código esté **bien organizado** y los mecanismos que tenemos a nuestro alcance quizás no resultan todo lo prácticos que deberían ser.

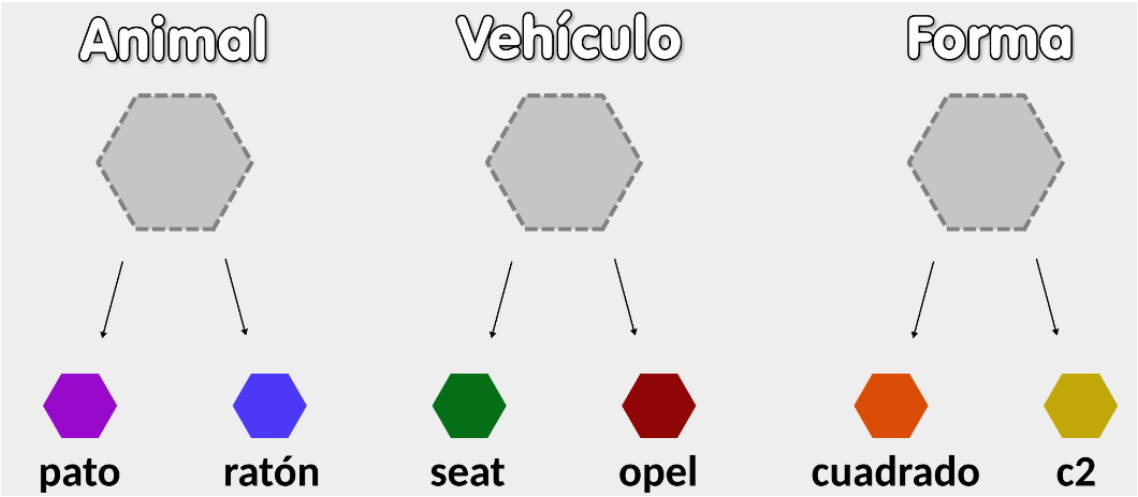
Tradicionalmente, Javascript no soportaba clases de forma nativa, pero en **ECMAScript 2015** se introdujo la posibilidad de usar clases simulando como se utilizan en otros lenguajes de programación. Internamente, Javascript traduce estas clases al sistema basado en prototipos que usa en realidad, sin embargo, los programadores no necesitarán saber como funcionan los prototipos, sino que les bastará con utilizar las clases a modo de **azúcar sintáctico**, es decir, un sistema que «endulza» la forma de trabajar para que sea más agradable y familiar.

CLASES EN JAVASCRIPT

¿Qué es una Clase?

Como mencionamos en el apartado anterior, una **clase** sólo 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.

Esto puede ser complicado de entender con palabras, pero se ve muy claro con ejemplos:



En primer lugar tenemos la **clase**. La clase es el **concepto abstracto** de un objeto, mientras que el **objeto** es el elemento final que se basa en la clase. En la imagen anterior tenemos varios ejemplos:

- En el **primer ejemplo** tenemos dos objetos: **pato** y **ratón**. Ambos son animales, por lo que son objetos que están basados en la clase **Animal**. Tanto **pato** como **ratón** tienen las características que estarán definidas en la clase **Animal**: color, sonido que emiten, nombre, etc...
- En el **segundo ejemplo** tenemos dos objetos **seat** y **opel**. Se trata de dos coches, que son vehículos, puesto que están basados en la clase **Vehículo**. Cada uno tendrá las características de su clase: color del vehículo, número de ruedas, marca, modelo, etc...
- En el **tercer ejemplo** tenemos dos objetos **cuadrado** y **c2**. Se trata de dos formas geométricas, que al igual que los ejemplos anteriores tendrán sus propias características, como por ejemplo el tamaño de sus lados. El elemento **cuadrado** puede tener un lado de **3** cm y el elemento **c2** puede tener un lado de **6** cm.

Instanciar una clase

Se le llama **instanciar una clase**, **crear un objeto** o **crear una instancia** a la acción de crear un nuevo objeto basado en una clase particular. Esta acción la realizamos a través de la palabra clave **new**, seguida del nombre de la clase, la cuál puede tener parámetros, en cuyo caso se controlarían desde un **constructor**, concepto que veremos más adelante.

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

```
// Declaración de una clase (de momento, vacía)
class Animal {}

// Crear (instanciar) un objeto basada en una clase
const pato = new Animal();
```

El nombre elegido debería hacer referencia a la información que va a contener dicha clase. Piensa que el objetivo de las clases es almacenar en ella todo lo que tenga relación (*en este ejemplo, con los animales*). Si te fijas, es lo que venimos haciendo hasta ahora con objetos como MAP, SET, ARRAY u otros(DATE,REGEXP…..).

Observa que luego creamos una variable donde hacemos un **new Animal()**. Estamos creando un objeto **pato** 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** (*nomenclatura llamada PascalCase*). Esto nos ayudará a diferenciarlas sólo con leerlas.

Miembros de una clase

Una clase tiene diferentes **características** que la forman, que generalmente se denominan **miembros**, y que normalmente son de dos tipos: **propiedades** y **métodos**. Vamos a ir explicándolas detalladamente. Pero primero, una tabla general para verlas en conjunto, con sus tipos:

Elemento	Descripción	Más información
Propiedad	Variable que existe dentro de una clase. Puede ser pública o privada.	Ver propiedades
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.	Ver métodos
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.	
Constructor	Método especial que se ejecuta automáticamente cuando se crea una instancia.	
Método estático	Método que se ejecuta directamente desde la clase, no desde la instancia.	
Inicializador estático	Bloque de código que se ejecuta al definir la clase, sin necesidad de instancia.	

Como vemos, todas estas características se dividen en dos grupos:

- Las **propiedades**: a grandes rasgos, variables dentro de clases
- Los **métodos**: a grandes rasgos, funciones dentro de clases

Un ejemplo sencillo de cada uno:

```
class Animal {
// Propiedades
   name = "changeme";//Garfield;
   type = "changeme";//cat
// Métodos  hablar() {
    return "Odio los lunes."}
}
```

La palabra clave this

Más adelante utilizaremos mucho la palabra clave **this**. Esta es una palabra clave que se utiliza mucho dentro de las clases para hacer referencia al objeto instanciado. Ojo, que hace referencia al **objeto instanciado** y no a la clase:

```
class Animal {
 name;              // Propiedad (variable de clase sin valor definido)

 constructor(name) {
  this.name = name;   // Hacemos referencia a la propiedad name del objeto instanciado
 }
}
```

Observa que la palabra clave **this** no se refiere a la clase **Animal** exactamente, sino a la variable que utilizamos al instanciarla. Es decir, si hacemos un **const pato = new Animal()**, se ejecutaría el constructor y la palabra clave **this** haría referencia a **pato**, por lo que **this.name** estaría haciendo referencia a **pato.name**.

Es importante tener mucho cuidado con la palabra clave **this**, ya que en muchas situaciones creemos que devuelve una referencia al elemento padre que la contiene, pero en su lugar, devolverá el objeto **Window**, ya que se encuentra fuera de una clase o dentro de una función con otro contexto:

```
function hello() {
 return this;
}

hello();                // Window
const object = { hello }     // Metemos la función dentro del objeto
object.hello() === object;   // true
```

En este caso, podemos ver que si ejecutamos la función **hello()** en un contexto global, nos devuelve el padre, es decir, el objeto **Window**. Sin embargo, si metemos la función **hello()** dentro de un objeto , al ejecutar **object.hello()** nos devuelve el padre, es decir, el propio objeto **object**.

Ten cuidado al utilizar **this**. Asegúrate siempre de que **this** tiene el valor que realmente crees que tiene.

Buenas prácticas

Veamos una serie de buenas prácticas a la hora de trabajar con clases, antes de profundizar en sus características.

Clases en ficheros externos

Generalmente, para tener el código lo más organizado posible, las clases se suelen almacenar en ficheros individuales, de forma que cada clase que creamos, debería estar en un fichero con su mismo nombre:

```
// Animal.js

export class Animal {
  /* Contenido de la clase */
}
```

Luego, si queremos crear objetos basados en esta clase, lo habitual suele ser importar el fichero de la clase en cuestión y crear el objeto a partir de la clase. Algo similar al siguiente fragmento de código:

```
// index.js

import { Animal } from "./Animal.js";

const pato = new Animal();
```

Si nuestra aplicación se complica mucho, podríamos comenzar a crear carpetas para organizar mejor aún nuestros ficheros de clases, y por ejemplo, tener la clase **Animal.js** dentro de una carpeta **classes** (*o algo similar*). Esto nos brindaría una mejor experiencia de desarrollo, pero el nombre de las carpetas o su organización ya dependería del desarrollador o del equipo de desarrollo.

Nombrado de miembros

En los próximos capítulos ya profundizaremos en las propiedades y los métodos de una clase, pero una buena práctica para no confundirnos a la hora de utilizarlos, es a la hora de ponerles nombres. Es muy aconsejable que las **propiedades de una clase** tengan nombre de **sustantivos** (*son elementos, valores, ítems...*), mientras que los **métodos de una clase** deberían tener nombre de **verbos** (*son acciones, operaciones, etc...*).

Otro buen consejo, mucho más general, es que intentes escribir código en inglés, ya que eso hará que el código que hagas sea mucho más universal, y sea más fácil de modificar por otras personas.

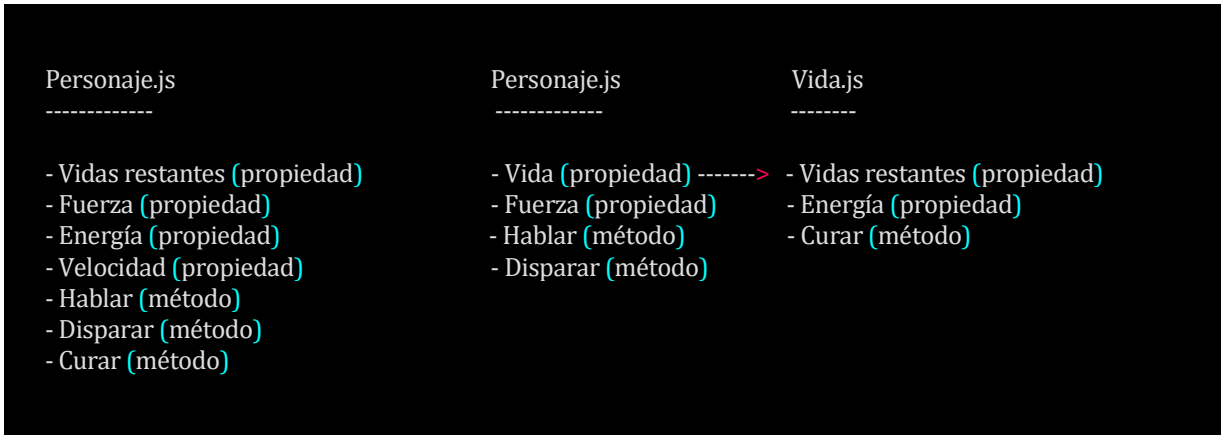
Clases con pocas líneas

Otro consejo interesante a la hora de trabajar con clases sería intentar que las clases se mantengan pequeñas, con **pocas líneas de código**. El número de líneas de código ideal es difícil de saber, pero un buen número, por ejemplo, podría ser entre 100-500 líneas

de código, como menciona la regla max-lines de la herramienta de revisión de código ESLint.

Si descuidamos la cantidad de **líneas de código por fichero** al programar, es muy probable que con el tiempo la clase (*o el fichero*) vaya creciendo en líneas y se vuelva muy difícil de mantener y modificar.

Para evitar esto, lo ideal siempre es mantener, siempre que sea posible, un número bajo de líneas de código, y si la clase se está haciendo muy grande, intentar dividirla en varias clases. Es decir, buscar una serie de criterios para poder **refactorizar** y separar ciertos detalles en una nueva clase:



En este ejemplo, hemos separado en una nueva clase **Vida** en el fichero **Vida.js**, los conceptos **Vidas restantes**, **Energía** y **Curar** ya que tienen relación entre sí (*hacen referencia a la vida del personaje*), de modo que ahora en la clase **Personaje** simplemente tenemos una propiedad que hace referencia a un objeto de esa clase **Vida**, con sus valores particulares.

Esto hará que, si el fichero **Personaje.js** ocupaba demasiadas líneas, consigamos reducirlas, puesto que hemos movido parte de su código a otro fichero, y de paso hemos mejorado mucho nuestro código, ya que ahora está separado en temas más específicos, que son más pequeños y más fáciles de controlar.

PROPIEDADES DE CLASE

Hasta ahora, hemos hablado de **Programación orientada a objetos** (*POO*) y del concepto de **clase** dentro de este estilo de programación. Sin embargo, tenemos que profundizar en los **miembros de clase**, que a grandes rasgos son **propiedades** o **métodos**. En este artículo vamos a explicar las **propiedades** de clase.

¿Qué es una propiedad de clase?

Las **clases**, siendo estructuras para guardar y almacenar información, tienen unas **variables** que viven dentro de la clase. Esta información (*también llamada estado*) se denomina **propiedad** o **propiedad de clase** y desde ECMAScript 2020 para crearlas se hace de la siguiente forma:

```
class Personaje {
 name;            // Propiedad sin definir (undefined)
 type = "Player";    // Propiedad definida
 lifes = 5;         // Propiedad definida con 5 vidas restantes
 energy = 10;       // Propiedad definida con 10 puntos de energía
}
```

Tradicionalmente en Javascript, las **propiedades** acostumbraban a definirse a través del constructor, mediante la palabra clave **this**, por lo que es muy probable que también te las encuentres declaradas de esta forma, sin necesidad de declararlas fuera del constructor:

```
class Personaje {
 constructor() {
  this.name;            // Propiedad sin definir (undefined)
  this.type = "Player";    // Propiedad definida
  this.lifes = 5;         // Propiedad definida con 5 vidas restantes
  this.energy = 10;       // Propiedad definida con 10 puntos de energía
 }
}
```

Puesto que se trata de **propiedades de clase** y el **constructor()** es un método que se ejecuta cuando se crea el objeto (*instancia de clase*), ambas son equivalentes, ya que al crear un objeto se ejecutará el **constructor** y se crearán esas propiedades.

A la hora de utilizarlas, simplemente accedemos a ellas de la misma forma que vimos en el último ejemplo, haciendo uso de la palabra clave **this**. Veamos un ejemplo un poco más elaborado, utilizando **propiedades** y **métodos**:

```
class Personaje {
  name;              // Propiedad sin definir (undefined)
  type = "Player";     // Propiedad definida
  lifes = 5;          // Propiedad definida con 5 vidas restantes
  energy = 10;        // Propiedad definida con 10 puntos de energía

  constructor(name) {
    this.name = name;  // Modificamos el valor de la propiedad name
    console.log(`¡Bienvenido/a, ${this.name}!`);  // Accedemos al valor actual de la propiedad name
  }
}

const mario = new Personaje("Mario");  // '¡Bienvenido/a, Mario!'
```

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.

Visibilidad de propiedades

Observa que, las **propiedades de clase** siempre van a tener una visibilidad específica, que puede ser **pública** (*por defecto*) o **privada**. En el primer caso, las propiedades pueden ser leídas o modificadas tanto desde dentro de la clase como desde fuera, en el segundo caso, sólo pueden ser leídas o modificadas desde el interior de la clase.

Nombre	Sintaxis	Descripción
Propiedad pública	`name` o `this.name`	Se puede acceder a la propiedad desde dentro y fuera de la clase.
Propiedad privada	`#name` o `this.#name`	Se puede acceder a la propiedad **sólo** desde dentro de la clase.

Vamos a echar un vistazo a un ejemplo para entenderlo mejor.

Propiedades públicas

Por defecto, las propiedades en las clases son públicas. Observa que siempre vamos a poder acceder a las propiedades desde el constructor u otros métodos (*dentro de la clase*), ya sean propiedades públicas o privadas:

```
class Personaje {
  name;
  energy = 10;

  constructor(name) {
```

```
    this.name = name;
  }
}

const mario = new Personaje("Mario");    // { name: "Mario", energy: 10 }
mario.name;                              // "Mario" (se puede acceder desde fuera)
mario.name = "Evil Mario";
mario.name;                              // "Evil Mario" (se ha modificado desde fuera)
```

Observa también que en las últimas líneas, accedemos a la propiedad **name** desde fuera de la clase y la modificamos. Esto ocurre porque es una propiedad pública, y es posible hacerlo.

Propiedades privadas

A partir de la versión **ECMAScript** , se introduce la posibilidad de crear propiedades de clase privadas. Por defecto, todas las propiedades y métodos son públicos por defecto, sin embargo, si añadimos el carácter **#** justo antes del nombre de la propiedad, se tratará de una **propiedad privada**:

```
class Personaje {
  #name;
  energy = 10;

  constructor(name) {
    this.#name = name;
  }
}

const mario = new Personaje("Mario");    // { name: "Mario", energy: 10 }

// Es incorrecto, el nombre correcto de la propiedad es #name
mario.name; // undefined

// Los dos siguientes dan el mismo error (no se puede acceder a la propiedad privada)
// Uncaught SyntaxError: Private field '#name' must be declared in an enclosing class
mario.#name;
mario.#name = "Evil Mario";

// Lo siguiente funcionará, pero ha creado otra propiedad 'name' que no es la misma que '#name'
mario.name = "Evil Mario";
```

Como se puede ver, las propiedades precedidas del carácter **#** son automáticamente privadas y sólo se podrá acceder a ellas desde un método de clase, ya que si se hace desde fuera obtendremos un error similar al siguiente:

```
Uncaught SyntaxError: Private field '#name' must be declared in an enclosing class
```

Sin embargo, si se llama a un método público, que a su vez accede a la propiedad privada mediante **this.#name** todo funcionará correctamente, ya que ese método público si es accesible desde fuera de la clase y la propiedad privada si es accesible desde dentro de la clase.

Ámbitos de propiedades de 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 propiedades dentro de un método con **let** o **const**, estos elementos existirán sólo en el método en cuestión. Además, no serán accesibles desde fuera del método:

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

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

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

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

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

En segundo lugar, tenemos el **ámbito de clase**. Si creamos propiedades de las dos formas que vimos al principio del artículo:

- Precedidas por **this.** desde dentro del constructor
- Al inicio de la clase, fuera del constructor

En cualquiera de estos dos casos, las propiedades tendrán alcance en toda la clase, por lo que podremos acceder a ellas tanto desde el constructor, como desde otros métodos de la clase:

```
class Personaje {
 name = "Manz";        // ES2020+

 constructor() {
  this.name = "Manz";   // ES2015+
  console.log("Constructor: " + this.name);
 }
```

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

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

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

Recuerda que si quieres evitar que estas propiedades de clase se puedan modificar desde fuera de la clase, añade el # antes del nombre de la propiedad al declararla. De esta forma serán propiedades privadas, y sólo se podrá acceder a ellas desde el interior de los métodos de la clase.

Propiedades computadas

En algunos casos nos puede interesar utilizar lo que se llaman **propiedades computadas**. Las **propiedades computadas** son un tipo de propiedad especial que se declara como una función, y que se ejecuta cuando accedemos a la propiedad con dicho nombre. Hay dos tipos de propiedades computadas, los **getters** y los **setters**.

Propiedades get (getters)

Veamos el primer caso, la propiedad computada **get** o también llamada **getter**. Para definirla, simplemente añadimos la palabra clave **get** antes del nombre de la función. De hecho, se define exactamente igual que una función:

```
class Personaje {
 name;
 energy;

 constructor(name, energy = 10) {
  this.name = name;
  this.energy = energy;
 }

 get status() {
  return '⭐'.repeat(this.energy);
 }
}

const mario = new Personaje("Mario");
mario.energy   // 10
mario.status   // '⭐⭐⭐⭐⭐⭐⭐⭐⭐⭐'
```

Observa que aunque la definimos como una función **status()**, luego accedemos a ella como una propiedad **mario.status**. Por eso se llama **propiedad** computada. La idea de

este tipo de propiedades, es permitir pequeñas modificaciones sobre propiedades ya existentes (*en nuestro caso, energy*). En lugar de devolver el valor numérico, devolvemos el número de estrellas que representa la vida del personaje.

Ten mucho cuidado con acceder a la misma propiedad definida desde dentro del **get**. Si dentro del **get status()** accedes a **this.status**, se produciría un bucle infinito que podría bloquear el navegador.

Propiedades set (setters)

De la misma forma que podemos crear un **get** para obtener un valor, podemos utilizar un **set** para establecerlo. La idea es exactamente la misma, pero para modificar el valor. En este caso, el ejemplo no es tan didáctico, pero vamos a dar la funcionalidad inversa. Si establecemos un número de estrellas a **status**, las cuenta y asigna el número a **energy**:

```
class Personaje {
 name;
 energy;

 constructor(name, energy = 10) {
  this.name = name;
  this.energy = energy;
 }

 get status() {
  return '⭐'.repeat(this.energy);
 }

 set status(stars) {
  this.energy = stars.length;
 }
}

const mario = new Personaje("Mario");
mario.energy   // 10
mario.status = '⭐⭐⭐'
mario.energy   // 3
mario.status   // '⭐⭐⭐'
```

Observa que ahora la "magia" está en el **set status(stars)**. Se comporta como una función, y al asignar tres estrellas a **mario.status**, automágicamente se ha cambiado el valor de **mario.energy**. Estas **propiedades computadas** nos pueden venir muy bien cuando queramos modificar ligeramente ciertos elementos de una forma automática y organizada.

MÉTODOS DE CLASE

Simplificando mucho, un **método** es el nombre que recibe una **función** que existe dentro de una clase. Se utilizan para englobar comportamientos o funcionalidades relacionadas en conjunto con la clase y mediante las cuales podemos segmentar y separar en bloques de código.

Por ejemplo, en el siguiente fragmento de código definimos una constante **text** que contiene el **"Manz"**. En la siguiente línea, ejecutamos el método **.repeat()**, que es una función que pertenece a todos los objetos que son definidos como , y que simplemente repite el texto el número de veces que le pasamos por parámetro:

```
const text = "Manz";
text.repeat(3);    // "ManzManzManz"
```

Cada variable de un determinado tipo de dato, tiene métodos asociados a dicho tipo de dato, los puedes ver muy claramente en la CheatSheet de Javascript. Los objetos de tipo tienen sus propios métodos, los objetos de tipo tienen los suyos, etc...

¿Qué es un método?

En nuestro caso, hablamos de **métodos** cuando nos referimos a **funciones** que existen en el interior de una clase. Observa el siguiente ejemplo, donde tenemos una función independiente:

```
function hablar() {
 return "Hola";
}
```

Esta función no está asociada a ningún otro elemento. Simplemente existe en el ámbito global de nuestro programa. Vamos ahora a convertirla en un método de la clase **Animal** (*y por lo tanto, devolver en ella algo más acorde*):

```
// Forma corta (recomendado)
class Animal {
 hablar() {
   return "Cuak";
 }
}
```

Observa que esto no es más una forma de incluir la función anterior dentro de nuestra clase **Animal**. Realmente, podríamos también hacerlo de la siguiente forma, sin embargo, la anterior es mucho más compacta y sencilla, por lo que es la que más se suele utilizar, pero quizás con esta te resulte más claro lo que se está haciendo dentro de la clase:

```
// Forma larga
class Animal {
 hablar = function() {
  return "Cuak";
 }
}
```

Una vez declarado el método **hablar()** dentro de la clase **Animal**, podemos instanciar el objeto mediante un **new Animal()** y tener ese método disponible. Ten en cuenta que podemos crear varias variables de tipo **Animal** y serán totalmente independientes cada una:

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

// Creación de otra instancia u objeto (donald)
const donald = new Animal();
donald.hablar(); // 'Cuak'
```

Observa que el método **hablar()** existe tanto en los objetos **pato** como **donald** porque ambas son de tipo **Animal**. Al igual que ocurre con una función normal, se le pueden pasar varios parámetros al método y trabajar con ellos como venimos haciendo normalmente con las funciones.

Constructor de clase

Se le llama **constructor** a un método de clase especial que se ejecuta automáticamente cuando se hace un **new** de dicha clase (*al instanciar el objeto*). 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:

```
class Animal {
 constructor() {
  console.log("Ha nacido un pato. 🦆");
 }
```

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

// Creación de instancia/objeto
const pato = new Animal();  // 'Ha nacido un pato' (Se ha ejecutado implicitamente el constructor)
pato.hablar();           // 'Cuak' (Se ha ejecutado explícitamente el método hablar)
```

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.

¿Qué es un método estático?

En el caso anterior, al utilizar un método como por ejemplo **hablar()**, debemos crear el objeto basado en la clase haciendo un **new Animal()**. Es lo que se denomina **crear un objeto**, crear una **instancia de clase** o **instanciar un objeto**.

Sin embargo, nos podría interesar crear **métodos estáticos** en una clase, ya que este tipo de métodos **no requieren crear una instancia**, sino que se pueden ejecutar directamente sobre la clase:

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

 hablar() {
  return "Cuak";
 }
}

Animal.despedirse();      // Método estático (no requiere instancia): 'Adiós'
Animal.hablar();          // Uncaught TypeError: Animal.hablar is not a function

const pato = new Animal();  // Creamos una instancia

pato.despedirse();        // Uncaught TypeError: pato.despedirse is not a function
pato.hablar();           // Método no estático (requiere instancia): 'Cuak'
```

Como veremos más adelante, lo habitual suele ser utilizar métodos normales (*no estáticos*), ya que normalmente nos suele interesar crear múltiples objetos y **guardar información** (*estado*) en cada uno de ellos, y para eso necesitaríamos 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.

Los **métodos estáticos** se suelen utilizar para crear funciones de apoyo que realicen tareas genéricas que no necesiten estado de la clase, pero siguen estando relacionadas con la clase y no queremos mantenerlas separadas.

Inicialización estática

Una reciente característica denominada **Class static initialization blocks** nos permite ejecutar un bloque de código de forma muy similar a una especie de constructor estático.

La diferencia radica en que, mientras el constructor se ejecuta cuando se crea el objeto (*se crea una instancia de clase*), el bloque estático **static {}** se ejecuta nada más declarar la clase (*antes de la instancia*), por lo que puede ser realmente útil para realizar tareas de inicialización donde no necesitas la instancia del objeto, o previas al constructor:

```
class Animal {
 static {
  console.log("Bloque inicializado");
 }

 constructor() {
  console.log("Constructor ejecutado");
 }
}
// <-- Aquí nos aparece "Bloque inicializado"

const pato = new Animal();  // <-- Tras el new Animal(), aparece "Constructor ejecutado"
```

Ten en cuenta que desde el bloque **static {}** tendrás acceso a propiedades estáticas, pero no a propiedades de clase (*necesitan instancia de clase*). Por otro lado, desde el **constructor()** podrás acceder tanto a las propiedades de clase como a las propiedades estáticas.

Visibilidad de métodos

Al igual que ocurre con las **propiedades de clase**, los métodos de una clase tienen una visibilidad específica que por defecto es **pública**. Esto es, los métodos son accesibles tanto desde fuera de la clase como desde dentro.

Nombre	Sintaxis	Descripción
Método público	**name()** o **this.name()**	Se puede acceder al método desde dentro y fuera de la clase.

Nombre	Sintaxis	Descripción
Método privado	**#name()** o **this.#name()**	Se puede acceder al método **sólo** desde dentro de la clase.

Veamos un ejemplo de cada caso.

Métodos públicos

Por norma general, los métodos de una clase son **públicos**, por lo que podemos acceder tanto desde dentro de la clase como desde fuera. Observa que desde el **constructor** estamos accediendo a **hablar()** desde dentro de la clase y al crear el objeto, se llamará a ese método:

```
class Personaje {
  name = "Mario";

  constructor() {
    this.hablar();
  }

  hablar() {
    console.log("It's me, Mario!");
  }
}

const mario = new Personaje();     // It's me, Mario! (se ha accedido a hablar() desde dentro de la clase)
mario.hablar();                    // It's me, Mario! (se ha accedido a hablar() desde fuera de la clase)
```

Por otro lado, al llamar a **mario.hablar()** se puede ver que se permite acceder desde fuera de la clase.

Métodos privados

¿Qué es lo que ocurre si definimos el método **hablar()** como un método privado? Para ello, simplemente le añadimos el símbolo # antes del nombre, asegurándonos también de incluirlo en las llamadas al método. Quedaría algo así:

```
class Personaje {
  name = "Mario";

  constructor() {
    this.#hablar();
  }

  #hablar() {
    console.log("It's me, Mario!");
  }
```

```
}

const mario = new Personaje();     // It's me, Mario! (se ha accedido a #hablar() desde dentro de la clase)

// Da error, no se permite acceder a un método privado desde fuera de la clase
// Uncaught SyntaxError: Private field '#hablar' must be declared in an enclosing class
mario.#hablar();

// Da error, el método hablar() no existe, ya que el nombre del método es #hablar()
// Uncaught TypeError: mario.hablar is not a function
mario.hablar();
```

Como se puede contemplar, en el caso de definir el método privado, no es posible ejecutarlo desde fuera de una clase, salvo que lo hagamos a través de un método público que llame internamente al método privado.