Índice

[**¿Qué es class?** 1](#_Toc128494959)

[**Buenas prácticas** 1](#_Toc128494960)

[**Métodos** 2](#_Toc128494961)

[**Acceso público** 2](#_Toc128494962)

[**Public** 2](#_Toc128494963)

[**Desventajas del alcance público** 3](#_Toc128494964)

[**Acceso privado** 3](#_Toc128494965)

[**Private** 3](#_Toc128494966)

[**Private class fields (#)** 3](#_Toc128494967)

[**Constructor** 4](#_Toc128494968)

[**Valores por defecto** 4](#_Toc128494969)

[**Getters** 5](#_Toc128494970)

[**Setters** 6](#_Toc128494971)

[**Herencia** 6](#_Toc128494972)

[**Super** 7](#_Toc128494973)

[**Acceso protegido** 8](#_Toc128494974)

[**¿Qué es protected?** 8](#_Toc128494975)

[**Static** 9](#_Toc128494976)

[**Interfaces** 10](#_Toc128494977)

[**Implements** 10](#_Toc128494978)

[**Clases abstractas** 11](#_Toc128494979)

[**Singleton: constructor privado** 11](#_Toc128494980)

[**Promesas** 13](#_Toc128494981)

[**Asincronismo y TypeScript** 13](#_Toc128494982)

[**Tipado y promesas** 13](#_Toc128494983)

[**Axios** 13](#_Toc128494984)

[**Tipando respuestas HTTP** 13](#_Toc128494985)

[**Quicktype** 13](#_Toc128494986)

[**Tipado en Axios** 14](#_Toc128494987)

[**Aserción AS** 14](#_Toc128494988)

[**Generics** 14](#_Toc128494989)

[**Decoradores** 15](#_Toc128494990)

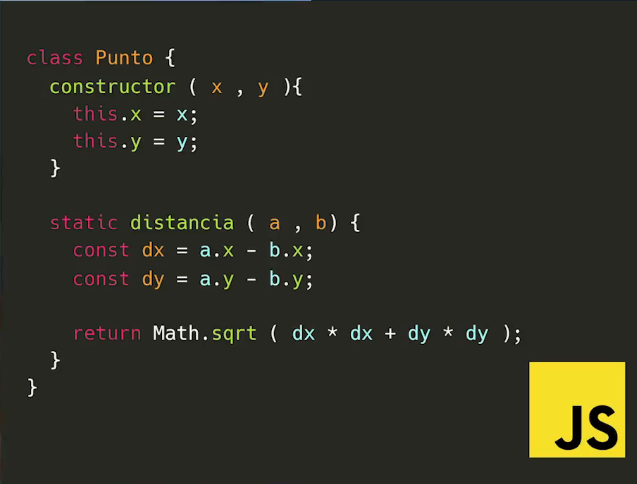
[**Importante** 16](#_Toc128494991)

# **¿Qué es class?**

* JavaScript es un lenguaje orientado a objetos, pero está basado en prototipos.
* Las clases son una sintaxis más amigable para trabajar con prototipos, las clases internamente son prototipos.
* La utilizamos para construir instancias de objetos, **es como un** **molde para crear cosas**.
* Los prototipos también se diferencian con la POO porque no solamente crean objetos mediante **instancias**, sino también mediante la **clonación**.

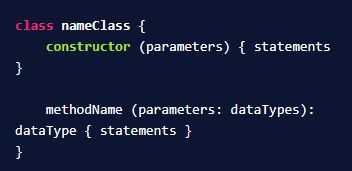
## **Buenas prácticas**

Es una buena práctica inicializar las **propiedades** que coloquemos en nuestra clase, ya sea por **fuera** o **dentro** de nuestro constructor.



# **Métodos**

Describen el **comportamiento** de los **objetos**.

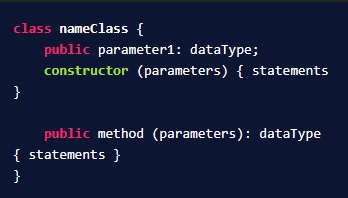


# **Acceso público**

Podemos acceder a las **propiedades** y **métodos** del objeto desde el exterior.

## **Public**

Por defecto, todas las **propiedades** y **métodos** son **públicos**, pero si queremos aclarar en nuestro código el alcance de los mismos podemos utilizar la palabra reservada **public**.



## **Desventajas del alcance público**

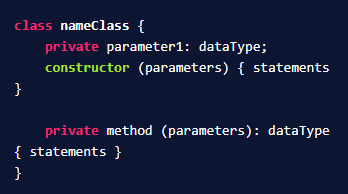
La desventaja que tenemos al tener una propiedad o método con alcance público es que estás pueden ser modificables desde fuera de la clase.

# **Acceso privado**

Podemos proteger nuestras **propiedades** y **métodos** para que nos sean accedidos y modificados por fuera de clase.

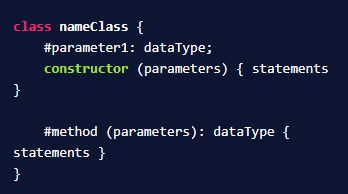
## **Private**

Con esta palabra reservada **private** estamos restringiendo el acceso de nuestros parámetros y métodos, solo podrán ser accedidos o modificados **dentro** de la clase.



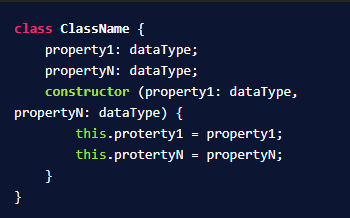
## **Private class fields (#)**

A desde hace muy poco JavaScript tiene incorporado la capacidad de poder limitar el alcance de los métodos y propiedades utilizando **#**. Funciona igual que **private**, solo que **#** es nativo de JavaScript y realmente limita el alcance.



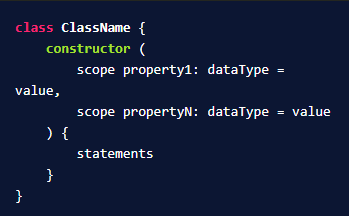
# **Constructor**

Es en el que construimos la **instancia** de una clase y mandamos los parámetros por defecto por el cual queremos que se inicialice esa instancia a un objeto.



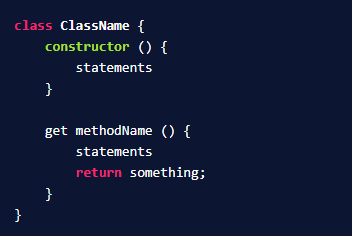
## **Valores por defecto**

Al tener valores por defecto, al momento de crear una instancia de esa clase ya no es obligatorio pasar ese parámetro que tiene un valor por defecto.



# **Getters**

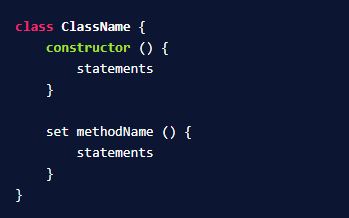
* Con **get** podemos acceder a un valor de una **propiedad** con un alcance **privado**, podemos obtener los datos de esa propiedad, pero **no modificarlos** o **mutarlos**.
* Lo podemos acceder como si fuese una propiedad más de la instancia.
* Por definición, todos los **getters** tienen que retornar algo, pero nunca **void**.



# **Setters**

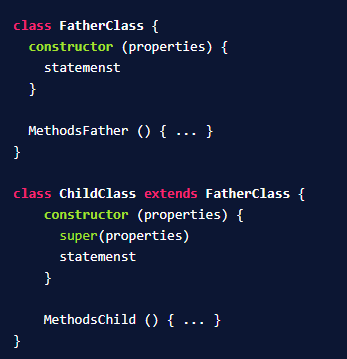
Es parecido a un **get**, solo que este no retorna nada, es un método **void**, pero no hace falta colocarle lo que retorna, ya que va a dar un error.

A **set** lo podemos usar para tener reglas de modificación para nuestros parámetros.



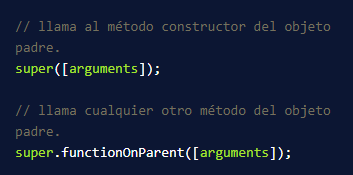
# **Herencia**

Las **propiedades** y **métodos** de una clase son heredadas por otra para evitar la redundancia de código en nuestro proyecto.



## **Super**

Es una palabra reservada para llamar las **propiedades** y **métodos** de la **clase** **padre**.

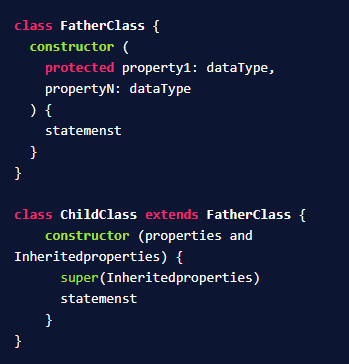


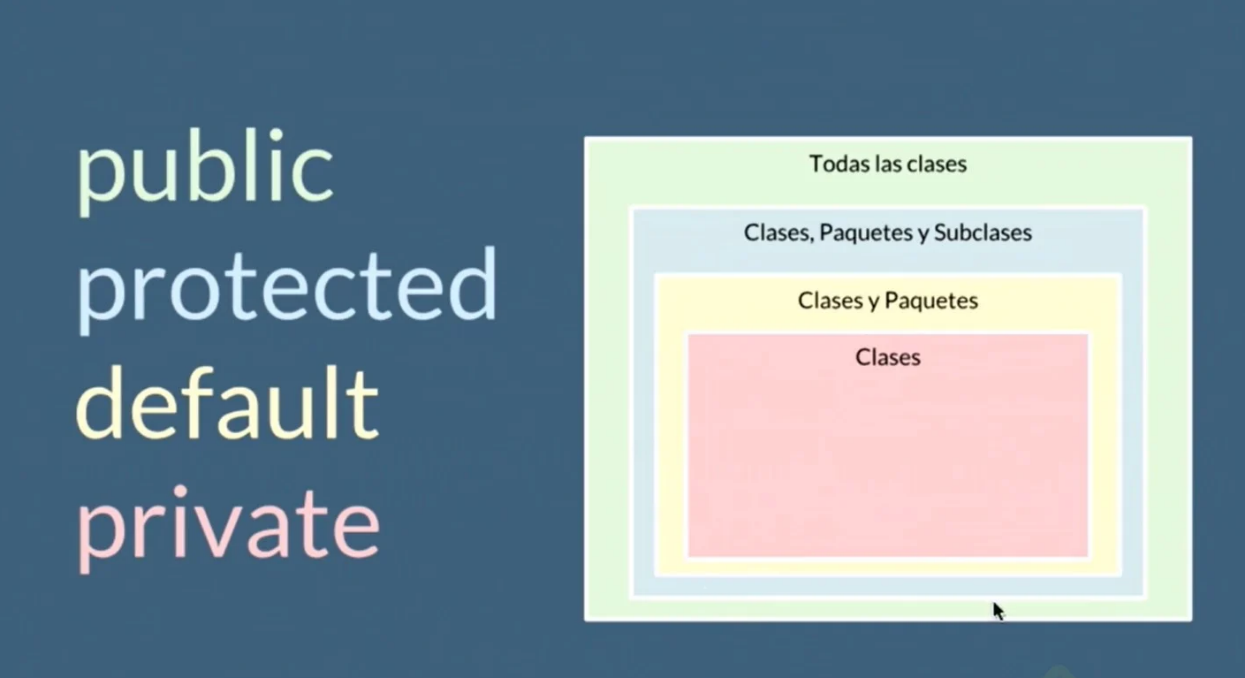
# **Acceso protegido**

## **¿Qué es protected?**

Básicamente, es la forma de heredar propiedades y métodos.

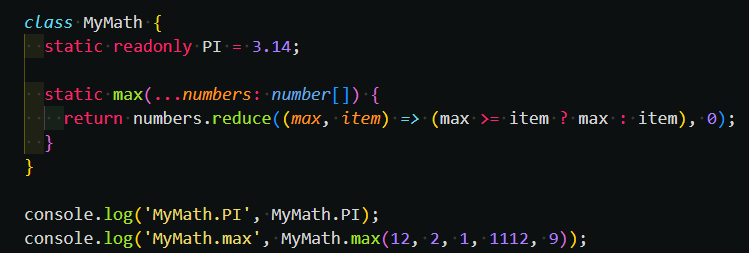
Es similar a como funciona una propiedad o método privado, pero nos permite realizar herencia.





# **Static**

Los métodos y propiedades estáticas pueden ser accedidos sin la necesitad de crear instancias de clases.



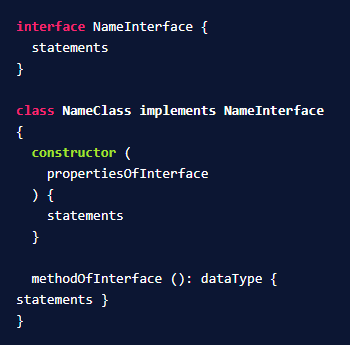
# **Interfaces**

* Las interfaces en clases funcionan como contratos, nos permiten tipar tanto los **atributos** como los **métodos**.
* Aquí no podemos utilizar encapsulamiento, **por defecto todos los atributos y métodos son públicos**.
* Todos los métodos y atributos son **public**.

## **Implements**

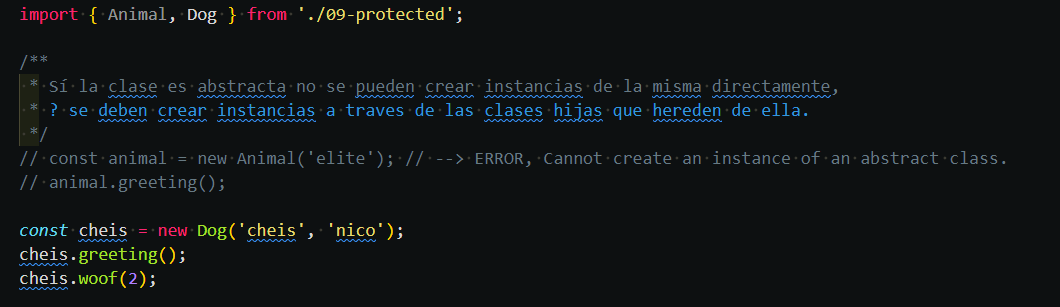
Tiene la misma sintaxis cuando aplicamos herencias en las clases, pero **implements** no es una herencia, **es un contrato que nos dice que nuestra clase tiene que cumplir con un estándar que es la interface**.

Este contrato es tanto como para los parámetros como para los métodos.



# **Clases abstractas**

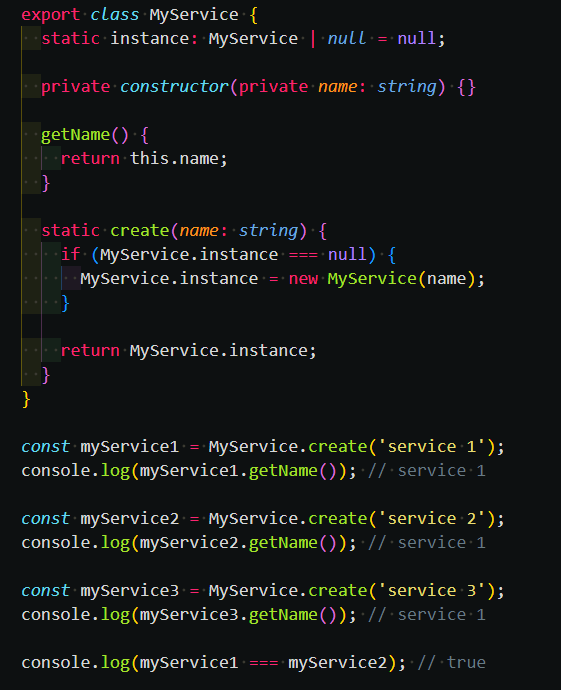
Las clases abstractas no permiten crear instancias de objetos directamente, sino que se deben crear en sus clases hijas.



Este tipo de clases es como una clase común y corriente, pero la diferencia es que no se pueden crear instancias directas.

# **Singleton: constructor privado**

**Singleton** nos previene crear **una única instancia de una clase**. Esto nos ayuda a ahorrar uso de memoria.



# **Promesas**

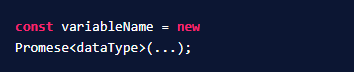
## **Asincronismo y TypeScript**

Podemos correr promesas y código asíncrono con TypeScript de la misma forma que lo haríamos con JavaScript.

## **Tipado y promesas**

En el caso de no colocar el tipo de dato de retorno de nuestra función asíncrona, TypeScript supondrá que es una promesa, pero no sabe el tipo de dato que retorna esa promesa.

Podemos tipar a una promesa con genéricos.



## **Axios**

Es una forma interesante de hacer **request** y conectarnos a servicios web, nos va a servir para hacer peticiones desde el **frontend** como en el **backend**.

# **Tipando respuestas HTTP**

## [**Quicktype**](https://app.quicktype.io/)

Esta página transforma nuestro archivo JSON en el tipado que le corresponde al mismo para poder tipar nuestras repuestas HTTP.

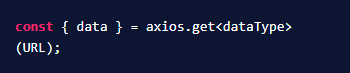
Tipado en funciones asíncronas con peticiones HTTP

Es importante poder tipar tanto nuestras funciones, ya sea inferidamente o explícitamente, como las variables que manejamos internamente dentro de ella.

## **Tipado en Axios**

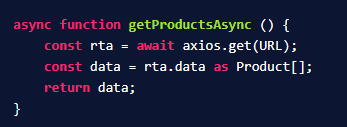
**Aserción angle bracket**

Axios nos permite tipar las peticiones que hacemos de la siguiente manera.



## **Aserción AS**

También podemos realizar el tipado por medio de aserción de tipos con **as**, pero lo ideal es hacerlo con el primer método.



# **Generics**

Por convención, se utiliza el nombre **T** (Type en inglés) para dar a conocer que es un generics, es muy común ver este tipo de nombre en otros repositorios, documentación y también es usado en otros lenguajes como java y C#.

Otros nombres utilizados en los generics son:

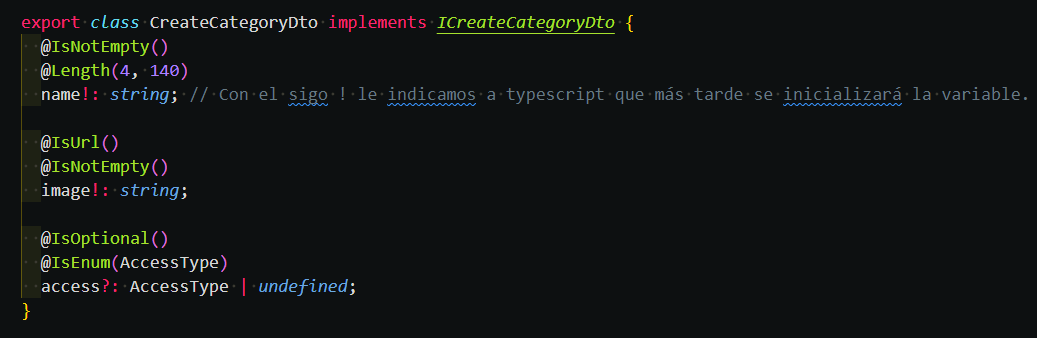
* **E** - Elemento (elementos en una colección.)
* **K** – Llave.
* **N** – Número.
* **V** – Valor.

Esto es aplicable para:

* Class.
* Method.
* Functions.

# **Decoradores**

Agregan validaciones extras a **clases**, **propiedades**, **métodos**, **parámetros** y **accesos**.



En el archivo **tsconfig.ts** se debe habilitar la línea **"experimentalDecorators": true** ya que viene deshabilitada por defecto, de lo contrario dará error.

**NOTA**: Esto es un feature experimental, aún no son nativos o compatible en JavaScript.

## **Importante**

* A diferencia de TypeScript, los decoradores **no** nos van a mostrar los errores de forma inmediata.
* Dan feedback hasta el **runtime** (tempo de ejecución).
* Agregan validaciones y comportamientos extra.

