# 7. Getters

Les comparto mis apuntes. 😄

## Get

Con get podemos acceder a 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.

## Sintaxis

**class** **ClassName** {

**constructor** () {

statements

}

get methodName () {

statements

**return** something;

}

}

## Una aclaración

En lo particular, prefiero utilizar el private class fields (#) para poder declarar propiedades privadas, ya que es nativo de JavaScript y además que realmente hace un encapsulamiento de las propiedades al momento de llevar el código a producción. Cabe agregar que con este nos podemos evitar realizar el uso de underscore (\_) y también tener problemas con los nombres de los métodos getters y setters.

La estructura del código depende del proyecto y del equipo, pero esa es forma preferida de emplear lo mejor de ambos mundos, tanto de JavaScript como de TypeScript.

## Código de la clase

Código adaptado por mí.

**type** PartOfTheCalendar = 'day' | 'month'| 'year';

**export** **class** **MyDate** {

#day: number;

**constructor** (

**public** year: number = 1993,

**public** month: number = 7,

day: number = 1

) {

this.#day = day;

}

printFormat (): string {

**const** { year } = this;

**const** day = this.#addPadding(this.#day);

**const** month = this.#addPadding(this.month);

**return** `${day}/${month}/${year}`;

}

add (amount: number, type: PartOfTheCalendar): void {

**if** (**type** === 'day') this.#day += amount;

**if** (**type** === 'month') this.month += amount;

**if** (**type** === 'year') this. year += amount;

}

#addPadding (value: number): string {

**if** (value < 10) **return** `0${value}`;

**return** `${value}`;

}

get day (): number {

// Code

**return** this.#day;

}

get isLeapYear (): boolean {

**const** { year } = this;

**if** (year % 4 === 0) **return** **false**;

**if** (year % 100 === 0) **return** **true**;

**return** year % 4 === 0;

}

// get myReturn () {

// } // Error:

}

**const** myDate = **new** MyDate(2020);

**const** myDate2 = **new** MyDate(2000);

**const** myDate3 = **new** MyDate(2001);

**const** myDate4 = **new** MyDate(2004);

myDate.day;

console.log('My Date:',myDate.day, myDate.isLeapYear);

console.log('My Date 2:',myDate2.day, myDate2.isLeapYear);

console.log('My Date 3:',myDate3.day, myDate3.isLeapYear);

console.log('My Date 4:',myDate4.day, myDate4.isLeapYear);

# 8. Setters

<https://platzi.com/clases/2880-typescript-poo/47259-setters/>

Les comparto mis apuntes. 😄

## Set

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 error.

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

## Sintaxis

**class** **ClassName** {

**constructor** () {

statements

}

set methodName () {

statements

}

}

# 9. Herencia

<https://platzi.com/clases/2880-typescript-poo/47260-herencia/>

## ¿Qué es?

Como su palabra lo dice, las propiedades y métodos de una clase son heredadas por otra para evitar la redundancia de código en nuestro proyector.

## Sintaxis

**class** **FatherClass** {

**constructor** (properties) {

statemenst

}

MethodsFather () { ... }

}

**class** **ChildClass** **extends** **FatherClass** {

**constructor** (properties) {

super(properties)

statemenst

}

MethodsChild () { ... }

}

## Super

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

// llama al método constructor del objeto padre.

super([arguments]);

// llama cualquier otro método del objeto padre.

super.functionOnParent([arguments]);

Al usar esta palabra reservada en el constructor de la clase hija, estamos invocando al constructor de la clase padre.

[Más información](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/super).

## Código de la clase

**export** **class** **Animal** {

**constructor**(**public** name: string) {}

**move**() {

console.log("Moving along!");

}

**gretting**() {

**return** `Hello, I'm ${this.name}`;

}

}

**export** **class** **Dog** **extends** **Animal** {

**constructor**(**public** owener: string, name: string) {

super(name);

}

woof(times: number): void {

**for** (**let** i = 0; i < times; i++) {

console.log("Woof!");

}

}

}

**const** fifi = **new** Animal("Fifi");

fifi.move();

console.log(fifi.gretting());

**const** cheis = **new** Dog("Cheis", "Galeed");

cheis.move();

console.log(cheis.gretting());

cheis.woof(5);

# 10. Acceso protegido

<https://platzi.com/clases/2880-typescript-poo/47261-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.

## Sintaxis

**class** **FatherClass** {

**constructor** (

**protected** property1: dataType,

propertyN: dataType

) {

statemenst

}

}

**class** **ChildClass** **extends** **FatherClass** {

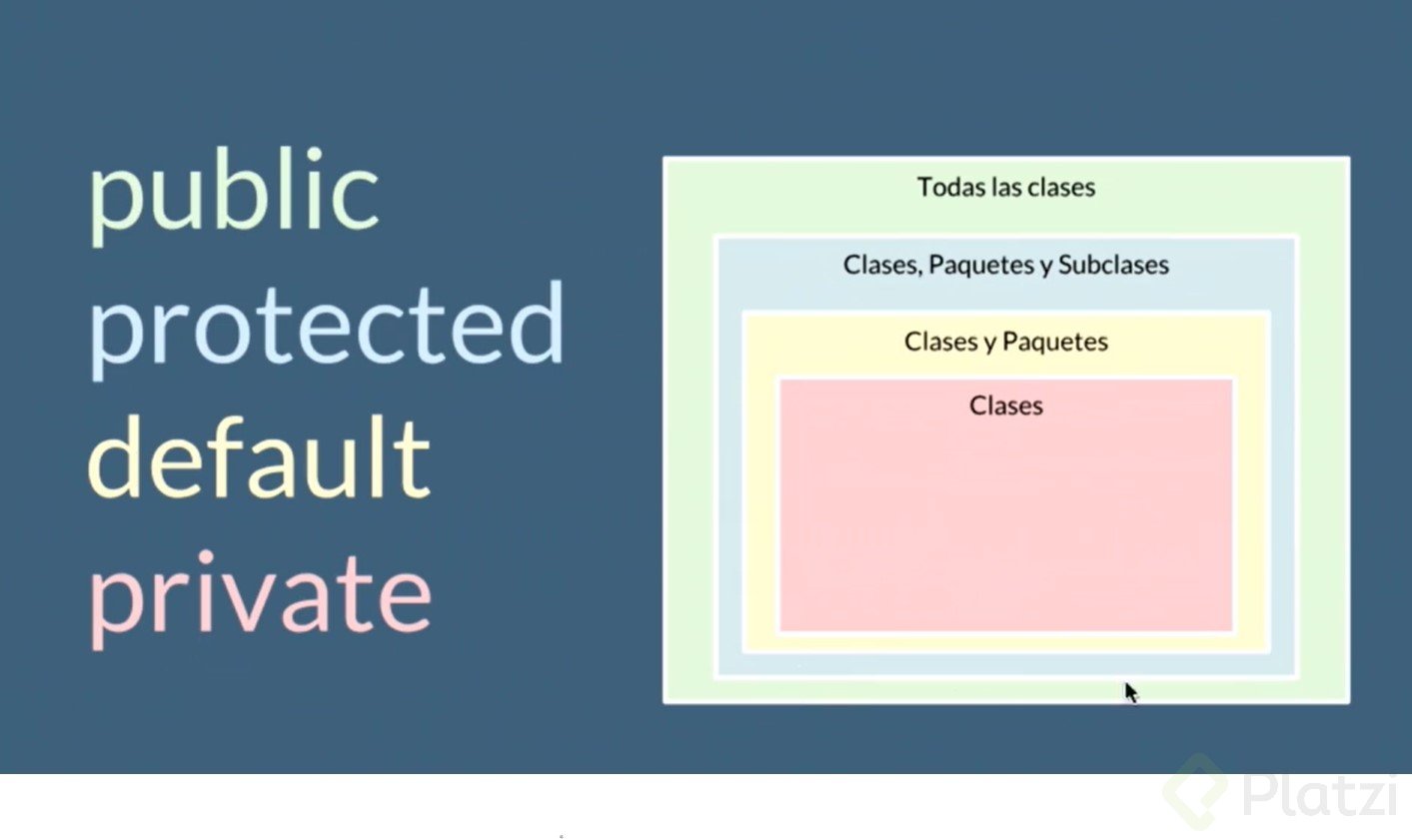
**constructor** (properties and Inheritedproperties) {

super(Inheritedproperties)

statemenst

}

}

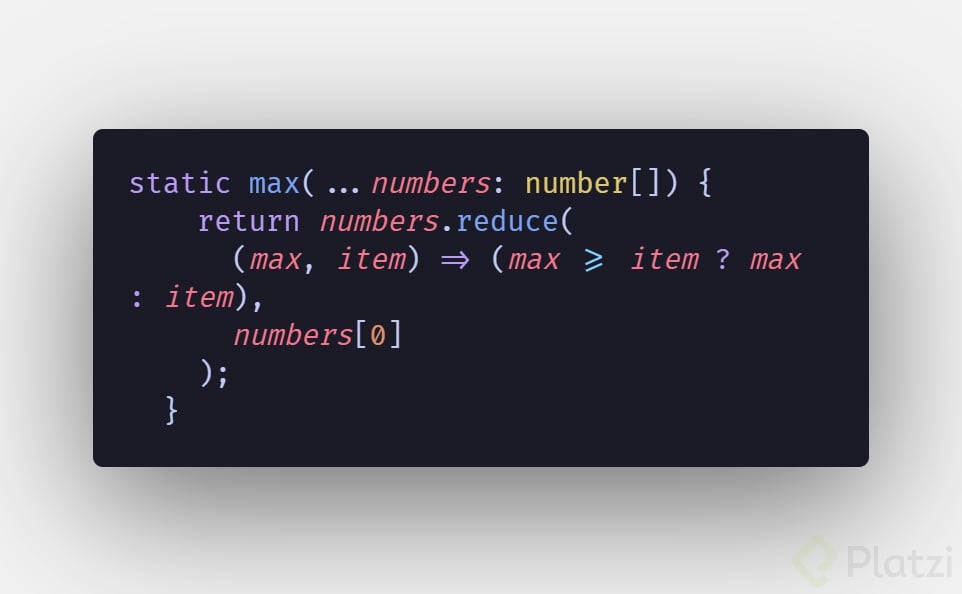


# 11. Static

<https://platzi.com/clases/2880-typescript-poo/47262-static/>

solo no inicializar el max

numbers.reduce((max, **item**) => max > **item** ? max : **item**);



La solución es simplemente no pasar el segundo parametro del reduce.

El primer parametro del reduce es el callback donde se ejecuta la operación del reduce y el segundo es un parametro opcional que indica el valor inicial del acumulador, por defecto, el valor inicial del acumulador es la primera posición del array y curr empezaría desde la siguiente posición.

**static** **max**(...numbers: number[]) {

**return** numbers.reduce((acc, curr) => acc > curr ? acc : curr);

}

# 12. Interfaces

<https://platzi.com/clases/2880-typescript-poo/47263-interfaces/>

## Interfaces en clases

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.

## 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.

## Sintaxis

**interface** NameInterface {

statements

}

**class** **NameClass** **implements** **NameInterface** {

**constructor** (

propertiesOfInterface

) {

statements

}

methodOfInterface (): dataType { statements }

}

## Código de la clase

**export** **interface** Driver {

database: string;

password: string;

port: number;

conect (): void;

disconnect (): void;

isConeected (name: string): boolean;

}

// const driver: Driver = {

// database: '',

// password: '',

// port: 23,

// }

**class** **PostresDriver** **implements** **Driver**{

**constructor** (

**public** database: string,

**public** password: string,

**public** port: number,

) {

}

conect (): void {}

disconnect (): void {}

isConeected (name: string): boolean {

**return** **true**;

}

}

**class** **Oracle** **implements** **Driver** {

**constructor** (

**public** database: string,

**public** password: string,

**public** port: number,

) {

}

conect (): void {}

disconnect (): void {}

isConeected (name: string): boolean {

**return** **true**;

}

}

# 13. Clases abstractas

<https://platzi.com/clases/2880-typescript-poo/47264-clases-abstractas/>

En conclusión, cada vez que implementamos una clase abstracta, no podemos crear instancias de esa clase, si no solo de las clases hijas.

**abstract** **class** **Pet** {} // no instancias de esta clase

**class** **Dog** **extends** **Pet** {} // de esta si podemos

# 14. Singleton: constructor privado

Qué interesante! 😮

<https://platzi.com/clases/2880-typescript-poo/47265-singleton-contructor-privado/>

#### Singleton: constructor privado

Singleton nos previene crear múltiples instancias de una clase.

Esto es muy usado en Arquitectura de Software, pues nos ayuda a ahorrar uso de memoria.

**export** **class** **MyService** {

**static** instance: MyService | **null** = **null**;

**private** **constructor**(

**private** name: string

) {}

**get** **Name**() {

**return** this.name;

}

**static** **create**(name: string) {

**if** (MyService.instance === **null**) {

console.log('Creating new instance');

MyService.instance = **new** MyService(name);

}

**return** MyService.instance;

}

}

**const** myService1 = MyService.create('MyService1');

console.log(myService1.Name)

**const** myService2 = MyService.create('MyService2');

console.log(myService2.Name)

**const** myService3 = MyService.create('MyService3');

console.log(myService3.Name)

console.log(myService1 === myService2); //\* true

console.log(myService1 === myService3); //\* true

# 15. Promesas

<https://platzi.com/clases/2880-typescript-poo/47266-promesas/>

## [Axios](https://axios-http.com/docs/intro)

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.

## 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

**const** variableName = **new** Promese<dataType>(...);

## Fetch con NODE v18

Podemos utilizar nodejs para poder realizar peticiones fetch pero solo lo podemos hacer con la versión 18, es una feature no estable así que puede tener sus errores.

### Ejemplo en base al código de la clase

**async** **function** **getProductsFetch** () {

**const** promise = **await** fetch('https://api.escuelajs.co/api/v1/products');

**const** rta = **await** promise.json();

**return** rta;

}

# 16. 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.

**const** { data } = axios.get<dataType>(URL);

### Aserción AS

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

**async** **function** **getProductsAsync** () {

**const** rta = **await** axios.get(URL);

**const** data = rta.data **as** Product[];

**return** data;

}

## Código de la clase

**import** axios **from** 'axios';

**import** { Product } **from** './models/product.model';

**const** URL = 'https://api.escuelajs.co/api/v1/products';

// let anyVar: any = 12;

// let boolVar: boolean = anyVar;

(**async** () => {

// async function getProductsAsync (): Promise<Product[]> {

// const { data } = await axios.get(URL);

// return data;

// }

// async function getProductsAsync () {

// const { data } = await axios.get<Product[]>(URL);

// return data;

// }

**async** **function** **getProductsAsync** () {

**const** rta = **await** axios.get(URL);

**const** data = rta.data **as** Product[];

**return** data;

}

**const** products = **await** getProductsAsync();

console.log(products.map(({id, title}) => `${id} - ${title}`));

})()

<https://platzi.com/clases/2880-typescript-poo/47267-tipando-respuestas-http/>

# 18. Consumiendo ProductMemoryService

# 19. ProductHttpService

Para implementar lo que nos pide ProoductService se puede generar con este shortcut(que es lo que creo utilizo el profesor):  
\*Primero posicionan el cursor sobre **ProductHttpService** y luego…

En Mac: [Cmd + . (punto)]  
En Windows: [Ctrl + .(punto)]

**Les aparecera el menu y le dan a implement interface**

