Índice

[**Enums** 1](#_Toc128167575)

[**Tuples** 1](#_Toc128167576)

[**Unknown type** 2](#_Toc128167577)

[**Diferencias con any** 3](#_Toc128167578)

[**Recomendación** 3](#_Toc128167579)

[**Never type** 4](#_Toc128167580)

[**Parámetros opcionales y nullish-coalescing** 5](#_Toc128167581)

[**Parámetros opcionales** 5](#_Toc128167582)

[**Nullish-coalescing** 5](#_Toc128167583)

[**Parámetros por defecto** 6](#_Toc128167584)

[**Parámetros rest** 6](#_Toc128167585)

[**Sobrecarga de funciones: el problema** 7](#_Toc128167586)

[**Uso de la sobrecarga de funciones** 8](#_Toc128167587)

[**Sobrecarga de funciones: la solución** 8](#_Toc128167588)

[**Buenas prácticas** 9](#_Toc128167589)

[**Interfaces** 11](#_Toc128167590)

[**Diferencias entre interfaces y clases** 12](#_Toc128167591)

[**Cuando utilizar interfaces** 12](#_Toc128167592)

[**Extender interfaces** 12](#_Toc128167593)

[**¿Qué es extender interfaces?** 12](#_Toc128167594)

[**Diferencia con type** 12](#_Toc128167595)

[**Propiedades de solo lectura** 13](#_Toc128167596)

[**Omit y Pick Type** 13](#_Toc128167597)

[**Omit** 13](#_Toc128167598)

[**Pick** 14](#_Toc128167599)

[**Buenas prácticas** 14](#_Toc128167600)

[**Partial y Required Type** 14](#_Toc128167601)

[**Partial** 14](#_Toc128167602)

[**Required** 15](#_Toc128167603)

[**Readonly Type** 15](#_Toc128167604)

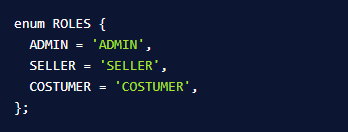
[**Anidamiento de utility types** 15](#_Toc128167605)

[**Acceder al tipado por índice** 16](#_Toc128167606)

[**ReadonlyArray** 17](#_Toc128167607)

# **Enums**

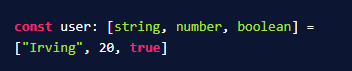
El **enum** nos permite configurar un set de opciones, es similar a los **literals types**.



Es de buena práctica que los **enums** vayan es mayúscula.

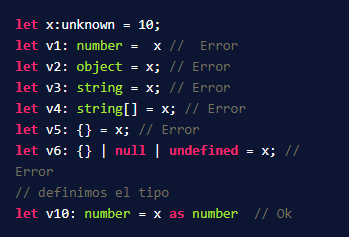
# **Tuples**

Las tuplas nos sirven para hacer un array fuertemente tipado especificando el tipo de dato de cada elemento del array, así como la cantidad de elementos en él.

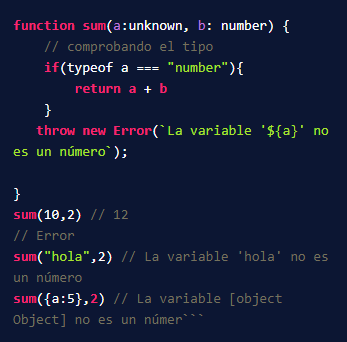


# **Unknown type**

Este nos dice que la **variable es desconocida**, es similar a **any**, pero es la forma recomendable de trabajar para evitar **any** en los casos que sean necesarios.



También nos permite realizar operaciones hasta poder definir un tipo de dato específico:



## **Diferencias con any**

Con **any** podemos hacer lo que queramos, no hay restricción alguna, pero con **unknown** vamos a tener advertencias al momento de usar alguna función o método con variables de tipo **unknown**.

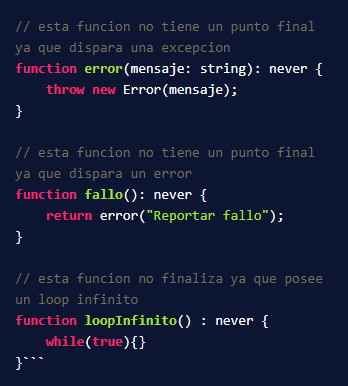
Para poder pasar las advertencias tenemos que usar un filtro o realizar una verificación, una estructura condicional, para poder hacer lo que queremos con esa variable.

## **Recomendación**

Utilizar **unknown** siempre por sobre **any**, pero solo en **caso de que sea necesario**.

# **Never type**

El tipo de dato **never**, sirve para tipar a una función que **nunca va a finalizar** o sencillamente que **pueda terminar el programa en el caso de lazar una excepción**.

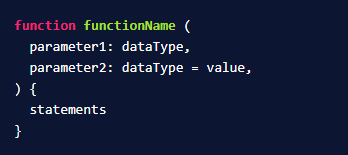


# **Parámetros opcionales y nullish-coalescing**

## **Parámetros opcionales**

* Estos son los últimos en ir declarados.
* Si no usamos o asignamos ese parámetro, tendrá por defecto el valor de **undefined**.

Valores predeterminados de parámetros



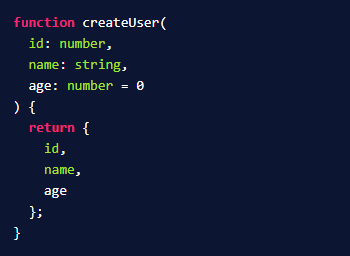
## **Nullish-coalescing**



De está manera **platziRank** contendrá el valor de la variable **rank**, en caso de que **rank** sea **null** o **undefined**, tomará el valor de **100**.

# **Parámetros por defecto**

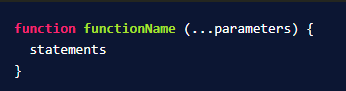
Para enviar un parámetro opcional usamos el nullish coalescing, sin embargo para enviar parámetros por defectos solo basta con igualarlo a la variable que queremos que tenga, si el usuario no nos indica el parámetro.



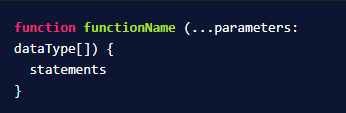
# **Parámetros rest**

Los parámetros **rest** nos permiten enviar la cantidad que queramos de parámetros a una función, casi sin límite.

**Sintaxis en JavaScript**

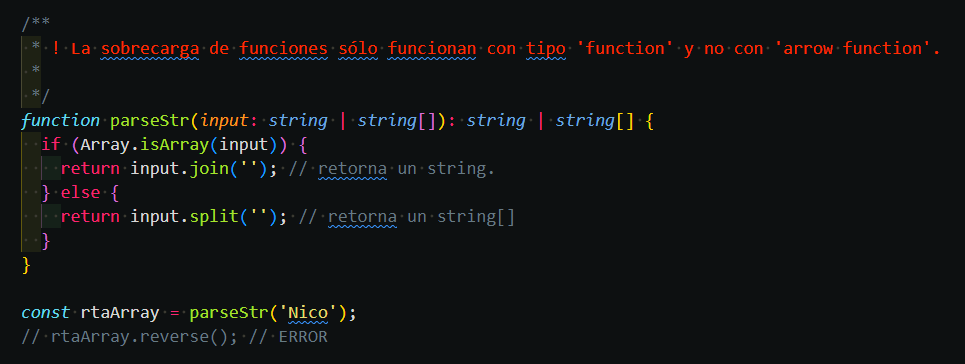


**Sintaxis en TypeScript**

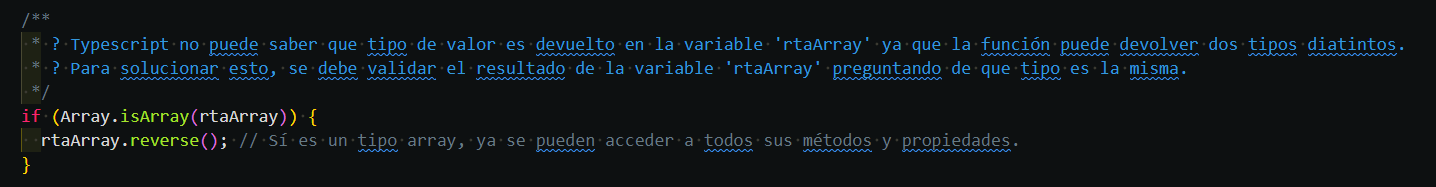


# **Sobrecarga de funciones: el problema**

Cuando tenemos una función que retorna más de un solo tipo de dato y a ese resultado lo queremos utilizar en otra parte de nuestro programa, no vamos a poder usarlo como tal, ya que TypeScript no sabe qué tipo de dato se retornó realmente.



Para solucionar esto, debemos hacer una validación de tipos.



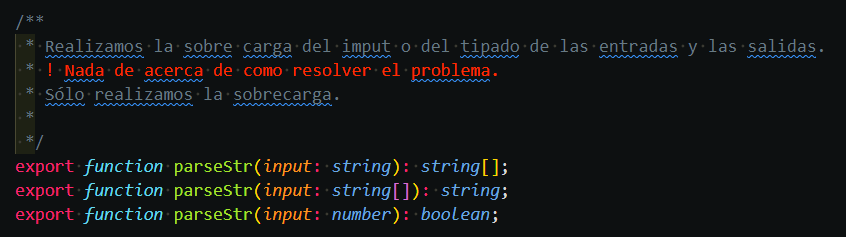
## **Uso de la sobrecarga de funciones**

Se suele emplear mucho en librerías, pero depende mucho de tu proyecto y de las ventajas que te puede llegar a dar.

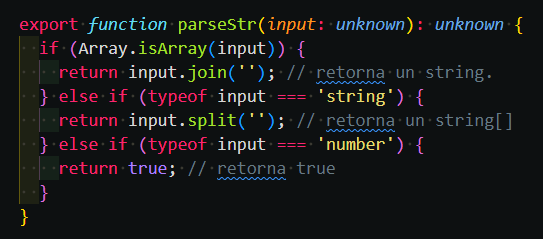
La sobre carga de funciones únicamente suelen darse con las funciones declarativas con la palabra reservada **function**.

# **Sobrecarga de funciones: la solución**

Para realizar la sobrecarga de funciones correctamente tenemos simplemente declarar otras funciones con el mismo nombre de la función que tiene la lógica implementada.

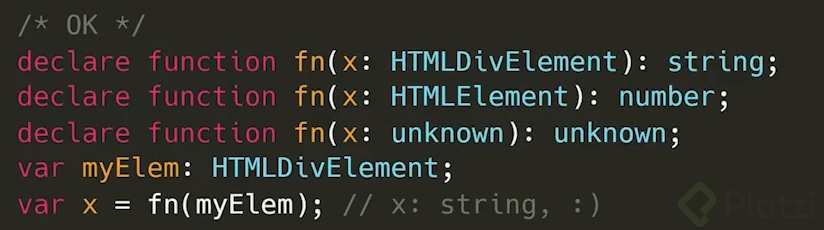


Y en la lógica de nuestra función, podemos utilizar el tipado **unknown**, validando dentro de la misma el tipo de valores de entradas y los de salidas.

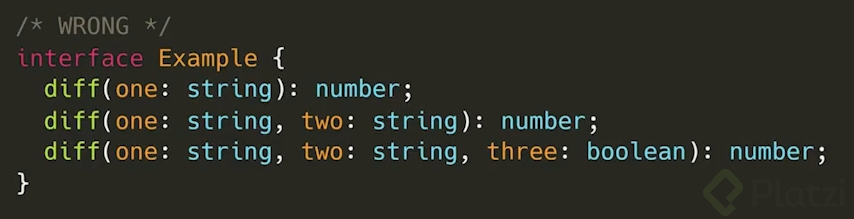


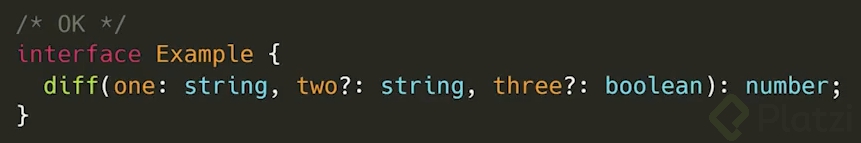
## **Buenas prácticas**

* Cuando tengamos sobre carga de métodos y por alguna razón tengamos un **unknown** o **any**, en esa sobre carga, lo mejor es dejarlo al final. Caso contario no funcionará correctamente esa aserción de tipos y por ende tampoco el autocompletado del editor.

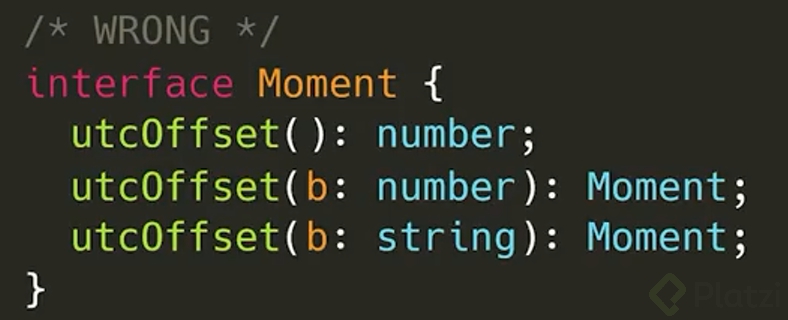


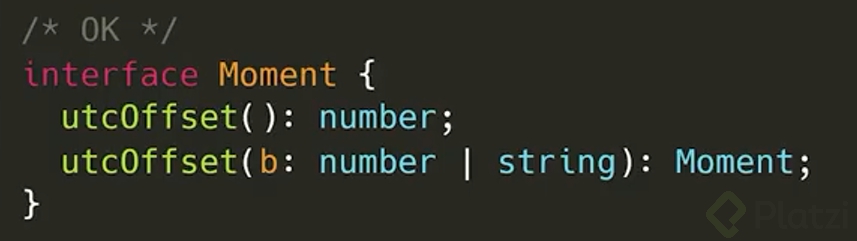
* Evaluar si realmente se necesita una sobre carga o simplemente puedes buscar otra forma de hacerlo como ser usando valores opcionales.





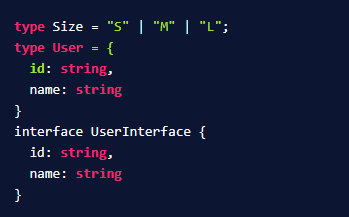
Más ejemplos





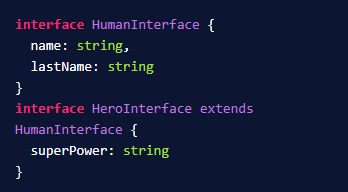
# **Interfaces**

Las interfaces funcionan muy similares a como lo hace **type**, pero en las interfaces solo aplica para los objetos.



**¿Entonces, porque usar interfaces si puedo usar type?**

La razón es sencilla, con las interfaces podemos heredar otras interfaces, y con los **type** no podemos hacer eso.



## **Diferencias entre interfaces y clases**

**Clase**

* Es un template que contiene **métodos**, **variables** y es el esqueleto de un **objeto**.
* Te permite inicializar nuevos objetos, una interfaz no.

**Interfaz**

* Es un **blueprint** o **plano** que describe que **propiedades** debe tener el **objeto**.

## **Cuando utilizar interfaces**

Cuando necesites crear un “**contrato**” de las propiedades y funciones que un objeto debe tener. Son muy útiles cuando el mismo objeto se debe repetir en varios archivos diferentes. Prácticamente, se utilizan para tipar nuestro código.

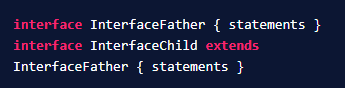
# **Extender interfaces**

## **¿Qué es extender interfaces?**

Es básicamente la **herencia** cómo funciona en la programación orientada a objetos.

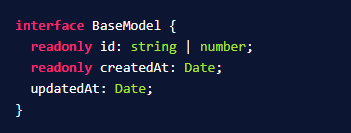
## **Diferencia con type**

Al usar interfaces podemos aplicar **herencia**, pero esto no ocurre con **type**.



# **Propiedades de solo lectura**

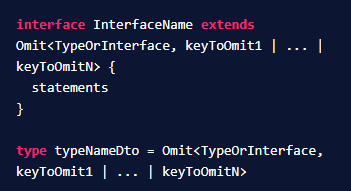
Como su nombre lo indica, este feature de TS nos ayuda a tener ciertos atributos **solo de lectura**. Lo que significa que no pueden ser modificados.



# **Omit y Pick Type**

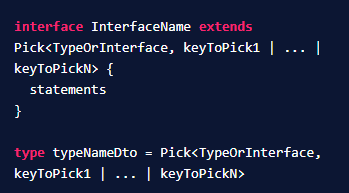
## **Omit**

Podemos omitir las propiedades, campos o llaves que queramos.



## **Pick**

Es lo contrario de **Omit**, aquí elegimos los campos que queremos que estén en nuestro **type** o **interface**.



## **Buenas prácticas**

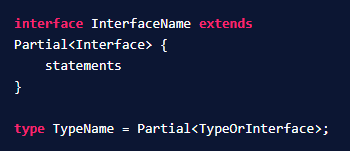
Una buena práctica es que los DTOs tengan su propio archivo.

# **Partial y Required Type**

## **Partial**

Nos permite colocar todos los parámetros de una interface como **opcionales**, sin necesidad que colocar **?** parámetro por parámetro.

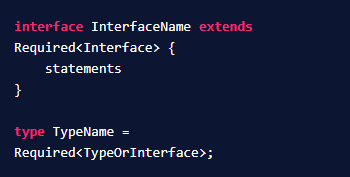
Sintaxis



## **Required**

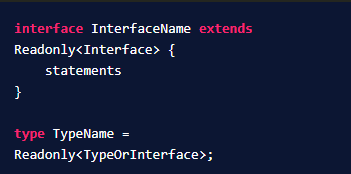
Nos permite colocar todos los parámetros de una interface como **obligatorios**.

Sintaxis



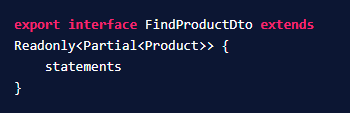
# **Readonly Type**

**Readonly** es un método para crear una interfaz de **solo lectura**, usualmente este es un método para búsqueda.



## **Anidamiento de utility types**

Podemos utilizar el anidamiento para poder fusionar las distintas características de los diferentes **utility types**.



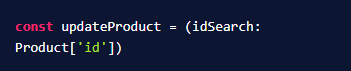
# **Acceder al tipado por índice**

Podemos acceder al **tipado** de una **interface** por medio de **bracket notation**, al colocar una propiedad entre comillas dentro de los brackets vamos a obtener por resultado el tipo de dato que tiene esa propiedad.

Al colocar el tipo de dato de esta forma nos evitamos los dolores de cabeza cuando se nos cambien algún tipo de dato de nuestra interface por alguna razón.



Ejemplo



En este caso estamos pasando como tipo de dato una interface y como índice tenemos a una de sus propiedades, por ende, lo que en realidad recibe la propiedad de la función (**idSearch**) es el tipo de dato de ese índice de la interface.

# **ReadonlyArray**

Este tipo de dato es un **array**, pero la principal diferencia con los arrays “**normales**” es que no existen ninguno de los métodos de mutación, tales como **pop**, **push**, **shift**, etc.

