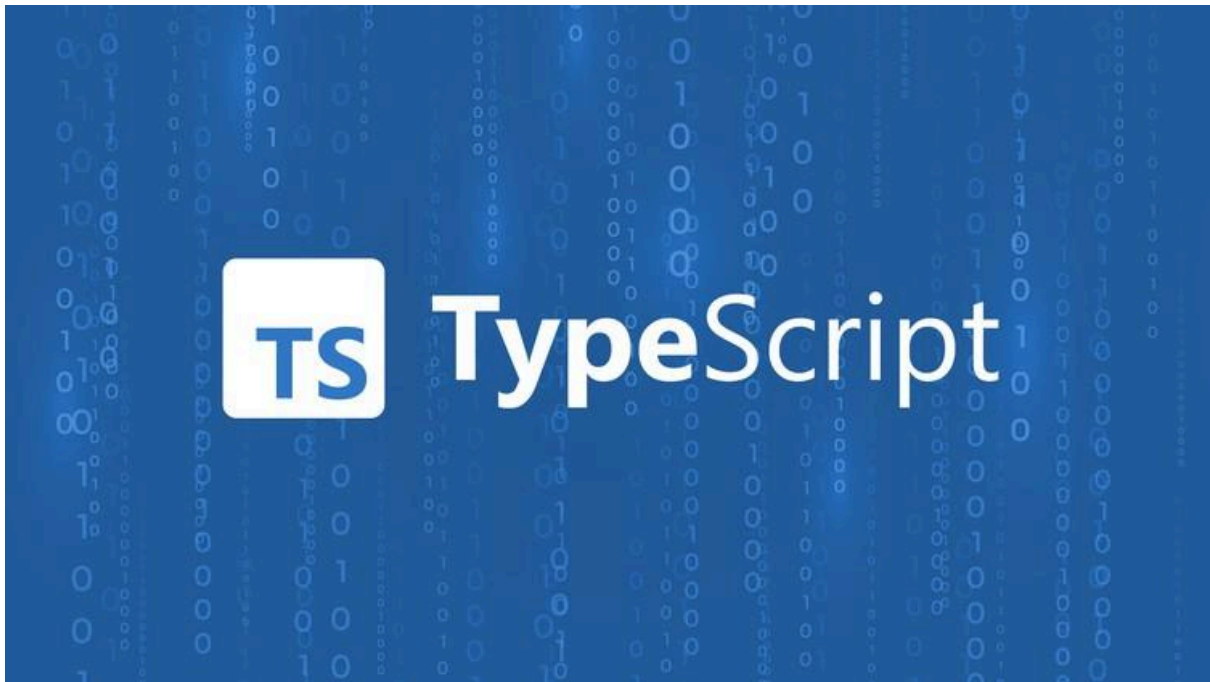


MANUAL DE TYPESCRIPT



Integrantes:

- David Dorante Lucas
- Álvaro Sánchez Moreno
- Juan Martín Candela
- José María Mañero Brenes
- Antonio Jesús León Fernández

ÍNDICE:

1. Introducción a Typescript.	4
1.1. Orígenes.	4
1.2. Características principales.	4
1.3. Ventajas.	5
1.4. Conclusión.	6
2. Compilación en TypeScript.	6
3. Herramientas.	7
4. Extensión recomendada.	8
5. Funciones en TypeScript.	8
5.1. Declaración de funciones.	8
5.2. Parámetros opcionales y predeterminados.	9
5.3. Funciones como tipos.	9
6. Tipar Parámetros Funciones.	9
6.1. Tipado explícito de parámetros.	9
6.2. Tipado implícito de parámetros.	10
6.3. Tipos de parámetros opcionales.	10
7. Tipar Arrow Functions.	11
7.1. Definir los parámetros y el tipo de retorno.	11
7.2. Sintaxis de las funciones de flecha.	11
7.3. Uso de tipos opcionales y valores por defecto.	11
8. Funciones never / void.	12
9. Objetos.	13
10. Mutabilidad.	14
11. Template union types.	15
12. Union types.	15
13. Intersection types.	16
14. Type en TypeScript.	16
14.1. Indexación de Tipos.	17
14.2. Tipo desde Valor (typeof).	17
14.3. Tipo desde Función (ReturnType).	18
14.4. Tipo en Matrices.	18
15. Enum en TypeScript.	18
16. Aserciones de tipos.	20
17. Funciones en interfaces.	22
18. Narrowing.	23
19. Ámbito de las variables y las funciones.	24
19.1. Private.	24
19.2. Protected.	24
19.3. Public.	25
20. Convención d.ts.	26

MANUAL DE TYPESCRIPT

1. Introducción a Typescript.

En el mundo del desarrollo web, TypeScript ha ganado una gran popularidad en los últimos años gracias a su capacidad para mejorar la productividad y la calidad del código en proyectos de gran escala. TypeScript es un superconjunto tipado de JavaScript que añade características de lenguajes de programación más tradicionales, como el tipado estático y la orientación a objetos, con un flexible y dinámico ecosistema de JavaScript.

1.1. Orígenes.

TypeScript fue desarrollado por Microsoft y lanzado públicamente en octubre de 2012. Su objetivo era abordar las limitaciones de JavaScript en proyectos complejos, donde la ausencia de tipado estático y otras características comunes en lenguajes de programación modernos podían resultar en código difícil de mantener y depurar. TypeScript se diseñó para ser compatible con el código JavaScript existente, permitiendo a los desarrolladores adoptarlo gradualmente en sus proyectos.

1.2. Características principales.

TypeScript, actualmente tiene un gran uso debido a las siguientes características:

- **Tipado Estático.**

TypeScript permite asignar tipos a variables, parámetros de funciones, propiedades de objetos y otros elementos del código, lo que ayuda a detectar errores de tipo en tiempo de compilación y a mejorar la robustez del código.

- **Orientación a Objetos.**

TypeScript soporta la programación orientada a objetos mediante la definición de clases, interfaces, herencia, encapsulamiento y otros conceptos propios de la POO.

- **Mejoras en la productividad.**

Con TypeScript, los IDEs pueden proporcionar características avanzadas de autocompletado, navegación de código, refactorización y detección de errores en tiempo real, lo que facilita el desarrollo y reduce el tiempo dedicado a la depuración.

- **Compilación a JavaScript.**

TypeScript se compila a JavaScript estándar, lo que significa que el código TypeScript puede ser ejecutado en cualquier entorno que soporte JavaScript, incluyendo navegadores web, servidores Node.js y dispositivos móviles.

- **Ecosistema de Herramientas.**

TypeScript cuenta con una amplia variedad de herramientas, bibliotecas y marcos de trabajo desarrollados por la comunidad y empresas líderes en tecnología, lo que facilita la construcción de aplicaciones web modernas y escalables.

1.3. Ventajas.

El uso de TypeScript en el trabajo ofrece ventajas significativas para mejorar la calidad del código, la productividad del equipo y la experiencia del desarrollo. Entre estas ventajas se encuentran:

- **Tiempo estático Opcional.** Permite agregar tipos estáticos de JavaScript, lo que atrapa errores en tiempo de compilación y mejora la legibilidad del código.
- **Mejorar Mantenibilidad del Código:** Los tipos estáticos facilitan la comprensión y el mantenimiento del código a lo largo del tiempo, además de ayudar en la colaboración y la integración de nuevas características.
- **Productividad Mejorada:** Los IDEs modernos ofrecen un sólido soporte para TypeScript, lo que incluye funciones como autocompletado inteligente y detección de errores en tiempo real, aumentando la productividad del equipo.
- **Desarrollo Escalable:** Es especialmente útil en proyectos grandes y complejos, proporcionando una estructura más sólida y comprensible para el código.
- **Compatibilidad con el Ecosistema de JavaScript:** TypeScript es un superconjunto de JavaScript, lo que facilita su adopción en proyectos existentes y su uso con bibliotecas y frameworks populares.
- **Menos Documentación.** TypeScript se documenta automáticamente cuando se usan las anotaciones de tipo.

1.4. Conclusión.

En conclusión, TypeScript ofrece una forma más robusta y estructurada de desarrollar aplicaciones web en comparación con JavaScript puro. Su combinación de tipado estático, orientación a objetos y herramientas avanzadas lo convierten en una opción atractiva para equipos de desarrollo que buscan mejorar la calidad y la mantenibilidad de su código. Con el continuo crecimiento de su popularidad y la adopción por parte de grandes empresas, TypeScript se ha convertido en una parte fundamental del ecosistema de desarrollo web moderno.

2. Compilación en TypeScript.

A la hora de la verdad, Typescript no es realmente compilado de manera completa. Al ser un superconjunto de Javascript, todo el código que se encuentre escrito con Typescript será traducido a Javascript en el momento de la compilación. De esta manera, aunque escribamos algo que Typescript considere como error, por ejemplo:

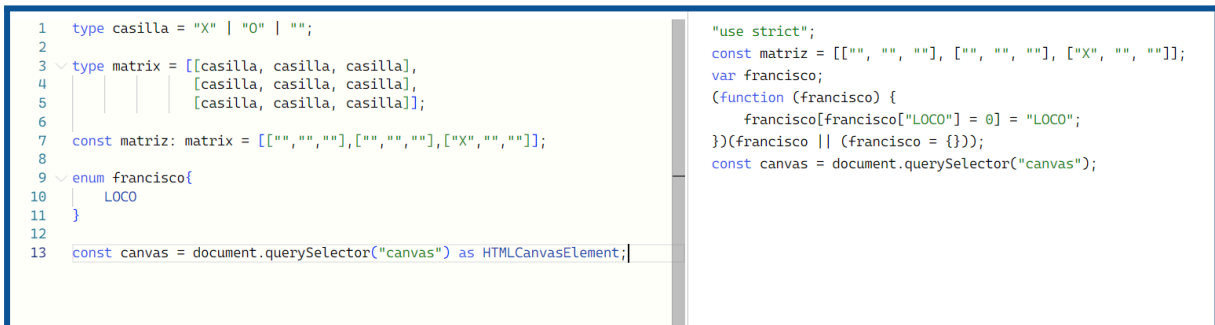
```
const num: number = "Hola"
```

Se compilará de manera correcta, y Javascript lo interpretará de la siguiente manera:

```
const num = "Hola"
```

Por lo tanto, podríamos considerar Typescript como un lenguaje que nos ayudará a tener un código pseudo tipado, ya que a la hora de la compilación no se tomarán en cuenta los tipos, sin embargo dentro del código si podrás mantener cierta coherencia con el tipado, lo que te será útil para asegurarte de que cierta variable esté inicializada o tenga un tipo esperado.

Veamos un ejemplo más práctico de cómo sería la compilación de Typescript a Javascript.



En este ejemplo, podemos ver cómo sería la conversión de Typescript (izquierda) a Javascript (derecha).

3. Herramientas.

Para trabajar crucialmente con TypeScript, es crucial evitar utilizar varias herramientas que faciliten el desarrollo de un entorno crucial, mejorando la calidad del código, la productividad del código y la experiencia de desarrollar. Las más importantes son:

- **Visual Studio Code (VS Code):** Potente editor de código con destacado soporte para TypeScript, incluyendo autocompletado inteligente y detección de errores en tiempo real.
- **TypeScript Compiler (tsc):** Herramienta de línea de comandos que convierte código TypeScript a JavaScript, integrable en flujos de trabajo de construcción con Webpack o Gulp.
- **tsconfig.json:** Archivo de configuración para especificar opciones de compilación en proyectos TypeScript, como la versión de ECMAScript o la inclusión de archivos.
- **Node.js y npm:** Integración perfecta con Node.js y npm, permitiendo el uso de bibliotecas y herramientas de terceros. Se pueden instalar definiciones de tipos con el prefijo @types.
- **Definiciones de Tipos (@types):** Mecanismo para definir tipos para bibliotecas de JavaScript. Se instalan a través de npm, por ejemplo, @types/react para definiciones de React.
- **Herramientas de Pruebas:** Jest, Mocha, Jasmine, Cypress y WebDriverIO ofrecen soporte para pruebas unitarias y de extremo a extremo con TypeScript.

- **Herramientas de Depuración:** Los navegadores modernos y Node.js permiten la depuración de aplicaciones TypeScript a través de mapas de origen. VS Code también proporciona un potente depurador integrado.
- **Herramientas de Gestión de Proyectos:** Utiliza Git para el control de versiones y herramientas como npm scripts, Gulp o Grunt para automatizar tareas comunes como compilación y pruebas.

4. Extensión recomendada.

Una de las extensiones más recomendadas es la extensión “Pretty TypeScript Errors”, ya que mejora la experiencia de desarrollo al hacer que los mensajes de error de TypeScript sean más legibles, navegables y comprensibles, ayudando esto a los desarrolladores a identificar y solucionar problemas de manera más eficiente, lo que conduce a un proceso de desarrollo más fluido y productivo.

5. Funciones en TypeScript.

5.1. Declaración de funciones.

En TypeScript, puedes declarar funciones utilizando la palabra clave `function`, seguida del nombre de la función y la lista de parámetros entre paréntesis. También puedes especificar el tipo de los parámetros y el tipo de retorno de la función, lo que ayuda a hacer tu código más seguro y legible.

```
function mostrarNombre(name: string): string {  
    return `Hola, ${name}!`;  
}
```

5.2. Parámetros opcionales y predeterminados.

Puedes hacer que los parámetros de una función sean opcionales agregando un signo de interrogación '?' después del nombre del parámetro. También puedes proporcionar valores predeterminados para los parámetros.

```
function sakudarPersona(name: string, mensaje: string = "Hola"): string {  
  return `${mensaje}, ${name}!`;  
}
```

5.3. Funciones como tipos.

En TypeScript, puedes tratar las funciones como tipos, lo que te permite declarar variables cuyo tipo es una función específica.

```
type MathFunction = (numero1: number, numero2: number) => number;  
const add: MathFunction = (numero1, numero2) => numero1 + numero2;
```

6. Tipar Parámetros Funciones.

6.1. Tipado explícito de parámetros.

Puedes especificar el tipo de cada parámetro de la función directamente en la declaración de la función.

6.2. Tipado implícito de parámetros.

TypeScript también puede inferir automáticamente los tipos de los parámetros según el contexto si no se especifican explícitamente.

```
ers > Davixu > Downloads > TS prueba.ts > ...  
function greet(nombre, edad) {  
  console.log(`Hola, ${nombre}! Tu tienes ${edad} años.`);  
}
```

6.3. Tipos de parámetros opcionales.

Puedes hacer que los parámetros de una función sean opcionales añadiendo un '?' después del nombre del parámetro y especificando un valor predeterminado en caso de que no se proporcione.

```
function saludar(nombre: string, edad?: number): void {  
  if (edad !== undefined) {  
    console.log(`Hola, ${nombre}! Tienes ${edad} años.`);  
  } else {  
    console.log(`Hola, ${nombre}!`);  
  }  
}
```

7. Tipar Arrow Functions.

Para tipar las funciones flechas en Typescript, puedes seguir estas pautas:

7.1. Definir los parámetros y el tipo de retorno.

Especifica los tipos de parámetros de entrada y el tipo de retorno de la función. Puedes utilizar los tipos primitivos de Typescript (como 'number', 'string', 'boolean', etc.), tipos personalizados definidos por el usuario, o tipos compuestos como arrays, objetos entre otros.

7.2. Sintaxis de las funciones de flecha.

Las funciones de flecha se definen utilizando la sintaxis '() => {}', donde los paréntesis contienen los parámetros de entrada (si los hay), seguidos por el operador de flecha '=>', y luego las llaves '{}' que contienen el cuerpo de la función.

7.3. Uso de tipos opcionales y valores por defecto.

Puedes hacer que los parámetros de la función sean opcionales añadiendo un signo de interrogación '?' al final del nombre del parámetro de la función sean opcionales añadiendo un signo de interrogación '?' al final del nombre del parámetro. También puedes asignar valores por defecto a los parámetros utilizando la sintaxis 'nombreParametro: tipo = valorPorDefecto'.

Ejemplo:

```
1  💡 Definición de una función de flecha con tipado explícito
2  const sumar = (num1: number, num2: number): number => {
3    |    return num1 + num2;
4    |};
5
6  // Llamada a la función sumar
7  const resultado = sumar(5, 3); // resultado será de tipo number
8
9  console.log(resultado); // Salida: 8
```

Como vemos en este ejemplo sumar está tipada por dos parámetros de tipo 'number', para asegurarse de que la función solo recibe números.

El tipado en TypeScript es opcional y se puede omitir si lo deseas, ya que TypeScript puede inferir automáticamente los tipos en la mayoría de los casos. Sin embargo, es una buena práctica tipar tus funciones para mejorar la legibilidad y la mantenibilidad del código, así como para detectar posibles errores de tipo durante el desarrollo.

8. Funciones never / void.

En TypeScript existen funciones que nunca devolverán nada al ser ejecutadas, como por ejemplo una que solo devuelve una excepción, estas funciones pueden tener el tipo never. Esto puede ser útil para evitar algunos problemas, pero no es necesario, es para asegurarse de que la función no devolverá nada inesperado.

Ejemplo:

```
function lanzarExcepcion(mensaje: string): never {  
  throw new Error(mensaje);  
}
```

En cambio, el tipo void, para una función, indica que ésta no tiene un valor de return, es decir, que no devuelve nada específico, pero eso no evita que pueda devolver un valor.

Ejemplo:

```
function imprimirMensaje(): void {  
  console.log("Hola, mundo");  
}
```

En resumen, el tipo never lo usas para funciones que nunca serán completadas normalmente, que siempre lanzarán alguna excepción, mientras que el tipo void lo usas en funciones que no hacen return.

9. Objetos.

Dicho de forma simple, los objetos son un conjunto de variables (propiedades) asociadas a éste. Por ejemplo:

```
let persona = {  
  nombre: "pepee",  
  edad: 23  
};
```

Después puedes acudir a estas variables de esta forma:

```
persona.nombre = "Pepe";
```

TypeScript es estricto, por ejemplo, si has inicializado una variable dentro de un objeto con una cadena, esa variable no puede contener otro tipo de dato que no sea cadena. Tampoco puedes definir nuevas variables para el objeto fuera de éste.

Type Alias

Imagina que teniendo el ejemplo anterior, creo esta función:

```
function crearPersona(nombre: string, edad: number) {  
  return {nombre, edad}  
};  
const pepe = crearPersona("Pepe", 23);
```

¿Cómo puede TS saber la diferencia entre uno y otro?, para aclarar esto podemos crear nuestro propios tipos.

Ejemplo:

```
type Persona = {  
  nombre: string,  
  edad: number  
};
```

Con nuestro propio tipo ahora podemos hacer objetos de nuestro tipo específico o funciones que devuelvan un objeto de nuestro tipo de esta forma:

```
let persona: Persona = {---};
```

```
function crearPersona(nombre: string, edad: number): Persona {---};
```

Por cierto, cuidado al crear un tipo, éste debe de tener la primera letra en mayúscula siempre.

Optional properties y optional chaining operator

A nuestro tipo le podemos poner variables (propiedades) que no son obligatorias (optional property) a la hora de crear un objeto de dicho tipo.

Ejemplo:

```
type Persona = {  
  nombre: string,  
  edad: number,  
  altura?: double  
};
```

De esta forma, podemos crear un objeto de tipo Persona sin la necesidad de poner la altura. Con eso dicho, ¿qué pasa si intentamos acceder a la propiedad altura de un objeto que no la tiene?, se pondrá un signo de interrogación de forma automática indicando que puede ser double o undefined. Ejemplo:

```
persona.altura.toString(); > persona.altura?.toString();
```

10. Mutabilidad.

Imagina que le añadimos una nueva propiedad al tipo Persona que no queremos que nadie lo altere, por ejemplo un id. Para evitar esto podemos usar una opción llamada readonly:

```
type Persona = {  
  nombre: string,  
  edad: number,  
  altura?: double,  
  readonly id?: number  
};
```

Esto hará que si alguien intenta cambiar el valor de la propiedad id de un objeto, le salga un error, pero en realidad eso no evitará que el valor sea inmutable, para evitarlo se podría usar `object.freeze()`, pero se trata de una función JavaScript..

11. Template union types.

¿Sabías que se pueden usar nuestros propios tipos dentro de otros tipos? Digamos que ahora queremos que el id de Persona siga un formato específico, podemos hacerlo así:

```
type PersonId = `${string}-${number}`;  
type Persona = {  
  nombre: string,  
  edad: number,  
  altura?: double,  
  readonly id?: PersonId  
};
```

De esta forma hacemos que el id deba estar formado por una cadena, un guión y un número. Esto es muy útil para tener el código organizado, por ejemplo para hacer que las variables que son de un mismo tipo sigan un formato específico.

Dato extra: Las propiedades de un tipo no tienen por qué estar separadas por comas.

12. Union types.

Los union types son tipos que cuentan con una serie de posibles valores que pueden tener, por ejemplo:

```
type Sexo = 'hombre' | 'mujer' | 'otro'
```

En este caso tenemos un tipo que puede tener tres valores distintos y no puede tener ningún valor distinto a los ya definidos. Esto puedes hacerlo también con variables y usando tipos y valores a la vez:

```
let numero1 = 1 | string
```

13. Intersection types.

Podemos crear tipos que se conforman de otros tipos, por ejemplo:

```
type InfoBasicaPersona = {  
  nombre: string,  
  edad: number,  
}  
type Sexo = 'hombre' | 'mujer' | 'otro'  
type Persona = InfoBasicaPersona & Sexo
```

Así, tenemos un tipo que está conformado por las propiedades de varios tipos.

14. Type en TypeScript.

El sistema de tipos en TypeScript es una característica esencial que proporciona una capa adicional de seguridad y mantenibilidad a nuestras aplicaciones. Por lo tanto, podemos decir que un tipo en TypeScript es como una etiqueta que describe el tipo de datos que puede contener una variable, un parámetro de función, el valor de retorno de una función, una propiedad de un objeto, etc. Vamos a ver los distintos casos en los que podemos aplicar los tipos en TypeScript.

14.1. Indexación de Tipos.

La indexación de tipos es un concepto que nos permite definir tipos para objetos con un conjunto dinámico de propiedades. En lugar de enumerar todas las propiedades individualmente, podemos utilizar una indexación para representarlas de manera más flexible.

```
type Persona = {  
  nombre: string;  
  edad: number;  
  [key: string]: any; // Propiedades adicionales de cualquier tipo  
};  
  
const person: Person = {  
  nombre: 'John',  
  edad: 30,  
  ciudad: 'Salamanca', // Propiedad adicional permitida  
};
```

En este ejemplo, Persona es un tipo que representa un objeto con propiedades estáticas {nombre, edad} y dinámicas {ciudad}, es decir, en las estáticas se definen dos propiedades las cuales están asignadas a nombre y edad, y la tercera es una propiedad la cual tiene que ser un String pero su valor puede ser cualquiera {any}, con lo cual al igual que hemos definido también podemos definir

14.2. Tipo desde Valor (typeof).

El operador typeof en TypeScript nos permite obtener el tipo de una variable y utilizarlo para definir otro tipo. Esto es útil cuando queremos crear un tipo basado en el tipo de una variable existente.

```
type TipoNombre = typeof nombre; // TipoNombre ahora es 'string'
```

Aquí, TipoNombre es un tipo que se basa en el tipo de la variable nombre, que es string.

14.3. Tipo desde Función (ReturnType).

El operador `ReturnType` nos permite obtener el tipo de retorno de una función y utilizarlo para definir otro tipo. Esto es útil cuando queremos crear un tipo basado en el tipo de retorno de una función existente.

```
type TipoSaludo = ReturnType<typeof saludar>; // TipoSaludo ahora es 'string'
```

En este caso, `TipoSaludo` es un tipo que se basa en el tipo de retorno de la función `saludar`, que es `string`.

14.4. Tipo en Matrices.

En TypeScript, podemos definir tipos para matrices utilizando la sintaxis de corchetes. Esto nos permite especificar el tipo de elementos que contendrá la matriz.

```
type MatrizNumeros = number[];  
const numeros: MatrizNumeros = [1, 2, 3, 4, 5];
```

Aquí, `MatrizNumeros` es un tipo que representa una matriz de números.

15. Enum en TypeScript.

El `enum` en TypeScript es un tipo de dato que te permite definir un conjunto de opciones predefinidas que puedes usar en tu código. Esto hace que tu código sea más claro y más fácil de entender. Es como tener una lista de opciones a tu disposición para elegir en cualquier momento. A continuación, vamos a ver un ejemplo simple:

```
enum DiaSemana {  
  Lunes, //esto sería el valor 0  
  Martes, //valor 1  
  Miercoles, //2  
  Jueves, //3  
  Viernes, //4  
  Sabado, //5  
  Domingo //6  
}
```

```
let dia = DiaSemana.Martes;  
console.log(dia);
```

```
// También puedes acceder al valor original a partir del índice  
console.log(DiaSemana[1]); //Imprime el índice 1
```

Aunque también puedes asignar un string a las opciones del enum si te resulta más fácil o más útil, por ejemplo:

```
enum DiaSemana {  
  Lunes = 'Lun',  
  Martes = 'Mar',  
  Miercoles = 'Mie',  
  Jueves = 'Jue',  
  Viernes = 'Vie',  
  Sabado = 'Sab',  
  Domingo = 'Dom'  
}
```

```
let dia: DiaSemana = DiaSemana.Martes;
```

```
console.log(dia); // Imprime: Mar
```

Una duda que surge mucho a los programadores es cuando usar y cuándo no usar const en los enum, por lo tanto vamos a ver cuando es más recomendable usarlo o no:

- Se utiliza const cuando el enum lo utilizas en tu aplicación ya que genera menos código cuando es transpilado, es decir, cuando quieres convertir TypeScript a JavaScript.

- No se utiliza const para definir el enum cuando estás creando una biblioteca, una librería o un componente el cual vamos a utilizar fuera de nuestra aplicación, ya que no puedes agregar o eliminar propiedades del enum para adaptarlo a tus necesidades.

16. Aserciones de tipos.

En TypeScript, las aserciones de tipos, también conocidas como type assertions, te permiten decirle al compilador que tratemos una variable como si fuera de un tipo específico, incluso cuando TypeScript no puede inferir ese tipo por sí mismo.

```
TS pruebas.ts > ...
1  let x: any = "123";
2  let y: number = x;
3
```

En este ejemplo básico podemos ver que si declaramos una variable con el tipo any (cualquier tipo), el compilador lo tratará como cualquier tipo de dato, por eso vemos justo debajo, la declaración de otra variable, en este caso de tipo number, a la que le asignamos el valor de la primera variable, tratándola como number en vez de string, pero cuidado, ya que en tiempo de ejecución nos dará error si queremos mostrar ese valor, ya que x sigue siendo una cadena, en ningún momento lo hemos “parseado” de ninguna forma a tipo number.

Si en vez de tipo any le ponemos cualquier otro tipo que no sea number, nos dará error también de la misma forma.

```
TS pruebas.ts ●
TS pruebas.ts > ...
1  let x: any = "123";
2  let y: number = x as number ;
3
```

En este caso, utilizamos la sintaxis de inserción de tipo “as number”, lo que le dice al compilador de TypeScript que trate “x” como si fuera de tipo number. Aunque “x” es de tipo “any”, la inserción de tipo “as number” le permite al compilador realizar la asignación a “y” como un número. Sin embargo, esto no cambia el tipo de “x”, solo permite que el compilador trate “x” temporalmente como number en esta asignación específica.

```
TS pruebas.ts ●
TS pruebas.ts > ...
1   let x: any = "123";
2   let y:number = <number> x;
3
```

Esta otra sintaxis sería otra forma con la misma funcionalidad que “as”, definiendo el tipo dentro de los signos menor que y mayor que.

Ahora veremos otro ejemplo utilizando una interfaz, que es una forma de definir la estructura que deben tener ciertos objetos en términos de propiedades y métodos, pero sin proporcionar una implementación concreta de esos métodos.

```
TS pruebas.ts ●
TS pruebas.ts > obtenerLongitud
1   function obtenerLongitud(s: any): number {
2       return (s as string).length;
3   }
```

Podemos ver otro ejemplo de aserción en una función, a la cual le introducimos un valor cualquiera de tipo any e indicamos que la función va a devolver un tipo number, ya que devuelve el número de longitud del parámetro pasado al que lo convierte en string para poder ejecutar esa función, pero cuidado, porque si introducimos otro valor que no sea una cadena de caracteres, nos devolverá un error.

17. Funciones en interfaces.

```
TS pruebas.ts •
TS pruebas.ts > ...
1 interface PersonaAcciones {
2   remove: (id: number) => string;
3   add: (persona: Persona) => void;
4   update: (id: number, persona: Persona) => boolean;
5 }
```

En este ejemplo podemos ver una interfaz que proporciona una estructura que las clases pueden implementar para garantizar que tengan estos métodos disponibles.

- **Método remove.**

Este método recibe un parámetro `id` de tipo `number`, que se espera que sea el identificador único de la persona que se desea eliminar.

El tipo de retorno de este método es `string`, lo que significa que devolverá un mensaje “Persona eliminada”, por ejemplo.

- **Método add.**

Este método recibe un parámetro `persona` de tipo `Persona`, representando a ésta.

Este método no devuelve ningún valor (`void`), simplemente lo agrega.

- **Método update.**

Este método recibe dos parámetros: un `id` de tipo `number` que identifica a la persona que se desea actualizar, y una `persona` de tipo `Persona`.

El tipo de retorno de este método es `boolean`, lo que indica que se espera que devuelva un valor booleano que indique si la actualización fue exitosa o no, por ejemplo.

18. Narrowing.

En Typescript, hay veces dónde no se sabe de manera concreta qué tipo de dato vamos a estar usando (Ya sea porque tiene tipo `any` o porque puede ser de dos tipos). Por lo tanto, hay veces dónde es preciso hacer narrowing para tener certeza de cuál es el tipo que vamos a trabajar. Entendemos el concepto de narrowing como una forma de filtrar un dato, de manera que si pasa cierta validación se puede saber de qué tipo es, un ejemplo de esto sería el siguiente:

```
1  function mostrarLongitud (objeto: number | string) {
2      if (typeof objeto === 'string') {
3          return objeto.length
4      }
5
6      return objeto.toString().length
7  }
8
9  mostrarLongitud('1')
```

De esta manera, nos aseguramos que dentro del `if` estemos trabajando con un tipo `"string"`, por lo que Typescript inferirá el tipo de dato de manera que podremos usar funciones de tipo `"string"` dentro de ese `if` sin que Typescript nos dé error.

Hay casos donde la `"typeof"` no será útil a la hora de determinar el tipo de ciertos elementos. Ya que `"typeof"` funciona exclusivamente con tipos primitivos como pueden ser strings, números, arrays...

Para estos casos, haremos uso de `"instanceof"`, que determinará la instancia de la variable con la que estemos trabajando.

```
mouseOut(event: Event){
    const elemento = event.target;

    if(elemento instanceof HTMLElement){
        elemento.style.backgroundColor = this.bgColor;
    }
}
```

De esta manera, podríamos utilizar las funciones propias de la instancia dentro del bloque *if*.

19. Ámbito de las variables y las funciones.

En programación, entendemos como ámbito el rango en el que un dato puede ser reconocido por otro. Es decir, desde cuáles lugares de una aplicación se puede acceder a un dato en concreto.

En el caso de Typescript, podremos asignar a nuestras variables ciertas propiedades que harán que ese rango sea de mayor o menor alcance.

19.1. Private.

La propiedad “private” es aquella que hace que las variables de ciertas clases u objetos, únicamente sean accesibles por la estructura que las contiene. Un ejemplo sería el siguiente:

```
1  class Persona{
2      private nombre: "manolo"
3
4      constructor(){
5
6      }
7
8  }
9
10 const personaInstancia = new Persona();
11 personaInstancia.nombre    Property 'nombre' is private and only accessible within class 'Persona'.
```

Al estar intentando acceder al valor del atributo “nombre” desde fuera de clase Typescript nos lanzará un error.

19.2. Protected.

La propiedad “protected” es aquella que hace que las variables de ciertas clases u objetos, sean accesibles únicamente desde la estructura que las contiene o

clases que extiendan o sean hijos de la clase que tiene el atributo marcado como “protected”. Un ejemplo sería el siguiente:

```
1  class Persona{
2      protected nombre: "manolo"
3
4      constructor(){
5
6      }
7
8  }
9
10 class Bombero extends Persona{
11
12     decirNombre(){
13         console.log("Mi nombre es " + this.nombre);
14     }
15 }
```

19.3. Public.

La propiedad “public”, es aquella es la predeterminada a la hora de crear una variable o atributo de una clase u objeto. Hace que el dato sea accesible incluso fuera de clase, es decir, que en cualquier lugar donde haya una instancia de ese dato se podrá acceder a esa propiedad. Un ejemplo de ello sería:

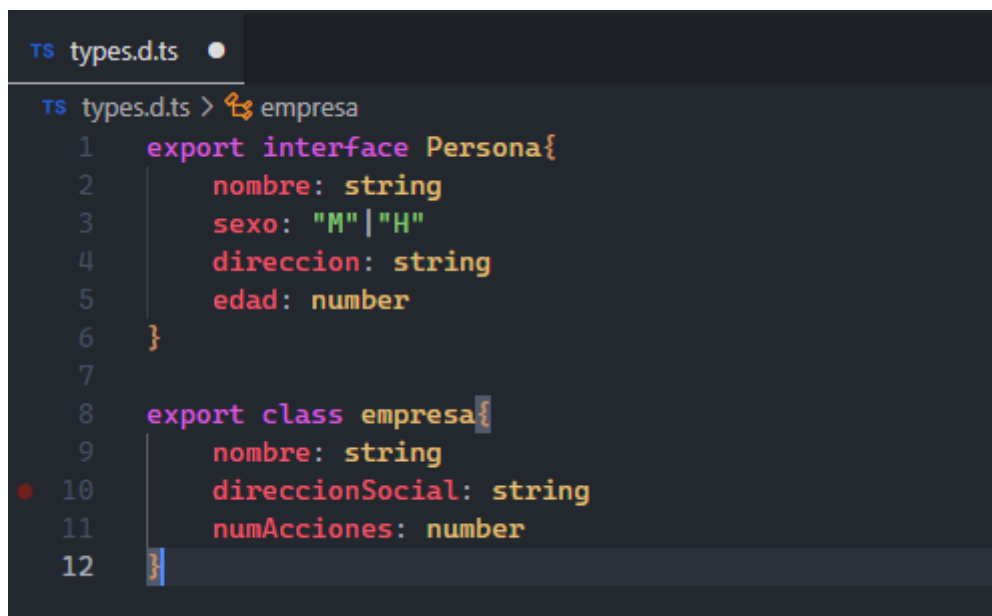
```
1  class Persona{
2      public nombre: "manolo"
3
4      constructor(){
5
6      }
7  }
8
9  const personaInstancia = new Persona();
10 personaInstancia.nombre;
```

Cómo podemos observar, al ser un atributo “public”, no dará ningún tipo de error al tratar de acceder a esa propiedad desde fuera de clase.

20. Convención d.ts.

Al usar aplicaciones que usen Typescript, será frecuente encontrarnos con archivos que tengan la convención d.ts.

Esta forma de definir un archivo Typescript, nos indicará que dentro de este fichero tendremos definidos interfaces, clases o tipos de Typescript que son exportados de manera que los podamos usar en cualquier lugar de nuestra aplicación. Un ejemplo de un archivo.d.ts sería el siguiente:



```
TS types.d.ts •
TS types.d.ts > empresa
1  export interface Persona{
2      nombre: string
3      sexo: "M"|"H"
4      direccion: string
5      edad: number
6  }
7
8  export class empresa{
9      nombre: string
10     direccionSocial: string
11     numAcciones: number
12 }
```