

Curso

TypeScript: Tu completa guía y manual de mano

Instructor: Fernando Herrera

Development > Web Development > Typescript

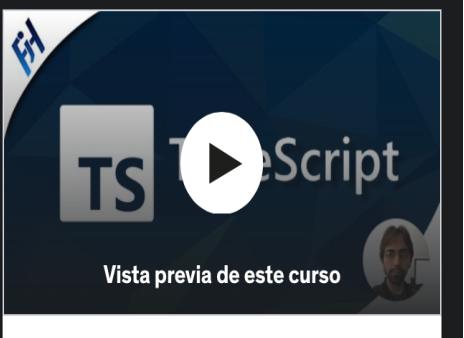
TypeScript: Tu completa guía y manual de mano.

La base sólida que necesitas para trabajar con TypeScript

4,9 ★★★★★ (5.815 calificaciones) 18.347 estudiantes

Creado por [Fernando Herrera](#)

Última actualización: 4/2023 [Español](#) [Español \[automático\]](#)



Vista previa de este curso

Lo que aprenderás

- ✓ Tener una base sólida de TypeScript.
- ✓ Saber utilizar TypeScript para mejorar la manera de programar en la web.
- ✓ Crear programas con una única importación de un sólo archivo de JavaScript.
- ✓ Tener la confianza de saber usar TypeScript en tus proyectos web.
- ✓ Tener el conocimiento para utilizar importadores de módulos.
- ✓ No cometer errores en jQuery, utilizando TypeScript.

Contenido del curso

14 secciones • 105 clases • 8 h 47 m de duración total

[Ampliar todas las secciones](#)

Garantía de reembolso de 30 días

Este curso incluye:

- ▶ 8,5 horas de vídeo bajo demanda
- 📄 25 artículos
- 📁 9 recursos descargables
- 💻 Acceso en dispositivos móviles y TV
- ♾️ Acceso de por vida
- 🖨 Certificado de finalización

[Compartir](#) [Regalar este curso](#)

Contenido del Curso.

▼ Introducción a TypeScript	4 clases • 24 min
▼ Introducción a TypeScript	5 clases • 25 min
▼ Tipos básicos	16 clases • 1 h 27 min
▼ Funciones y objetos	8 clases • 41 min
▼ Objetos y tipos personalizados en TypeScript	10 clases • 30 min
▼ Depuración de Errores y el archivo tsconfig.json	6 clases • 25 min
▼ Características de ES6 o JavaScript2015 disponibles a través TypeScript	7 clases • 41 min
▼ Clases en TypeScript	9 clases • 55 min
▼ Interfaces	9 clases • 34 min
▼ NameSpaces	7 clases • 39 min
▼ Genéricos - Generics	9 clases • 51 min
▼ Decoradores	8 clases • 48 min
▼ Usando librerías que no están escritas en TypeScript (Como jQuery)	4 clases • 25 min
▼ Final del curso	3 clases • 4 min

CONTENIDO

Sección 1: Introducción

1. Introducción a TypeScript

Cuando JavaScript fue concebido no fue pensado en ser un lenguaje de programación robusto. Con el paso del tiempo JavaScript creció en gran manera, por lo cual, es muy común que un proyecto tenga miles de líneas y varios miembros en un equipo de desarrollo.

Debido a la flexibilidad de JavaScript puede ser complicado comprender y manejar las variables, funciones, objetos, módulos, etc.

Aquí es donde entra TypeScript para alinear el desarrollo añadiendo Tipado al proyecto, documentación, buenas prácticas, etc.

En muchos casos los errores podrían no ser detectados por el editor y el linter, pero eso no pasará con TypeScript.



TypeScript permite utilizar lo más actual de JavaScript (ESNext) para luego transpilar el código a JavaScript estándar (JS) y ser ejecutado por el navegador con la compatibilidad deseada.

Con TypeScript se pueden hacer aplicaciones web, aplicaciones móviles, backend, aplicaciones que estarán alojadas en la nube, etc.

- 2. ¿Cómo funcionará el curso?**
- 3. ¿Cómo hacer preguntas?**

4. Instalaciones necesarias

Enlace en Github: <https://gist.github.com/Klerith/384b707f9b08698655280a3d4cc4da12>

- Node JS
- Visual Studio Code
- Google Chrome

Extensiones de VSCode

- [Activitus Bar](#)
- [TypeScript importer](#)

Extensiones de Chrome

- [Json Viewer Awesome](#)

5. ¡Únete a Nuestra Comunidad de DevTalles en Discord!



Te invitamos a que formes parte de nuestra comunidad de DevTalles en Discord, un espacio donde tendrás la oportunidad de establecer conexiones con otros estudiantes, compartir y colaborar.

¿Cómo unirse?

- Haz clic en el siguiente enlace de invitación: <https://discord.io/DevTalles>
- Una vez dentro, cuéntanos un poco de ti en el canal de bienvenida(#presentate).

Estamos entusiasmados de tener nuevos miembros y crecer juntos como comunidad.

¡Esperamos verte pronto en Discord!

Atentamente,

El equipo de DevTalles

Sección 2: Introducción a TypeScript

6. Introducción a la sección

En esta sección comenzaremos nuestros primeros pasos para comprender TypeScript y su sintaxis, pero nuevamente es básicamente JavaScript con tipado de variables, funciones, clases y nuevos tipos que no existen en JavaScript.

TypeScript, ayuda mucho a cometer menos errores de programación por el costo de más código y tiempo de desarrollo, pero lo recuperamos a la hora de refactorizar o encontrar errores en nuestro programa a la hora de escribirlo.

En esta sección vamos a realizar ejercicios iniciales, exposiciones y generalidades que nos permitan seguir trabajando en el curso.

7. Instalación de TypeScript

La instalación se hará de manera global, aunque normalmente se instala basado en el proyecto.

- Instalación basada en un proyecto: **npm install typescript --save-dev**
- Instalación global: **npm install -g typescript**
- Para comprobar la versión de TypeScript instalada: **tsc --version** ó **tsc -v**

8. Hola Mundo en TypeScript

- Crear el archivo HTML de entrada, en este caso: **index.html**
- Crear el archivo TypeScript, en este caso: **app.ts**
- Agregar el archivo **app.ts** en el **index.html**:

```
<body>
    <script type="text/typescript" src="app.ts"></script>
</body>
```

Para transpilar el archivo app.ts de TypeScript a JavaScript:

- **tsc app.ts**

Ciclo de desarrollo en TypeScript:

1. Se crea el proyecto en TypeScript
2. Se transpila el código de TypeScript a JavaScript
3. Se ejecuta el código JavaScript generado

9. TSConfig.json

El archivo de configuración de TypeScript es: **tsconfig.json** y se crea de la siguiente manera:

- **tsc --init**

```
→ Bases tsc --init

Created a new tsconfig.json with:

target: es2016
module: commonjs
strict: true
esModuleInterop: true
skipLibCheck: true
forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig
→ Bases
```

Cuando se ha creado el archivo de configuración, para hacer la transpilación, solo basta ejecutar: **tsc**

10. Modo observador – Watch mode

TypeScript trabaja transpilando el código a su sinónimo en JavaScript. Para no hacer la transpilación en cada momento, se utiliza el **modo observador**, el cual detecta cualquier archivo de TypeScript que cambie y hará la transpilación de manera automática.

- **tsc --watch ó tsc --w**

Sección 3: Tipos básicos

11. ¿Qué veremos en esta sección?

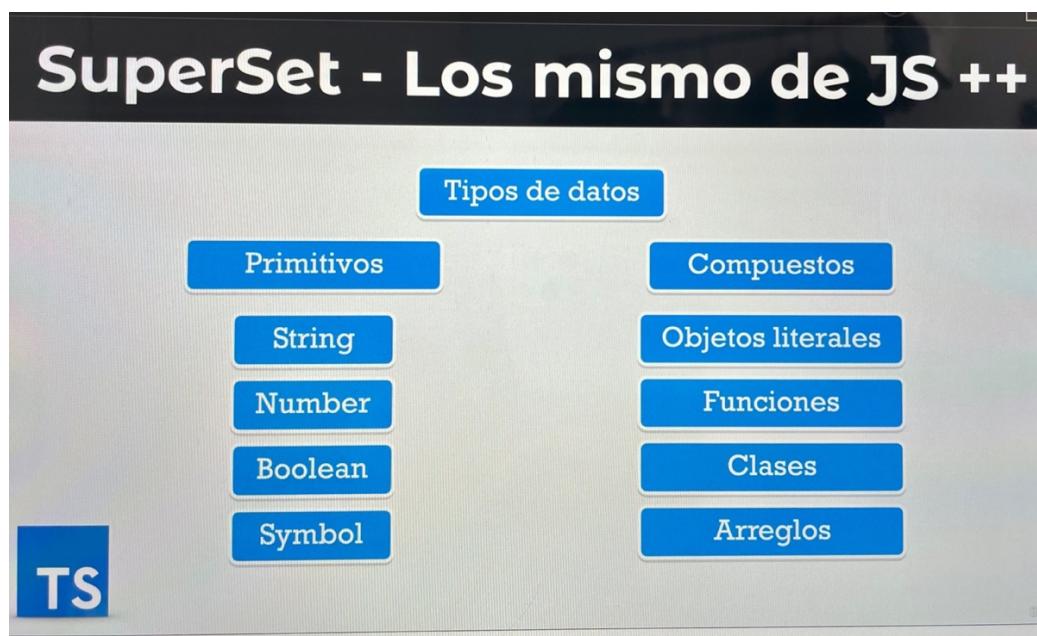
En esta sección aprenderemos:

- ¿Qué son los tipos de datos?
- Una introducción a los diferentes tipos de datos que existen en TypeScript
- Booleanos
- Números
- Strings
- Tipo Any
- Arreglos
- Tuplas
- Enumeraciones
- Retorno void
- Null
- Undefined

Y al final un examen práctico y seguidamente un examen teórico.

12. Introducción a los tipos de datos

Existen más tipos de datos en TypeScript que en JavaScript. Los tipos de datos en JS son **Primitivos** y **Compuestos**:



Todos estos tipos de datos los podemos usar de la misma manera en TypeScript, son lo mismo que en JS. Sin embargo, TypeScript nos permite:

- Crear nuevos tipos
- Interfaces
- Genéricos
- Tuplas

13. Más información sobre los tipos de datos

A continuación, explicaremos todos los tipos de datos que soporta TypeScript uno por uno. Si desean tener más información, pueden ver la documentación oficial de TypeScript sobre los tipos de datos aquí:

- [Documentación oficial](#)

14. Inferir tipos y modo estricto

En TS cuando se crea una variable lo que va después de los (:) es el tipo de dato:

```
let b: number = 10;
```

Sin embargo, cuando se crea una constante, su tipo lo define por su contenido:

```
const a = 10;
```

En este caso, el tipo de dato de **a** es 10: `const a: 10`

Y cuando se crea una variable TS infiere el tipo de dato por su contenido:

```
let b = 10;
```

En este caso, el tipo de dato de **b** es **number**: `let b: number`

Nota: Al declarar variables, no se debe dejar que TS infiera el tipo de dato, sino que debe ser establecido. Si no se asigna un valor inicial a la variable, el tipo de dato será **any**: `let c: any`

```
let c;
```

El crear variables sin un valor inicial nos da **flexibilidad**, pero perdemos **control**, y eso es un riesgo importante. Además, al crear una variable sin valor inicial llega a permitir que el tipo de dato mute sin control de errores:

```
let c;
c = 'A';
c = [];
c = {};
c = true;
```

En este caso, aunque (`c`) tiene contenidos de diferentes tipos, su tipado sigue siendo **any**.

Además, cuando se crea una función sin establecer el tipo de dato de los argumentos, estos implícitamente serán establecidos como **any**:

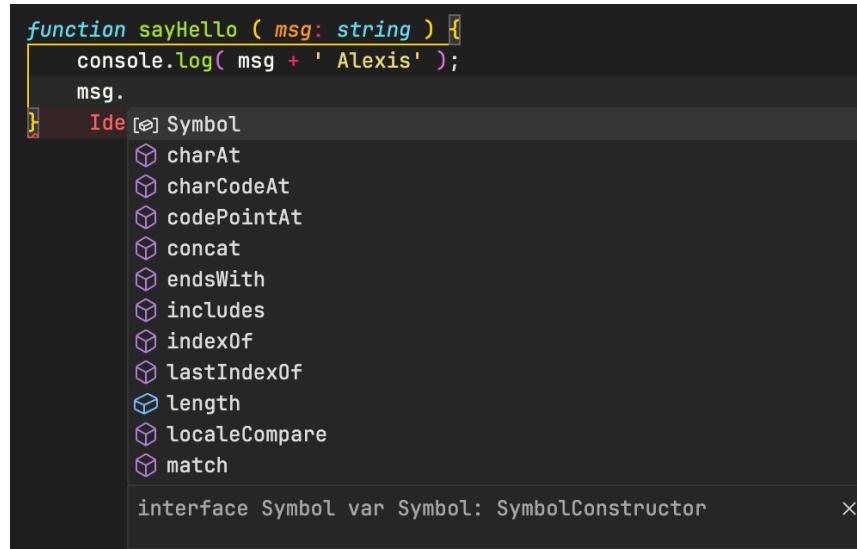
```
function sayHello ( msg ) { console.log( msg ) }
```

Esto es marcado como un error porque por defecto, en el archivo **tsconfig.json** la propiedad esta activada:

```
"noImplicitAny": true, /* Enable error reporting for expressions and declarations with an implied 'any' type. */
```

Otro inconveniente al no establecer el tipo de dato de los argumentos de la función es el hecho de que TS no dará ninguna ayuda de autocompletado porque no puede determinar que tipos de métodos se pueden utilizar con ese argumento **any**: `msg`.

El tipo de dato se establece con (`:`) después del nombre del argumento, lo cual brinda muchos beneficios:



Al crear una variable asignando el tipo de dato, TS en ciertos casos marcará un error cuando se use la variable:

```
let isAdmin: boolean;
let isLoggedIn: boolean = isAdmin ? true : false;
```

```
Variable 'isAdmin' is used before being assigned.ts(2454)
```

15. Booleans – Booleanos

El tipo de dato **boolean** en el modo estricto de TS solo acepta los valores de **true** o **false**. En el modo no estricto podría recibir null, undefined, etc., por lo cual, no es aconsejable desactivar el modo estricto de TS.

```
let isAdmin: boolean = true;
```

Al establecer el tipo de dato boolean, TS hará validaciones en todo tipo, mostrando errores como en este caso:

```
let isLogin: boolean = isAdmin ? true : 'false';
```

```
Type 'string' is not assignable to type 'boolean'.ts(2322)
```

16. Numbers – Números

Gracias al uso del tipo de dato **number** TS ayudará a validar casos inesperados con el manejo de los números:

```
let empleados;
const developers = 20;

if (empleados < developers) {
    console.log('Disponibilidad');
} else{
    console.log('No vacantes');
}
```

```
Este es el error que TS detectará: 'empleados' is possibly 'undefined'.ts(18048)
```

Es importante saber que NaN en JS es considerado como un número, por lo que, no marcará un error y podrá ser validado dando resultados inesperados, sin confirmar el valor de la variable utilizada puede llevar a errores:

```
empleados = Number('55A');
```

```
console.log(empleados); → NaN
```

Este tipo de errores no son de sintaxis sino errores de lógica.

17. Strings - Cadenas de caracteres

En TS existen diferentes maneras de crear una variable de tipo String: (" "), (' '), (` `)

```
const admin: string = 'Alexis';
const user: string = "alx";
const anonimo: string = `Anonimo`;
```

Al crear una variable con el tipo de dato **any** se perderá el autocompletado de los métodos para String.

Cuando trabajamos con String es importante tener presente el caso cuando sea undefined para evitar errores de lógica: **Cannot read properties of undefined (reading 'toUpperCase')**

```
console.log( admin[10].toUpperCase() );
```

Para controlar esto se puede usar el operador **optional chaining** (Encadenamiento opcional: **?**)

```
console.log( admin[10]?.toUpperCase() || 'Admin incorrecto' );
```

Salida: Admin incorrecto

18. Tipo Any

Es recomendable utilizar lo menos posible el tipo de dato **any**. Cuando el tipo de dato es **any**, no se proporcionará la ayuda de autocompletado porque TS no puede determinar el tipo de dato de la variable, aunque tenga un valor de cierto tipo de dato.

```
let codigo: any = 123;
```

Cuando un valor es de tipo **any** pero desea tratarse de una manera particular, se utiliza el casting de TS:

```
codigo = 'ABC';
console.log( (codigo as string).charAt(0) );
```

De esta manera, TS tratará la variable código como un string independientemente del tipo de dato **any**, por lo que también proporcionará la ayuda de autocompletado de TS para string.

Esta es otra forma de hacer casting en TS:

```
codigo = 123.3555;
console.log( <number> codigo ).toFixed(2) );
```

19. Arrays – Arreglos

En TS el símbolo **[]** normalmente significa que es un arreglo, pero también puede indicar una tupla, trio, cuarteto, etc.

Arreglo de números: `const numbers: number[]`

```
const numbers = [1,2,3,4,5,6,7,8,9,10];
```

Si cambiamos un elemento a otro tipo, cambia la definición del arreglo: `const numbers: (string | number)[]`

```
const numbers = [1,2,3,4,5,6,'6',8,9,10];
```

La definición del arreglo anterior nos indica que contiene caracteres o números, por lo tanto, solo permitirá añadir de este tipo de elementos.

Como el arreglo es válido, TS nos dará la ayuda de autocompletado para arreglos.

En el caso de que necesitemos añadir otro tipo de elementos, esto debe especificarse en la definición del arreglo:

```
const numbers: (string | number | boolean)[] = [1,2,3,4,5,6,'6',8,9,10];
```

En este caso el arreglo aceptara caracteres, números y boléanos.

Cuando se define el tipo de elementos del arreglo, TS proporciona los métodos y el autocompletado correcto para trabajar con el arreglo, por ejemplo, en el uso de `forEach` sabrá qué tipo de datos recorrerá.

TS previene muchos errores que en JS serían procesados como válidos, por lo cual, TS hace que el código sea más confiable.

20. Tuples – Tuplas

En JS no existen las Tuplas, esto es un complemento de TS. El contenido de estos elementos puede ser un trío, cuarteta, etc.

Para definir una tupla donde el 1er elemento siempre será un string y el 2º elemento siempre será un número:

```
const perfiles: [ string, number ] = ['Admin', 1];
```

¿Cómo podemos saber que es una tupla? Es por los corchetes: `const perfiles: [string, number]`

Podemos ver la diferencia con la definición anterior del arreglo y sus diferencias con la tupla:

`const numbers: (string | number)[] → Arreglo`

`const perfiles: [string, number] → Tupla`

En este caso se separan por una coma (,) y no por el pipe (|) como en los arreglos.

En JS no habrá ninguna diferencia, solo se aplican las reglas de tuplas en TS. Ejemplo:

```
perfiles[0] = 1;
```

Esta asignación en JS es válida, pero en TS se marcará como un error:

```
Type 'number' is not assignable to type 'string'.ts(2322)
```

Con las tuplas en TS podemos controlar que tipo de datos recibirá un arreglo y cuidar su integridad.

Sin embargo, aunque se llama Tupla, puede contener más de 2 elementos:

```
const perfiles: [ string, number, boolean ] = ['Admin', 1, true];
perfiles[0] = 'Anonimo';
perfiles[1] = 3;
perfiles[2] = false;
```

En esta definición, vemos que hay 3 valores en la tupla. Por eso, una tupla puede ser trio, cuarteto, etc.

21. Enum - Enumeraciones

Las Enumeraciones se definen en mayúsculas usando la casing **UpperCamelCase ó Pascal Case**. Este es un tipo de archivo que solo existe en TS, este tipo de archivo ayuda a trabajar con valores que tengan un sentido semántico visualmente fácil de leer. Al usar enumeraciones, podemos establecer valores:

```
enum AudioLevel {
    min = 1,
    medium,
    max = 10
}
```

Aquí el valor de médium no será la mitad sino el siguiente valor después de min porque es una numeración: 2

```
let currentAudio = AudioLevel.medium;
```

```
console.log( currentAudio );
```

Es posible hacer una asignación directa solo cuando el valor ingresado cae dentro de los valores min, medium o max, en este caso, los valores directos posibles a ingresar serían: 1, 2, 10

```
let currentAudio: AudioLevel = 2;
```

En otro caso dará un error cualquier asignación directa: Type '9' is not assignable to type 'AudioLevel'.ts(2322)

```
let currentAudio: AudioLevel = 9;
```

La manera correcta de asignar valores o utilizarlos es con las variables: min, medium y max.

22. Void - Vacío

En TS Void se utiliza para indicar que no hay un valor de retorno: `const result: void`

```
function callPerfiles(){  
}  
  
const result = callPerfiles();  
console.log(result);
```

Debemos recordar que en JS, **undefined** no es lo mismo que **null**: `undefined === null` -> false

Cuando una función no retorna nada, es correcto especificar el tipo void:

```
function callPerfiles(): void {  
}
```

Con esta definición, si se intenta hacer un return TS marcará un error: `Type 'number' is not assignable to type 'void'.ts(2322)`

```
function callPerfiles():void{  
    return 1;  
}
```

Si el return no contiene valor, se considera como Void y será válido.

Para especificar el tipo de retorno **Void** en una función de flecha será el siguiente:

```
const callPerfiles = (): void => {  
}
```

Esta definición de Void, ayuda mucho a la documentación y lectura de las funciones.

23. Never – Nunca

El tipo de dato never no es igual a null ni a undefined y se usa para especificar que el valor de retorno de una función no debe terminar exitosamente y no permite que continue la ejecución del código, generando un error:

```
const error = ( message: string ): never => {  
    throw new Error( message );  
}  
  
error('Auxilio');  
console.log('Hola mundo');
```

```
✖ ▶ Uncaught Error: Auxilio never.js:4
  at error (never.js:4:15)
  at never.js:6:5
  at never.js:8:3
```

>

El tipo de dato `never` no debe tener un punto alcanzable para terminar la función, aunque se puede especificar que existan otras posibilidades:

```
const error = ( message: string ): ( never | number ) => {
  if ( false ) {
    throw new Error( message );
  }
  return 1;
}

error('Auxilio');
console.log('Hola mundo');
```

Hola mundo

never.js:10

>

Normalmente un tipo de dato `never` terminará en un error.

24. Null y Undefined

El tipo de dato `Null` y `Undefined` podrían ser permitidos en JS pero en TS en modo estricto podría no ser permitido.

```
let nada: undefined = undefined;
console.log( nada );
```

undefined

null-undefined.js:4

>

Si se intentará asignar un valor `undefined` a un tipo de dato diferente en las nuevas versiones de TS no sería permitido:

```
let isActive: ( boolean | undefined ) = undefined;
console.log( isActive );
```

La única manera para que se aceptado es especificar que puede ser de tipo `undefined` además de los otros tipos a utilizar.

En el archivo **tsconfig.json** existe una propiedad llamada "**strictNullChecks**": **true**, que por defecto esta activada y sirve para permitir o no asignaciones de undefined. Si se desactiva, entonces la siguiente línea no será un error:

```
let isActive: boolean = undefined;
```

No se recomienda desactivar esta propiedad. El error mostrado cuando esta activada la propiedad será:

```
Type 'undefined' is not assignable to type 'boolean'.ts(2322)
```

Lo mismo pasará si se asigna el valor de null: **Type 'null' is not assignable to type 'boolean'.ts(2322)**

```
let isActive: boolean = null;
```

Por esto, se puede decir que en teoría **boolean** puede aceptar 4 tipos de valores: **true**, **false**, **undefined** y **null**.

Debes recordar siempre que null es diferente de undefined.

25. Ejercicio práctico #1

26. Tarea y Resolución del Ejercicio #1

Usando el archivo adjunto **app.ts** realizar los ejercicios solicitados, a continuación, se muestran las soluciones:

```
((() => {

    // Tipos
    const batman: string = 'Bruce';
    const superman: string = 'Clark';
    const existe: boolean = false;

    // Tuplas
    const parejaHeroes: [ string, string ] = [batman,superman];
    const villano: [ string, number, boolean ] = ['Lex Lutor',5,true];

    // Arreglos
    const aliados: string[] = ['Mujer Maravilla', 'Acuaman', 'San', 'Flash'];

    // Enumeraciones
    enum Fuerza {
        acuaman = 0,
        batman = 1,
        flash = 5,
        superman = 100
    }
})()
```

```

const fuerzaFlash: Fuerza = Fuerza.flash;
const fuerzaSuperman: Fuerza = Fuerza.superman;
const fuerzaBatman: Fuerza = Fuerza.batman;
const fuerzaAcuaman: Fuerza = Fuerza.acuaman;

// Retorno de funciones
function activar_batiseñal(): string {
    return 'activada';
}

function pedir_ayuda(): void {
    console.log('Auxilio!!!!');
}

// Aserciones de Tipo
const poder: any = '100';
const largoDelPoder: number = (poder as string).length;
console.log(largoDelPoder);

})()

```

Cuestionario 1: Examen teórico #1

Pregunta 1:

¿Quién es el fundador de TypeScript?

Pregunta 2:

¿Cómo se define un arreglo de Strings en TypeScript?

Pregunta 3:

¿El siguiente código es válido en TypeScript?

1. let arr:string[] = ["Texto", "Texto", "Texto", "Texto"];
2. arr.push("10");

Pregunta 4:

¿El siguiente código es válido en TypeScript?

```
let arr:number = [1,2,3,4,5,6,7,8,9,10];
```

Pregunta 5:

¿El siguiente código es válido en TypeScript?

```
let arr:any = [1,2,3,4,5,6,7,8,9,10];
```

Pregunta 6:
¿Qué es esto?

```
let variable:[number,string,boolean] = [10,"texto",true];
```

Pregunta 7:
¿El siguiente código es una declaración válida de un string?

1. `let string = `1.`
2. `2.`
3. `3.`
4. `4.`
5. `5.`
6. `6.`;`

Pregunta 8:
¿El siguiente código es válido en TypeScript?

```
let vacio:null = undefined;
```

Pregunta 9:
Dada la siguiente enumeración, que valor tiene "C"

1. `enum enumeracion {`
2. `a,`
3. `b,`
4. `c,`
5. `d`
6. `}`

Pregunta 10:
Dada la siguiente enumeración, ¿Qué valor tiene "d"?

1. `enum enumeracion {`
2. `a=10,`
3. `b,`
4. `c=9,`
5. `d`
6. `}`

Sección 4: Funciones y objetos

27. ¿Qué veremos en esta sección?

Esta sección está enfocada en aprender cómo trabajan las funciones en TypeScript y también nos enfocaremos en aplicar buenas prácticas a la hora de crearlas.

Puntualmente tenemos:

1. Declaraciones básicas de funciones
2. Parámetros obligatorios
3. Parámetros opcionales
4. Parámetros por defecto
5. Parámetros REST
6. Tipo de datos "Function"

Al final de la sección, tendremos el examen práctico y el examen teórico.

28. Funciones básicas

Esta es la función más básica posible que no retorna nada: `function(): void`.

```
(  
  ()=>{  
    }  
)();
```

Después del nombre de la función, se especifica el tipo de retorno de la función, `() : string { }`

```
const isActive = (): string => {  
  return 'Activate';  
}
```

TS también infiere el tipo de dato de retorno, en este caso es `string`: `const isActive: () => string`

```
const isActive = () => {  
  return 'Activate';  
}
```

Sin embargo, no se recomienda dejar a TS que infiera el tipo de retorno, sino que debe ser especificado. Esto ayudará a dar información sobre el tipo de dato que tendrá una variable que utilice el valor de retorno:

```
const perfil = isActive();
```

Al crear la variable `perfil` con el valor de retorno de la función `isActive()`, TS define a la variable `perfil` de tipo `string`: `const perfil: string`

Esto brinda integridad en el desarrollo e incluso autocompletado según el tipo de dato.

29. Parámetros obligatorios de las funciones

Al declarar la siguiente función, TS marca un error porque esta activada la función de no permitir parámetros de tipo any:

```
const fullName = ( firstName, lastName ): string => {
    return `${ firstName } ${ lastName }`;
}
fullName('Alexis','Sandoval');
```

Para eliminar el error, basta con especificar el tipo de dato de los parámetros, sin embargo, esto hace obligatorio el uso de estos. Al llamar la función marcará un error al no pasar un parámetro como en el siguiente caso:

```
const fullName = ( firstName:string, lastName:string ): string => {
    return `${ firstName } ${ lastName }`;
}
fullName('Alexis');
```

Para resolver este error, podría ser especificar que un parámetro puede ser null, undefined, o boolean y manejar la ausencia del argumento en la función:

```
const fullName = ( firstName:string, lastName:string|undefined ): string => {... }
fullName('Alexis', undefined);
```

La manera correcta de manejar los parámetros opcionales los veremos en el siguiente punto, por lo pronto, en conclusión, siempre se debe especificar el tipo de dato del argumento y buscar que sea de un solo tipo de dato.

30. Parámetrosopcionales de las funciones

Para los parámetros opcionales se utiliza el símbolo (?) en la definición de la función y se maneja en el interior de la función la ausencia del argumento:

```
const fullName = ( firstName:string, lastName?:string ): string => {
    return `${ firstName } ${ lastName || '---' }`;
}
const name = fullName('Alexis');
```

El operador **optional chaining** (Encadenamiento opcional: ?) va después del nombre del parámetro.

31. Parámetros por defecto

En las nuevas versiones de TS un argumento requerido no puede ir después de uno opcional:

```
const fullName = ( firstName:string, lastName?:string, upper:boolean ): string => {
Este es el error que marca: A required parameter cannot follow an optional parameter.ts(1016)
```

Para resolver este error, se pueden reacomodar los parámetros o hacer que el parámetro siguiente sea por defecto y para especificar parámetros por defecto se utiliza el símbolo (=):

```
const fullName = ( firstName:string, lastName?:string, upper:boolean = false ): string => {

    if ( upper ) {
        return `${ firstName } ${ lastName || '---' }`.toUpperCase();
    } else {
        return `${ firstName } ${ lastName || '---' }`;
    }

}

let name = fullName('Alexis', 'Sandoval', true);
```

32. Parámetros REST

Cuando no se conoce la cantidad total de parámetros que una función va a recibir, se usan los REST Arguments especificando su tipo de datos que contendrá porque implícitamente TS lo toma como any, lo cual, es un error en el modo estricto de TS:

```
const fullName = ( firstName: string, ...restArgs: string[] ) => {
    return `${ firstName } `;
}

const perfil = fullName('Alexis', 'Sandoval', 'alxsandoval');
```

En este caso, se pueden unir los argumentos restantes con los métodos de los arreglos: join()

```
return `${ firstName } ${ restArgs.join(' ') } `;
```

Existe una regla de oro que es: “Si se puede hacer en JS, se puede hacer en TS”

33. Tipo Función (Function)

El tipo de dato función se utiliza cuando deseamos que una variable o constante acepte únicamente funciones:

```
let myFunction: Function;
```

Es importante notar que el tipo de dato se especifica con Mayúscula inicial (**Function**).

También se puede definir el tipo de dato de los parámetros que recibirá y el valor de retorno de la función para que acepte únicamente definiciones de funciones que cumplan con esa regla establecida:

```
let myFunction: (cualquierNombre:number, nombreDescriptivo:number) => number;
```

Veamos un ejemplo de asignación a una variable de tipo función, primero mostraremos 3 funciones:

```
const addNumber = ( a: number, b: number ) => a + b;
const greet = ( name: string ) => `Hola ${ name }`;
const msg = () => `¡Sesión iniciada!`;
```

Ahora asignamos a una variable una función que cumpla con la regla establecida, en este caso, es addNumber:

```
myFunction = addNumber;
```

Ejecutamos la función y obtenemos el resultado esperado:

```
console.log( myFunction( 1, 2 ) );
```

Para que la variable myFunction sea válida con la 2^a función, se debe cambiar su definición:

```
let myFunction: (cualquierNombre: string) => string ;
myFunction = greet;
console.log( myFunction('Alexis') );
```

Para que la variable myFunction sea válida con la 3^a función, se debe cambiar su definición:

```
let myFunction: () => void ;
myFunction = msg;
console.log( myFunction() );
```

La parte más importante en este punto es poder leer la definición de la función desde la ayuda del editor.

34. Tarea y Resolución del ejercicio práctico #2

Usando el archivo adjunto **app.ts** realizar los ejercicios solicitados, a continuación, se muestran las soluciones:

```
// Funciones Básicas
function sumar( a: number, b:number ): number {
  return a + b;
}

const contar = ( heroes: string[] ): number => {
  return heroes.length;
}

const superHeroes: string[]  = ["Flash", "Arrow", "Superman", "Linterna Verde"];
contar( superHeroes );

//Parametros por defecto
const llamarBatman = ( llamar: boolean = true ): void => {
  if( llamar ){
    console.log("Batiseñal activada");
  }
}

llamarBatman();
```

```
// Rest?
const unirheroes = ( ...personas:string [] ): string => {
  return personas.join(", ");
}

// Tipo funcion
const noHaceNada = ( numero: number, texto: string, booleano: boolean, arreglo:string[] ): void => {

}

// Crear el tipo de funcion que acepte la funcion "noHaceNada"
let noHaceNadaTampoco: ( n:number, s:string, b:boolean, a:string[] ) => void;
noHaceNadaTampoco = noHaceNada;
```

Cuestionario 2: Examen teórico #2

Pregunta 1:

¿Toda función en JavaScript, es código válido de TypeScript?

Pregunta 2:

¿La siguiente función válida en TypeScript?

1. `function saludar():string{`
2. `2.`
3. `3. console.log("Hola mundo!");`
4. `4.`
5. `5. }`

Pregunta 3:

¿En TypeScript es posible obligar al desarrollador que debe de cumplir todos los parámetros de una función?

Pregunta 4:

¿En JavaScript, todos los parámetros son obligatorios?

Pregunta 5:

¿Con qué carácter específico un parámetro opcional?

Pregunta 6:

¿Qué es un parámetro por defecto?

Pregunta 7:

¿Los parámetros por defecto sólo pueden ser tipos primitivos?

Pregunta 8:

¿Qué imprime en consola el siguiente código de TypeScript?

1. `1. function saludar(mensaje:string = "mundo"){`
2. `2.`
3. `3. console.log("Hola " + mensaje);`

```
4.  
5. }  
6.  
7. saludar("holo");
```

Pregunta 9:

¿Qué es un parámetro REST?

Pregunta 10:

¿Una función es a su vez, un tipo en TypeScript?

Sección 5: Objetos y tipos personalizados en TypeScript

35. ¿Qué veremos en esta sección?

Aprenderemos a utilizar los objetos en TypeScript, su uso y mantener nuestro código bien limpio mediante tipos personalizados.

Los temas serán:

1. Objetos básicos
2. Crear objetos con tipos específicos
3. Crear métodos dentro de objetos
4. Tipos personalizados
5. Crear variables que soporten varios tipos a la vez.
6. Comprobar el tipo de un objeto.

Al final, el respectivo examen práctico y teórico.

36. Objetos básicos

En JS está permitido que una vez definido un objeto se puedan agregar propiedades, sin embargo, en TS eso no está permitido, veamos un ejemplo:

```
let persona = {  
    name: 'Alexis',  
    age: 37,  
    rol: ['Admin', 'DB']  
}
```

```
persona = {
    username = 'alxsandoval'
}
```

TS aquí marca que **username** no es un tipo assignable al objeto persona y la razón es porque no existe esa propiedad en el objeto:

```
Type '{ username: string; }' is not assignable to type '{ name: string; age: number; rol: string[]; }'.
Object literal may only specify known properties, and 'username' does not exist in type '{ name: string; age: number; rol: string[]; }'.ts(2322)
```

Aunque lo anterior no está permitido en TS, si es posible reemplazar el objeto siempre y cuando todas las propiedades tengan un valor, como en el siguiente caso:

```
let persona = {
    name: 'Alexis',
    age: 37,
    rol: ['Admin', 'DB']
}

persona = {
    name: 'Sandoval',
    age: 27,
    rol: ['user']
}
```

Por lo anterior, es muy recomendable siempre pensar en el objeto inicial al definirlo para evitar inconvenientes futuros como querer añadir propiedades o incluso métodos que no existen inicialmente. Otra alternativa es crear objetos con tipos específicos como veremos a continuación.

37. ¿Cómo crear objetos con tipos específicos?

Primero debemos saber que el tipo de dato de las propiedades de un objeto se definen después del nombre y los (:), veamos:

```
let persona: { name: string, age: number, rol: string[] } = {
    name: 'Alexis',
    age: 37,
    rol: ['Admin', 'DB']
}
```

También se podría definir de manera muy genérica pero no es nada recomendable:

```
let persona: object = { ... }
let persona: {} = { ... }
```

En ocasiones las propiedades de un objeto se crean de forma alfabética, aunque también suelen crearse conforme van siendo utilizadas.

38. Métodos dentro de los objetos

Al definir los tipos de las propiedades de un objeto no necesariamente se debe hacer en el orden como están definidos, aunque si es una buena recomendación:

```
let persona: { name: string, getName?: () => string, age: number, rol: string[] } = { ... }
```

Cuando se definen métodos en las propiedades se establecen los parámetros y el retorno, veamos:

```
let persona: { name: string, age: number, rol: string[], getName?: () => string } = {
    name: 'Alexis',
    age: 37,
    rol: ['Admin', 'DB']
}
```

En este caso el método **getName** es opcional debido al operador (?) y este método no recibe nada (no se pone void) y retorna un string:

```
persona = {
    name: 'Sandoval',
    age: 27,
    rol: ['user'],
    getName(){
        return this.name;
    }
}
```

En la definición inicial el método **getName** no existe, pero fue especificado, por lo cual, se pudo añadir más adelante cuando se necesitó.

39. Problema con la definición en línea

Existe el inconveniente que cuando creamos un objeto y definimos los tipos de sus parámetros, pero en el futuro los cambiamos y otro objeto fue creado copiando la estructura o definición, entonces, tendríamos que cambiarlos en todo lugar donde se usaron, ejemplo:

```
let persona: { name: string, age: number, rol: string[], getName?: () => string } = {
    name: 'Alexis',
    age: 37,
    rol: ['Admin', 'DB']
}
```

Si ahora se debe cambiar la definición, por ejemplo, el rol ahora es un arreglo de números, entonces también se debe corregir su manejo interno para que funcione:

```
let persona: { name: string, age: number, rol: number[], getName?: () => string } = {
    name: 'Alexis',
    age: 37,
    rol: [1,2]
}
```

¿Cómo hacemos para evitar estos inconvenientes? Veamos una solución en el siguiente punto.

40. Tipos personalizados

En TS existe un tipo de dato llamado **type**, veamos su uso:

```
type persona = {
    name: string;
    age: number,
    rol: string[],
    getName?: () => string
}
```

Una vez definido el tipo **persona**, ahora podemos reescribir la lógica del programa:

```
let admin: persona = {
    name: 'Alexis',
    age: 37,
    rol: ['Admin', 'DB']
}

let user: persona = {
    name: 'Sandoval',
    age: 27,
    rol: ['User'],
    getName() {
        return this.name;
    },
}
```

Ahora si el rol cambia a un arreglo de números, ya no es necesario cambiar en otras definiciones sino solo en el manejo de la propiedad:

```
let admin: persona = {
    name: 'Alexis',
    age: 37,
    rol: [1,2] ← "Aquí se cambió el manejo de la propiedad"
}
```

```
let user: persona = {
  name: 'Sandoval',
  age: 27,
  rol: [3], ← Aquí se cambió el manejo de la propiedad
  getName() {
    return this.name;
  },
}
```

El uso de tipos específicos ayuda para tener un código más integro, mejor mantenimiento y más legible.

41. Múltiples tipos permitidos

Para asignar múltiples tipos permitidos se utiliza el símbolo (|) y pueden separarse o no los tipos por espacios y tener o no paréntesis, aunque eso ayuda a su mejor lectura:

```
let myCustomVariable: ( string | number | persona ) = 'Alexis';
```

En este caso TS es capaz de inferir que tipo de dato será myCustomVariable según el contenido que puede ser carácter, numérico o de tipo **object**.

Aunque en TS se diría que es de tipo **persona**, en JS no existe el **type persona**, por lo cual, se transpila como un objeto.

42. Ejercicio práctico #3

Descargue el material adjunto, trabaje con los tipos de datos y la información que aprendió en esta sección. Sea lo más específico en los tipos posibles y reutilice el primer tipo de dato (el del automóvil)

43. Tarea y Resolución del ejercicio práctico #3

Usando el archivo adjunto **app.ts** realizar los ejercicios solicitados, a continuación, se muestran las soluciones:

```
// Objetos

type Automovil = {
  carroceria: string,
  modelo: string,
  antibalas: boolean,
  pasajeros: number,
  disparar?: () => void
}
```

```
const batimovil: Automovil = {
  carroceria: "Negra",
  modelo: "6x6",
  antibalas: true,
  pasajeros: 4
};

const bumblebee: Automovil = {
  carroceria: "Amarillo con negro",
  modelo: "4x2",
  antibalas: true,
  pasajeros: 4,
  disparar() { // El metodo disparar es opcional
    console.log("Disparando");
  }
};

// Villanos debe de ser un arreglo de objetos personalizados
type Villano = {
  nombre: string,
  edad?: number,
  mutante: boolean
}

const villanos: Villano[] = [
  {
    nombre: "Lex Luthor",
    edad: 54,
    mutante: false
  },
  {
    nombre: "Erik Magnus Lehnsherr",
    edad: 49,
    mutante: true
  },
  {
    nombre: "James Logan",
    edad: undefined,
    mutante: true
  }
];

// Multiples tipos
// cree dos tipos, uno para charles y otro para apocalipsis

type Charles = {
  poder: string,
  estatura: number
}

const charles: Charles = {
  poder: "psiquico",
  estatura: 1.78
};
```

```

type Apocalipsis = {
  lider:boolean,
  miembros:string[]
}

const apocalipsis: Apocalipsis = {
  lider:true,
  miembros: ["Magneto","Tormenta","Psylocke","Angel"]
}

// Mystique, debe poder ser cualquiera de esos dos mutantes (charles o apocalipsis)
let mystique: ( Charles | Apocalipsis );
mystique = charles;
mystique = apocalipsis;

```

Cuestionario 3: Examen teórico #3

Pregunta 1:

¿Qué tipo de objeto es el batimovil?

1. var batimovil = {
2. puertas:10,
3. marca: "Sedan"
4. }

Pregunta 2:

¿Es posible agregar métodos dentro de los tipos?

Pregunta 3:

¿El siguiente código es válido en TypeScript?

1. let batimovil: { getNombre:()=> string } = {
- 2.
3. getNombre(carro){
4. return carro.toUpperCase();
5. }
- 6.
7. }

Pregunta 4:

¿Es posible especificar en TypeScript que una variable puede ser de 4 tipos a la vez?

Pregunta 5:

¿El siguiente código de TypeScript es válido?

1. let mutable: [string | string[]];
- 2.
3. mutable = ["Hola","Hola"];
4. mutable = "hola";

Pregunta 6:

¿El siguiente código es válido TypeScript?

1. `let mutable: number | string[];`
- 2.
3. `mutable = ["Adios","Hola"];`
4. `mutable = 123;`

Pregunta 7:

¿Qué instrucción nos permite saber que tipo de dato contiene una variable?

Pregunta 8:

¿Con qué palabra podemos crear tipos específicos?

Pregunta 9:

¿Un tipo de dato puede tener métodos obligatorios?

Pregunta 10:

¿Los tipos son traducidos a JavaScript?

44. Código fuente de la sección

Les dejo mi código fuente por si lo llegan a necesitar o comparar con el mío:

- [Github - Fin-seccion-5](#)

Sección 6: Depuración de Errores y el archivo tsconfig.json

45. ¿Qué veremos en esta sección?

La sección se enfoca en la depuración de errores y comprender el archivo de configuración de TypeScript (el tsconfig.json)

Puntualmente:

1. Aprenderemos el ¿por qué siempre compila a JavaScript?
2. Para que nos puede servir el archivo de configuración de TypeScript
3. Realizaremos depuración de errores directamente a nuestros archivos de TypeScript
4. Removeremos todos los comentarios en nuestro archivo de producción.
5. Restringiremos al compilador que sólo vea ciertos archivos o carpetas
6. Crearemos un archivo final de salida

7. Aprenderemos a cambiar la versión de JavaScript de salida

Adicionalmente tendrán el conocimiento necesario para compilar automáticamente cualquier archivo que se vaya creando al momento de ser insertado a nuestro proyecto.

46. ¿Qué es el archivo tsconfig y para qué nos puede servir?

Documentación del archivo tsconfig.json: <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

Inicialmente el archivo de configuración de TS viene con los valores por defecto recomendados. Si queremos que TS funcione de una manera específica se deben quitar los comentarios y activar o desactivar un valor.

Usualmente no se hacen muchas modificaciones en el archivo de configuración, pero valdría la pena conocer las más importantes o usadas.

Existe el inconveniente que, al ver la salida de la consola, los errores se marcan en el archivo JS y no en el archivo TS, para este y otros casos el archivo de configuración de TS nos puede ayudar.

47. ¿Es posible la depuración del código de TypeScript?

Para el proceso de depuración, en muchas ocasiones, no basta usar solo un console.log(), sino que se requieren otras herramientas.

En el archivo de configuración de TS existe el valor: “**sourceMap**”

```
// "sourceMap": true,          /* Create source map files for emitted JavaScript files. */
```

Cuando quitamos los comentarios y activador este valor, ahora cada archivo del proyecto tendrá un 3er archivo con extensión (*.map). Además, ahora la salida de la consola apuntará al archivo de TS:



Gracias a la activación del valor “**sourceMap**” de TS, ahora podemos trabajar con las herramientas de depuración del navegador el cual siempre apuntara al archivo de TS y ya no al archivo de JS.

48. Remover los comentarios de los archivos de JavaScript

Cuando desarrollamos con TS siempre debemos documentar el código, sin embargo, en muchas ocasiones no deseamos que esos comentarios se transpilen hacia JS, veamos cómo lograr esto:

En el archivo de configuración de TS (`tsconfig.json`), tenemos el valor: “**removeComments**”

```
// "removeComments": true,           /* Disable emitting comments. */
```

La opción recomendada para esto es (`true` - activado), así el archivo transpilado a JS no tendrá comentarios.

También los archivos (`*.map`) solo deberían existir en el modo desarrollo y no en el modo de producción, veamos a continuación como excluir estos archivos.

49. Incluir y excluir carpetas y/o archivos

En TS existe por defecto que los archivos de **node** (`node_modules`) sean excluidos en la transpilación. Sin embargo, podemos controlar los archivos que serán incluidos o excluidos a través del archivo de configuración:

```
}, <- /* Fin del archivo tsconfig.json */
"exclude": [
  "node_modules2" <- Carpetas o archivos que se desean excluir
],
"include": [
  "node_modules" <- Carpetas o archivos que se desean incluir
]
```

50. outFile – Archivo de salida

Cuando trabajamos con algún framework como React con TS, ya se tiene una estructura de carpetas y se hace la generación de un archivo de salida (**bundle**), pero también podemos hacerlo nosotros mismos y normalmente debemos generar un único archivo de JS.

En el archivo de configuración de TS (`tsconfig.js`) podemos definir el archivo de salida: “**outFile**”

```
// "outFile": "./",           /* Specify a file that bundles all outputs into one JavaScript file.
                                If 'declaration' is true, also designates a file that bundles all .d.ts output. */
```

En este ejemplo, deseamos que el archivo de salida sea: **main.js**

```
"outFile": "./main.js",
```

Cuando activamos la función aparece un error: `Only 'amd' and 'system' modules are supported alongside --outFile.ts`

Esto significa que el módulo de paquetes soportado por outFile es solo: **amd** y **system**:

```
"module": "commonjs",                                     /* Specify what module code is generated. */
```

Debemos cambiarlo a:

```
"module": "amd",                                         /* Specify what module code is generated. */
```

Debemos cuidar que en el “include” este el archivo especificado o de lo contrario, debemos quitar el **include**.

Al reiniciar el observador de TS (tsc --w), vemos que el archivo generado main.js tiene mucho código, y esto es porque unifica todos los archivos contenidos dentro del proyecto.

En el caso de haber creado por el observador archivos *.js y *.map, estos ya pueden ser eliminados para quedarnos solamente con los archivos de TS.

Al ver la salida de la consola, en caso de haber errores, estos serán dirigidos hacia los archivos TS y ya no a los archivos JS.

Sección 7: Características de ES6 o JavaScript2015 disponibles a través de TypeScript

51. ¿Qué veremos en esta sección?

JavaScript va actualizando año con año, y tenemos que estar enterados de todo lo nuevo para saber cómo le sacamos el máximo provecho.

Esta sección está orientada a enseñarles un par de cosas muy útiles y necesarias del ES6 (ES2015 o ECMAScript 6), que ya podemos utilizar con toda confianza en TypeScript.

Aprenderemos sobre:

1. Diferencia entre declarar variables con VAR y con LET
2. Uso de constantes
3. Plantillas literales

4. Funciones de flecha
5. Destructuración de objetos
6. Destructuración de Arreglos
7. Nuevo ciclo, el FOR OF
8. Conocer sobre la programación orientada a objetos
9. Clases

Al final, un examen práctico y teórico para afianzar los conocimientos.

52. Variables LET

JavaScript continuamente sigue actualizando la especificación del lenguaje, sin embargo, el estándar de JS más compatible es la ES5.

No se recomienda desarrollar usando la palabra reservada VAR por muchos problemas que genera. En lugar de VAR siempre se recomienda utilizar LET. Sin embargo, es muchísima mejor recomendación utilizar CONST para las variables y funciones a menos que sepamos que va a cambiar el valor se usaría LET.

Con respecto a las funciones, se recomienda utilizar CONST y la definición de Function Expression (Expresión de Función) ya sea como función anónima o función tipo flecha:

```
const myFunction = function () {
    console.log('Test');
}

const myFunction2 = () => {
    console.log('Test');
}
```

Esto ayudará a que **myFunction** no pueda ser remplazada por el uso del CONST.

53. Desestructuración de Objetos

En la actualidad, la mayoría de los desarrollos apuntan hacia el uso del ECMA Script 6 (ES6) por varias buenas mejoras, como la desestructuración que no existe en ES5.

El mecanismo de desestructuración permite extraer los valores de un objeto y al mismo tiempo crear una variable con el valor obtenido.

Es importante saber que el orden en que se extraen no es importante sino el valor de la propiedad y en el caso en que se quiera extraer un valor poniendo un nombre diferente de la propiedad, dará un error.

```
const perfiles = {
    admin: 'Alexis',
    user: 'Sandoval',
    anonimo: 'alxsandoval',
    active: true,
    total: 1000
}

const { total, active, user } = perfiles;
console.log( active, total, user.toUpperCase() );
```

Con la inferencia de TS, se proporciona ayuda de autocompletado y los métodos según el tipo de dato.

Cuando se hará la desestructuración de objetos, es recomendable crear un tipo de objeto (type) para continuar usando la inferencia de TS y la ayuda de autocompletado:

```
type perfil = {
    admin: string;
    user: string;
    anonimo: string;
    active: boolean;
    total: number;
}

const perfiles: perfil = {
    admin: 'Alexis',
    user: 'Sandoval',
    anonimo: 'alxsandoval',
    active: true,
    total: 1000
}

const printPerfiles = ( p: perfil ) => {
    console.log( p.admin ); // Ayuda de autocompletado
    console.log( p.user.toUpperCase() ); // Ayuda con los métodos
}

printPerfiles( perfiles );
```

También podemos hacer la desestructuración en la definición de la función:

```
const printPerfiles = ( { admin, user, ...resto }: perfil ) => {

    console.log( admin ); // Ayuda de autocompletado
    console.log( user.toUpperCase() ); // Ayuda con los métodos
    console.log( resto ); // Uso del operador ...rest

}
```

De esta forma ahorraremos un poco de código al utilizar directamente el nombre de la propiedad.

54. Desestructuración de Arreglos

La forma tradicional de trabajar antes de la incorporación de la desestructuración de arreglos era de la siguiente manera:

```
const perfilesArr = ['Admin', 'User', 'Anonimo'];
const user = perfilesArr[1];
console.log( {user} );
```

El uso de la forma anterior podría generar problemas de mantenimiento del software y más dificultad para leer el código, veamos la nueva forma con la desestructuración de arreglos:

```
const perfilesArr = ['Admin', 'User', 'Anonimo'];
const [ , user ] = perfilesArr;
console.log( {user} );
```

A diferencia de la desestructuración de los Objetos, con los arreglos, el nombre de la variable puede ser cualquiera porque la extracción se hace por la posición:

```
const perfilesArr = ['Admin', 'User', 'Anonimo'];
const [ administrador , usuario ] = perfilesArr;
console.log( {administrador, usuario} );
```

Siempre debemos recordar asignar los tipos de datos para evitar errores futuros y recibir la ayuda e inferencia de TS:

```
const perfilesArr: [string, boolean, number] = ['Admin', true, 1];
const [ admin, activo, codigo ] = perfilesArr;
console.log( {admin, activo, codigo} );
```

Debemos recordar siempre evitar el tipo de dato: **any**.

55. Ciclo – For of

Desde el ECMA Script 6 tenemos el ciclo FOR OF y el FOR IN:

```
type User = {
    name: string,
    rol: string
}

const user: User = {
    name: 'Alexis',
    rol: 'User'
}
```

```

const admin: User = {
    name: 'Sandoval',
    rol: 'Admin'
}

const anonimo: User = {
    name: 'alxsandoval',
    rol: 'Anonimo'
}

const perfiles = [ user, admin, anonimo ];

for (const perfil of perfiles) {
    console.log( perfil );
}

```

El uso del FOR OF, es más entendible para leer porque usa menos código que un FOR tradicional.

Para evitar resultados inesperados con el uso del FOR OF, se recomienda especificar el tipo de dato del arreglo que se recorre:

```
const perfiles: User[] = [ user, admin, anonimo ];
```

Al mantener un buen tipado TS nos permite seguir utilizando el autocompletado y la ayuda con los métodos:

```

for (const perfil of perfiles) {
    console.log( perfil.name.toUpperCase() );
}

```

56. Clases en ES6

Desde ES6 ya se incluyó el concepto de las clases. En JS la definición de las propiedades es un poco ambigua y no muy fácil de ver:

```

class User {
    name;
    rol
}

```

Para ayudar un poco con este asunto, tenemos el uso del constructor:

```

constructor( name = 'Anonimo' , rol = -1 ){
    this.name = name;
    this.rol = rol
}

```

Por otra parte, cuando se usaban las herencias en JS no había ningún error al crear una instancia de una clase extendida sin iniciar la clase Padre, pero ya se incorporó esta revisión en ES6.

```
class Perfil extends User {  
  
    isActive;  
  
    constructor(){  
        super();  
        this.isActive = true;  
    }  
  
}
```

Para que la inicialización de la clase Padre se realice de manera correcta, se tienen que enviar los argumentos:

```
class Perfil extends User {  
  
    isActive;  
  
    constructor( name, rol ){  
        super( name, rol );  
        this.isActive = true;  
    }  
  
}  
  
const alx = new User('Alexis', 1);  
const p = new Perfil('Sandoval', 2);  
  
console.log( alx );  
console.log( p );
```

```
classes-es6.js:29  
▶ User {name: 'Alexis', rol: 1}  
classes-es6.js:30  
▶ Perfil {name: 'Sandoval', rol: 2, isActive: true}
```

Se dice que JS es ambiguo porque lo que hemos visto incluso podría funcionar así:

```
class User {  
  
    constructor( name = 'Anonimo' , rol = -1 ){  
        this.name = name;  
        this.rol = rol  
    }  
  
}
```

```

class Perfil extends User {

    constructor( name, rol ){
        super( name, rol );
        this.isActive = true;
    }

}

const alx = new User('Alexis', 1);
const p = new Perfil('Sandoval', 2);

console.log( alx );
console.log( p );

```

Como vemos no se han definido las propiedades antes de usarlas, en JS, incluso las propiedades pueden aparecer de la nada:

```

class User {

    constructor( name = 'Anonimo' , rol = -1 ){
        this.name = name;
        this.rol = rol

        this.otraPropiedad = true;
    }

}

```

Este tipo de situaciones puede llevar a errores difíciles de encontrar. Al final, esta es la razón por la que TS incluyó las Clases pensando en la POO que veremos en la siguiente sección.

Cuestionario 4: Examen teórico #4

Pregunta 1:

¿Las clases son una característica nueva del ES6?

Pregunta 2:

¿El siguiente código es válido?

1. `const numero:number = 10;`
- 2.
3. `if(numero >0){`
- 4.
5. `const numero:number = 10;`
- 6.
7. `}`

Pregunta 3:

¿La destrucción de arreglos permite extraer valores y asignarlos directamente a variables?

Pregunta 4:

¿Qué hace el siguiente código?

1. `let frutas:string[] = ["Pera", "Manzana"];`
2. `let [pera, manzana] = frutas`

Pregunta 5:

¿La destrucción de objetos permite extraer las propiedades directamente de un objeto?

Pregunta 6:

¿Puedo reemplazar VAR por LET en mis futuros desarrollos usando TypeScript?

Pregunta 7:

En una función de flecha, ¿Qué valor tiene el objeto "THIS"?

Pregunta 8:

¿Qué hace la siguiente función?

```
let funcion = () =>{};
```

Pregunta 9:

¿Por qué es importante conocer sobre sobre las actualizaciones de JavaScript o ECMAScript?

Pregunta 10:

¿Qué son las plantillas literales (Templates literales)?

57. Código fuente de la sección

Aquí les dejo el código fuente de la sección como material adjunto o bien el enlace al repositorio de Github del proyecto:

- [Github - Fin-seccion-7](#)

Sección 8: Clases en TypeScript

58. ¿Qué veremos en esta sección?

La programación orientada a objetos es un tema sumamente importante, especialmente si nuestras aplicaciones van de mediana a gran escala. TypeScript trae toda la potencia de una programación orientada a objetos a la web.

Toda la sección se enfoca en enseñar sobre el uso de clases.

Puntualmente aprenderemos sobre:

1. Crear clases en TypeScript
2. Constructores
3. Accesibilidad de las propiedades:
 - a. Públicas
 - b. Privadas
 - c. Protegidas
4. Métodos de las clases que pueden ser:
 - a. Públicos
 - b. Privados
 - c. Protegidos
5. Herencia
6. Llamar funciones del padre, desde los hijos
7. Getters
8. Setters
9. Métodos y propiedades estáticas
10. Clases abstractas
11. Constructores privados.

59. Definición de una clase básica en TypeScript

Desde el ECMA Script 6 fueron incorporadas las clases. En TS existe la propiedad: “**strictPropertyInitialization**” para revisar que al crear una clase sea declarada con su constructor, por recomendación TS la tiene activa:

```
"strictPropertyInitialization": true,      /* Check for class properties that are declared but not set in the constructor. */
```

Las propiedades de una clase pueden ser públicas, privadas, protegidas o estáticas. Por cierto, una propiedad también puede ser opcional y también podemos establecer un valor por defecto. La definición básica de una clase en TS es de la siguiente manera:

```
class User {  
  
    private name: string = 'Anonimo';  
    private team: string;  
    public realName?: string;  
    static age: number = 35;
```

```
}
```

```
const alx: User = new User();
console.log( alx.realName );
```

Es posible acceder a las propiedades públicas, pero no a las privadas. Si deseamos acceder a las propiedades estáticas se usa la definición de la clase misma:

```
console.log( User.age );
```

Para inicializar una instancia con valores, debemos usar el constructor:

```
constructor ( name: string, team: string, realName?: string ){
    this.name = name;
    this.team = team;
    this.realName = realName
}
```

Las propiedades estáticas no van dentro del constructor sino en la definición porque no deben cambiar a lo largo de su ciclo de vida.

```
const alx: User = new User('Alexis', 'Admin');
console.log( alx );
```

60. Forma corta de asignar propiedades

Es muy común crear el constructor e iniciar las variables como vimos en el punto anterior, sin embargo, TS tiene una forma corta de hacer esto:

```
class User {

    // private name: string = 'Anonimo';
    // private team: string;
    // public realName?: string;
    static age: number = 35;

    constructor (
        private name: string = 'Anonimo',
        private team: string,
        public realName?: string
    ){ }

}
```

La inicialización se hace en el constructor especificando el tipo de acceso de la propiedad, su nombre, su tipo y también puede ponerse un valor por defecto.

Si no hubiéramos comentado la forma de inicializar tradicional nos marcaría un error de duplicidad:

```
Duplicate identifier 'name'.ts(2300)
```

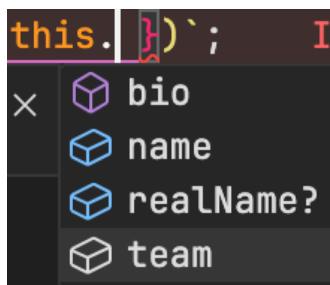
Las propiedades estáticas si van dentro de la definición de la clase y no dentro del constructor.

61. Métodos públicos y privados

Así como existen las propiedades públicas y privadas, también existen los métodos públicos y privados.

```
bio (){  
    return `${ this.name } (${ this.team })`;  
}
```

Podemos saber qué es una propiedad y qué es un método por el color que VSCode muestra los autocompletados:



En este caso **bio** es un método público y se muestra en morado mientras que **name**, **realName** y **team** son propiedades y se muestran en azul.

¿Cómo sabemos que bio es un método público? Es porque es accesible desde fuera de la clase.

Cuando se define un método, si no se establece su acceso, por defecto, es público como en el caso anterior:

```
public bio (){  
    return `${ this.name } (${ this.team })`;  
}
```

Cuando un método es privado, no puede ser accedido desde fuera de la clase y TS marcará un error, aunque se terminará transpilando a JS donde no habrá ninguna advertencia o error.

```
private bio (){  
    return `${ this.name } (${ this.team })`;  
}  
console.log( alx.bio() );
```

```
Property 'bio' is private and only accessible within class 'User'.ts(2341)
```

En el caso de los métodos estáticos, estos se normalmente al inicio de la definición de la clase:

```
class User {  
  
    static age: number = 35;  
    static getAge (){  
        return this.name;  
    }  
}
```

Para acceder a los métodos estáticos se usa la definición de la clase misma y no la instancia:

```
console.log( User.getAge() );
```

Por cierto, cada Clase tiene una propiedad llamada “name”, por lo que, el return en este caso no mostrará ‘Anonimo’ sino el nombre de la clase que es “User”

```
console.log( alx );  
console.log( alx.bio() );  
console.log( User.getAge() );
```

basica.ts:24
► User {name: 'Alexis', team: 'Admin', realName:
'Aris'}

Alexis (Admin)

basica.ts:26

User

basica.ts:28

62. Herencia, super y extends

Estos términos son parte de la POO implementada en TS. La definición de una clase heredada es:

```
class Admin extends User {  
  
}  
const alx = new Admin('Alexis', 'Aris');  
console.log( alx );
```

Algo interesante de TS es que al crear una instancia de “Admin”, como no se definió ningún constructor entonces TS llama al constructor de la clase Padre y por eso funciona:

Constructor User llamado! extends.ts:9
 extends.ts:24
► Admin {name: 'Alexis', realName: 'Aris'}

En el momento en que se define un constructor en la clase “Admin” surgirán otros puntos a considerar:

```

class Admin extends User {

    constructor(
        public isValid: boolean = true
    ){}

}

const alx = new Admin('Alexis', 'Aris');

```

En este caso aparecen 2 errores, uno de ellos es al instanciar la clase “Admin”:

`Expected 0-1 arguments, but got 2.ts(2554)`

El otro es sobre el mismo constructor de “Admin”:

`Constructors for derived classes must contain a 'super' call.ts(2377)`

Este tipo de errores es difícil de detectarlos si se trabajará únicamente con JS. Para solucionar esto en TS debemos recibir y enviar los datos al constructor de la clase Padre:

```

class Admin extends User {

    constructor(
        name: string,
        realName: string,
        public isValid: boolean = true
    ){
        super( name, realName );
        console.log('Constructor Admin llamado!');
    }

}

const alx = new Admin('Alexis', 'Aris');
console.log( alx );

```

Constructor User llamado! [extends.ts:9](#)

Constructor Admin llamado! [extends.ts:26](#)

[extends.ts:33](#)

► `Admin {name: 'Alexis', realName: 'Aris', isValid: true}`

Aquí es importante mencionar que para no crear doblemente las variables **name** y **realName**, no se establece su tipo de acceso, sino que son solamente los nombres de los parámetros que se reciben y que luego en el super se envían a la clase Padre.

Es importante mencionar que el **super()** debe ser llamado como la primera declaración dentro del constructor.

Ahora veamos el uso del tipo de acceso: “**protected**”, este tipo de acceso permite acceder a métodos y propiedades dentro de clases que extiendan de una clase Padre.

```
protected getFullName (){
    return `${ this.name } ${ this.realName }`;
}
```

Para llamar a un método “**protected**” que está en la clase Padre se utiliza la palabra reservada: **super**

```
getFullNameDesdeAdmin(){
    console.log( super.getFullName() );
}
```

Ahora veamos el ejemplo completo:

```
class User {

    constructor(
        public name: string,
        public realName: string
    ){
        console.log('Constructor User llamado!');
    }

    protected getFullName (){
        return `${ this.name } ${ this.realName }`;
    }

}

class Admin extends User {

    constructor(
        name: string,
        realName: string,
        public isValid: boolean = true
    ){
        super( name, realName );
        console.log('Constructor Admin llamado!');
    }

    getFullNameDesdeAdmin(){
        console.log( super.getFullName() );
    }

}

const alx = new Admin('Alexis', 'Aris');
alx.getFullNameDesdeAdmin();
```

Aquí, la instancia **alx** puede acceder el método **getFullNameDesdeAdmin** que es público porque no se definió su tipo de acceso y a su vez éste accede al método “**protected**” **getFullName** mediante el uso de **super**.

63. Gets y Sets

Los Getters y los Setters vistos desde afuera parecen propiedades normales pero vistos desde su interior son como métodos:

```
get fullName(){
    return `${ this.name } - ${ this.realName }`;
}
```

Es importante mencionar que los Getters siempre devuelven algo y también no es necesario ejecutarlos como los métodos:

```
console.log( alx.fullName );
```

Con respecto a los Setters se usan para asignar valores a las propiedades y solo pueden recibir un solo valor:

```
set fullName( name: string ){
    this.name = name;
}
```

Veamos cómo se utilizan:

```
alx.fullName = 'Sandoval'
```

Una ventaja de usar los Setters es que podemos aplicar lógica de programación antes de su asignación:

```
set fullName( name: string ){

    if( name.length < 3 ){
        throw new Error('El nombre debe ser mayor a 3 letras');
    }
    this.name = name;
}
```

También podemos aplicar lógica de programación con los Getters.

64. Clases Abstractas

Al crear instancias de clases, TS infiere el tipo según la clase y esto es útil sobre todo cuando se crea una variable sin inicializarla, así cuando se agreguen los valores de las propiedades aceptará solo los permitidos:

```
class User {
    constructor(
        public name: string,
        public realName: string
    ){} {}
}
```

```
let usuario: User;
usuario = 1; ← Esto lo marcaria como un error porque no cumple la firma de la clase User
```

Con respecto a las clases abstractas debemos saber que no se pueden crear instancias porque solo sirven para crear otras clases y se definen escribiendo la palabra reservada “**abstract**” antes de la palabra “**class**”:

```
abstract class User {
    constructor(
        public name: string,
        public realName: string
    ){}
}
```

Esto no está permitido en una clase abstracta: `Cannot create an instance of an abstract class.ts(2511)`

```
const alx = new User('Alexis','Aris');
```

Veamos cómo se utilizan las clases abstractas:

```
class Perfil extends User{
    // Al no definir constructor, se llama el de la clase Padre
}
const alx = new Perfil('Alexis','Aris');
console.log( alx );
```

La ventaja de utilizar clases abstractas es que podemos tener métodos y propiedades especializados para cada una de las clases:

```
class Perfil extends User{
    // Al no definir constructor, se llama el de la clase Padre
    isValid(){
        return '¡Usuario Valido!';
    }
}

class Anonimo extends User{
    access(){
        return 'Sin acceso';
    }
}

const alx = new Perfil('Alexis','Aris');
const noName = new Anonimo('No Name','Anonimo');

console.log( alx.isValid() );
console.log( noName.access() );
```

Una clase abstracta también sirve para especificar que se espera una clase, objeto o argumento que haya extendido de la clase Padre:

```

const printName = ( character: User )=>{
    console.log( character.realName );
}

printName( alx );
printName( noName );

```

En este caso podemos utilizar las propiedades **realName** porque enviamos el argumento **alx** que extiende de **User** por lo que cumple con la firma, por el contrario, si la clase **Anonimo** no extendiera de **User** no podríamos usar la función **printName**:

```

class Anonimo {
    access(){
        return 'Sin acceso';
    }
}
const printName = ( character: User )=>{
    console.log( character.realName );
}
const alx = new Perfil('Alexis','Aris');
const noName = new Anonimo('No Name','Anonimo'); ← Expected 0 arguments, but got 2.ts(2554)

printName( alx );
printName( noName ); ← Aquí Ts mostrará un error

```

```

Argument of type 'Anonimo' is not assignable to parameter of type 'User'.
Type 'Anonimo' is missing the following properties from type 'User': name, realName.ts(2345)

```

Una vez más recordemos que no se puede crear una instancia de una clase abstracta.

65. Constructores privados

Los constructores privados se utilizan para controlar la manera en la cual las instancias son ejecutadas, es común usarlas para crear Singletos (Es una única instancia durante toda la aplicación).

Cuando un constructor es privado, solo se puede llamar desde la misma clase de lo contrario mostrará un error:

```

class User{

    static instance: User;

    private constructor ( public name: string ){

    }

    static callUser(): User {

```

```

    // Si User.instance es null o undefined
    if ( !User.instance ){
        // Aquí es donde se crea la instancia
        User.instance = new User('¡Soy Alexis... el único usuario!');
    }

    return User.instance;
}

}

```

Cuando se trabaja con constructores privados siempre se devolverá la misma instancia y si se modifican valores de las propiedades el cambio será global:

```

class User{

    static instance: User;
    private constructor ( public name: string ){

    }

    static callUser(): User {
        // Si User.instance es null o undefined
        if ( !User.instance ){
            // Aquí es donde se crea la instancia
            User.instance = new User('¡Soy Alexis... el único usuario!');
        }

        return User.instance;
    }

    changeName( newName: string ):void {
        this.name = newName;
    }
}

const usuario = User.callUser();
const usuario2 = User.callUser();
const usuario3 = User.callUser();

usuario.changeName('Sandoval');

console.log( usuario );
console.log( usuario2 );
console.log( usuario3 );

```

```

private-constructor.ts:31
▶ User {name: 'Sandoval'}
private-constructor.ts:32
▶ User {name: 'Sandoval'}
private-constructor.ts:33
▶ User {name: 'Sandoval'}

```

Aquí, aunque se crearon 3 variables usuario y aunque solo se modificó en la variable **usuario** el nombre cambio para todas las demás variables.

66. Código fuente de la sección

Les dejo el código del proyecto hasta este punto y también el repositorio de GitHub por si lo quieren tener a la mano.

- [Github - Fin-seccion-8](#)

Sección 9: Interfaces

67. ¿Qué veremos en esta sección?

Esta sección está dedicada a crear interfaces, las cuales nos permitirán crear reglas o planos de cómo se deben de construir clases, métodos u objetos.

Puntualmente aprenderemos:

1. ¿Por qué es necesario una interfaz?
2. ¿Cómo creamos una interfaz básica?
3. Crear propiedades opcionales
4. Crear métodos
5. Asignar interfaces a las clases

Al final, tendremos un examen práctico y teórico sobre las interfaces.

68. Interfaz básica

Básicamente una interfaz funciona como el **type** que ya hemos visto, incluso solo cambiando la palabra reservada de **type** por **interface** funcionaria, solo hay que quitar el signo (=):

```
interface persona {  
    name: string;  
    age: number,  
    rol: number[],  
    getName?: () => string
```

```
}
```

Al igual que type, la interface no tiene una representación visual en JS, ambas nos ayudan a restringir como lucen nuestros objetos ya que principalmente se emplea al trabajar con objetos:

Una gran diferencia a tener en cuenta es que los tipos no pueden expandirse porque una vez que se definen así se quedan mientras que las interfaces si se pueden expandir.

Las interfaces es común utilizarlas para hacer peticiones HTTP y los type se usan para cuando sabemos que algo no va a cambiar y se usa en patrones como redux para definir que tipos de acciones son permitidas en algún objeto.

En apariencia un type va con el signo (=):

```
type persona = {
    name: string;
    age: number,
    rol: number[],
    getName?: () => string
}
```

Mientras que la interfaz tiene la apariencia de una clase:

```
interface persona {
    name: string;
    age: number,
    rol: number[],
    getName?: () => string
}
```

En la interfaz tampoco se implementan los métodos sino solo definen y puede tener valores opcionales. Para más información sobre las diferencias se puede consultar la documentación:

- [Differences between Type and Interfaces](#)

Por último: Es posible heredar interfaces con la palabra "extends"

```
interface Carro{
    llantas:number;
    modelo:string;
}

interface Volvo extends Carro{
    seguro:boolean;
}
```

```
var volvo:Volvo = {  
    llantas: 4,  
    modelo:"sedan",  
    seguro:true  
}
```

69. Estructuras complejas

Cuando un objeto tiene más de un nivel de anidación de otros objetos, se recomienda definir otra interfaz porque el objeto se vuelve completo de mantener y leer:

```
interface Client {  
  
    name: string,  
    age?: number,  
    address?: {  
        id: number,  
        zip: string,  
        city: string  
    }  
}
```

En lugar de hacer lo anterior, esto es lo que se recomienda:

```
interface Client {  
  
    name: string,  
    age?: number,  
    address?: Address  
}  
  
interface Address {  
  
    id: number,  
    zip: string,  
    city: string  
}
```

Cuando se crean interfaces de este tipo, siempre la principal va arriba y las secundarias abajo conforme se van creando.

Es importante mencionar que, al asignar valores a los objetos, el orden en que se definen las propiedades no es importante.

Al trabajar con interfaces mantenemos un código que, aunque sea complejo se vuelve más fácil de leer y mantener a lo largo del tiempo.

```

const client: Client = {
    name: 'Alexis',
    age: 25,
    address: {
        id: 18,
        zip: '22124',
        city: 'Tijuana'
    }
}

const client2: Client = {
    name: 'Sandoval',
    age: 30,
    address: {
        city: 'TJ',
        id: 120,
        zip: '22456'
    }
}

```

Al ver su equivalente en JS, podremos observar que todo pasa como objetos normales, por lo cual, aunque tengamos muchas interfaces pasan como 0 KB de código a JS.

70. Métodos en la interfaz

Los métodos dentro de las interfaces pueden ser opcionales u obligatorios. Una diferencia con los **types** y las **interface** con respecto a los métodos está en su declaración:

```

type persona = {
    name: string;
    age: number,
    rol: number[],
    getName?: () => string
}

```

La declaración de los métodos en las interfaces lleva (:) después de la declaración de los parámetros:

```

interface Client {
    name: string,
    age?: number,
    address: Address,
    getFullAddress( id: string ): void
}

```

TS reconoce esto como un método: `(method) Client.getFullAddress(id: string): void`

No se recomienda crear definiciones de los métodos en las interfaces, aunque si es posible hacerlo. Cuando se necesitan implementación de métodos lo ideal es crear una clase y no una **interface**.

```
const client: Client = {
    name: 'Alexis',
    age: 25,
    address: {
        id: 18,
        zip: '22124',
        city: 'Tijuana'
    },
    getFullAddress( id: string ){
        return this.address.city;
    }
}
```

Algunos consideran las interfaces como clases simples, pero recuerden que estas no crean instancias.

71. Interfaces en las clases

Es importante saber que no se puede extender de una interfaz, pero si es posible implementar una interfaz:

```
interface User {
    name: string,
    realName: string,
    isValid( id:number ): string,
}

interface Person {
    age: number
}

class Perfil implements User, Person {

    public age: number;
    public name: string;
    public realName: string;

    isValid( id: number ) {
        return this.name + ' ' + this.realName;
    }

}
```

El **implements** en una clase se utiliza para forzar que la clase implemente todo lo que se requiere. Recordemos que:

- **type** se utiliza cuando sabemos que un objeto no va a expandirse como en el uso de patrones
- **interface** se utiliza cuando sabemos que va a variar o cambiar un objeto, su uso es mayor

72. Interfaces para las funciones

Esto es muy poco común, pero podría ser necesario saberlo. Usaríamos la interfaz para restringir una función o para asegurarnos que luzca de cierta manera:

```
interface addTwoNumbers {
    ( a: number, b: number ) : number
}
```

Al implementar la función debe cumplir la firma de la declaración para que sea aceptada:

```
interface addTwoNumbers {
    ( a: number, b: number ) : number
}

let addNumbersFunction: addTwoNumbers;

addNumbersFunction = (a:number, b:number) => {
    return 10;
}
```

Si no recibe los parámetros o no regresa el tipo de dato especificado, esto sería considerado como un error.

73. Ejercicio práctico #5: Implementación

Por favor descarguen y descompriman el archivo adjunto y procedan a la siguiente clase donde les daré la introducción de lo que quiero que hagan.

74. Tarea y Resolución del ejercicio práctico #5

Usando el archivo adjunto app.ts realizar los ejercicios solicitados, a continuación, se muestran las soluciones:

```
// Crear interfaces
interface Auto {
    encender: boolean,
    velocidadMaxima: number,
    acelerar(): void
}

// Cree una interfaz para validar el auto (el valor enviado por parametro)
const conducirBatimovil = ( auto: Auto ):void => {
    auto.encender = true;
    auto.velocidadMaxima = 100;
    auto.acelerar();
}
```

```
const batimovil: Auto = {
  encender:false,
  velocidadMaxima:0,
  acelerar(){
    console.log("..... gogogo!!!");
  }
}

// Cree una interfaz con que permita utilizar el siguiente objeto
// Empleando propiedades opcionales

interface Guason {
  reir?: boolean,
  comer?:boolean,
  llorar?: boolean
}

const guason: Guason = {
  reir: true,
  comer: true,
  llorar: false
}

const reir = ( guason: Guason ):void => {
  if( guason.reir ){
    console.log("JAJAJAJA");
  }
}

// Cree una interfaz para la siguiente funcion

interface ciudadGoticaFn {
  ( ciudadanos:string[] ): number
}

const ciudadGotica: ciudadGoticaFn = ( ciudadanos:string[] ):number => {
  return ciudadanos.length;
}

// Cree una interfaz que obligue crear una clase
// con las siguientes propiedades y metodos

/*
propiedades:

  - nombre
  - edad
  - sexo
  - estadoCivil
  - imprimirBio(): void // en consola una breve descripcion.
*/

```

```
interface interfacePersona {  
    nombre: string,  
    edad: number,  
    sexo: string,  
    estadoCivil: string,  
    imprimirBio():void  
}  
  
class Persona implements interfacePersona {  
    nombre: string;  
    edad: number;  
    sexo: string;  
    estadoCivil: string;  
    imprimirBio(): void {  
        console.log(`Hola, soy ${this.nombre}, tengo ${this.edad} años y soy ${this.estadoCivil}`);  
    }  
}
```

75. Cuestionario 5: Examen teórico #5

Pregunta 1:

¿Qué son las interfaces?

Pregunta 2:

¿Es posible crear interfaces para permitir o denegar qué podemos asignar a una función?

Pregunta 3:

¿Este es el producto de la interfaz "Carro" en JavaScript?

```
1. // TypeScript  
2. interface Carro{  
3.  
4.     nombre:string  
5.  
6. }  
7.  
8. // JavaScript  
9. function Carro(carro){  
10.  
11.     this.carro = carro;  
12.  
13. }
```

Pregunta 4:

¿Con qué palabra reservada podemos implementar una interface en una clase?

Pregunta 5:

¿Cuál es el objetivo de una implementación de una interface en una clase?

Pregunta 6:

¿Es posible asignar a una variable, el tipo de una interfaz ?

Pregunta 7:

En una interfaz, ¿Sólo hay que definir las propiedades y métodos que son obligatorios?

Pregunta 8:

En la creación de un método de una interfaz, ¿Qué puedo detallar?

Pregunta 9:

¿Con qué carácter definimos que una propiedad o método puede ser opcional en la interfaz?

Pregunta 10:

¿El siguiente código es valido en TypeScript?

```
1. interface Carro{  
2.   llantas:number;  
3.   modelo:string;  
4. }  
5.  
6. interface Volvo extends Carro{  
7.   seguro:boolean;  
8. }  
9.  
10. var volvo:Volvo = {  
11.   llantas: 4,  
12.   modelo:"sedan",  
13.   seguro:true  
14.  
15. }
```

76. Código fuente de la sección

Aquí les dejo el código fuente por si lo llegan a necesitar o comparar contra el mío.

- [Github - Fin-seccion-8](#)

Sección 10: NameSpaces

77. ¿Qué veremos en esta sección?

TypeScript, es un lenguaje de programación web, que nos permite crear objetos que nos servirán a lo largo de nuestro programa. Los namespaces, existen para ayudarnos en la reutilización de nuestras variables, constantes y métodos.

Puntualmente aprenderemos sobre:

1. Explicación del ¿por qué son necesarios los namespaces?
2. Crear namespaces
3. Múltiples namespaces en un mismo proyecto
4. Importar namespaces
5. Problemática que se puede presentar utilizando un namespace.

78. Creando un Namespace

Cada vez es menos común que se utilicen digamos que personalmente los Namespaces, sin embargo, por debajo son muy utilizados por librerías, frameworks, etc.

Es más común ver Namespaces en el backend que en el frontend. Pero ¿qué es un Namespace?

Un IIFE podría considerarse como un Namespace, pero el problema es que no se tiene acceso desde fuera de la función, en cambio, un Namespace sirve como agrupador que puede ser utilizado en cualquier otro lugar.

Veamos cómo se crea un Namespace:

```
namespace Validations {  
  
    const validate = ( text: string ) => {  
  
        return ( text.length > 3 ) ? true : false;  
    }  
  
    const validateDate = ( myDate : Date ) : boolean => {  
        return ( isNaN( myDate.valueOf() ) ) ? false : true;  
    }  
}
```

Ahora, la pregunta es: ¿cómo puedo usar ese Namespace? Para que sean visibles se hace el export:

```
export const validateText = ( text: string ) => {  
    return ( text.length > 3 ) ? true : false;  
}  
  
export const validateDate = ( myDate : Date ) : boolean => {  
    return ( isNaN( myDate.valueOf() ) ) ? false : true;  
}
```

Ahora ya podrán ser visibles y utilizadas las funciones desde afuera:



The screenshot shows a code editor with a tooltip for the variable 'Validations'. The tooltip contains two items: 'validateDate' and 'validateText'. The code snippet is as follows:

```
console.log( Validations. ); Identifier expected.
```

[?] validateDate const Validations.validateDate: (myDate: Dat...
[?] validateText

Se puede ver un Namespace como un agrupador que puede contener clases, funciones, etc.

79. Inicio de proyecto – Módulos y Webpack

Para la continuación de esta sección primero es necesario descargar lo siguiente:

- <https://github.com/Klerith/curso-typescript/tree/codigo-inicial>

- Despues es necesario renombrar la carpeta descargada por el nombre: **ts-node**
- Ahora, hay que abrir la carpeta en VSCode.
- Una vez hecho lo anterior, ejecutar el comando: **npm install**
- Si el puerto especificado en el archivo (package.json) no genera conflictos: **--port=8081**
- Ejecutar el comando: **npm start**

Lo que tenemos es un proyecto de webpack para frontend basado en TypeScript. Esto es un proyecto personalizado y no un estándar.

Al respecto, vale la pena mencionar que el estándar para la resolución de modulos es “node”, por lo que la configuración del tsconfig.json quedo así:

```
"moduleResolution": "node", /* Specify module resolution strategy: 'node' (Node.js) or 'classic' (TypeScript pre-1.6). */
```

80. Imports y Exports

Ahora con el proyecto con Webpack, veamos el uso de los imports y los exports. Primero tenemos que **exportar** lo que deseamos que sea visible:

```
export class User {
  constructor(
    public name: string,
    public realName: string,
    public id: number
  )
}
```

Ahora, ya podemos **importar** y utilizar lo que hemos exportado:

```
import { User } from "./classes/User";

const usuario = new User('alx','Aris',1);
```

Cuando necesitamos hacer cambios en la clase ya que tenemos dividido y organizado el código podremos dar mantenimiento de una mejor manera.

81. Export default y exportación con alias

Cuando trabajamos con múltiples imports y exports es posible que tengamos funciones, clases o variables repetidas. Para resolver estos inconvenientes podemos hacer uso de los alias:

```
export class User {
  constructor(
    public name: string,
    public realName: string,
    public id: number
  ){}

}

export class User2 { }
export class User3 { }
export class User4 { }
```

La importación de esto sería:

```
import { User,User2, User3, User4 } from "./classes/User";

const usuario = new User('alx', 'Aris', 1 );
const User = 'alxsandoval'; ← Esto daria un error de conflicto con el nombre del import

console.log( usuario );
```

Si renombramos el import, podemos solucionar el problema:

```
import { User as superUser,User2, User3, User4 } from "./classes/User";

const usuario = new superUser('alx', 'Aris', 1 );

console.log( usuario );
const User = 'alxsandoval';
```

También podemos utilizar un alias completo a todo lo que importamos:

```
import * as UserClases from "./classes/User";
```

Para utilizarlo se debe emplear el alias:

```
const usuario = new UserClases.User('alx', 'Aris', 1);
```

Ahora, veamos cómo se trabaja con las exportaciones por defecto, para esto tenemos 2 maneras:

```
interface Rol {
  id: number,
  descripcion: string
}

const roles: Rol[] = [
  {
    id: 1,
    descripcion: 'Administrador'
  },
  {
    id: 2,
    descripcion: 'Anonimo'
  }
];

export default roles;
```

Cuando se hace una exportación por defecto, al importar ya no se usan llaves porque ya no es una importación independiente:

```
import cualquierNombre from "./data/roles";
```

Cuando se trabaja con importaciones por defecto, el nombre del import puede ser cualquiera y funcionara todo:

```
console.log( cualquierNombre );
console.log( cualquierNombre[1] );
```

Por último, es posible tener importaciones por defecto e importaciones independientes:

```
export interface Rol {
  id: number,
  descripcion: string
}

const roles: Rol[] = [
  {
    id: 1,
    descripcion: 'Administrador'
  },
  {
    id: 2,
    descripcion: 'Anonimo'
  }
];

export default roles;
```

Al importarlas sería de la siguiente manera:

```
import roles, { Rol } from "./data/roles";
```

82. Tarea – Resolver errores en TypeScript

Teniendo lo siguiente:

```
import { User } from "./classes/User";

const usuario = new User('alx', 'Aris', 1);

console.log( usuario.rol );
```

Se importo roles:

```
import roles from '../data/roles';
```

y luego, se completó el Getter:

```
get rol(): string { // return string
    return roles.find( rol => rol.id === this.id )?.descripcion || 'not found';
}
```

Para solucionar que roles.find podría dar un undefined se usó el operador (?) null check, aunque también se puede usar el operador (!) que significa que TS confié en lo que estamos haciendo porque nunca será null o undefined, pero este operador se debe usar con cuidado.

83. Código fuente de la sección

Aquí les dejo el código fuente y repositorio de GitHub por si quieren tenerlo a la mano o compararlo contra el mío:

- [Github - Fin-seccion-10](#)

Sección 11: Genéricos - Generics

84. ¿Qué veremos en esta sección?

JavaScript por ser un lenguaje dinámico, conlleva a tener varios problemas por esa misma flexibilidad, pero a su vez, permite resolver problemas de una forma muy sencilla. Esta sección está destinada a comprender como mantener la programación estructurada del TypeScript con el dinamismo de JavaScript.

Puntualmente aprenderemos sobre:

1. Uso de los genéricos
2. Funciones genéricas
3. Ejemplos prácticos sobre los genéricos
4. Arreglos genéricos
5. Clases genéricas

85. Introducción a los Genéricos

Una función genérica básicamente es una función que puede recibir cualquier tipo de argumento. Las funciones genéricas son de JS, pero veamos cómo funcionan en TS:

```
export const printObject = ( arg: any ) => {  
    console.log( arg );  
}
```

Veamos cómo se utilizaría:

```
import { printObject } from "./generics/generics";  
  
printObject( 123 );  
printObject( new Date() );  
printObject( { a:1, b:2, c:3 } );  
printObject( [ 1, 2, 3, 4, 5 ] );  
printObject( "Hola Mundo" );
```

Se imprime todo lo que se recibe, pero el inconveniente es que TS no está al pendiente del tipo de dato que recibe.

Aquí se necesita que TS ayude con el tipado. Veamos cómo se hace con una función tradicional:

```
export function genericFunction ( arg: any ) {  
    return arg;  
}
```

Cuando se utiliza vemos que pueden existir los errores del tratado de la información:

```
console.log( genericFunction( 3.1416 ).toFixed(2) );  
console.log( genericFunction( "Hola Mundo" ).toFixed(2) ); ← Esto es un error
```

Lo que deseamos es que se procese y retorne un tipo de dato según lo que recibimos. Veamos como hacerlo en el siguiente tema.

86. Funciones Genéricas

Para transformar una función tradicional en una función genérica se usa el símbolo <T>

```
export function genericFunction<T> ( arg: T ): T {  
    return arg;  
}
```

Ahora la función se vuelve genérica y recibirá un tipo T como argumento y la salida será de tipo T. Aquí T es una expresión genérica.

La misma función genérica expresada como función de flecha sería así:

```
export const genericFunctionArrow = <T> ( arg: T ): T => {  
    return arg;  
}
```

Gracias a esto, ya podemos tratar los valores de retorno de manera correcta según su tipo de dato:

```
console.log( genericFunctionArrow( 3.1416 ).toFixed(2) );  
console.log( genericFunctionArrow( "Hola Mundo" ).toUpperCase() );
```

87. Ejemplo de función genérica en acción

Supongamos que tenemos la siguiente **interface** User:

```
export interface User {  
    name: string,  
    realName: string  
}
```

Y que también tenemos la **interface** Admin:

```
export interface Admin {  
    name: string,  
    accessLevel: Number  
}
```

Como vemos cada **interface** tiene cada una sus propiedades, al crear un objeto y usar genéricos sería así:

```
const alx = {  
    name: 'Alexis',  
    realName: 'Aris',  
    accessLevel: 1  
}  
  
console.log( genericFunctionArrow<Admin>( alx ) );
```

También podría ser de tipo <User> porque cumple con la firma de la **interface**:

```
console.log( genericFunctionArrow<User>( alx ) );
```

Pero si en el objeto “alx” faltara una propiedad entonces TS mostrará un error.

88. Agrupar exportaciones

Cuando tenemos una aplicación y tenemos muchas interfaces, podemos agrupar de la siguiente manera:

1. Creamos un archivo index.ts (es el archivo que se carga cuando se hace referencia a un directorio)
2. Importamos las interfaces en index.ts

```
import { User } from './User';
import { Admin } from './Admin';
```

3. Cambiamos el import por export

```
export { User } from './User';
export { Admin } from './Admin';
```

4. Importamos el directorio donde tenemos el index.ts y también lo que necesitemosm (User, Admin)

```
import { User, Admin } from "./interfaces";
```

Ahora ya podemos trabajar con lo que hemos importado:

```
console.log( genericFunctionArrow<User>( alx ) );
```

De esta manera ya podemos agregar o quitar imports y exports en el archivo index.ts para poder mantener mejor organizado nuestro proyecto.

89. Ejemplo aplicado de genéricos

Para este ejemplo utilizaremos “**Axios**” que se ha vuelto como un estándar para el manejo de las peticiones http:

- **npm install axios**

Creamos nuestro archivo “get-pokemon.ts”:

```
import axios from 'axios';

export const getPokemon = async ( pokemonId: number ) => {

  const resp = await axios.get(`https://pokeapi.co/api/v2/pokemon/${ pokemonId }`);

  console.log( resp );
}
```

Ahora llamamos la función getPokemon:

```
import { getPokemon } from "./generics/get-pokemon";

getPokemon( 4 )
  .then( resp => console.log( resp ) )
  .catch( error => console.log( error ) )
  .finally( () => console.log("Fin de getPokemon") );
```

Hasta aquí vemos como se utiliza una petición con axios pero veamos a continuación como trabajar con interfaces y genéricos, aplicando al mismo ejemplo.

90. Mapear respuestas http

Cuando utilizamos async en una petición HTTP debemos saber que siempre regresa una promesa:

```
import axios from 'axios';

export const getPokemon = async ( pokemonId: number ): Promise<number> => {
  const resp = await axios.get(`https://pokeapi.co/api/v2/pokemon/${ pokemonId }`);
```

Podemos obtener los datos de la respuesta si conocemos los nombres de las propiedades:

```
console.log( resp.data.name );
```

Pero aquí surge el asunto de que TS no tiene idea de cómo viene la respuesta y es aquí donde debemos mapear la respuesta. Primero podemos crear una interface para determinar cómo lucirá la respuesta:

```
export interface Pokemon {

  name: string,
  picture: string

}
```

Ahora, a la petición de axion.get ya podemos especificarle que la respuesta genérica es del tipo de la interface:

```
const resp = await axios.get<Pokemon>(`https://pokeapi.co/api/v2/pokemon/${ pokemonId }`);
```

Y ahora sí, TS ya nos da ayuda de autocompletado porque reconoce las propiedades de la respuesta:

```
console.log( resp.data. ); Identifier expected.
  ↗ name
  ↗ picture
```

(property) Pokemon.name: string

Es importante recalcar que la **interface** que se crea es para decir como luce la respuesta de la petición, pero no necesariamente será exactamente igual, no debemos confundirnos, es solo un mapeo de la respuesta.

Por cierto, también podemos devolver en la respuesta algo del tipo de la interface:

```
export const getPokemon = async ( pokemonId: number ): Promise<Pokemon> => {

    const resp = await axios.get<Pokemon>(`https://pokeapi.co/api/v2/pokemon/${pokemonId}`);

    console.log( resp.data.name );

    return resp.data;
}
```

Ahora, cuando llamemos a la función, esta sabrá que propiedades tiene la respuesta porque la interface viene heredada:

```
getPokemon( 4 )
    .then( resp => console.log( resp ) )
    .catch( error => console.log( error ) )
    .finally( () => console.log("Fin de getPokemon") );
```

Tenemos las propiedades name y picture:

```
getPokemon( 4 )
    .then( resp => console.log( resp ) ) Identifier expected.
    .catch( error => console.log( error ) )
    .finally( () => console.log("Fin" picture)
```

Sin embargo, mapear todas las propiedades sería mucho trabajo innecesario. Por lo cual, necesitamos herramientas como: <https://app.quicktype.io>

Seleccionando de la API el RAW JSON lo copiamos y pegamos en quicktype seleccionando solo interfaces:

The screenshot shows the quicktype interface generator interface. At the top, there's a checkbox labeled "View raw JSON (32.732 kB, 948 lines)". Below it, the raw JSON of the Pokemon API is pasted into the "Source type" field. On the right side, the generated TypeScript code for the Pokemon interface is displayed. The interface includes properties like base_experience, forms, game_indices, height, held_items, id, is_default, location_area_encounters, moves, name, order, past_types, species, sprites, stats, types, and weight. Below the code, there are settings for "Language" (TypeScript), "Other" (Interfaces only, Transform property names to be JavaScript, Explicitly name unions, Verify JSON parse results at runtime, Use union type instead of enum, Use types instead of interfaces, Make all properties optional), and a "Copy Code" button.

```
export interface Pokemon {
    abilities: Ability[];
    base_experience: number;
    forms: Species[];
    game_indices: GameIndex[];
    height: number;
    held_items: HeldItem[];
    id: number;
    is_default: boolean;
    location_area_encounters: string[];
    moves: Move[];
    name: string;
    order: number;
    past_types: PastType[];
    species: Species;
    sprites: Sprites;
    stats: Stat[];
    types: Type[];
    weight: number;
}

export interface Ability {
    ability: Species;
    is_hidden: boolean;
    slot: number;
}

export interface Species {
    name: string;
    url: string;
}

export interface GameIndex {
    game_index: number;
    version: Species;
}

export interface HeldItem {
    item: Species;
    version_details: VersionDetail[];
}

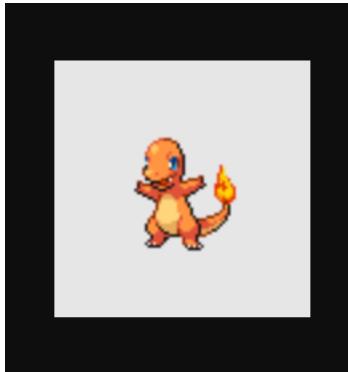
export interface VersionDetail {
```

Así ya tendremos mapeada en segundos la respuesta de la petición HTTP:

```
const { data } = await axios.get<Pokemon>(`https://pokeapi.co/api/v2/pokemon/${ pokemonId }`);  
  
// console.log( data.abilities[0].ability.url );  
  
return data;
```

Ahora, al llamar la función tendremos todas las propiedades y valores permitidos para trabajar con la respuesta:

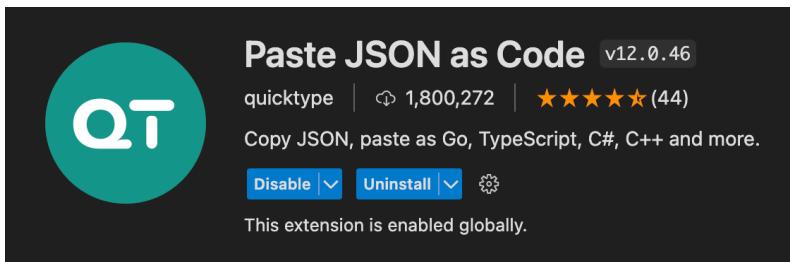
```
getPokemon( 4 )  
.then( pokemon => console.log( pokemon.sprites.front_default ) )  
.catch( error => console.log( error ) )  
.finally( () => console.log("Fin de getPokemon") );
```



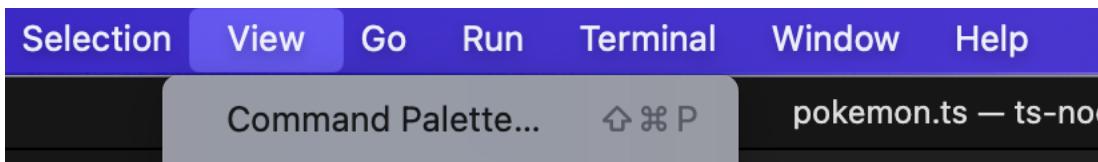
Sin embargo, hay una forma más fácil para no tener que ir a la aplicación y es usar una extensión que veremos a continuación.

91. Quicktype.io extensión

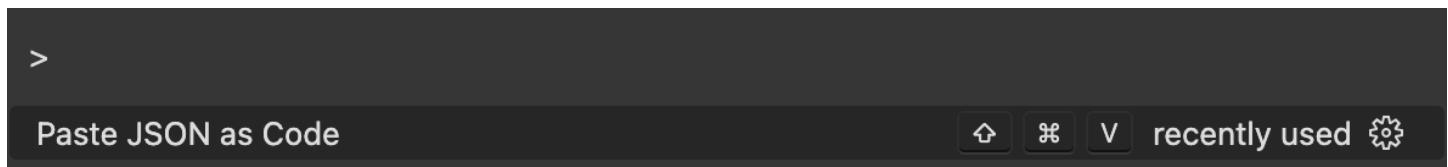
La manera más rápida es instalar una extensión en VSC: **Paste JSON as Code**



Utilizando la paleta de comandos en un archivo TS:



Seleccionamos: Paste JSON as Code



Luego damos el nombre principal y listo:

```
// Generated by https://quicktype.io

export interface Pokemon {
    abilities: Ability[];
    base_experience: number;
    forms: Species[];
    game_indices: GameIndex[];
```

Si en el futuro cambiará la respuesta TS capturará los errores y podemos resolverlos rápidamente:

```
getPokemon( 4 )
    .then( pokemon => console.log( pokemon.name2 ) )
```

Intentar hacer esto mismo con JS se puede convertir en un verdadero dolor de cabeza. Esta es una gran ayuda de TS.

92. Código fuente de la sección

Aquí les dejo el código fuente de la sección por si la llegan a necesitar o comparar contra el suyo:

- [Github - Fin-seccion-11](#)

Sección 12: Decoradores

93. ¿Qué veremos en esta sección?

Los decoradores son una característica nueva en el TypeScript que cada vez es más utilizada por otros frameworks como Angular 2. Pero vamos a aprender a utilizar decoradores en nuestros proyectos.

Puntualmente aprenderemos sobre:

1. ¿Qué son los decoradores?
2. ¿Para qué sirven?

3. Decoradores de clases
4. Decoradores de fabrica
5. Ejemplos prácticos
6. Decoradores anidados
7. Decoradores de métodos
8. Decoradores de propiedades
9. Decoradores de parámetros

94. Introducción a los decoradores

Esta es la documentación oficial de los decoradores:

- <https://www.typescriptlang.org/docs/handbook/decorators.html>

En esta sección veremos el uso principal de los decoradores.

Un decorador podemos decir que es una función que se ejecuta en el momento de la transpilación, es algo, que permite expandir la funcionalidad de una clase, método o propiedad a un objeto.

Los decoradores se crearon principalmente para hacer más fácil la lectura del código. Angular fue de los primeros que empezaron a popularizar los decoradores.

Podemos ver que algo es un decorador por el símbolo (@).

Nest.js que es un framework de backend también utiliza los decoradores casi para cualquier cosa.

Es más común consumir los decoradores que crearlos, sin embargo, aprenderemos como se crean de manera básica.

95. Decoradores de clases

Los decoradores más fáciles de utilizar son los decoradores de clase. Los decoradores se utilizan con la forma tradicional **function** pero se pueden usar funciones de flecha.

```
function printToConsole( constructor: Function ){
  console.log( constructor );
}
```

```

@printToConsole
export class Pokemon {

    public publicApi:string = 'http://pokeapi.co';

    constructor(
        public name:string
    ){
        }

    }
}

```

En algunas versiones de TS podría aparecer el error de la configuración del archivo **tsconfig.json**, por lo cual, se debe activar la propiedad del uso de decoradores: **"experimentalDecorators": true**

```
"experimentalDecorators": true           /* Enables experimental support for ES7 decorators. */
```

Este decorador se aplica a la clase en el momento de la transpilación, pero solamente 1 vez:

```

function printToConsole( constructor: Function ){
    console.log( new constructor('Pika') );
}

```

Aunque se hagan varias instancias de la clase:

```

const charmander = new Pokemon('Charmander');
const charmander2 = new Pokemon('Charmander');
const charmander3 = new Pokemon('Charmander');

console.log( charmander ); console.log( charmander2 ); console.log( charmander3 );

```

```

pokemon-class.ts:12
▶ Pokemon {name: 'Pika', publicApi: 'http://pokeapi.co'}
index.ts:7
▶ Pokemon {name: 'Charmander', publicApi: 'http://pokeapi.co'}
index.ts:8
▶ Pokemon {name: 'Charmander', publicApi: 'http://pokeapi.co'}
index.ts:9
▶ Pokemon {name: 'Charmander', publicApi: 'http://pokeapi.co'}

```

96. Decoradores de fabrica – Factory decorators

Estos decoradores son comunes de verlos en uso. Si nosotros quisieramos imprimir los valores de los argumentos no podríamos hacerlo de la forma anterior.

Para que un decorador sea Decorador de fábrica, debe retornar una función:

```
const printToConsoleConditional = (): Function => {
    return () => console.log('Hola mundo');
}

@printToConsoleConditional()
export class Pokemon {
```

Aquí es importante mencionar que se debe especificar que se devuelve una Function.

El console.log() se ejecutará una vez a la hora de la transpilación.

Lo interesante de este decorador es que pude recibir argumentos al llamar la función:

```
function printToConsole( constructor: Function ){
    console.log( constructor );
}

const printToConsoleConditional = ( print:boolean = false ): Function => {

    if ( print ) {
        return printToConsole
    } else{
        return () => {}
    }
}

@printToConsoleConditional( true )
export class Pokemon {
```

Gracias a los decoradores de fabrica podemos cambiar la funcionalidad de una clase, regresar una nueva instancia y hacer varias cosas más.

Recordemos nuevamente que siempre deben devolver una función y se ejecutan con el paréntesis ()

```
@printToConsoleConditional( true )
```

A diferencia de los decoradores de clase que solo se pone el decorador:

```
@printToConsole
```

97. Ejemplo de un decorador – Bloquear prototipo

A veces en lugar de expandir un objeto queremos bloquear esa posibilidad, para eso, debemos bloquear el prototipo:

```
const bloquearPrototipo = function ( constructor: Function ){
```

```
Object.seal( constructor );
Object.seal( constructor.prototype );
}
```

Ahora debemos anidar los decoradores, pero es importante saber que se hacen de manera secuencial y es posible añadir tantos como se necesiten:

```
@bloquearPrototipo
@printToConsoleConditional( true )
export class Pokemon {
```

Veamos cómo se vería esta posibilidad bloqueada y el error que aparecerá:

```
(Pokemon.prototype as any).customName = 'Pikachu'
```

```
✖ ▶ Uncaught TypeError: Cannot add      index.ts:5
  property customName, object is not extensible
    at Object.<anonymous> (index.ts:5:1)
    at ./src/index.ts (index.ts:7:2)
    at __webpack_require__ (bootstrap:19:1)
    at startup:4:1
    at startup:5:1
```

Si no se hubiera bloqueado el constructor, bien se podría modificar el objeto.

98. Decoradores de métodos

Ya vimos que podemos implementar cierto tipo de bloqueo que llega hasta JS. Ahora veamos los decoradores para los métodos.

A veces necesitamos hacer validaciones con los datos que se reciben en los métodos, para no repetir la misma lógica varias veces se utilizan los decoradores de métodos

```
function checkValidPokemonId( ) {
  return function( target: any, propertyKey: string, descriptor: PropertyDescriptor ){
    console.log( {target, propertyKey, descriptor});
  }
}
```

```
pokemon-class.ts:28
  ↴ {target: {...}, propertyKey: 'savePokemonToDB',
    descriptor: {...}} ⓘ
    ▶ descriptor: {writable: true, enumerable: false}
      propertyKey: "savePokemonToDB"
    ▶ target: {constructor: f, savePokemonToDB: ...}
    ▶ [[Prototype]]: Object
```

Veamos cómo se utiliza:

```
@checkValidPokemonId()
savePokemonToDB( id:number ){
    console.log(`Pokemon guardado en la BD ${ id }`);
}
```

Las validaciones se ejecutan primero en el decorador del método y después continua con el orden de la ejecución:

```
function checkValidPokemonId( ) {
    return function( target: any, propertyKey: string, descriptor: PropertyDescriptor ){

        const originalMethod = descriptor.value;

        descriptor.value = ( id:number ) => {
            if ( id < 1 || id > 800 ) {
                return console.error('El id del Pokemon debe estar entre 1 y 800');
            } else {
                return originalMethod( id );
            }
        }

        console.log( {target, propertyKey, descriptor});
    }
}
```

Veamos como lo utilizamos:

```
charmander.savePokemonToDB( 4 );
```

Si cumple con las validaciones continuará con la ejecución, de lo contrario se mostrará un error. Por último, el descriptor solo se recibe cuando vamos a decorar un método.

99. Decoradores de propiedades

En JS aún no están completamente implementadas las propiedades privadas, por lo cual, si se hace un cambio en TS aunque una propiedad este como privada si llega hasta JS.

Es importante recordar que dependiendo donde utilicemos el decorador, podemos recibir:

- **Un constructor – Decorador de Clase**

```
function printToConsole( constructor: Function ){
    console.log( constructor );
}
```

```
@printToConsole
export class Pokemon {
```

- Un target, propertyKey y descriptor – Decorador de Método

```
function checkValidPokemonId( ) {
    return function( target: any, propertyKey: string, descriptor: PropertyDescriptor ){
        console.log( {target, propertyKey, descriptor});
    }
}
```

```
@checkValidPokemonId()
savePokemonToDB( id:number ){
    console.log(`Pokemon guardado en la BD ${ id }`);
}
```

- Un target y propertyKey – Decorador de Propiedad

```
function readOnly( isWritable:boolean = true ): Function{
    return function( target: any, propertyKey: string ){
        ...
    }
}
```

```
@printToConsoleConditional( false )
export class Pokemon {

    @readOnly()
    public publicApi:string = 'http://pokeapi.co' ;
```

Cuando decoramos una propiedad, retornamos un **PropertyDescriptor**, lo que significa que debemos crearlo.

Por cierto, la razón por la que se usan funciones tradicionales al crear un **factory decorator** es porque el **this** cambia a lo que apunta cuando se utilizan funciones de flecha.

```
function readOnly( isWritable:boolean = true ): Function{
    return function( target: any, propertyKey: string ){

        const descriptor: PropertyDescriptor = {

            get(){
                console.log( this, 'getter' );
                return 'Hola mundo';
            },
        };
    }
}
```

```

        set( this, val ){
            // console.log( this, val );
            Object.defineProperty( this, propertyKey, {
                value: val, // Es el valor que se recibe inicialmente
                writable: !isWritable, // Para que no se pueda escribir
                enumerable: false // Para que no se pueda ver
            })
        }
    }

    return descriptor;
}
}

```

Al decorar la propiedad, no se podrá modificar la propiedad:

```

@bloquearPrototipo
@printToConsoleConditional( false )
export class Pokemon {

    @readonly(true)
    public publicApi:string = 'http://pokeapi.co';

    constructor(
        public name:string
    ){
    }
}

```

Para finalizar recordemos:

- Los decoradores de pueden encadenar
- Los decoradores se generan cuando se transpila el código
- Los decoradores se crearon para una lectura más fácil del código
- Existen diversos tipos de decoradores como: De clase, Factory Decorator, De método, De propiedad, etc.

100. Código fuente de la sección

Aquí les dejo el código fuente de la sección por si lo llegan a necesitar en algún momento o bien para compararlo contra el mío:

- [Github - Fin-seccion-12](#)

También lo pueden descargar del material adjunto.

Sección 13: Usando librerías que no están escritas en TypeScript(Como jQuery)

101. ¿Qué veremos en esta sección?

Sabemos muy bien que nuestras aplicaciones web, no serán programadas únicamente con TypeScript puro, por lo cual es importante aprender cómo utilizar librerías de terceros en nuestros proyectos de TypeScript.

Puntualmente aprenderemos sobre:

1. Configuración de un proyecto utilizando el package.json y realizar instalaciones con node
2. Utilizar archivos de definiciones "*.d.ts" o Typings
3. Agregar definiciones de archivos mediante node

102. Inicio de proyecto – Express API

- Para este punto debemos crear una nueva carpeta.
- Luego iniciamos el proyecto con el comando: npm init
- Después de llenar los campos se crea un archivo: package.json
- Crear un archivo base index.js con el código: **console.log('Hola Mundo');**
- Ejecutar el archivo: node index

103. Creando un Rest API con Express

Para esta parte utilizaremos Express: <https://expressjs.com/>

Ejecutamos el comando: npm install express --save

Ahora reemplazamos el index.js con el siguiente contenido:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {

  res.send('Hello World!')
})
```

```
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Este es solamente un ejemplo de un servidor web, pero esto está en JS. Si quisiéramos pasarlo a TS no basta con reemplazar el archivo de *.js a *.ts, sino que se necesita un archivo de definición de TS que tienen la extensión *.d.ts.

Estos archivos de definición ayudan para el autocompletado.

104. Trabajar con TypeScript en lugar de JavaScript

Los archivos de distribución deben ponerse en la carpeta **/dist** por recomendación.

Ahora debemos crear un archivo de definición de TS con el comando: **tsc -init**

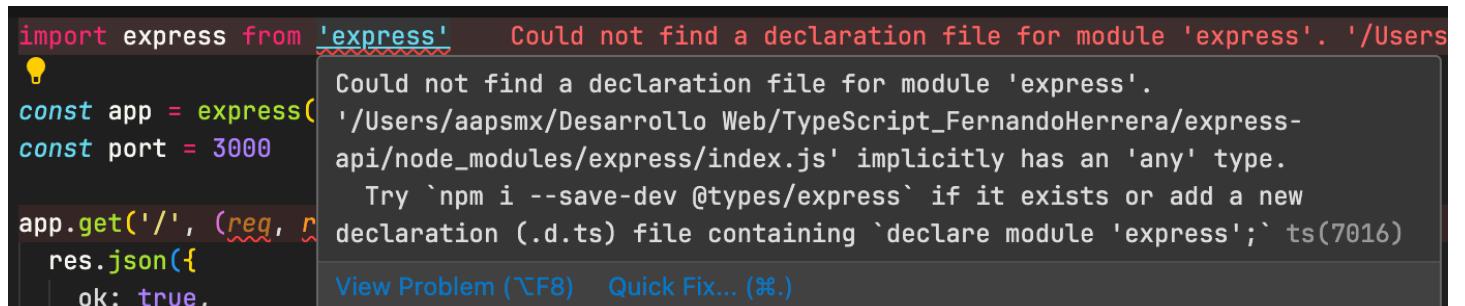
Modificamos la configuración del archivo **tsconfig.json** según nuestras necesidades:

```
"outDir": "./dist",                                     /* Specify an output folder for all emitted files. */
```

Ahora TS mostrará los errores que ha detectado el archivo.

Si hacemos una compilación con el comando: **tsc**, se genera el archivo final en la carpeta dist: **index.js**

Ahora se hace el cambio para la importación de express pero vemos que aparece este error:



```
import express from 'express'      Could not find a declaration file for module 'express'. '/Users
  const app = express()
  const port = 3000

  app.get('/', (req, res) =>
    res.json({
      ok: true,
```

Esto es porque no existe un archivo de definición de TS para express, sin embargo, también nos da la solución que es ejecutar el comando: **npm i --save-dev @types/express**

También en MAC podemos presionar el botón CMD y punto sobre ‘express’ para recibir la solución directa.

Entonces, se instalará la dependencia que podemos confirmar en el archivo **package.json**:

```
"devDependencies": {  
  "@types/express": "^4.17.17"  
}
```

Ahora ya tendremos la ayuda de TS y el autocompletado y si ejecutamos el comando **tsc**, se genera sin errores el archivo de salida index.js.

Entonces, ya podemos correr el servidor web de express empleando TS:

- **node dist/index.js**

Sin embargo, para cada cambio siempre debemos generar nuevamente el archivo de salida con el comando:

- **tsc**

Y luego tenemos que volver a levantar el servidor de express con node:

- **node dist/index**

Existe otra forma para no tener que hacer esto manualmente, pero eso es otra configuración adicional a este curso y corresponde más a Node.

Por último, debemos recordar que los archivos de definición de TS tienen esta estructura: `@types/loqueseTipa`,

```
"@types/express": "^4.17.17"
```

Cuando una librería fue escrita en TS tiene este icono:



Y cuando una librería tiene archivo de definición utiliza este icono:



Hasta aquí, hemos estudiado sobre TS. Estas son las bases y aún se puede profundizar más, sin embargo, eso lo dará sobre todo la práctica.

Por lo tanto, desde ahora a desarrollar usando el Set de super poderes de JS llamada TypeScript.

Sección 14: Final del curso

105. Más sobre mis cursos

Otros cursos impartidos por mí

Aquí pueden encontrar todos mis cursos al menor precio posible que puedo dejarlos, todo el año

[Cursos relacionados a TypeScript](#)

Subir el certificado de Udemy:

Pueden subir su certificado que les genera Udemy a mi página web, y participar en promociones, regalos, otras cosas que se me ocurran, simplemente descarguen el certificado que les genera Udemy y subanlo aquí:

[Logros de Alumnos - Fernando Herrera](#)

106. Despedida del curso

Agradecimiento y petición para compartir este curso que es una regrabación.

107. Recursos adicionales disponibles

Recursos adicionales disponibles

Si deseas explorar otro contenido creado por Fernando Herrera, te invitamos a que visites los siguientes enlaces:

Web Personal: <https://fernando-herrera.com>

Podcast: <https://anchor.fm/fernando-her85>

Twitter: https://twitter.com/Fernando_Her85

YouTube: <https://www.youtube.com/@DevTalles>

Perfil de instructor | Udemy: <https://www.udemy.com/user/550c38655ec11/>

{d/t} DevTalles: <https://cursos.devtalles.com>

{d/t} DevTalles LinkedIn: <https://www.linkedin.com/company/devtalles>

{d/t} DevTalles Twitter: <https://twitter.com/DevTalles>

{d/t} DevTalles-Recursos gratuitos: <https://cursos.devtalles.com/pages/mas-talento>

{d/t} DevTalles-Programas de Estudio: <https://cursos.devtalles.com/pages/programas-de-estudio>