Índice

Typescript....2
Tipos de datos....3
Union....5
Arrays....6
Interfaces....7
Funcións....8

Typescript

Typescript é unha linguaxe de programación baseada en JavaScript que ten coma obxectivo a compilación ou transformación (transpilado) de dito código en código JavaScript. É código aberto e foi creado por Microsoft. Engade características adicionais a JavaScript coma o tipado estático (podo ter tipos de datos).

Para traballar con Typescript, como hai que compilar código, deberemos instalar a nosa contorna de traballo. Para elo comezaremos con node, que será a contorna de execución. Para elo deberás instalalo no teu equipo:

https://nodejs.org/es/download

Tamén pode ser interesante traballar con:

nvm (para utilizar diferentes versións de node).

npm (package manager ou manexador de paquetes).

Podes instalalos dende a respectiva páxina web oficial. Unha vez instalados poderemos comprobar se todo está correcto accedendo ás versións de software que temos:

```
node -v
npm -v
```

Agora xa podemos instalar Typescript:

```
npm i typescript
```

`npm i typescript -g` (se queremos que se instale a nivel global, a nivel de S.O.)

Por regra xeral debería indicarnos se todo foi correcto e a versión de Typescript que se instalou, de todos xeitos, podemos comprobalo tamén con:

```
tsc --version
```

Xa podemos comezar coa escritura do noso código en Typescript. Para elo podemos crear un ficheiro sinxelo coa extensión `.ts`. Por exemplo podemos escribir un Ola Mundo:

```
console.log('Ola Mundo');
```

Para compilalo non teremos máis que executar:

```
tsc ficheiro.ts
```

E crearase un arquivo `ficheiro.js` co código do `ficheiro.ts` compilado/transpilado a Javascript entendible polos navegadores.

Existe un modo "watch" para non ter que andar compilando ou transpilando cada vez que facemos algún cambio no arquivo Typescript. Para elo só teremos que lanzalo así:

```
tsc ficheiro.ts --watch
```

Tipos de datos

A continuación veremos os tipos de datos básicos de Typescript:

- Booleanos: poden ser true ou false (non 0 ou 1). Para definir un booleano faremos:

```
var meuBool: boolean;
meuBool = true;
```

  Ollo porque agora non podemos facer isto:

```
var meuBool: boolean;
meuBool = true;
meuBool = 2;
```

  Isto daría un error de tipo e o editor (Visual Studio Code) xa nos daría un erro. Pero incluso o compilador en modo watch diría o seguinte:

```
app.ts:3:1 - error TS2322: Type 'number' is not assignable to type 'boolean'.

3 meuBool = 2 ;
 ~~~~~~~

[16:54:31] Found 1 error. Watching for file changes.
```

- Numbers: só existe un tipo numérico, non hai nin floats, nin integer, nin... Para definir un número faremos:

```
var meuNum: number;
meuNum = 2;
```

  E o mesmo que no caso anterior, xa non poderemos mudar o tipo do dato.

  Pódense, tamén, definir tipos de datos dun xeito implícito:

```
var meuNum = 2;
```

- String: Para definir unha cadea faremos:

```
var cadea: string;
cadea = 'Ola Mundo';
```

  Podemos inicializala con comiñas simples (‘), comiñas dobres (“) ou comiñas “francesas” ou *back tick* (`) como vimos coas ***Template String*** de ECMA6. Ao facer o transpilado pode que transforme estas últimas (back tick) por comiñas dobres, dependendo da versión de JS que indiquemos (por defecto penso que usa ECMAScript 5).

  E o mesmo que no caso anterior, xa non poderemos mudar o tipo do dato.

- Any: tipo de dato xenérico:

```
var elemento: any;
elemento = 'Ola';
```

Que pasa, que a partir deste momento non poderemos acceder ás propiedades específicas de cada tipo de dato (cadeas, números, booleanos,...).

- Type: serve para definir os nosos propios tipos. Podemos pórlle o nome que queiramos ao tipo pero tendo sempre coidado de non utilizar palabras reservadas. Deste xeito pódese, por exemplo, xeneralizar elementos (obxectos) moi utilizados e protexelos fronte a declaracións incorrectas:

```
type Persoa = {
      nome: string;
}

const p1 = {
      nome: 'Gerardo',
}

const p2 = {
      nome: 'Susana',
}
```

Aínda que isto está moi relacionado co tema de obxectos (que veremos máis adiante) tamén se poden definir tipos con elementos opcionais do seguinte xeito:

```
type Persoa = {
      nome: string;
      age?: number;
}
```

E para utilizalo:

```
const p1: Persoa = {
      nome: 'Gerardo',
}
```

Aos que tedes máis experiencia coa teoría de obxectos isto seguro que vos sona a ***Interfaces***. Certo. Pero iso verémolo máis adiante.

Union

Podo “unir” directamente diferentes tipos creados por min mesmo co carácter &:

```
type NomePersoa = {
      nome: string;
}

type IdadePersoa = {
      idade: number;
}

type PersoaOla = {
      DiOla: () => void;
}

type Persoa = NomePersoa & IdadePersoa & PersoaOla;

const p: Persoa = {
      nome: 'Gerardo',
      idade: 47,
      DiOla: function () {
            console.log('Ola');
      }
}
```

E xerarase o seguinte JS:

```
var p = {
      nome: 'Gerardo',
      idade: 47,
      DiOla: function () {
            console.log('Ola');
      }
};
```

Así como existe a Unión de tipos (&) existe o Or de tipos (|). Creo que podedes facervos unha idea do que faría nese caso, que non é máis que comprobar que se cumpre con algún dos tipos indicados.

Arrays

Os Arrays en Typescript se definen do seguinte xeito:

```
var meuArray: [];
```

Neste caso teremos un Array baleiro de “any” elementos, xa que non definimos ningún tipo de dato. Ao igual que JavaScript poderemos engadirlle valores dun xeito dinámico:

```
var meuArray = [1, 'ola', true];
```

Ou tamén:

```
var meuArray: any[] = [1, 'ola', true];
```

Como estamos a ver podemos crear un Array de elementos heteroxéneos (*number*, *string*, *boolean*) e, incluso, engadir novos elementos tamén heteroxéneos:

```
meuArray.push(2);

meuArray.push('adeus');
meuArray.push(false);

console.log(meuArray.length); // Amosaría 6 elementos
```

De todos xeitos, xa que estamos con Typescript será mellor que aproveitemos as súas capacidades e que usemos os tipos de datos do seguinte xeito, por exemplo:

```
var meuArrayTs: string[] = ['Volvo', 'Honda'];

console.log(meuArrayTs); // Amosaría [ 'Volvo', 'Honda' ]
```

Ollo por que a partires deste momento xa non poderemos crear un Array heteroxéneo coma o do primeiro exemplo. Así que isto daría un erro:

```
meuArrayTs.push(1);
```

```
04_arrays.ts:13:17 - error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.
13 meuArrayTs.push(1);
                   ~
Found 1 error in 04_arrays.ts:13
```

Pero si que me permitiría realizar o seguinte:

```
meuArrayTs.push('Toyota');

console.log(meuArrayTs); // Amosaría [ 'Volvo', 'Honda' ]
```

Interfaces

Moi semellante ao Type pero os programadores cunha formación en OO máis pura preferirán este elemento fronte ao outro. Ao estar máis ligado a esta metodoloxía de programación xa che obriga a definir unha estrutura dun obxecto.

```
interface Persoa {
	nome: string;
	age?: number;
}
```

Pódense incluír funcións ou métodos como atributos do prototipo/contrato/estrutura:

```
interface Persoa = {
	nome: string;
	age?: number;
	diOla(): void;
}
```

E para utilizalo:

```
const p1: Persoa = {
	nome: 'Gerardo',
	diOla: function () {
		console.log('Ola');
	}
}
```

Funcións

A diferenza inicial entre as funcións en *JavaScript* e as funcións en *Typescript* é a declaración do valor/tipo que devolven:

```
function nomeFuncion(): tipoRetorno {
}
```

Poderán devolver (**`tipoRetorno`**) **`void`** (ou **`never`**) ou calquera dos tipos vistos nos apartados anteriores. En canto aos parámetros podemos dicir o seguinte:

```
function suma(a: number,b: number): number {
	return a+b;
}
const resultado = suma(2,3);
```

É dicir, exactamente igual que en *JavaScript* salvo o tema dos tipados nos parámetros e no retorno.

Se queremos indicar que algún parámetro é opcional podemos facelo do seguinte xeito:

```
function suma(a: number,b: number | undefined): number {
	return a+b;
}
```

Ollo porque na chamada teremos, tamén, que utilizar o undefined, non podemos deixalo baleiro:

```
const resultado = suma(2,undefined);
```

Unha alternativa a isto podería ser utilizar o interrogante que xa usamos na definición de tipos:

```
function suma(a: number,b?: number): number {
	return a+b;
}
const resultado = suma(2);
```

Deste xeito xa non é preciso andar co **`undefined`** nin na definición nin na chamada.

Ollo porque precisamente neste exemplo estamos tratando de sumar un **`number`** con un **`undefined`**, co cal o resultado non será un **`number`**. Se queres controlar iso terás que utilizar os parámetros con valores por defecto (igual que facíamos en *JavaScript*).

```
function suma(a: number,b?: number = 0): number {
	return a+b;
}
```

Poderemos definir un tipo función do seguinte xeito:

```
let miFuncion: Function;
function suma(a, b) {
      return a + b;
}
function diOla() {
      console.log('Ola');
}
let variable = 1;
if (variable === 2) {
      miFuncion = suma;
} else {
      miFuncion = diOla;
}
miFuncion(1,2);
```

Como se pode ver podemos, en tempo de execución, executar unha ou outra función. A única pega que ten isto é que, ao utilizar un tipo xenérico, non se fai comprobación de parámetros, por exemplo. Para iso teríamos que realizar o seguinte:

```
let miFuncion: (a: number, b: number) => number;
function suma(a, b) {
      return a + b;
}
function diOla() {
      console.log('Ola');
}
let variable = 1;
if (variable === 2) {
      miFuncion = suma;
} else {
miFuncion = diOla;
}
miFuncion(1,2);
```

Coa función frecha específica xa se fai unha comprobación de parámetros e xa da un erro na compilación/transpilación.

```
error TS2322: Type '() => void' is not assignable to type '(a: number, b: number) =>
number'.
Type 'void' is not assignable to type 'number'.
16     miFuncion = diOla;
[10:58:35] Found 1 error. Watching for file changes.
```

Por último teríamos que falar de **`never`**. Poderíamos dicir que se trata dunha variante do **`void`** coa diferenza de que **`never`** se utiliza cando se queren lanzar erros.

```
function xeraUnErro (m: string): never {
      throw new Error(m);
}

xeraUnErro('Ocurriu un erro');
```