TYPESCRIPT

Iniciamos aquí una serie de artículos dedicados a TypeScript. En este post vamos a describir qué y cómo es TypeScript, y por qué lo consideramos adecuado para incluirlo en el ámbito de HTTP Masters.

TypeScript es un lenguaje publicado por Microsoft en 2012. Al igual que ya han hecho en alguna ocasión con otros productos (y en contra de la política general de esta firma), es de código abierto y de uso libre. Se trata de una potenciación de JavaScript. De hecho, TypeScript es a JavaScript lo mismo que SASS es a CSS:

- Permite integrar nuevas características a JavaScript.
- No es directamente usable en nuestros documentos web, sino que debe ser transpilado, mediante la correspondiente herramienta, a JavaScript.
- Incorpora prestaciones y funcionalidades muy fáciles y cómodas de gestionar, que en JavaScript puro y duro serían de difícil uso y desarrollo y que, al transpilarse, se implementan correctamente.

Y, lo más importante, es una herramienta fundamental a la hora de trabajar con Angular JS, desde la versión 2 de este framework (actualmente se emplea la 4, y está en fase de implementación la 5).

Este artículo es meramente introductorio e inicia una andadura que nos permitirá mejorar nuestra calidad como desarrolladores.

INTRODUCCIÓN

¿Por dónde empezamos a conocer TypeScript? Sin duda, la página oficial es el mejor punto de partida. La encuentras en https://www.typescriptlang.org/. Lo primero que deberías hacer es echarle un vistazo general rápido a la documentación que allí encontramos, en https://www.typescriptlang.org/docs/home.html.

Las principales características de TypeScript son:

- Extiende la sintaxis de JavaScript. Eso significa que cualquier código escrito en JavaScript (incluso en la versión 6) funciona sin problemas.
- Al contrario que JavaScript, es fuertemente tipado. Un dato es de un tipo, y no puede ser de otro. Además, el tipado de datos se define en el contenedor (al estilo de Java), no inherentemente en el dato (cómo por ejemplo, al estilo de Python).
- A pesar del tipado, nos proporciona una vía de escape para poder saltárnoslo (aunque, si somos coherentes, deberemos acostumbrarnos a usarla lo menos posible).
- Implementa el paradigma POO de una forma real, con todas las características de este modo de programación.
- Está concebido para proyectos de gran envergadura, en los que la programación en JavaScript sería engorrosa e insostenible. En TypeScript, en cambio, es mucho más manejable.

- Es de muy fácil aprendizaje, y nos va a dar muchas facilidades, ahorrándonos gran cantidad de horas en nuestro día a día.

LO QUE NECESITAMOS

Antes de empezar a trabajar con TypeScript necesitamos tener el propio TypeScript instalado en nuestro ordenador. Por "el propio TypeScript" me refiero a la herramienta que permitirá, a partir de un código TypeScript, obtener su transpilación a JavaScript.

TypeScript se instala con el Node Package Manager (npm), de NodeJS. Si has seguido las series de artículos de HTTP Masters, sin duda ya tiene npm en tu ordenador. Si no lo tienes, entra en https://nodejs.org/es/ y descarga e instala la versión correspondiente a tu sistema operativo. Si estás trabajando con un ordenador diferente al tuyo habitual, puedes confirmar si npm está instalado o no, tecleando, en la consola de mandatos, lo siguiente:

```
npm -v
```

Si te responde con el número de versión, npm está está correctamente instalado. Si no, te dará un mensaje de error, informándote de que npm no es un comando.

El siguiente paso, si npm ya está instalado, es comprobar si está instalado TypeScript. Para ello, teclea en la consola:

```
tsc -v
```

Si TypeScript está instalado, te devolverá el número de versión. En caso contrario, deberás instalarlo tecleando:

```
npm install -g typescript
```

El modificador -g nos va a permitir que TypeScript esté disponible de forma global desde cualquier parte de nuestro equipo.

La versión actual de TypeScript, en el momento de escribir estas líneas, es la 2.7.2. Si tú tienes una versión anterior, puedes actualizarte con la siguiente línea:

```
npm update -g typescript
```

Lo siguiente que necesitamos es instalar los TSD de NodeJS. Los TSD (TypeScript Definitions) son un "puente" que permitirá que TypeScript se entienda, a la hora de comunicar datos, con diferentes librerías y frameworks de JavaScript, como pueden ser jQuery, Angular, Backbone, etc. Así pues, los TSD los vamos a instalar también, globalmente, en nuestro equipo. Lo primero, es comprobar si ya los tenemos, así:

```
tsd --version
```

En caso de mensaje de error, no los tenemos, los instalaremos así:

```
npm install -g tsd
```

Una vez instalados, si necesitamos actualizarlos (la versión actual, en el momento de escribir estas líneas, es 0.6.5), teclearemos:

```
npm update -g tsd
```

También debemos instalar el inicializador de TypeScript de forma global a nuestro equipo. Para ello, en la consola de mandatos teclearemos:

```
npm install tsc-init -g
```

CONCLUYENDO

Con esto ya tenemos en nuestro ordenador, todo lo que necesitamos para empezar a trabajar con TypeScript. En el próximo artículo empezaremos a ver cómo debemos preparar individualmente cada proyecto en el que vayamos a usar TypeScript, y montaremos un pequeño ejemplo para empezar a familiarizarnos con esta herramienta.

ATENCIÓN. Todos los pasos descritos en este artículo debes llevarlos a cabo SÓLO un vez en tu equipo. Son un poco engorrosos, pero es algo que sólo tienes que hacer una vez, para todo tu trabajo con TypeScript. En el próximo artículo veremos los pasos que hay que llevar a cabo en cada proyecto individual que emplee TypeScript.

Hola, Mundo con TypeScript

En el artículo anterior vimos un breve resumen, por encima, de lo que es TypeScript, y aprendimos a preparar nuestro ordenador para trabajar con este lenguaje.

En este artículo, vamos a ver cómo crear nuestro primer código con TypeScript, cómo transpilarlo a JavaScript, y a exponer algunos conceptos fundamentales, como son, por ejemplo, las opciones de compilación.

Vas a ver que trabajar con TypeScript es fácil y, dadas las prestaciones que ofrece, sumamente interesante. Espero que este lenguaje te guste tanto como a mí. Algunas veces, los de Microsoft hacen algo bien :-P . De hecho, para este tipo de proyectos, yo empleo, cada vez con más frecuencia, el Visual Studio Code, un IDE gratuito, bastante bueno, de los chicos de Redmond, que puedes obtener en este enlace. Sin embargo, en estos artículos nos centraremos en el código, no en el funcionamiento del IDE, salvo que, puntualmente, haya algún aspecto específico que me parezca interesante comentar.

EL ESCENARIO

Vamos a preparar un directorio en el que meter nuestro primer proyecto TypeScript. Lo vamos a hacer a partir de la raíz de nuestros proyectos web. En mi caso, como trabajo con Windows y Xampp, la ruta raíz de trabajo es c:/xampp/htdocs/. Si tú usas, por ejemplo, Wamp, tu ruta raíz podría ser c:/wamp/www/. En un equipo Linux, por ejemplo, seria /var/www/html/. En cualquier caso, vamos a crear un directorio llamado hola_mundo_ts.

Lo primero, al igual que cuando trabajamos con gulp, es inicializar el package.json, tecleando, en la consola (en la raíz de nuestro proyecto), lo siguiente:

```
npm init -y
```

A continuación, lo que tenemos que hacer es instalar el transpilador de TypeScript de forma local al proyecto. En el artículo anterior aprendimos a instalarlo de forma global al equipo (con el modificador -g). Sin embargo, es posible que tengas problemas para usarlo desde un proyecto concreto, por lo que, en cada nuevo proyecto, lo instalaremos de forma local al mismo. Es un poco al estilo de lo que ocurre con gulp. En teoría, teniéndolo instalado globalmente funciona sin problemas, pero, en algún caso concreto, podría no ser así. Por lo tanto, una vez creado el directorio del proyecto, abre la consola de mandatos y teclea:

```
npm install typescript
```

Ahora teclea en la consola (sin salir de la raíz de tu proyecto), lo siguiente:

```
tsc --init
```

La consola te responderá algo parecido a lo siguiente:

```
message TS6071: Successfully created a tsconfig.json file.
```

Te informa de que se ha creado un fichero llamado tsconfig.json. Ábrelo con tu editor de textos favorito. Casi al principio vas a encontrar la siguiente línea:

```
"target": "es5",
```

Modifícala para que quede así:

```
"target": "es2015",
```

Esto te permitirá transpilar tus códigos TypeScript al estándar ES6, en lugar de a versiones anteriores de JavaScript.

ATENCIÓN. Transpilar a ES6 (JavaScript 2015) tiene sus ventajas y sus inconvenientes. A día de hoy las últimas versiones de los navegadores soportan perfectamente este estándar. Sin embargo, si crees que tu usuario target medio puede estar empleando, por ejemplo, un IE10, o versiones relativamente antiguas de los navegadores más habituales, deberás dejar en valor del atributo target en es5. Eso producirá, al transpilar a JavaScript, un código final más "tosco" (aunque perfectamente operacional). Sin embargo, si piensas que la mayoría de los visitantes de tu web tendrán navegadores actualizados, puedes ponerlo como hemos indicado, para transpilar a ES6 (te valen los valores es2015 o, en su lugar, es6).

ATENCIÓN. Lo que hemos hecho hasta ahora es preparar el proyecto para poder emplear TypeScript. Estos pasos sólo debes llevarlos a cabo una vez por cada proyecto nuevo. A partir de ahora, tu proyecto ya está preparado para usar TypeScript y poder transpilarlo a JavaScript.

HOLA, MUNDO

Vamos a crear, en la raíz de nuestro proyecto, un simple archivo de pruebas para ver de qué estamos hablando. Lo primero es crear un simple index.html:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Document</title>
</head>
<body>
   <script language='javascript' src='hola_mundo.js'></script>
   <script language='javascript'>
      saludar();
   </script>
</body>
</html>
```

Como ves en la primera línea resaltada (la 10), se llama a un archivo JavaScript que aún no existe, por lo que, si tratas de cargar esta página en tu navegador, no cargará nada. Abre la consola de tu navegador, porque la usaremos en seguida. Estás viendo un error, porque la línea 12 llama a una función (saludar()), que aún no existe.

Ahora crea un archivo llamadohola_mundo.ts (los archivos de TypeScript siempre se guardan con la extensión .ts). Su código es tan simple como el siguiente:

```
function saludar() {
    console.log("Hola, Mundo");
}
```

Como ves, no tiene nada especial. La verdad, cómo código es un muy soso, y la sintaxis tampoco aporta nada nuevo, de momento. Es sólo sintaxis vulgar de JavaScript de toda la vida. Aun así, nos va a venir bien para ver el proceso.

En la consola de mandatos de tu ordenador (que está en tu directorio del proyecto), teclea:

```
tsc hola_mundo
```

Al cabo de unos segundos, se ha transpilado el código, y se ha creado un archivo llamado hola_mundo.js. Como ves, dada la simplicidad extrema de esta primera prueba, el código es el mismo que el original en TypeScript, pero ya es un archivo JavaScript. Si ahora recargas tu página HTML, ves en la consola del navegador el saludo Hola, mundo. Ya no hay errores, y ha funcionado como esperábamos.

ATENCIÓN. Aunque el código de este primer ejemplo es tan simple que no hay diferencia entre la sintaxis de TypeScript y JavaScript, observa que el transpilador inicia el archivo JavaScript con la instrucción "use strict"; lo cual es algo bueno, ya que, por razones de buenas prácticas de programación, debemos acostumbrarnos a "forzar" a JavaScript a que nos exija siempre la declaración de variables. Esto es algo que, seguramente, ya conoces (o deberías), pero, si deseas conocer más al respecto, puedes leer este artículo.

OPCIONES DE TRANSPILACIÓN

El ejemplo anterior ha sido muy pobre (yo diría casi decepcionante), en el sentido de que no nos ha enseñado nada específicamente sobre TypeScript. El objetivo era, simplemente, empezar a familiarizarnos con la idea de crear un archivo en TypeScript y ver qué y cómo es la transpilación. Sin embargo, no tiene ningún sentido, en un proyecto, tener en el directorio raíz del mismo todos los archivos TypeScript que vayamos a usar, y sus correspondientes transpilaciones.

Vamos a modificar la estructura de nuestro proyecto, de forma que tengamos, dentro de la raíz del mismo, al menos dos directorios, llamados ts y js. En el primero almacenaremos los archivos TypeScript y le diremos al transpilador que almacene los JavaScript en el segundo directorio. Por lo tanto, vamos a empezar por eliminar hola_mundo.js (el JavaScript), y mover hola_mundo.ts al directorio ts.

A continuación, modificamos la línea 10 de index.html, sustituyéndola por la siguiente:

```
<script src='js/hola_mundo.js' language='javascript'></script>
```

Como ves, lo que hacemos es especificar la ruta relativa donde deberá buscarse el archivo JavaScript.

Ahora, en la consola del ordenador, y desde la raíz del proyecto, tecleamos:

```
tsc --outDir js
```

Al no especificar un nombre de fichero TypeScript, el transpilador busca todos los de este tipo que hay en todas las ubicaciones desde la raíz del proyecto, de forma recursiva, con lo que va a encontrar nuestro archivo ts/hola_mundo.ts. La opción --outDir nos permite especificar el directorio de salida de las transpilaciones que, en este ejemplo, es js. Si el directorio no existe, el transpilador lo creará.

ATENCIÓN. El transpilador puede grabar los archivos JavaScript de salida en el directorio que le indiquemos y, cómo hemos mencionado, si no existe dicho directorio, este será creado. Sin embargo, si trabajas en un sistema Linux, recuerda que el directorio raíz de tu proyecto debe tener los adecuados permisos de escritura. En entornos Windows no debes, en principio, preocuparte de esto.

Otra opción interesante es la vigilancia permanente de archivos TypeScript. Al igual que en gulp podíamos programar una tarea para que se ejecutara cada vez que hacíamos cambios en los archivos "vigilados", el transpilador de TypeScript puede lanzarse para que se quede en modo vigilancia, de modo que, si modificamos alguno de nuestros códigos TypeScript se regeneren los archivos JavaScript correspondientes. Para hacer esto, escribe, en la consola de tu ordenador el siguiente mandato:

```
tsc --outDir js -w
```

Se inicia un proceso de transpilación en tiempo real, de modo que, si ahora cambias lo que sea en el archivo original de TypeScript y grabas los cambios (o si tu editor está en modo de "guardado automático"), el archivo JavaScript se actualiza automáticamente. Esto es muy útil durante la fase de desarrollo, cuando estamos corrigiendo constantemente cosas. El inconveniente (si quieres llamarlo así), es que la consola del sistema queda bloqueada, no permitiéndote teclear otros mandatos. Si necesitas interrumpir la transpilación vigilada y recuperar el control de la consola puedes hacerlo con CTRL-C.

Las opciones del transpilador que acabamos de ver son las más llamativas, y las que más usaremos, por lo menos en los primeros artículos. Más adelante iremos viendo otras opciones. De todos modos,

si quieres echarle un vistazo general a las posibilidades de esta herramienta, puedes teclear en la consola la siguiente línea:

tsc --help

La respuesta será parecida a la siguiente:

```
Version 2.7.1
 Syntax:   tsc [options] [file ...]
  Examples: tsc hello.ts
        tsc --outFile file.js file.ts
        tsc @args.txt
  Options:
  -h, --help                                 Print this message.
  --all                                      Show all compiler options.
  -v, --version                              Print the compiler's version.
  --init                                     Initializes a TypeScript project and creates a tsconfig.json file.
  -p FILE OR DIRECTORY, --project FILE OR DIRECTORY  Compile the project given the path to its
configuration file, or to a folder with a 'tsconfig.json'.
  --pretty                                   Stylize errors and messages using color and context (experimental).
  -w, --watch                                Watch input files.
  -t VERSION, --target VERSION               Specify ECMAScript target version: 'ES3' (default), 'ES5',
'ES2015', 'ES2016', 'ES2017','ES2018' or 'ESNEXT'.
  -m KIND, --module KIND                     Specify module code generation: 'none', 'commonjs', 'amd',
'system', 'umd', 'es2015', or 'ESNext'.
  --lib                                      Specify library files to be included in the compilation.
                                             'es5' 'es6' 'es2015' 'es7' 'es2016' 'es2017' 'es2018' 'esnext' 'dom'
'dom.iterable' 'webworker' 'scripthost' 'es2015.core' 'es2015.collection' 'es2015.generator'
'es2015.iterable' 'es2015.promise' 'es2015.proxy' 'es2015.reflect' 'es2015.symbol'
'es2015.symbol.wellknown' 'es2016.array.include' 'es2017.object' 'es2017.sharedmemory'
'es2017.string' 'es2017.intl' 'es2017.typedarrays' 'esnext.array' 'esnext.asynciterable' 'esnext.promise'
  --allowJs                                  Allow javascript files to be compiled.
  --jsx KIND                                 Specify JSX code generation: 'preserve', 'react-native', or 'react'.
  -d, --declaration                          Generates corresponding '.d.ts' file.
  --sourceMap                                Generates corresponding '.map' file.
  --outFile FILE                             Concatenate and emit output to single file.
  --outDir DIRECTORY                         Redirect output structure to the directory.
  --removeComments                           Do not emit comments to output.
  --noEmit                                   Do not emit outputs.
  --strict                                   Enable all strict type-checking options.
  --noImplicitAny                            Raise error on expressions and declarations with an implied 'any'
type.
  --strictNullChecks                         Enable strict null checks.
  --strictFunctionTypes                      Enable strict checking of function types.
  --strictPropertyInitialization             Enable strict checking of property initialization in classes.
  --noImplicitThis                           Raise error on 'this' expressions with an implied 'any' type.
  --alwaysStrict                             Parse in strict mode and emit "use strict" for each source file.
  --noUnusedLocals                           Report errors on unused locals.
  --noUnusedParameters                       Report errors on unused parameters.
  --noImplicitReturns                        Report error when not all code paths in function return a value.
  --noFallthroughCasesInSwitch               Report errors for fallthrough cases in switch statement.
  --types                                    Type declaration files to be included in compilation.
  --esModuleInterop                          Enables emit interoperability between CommonJS and ES
Modules via creation of namespace objects for all imports. Implies 'allowSyntheticDefaultImports'.
  @<file>                                    Insert command line options and files from a file.
```

CONCLUYENDO

Llegados a este punto ya estamos familiarizados con las bases del desarrollo en TypeScript. Por supuesto, es mucho aún lo que nos queda por aprender (todo, en realidad), pero ya sabemos por dónde empezar a trabajar. Los códigos de este artículo los tienes en este enlace. No te he incluido los archivos JSON, ni la ruta node_modules, para que, siguiendo las instrucciones del artículo te los crees tú mism@. En sucesivos artículos, como esto es algo rutinario, te los incluiré, pero no en éste.

Tipos de datos en TypeScript (I)

En el primer artículo de la serie de TypeScript decíamos que este es un lenguaje fuertemente tipado. Esto significa que los datos que definimos son de un tipo concreto, y no de otro, y no se puede andar "jugando" con los tipos de datos al estilo de cómo se hace, por ejemplo, en PHP o en JavaScript nativo.

Esta característica, que a los novatos de la programación les parece engorrosa es, desde luego, una de las mejores prestaciones de un lenguaje de programación, ya que nos ayuda a manejar nuestros datos de un modo estructurado y organizado correctamente, evitándonos una increíble cantidad de errores. De verdad que quien haya escrito cualquier programa sin estas restricciones sabe la cantidad de despropósitos que se pueden llegar a cometer por algo así, que parece, a ojos poco experimentados, una tontería.

Cómo compensación por la pequeña molestia que este tipado pueda suponer en principio, el abanico de tipos de datos de TypeScript es tan amplio, que siempre podremos recurrir a lo que necesite nuestro proyecto. Vamos a empezar en este artículo a conocer los tipos de datos de que disponemos en esta tecnología.

ACLARACIÓN INICIAL

Lo primero, antes de entrar en los tipos de datos que puede manejar TypeScript vamos a hacer una aclaración que a muchos os parecerá más que obvia, pero que, si la ignoramos, nos encontraremos infrautilizando TypeScript y con más problemas que soluciones. Se trata de lo siguiente. TypeScript es un superconjunto de JavaScript. Eso significa que cualquier código que puedas escribir en JavaScript puedes escribirlo, exactamente igual, en TypeScript, y se transpilará sin problemas. Por supuesto, que "puedas" hacerlo no significa que "debas" hacerlo así. Para eso no empleas TypeScript, sigues con tu JavaScript de toda la vida, y no te complicas más. Veamos de lo que te hablo. Imagina que en tu proyecto creas un código TypeScript como el siguiente:

```
var a = 3;
 var b = 2;
 let c = a + b;
 console.log (c);
```

Cómo ves, es JavaScript puro y duro (vale, la instrucción let es de JavaScript 2015 -o ES6, si lo prefieres-, pero es JavaScript, al fin y al cabo). Al transpilar te genera el JavaScript siguiente:

```
"use strict";
 var a = 3;
 var b = 2;
 let c = a + b;
 console.log(c);
```

Aparte de añadirte el "use strict";, el código es el mismo. No hemos adelantado nada por el hecho de estar usando TypeScript.

La sintaxis correcta para declarar pares nombre-valor en TypeScript es la siguiente;

let nombre_de_dato: tipo_de_dato = valor;

donde nombre_de_dato es el nombre que queremos darle al dato. A continuación usamos el signo dos puntos seguido de un espacio en blanco (:) y tipo_de_dato es el tipo de dato que vamos a usar (numérico, de cadena, booleano, etc). Después el operador de asignación y el valor inicial. Darle un valor inicial es opcional. Si no queremos darle un valor inicial, simplemente usaremos una sintaxis como la siguiente:

let nombre_de_dato: tipo_de_dato;

Evidentemente, cuando asignemos datos a una variable estos deberán ser del tipo del que se haya declarado dicha variable. Si una variable la declaramos como numérica y tratamos de asignarle, por ejemplo, una cadena literal, al intentar transpilar se producirá un error y el código JavaScript no será creado.

UN EJEMPLO SIMPLE

Antes de detallar los tipos básicos de datos en TypeScript te voy a mostrar un ejemplo de uso de la sintaxis que acabamos de comentar. Lo hemos llamado tipos_de_datos_1.ts:

```
/* Declaramos dos operandos de tipo numérico, asignándole al primero un valor
 durante la declaración y al segundo un valor después de la declaración. */
 let operando_1: number = 30;
 let operando_2: number;
 operando_2 = 2;
 /* Declaramos una variable para el resultado de sumar los dos operandos anteriores. */
 let resultado_suma: number = operando_1 + operando_2;
 /* Volcamos los datos a la consola del navegador. */
 console.log("El primer operando vale: ", operando_1);
 console.log("El segundo operando operando vale: ", operando_2);
 console.log("La suma de ambos vale: ", resultado_suma);
```

Cuando hacemos la transpilación obtenemos tipos_de_datos_1.js:

```
"use strict";
 /* Declaramos dos operandos de tipo numérico, asignándole al primero un valor
 durante la declaración y al segundo un valor después de la declaración. */
 let operando_1 = 30;
 let operando_2;
 operando_2 = 2;
 /* Declaramos una variable para el resultado de sumar los dos operandos anteriores. */
 let resultado_suma = operando_1 + operando_2;
 /* Volcamos los datos a la consola del navegador. */
 console.log("El primer operando vale: ", operando_1);
 console.log("El segundo operando operando vale: ", operando_2);
 console.log("La suma de ambos vale: ", resultado_suma);
```

Este archivo JavaScript lo cargamos en un HTML, como el siguiente index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Pruebas de TS</title>
</head>
<body>
    <script language='javascript' src='js/tipos_de_datos_1.js'></script>
</body>
</html>
```

Cuando lo cargas en tu navegador y abres la consola del mismo, ves que el resultado es correcto. Observa, en las líneas resaltadas del código JavaScript que no aparecen, en las declaraciones de las variables, los tipos de datos. Esto es porque el hecho de especificar el tipo de dato es una característica de TypeScript que JavaScript no implementa. Desde luego, esto carece de importancia. Si la transpilación se ha hecho sin problemas, significa que en TypeScript hiciste las declaraciones y las asignaciones correctamente.

ATENCIÓN. Cuando escribas tu código JavaScript en TypeScript y lo transpiles luego, debes acostumbrarte a no hacer "retoques" a mano en el JavaScript. Si tienes que cambiar o retocar cualquier cosa, por nimia que te parezca, hazlo en el TypeScript original y vuelve a transpilar, siguiendo las reglas de TypeScript. De otro modo, podrías estar infringiéndolas y encontrarte cometiendo los mismos errores una y otra vez. Cuando escribimos un código en TypeScript y, para poder usarlo, debemos transpilarlo a JavaScript, si hemos cometido errores de sintaxis el transpilador nos lanzará mensajes de error que nos ayudarán a escribir un código limpio y bien organizado.

ATENCIÓN. Para declarar variables en TypeScript podemos emplear la sentancia `var` o `let`. La diferencia entre una y otra está detallada en este artículo. Si tienes dudas, échale un vistazo.

TIPOS BÁSICOS DE DATOS

A continuación te incluyo una chuleta con los tipos básicos de datos que podemos manejar en TypeSrcipt. Guárdala a modo de referencia.

TIPO	EJEMPLO(S)	COMENTARIOS

number	let dato_1: number = 28; // Dato decimal
	let dato_2: number = 0b11001011; // Dato binario
	let dato_3: number = 0o452; // Dato octal
	let dato_4: number = 0XF42C; // Dato hexa
	let dato_5: number = 4.56; // Dato flotante
		Los datos de tipo number almacenan valores numéricos. Estos pueden ir expresados como números sin más, lo que le indica al transpilador que son decimales (pueden ser, como ves en los ejemplos, enteros o con punto flotante). Si queremos expresar un número en binario, lo precedemos con 0b. Si queremos expresarlo en octal, lo precedemos con 0o. Por último, si queremos expresar un valor hexadecimal lo precedemos con 0x.

string	let nombre: string = "Antonio"; let otro_nombre: string = 'Laura';	Los datos que van a almacenar cadenas literales se declaran como de tipo string. La cadena que le asignemos va acotada con comillas simples o dobles.

boolean	let dato_logico_1: boolean = true;
	
	let dato_logico_2: boolean = false;
		Los datos lógicos sólo pueden almacenar uno de dos estados: true o false. Por lo tanto, a estos datos hay que asignarles uno de estos valores en la misma declaración del dato. Si no, tiene un valor de tipo undefined que, por su propia naturaleza, no es de tipo boolean.[(1)]

any	let cualquier_cosa: any = "Lo que sea";	Los datos que se declaran como de tipo any pueden tener cualquier valor y, además, este puede variar durante la ejecución. Es decir. Podemos declararlo, como en el ejemplo de la izquierda, con una cadena y, posteriormente, a la misma variable, sustituirle el contenido por un número. Esto parece una aberración, ya que es lo mismo que no usar tipado de datos. TypeScript incorpora esto como mecanismo especial para casos muy concretos en los que un dato pueda proceder de un servicio externo, y el proveedor del mismo no pueda garantizar el tipado de datos. Salvo casos muy puntuales, no deberíamos usar este tipo en casi ninguna circunstancia.

[(1)] Los datos que se manejan en JavaScript son undefined cuando se declara una variable y no se le asigna un valor. Mientras no se asigne un valor a una variable declarada, esta tendrá el valor undefined. No es, realmente, un tipo de dato específico, si no, más bien, la ausencia de dato. También debes tener en cuenta que en programación (y TypeScript no es una excepción) se reconoce un tipo especial de dato llamado null. Es lo que obtenemos cuando una variable no es accesible (por haberse declarado, por ejemplo, en un ámbito interno, como un bloque de código específico) y tratar de acceder a ella desde otro ámbito. Los datos undefined y null a veces inducen a confusión si tratamos de recuperar un valor que no está preasignado, o no es accesible.

CONCLUYENDO

En este artículo hemos empezado a conocer los tipos de datos que se manejan en TypeScript. A partir del siguiente empezaremos a ver la potencia que realmente ofrecen estos tipos de datos, usándolos con las prestaciones propias de este lenguaje. Este artículo (y el siguiente) son un poco demasiado teóricos para mi gusto, pero necesarios para sentar bien unas bases que luego necesitaremos más adelante. En este enlace tienes los códigos de este artículo, para que experimentes con ellos.

Tipos de datos en TypeScript (II). Numéricos

En el artículo anterior hicimos una vista previa de los tipos de datos primitivos que maneja TypeScript. En realidad, todos los tipos de datos que se manejan en TS son extensiones de los tipos primitivos, tal como vamos a ir viendo.

Este artículo está dedicado a cómo podemos manejar datos numéricos en TS, lo que podemos y no podemos hacer con ellos y, lo que es más importante, como "queda todo" cuando transpilamos a JS.

Vamos a ver que ocurre cuando manejamos datos en distintas bases de numeración, y como podemos obtener los resultados esperados.

DECLARANDO Y MOSTRANDO VALORES

En el artículo anterior vimos que en TS (en realidad, también en JS) se pueden declarar valores numéricos en las principales bases de numeración que se emplean en programación (binario, octal, decimal y hexadecimal). Vamos a crear un pequeño script que nos declare unos datos numéricos en estas bases y nos los muestre en consola. Lo llamaremos tipos_de_datos_2_1.ts:

```
/* OPERACIONES CON DATOS NUMÉRICOS EN DISTINTAS BASES DE NUMERACIÓN */
  /* Declaramos las variables */
 let valor_1: number = 350; // Valor decimal
 let valor_2: number = 0b1100; // Valor 12 expresado en binario
 let valor_3: number = 0o101; // Valor 65 expresado en octal
 let valor_4: number = 0xCD; // Valor 205 expresado en hexadecimal
  /* Las mostramos en consola */
 console.log("La variable 'valor_1' vale ", valor_1);
 console.log("La variable 'valor_2' vale ", valor_2);
 console.log("La variable 'valor_3' vale ", valor_3);
 console.log("La variable 'valor_4' vale ", valor_4);
```

Cuando transpilamos a JS obtenemos algo muy parecido:

```
"use strict";
 /* OPERACIONES CON DATOS NUMÉRICOS EN DISTINTAS BASES DE NUMERACIÓN */
 /* Declaramos las variables */
 let valor_1 = 350; // Valor decimal
 let valor_2 = 0b1100; // Valor 12 expresado en binario
 let valor_3 = 0o101; // Valor 65 expresado en octal
 let valor_4 = 0xCD; // Valor 205 expresado en hexadecimal
 /* Las mostramos en consola */
 console.log("La variable 'valor_1' vale ", valor_1);
 console.log("La variable 'valor_2' vale ", valor_2);
 console.log("La variable 'valor_3' vale ", valor_3);
 console.log("La variable 'valor_4' vale ", valor_4);
```

Si ahora cargamos el JavaScript en un HTML, tal cómo hicimos en en artículo anterior, obtenemos, en la consola del navegador, lo siguiente (observa que yo tiro mucho de consola del navegador, porque es una forma muy cómoda de obtener resultados de pruebas en desarrollo):

```
La variable 'valor_1' vale  350                    tipos_de_datos_2_1.js:9

 La variable 'valor_2' vale  12                    tipos_de_datos_2_1.js:10

 La variable 'valor_3' vale  65                    tipos_de_datos_2_1.js:11

 La variable 'valor_4' vale  205                   tipos_de_datos_2_1.js:12
```

Y aquí viene la primera sorpresa. Los datos han sido declarados en distintas bases de numeración (y así están tanto en el original en TS como en el transpilado en JS), pero la consola nos muestra todos en decimal. Si pruebas a hacer operaciones con estos datos, los resultados de las mismas también aparecen en decimal. Tienes unos ejemplos en tipos_de_datos_2_2.ts:

```
/* OPERACIONES CON DATOS NUMÉRICOS EN DISTINTAS BASES DE NUMERACIÓN */

  /* Declaramos las variables */

let valor_5: number = 350; // Valor decimal
let valor_6: number = 0b1100; // Valor 12 expresado en binario
let valor_7: number = 0o101; // Valor 65 expresado en octal
let valor_8: number = 0xCD; // Valor 205 expresado en hexadecimal
```

```
let suma: number = valor_5 + valor_6; // La suma de dos datos en distintas bases.
let producto: number = valor_7 * valor_8; // El producto de valores en distintas bases
 /* Las mostramos en consola */
console.log("La variable 'valor_5' vale ", valor_5);
console.log("La variable 'valor_6' vale ", valor_6);
console.log("La variable 'valor_7' vale ", valor_7);
console.log("La variable 'valor_8' vale ", valor_8);
console.log("La suma de ", valor_5, " y ", valor_6, " da ", suma);
console.log("El producto de ", valor_7, " y ", valor_8, " da ", producto);
```

El resultado en la consola del navegador es:

```
La variable 'valor_5' vale  350                tipos_de_datos_2_2.js:11
 La variable 'valor_6' vale  12                tipos_de_datos_2_2.js:12
 La variable 'valor_7' vale  65                tipos_de_datos_2_2.js:13
 La variable 'valor_8' vale  205               tipos_de_datos_2_2.js:14
 La suma de  350  y  12  da  362               tipos_de_datos_2_2.js:15
 El producto de  65  y  205  da  13325         tipos_de_datos_2_2.js:16
```

Cómo puedes ver, todas las operaciones se han procesado en decimal. Es decir. Tanto JavaScript como TypeScript te permiten que asignes valores numéricos en distintas bases de numeración pero, una vez asignados, son almacenados y procesados en decimal.

LOS NOMBRES DE LAS VARIABLES

Observa que en el segundo código hemos cambiado los nombres de las variables. Esto se debe a lo siguiente: Tenemos dos scripts de TypeScript en el mismo proyecto. El transpilador "asume" que los JavaScript correspondientes pueden, en algún momento, ser invocados en el mismo contexto de ejecución (en el mismo documento HTML, para entendernos). Cómo las variables son globales en cada script (es decir, se declaran en el ámbito global del cuerpo del script) y no locales a un ámbito cerrado (una función u otro bloque de código), si le damos los mismos nombres en ambos scripts el transpilador "entiende" que las estamos declarando dos veces. Una variable puede declararse una sola vez en un contexto de ejecución, aunque pueda reasignarse su valor tantas veces cómo sea necesario. Por lo tanto, si las variables tuvieran los mismos nombres en ambos scripts, el transpilador dará errores por esta causa, y no efectuará la transpilación.

CONCLUYENDO

En este artículo hemos empezado a familiarizarnos con los datos numéricos en TS, y hemos aprendido un concepto importante respecto de los nombres de las variables. En el siguiente artículo empezaremos a trabajar con variables de tipo cadena. Los códigos de este artículo los tienes en este enlace para experimentar con ellos.

Tipos de datos en TypeScript (III). Cadenas y tipos personalizados

Las cadenas literales es un tipo de datos del que disponemos en cualquier lenguaje de programación. En TypeScript, sin embargo, nos ofrecen algunas peculiaridades que debemos conocer.

En este artículo vamos a ver cómo declarar y manejar cadenas de datos. No entraremos en los métodos propios de String de JavaScript, ya que, como ya sabemos, al ser TypeScript un superconjunto de JavaScript, todos los métodos y propiedades de este lenguaje están disponibles

también aquí, y funcionan exactamente igual. Sí veremos, sin embargo, construcciones que están disponibles en la versión ES6, pero que resulta interesante destacar.

Este artículo está orientado a conocer los datos de cadena vistos desde el prisma específico de TS, y cómo podemos sacarles partido en nuestro código. Algunas peculiaridades te sonarán familiares de otros lenguajes, aunque aquí tienen su sintaxis específica.

DECLARANDO VARIABLES DE CADENA Y PLANTILLAS

Declarar variables de cadena en TS es tan sencillo como asignarles el tipo string. Por ejemplo, así:

```
let nombre: string = "Antonio";
```

Además, podemos crear plantillas literales. Estas son cadenas que, en lugar de acotarse con comillas simples ('valor') o dobles ("valor") del modo tradicional, se acotan con acentos invertidos, también llamados graves (`valor`). A título meramente anecdótico, este tipo de acotado, poco habitual, se emplea en algunos contextos, como, por ejemplo, los nombres de campos en tablas MySQL.

Las plantillas permiten introducir variables de tipo string, o de otro tipo, acotándolas entre ${ y }, de forma que el valor de estas variables se integre en la plantilla, formando una cadena. Veamos un ejemplo en tipos_de_datos_3_1.ts:

```
/* Declaramos las variables */
 let nombre: string = "Antonio"; // Una cadena se declara con la palabra reservada string
 let salario: number = 1218.12; // Las variables number pueden ser, indiferentemente, enteros o
flotantes.
 let frase: string = `Me llamo ${nombre} y gano ${salario} € al mes.`; // Atento a la sintaxis
  /* Las mostramos en consola */
 console.log("La variable 'nombre' contiene: ", nombre);
 console.log("La variable 'salario' contiene: ", salario);
 console.log("La variable 'frase' contiene: ", frase);
```

Lo más llamativo es que, cuando transpilas, obtienes (casi) exactamente lo mismo:

```
"use strict";
 /* Declaramos las variables */
 let nombre = "Antonio"; // Una cadena se declara con la palabra reservada string
 let salario = 1218.12; // Las variables number pueden ser, indiferentemente, enteros o flotantes.
 let frase = `Me llamo ${nombre} y gano ${salario} € al mes.`; // Atento a la sintaxis
 /* Las mostramos en consola */
 console.log("La variable 'nombre' contiene: ", nombre);
 console.log("La variable 'salario' contiene: ", salario);
 console.log("La variable 'frase' contiene: ", frase);
```

Esto se debe a que las plantillas son una construcción que ya ha sido implementada en ES6 (JavaScript 2015). El resultado en la consola del navegador, cuando cargas el script desde un HTML, es el siguiente:

```
La variable 'nombre' contiene:  Antonio

 La variable 'salario' contiene:  1218.12

 La variable 'frase' contiene:  Me llamo Antonio y gano 1218.12 € al mes.
```

Las plantillas permiten escribir el contenido en varias líneas en el código, lo que es especialmente interesante cuando dicho contenido es especialmente largo. Si tuviéramos que escribirlo en una sólo línea, el código resultaría más complicado de leer y mantener. Observa tipos_de_datos_3_2.ts:

```
/* Declaramos unas variables convencionales */
 let otro_nombre: string = "Laura";
 let otro_salario: number = 1342.69;
  // Declaramos una plantilla multilinea
 let plantillaML: string = `<div> El nombre de esta persona es ${otro_nombre}</div>
 <div>Su salario es de ${otro_salario} euros/mes.</div>`;
  // Mostramos la plantilla en el documento.
 document.write(plantillaML);
```

Cuando transpilamos y cargamos el resultado en un navegador vemos como se muestra el HTML final en la página, así:

El nombre de esta persona es Laura

Su salario es de 1342.69 euros/mes.

TIPOS PERSONALIZADOS DE DATOS

Una característica de TypeScript es que se pueden definir tipos personalizados de datos. Esto se sale un poco del contexto de este artículo (vale, se sale mucho), pero me ha parecido interesante incluirlo aquí, porque nos permite jugar de una forma muy particular con los datos, y nos va a servir de introducción al siguiente artículo de esta serie. Además, aunque los tipos de datos personalizados pueden incluir cadenas o números, se suele emplear más con cadenas. Lo que hacemos es usar la palabra reservada type, seguida del nombre que queremos darle a nuestro tipo, y los valores que puede asumir, separados por pipes (|). Por ejemplo, imaginemos un tipo de datos que contenga una colección de nombres de colores, así:

```
type Colores = "Verde" | "Negro" | "Rojo" | "Azul";
```

Si ahora declaramos una variable del tipo Colores podrá tener cualquier valor de los que le hemos asignado a ese tipo, pero ningún otro. Mira el código tipos_de_datos_3_3.ts:

```
/* Declaramos un tipo específico de datos con una serie de opciones establecidas */

 type Colores = "Verde" | "Negro" | "Rojo" | "Azul";
  function mostrarColor(color:Colores){
    let colorElegido: string = `<div>El color es ${color}.</div>`;
    document.write(colorElegido);
 }
  mostrarColor("Rojo");
 mostrarColor("Verde");
```

Observa la función que declaramos en la línea resaltada. Como argumento recibe una variable, llamada color, del tipo Colores. Esta sintaxis, en la que el tipo de una variable se especifica en el parámetro de la declaración de la función es muy habitual en TS, y debes familiarizarte con ella. Cuando transpilamos, obtenemos tipos_de_datos_3_3.js:

```
"use strict";
 function mostrarColor(color) {
```

```
    let colorElegido = `<div>El color es ${color}.</div>`;
    document.write(colorElegido);
}
mostrarColor("Rojo");
mostrarColor("Verde");
```

Observa que la declaración del tipo de dato Colores no aparece por ninguna parte. Esto se debe a que los tipos de datos personalizados no es una construcción que acepte JavaScript. Si cargas este código JS en un documento HTML y lo llamas desde tu navegador verás que funciona correctamente. Sin embargo, imagina que ahora, en tu TypeScript intentas añadir la siguiente línea:

```
mostrarColor("Blanco");
```

Como Blanco no es un valor que esté incluido en el tipo Colores, al intentar transpilar se producirá un error y no se generará el código JavaScript.

Este concepto es muy interesante, porque nos permite vislumbrar una de las muchas formas en que TypeScript extiende JavaScript, y nos servirá de introducción a las colecciones de datos.

CONCLUYENDO

En este artículo hemos visto como podemos sacarle partido a las cadenas literales, y también hemos aprendido a crear y usar tipos personalizados de datos. Los códigos para que experimentes con ellos están en este enlace.

Tipos de datos en TypeScript (IV). Any y constantes

Vamos a dedicar esta entrada a unos conceptos muy simples respecto a los datos en TypeScript, pero que es necesario conocer. Ya sabemos que TypeScript es un lenguaje tipado (lo que, realmente, es un rasgo muy positivo). Sin embargo, en determinadas ocasiones, es necesario obviar ese tipado. Aquí veremos como hacerlo.

También sabemos como crear variables. Sin embargo, en algunos scripts es necesario contar con datos que no deben poderse modificar (ni siquiera de modo accidental) durante todo el ciclo de vida del script. Estos los creamos como constantes, lo que hace que si, una vez declarados, tratamos de modificar su valor, se produzca una excepción.

Vamos a ver ambas cuestiones.

LOS DATOS DE TIPO any

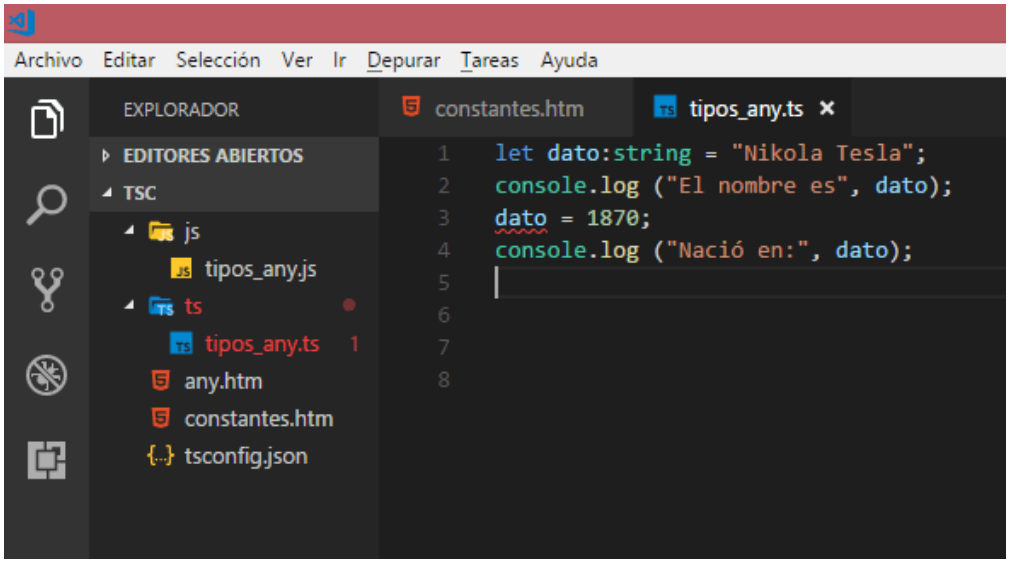
Ya sabemos que, cuando declaramos una variable en TypeScript, además de su valor (que podemos asignarlo en la declaración o no), establecemos su tipo, de modo que si, durante la programación, se le intenta asignar un valor de otro tipo, se produce un error de transpilación. Supongamos un script de TS como el siguiente:

```
let dato:string = "Nikola Tesla";
console.log ("El nombre es", dato);
dato = 1870;
console.log ("Nació en:", dato);
```

Como ves, es muy simple. Declaramos un dato de tipo string en la primera línea, y en la tercera tratamos de darle un valor numérico. Al ejecutar el transpilador en la terminal de comandos obtenemos un mensaje muy explícito:

ts/tipos_any.ts(3,1): error TS2322: Type '1870' is not assignable to type 'string'.

Además, si empleamos un editor con marcado de errores, como Visual Studio Code, también nos aparece una indicación, con el nombre del archivo en rojo, indicándonos el error, tal como vemos a continuación:



El problema es que este es un tipo de error con el que el transpilador es muy tolerante. Nos muestra el mensaje que hemos visto, avisándonos del error pero, aún así, efectúa la transpilación. El código JavaScript obtenido es el siguiente:

```
"use strict";
let dato = "Nikola Tesla";
console.log("El nombre es", dato);
dato = 1870;
console.log("Nació en:", dato);
```

ATENCIÓN. Esto se debe a que, aunque TypeScript es un lenguaje tipado, JavaScript no lo es. De hecho, como ves, en el código JS obtenido no aparece ninguna referencia a ningún tipo de dato, y el código final funciona en el navegador. Por esta razón, cuando transpilemos TypeScript es muy importante estar muy pendiente de los mensajes del transpilador.

Sin embargo, en determinadas ocasiones, es imprescindible "saltarse" el tipado. Para ello, lo que hacemos es declarar una (o más, claro) variable como de tipo any, así:

```
let dato:any = "Nikola Tesla";
console.log ("El nombre es", dato);
dato = 1870;
console.log ("Nació en:", dato);
```

Esto transpila perfectamente, y sin ningún mensaje de advertencia. Una variable de tipo any puede contener datos de cualquier tipo, en cualquier momento.

Y ahora viene la pregunta obvia. Si TypeScript le da tanta importancia al tipado de datos, ¿por qué han incluido un tipo de datos que vale para todo? Aquí tengo que decirte lo que me enseña la experiencia. En la mayoría de los casos (yo te diría que en el 99%, y quizá me quede corto), lo que vale para todo, al final no sirve para nada. Sin embargo, existe un pequeño porcentaje de situaciones en las que debemos usar tipo universal como este. Cuando hablemos del uso de TypeScript en ciertos escenarios con Angular, NodeJS u otros contextos veremos que esto es necesario. De acuerdo. Es extremadamente poco frecuente, y debemos acostumbrarnos a no usar este "tipo" de dato de modo sistemático, pero cuando hace falta, hace falta.

CONSTANTES

Si declaramos un dato usando la palabra reservada const, en lugar de let o var, el valor debe ser asignado en la declaración, y no puede cambiarse en ningún momento. Supongamos un código TypeScript como el siguiente:

```
const nombre:string = "Nikola Tesla";
 console.log ("El nombre es:", nombre);
  nombre = "John F. Kennedy";
 console.log ("El nombre es:", nombre);
```

En la primera línea declaramos una constante. En este caso, especificar el tipo de dato podría ser innecesario, ya que su valor no puede cambiar, una vez declarada (por eso debemos asignarle un valor en la misma declaración). Sin embargo, es bueno que lo pongamos, a efectos documentales, y también de buenas prácticas.

Sin embargo, más adelante, intentamos reasignar un valor a un dato que hemos declarado como constante. El transpilador nos lanza un mensaje de advertencia, como el que se muestra a continuación:

ts/constantes.ts(4,1): error TS2540: Cannot assign to 'nombre' because it is a constant or a read-only property.

A pesar de ello, una vez más, la transpilación se efectúa, obteniendo el siguiente código JavaScript:

```
"use strict";
 const nombre = "Nikola Tesla";
 console.log("El nombre es:", nombre);
 nombre = "John F. Kennedy";
 console.log("El nombre es:", nombre);
```

Afortunadamente, aquí las cosas no son tan fáciles. El navegador no "se lo traga", y en su consola nos muestra lo siguiente:

El nombre es: Nikola Tesla

Uncaught TypeError: Assignment to constant variable. at constantes.js:4

Como ves, esto produce un claro error en tiempo de ejecución. Entre esto, y el mensaje de advertencia del transpilador, ya deberíamos tener claro cual es la naturaleza de las constantes.

CONCLUYENDO

Con esto ya tenemos una visión de conjunto de los tipos simples de datos en TypeScript. En el próximo artículo empezaremos a ver tipos de datos complejos, como paso previo para el trabajo con objetos y clases, que es el punto fuerte de este lenguaje. Pero paciencia. Cada cosa a su tiempo. De momento, puedes descargar los scripts de prueba de este artículo en este enlace.

Datos compuestos en TypeScript (I). Matrices

Los arreglos son colecciones de datos que se almacenan bajo un mismo nombre, identificando cada uno de los datos de modo individual mediante un índice. Vamos, para entendernos, las matrices de JavaScript de toda la vida.

En TypeScript, sin embargo, contemplamos varios tipos de arreglos: las **matrices**, y las **tuplas**, las **listas** y los **objetos**. Son conceptualmente muy similares, aunque matices y operativas que los diferencian.

En este artículo nos familiarizaremos con las matrices que son, sin duda, los arreglos básicos en cualquier estructura.

LAS MATRICES

Las matrices en TypeScript son colecciones de datos, todos ellos del mismo tipo, que se declaran indicando el tipo junto al nombre (tal como con las variables), siguiéndolo de un par de corchetes, para indicar que se trata de una matriz. Por ejemplo, vamos a crear una matriz con nombres de países europeos. El código se llama matrices.ts, y es el siguiente:

let paises_europeos:string[] = [

Cuando efectuamos la transpilación, el JavaScript resultante es extremadamente parecido, como ves en matrices.js:

"use strict";

Si ahora cargamos este JavaScript en un documento HTML y vemos la consola del navegador, nos muestra correctamente la matriz creada:

Array(7)

A partir de aquí, podemos hacer con la matriz todo lo que podríamos hacer trabajando directamente con JavaScript (agregar elementos, eliminarlos, mostrar un elemento, etc). Mira el siguiente código:

let paises_europeos:string[] = [

Esto se transpila correctamente, y el resultado en la consola del navegador es el esperado:

Matriz inicial:

ATENCIÓN. Como hemos declarado la matriz como de tipo string, si le añadimos elementos, estos deberán de ser cadenas literales, es decir, datos del tipo que hemos declarado para la matriz. Si, por ejemplo, tratas de hacer lo siguiente:

```
paises_europeos.push(7);
```

el transpilador te lanzará un mensaje de aviso:

```
ts/matrices.ts(14,22): error TS2345: Argument of type '7' is not assignable to parameter of type 'string'.
```

Como ya vimos en el artículo anterior, a pesar de esto se efectúa la transpilación, por lo que deberás estar muy atento a los mensajes del transpilador para detectar incongruencias de este tipo.

ITERAR SOBRE MATRICES

En TypeScript tenemos dos modos de iterar sobre los elementos de una matriz. En realidad, no son iteraciones propias de TypeScript, sino de JavaScript. La primera es el construcción for...in, que ya estaba disponible en JavaScript 5. La segunda, la construcción for...of, es nativa de ES6. Aunque ambas son parecidas, y sirven para lo mismo, existen diferencias de uso. Veamos como funcionan:

```
let paises_europeos:string[] = [
```

Transpila el código, y cárgalo en tu navegador en un documento HTML. Observa los resultados en la consola del navegador. En la primera construcción iterativa (for...in) se usa una variable numérica (i) que recorre los índices de la matriz. A través de este índice se accede al valor de cada elemento. Observa que el índice no lo declaramos específicamente como de tipo number. Esto se debe a que, al ser, precisamente, un índice en una construcción de este tipo, TypeScript ya lo asume, implícitamente, como de este tipo. De hecho, si intentas especificarlo el transpilador te dará un error.

La construcción for...of es similar, sólo que accede, directamente, al valor de cada elemento, si emplear el índice. En este caso declaramos una variable (pais) que actúa como elemento de control de la estructura, recibiendo, en cada iteración, el valor del elemento en curso. Como la matriz es de datos de tipo string, la variable pais también lo es, de forma implícita, de modo similar a lo que ocurre con el índice en la construcción anterior.

CONCLUYENDO

En este artículo nos hemos introducido en el uso de matrices en TypeScript. En el siguiente conoceremos otros tipos de arreglos. Los códigos de este artículo los tienes en este enlace.

Datos compuestos en TypeScript (II). Listas y tuplas

Existen otras construcciones en TypeScript, que nos permiten agrupar datos, como con las matrices. Sin embargo, en estas construcciones hay una diferencia fundamental: los datos no tienen por qué ser todos del mismo tipo. Se trata de las **listas** y de las **tuplas**.

Estos arreglos nos permiten agrupar datos de distintos tipos. Esto convierte a cada uno de estos arreglos en un dato compuesto del tipo any, de una forma implícita, ya que un mismo arreglo puede tener datos numéricos y de cadena, por ejemplo.

Estos arreglos, además, sientan la base para la construcción de objetos, tema que trataremos más adelante en esta misma serie.

LAS LISTAS

Las listas son muy parecidas a las matrices, con la diferencia (aparte de la que acabamos de comentar), de que los índices pueden ser literales, es decir, no tienen que ser numéricos secuenciales. Observa el listado listas.ts:

```
/* Se declara la lista, con los índices y valores que deseemos. */
let cubo = {
    peso: 120,
    largo: 200,
    anchura: 100,
    altura: 57,
    color: "rojo"
}
 /* Se muestra la lista completa */
console.log("Los datos del cubo son:");
console.log(cubo);
 /* Se recorre cada lista, mostrando el índice y valor de cada elemento. */
for (let i in cubo){
    console.log ("El", i, "del cubo es", cubo[i]);
}
```

Cuando transpilas a JavaScript, se te muestra un mensaje de advertencia respecto a lo que te comentaba sobre el hecho de que la lisrta es, implícitamente, de tipo any. El mensaje es similar al siguiente:

ts/listas.ts(16,42): error TS7017: Element implicitly has an 'any' type because type '{ peso: number; longitud: number; anchura: number; altura: number; color: string; }' has no index signature.

Sin embargo, la tanspilación se lleva a cabo correctamente. Cuando cargas el JS resultante en un documento HTML, en la consola del navegador ves el resultado, así:

Los datos del cubo son:

{peso: 120, largo: 200, anchura: 100, altura: 57, color: "rojo"}

El peso del cubo es 120

El largo del cubo es 200

El anchura del cubo es 100

El altura del cubo es 57

El color del cubo es rojo

LAS TUPLAS

Las tuplas suponen una especie de híbrido entre las matrices y las listas. Se trata de un arreglo que puede tener datos de varios tipos, pero en los que las claves son, una vez más, índices numéricos.

Para construir una tupla, se define, en primer lugar, la estructura, indicando, específicamente, el tipo de dato que será cada elemento. Luego se le asignan valores a los elementos. Imagina que queremos una tupla con datos de un país. En concreto, vamos a tener cuatro datos: el primero será el nombre del país, de tipo string; el segundo será el nombre de su capital, también de tipo string; el tercer dato será el número de habitantes, un dato de tipo number; por último, el idioma oficial del país, que también es string. Después, asignamos los datos. Observa el listado tuplas.ts:

// Definimos la estructura de la tupla, indicando

// el tipo de cada uno de los datos que va a contener.

```
let pais :[
   string,
   string,
   number,
   string
];
```

// Asignamos los datos, que deben ser del tipo correspondiente.

```
pais = [
   "Argentina",
   "Buenos Aires",
   44664694,
   "Español"
];
 /* Se muestra la tupla completa */
console.log("Los datos del país son:");
console.log(pais);
```

Cuando transpilas, recibes una advertencia como la que ya hemos visto anteriormente, indicándote que la tupla, en conjunto, es implícitamente de tipo any. Por lo demás, se transpila sin problemas. Si luego cargas el JavaScript resultante en un HTML, en la consola del navegador verás lo siguiente:

Los datos del país son:

(4) ["Argentina", "Buenos Aires", 44664694, "Español"]

TIPOS DE VARIOS DATOS

Vamos a aprovechar la coyuntura para introducir aquí un concepto interesante sobre la declaración de los tipos de datos. Son los datos multitipo. Imagina el ejemplo anterior, en el que la población es un número bastante grande, cuya legibilidad no es, en modo alguno, cómoda. Si vas a hacer algún cálculo con este valor, tiene que estar así, en formato numérico. Sin embargo, si sólo lo quieres para mostrarlo, es probable que 44.664.694 sea más legible que 44664694. En ese caso, la población también tendría que ser un string. Sin embargo, como queremos reservarnos la posibilidad de meter

un número, por si necesitamos el valor, tenemos que decirle a TypeScript que ese dato en concreto puede ser de tipo string o number. Para ello, tenemos que redefinir la estructura de la tupla, así:

```
let pais :[
    string,
    string,
    number|string,
    string
 ];
```

Observa que para el tercer dato hemos indicado dos posibles tipos, separados por un pipe (|). Ahora podríamos cambiar la asignación de los datos de la tupla, así:

```
pais = [
    "Argentina",
    "Buenos Aires",
    "44.664.694",
    "Español"
 ];
```

Esto es perfectamente válido en TypeScript, y no solo para los datos de una tupla. A cualquier variable podemos asignarle múltiples tipos (incluso los tipos personalizados que vimos en este artículo). Mientras luego el dato responda a alguno de los tipos indicados, no habrá problemas.

CONCLUYENDO

En este artículo hemos conocido las listas y las tuplas. Los códigos para las pruebas te los puedes descargar en este enlace. En posteriores artículos hablaremos de conceptos que necesitamos para usar POO en TypeScript.

Datos compuestos en TypeScript (III). Objetos

Los objetos en TypeScript son estructuras muy similares a las listas, que establecen la base de la POO. Un objeto es una estructura de datos rígida, es decir, en la que sus datos son inamovibles en cuanto a estructura, aunque no en cuanto a contenido.

No te dejes impresionar por lo de "estructura rígida". Una vez más, tener la posibilidad de establecer los tipos de datos que se van a manejar, es algo positivo para la coherencia programática. Además, cada objeto en sí mismo mantiene siempre los mismos datos (aunque puedan cambiar sus valores), lo que los convierte en artefactos imprescindibles para mantener buenas prácticas de programación.

En este artículo vamos a ver como crear y manipular objetos desde TypeScript, como paso previo al uso de clases, tema que veremos en un artículo posterior.

CREANDO UN OBJETO

Crear un objeto pasa, realmente, por dos fases: definir la estructura del mismo, y poblarlo con datos. Dicho así, esto se parece cada vez más a declarar una clase e instanciarla en objetos, pero no lo es, aún. De momento, aún estamos trabajando sólo con objetos individuales. Vamos por partes. Empecemos declarando un objeto al que llamaremos miVehiculo, en el que pretendemos almacenar

ciertas propiedades de un vehículo. La estructura genérica (que podría variar, dependiendo de las propiedades que necesitemos), podría ser algo así:

```
let miVehiculo: {
    ruedas:number,
    motor:number,
    color:string,
    deposito:number,
    autonomia:number,
    automatico:boolean
 }
  console.log(miVehiculo);
```

Esto transpila a JavaScript de una forma bastante decepcionante:

```
"use strict";
 let miVehiculo;
 console.log(miVehiculo);
```

Cómo ves, el resultado es muy pobre, ya que la estructura de datos y sus tipos son características que JavaScript no implementa. Cuando muestras en consola el valor de la variable, tal como está actualmente, lo único que obtienes es undefined.

Para que el objeto tenga datos con valores con los que podamos trabajar, debemos asignárselos. Podemos hacer la asignación en la misma sentencia en la que hemos hecho la declaración de la estructura, así:

```
let miVehiculo: {
    ruedas:number,
    motor:number,
    color:string,
    deposito:number,
    autonomia:number,
    automatico:boolean
 } = {
    ruedas: 4,
    motor: 1600,
    deposito: 100,
    autonomia: 1500,
    automatico: true,
    color: "verde"
 }
  console.log(miVehiculo);
```

Esto transpila a un JavaScript que ya tiene algo de "sustancia":

```
"use strict";
 let miVehiculo = {
    ruedas: 4,
    motor: 1600,
    deposito: 100,
    autonomia: 1500,
    automatico: true,
    color: "verde"
```

```
};
console.log(miVehiculo);
```

En la consola del navegador, esta vez, sí obtenemos un resultado relevante:

```
automatico:true
 autonomia:1500
 color:"verde"
 deposito:100
 motor:1600
 ruedas:4
 __proto__:Object
```

Observa especialmente la última línea, que nos dice que los datos anteriores son parte de un objeto.

También podemos hacer la asignación de datos en una sentencia aparte de la declaración, así:

```
let miVehiculo: {
    ruedas:number,
    motor:number,
    color:string,
    deposito:number,
    autonomia:number,
    automatico:boolean
}
miVehiculo = {
    ruedas: 4,
    motor: 1600,
    deposito: 100,
    autonomia: 1500,
    automatico: true,
    color: "verde"
}
 console.log(miVehiculo);
```

Al transpilar veremos que la declaración y la asignación de valores también quedan separadas, y funciona perfectamente, mostrando en la consola del navegador los valores.

MANEJANDO VALORES

Una vez creado un objeto, podemos manejar los valores de sus datos de la forma que nos convenga, ateniéndonos siempre a unas premisas fundamentales:

- Sólo podemos trabajar con los valores que existen en la declaración del objeto. Esto significa que no podemos sacarnos de la manga propiedades que no hayan sido incluidas en la declaración. En el caso del vehículo que hemos visto, por ejemplo, no podemos crear una propiedad que sea, digamos, el número de asientos, ya que no consta en la declaración de la estructura.
- Los datos que sí constan, podemos cambiarles el valor, siempre que sean del tipo especificado en la declaración. Así, en el caso anterior, podremos establecer, en la propiedad motor, el valor 1800, pero no "1800 cc", por ejemplo.
- TypeScript permite que se asignen las propiedades del objeto, o que se reasignen posteriormente, pero todas en conjunto. En seguida comentamos esto.

Si bien estas son limitaciones que JavaScript no impone, TypeScript sí lo hace, y el transpilador nos avisará de los errores en el momento de transpilar.

Si queremos reasignar (modificar) el valor de alguna de las propiedades (o todas) del objeto, no debemos reasignar todo el objeto en su conjunto. sólo con la propiedad que queremos modificar. Observa el siguiente fragmento:

```
let miVehiculo: {
    ruedas:number,
    motor:number,
    color:string,
    deposito:number,
    autonomia:number,
    automatico:boolean
 } = {
    ruedas: 4,
    motor: 1600,
    deposito: 100,
    autonomia: 1500,
    automatico: true,
    color: "verde"
 }
  console.log(miVehiculo);
  /* No podemos asignar todo el objeto si no nos atenemos a la estructura
 declarada. Esto da un error en TypeScript. Después de todo, para eso
 se declara una estructura. */
 miVehiculo = {color: "rojo"};
 console.log(miVehiculo);
```

Si lo que queremos es modificar sólo el valor de una propiedad lo haremos empleando la notación clásica objeto.propiedad, así:

```
let miVehiculo: {
    ruedas:number,
    motor:number,
    color:string,
    deposito:number,
    autonomia:number,
    automatico:boolean
 } = {
    ruedas: 4,
    motor: 1600,
    deposito: 100,
    autonomia: 1500,
    automatico: true,
    color: "verde"
 }
  console.log(miVehiculo);
  miVehiculo.color = "rojo";
```

Esto sí es perfectamente legítimo en TypeScript, y da una transpilación limpia (sin avisos de errores).

CONCLUYENDO

En este artículo hemos aprendido a crear y manipular objetos en TypeScript. En realidad, más que objetos como tal son simples estructuras de datos, ya que pensamos en el término objeto como una instancia de una clase, y ese es un tema que aún no hemos tratado, y del que empezaremos a hablar en una artículo próximo. Lo que hemos visto aquí está bien como introducción, pero necesitamos "más" y, como veremos, TypeScript nos lo ofrece. De momento, los códigos de este artículo te los puedes descargar en este enlace.

Datos compuestos en TypeScript (IV). Más sobre objetos

En el artículo anterior aprendimos a manejar objetos de una forma eficaz, pero bastante simple. Los objetos son, como ya sabemos, estructuras de datos pero, combinando eso con lo que ya sabemos, y con un conocimiento cierto de ES6, podemos sacarle mucho más partido a los objetos en nuestra programación.

En este artículo vamos a crear un tipo personalizado de objeto. Después, crearemos una matriz de objetos de ese tipo. Por último, veremos como podemos iterar sobre esa matriz, empleando una construcción que, hasta ahora no hemos utilizado: la instucción forEach. No es específica de TypeScript, sino que fue implementada, como ya sabes, en ES6.

EL ESCENARIO

Vamos a usar un tipo personalizado de datos, para almacenar actores y actrices famos@s. Para cada uno almacenaremos su nombre artístico, su fecha de nacimiento, y una lista de cinco de sus películas más emblemáticas (le pondremos cinco a cada uno por poner algo, pero podríamos haberle puesto un arreglo con distintas cantidades de elementos).

Lo primero que hacemos es definir el tipo de dato que vamos a crear, así:

```
/* Creamos un tipo de datos personalizado, que es una lista. */

 type actor = {nombre:string, nacimiento:string, peliculas:string[]};
```

Esto no es nada nuevo. Ya aprendimos en un artículo anterior la forma de crear tipos de datos personalizados mediante la instrucción type.

A continuación creamos una matriz de objetos del tipo que hemos definido. Observarás que cada objeto mantiene, como debe de ser, la estructura del tipo actor (para eso la hemos definido como un tipo personalizado):

```
/* Creamos una matriz llamada actores, en la que cada objeto es una estructura del tipo actor */
 let actores: actor[] = [
  {
    nombre: "John Travolta",
    nacimiento: "18-02-1954",
    peliculas: [
      "Fiebre del sábado noche",
      "Grease",
      "Pulp Fiction",
      "El castigador",
      "Asalto al tren Pelham 123"
    ]
  },
  {
```

```
    nombre: "Natalie Portman",
    nacimiento: "09-06-1981",
    peliculas: [
       "Thor: Un Mundo Oscuro",
       "Una loca aventura medieval",
       "Star Wars: Episodio III - La venganza de los Sith",
       "V de Vendetta",
       "Star Wars: Episodio II - El ataque de los clones"]
  },
  {
    nombre: "Jennifer Garner",
    nacimiento: "17-04-1972",
    peliculas: [
       "Daredevil",
       "La extraña vida de Timothy Green",
       "Elektra",
       "Valentine's Day",
       "Nueve vidas"
    ]
  },
  {
    nombre: "Sean Connery",
    nacimiento: "25-08-1930",
    peliculas: [
       "La caza del Octubre Rojo",
       "Los inmortales",
       "Desde Rusia con amor",
       "Sólo se vive dos veces",
       "Atmósfera cero"
    ]
  },
  {
    nombre: "Milla Jovovich",
    nacimiento: "17-15-1975",
    peliculas: [
       "Saga Resident Evil",
       "El quinto elemento",
       "Zoolander",
       "Los tres mosqueteros",
       "Dirty Girl"
    ]
  },
];
```

A continuación creamos una cadena que contiene una serie de instrucciones HTML, para crear una tabla, con sus cabeceras incluidas, y dejamos la tabla "abierta", para luego crear una fila por cada actor o actriz de la lista:

```
/* Creamos una tabla HTML y la dejamos "abierta" */
 let tablaDeActores:string = `
    <table width="750" border="1">
       <tr>
          <th width="250">Actor / Actriz</th>
          <th width="120">F.Nac</th>
          <th width="*">Películas</th>
       </tr>
```

```
`;
```

Ahora viene la parte que es de ES6. La estructura forEach recorre una matriz y recibe, como argumento, una función de callback que, a su vez, recibe cada uno de los elementos de la matriz. Como cada elemento es, a su vez, una estructura, podemos acceder a sus propiedades, incluyéndolas en una tabla HTML.

La gracia es cuando llegamos al elemento pelicula. Este es, a su vez, una matriz de strings. Por lo tanto, creamos otra construcción forEach anidada, lo que nos permite recorrer cada una de las películas del actor en curso, como vemos a continuación:

```
/* Recorremos cada objeto con dos forEach anidados, para poder recorrer los actores y, dentro de
estos, sus películas. */
actores.forEach(function(actor){
   tablaDeActores += `
      <tr>
         <td valign="top">${actor.nombre}</td>
         <td valign="top">${actor.nacimiento}</td>
         <td valign="top"><ul>
   `;
   actor.peliculas.forEach(function(pelicula){
      tablaDeActores += `<li>${pelicula}</li>`;
   });
   tablaDeActores += `</ul></td></tr>`;
});
```

Para finalizar, simplemente cerramos la tabla y la mostramos en el documento.

```
/* Cerramos la tabla y la mostramos en el documento. */
 tablaDeActores += `</table>`;
 document.write(tablaDeActores);
```

Si transpilas el código y lo ejecutas en tu navegador, verás la tabla de datos completa, tal cómo queda.

CONCLUYENDO

En este artículo hemos visto un uso más avanzado de los objetos en TypeScript. Los códigos de ejemplo completos puedes descargarlos en este enlace. En el próximo artículo empezaremos a ver clases y POO empleando

Datos compuestos en TypeScript (V). POO

La programación orientada a objetos se basa, como ya sabes, en una estructura de clases y objetos creados a partir de esas clases, así como en unas adecuadas prácticas de programación. En los artículos anteriores hemos aprendido a crear objetos en TypeScript como estructuras de datos articuladas a partir de tipos personalizados de datos. Como el tipo de un dato no trasciende a JavaScript (al menos todavía, con las versiones más actuales), en definitiva nos quedaban como conjuntos de datos agrupados, sin más.

JavaScript, a partir de la versión ES6, ya soporta una rudimentaria programación de clases y objetos. En este artículo vamos a ver que nos aporta al respecto TypeScript, y como debemos trabajar con la POO.

LA POO EN TypeScript

Vamos a centrarnos en este apartado en crear un ejemplo muy simple de una clase, e instanciar un objeto a partir de dicha clase, Sólo para ver como transpila a JS y la mecánica básica del uso de clases. El código, basado en el ejemplo del artículo anterior, aunque con menos datos, para simplificar, es actores.ts:

```
class Actor
 {
    /* Declaramos las propiedades */
    nombre: string;
    f_nacimiento: string;
    private static datos:string; // propiedad de clase y privada.
     constructor (nombre:string, f_nacimiento?:string)
    {
       this.nombre = nombre;
       if (f_nacimiento !== undefined)
       {
          this.f_nacimiento = f_nacimiento;
       } else {
          this.f_nacimiento = "Sin determinar";
       }
    }
     mostrar()
    {
       Actor.datos = `
          <div>
             El actor <strong>${this.nombre}</strong> nació el <cite>${this.f_nacimiento}</cite>.
          </div>
       `;
    }
 }
  let actor = new Actor("John Travolta", "18-02-1954");
 actor.mostrar();
```

Como ves, la programación es extremadamente sencilla. De hecho, si estás familiarizado con todo lo relativo a POO (herencia, interfaces, instanciación, sobreescritura de métodos, encapsulación, etc.) que sepas que TODO lo que sabes al respecto te vale perfectamente en TypeScript. De hecho, a título de curiosidad, y sólo como muestra, en la línea 6 ves que hemos creado, por ejemplo, una variable de clase encapsulada (privada). Si no estás familiarizado con la POO, te sugiero que leas estos artículos.

Sólo hay algunos matices en los que debes reparar. En primer lugar, observa la línea 8. El constructor de una clase se declara con el nombre genérico constructor, y debe tener el modificador de acceso public (a la fuerza, claro). O no debe tener modificador de acceso, como en este ejemplo, ya que public es el acceso por defecto si no se especifica nada.

También quiero llamar tu atención respecto a la línea 21. Cuando nos referimos a una propiedad de clase, en lenguajes como PHP encadenamos el nombre de la clase y de la propiedad mediante el signo :: (que, en el mundillo de la programación tiene el pintoresco e impronunciable nombre de *Paamayim Nekudotayim* o, de una forma más familiar, operador de resolución de ámbito). Cuando usamos variables de objeto en TypeScript (y esto viene heredado de lenguajes totalmente orientados a objetos, como Java) se emplea el punto (.).

DEFINIR PROPIEDADES EN EL CONSTRUCTOR

Cuando creamos una clase estamos acostumbrados a declarar las propiedades que tendrán los objetos de dicha clase (con su ámbito de visibilidad público o privado, según el caso), y luego, si es necesario inicializarlas al crear el objeto, se le asignan valores en el constructor, así:

```
class Members {
  public nombre: string;
  public anyo_nac: number;
   constructor (NombreRecibido: string, AnyoRecibido: number) {
    this.nombre = NombreRecibido;
    this.anyo_nac = AnyoRecibido;
  }
}
```

Esto permite que las variables nombre y anyo_nac estén disponibles en toda la clase. Es decir. Si, posteriormente tratamos de mostrarlas, o creamos algún método que las use de alguna manera, tendremos estas variables disponibles con los valores que se les han asignado en el constructor.

Existe otra manera de manejar las variables en el constructor, así:

```
class Members {
  constructor (nombre: string, anyo_nac: number) { }
}
```

Esto crea el objeto con los valores recibidos asignándolos, directamente, a las variables nombre y anyo_nac. Sin embargo, esto tiene un problema (en realidad, un problemón). Si examinas el código, te das cuenta en seguida. Las variables del objeto nombre y anyo_nac se crean en el constructor, y sólo viven en el ámbito del mismo. Por lo tanto, todo lo que quieras hacer con ellas hay que hacerlo en el constructor, en el momento de la creación del objeto. Una vez finalizado el ciclo de vida del constructor, las variables no están disponibles para usarlas en otros métodos, o hacer lo que sea con ellas.

La solución pasa por usar la misma sintaxis, pero precediendo las variables con un modificador de acceso. Por ejemplo, lo siguiente:

```
class Members {
  constructor (public nombre: string, public anyo_nac: number) { }
}
```

El modificador de acceso puede ser el que desees, pero debe estar. Incluso si las variables van a ser de visibilidad pública, que es el modificador de acceso por defecto y, por lo tanto, parece que no debería ser necesario, si usas esta sintaxis debes indicarlo específicamente. Declarando así las variables en el constructor, están disponibles en todo el resto de la clase, para cualesquiera otros métodos que hubiera.

Esta sintaxis es muy habitual, por lo que debes conocerla. Sobre todo, en entornos de desarrollo como Angular.

TRANSPILANDO

La transpilación de un TypeScript a JavaScript puede ser diferente, según tengamos nuestro transpilador configurado para obtener JavaScript 5 o ES6. Esto es algo de lo que ya hablamos en este artículo. Salvo que tu código sea ridículamente simple, notarás diferencias en el resultado final. Sin embargo, estás son especialmente relevantes cuando empleas clases y objetos. Observa como transpila el código anterior a ES6:

```
"use strict";
 class Actor {
    constructor(nombre, f_nacimiento) {
       this.nombre = nombre;
       if (f_nacimiento !== undefined) {
          this.f_nacimiento = f_nacimiento;
       }
       else {
          this.f_nacimiento = "Sin determinar";
       }
    }
    mostrar() {
       Actor.datos = `
          <div>
             El actor <strong>${this.nombre}</strong> nació el <cite>${this.f_nacimiento}</cite>.
          </div>
       `;
    }
 }
 let actor = new Actor("John Travolta", "18-02-1954");
 actor.mostrar();
```

Como ves, dejando aparte el tema del tipado de datos, el resultado es claro, legible y muy similar al original en TS. Ahora, sin embargo, retrocedamos un poco en el tiempo (hasta JavaScript 5), y veamos lo que ocurre:

```
"use strict";
 var Actor = /** @class */ (function () {
    function Actor(nombre, f_nacimiento) {
       this.nombre = nombre;
       if (f_nacimiento !== undefined) {
          this.f_nacimiento = f_nacimiento;
       }
       else {
          this.f_nacimiento = "Sin determinar";
       }
    }
    Actor.prototype.mostrar = function () {
       Actor.datos = "n          <div>n             El actor <strong>" + this.nombre + "</strong> naciu00F3 el
<cite>" + this.f_nacimiento + "</cite>.n          </div>n       ";
    };
    return Actor;
 }());
 var actor = new Actor("John Travolta", "18-02-1954");
 actor.mostrar();
```

A ver. Funcionar, lo que se dice funcionar, sí, vale. Funciona. Pero nadie puede negar que es tosco y que, desde luego, es cualquier cosa menos programación orientada a objetos. Es ilegible, inmantenible... y eso sólo porque el código original era extremadamente simple. Imagina ahora un

código con clases con veinte propiedades y diez métodos, que use herencia y alguna interface, por ejemplo. Te he mostrado esto para que entiendas la importancia de transpilar a JavaScript 2015 siempre que te sea posible. Sí, de acuerdo Si tienes que hacer mantenimiento sobre el código, siempre trabajarás sobre el TypeScript original, nunca sobre la transpilación, lo que, evidentemente, hace que, como desarrollador, esto no te afecta. Sin embargo, si alguien, en algún momento, tiene que tocar el JavaScript... bueno, que no le pase nada.

CONCLUYENDO

En este artículo nos hemos introducido en la POO en TypeScript. Como ya te he comentado, todo lo que ya sabes al respecto te vale en TypeScript, aplicando, eso sí, las construcciones y sintaxis propias de este lenguaje, que hemos aprendido en los artículos anteriores de esta serie. Aún hay algunas cosas que nos quedan por aprender respecto a TypeScript. Sin embargo, son conceptos que entenderemos mejor en su contexto, por lo que hablaremos de ellos en artículos en los que empleemos TS como herramienta, en lugar de centrarnos en el desarrollo en TS en sí mismo. Los códigos de este artículo los tienes, como siempre, en este enlace.

Funciones en TypeScript

Las funciones en TypeScript guardan una sintaxis muy similar a la empleada en ES6 y retrocompatible con JS5. Si bien el código final dependerá de la versión de JavaScript que tengamos determinada en nuestro archivo tsconfig.json (lo que, a su vez, viene determinado por el público target al que nos dirigimos), en ambos casos la transpilación generará las funciones de forma que sean totalmente compatibles.

Con carácter general, TypeScript permite definir tres tipos de funciones:

- Funciones con nombre.
- Funciones anónimas.
- Funciones lambda.

Hay quien habla de un cuarto tipo, que son los métodos. Esta no es más que una cuestión semántica para referirnos a funciones dentro del contexto de clases. En todo caso, los métodos pueden ser de cualquiera de los tipos mencionados anteriormente. De los métodos hablaremos en los artículos que dediquemos a clases y objetos. En este, conoceremos los tres tipos de funciones que he mencionado, y que no son si no distintas sintaxis para escribir funciones (entiendo que si conoces JavaScript, el concepto de "función" lo tienes más que asumido). Además, al final de este artículo conoceremos un par de peculiaridades de los parámetros de funciones en TypeScript.

FUNCIONES CON NOMBRE

Aquí estamos ante la declaración típica de un función "de las de toda la vida". Simplemente, se declara como se ha hecho siempre en JavaScript, con la palabra reservada function, seguida del nombre de la función. La única diferencia es que, como ya sabemos, TypeScript nos permite establecer el tipo de datos, y es algo que deberemos hacer a la hora de establecer los parámetros que recibe la función y el retorno, si es el caso. Este tipo de funciones no tienen nada de particular. Tienes un código de ejemplo en funciones_con_nombre_1.ts, en el paquete de códigos de este artículo, por si quieres echarle un vistazo. No lo reproduzco aquí porque ocupa su sitio y, como hemos comentado, no aporta nada nuevo.

También podemos emplear una sintaxis alternativa, que lo que hace es crear un elemento y asignarle, como contenido, una función. Tienes un ejemplo en el código funciones_con_nombre_2.ts.

La función se sigue invocando por su nombre (el del elemento que hemos creado al efecto) y su operativa es la misma que en caso anterior.

Escoger una manera de declarar las funciones u otra es una cuestión de criterio personal.

FUNCIONES ANÓNIMAS

Estas son las funciones que se asignan, directamente, como respuesta a un evento. No tienen un nombre, por lo que no pueden ser invocadas posteriormente desde ninguna parte del documento. Solo se disparan si se detecta el evento. Tienes un ejemplo que se dispara a la carga de la página, en funciones_anonimas.ts. Es una forma muy cómoda de declarar funciones para determinados eventos pero, como sabes, los posibles eventos están bastante limitados. Hay determinados elementos sobre los que no se pueden registrar eventos en JavaScript. Por ejemplo, las colecciones de elementos (algo así como document.getElementsByTagName("input")) no detentan eventos de forma individualizada, por lo que no se puede emplear este tipo de declaraciones de funciones sobre dichas colecciones (a menos, claro, que recurras a herramientas como jQuery).

Normalmente, estas funciones se emplean para eventos globales como la carga de la página en la ventana, o el reescalado de esta.

FUNCIONES LAMBDA

También llamadas coloquialmente con el nombre de "funciones de flecha gorda" son, realmente, las mismas funciones, con una notación diferente, sin emplear la palabra reservada function. Esto puede inducir a confusión, cuando ves el código, por lo que las mencionamos aquí para que sepamos "de que va". Personalmente, no soy partidario de esta notación, pero te la he incluido en un ejemplo en los listados, para que la veas y juegues con ella. El listado se llama funciones_lambda.ts.

ERRORES Y HORRORES

Durante la transpilación de estos códigos te vas a encontrar con tantos mensajes de error del transpilador que, seguramente, pienses que "algo está mal". No te asustes. Ya nos hemos enfrentado anteriormente a estas situaciones. Aunque el transpilador te habla de un error, en realidad es sólo una advertencia que, si tu código está correcto, no te va a dar ningún problema. En concreto, los supuestos "errores" que vas a encontrar aquí son de dos tipos:

```
ts/funciones_con_nombre_1.ts(4,5): error TS2531: Object is possibly 'null'.
```

Te avisa de que un objeto determinado es "probablemente" de tipo null. Realmente, lo que hace esto es llamar tu atención para que, si te refieres a un objeto, estés seguro de que el mismo existe en el documento. Si, por ejemplo, te has equivocado al asignarle el valor al atributo id, tu JavaScript transpilado lanzará un error en tiempo de ejecución, por no encontrar el objeto. Si tus identificadores están correctamente asignados, no habrá ningún problema.

El otro aviso que vas a encontrar es como el siguiente:

```
ts/funciones_con_nombre_1.ts(4,43): error TS2339: Property 'value' does not exist on type
'HTMLElement'.
```

Puedes ignorarlo directamente. Este aviso se lanza porque, a priori, existen elementos HTML que no tienen la propiedad value. Sin embargo, cómo en estos ejemplos la estamos usando sobre elementos que si la tienen, esto no te dará tampoco ningún problema.

Y sí. Como vamos viendo, el transpilador es extremadamente conservador en cuanto al número de avisos que da pero, como ves, la mayoría no son errores.

PARÁMETROS POR DEFECTO, OPCIONALES Y MÚLTIPLES

A los parámetros de una función se les pueden asignar valores por defecto, de forma que si la función es invocada sin recibir un parámetro concreto, se le asigne el valor por defecto especificado en la declaración de la misma. Tienes un ejemplo en parametros_por_defecto.ts.

Los parámetros pueden ser opcionales, es decir, que se le pasen a la función en el momento de llamarla, o que no se le pasen. Esto, que muchos presentan como una novedad de TypeScript es, en realidad, el comportamiento de JavaScript de toda la vida. Si no pasas un parámetro, dentro de la función este se asumirá con el valor undefined. Lo que se hace en TypeScript es marcar, en la declaración, los parámetros opcionales con el signo ?, así:

```
function opcional(parametro?:string)
```

Esto no modifica el comportamiento de la función. Aún tienes que seguir comprobando en el interior si el valor del parámetro es undefined. De hecho, al transpilar, verás que no hay diferencia en el JavaScript si pones el signo ? o no lo pones. Entonces, ¿por qué ponerlo? Es una cuestión semántica, un convencionalismo adoptado para indicar que ese parámetro es opcional, es decir, que puede haber llamadas a la función que lo incluyan, y otras que no.

Por último, cuando se declara una función, puede que alguno de los parámetros que reciba sea una matriz, en lugar de un valor simple. Tienes un ejemplo en parametros_multiples.ts. En ese caso tenemos dos posibles formas de declarar el parámetro que es una matriz. La primera es la siguiente:

```
function multiples(color:string, ...banderas:string[])
```

Los tres puntos tienen una finalidad semántica, para indicar que el argumento debe ser una matriz. Declarándolo así, si queremos iterar la matriz deberemos referirnos a su primer elemento (el [0]), que es donde queda encapsulado el contenido real de la matriz. Lo haríamos así:

```
for (let bandera of banderas[0])
```

La alternativa es declarar la función sin preceder la matriz de los tres puntos, así:

```
function multiples(color:string, banderas:string[])
```

En ese caso, el acceso a la matriz es directo, así:

```
for (let bandera of banderas)
```

Usar una notación u otra es cuestión de elección personal, pero las notaciones semánticas ayudan mucho cuando se trata de trabajar con códigos grandes, con muchas líneas y variables, de forma que nos facilite saber lo qué es cada cosa.

CONCLUYENDO

Si conoces ES6 (JavaScript 2015) verás que en este artículo no hemos visto nada nuevo. Salvo el tipado de datos, todo lo que hay aquí es solo para que veas que las funciones se siguen empleando del mismo modo. La única razón para emplear TypeScript, en lo que a funciones se refiere, es que lo uses en todo tu proyecto, principalmente por el tipado de datos. Los códigos de ejemplos de este artículo, que no he incluido en el texto por no hacerlo demasiado extenso, están en este enlace.