TypeScript

TypeScript

- Como hemos visto, ES6 es la versión actual de Javascript.
- TypeScript es un subconjunto de ES6, lo que significa que todas las características de ES6 se encuentran implementadas en TypeScript, pero no todas las características de TypeScript las encontramos en ES6.
- Por eso, para poder trabajar con TypeScript, debemos transformar nuestro código a ES5, para que pueda ser ejecutado por la mayoría de los navegadores.

TypeScript

- Una de las características más importantes en el desarrollo con TypeScript es la inclusión del tipo *information*.

- Este tipo permite la creación de un código mucho más predecible y de fácil lectura.

- Por ejemplo, la declaración de funciones es mucho más explícita.

```
function add(a: number, b: number) {
  return a + b;
}

add(1, 3); // 4
```

TypeScript - TSC

- Para poder compilar nuestras aplicaciones creadas en TypeScript, necesitamos usar el comando **tsc**.

- Podemos instalarlo para su uso local a través de NPM

npm install -g typescript

- Si la instalación se completa correctamente, vamos a poder compilar nuestros ficheros TypeScript de la siguiente manera:

tsc nombreFichero.ts

TypeScript - TSC

- Este proceso de compilación, no devuelve ningún tipo de mensaje de confirmación si todo ha ido correctamente.

- Si encuentra cualquier tipo de error en nuestro código, obtendremos los errores asociados.

- Una vez se termine correctamente el proceso de compilación, obtendremos un fichero con extensión .js que representa la transformación del código escrito en TypeScript a un fichero de tipo ES5 legible por la mayoría de navegadores.

TypeScript - TSC

- En un proyecto típico, el número de ficheros a compilar es más de uno. Tsc nos permite manejar más de un fichero a la vez, pasándoselos como argumentos en la línea de comandos.

- Para que no existan problemas en su compilación, debemos especificar qué esquema vamos a seguir. El más común es **commonjs**.

- Por lo tanto, si tenemos dos ficheros a.ts y b.ts, podríamos proceder a su compilación de la siguiente manera:

tsc -m commonjs a.ts b.ts

TypeScript - TSC

- A la hora de compilar, podemos indicarle muchos modificadores al comando tsc para que realice las acciones que necesitamos. Esto puede ser una tarea bastante tediosa.

- Por suerte, contamos con una manera de simplificar todo este trabajo.

- Con la definición del fichero **tsconfig.json** podemos especificar las condiciones que vamos a pasarle al compilador.

- El compilador comprueba las opciones definidas en dicho fichero para generar a partir de ahí el código javascript.

TypeScript - TSC

- Vemos un ejemplo de tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "noImplicitAny": false,
    "removeComments": false,
    "sourceMap": true
  },
  "exclude": [
    "node_modules",
    "dist/"
  ]
}
```

TypeScript - TSC

- Mediante el campo **target** estamos indicando qué tipo de código vamos a generar en la compilación.

- En este caso tendremos soporte para todos aquellos navegadores con ES5.

- Mediante **module** indicamos el esquema que vamos a usar para compilar. En este caso, utilizamos commonjs, el cual es compatible con NodeJS.

- Activamos también todas las opciones que nos permiten activar el uso de Decoradores, como veremos más adelante en este tema.

- Con el parámetro **exclude** indicamos qué directorios no queremos compilar.

TypeScript - TSC

- Una vez definido este fichero de configuración, para compilar nuestros proyectos, únicamente tendríamos que llamar a **tsc** sin argumentos y ya se encarga de recoger todos los ficheros de tipo .ts que encuentre en el mismo directorio donde se encuentra el fichero de configuración.

- Podemos especificar también qué ficheros son los que queremos compilar:

```
{
  "compilerOptions": {
    …
  },
  "files":[
    "test.ts",
    "prueba.ts"
  ]
  "exclude": [
   …
  ]
}
```

TypeScript - Types

- Aunque hayamos escuchado muchas veces la frase “*Javascript no tiene tipos”*, no es del todo cierto.
- Sí que tiene tipado pero, el desarrollador no tiene que preocuparse por ellos.
- Los tipos de Javascript también existen en TypeScript:
  - **boolean** (true/false)
  - **number** enteros, float, Infinity y NaN
  - **string** caracteres y cadenas
  - **[ ]** arrays de diferentes tipos
  - **{ }** objetos literales
  - **undefined** algo no definido.

TypeScript - Types

number

- Como pasa en Javascript, todos los números que podemos utilizar en TypeScript son de tipo coma flotante.

- Además de los números enteros y decimales, en TypeScript también podemos trabajar con números hexadecimales, valores binarios y octales.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

TypeScript - Types

string

- Uno de los tipos más importantes para el trabajo con aplicaciones web o sobre servidores es el encargado de trabajar con literales.

- Las cadenas de texto literales en TypeScript, al igual que en Javascript, las definimos a partir de comillas simples o comillas dobles.

- Aparte, podemos utilizar las plantillas de string para poder expandirlas a más de una línea e intercalar cómodamente las variables que necesitemos.

```
let cadena: string = "Hola Mundo";

let nombre: string = "Anabel";
let direccion: string = "Mayor, 7";

let frase: string = `Me llamo ${nombre}
y vivo en la calle ${direccion}`;
```

TypeScript - Types

array

- TypeScript permite el trabajo con arrays de la misma manera que en Javascript.
- Podemos definir los arrays de dos maneras diferentes, siempre y cuando definamos el tipo de los elementos que vamos insertar.

```
let lista: number[] = [1,2,3];

let lista2: Array<number> = [1,2,3];
```

TypeScript - Types

tuplas

- Las tuplas nos permiten expresar un array donde los tipos de los elementos que vamos a utilizar son conocidos, pero no tienen por qué ser el mismo.
- Por ejemplo, podemos expresar un par de valores representados por un string y un número.

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

x = ["cadena", 30];

x = [30, "cadena"]; //ERROR
```

TypeScript - Types

- Aparte, TypeScript agrega los siguientes tipo:
  - **enum** enumeraciones como *{Rojo, Verde, Azul}*
  - **any** posibilidad de usar cualquier tipo
  - **void** nada

TypeScript - Types

enum

- Los elementos de tipo enum, agregan la posibilidad de asignar una serie de nombres más accesibles para valores de tipo numérico.

```
enum Color {Rojo, Verde, Azul};
let c: Color = Color.Verde;
```

- Por defecto, los valores asignados en un enum, empiezan desde 0. Se puede cambiar el valor de comienzo para los elementos, o incluso especificar uno concreto para cada uno de ellos.

```
enum Color {Rojo = 1, Verde, Azul}; // Empieza por 1
enum Color {Rojo = 2, Verde = 4, Azul = 6};
```

TypeScript - Types

enum

- Una de las características más útiles del tipo enum, es la posibilidad de poder recuperar el nombre de cada uno de los elementos contenidos en su interior.

- Simplemente tenemos que acceder a través de la posición.

```
enum Color {Rojo = 1, Verde, Azul};

console.log(Color[2]); // Devuelve Verde
```

TypeScript - Types

any

- En muchas ocasiones necesitamos definir variables de las que desconocemos, de primeras, el tipo que representan.

- Suelen ser valores obtenidos desde algún tipo de contenido dinámico o desde la carga de datos del usuario o de librerías de terceros.

- Para estos casos, disponemos del tipo **any**, el cual nos permite realizar las comprobaciones de tipo en tiempo de ejecución.

```
let notSure: any = 4;
notSure = "podria ser una cadena";
notSure = false; // finalmente un boolean
```

TypeScript - Types

any

- El tipo any también es útil cuando trabajamos con colecciones en las que conocemos sólo alguno de los tipos de los elementos que las conforman.

- Por ejemplo, podríamos utilizarlo en un array que contiene diferentes elementos.

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

TypeScript - Types

void

- Podríamos definir el tipo void como el caso contrario a any.

- En este caso queremos definir la ausencia de tipo en la variable con la que estemos trabajando.

- El lugar más común donde vamos a poder encontrarlo es en la definición del valor de retorno en una función.

```
function sinRetorno(): void{
  console.log("Función sin return");
}
```

- Declarar variables con el tipo void no es muy útil ya que sólo podemos asignarle los valores *undefined* o *null*.

TypeScript - Types

Type assertions

- Vamos a encontrarnos muchos casos, sobre todo al trabajar con any, en los que necesitemos justificar qué tipo tiene alguna variable y así poder usar los métodos específicos de ese tipo.

- Lo podríamos comparar con los *casting* de otros lenguajes de programación.

- Al contrario que otros lenguajes, este tipo de comprobaciones son a nivel de compilador y no tienen ninguna repercusión en tiempo de ejecución.

TypeScript - Types

Type assertions

- Tenemos dos maneras para poder llevar a cabo este tipo de *casting*.

```
let valor: any = "Esto es un string";
let strLength: number = (<string>valor).length;
```

```
let valor: any = "Esto es un string";
let strLength: number = (valor as string).length;
```

- Las dos formas de proceder son equivalentes.

TypeScript - Funciones

- En el ejemplo anterior hemos visto la implementación de una función sencilla.

- Observamos como, a la hora de declarar la función, especificamos el tipo del parámetro con el que vamos a trabajar y, de igual manera, el tipo del valor devuelto por la función.

- Estos tipos serán comprobados por **tsc** cuando se ejecute, para comprobar la integridad de la función.

TypeScript - Funciones

- Es muy común en Javascript, el uso de parámetros opcionales dentro de las funciones.

- Mediante el uso de **?** podemos indicar qué parámetros serán opcionales en nuestras funciones y de esta manera, avisaremos al compilador de este hecho.

```
function mandarMensaje(mensaje: string, esDebug?: boolean){
  if (esDebug){
    console.log ("DEBUG: " + mensaje);
  }else{
    console.log (mensaje);
  }
}

mandarMensaje("Hola Mundo");
mandarMensaje("probando", true);
```

TypeScript - Interfaces

- Las interfaces definen descripciones abstractas de diferentes elementos y pueden ser usadas para representar cualquier objeto no primitivo de Javascript.

- Dentro de estas interfaces definimos cómo son las declaraciones de los métodos que forman parte de la misma.

- Los elementos que implementan dichas interfaces serán los encargados de implementar dichas funciones.

TypeScript - Interfaces

- Vemos un ejemplo de una interfaz que define una única función.

```
interface Callback {
  (error: Error, data: any): void;
}

function callServer(callback: Callback) {
  callback(null, 'hola');
}

callServer((error, data) => console.log(data)); // 'hola'
callServer('hi'); // Error de compilación
```

TypeScript - Interfaces

- Este ejemplo también podemos usarlo para trabajar con sobrecarga de funciones, definiendo dentro de la interfaz las diferentes posibilidades que tenemos:

```
interface MostrarMensaje{
  (mensaje: string): void;
  (mensaje: string[]): void;
}

let mostrar: MostrarMensaje = (mensaje) => {
  if (Array.isArray(mensaje)){
    console.log (mensaje.join(' '));
  }else{
    console.log (mensaje);
  }
}

mostrar("Hola Mundo");
mostrar(["3","5","6"]);
```

TypeScript - Interfaces

- Si quisiéramos definir una interfaz para un objeto literal, podemos hacerlo de la siguiente manera

```
interface Action {
  type: string;
}

let a: Action = {
  type: 'literal'
}

console.log(a.type);
```

TypeScript - Decoradores

- Los decoradores estaban planificados para usarse en una versión posterior de Javascript, pero los desarrolladores de Angular querían usarlos y los han incluido en TypeScript.

- Los decoradores son funciones declaradas a través del símbolo @ e inmediatamente seguidas por una clase, un parámetro o un método.

- Existen 4 elementos diferentes que pueden ser *decorados* (clases, parámetros, métodos y propiedades)

- Por lo tanto, existen 4 formas diferentes de declarar los decoradores.

TypeScript - Decoradores

- **class:** *declare type ClassDecorator = <TFunction extends Function>(target: TFunction)=> TFunction* | *void;*

- **property:** declare type PropertyDecorator = (target: Object, propertyKey: string |symbol) => void;

- **method:** declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;

- **parameter:** declare type ParameterDecorator = (target: Object, propertyKey: string |symbol, parameterIndex: number) => void;

TypeScript - Decoradores

- Los decoradores podemos aplicarlos sobre las **propiedades de una clase**.

```
function Override(label: string) {
  return function (target: any, key: string) {
    Object.defineProperty(target, key, {
      configurable: false,
      get: () => label
    });
  }
}

class Test {
  @Override('test') // llama a la función Override
  name: string = 'pat';
}

let t = new Test();
console.log(t.name); // 'test'
```

TypeScript - Decoradores

- En el caso anterior, el decorador Override es una función que se encarga de modificar el valor de la propiedad sobre la cual se aplica (target).

- El valor que asignamos, es el que estamos pasando por parámetro al método (label).

- Para poder ejecutar dicho ejemplo, tenemos que hacerlo con el modificador —*experimentalDecorators.*

TypeScript - Decoradores

- Podemos incluso crear decoradores, los cuales no necesiten ningún tipo de parámetro para poder realizar sus funciones.

```
function ReadOnly(target: any, key: string) {
  Object.defineProperty(target, key, { writable: false });
}

class Test {
  @ReadOnly // no
  name: string;
}

const t = new Test();
t.name = 'jan';
console.log(t.name); // 'undefined'
```

- En este caso, se define la propiedad como de sólo lectura y no se puede modificar una vez creada la instancia de la clase.

TypeScript - Decoradores

- Si aplicamos un decorador sobre una clase, el *target* que usemos será la propia clase.

- La ejecución de la función representada por el decorador se producirá cada vez que se cree una instancia de la clase decorada.

TypeScript - Decoradores

- En el siguiente ejemplo vemos cómo aplicar un decorador sobre uno de los parámetros de un método.

```
function logPosition(target: any, propertyKey: string, parameterIndex: number) {
  console.log(parameterIndex);
}

class Cow {
  say(b: string, @logPosition c: boolean) {
    console.log(b);
  }
}

new Cow().say('hello', false);
```