

Semana 1

Curso de Angular



Semana 1 – Anexo 1 **Introducción a TypeScript**

Programación web con Angular
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

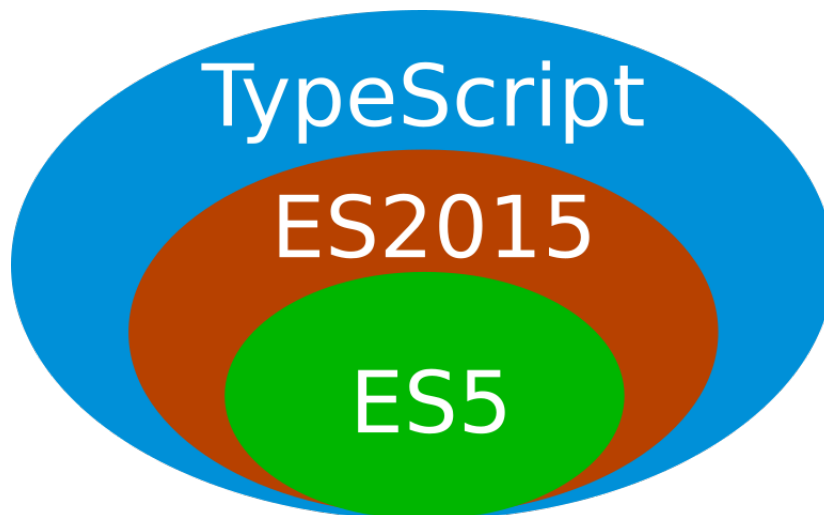
Index

Introduction.....	3
Valores tipados.....	4
Tipo Array.....	4
Tipo Object.....	4
Tipo Function.....	5
Clases e interfaces en TypeScript.....	6
Constructor.....	6
Getters y setters.....	6
Casting de tipos.....	7
Herencia de clases (extends).....	8
Implementar interfaces en clases.....	8
Definir propiedades de objetos con interfaces.....	8
Otras características de TypeScript.....	10
Definición de archivos (TypeScript 2.x).....	10
Depuración en Chrome.....	10

Introduction

Para un gran número de aplicaciones, JavaScript quizás no sea el lenguaje más apropiado. Puede serlo si somos muy cuidadosos y separamos nuestra aplicación en clases y módulos utilizando las últimas características de ES2015, pero muchos desarrolladores tienden a escribir código espagueti (largos archivos con mucha funcionalidad mezclada), y no tienen cuidado con los tipos de datos que se le envían a las funciones (usando `==` en lugar de `===`, por ejemplo, simplemente por no preocuparse por los tipos de datos, lo cual es un error).

[TypeScript](#) es un superconjunto de JavaScript. Eso significa que tiene todas las características que tiene JavaScript (la última versión incluida), además del tipado de variables, y otras que formarán parte de JavaScript en el futuro.



TypeScript es un lenguaje muy accesible para desarrolladores que ya están familiarizados con lenguajes como Java o C#. Además, no hay problemas de compatibilidad con navegadores (o entornos servidor como NodeJS), ya que se compila (transpila) a JavaScript puro (lo haremos con Webpack).

Otros lenguajes que compilan a JavaScript son [CoffeeScript](#) y [Dart](#), pero TypeScript es el más próximo a JavaScript (como ya hemos mencionado, es un superconjunto). De hecho, si copiamos y pegamos JavaScript dentro de un archivo TypeScript (.ts), funcionará perfectamente.

Conociendo la versión ES2015 y posteriores, aprender TypeScript es muy fácil, ya que hay pocas diferencias entre ambos. De hecho, las diferencias más notables son que TypeScript soporta tipos (declarar una variable o una función de tipo número, string, booleano, objeto, etc.) y muchas características avanzadas de los Lenguajes de Programación Orientados a Objetos, como el ámbito público y privado de métodos y propiedades, propiedades opcionales, interfaces, etc.

Valores tipados

En TypeScript, todas las variables están tipadas por defecto (number, boolean, string, *nombre de clase*, ...), por tanto, el compilador no nos permitirá asignarle posteriormente un valor de tipo diferente. Podemos usar el tipo especial **any**, que nos permite que una variable se comporte como en JavaScript (sin tipo definido).

```
let a: any; // La variable a puede tener cualquier valor de cualquier tipo
let b; // Igual que a: cualquier tipo (por omisión es any).
a = 3;
a = "asdf";
b = 5;
b = true;
```

Si queremos forzar una variable para que contenga valores de un tipo específico podemos hacerlo de dos formas: declarar la variable con **nombre:tipo**, o asignar directamente un valor cuando la declaramos.

```
let a: number; // La variable a puede solo contener números
let b = 4; // Igual que b: números (tipo implícito en la asignación).
a = 3; // OK
a = "asdf"; // Error: 'string' not assignable to 'number'
b = true; // Error
let c: number = 12; // OK (tipo y asignación al mismo tiempo, aunque es redundante)
let d: number = "hello"; // Error
```

Podemos declarar los tipos de datos que se le pasan a una función o método y lo que nos devuelve. El compilador nos dará un error si no se cumple lo prometido.

```
class Person {
  private name: string;

  constructor(name: string) { // Recibe un parámetro de tipo string
    this.name = name;
  }

  getName(): string { // Devuelve un valor string
    return this.name;
  }
}
```

Tipo Array

Como los arrays son objetos de la clase **Array**. Es válido declararlos como **tipo[]**, o como **Array<tipo>** (siendo **tipo** el tipo de dato que contiene el array). En este caso estamos limitando a que en el array sólo puede haber elementos de un tipo de datos.

```
let a: Array<string>;
let b: string[];
a = ["hello", "goodbye"];
b = ["a", "b", "c"];
```

Tipo Object

Aunque es mejor usar interfaces, como pronto veremos, para definir las propiedades de un objeto, podemos tipar una variable indicando las propiedades que tendrá el objeto asignado (y su tipo):

```
// Recibe un parámetro que es un objeto con las propiedades length (number) y
str (string)
function isRightLength (obj: { length: number, str: string}): boolean {
    return obj.str.length == obj.length;
}

let o : { length: number, str: string} = { // La variable contendrá un objeto con
estas propiedades
    length: 5,
    str: "hello"
};
console.log(isRightLength(o)); // true
```

Las funciones lambda funcionan exactamente igual. A veces no es necesario definir el tipo de datos que devuelve una función. El compilador en este caso, analizará la sentencia return para saber qué devuelve.

```
// (obj: { length: number, str: string}) : boolean no es necesario (devuelve una
condición → boolean).
let isRightLengthArrow = (obj: { length: number, str: string}) => obj.str.length
== obj.length;;
console.log(isRightLengthArrow(o)); // true
```

Podemos también declarar un parámetro como opcional (puede no estar en el objeto) mediante el símbolo ?. Tras el nombre del parámetro.

```
function areaRect(rect: {a: number, b?: number}): number { // La propiedad "b" es
opcional
    if(!rect.b) return rect.a * rect.a;
    return rect.a * rect.b;
}

console.log(areaRect({ a: 10 })); // Imprime 100. El objeto no contiene la
propiedad "b" pero es opcional
console.log(areaRect({ a: 5, b: 8 })); // Imprime 40
```

Tipo Function

Podemos asignar como tipo de una variable una función. Esto se representa con un tipo de construcción parecida a una lambda. Entre paréntesis irán los parámetros que se reciben y después de la flecha el tipo de valor que devolverá. Esa variable sólo podrá referenciar una función con las mismas características. Usa **void** como tipo de dato cuando no devuelva nada.

```
let func : (str: string, num: number) => string; // Dos parámetros (string, number)
y devuelve un string
func = function (str, num) {
    let result = ""; // No es necesario decirle al compilador que es un string
    (implícito)
    for(let i = 0; i < num; i++) result += str;
    return result; // Debe devolver un string, en el caso contrario no compilará
}
console.log( func("a", 6) ); // Imprime "aaaaaa" (debe ser llamado con un string
y un number o no compilará)
```

Clases e interfaces en TypeScript

Las clases en TypeScript son muy parecidas a las clases en ES2015, con más características. Una de las principales diferencias es que podemos establecer las propiedades y métodos como **públicos** o **privados** (en ES2015 todo es público). Además, declararemos las propiedades fuera del constructor:

```
class Rect {
  private width: number = 5;
  private height: number = 8;

  public getArea(): number { // 'public' puede ser omitida (por defecto todo
es public)
    return this.width * this.height;
  }
}

let rect = new Rect();
rect.width; // Error: Property 'width' is private and only accessible within
class 'Rect'
```

Constructor

El método del constructor es igual al que usamos en ES2015, pero tiene otra característica. Podemos crear propiedades directamente en los parámetros del constructor. Para ello, ponemos la palabra reservada **public** o **private** delante del parámetro. Internamente se creará una propiedad con el mismo nombre en el objeto y se le asignará el valor (**this.propiedad = propiedad**).

```
class Rect {
  constructor(public width: number, private height: number) { } // propiedades
creadas y asignadas

  public getArea(): number {
    return this.width * this.height;
  }
}

let rect = new Rect(5, 7);
console.log(rect.getArea()); // Imprime 35
```

Getters y setters

Hay dos modos para crear getters y setters. La forma clásica es tener un atributo privado, y dos métodos llamados `getAttribute` y `setAttribute` para obtener y modificar el valor de forma controlada:

```
class Person {
  constructor(private name: string) {}

  getName(): string { // Devuelve un string
    return this.name;
  }

  setName(name: string) { // Devuelve un string
    this.name = name;
  }
}
```

```
}
```

Y la equivalente a usar **Object.defineProperty** en JavaScript, donde se llaman automáticamente cuando se asigna (set) o se lee (get) el valor de la propiedad.

Archivo: src/main.ts

```
class Person {
  private _name: string;

  constructor(name: string) {
    this._name = name; // Llamada
    // implícita a setter
  }

  get name(): string { // Devuelve un
    // string
    return this._name;
  }

  set name(name: string) { // Devuelve
    // un string
    this._name = name;
  }
}

let p: Person = new Person("Peter");
console.log(p.name); // Llamada al
// getter implícito
```

Archivo: js/main.js (compiled version)

```
var Person = (function () {
  function Person(name) {
    this._name = name; // llama
    // implícita al setter
  }

  Object.defineProperty(Person.prototype,
    "name", {
      get: function () {
        return this._name;
      },
      set: function (name) {
        this._name = name;
      },
      enumerable: true,
      configurable: true
    });
  return Person;
})();
```

Casting de tipos

Si tenemos un objeto de tipo **ClassA** (por ejemplo), y sabemos que ese objeto es también una instancia de la clase **ClassB**, la cual hereda de ClassA (un subtipo, vamos), podemos hacer un casting (conversión) al nuevo tipo poniendo **<ClassB>** delante. De una clase derivada a una clase padre, el casting es automático.

```
let table: HTMLTableElement =
  <HTMLTableElement>document.getElementById("table1"); // HTMLElement
let elem: HTMLElement = table; // Puede ser asignado directamente (HTMLElement
// es una clase padre)
```

Algunos métodos como `getElementById`, devuelven un objeto del tipo `HTMLElement`. Es una interfaz que todos los elementos del DOM implementan, pero solo permite el acceso a las propiedades comunes. Si queremos acceder a las propiedades específicas de un elemento, debemos hacer un cast al tipo de objeto específico que sabemos que es (o podemos averiguar usando `instanceof`).

```
let input: HTMLInputElement = <HTMLInputElement>document.getElementById("input1");
// HTMLElement
let val: string = input.value; // Ahora podemos acceder a la propiedad "value"

let val2: string = (<HTMLInputElement>document.getElementById("input1")).value; //
// En un solo paso

let element = document.getElementById("elem1"); // HTMLElement

if(element instanceof HTMLImageElement) { // Comprobamos si el objeto es del
  // tipo HTMLImageElement
  console.log(<HTMLImageElement> element).src; // Hacemos el cast
```

```
}
```

Herencia de clases (extends)

La principal diferencia con ES2015 en la herencia es que ahora usamos atributos y métodos privados. Estos atributos y métodos no estarán disponibles en la clase derivada, tal como ocurre en otros lenguajes de programación: Java, C#, ...

```
class Vehicle {
  constructor(private name: string) {}

  public setName(name: string) {
    this.name = name;
  }

  public getName(): string {
    return this.name;
  }
}

class Plane extends Vehicle {
  constructor() {
    super("Plane"); // Debemos llamar al constructor del padre siempre
  }

  upgrade() {
    this.name = "Spaceship"; → Property 'name' is private and only
    accessible within class Vehicle
    this.setName("Spaceship"); // OK, esto sí funciona porque setName es
    público
  }
}
```

Implementar interfaces en clases

Las interfaces son contratos de código. Se usan para tener consistencia entre las diferentes clases que tienen características en común. Con una interfaz podemos establecer los métodos que debe tener una clase que implementa dicha interfaz.

```
interface IDatabase {
  getData(Json: any): any[]; // Las clases que la implementen deberán tener
  este método
}

class CarDataBase implements IDatabase {
  getData(json: any): any[] {
    return json.cars;
  }
}

let data: any = {
  people: [ { name: "John", age: 23 }, { name: "Rose", age: 28 } ],
  cars: [ { model: "Tesla", price: 70000 }, { model: "Ford", price: 25000,
  year: 2015 } ]
};

let carDb = new CarDataBase();
console.log(JSON.stringify(carDb.getData(data))); //
[{"model":"Tesla","price":70000},{ "model":"Ford","price":25000,"year":2015}]
```

Definir propiedades de objetos con interfaces

Con las interfaces podemos definir qué propiedades (públicas) debe tener un objeto, e incluso propiedades opcionales. Esto es útil cuando trabajamos con objetos JSON por ejemplo, donde podemos definir las propiedades de un objeto.

```
interface ICar {
    model: string;
    price: number;
    year?: number; // Opcional
}

interface IPerson {
    name: string;
    age: number;
}

interface IAppData {
    people: IPerson[]; // Array de objetos con las propiedades de IPerson
    cars: ICar[]; // Array con las propiedades de ICar
}

interface IData {
    getData(Json: IAppData): Object[];
}

class CarDataBase implements IData {
    getData(json: IAppData): ICar[] { // Recibe un IAppData y devuelve array de ICar
        return json.cars;
    }
}

let data: IAppData = { // Mira la interfaz IAppData
    people: [ { name: "John", age: 23 }, { name: "Rose", age: 28 } ],
    cars: [ { model: "Tesla", price: 70000 }, { model: "Ford", price: 25000,
year: 2015 } ]
};

let carDb: CarDataBase = new CarDataBase();
let cars: ICar[] = carDb.getData(data);
let car: ICar = cars[0];
console.log(JSON.stringify(car)); // {"model":"Tesla","price":70000}
```

Las interfaces también pueden heredar de otras interfaces:

```
interface IA {
    a: string;
}

interface IB extends IA { // Hereda la propiedad "a" de IA
    b: number;
}

let obj: IB = {a: "hello", b: 5}; // Debería tener las propiedades 'a' y 'b'.
```

Importante: Las interfaces y en general el tipado es muy útil de cara a que el editor (uno que entienda TypeScript como Visual Studio Code) nos autocomplete y sepa qué propiedades (o métodos) y de qué tipo deben aparecer cuando ponga el punto detrás de la variable que referencia al objeto, o al crear un objeto, etc. Esta práctica además evita fallos de que nos equivoquemos en el nombre y en el tipo de propiedades, parámetros, variables, etc.

Otras características de TypeScript

Cuando usamos una librería externa de JavaScript, como JQuery, podemos perder la posibilidad del autocompletado ya que no tendremos acceso a los tipos (no están escritas en TypeScript). Para integrar estas librerías con TypeScript debemos instalar (si están disponibles), los **archivos de definición de tipos (TypeScript)**. Este archivo contendrá principalmente **interfaces** que representan las propiedades, métodos (y sus tipos), de las clases y objetos de dicha librería.

Definición de archivos (TypeScript 2.x)

Desde la versión de TypeScript 2.0.0, se ha vuelto más fácil (e integrarlo con NPM) la instalación de archivos de definición en nuestro proyecto.

Primero, para hacer una definición de búsqueda, podemos usar:

<https://microsoft.github.io/TypeSearch/>

Una vez que lo has encontrado, lo instalas usando NPM (Como dependencia de desarrollo). Ejemplo: JQuery:

```
npm install --save-dev @types/jquery
```

Ahora tendremos JQuery integrado con el código TypeScript. Todo lo que debes hacer es importar la librería cuando quieras usarla:

```
import * as $ from "jquery";  
$.ajax(....);
```

Depuración en Chrome

Para cada archivo de TypeScript, el compilador genera dos archivos. El archivo traducido de .js y un mapa (.js.map). Este último archivo es bastante útil para referenciar el código TypeScript a partir del .js, y así depurar directamente desde nuestro código TypeScript en las herramientas de desarrollo.

Para poder depurar, abrimos las herramientas de desarrollo y vamos a la pestaña de **sources**. Podemos ver el código TypeScript y establecer puntos de parada, inspeccionar variables, etc. Recarga la página y debería funcionar.

