Curso de Typescript

Daniel Rojo Pérez

Contenidos

Introducción a TypeScript

TypeScript es un lenguaje de programación libre y de código abierto diseñado por el equipo de C# con soporte de Microsoft. Typescript puede ser usado para desarrollar aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor.

TypeScript extiende la sintaxis de JavaScript, por tanto cualquier código JavaScript existente debería funcionar sin problemas.

Lo necesario para empezar a utilizar TypeScript

Typescript está pensado para grandes proyectos, los cuales a través de un compilador de TypeScript se traducen a código JavaScript original. Para instalar el transpilador de Typescript a Javascript en tu máquina, tienes que escribir el siguiente comando de npm:

> npm install -g typescript

Una vez instalado, llamamos al transpilador con el siguiente comando:

> tsc hello.ts

Fundamentos de Typescript

Tipos en Typescript

any	Any type (explicitly untyped)
void	void type (undefined or null, use for function returns only)
undefined	Undefined type
null	Null type
never	never type
readonly	readonly
string	String (including ES6 multi-line string templates)
number	Number
boolean	Boolean
object	object (may be an Object or non-primitive)

Modificadores de alcance

Definen la visibilidad o las propiedades de nuestra variable dentro del objeto, el método o el namespace

Public (default)	public firstName: string;
Protected	protected inventory: number;
Private	private outOfS-tock: boolean;
Read Only	readonly pi: number = 3.14159;
Static	static log(msg: string) { consol-e.l-og(msg) };

Definición de variables

```
// Ésta es la forma de declarar variables
let isDone: boolean = false;
let lines: number = 42;
let name: string = "Anders";
// Typescript infiere el tipo de las variables por la asignación
let isDone = false;
let lines = 42;
let name = "Anders";
// El uso del tipo any
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // Cambia el tipode variable!
```

```
// Para definir una constante:
const numLivesForCat = 9;
numLivesForCat = 1; // Error
```

Arrays

Los arrays en Typescript son tipados y pueden tener un tamaño variable. Hay métodos para añadir elementos a un array, concatenar arrays, copiar…

Tenemos varias formas de definir arrays, aquí un par de ellas:

```
// Array de number
let list: number[] = [1, 2, 3];
// Mismo array usando la definición genérica
let list: Array<number> = [1, 2, 3];
```

Enum

Nos permite tener una colección ordenada de constantes, por defecto tienen el valor de 0 a n, pero podemos asignarles otros valores o incluso otros tipos.

```
enum Options {
   FIRST,
   EXPLICIT = 1,
   BOOLEAN = Options.FIRST | Options.EXPLICIT
 }
 enum Colors {
   Red = "#FF0000",
   Green = "#00FF00",
   Blue = "#0000FF"
 }
```

Namespace

Es la estructura donde podemos poner varias clases para dar coherencia a nuestro modelo

```
namespace AcmeCorp.Logging {
   export class Logger {
        static log(msg: string) : void {
         console.log(msg);
        };
   }
 }
```

Funciones - Definición

Como en otros lenguajes, definimos parámetros de entrada (opcional), el tipo de salida que da la función (también opcional). Cuando la función no está contenida en una clase, hace falta poner el la palabra reservada function

```
function birthday(year) {
    return year + 1;
}
```

```
class Util {
    sum(num1: number, num2: number): number {
      return num1 + num2;
    }
 }
```

Funciones - definición

Otra forma de definir una función, es la llamada “fat arrow” y de esta forma podemos también definir las funciones anónimas

```
// Parámetros y return explícitos
let f1 = function (i: number): number { return i * i; }
// Return inferido
let f2 = function (i: number) { return i * i; }
// Sintaxis "Fat arrow"
let f3 = (i: number): number => { return i * i; }
// Sintaxis "Fat arrow" con return inferido
let f4 = (i: number) => { return i * i; }
// Sintaxis "Fat arrow" sin llaves significa que no devuelve nada
let f5 = (i: number) => i * i;
```

Funciones - parámetros opcionales

Es conveniente evitar la sobrescritura de métodos, una de las formas que tenemos de evitarlo, es usando los parámetros opcionales para permitir que un método sea llamado con diferente número de parámetros

```
class Util {
   log(msg: string, logDate?: Date) {
     if (logDate)
       console.log(logDate + ' ' + msg);
     else
       console.log(new Date() + ' ' + msg);
   }
 }
```

Funciones - Parámetros adicionales

Son los llamados ‘Rest parameters’, podemos definir un array de entrada de parámetros y recibir tantos como tamaño tenga el array

```
class Order {
   addOrderDetails(...orderDetails: IOrderDetail[]) {
   }
 }
```

Interfaces

Especifica métodos y atributos que tiene que implementar una clase que herede el interfaz, una clase puede heredar más de un interfaz

```
interface Child extends Parent, SomeClass {
   property: Type;
   optionalProp?: Type;
   method(arg1: Type): ReturnType;
   optionalMethod?(arg1: Type): ReturnType;
}
```

Interfaces

Podemos crear un objeto desde un interfaz, de alguna de estas formas

```
interface Person {
   name: string;
   age?: number;
   move(): void;
 }
  // Implemento el Objeto
 let p: Person = { name: "Bobby", move: () => { } };
 // Lo mismo con el atributo opcional
 let validPerson: Person = { name: "Bobby", age: 42, move: () => { }
};
 // Da error
 let invalidPerson: Person = { name: "Bobby", age: true };
```

Interfaces

Podemos definir un método con un interface, en lugar de un objeto completo

```
// Defino el interface
interface SearchFunc {
   (source: string, subString: string): boolean;
 }
 // Solo tiene que coincidir el tipo de los parámetros, no el nombre
 let mySearch: SearchFunc;
 mySearch = function (src: string, sub: string) {
   return src.search(sub) != -1;
 }
```

Definición de atributos en el constructor

Si en un parámetro del constructor viene definido el alcance, significa que ese parámetro es un atributo del objeto que crea

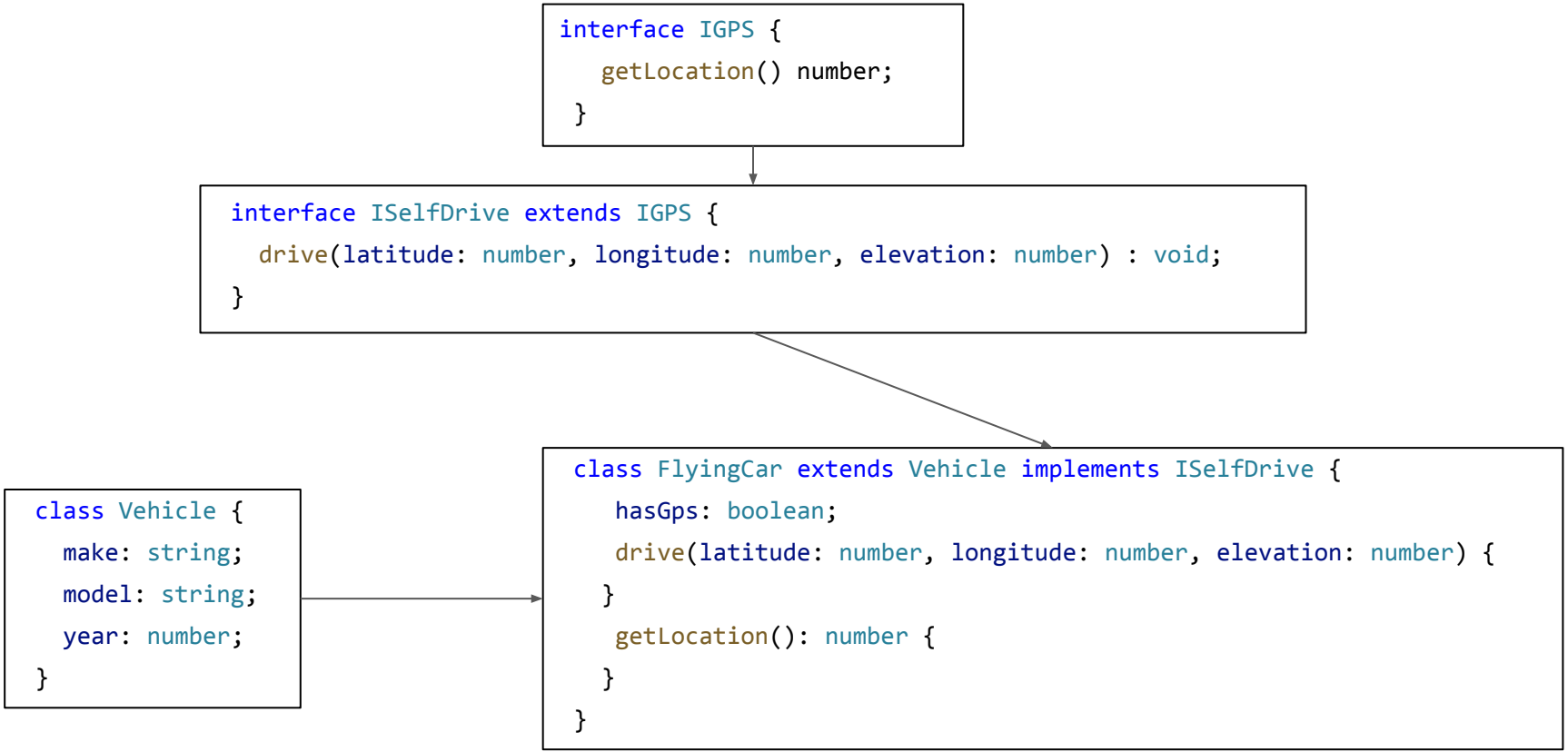
```
class OrderLogic {
   constructor(public order: IOrder) { }

   getOrderTotal(): number {
       let sum: number = 0;

       for (let orderDetail of this.order.orderDetails)
       {
           sum += orderDetail.price;
       }
       return sum;
   }
}
```

Clase abstracta

Es una clase que no puede generar objetos, puede tener métodos static o definir atributos y métodos que sean heredados por clases que no sean abstractas

```
abstract class Person {
   name: string;
   monthlySalary: number;
   monthlyBenefits: number;
    abstract calcSalary(): number;
 }
```

Herencia de clases



Genéricos

Podemos definir interfaces, clases y funciones genéricas, en las que no especificamos el tipo de atributos/parámetros y se pueden usar con varios tipos y objetos

```
class Tuple<T1, T2> { // Classes
   constructor(public item1: T1, public item2: T2) {
   }
 }
 interface Pair<T> { // Interfaces
   item1: T;
   item2: T;
 }
 let pairToTuple = function <T>(p: Pair<T>) { // funciones
   return new Tuple(p.item1, p.item2);
 };
```

Decorators

Para poder usar decorators en nuestro código, primero tenemos que habilitar ‘experimentaldecorators’ en tsconfig.json

> tsc --target ES5 --experimentalDecorators

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators":
true
    }
}
```

Un decorator es una definición que podemos asignar a una clase, método, propiedad...

Ejemplo de decorator

```
@sealed
class Greeter {
   greeting: string;
   constructor(message: string) {
       this.greeting = message;
   }
   greet() {
       return "Hello, " + this.greeting;
   }
}

function sealed(constructor: Function) {
   Object.seal(constructor);
   Object.seal(constructor.prototype);
}
```

Otro ejemplo de decorator aplicado en la clase

```
function classDecorator<T extends
{new(...args:any[]):{}}>(constructor:T) {
   return class extends constructor {
       newProperty = "new property";
       hello = "override";
   }
}
@classDecorator
class Greeter {
   property = "property";
   hello: string;
   constructor(m: string) {
       this.hello = m;
   }
}
console.log(new Greeter("world"));
```

```
class_1 {
  property: 'property',
  hello: 'override',
  newProperty: 'new property'
}
```

Decoradores de métodos

```
function log(target, key, descriptor) {
 const originalMethod = descriptor.value;
 descriptor.value = function () {
   console.log(`${key} was called with:`,arguments);
   var result = originalMethod.apply(this, arguments);
   return result;
 };
 return descriptor;
 }
  class P2 {
 @log
 foo(a, b, c, d, e) {
 console.log(`Do something`);
 }
 }
 const p2 = new P2();
 p2.foo('hello', 1, true, [1,2,3], {property1:'prop'});
```

```
foo was called with: { '0': 'hello',
  '1': 1,
  '2': true,
  '3': [ 1, 2, 3 ],
  '4': { property1: 'prop' } }
Do something
```

Decoradores de propiedad

```
class Greeter {
    @format("Hello, %s")
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        let formatString = getFormat(this, "greeting");
        return formatString.replace("%s", this.greeting);
    }
}
```

Decoradores de acceso

```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }
    @configurable(false)
    get x() { return this._x; }
    @configurable(false)
    get y() { return this._y; }
}

function configurable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor)
{
        descriptor.configurable = value;
    };
}
```

Mixins

Entre las técnicas que tenemos para desarrollar clases de objetos a partir de componentes reutilizables, una de las más populares es el empleo de mixins.

Un mixin es una función que hace:

1. Recibe un constructor
2. Declara una clase que extiende un constructor
3. Añade propiedades y métodos a esa clase
4. Devuelve la clase

Mixins

```
type Constructor<T = {}> = new (...args: any[]) => T;

function Timestamped<TBase extends Constructor>(Base:
TBase) {
   return class extends Base {
       timestamp = Date.now();
   };
}
```

```
class User {
 name: string;

 constructor(name: string) {
     this.name = name;
 }
}

const TimestampedUser =
Timestamped(User);

const user = new TimestampedUser("John
Doe");

console.log(user.name);
console.log(user.timestamp);
```

Mixins con constructores

Podemos definir el constructor del objeto nuevo que creamos

```
type Constructor<T = {}> = new (...args: any[]) =>
T;

function Tagged<TBase extends Constructor>(Base:
TBase) {
 return class extends Base {
     tag: string | null;

     constructor(...args: any[]) {
         super(...args);
         this.tag = null;
     }
 };
}
```

Mixins con métodos

Podemos también añadir métodos a los objetos que creamos con mixins

```
type Constructor<T = {}> = new (...args: any[]) => T;
function Activatable<TBase extends Constructor>(Base: TBase) {
 return class extends Base {
     isActivated = false;

     activate() {
         this.isActivated = true;
     }

     deactivate() {
         this.isActivated = false;
     }
 };
}
```