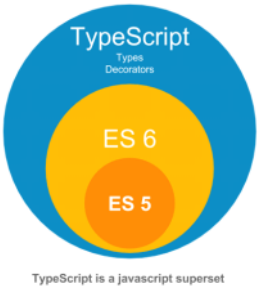
Typescript

sábado, 9 de septiembre de 2017 13:34

http://www.typescriptlang.org/

Open source

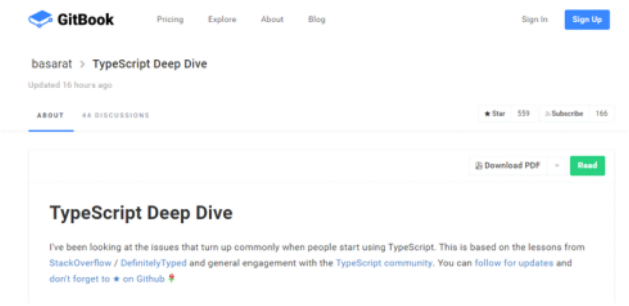
Super Set de JavaScript ES6
(=> es JavaScript)



TypeScript is a javascript superset

- Orientado a Objetos con clases
- Añade **tipos estáticos**
  - Inferencia de tipos
    (no hay que declararlos en muchos sitios)
  - Tipos opcionales (si no quieres, no los usas)
- **Anotaciones / Decoraciones**
- Es ***transpilado***: se genera código JavaScript ES5
  (compatible con los navegadores web actuales)

Documentación on-line

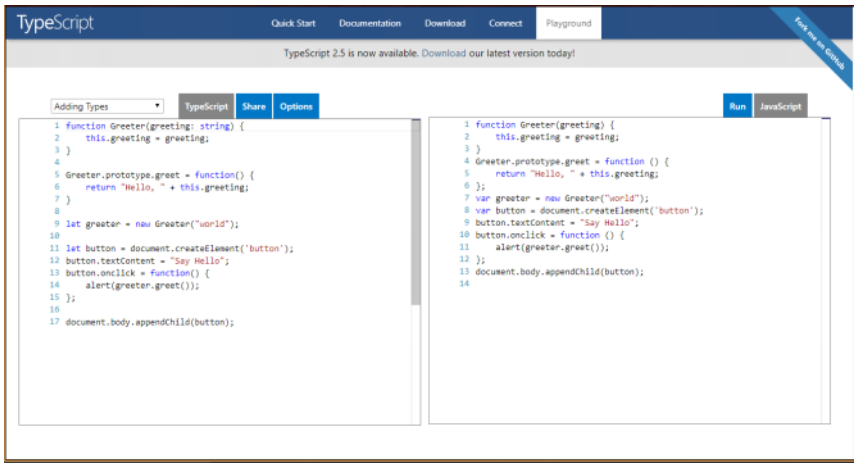


https://www.gitbook.com/book/basarat/typescript/details

Basarat Ali Syed

Pruebas del código

http://www.typescriptlang.org/play/



Clases

sábado, 29 de julio de 2017 15:33

clase

propiedades

constructor

métodos

```
// ejemplo de clase en TypeScript

export class Empleado {

    private nombre: string;
    private salario: number;

    constructor(nombre: string, salario: number) {
        this.nombre = nombre;
        this.salario = salario;
    }

    getNombre() {
        return this.nombre;
    }

    toString() {
        return "Nombre:" + this.nombre +
        ", Salario:" + this.salario;
    }
}
```

Al estándar de ES6 se le añaden

- Propiedades definidas fuera de los métodos
- Modificadores de acceso (*private, protected, public*)
- Interfaces

Herencia

```
class Animal {
    constructor(public name: string) { }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}
```

Herencia a partir de una clase padre

Llamada al constructor de la clase padre

Sobre escritura de los métodos de la clase padre

Interfaces

Los interfaces son siempre públicos, por lo que no se utilizan modificadores de acceso

```
interface Usuario {
    id: number,
    name: string,
    direccion: {calle : string,
                num : number,
                zip: string}
    formación?: Array<string>
    saludar: Function;
    calcularPrecio(iva: number): void;
}
class Socio implements Usuario {
    id: number;
    name: string;
    direccion: {calle : string,
    num : number,
    zip: string}
    saludar() {
        console.log(`Hola, saludos de ${name}`)
    };
    calcularPrecio(iva) {
        …
    }
}
```

elemento opcional, no tiene que estar en la implementación

Método declarado en el interfaz y su correspondiente implementación

Los interfaces también se utilizan para definir tipos, como veremos a continuación

Tipos

domingo, 10 de septiembre de 2017 22:40

Tipos estáticos en tiempo de desarrollo;
evidentemente desaparecen tras la compilación (*traspilación*) a JS

tras el nombre con ":" (dos puntos) →

```
nombre: string;
salario: number;
esFijo: boolean
```

tipos de datos complejos (i.e. objetos JS : *arrays*, objetos…) →

```
aDatos : string[]
aUsuarios: Array<string>

oUser: {name: string}
```

← "genéricos" para complementar los tipos que los admiten

Inferencia de tipos
(no hay que declararlos en muchos sitios) →

```
getNombre() {return this.nombre;}
```

equivale a →

```
getNombre : string () {return this.nombre;}
```

Tipos unión →

```
let id: string | number;
```

Tipo *any*

→ flexibilidad: tipos opcionales (pueden no usarse)

Interfaces y tipos

Se pueden declarar interfaces exclusivamente con propiedades *tipadas* →

```
interface Usuario {
    id: number,
    name: string,
    direccion: {calle : string,
                num : number,
                zip: string}
    formación: Array<string>
}
```

Posteriormente, dentro de las clases se pueden usar esos interfaces, sin necesidad de implementarlos, para la declaración de tipos →

```
class Empleado {
    …
    public oUser: Usuario;
    public aUsers: Array<Usuario>;
```

A partir de ahí se pueden crear objetos literales “tipados” por interfaces

Alias

Crea un nuevo nombre para un tipo o conjunto de tipos

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
```

- aumenta el contenido semántico de los tipos
- agrupa conjuntos de tipos

https://www.typescriptlang.org/docs/handbook/advanced-types.html

Módulos (y elementos de ES6)

domingo, 10 de septiembre de 2017 22:49

empleado.ts
fichero en el que se crea y exporta una clase

```
export class Empleado {
    nombre : string;
    salario: number;

    constructor (pNombre, pSalario)
    {
        this.nombre = pNombre;
        this.salario = pSalario;
    }
}
```

sample.ts
fichero en el que se importa y utiliza la clase anterior

```
import { Empleado } from "./empleado";

let emps = new Array<Empleado>();

emps.push(new Empleado('Pepe', 500));
emps.push(new Empleado('Juan', 200));
emps.push({"Luis",400}}

for (let emp of emps) {
    console.log(emp.getNombre());
}

emps.forEach(emp => {
    console.log(emp);
});
```

Importación desde otro módulo (fichero). Se sobreentiende la extensión *.ts*

Tipo Array de objetos de la clase importada

código ES6 dentro de *TypeScript*

La ausencia de soporte del estándar ES6 en los navegadores o en *Node* hace necesaria la transpilación a ES5 cuando se utilizan módulos en TS
Desde TS se configura el uso de *Node*/*CommonJS* o sistemas de módulos alternativos.

Decoradores o anotaciones

sábado, 14 de octubre de 2017 19:13

Ejemplo de función que define un *decorator* sencillo, sin argumentos

```
function course(target) {
    Object.defineProperty(
        target.prototype,
        'course',
        {value: () => "Angular 2"}
    )
}
```

Uso del anterior *decorator* para modificar una clase

```
@course
class Person {
    firstName;
    lastName;
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Instanciación de un objeto de esta clase

```
let oPersona = new Person("Pepe", "Pérez");
console.log(oPersona.course()); // Angular 2
```

Ejemplo de función que define un *decorator* con argumentos

```
function Student(config) {
    return function (target) {
        Object.defineProperty(
            target.prototype,
            'course',
            {value: () => config.course}
        )
    }
}
```

La función recibe como parámetro el objeto de configuración

al definir el decorador utiliza la clave correspondiente del objeto recibido

Uso del anterior *decorator* para modificar una clase, pasándole un argumento en forma de objeto

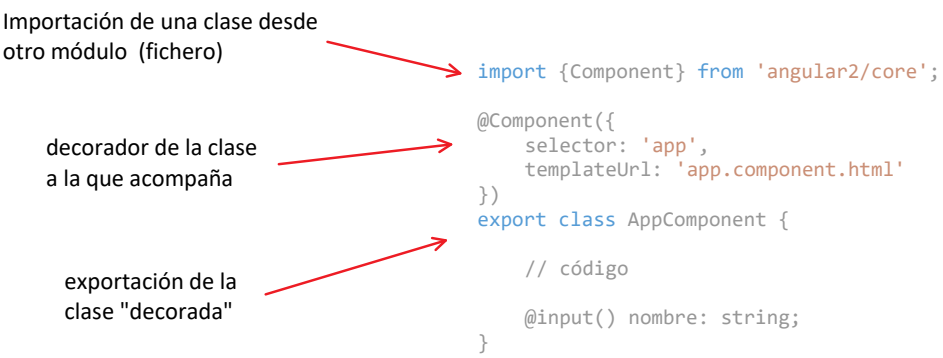
```
@Student({
    course: "Angular 2"
})
class Persona {
    firstName;
    lastName;
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Instanciación de un objeto de esta clase

```
let oEstudiante = new Persona("Pepe", "Pérez");
console.log(oEstudiante.course()); // Angular 2
```

Decoradores en Angular

domingo, 10 de septiembre de 2017 23:04



El decorador es un objeto JSON que da valor a una serie de METADATOS (propiedades) que esperan ser definidas y que pasarán a formar parte de la clase decorada

Otras características

domingo, 10 de septiembre de 2017 23:02

- *Getter* / *Setter* con sintaxis de atributo
- *Type guards Instanceof / typeof*
- Compatibilidad de tipos estructural
- Sobrecarga de métodos “especial”