Applicazioni e framework per PWAs: TypeScript

Ing. Luigi Brandolini

Programma del corso

- TypeScript
- Angular Core
- Tecniche di sviluppo PWAs
- Reactive Programming: NgRx

TypeScript

L. Brandolini

Sillabo

1. Introduzione
2. Setup
3. Type inference / Type *checking*
4. Basics
5. Data Types
6. Template strings (*Back Ticks*)
7. Destructuring
8. Object Oriented syntax
9. Type Compatibility
10. Get / Set
11. Arrow Functions (properties)
12. Namespace e Moduli
13. Decorators

https://www.typescriptlang.org/docs/handbook/

Introduzione

- Il linguaggio maggiormente consigliato per lo sviluppo di applicazioni in Angular è il TypeScript, ovvero un superset di JS

- Serve a superare le difficoltà di sviluppare codice basato su oggetti senza regole di tipizzazione

- È stato sviluppato e rilasciato da Microsoft nel 2012

- Rappresenta un superset di funzioni JavaScript

Introduzione

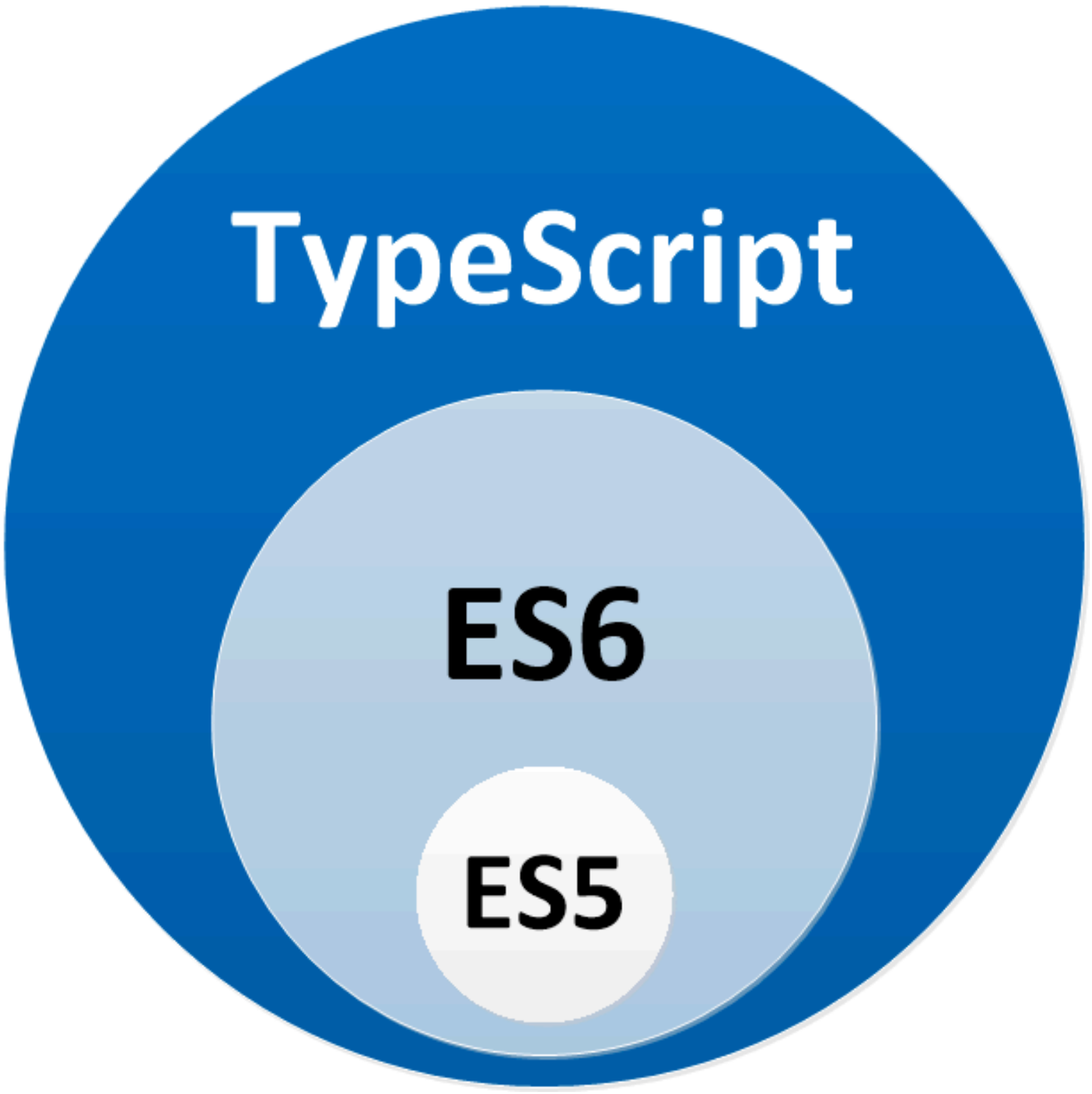
- Strumenti di sviluppo:
  - **Node.Js**
    https://nodejs.org/it/
  - **TypeScript**
    https://www.typescriptlang.org/
  - **Visual Studio Code**
    https://code.visualstudio.com/Download

Introduzione

- Supporta la tipizzazione stretta e genera, dopo un processo di compilazione, JavaScript puro
- Il compilatore è chiamato transpiler
- È basato su un importante principio:

Ogni file JS valido (senza errori) è anche un file TypeScript valido.

Introduzione



Setup

- Installazione della piattaforma (transpiler):

```
npm install -g typescript
```

(in alcuni sistemi è necessario eseguire in modalità 'sudo')

- Controllo della versione installata:

```
tsc -v
```

Setup

- I file TypeScript hanno estensione `.ts`
- La compilazione del sorgente avviene tramite il comando:

```
tsc <filename.ts>
```

- L'output sarà un file `<filename>.js`, importabile nel progetto
- In caso di errori riportati in fase di compilazione, il file di output viene comunque generato
- Per eseguire lo script in assenza di browser è possibile ricorrere al comando da eseguire nel terminale:

```
node <filename.js> o node <filename>
```

JavaScript

- Esempi:

```
var person = {nome : 'Mario', nome: 'Rossi'}
```

JS non genera un errore: la seconda occorrenza della property 'nome' sostituisce la prima

JavaScript

Supponiamo di avere la seguente funzione:

```
function somma(a,b) {

return a + b;

}

somma(1,2); // OK 3

somma("Hello ","World!"); // OK "Hello World!"
```

TypeScript

Il TypeScript serve ad evitare ambiguità di questo tipo, associando con precisione un tipo ad ogni parametro:

```
function somma(a:number, b:number) {

 return a + b;

}


somma(1,2.5); // OK 3.5

somma("Hello ","World!"); // ERRORE
```

TypeScript

- Non essendo stato espresso alcun tipo di dato di ritorno, la funzione può ancora restituire qualsiasi cosa:

```
function somma(a:number, b:number) {
 return "Hello " + "World!";
}

somma(1,2.5); // OK "Hello World"
```

TypeScript

- TypeScript ci permette, però, di eseguire un controllo anche sul valore di ritorno:

```
function somma(a:number, b:number):number {
  return "Hello " + "World!"; //ERRORE
}
```

Per cui la forma corretta diventa:

```
function somma(a:number, b:number):number {
  return a + b; //OK
}
```

Variabili

- Una variabile può essere dichiarata mediante la classica keyword JS **`var`**:

```
var a = 10;

function f() {
  var message = "Hello, world!";

  return message;
}
```

- Ma questa keyword presenta alcune problematiche di **scoping**. Consideriamo il seguente esempio:

```
for(var i = 0; i < 10; i++) {
 console.log(i);
}

 console.log(i); // OK
```

Variabili

- Già in ES 6 è presente la keyword `let` che permette di creare variabili con scoping ridotto:

```
let x = 10;
let isDone = true;
let name: string = 'Hello';
```

- L'esempio precedente, diventa:

```
for(let i = 0; i < 10; i++) {
  console.log(i);
}


console.log(i); // ERROR
```

Variabili

- Altra problematica presente con la keyword var è quella della **shadowing**, ovvero, in JS sono ammesse cose del tipo:

```
function f() {
  var x = 100;
  var x = 200; // OK
  console.log(x); //200
}
```

Variabili

- L'uso di `let` aiuta anche in questo caso a prevenire certe situazioni potenzialmente errate:

```
function f() {
  let x = 100;
  let x = 100; // Error
}
```

```
function g() {
  let x = 100;
  var x = 100; // Error
}
```

Const

- Un'evoluzione del `let` è il `const` che consente di definire variabili il cui valore non può essere riassegnato, ovvero **costanti**:

```
const pi = 3.14;
```

- Fare, però, attenzione che il valore dello stato contenuto nella variabile non è *immutabile*, ma, al contrario, continua ad essere *modificabile*:

```
const person = { name: "Luigi", surname: "Brandolini" };

person.name = "Pippo"; // OK
```

Basic data-types

- Boolean
- Number
- String
- Array
- Tuple
- Enum
- Any
- Unknown
- Void
- Null and Undefined
- Object
- Union Types

Boolean

- È la tipologia di dato più semplice (true o false):

```
let isDone: boolean = true;
```

Number

- È basato sulla stessa rappresentazione in virgola mobile dei numeri in JS:

```
let decimal: number = 6;
let hex: number = 0xf00d;
```

- Oltre ai decimali ed esadecimali, TS supporta anche **binari** ed **ottali** aggiunti a partire dalle specifiche ES6:

```
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

- È possibile definire valori testuali mediante variabili string, utilizzando apici doppi (" ") o singoli (' ') (scelta consigliata da convenzione)

```
let color: string = "blue";
color = 'red';
```

Array

- Anche gli array possono essere definiti in due modi diversi, ma equivalenti:

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

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

Tuple

- Il tipo 'tuple' permette di specificare un array con un numero prefissato di elementi di tipo **noto**, **ma non ricorrente**:

```
let personTuple: [number, string, string, Date]
= [1, 'Pippo', 'Pluto', new Date("15/02/1963")];
```

- È possibile definire anche un array di tuple:

```
let personsTuple: [number, string, string, Date] []
= [
    [1, 'Mario', 'Rossi', new Date("01/01/1970")],
    [1, 'Sara', 'Bianchi', new Date("01/01/1973")]
];
```

Enum

- Come già diffuso in altri linguaggi (Java e C#), TS ha adottato il tipo Enum per creare costanti strutturate con nomi *user-friendly* da associare a valori numerici

- Per default ogni valore è un numero progressivo che parte da 0, anche se può essere sovrascritto

```
enum  CompassPoint {

  NORTH = 0, SOUTH, WEST, EAST

}


var est: CompassPoint = CompassPoint.EAST;
```

Any

- Serve quando non è possibile definire staticamente il tipo di una variabile, ad esempio quando il suo valore proviene da un sistema esterno o se cambia dinamicamente

- Se a una variabile non viene associato un tipo, sarà `any` per default

```
let anyVal: any = 100;
anyVal = "Ciao";
anyVal = false;
anyVal = {nome: 'Luigi', cognome: 'Brandolini'};
let list: any[] = [1, true, "free"];
```

- ATTENZIONE: la variabile può anche eseguire dei metodi, ma il compilatore non effettuerà alcun controllo statico sulla loro esistenza

```
let vAny: any = 5;
vAny.toUpperCase(); //ERROR
```

Unknown

- E' stato introdotto questo nuovo datatype a partire dalla versione 3.0 di TS

- Indica che una variabile avrà un valore *sconosciuto*

- Rappresenta la controparte di any con *typesafety*

```
let vAny: any = "ciao";
vAny.toUpperCase(); //OK

let vUnknown: unknown;
vUnknown = "benvenuto";

console.log(vUnknown.toUpperCase()); //ERROR: la proprietà non esiste nel
tipo unknown
```

Unknown

- E' utile per "forzare" l'utente ad effettuare un controllo sul tipo, prima di eseguire un metodo:

```
let vAny: any = "ciao";
vAny.toUpperCase(); //OK

let vUnknown: unknown;
vUnknown = "benvenuto";

if (typeof vUnknown === "string") {
    console.log(vUnknown.toUpperCase()); // OK
}
```

Void

- Può essere considerato come l'opposto di any e indica un valore di ritorno vuoto per le funzioni

- Lo si utilizza praticamente solo con le funzioni

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

- Non è utile utilizzarlo nella dichiarazione di variabile, per quanto sintatticamente corretto

Null & Undefined

- Sono tipi di dato propri in TS, ma poco utili di per sé

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

- Per default `null` and `undefined` sono anche subtypes di tutti gli altri tipo. Questo significa che è possibile assegnare `null` e `undefined` anche, ad esempio, ad un `number`

Never

- Si può applicare sia alle variabili, che alla funzioni
- Nelle variabili, indica che non potrà mai essere assegnato loro un valore
- L'unica variabile che può essere assegnata ad una variabile `never` è un'altra variabile `never`
- Nelle funzioni è ammesso solo se queste genereranno **certamente** un errore o se **non termineranno** la loro esecuzione

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Object

- Indica un tipo di dato generico, che possa essere ricondotto ad un object

```
let person: Object = {name: "Luca", surname: "Rossi"};
let age: Object = 38;
```

Union Operator

- In TS è frequente l'utilizzo dell'operatore union |
- Un classico caso d'uso è il seguente:

```
let pet: 'gatto' | 'cane';
pet = 'gatto';
pet = 'criceto'; //ERROR
```

Union Type

- E' possibile utilizzare lo stesso operatore per combinare più tipi diversi, ottenendo l'*union type*

ESEMPIO I:

```
function padding(padding: number | string) {
 console.log(`PADDING:  ${padding}`);
}

padding(5);
padding("5px");
```

ESEMPIO II:

```
function narrowingExample(x: string | number, y: string | boolean) {
    if (x === y) {
        x.toUpperCase();
        y.toLowerCase();
     } else {
        console.log(x);
        console.log(y);
     }
}
```

Union Type

- Spesso questo operatore viene utilizzato in combinazione con il JS `typeof`
- I controlli effettuati con questo operatore sono chiamati *type guard*
- Le azioni intraprese in base ai type guard verificati costituiscono il *narrowing* (restringimento)

```
function padLeft(padding: number | string, input: string) {
   if (typeof padding === "number") {
     return new Array(padding + 1).join(" ") + input;
   }
   return padding + input;
}
```

Template string

- TS offre anche un meccanismo dinamico di interpolazione delle stringhe, mediante *template strings*, che supporta *embedded expressions* su linee multiple

- Queste stringhe vengono delimitate dai caratteri backtick/backquote (`` ` ``)

- Le embedded expressions sono definite nella forma `${ expr }`

Template string

- Esempio:

```
let fullName: string = 'Luigi Brandolini';
let age: number = 36;
let sentence: string = `Hello, my name is ${fullName}.
I'll be ${age + 1} years old next month.`;

console.log(sentence);
```

Destructuring

- Come ES6, anche TS offre un meccanismo di *destructuring* che può essere applicato su array, tuple e oggetti

- Consiste in una sitassi abbreviata che permette di creare velocemente variabili

- **Array**:

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2

let [, second, , fourth] = [1, 2, 3, 4];
console.log(second); // outputs 2
console.log(fourth); // outputs 4
```

Destructuring

- Offre l'operatore '...' per associare la restante parte dell'array ad una nuova variabile:

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

Destructuring

- **Tuple**:

```
let tuple: [number, string, boolean] = [7, "hello", true];
let [a, b, c] = tuple; // a: number, b: string, c: boolean

let [a, b, c, d] = tuple; // Error: no element at index 3
```

```
let [a, ...bc] = tuple; // bc: [string, boolean]
let [a, b, c, ...d] = tuple; // d: [], the empty tuple
```

Destructuring

- **Object**:

```
let o = {
    fullname: "Anna Verdi",
    age: 30,
    city: "Roma"
  };
```

```
let { fullname, age } = o; // fullname = o.fullname and age = o.age
```

- Attenzione nel rispettare il nome delle properties (`fullname` e `age`, in questo caso)
- È anche possibile scegliere nuovi identificatori per le variabili con la seguente sintassi

```
let { fullname: newId1, age: newId2 } = o;
```

Object Oriented Syntax

L. Brandolini

Classi

- Le specifiche ECMAScript 6 (2015) hanno introdotto il costrutto sintattico `class` che tenta di colmare il gap tra la programmazione ad oggetti classica e quella **prototipale** di JavaScript

  NB:

  Non si tratta di un cambio del modello di programmazione, piuttosto di una semplificazione sintattica (*syntactic sugar*)

- Le classi, in ES6 e TS, sono concepite anche per offrire il meccanismo di ereditarietà (estensione) tra classi, così come anche overriding

- Il meccanismo di ereditarietà multipla, allo stesso modo di Java e C# non è supportato, ma in TS è presente un meccanismo alternativo chiamato `mixins`

Class in ECMAScript 6

```
class Person {

    constructor(name,surname) {

        this.name = name;

        this.surname = surname;

    }

}



var luigi = new Person("luigi","brandolini");
```

Class in ECMAScript 6 + TypeScript

```
class Person {

    name:string;

    surname:string;

    constructor(name:string, surname:string) {

        this.name = name;

        this.surname = surname;

    }

}


var luigi = new Person("Luigi","Brandolini");
```

Modifiers

- TypeScript offre i seguenti modificatori di acceso: `public`, `private`, `protected` e `readonly`

- Per **default** attributi e metodi sono `public`, ovvero accessibili anche esternamente alla classe

- Non possono essere utilizzati nelle interfacce, ad eccezione di `readonly`

- Aggiungendo la keyword `private` ad un campo/metodo, andiamo a rendere tale elemento non accessibile esternamente

- Aggiungendo la keyword `protected` ad un campo/metodo, tra le classi esterne, andiamo a rendere tale elemento accessibile solo a quelle derivate

Public modifier

```
class Person {
  public name: string;
  public surname: string;
  public constructor(name:string) {
    this.name = name;
  }
}

new Person("Mario","Rossi").name //OK
```

Private modifier

```
class Person {
  private name: string;
  private surname: string;
  public constructor(name:string) {
    this.name = name;
  }
}

new Person("Mario","Rossi").name //ERROR
```

Private modifier

- A partire da TypeScript 3.8 è anche possibile ricorrere, in alternativa, al simbolo # per dichiarare un campo "private":

```
class Person {
   #name: string;
   #surname: string;
   public constructor(name:string) {
     this.#name = name;
   }
}

new Person("Mario","Rossi").#name //ERROR
```

Protected modifier

```
class Person {
  protected name: string;
  protected surname: string;
  public constructor(name:string) {
    this.name = name;
  }
}

new Person("Mario","Rossi").name //ERROR
```

Protected modifier

```
class Employee extends Person {

    private department: string;

    constructor(name: string, department: string) {
        super(name);

        this.department = department;

    }

}


new Employee("Mario","Rossi","JS");
```

Readonly modifier

- Per concludere, è anche possibile definire un campo readonly
- Indica che, il campo a cui è assegnato, dovrà essere inizializzato contestualmente alla sua dichiarazione o attraverso il costruttore

```
class Employee {

    private name: string;

    readonly empCode: number;

    private department: string;

    constructor(name: string, empCode: number, department: string) {

        this.name = name;

        this.empCode = empCode;

        this.department = department;

    }

}
```

Abstract classes

- Sono classi base derivabili e che non possono essere instanziate direttamente.

- Diversamente dalle interfacce, una classe astratta può contenere dettagli implementativi per i suoi membri.

- La keyword abstract è usata per definire classi astratte come anche metodi astratti al suo interno.

- I metodi vanno implementati tramite override.

Abstract classes

```
abstract class Animal {

    private _name:string;

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

    abstract move():void;

}


new Animal("Cat"); //ERROR
```

Abstract classes

```
class Cat extends Animal {
    constructor(name:string) {
        super(name);
    }
    move():void {
        console.log("The cat is moving");
    }
}
let animal = new Cat("Lillo"); //OK
animal.move(); //"The cat is moving"
```

Interface

- È una componente che rappresenta una sorta di "contratto", ovvero un elenco di metodi astratti che devono essere presenti in una classe dello stesso tipo

- Può contenere, oltre alla definizione di metodi astratti, attributi

- Viene dichiarata con la keyword `interface`

- La classe che la implementa deve specificare la keyword `implements`

Interface

```
interface IAnimal {

    cry(): void;

}
```

- Potrà essere implementata sia da ***Animal*** (in questo caso il metodo sarà riportato come astratto), che da *Cat*

```
let animal = new Cat("Lillo"); //OK

animal.cry(); //"The cat is meowing"
```

Interface

- Diversamente da altri linguaggi che implementano il paradigma Object Oriented (es. Java), le interfacce possono contenere anche proprietà, oltre ai metodi

- E' possibile aggiungere proprietà ad un interfaccia già esistente, dopo averla definita

```
interface Person {
  name: string;
}

interface Person {
  surname: string;
}

let mario: Person = { name: "Mario", surname: "Rossi" };

console.log(JSON.stringify(mario));
```

Type Alias

- In TS è presente anche la keyword `type` tramite cui definire dei *type aliases*

- Un *type alias* può essere esteso (ereditarietà) mediante l'operatore '`&`'

- Può essere usato con un'accezione simile a quella di un'interfaccia, ma, a differenza di quest'ultima, non permette di aggiungere successivamente nuove proprietà

Type Alias

- Esempio:

```
type pet = 'cat' | 'dog';

let pet1: pet = 'cat';
let pet2: pet = 'dog';
```

Type Alias

- Esempio:

```
type Animal = {
     name: string;
 }

type Animal = {
     breed: string; //ERRORE: identificatore duplicato
  }

 type Cat = Animal & { //OK
       siamese: boolean
  }

let arcy: Cat = {name: "Arcy", siamese: true};
```

Type Compatibility

- TS adotta lo **structural typing** a differenza di linguaggi come Java o C#, che al contrario sono basati sul **nominal typing**

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

Mixins

- Rappresentano un modo per ovviare alla mancanza di supporto all'ereditarietà multipla

- Fanno leva su due funzionalità di TS:

  - Interface class extension
  - Declaration merging

Mixins

```
class Student {

    study(university: string) {
        console.log(`The student is studying at ${university}
University.`);
    }
}

class Employee {

    work(jobTitle: string) {
        console.log(`The employee works as ${jobTitle}.`);
    }
}

interface WorkingStudent extends Student, Employee {
}

class WorkingStudent {
}
```

Mixins

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            Object.defineProperty(derivedCtor.prototype, name,
Object.getOwnPropertyDescriptor(baseCtor.prototype, name));
        });
    });
}
```

```
applyMixins(WorkingStudent, [Student, Employee]);

let ws = new WorkingStudent();
ws.study("L'Aquila");
ws.work("software developer");
```

Overloading

- In TS l'overloading è supportato a livello di `function`

- E' possibile avere funzioni <u>con lo stesso nome, ma con parametri diversi in tipo o in numero</u>

- Esempio:

```
function add(a:string, b:string): string;

function add(a:number, b:number): number;

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

get / set

- È possibile definire delle funzioni allo scopo di esportare dei valori, accedendovi come se fossero attributi:

```
export class Employee {
   private _job: string;

    constructor(private fullname: string) {
    }

    getFullname() {
        return this.fullname;
    }

    get job() {
        return this._job;
    }

    set job(j: string) {
        this._job = j;
    }
}
```

get / set

- Test:

```
let mario: Employee = new Employee("Mario Bianchi");
mario.job = "software developer";

console.log(`${mario.getFullname()} is a ${mario.job}.`);
```

Arrow functions

- Rappresentano funzioni anonime, basate sulla definizione delle stesse attraverso una sintassi concisa

```
(param1, param2, ..., paramN) => expression
```

- L'operatore "=>" sostituisce la keyword "`function`"

- Se la funzione (anonima) prevede più di una riga di codice di programmazione, allora tutto il contenuto dev'essere necessariamente racchiuso all'interno di un blocco `{..}`

- Se la funzione (anonima) NON prevede più di una riga di codice di programmazione, allora non è indispensabile racchiudere tutto il codice all'interno di un blocco `{..}`

- I parametri restano sempre opzionali

Arrow functions

Esempi:

1.

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

2.

```
//ARROW FUNCTION
let sum = (a: number, b: number): number => {
    return a + b;
}
```

2.

```
let Hello = (): void => console.log("Hello World!");
```

Namespace

- È uno dei modi previsti dal TypeScript per organizzare il codice in unità separate e riutilizzabili (in Java, package).

- Un namespace è considerato un "modulo interno" (denominazione iniziale)

- Può contenere variabili, classi, interfacce, funzioni, ecc. Ciascuna di queste componenti dev'essere preceduta dalla parola chiave export per renderle accessibili esternamente

- È possibile definire gerarchie annidate di namespace (in Java, sub-packages).

Namespace

```
namespace Geometria {

    var pigreco = 3.14;

        export function circonferenza(raggio:number) {

        return 2 * pigreco * raggio;

    }

}


Utilizzo:

var geometria = Geometria.circonferenza(100);
```

Namespace

```
namespace Geometria {
    export namespace FigurePiane {

        export class Cerchio {...}
        export class Triangolo {...}
    }

    export namespace FigureSolide {

        export class Sfera {...}
        export class Piramide {...}
    }
}
```

Namespace

- Utilizzi:

```
let cerchio = new Geometria.FigurePiane.Cerchio();
```

- È possibile effettuare questa invocazione anche dall'esterno, includendo il file dove sarà presente la definizione del namespace

```
<script src="geometria.js"></script>
```

Moduli

- È il secondo metodo offerto da TypeScript per strutturare il codice su più unità separate e riusabili

- Principi di base:

  - Un modulo è contenuto in un file ed un file può contenere un solo modulo

  - Un modulo ha un proprio scope privato non accessibile dall'esterno

- Per esportare elementi all'esterno del modulo, ricorriamo alla keyword export

- Per importare elementi all'interno di un modulo, ricorriamo alla parola chiave import

Moduli: export

- Ogni dichiarazione (variabili, funzioni, classi, tipi, alias o interfacce) possono essere esportate mediante la keyword **export**

Validation.ts

```
export interface StringValidator {

    isAcceptable(s: string): boolean;

}
```

- *ZipCodeValidator.ts*

```
class ZipCodeValidator implements StringValidator {

    isAcceptable(s: string) {

        return s.length === 5 && numberRegexp.test(s);

    }

}

export { ZipCodeValidator };

export { ZipCodeValidator as mainValidator };
```

Moduli: import

- Le dichiarazioni appena viste possono essere importate mediante la keyword **import**

- **Import da file (standard)**

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

- **Import da file con alias**

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";

let myValidator = new ZCV();
```

- **Import da file in una variabile**

```
import * as validator from "./ZipCodeValidator";

let myValidator = new validator.ZipCodeValidator();
```

Decorators

- Un *Decorator* è un una dichiarazione particolare che può essere associata alle definizioni di:
  - Class
  - Properties
  - Accessors
  - Methods
  - Parameters

- Una dichiarazione può avere più presentare più decorators
- Viene applicato tramite un'espressione della forma `@Expression`
- Quest'ultima rappresenta una funzione che verrà eseguita a runtime con l'informazione riguardante la dichiarazione decorata
- Indipendentemente dal modo in cui viene applicato, l'obiettivo rimane sempre quello di modificare la definizione dell'oggetto target a cui si riferisce, a tempo di esecuzione
- Oltre al riferimento all'oggetto target, è possibile specificare anche ulteriori attributi via ***decorator factory***

Esecuzione

- Per eseguire via comando '`tsc`' codice contenente *decorators* è necessario specificare le seguenti opzioni:

```
tsc --target ES5 --experimentalDecorators
```

- Alternativamente, se nel progetto è presente il file `tsconfig.json`, aggiungere la seguente configurazione:

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

Per avviare la compilazione, sarà sufficiente eseguire il comando tsc senza argomenti

Esempio 1

- Per ottenere un decorator del tipo `@Log` *senza parametri*, creeremo prima una funzione con lo stesso identificatore:

```
function Log(target) {
..
}

@Log
class A {
..
}
```

Esempio 2

- Per ottenere un decorator del tipo `@Log` con personalizzazione, ad esempio prevedendo *parametri*, utilizzeremo una *Decorator Factory*

- D*ecorator factory* è una funzione che restituisce l'espressione invocata dal decorator a runtime, con, in questo caso, dei parametri iniziali

```
function Log(severity: string) {
..
  return function(target) {
  ..
  }
}

@Log("HIGH")
class A {
..
}
```

Credits

Dott. Ing. Luigi Brandolini

Software Engineer, IT Professional Instructor

Contacts:

E-Mail: luigi.brandolini@univaq.it