JESUÏTES educació

DAWM12
GP2 APPS HÍBRIDES EN ENTORNS MÒBILS

GP2 7.1 TYPESCRIPT

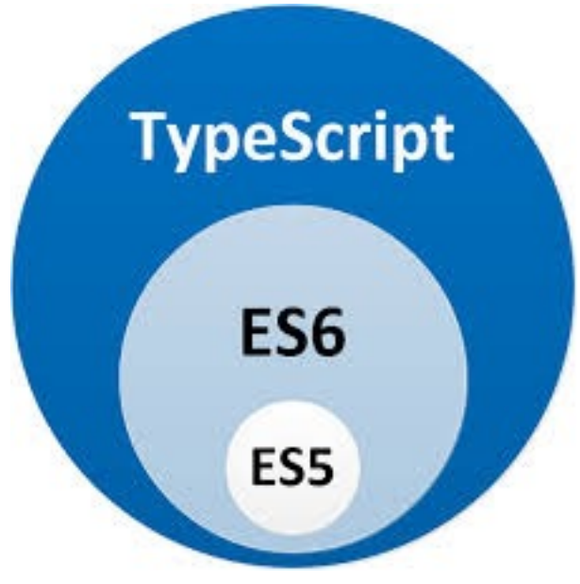
CFGS Desenvolupament d'Aplicacions Web
M12. Projecte Final - Global Project
Fundació Jesuïtes Educació - Escola del Clot
Sergi Grau sergi.grau@fje.edu

OBJECTIUS

- Aprendre les característiques del llenguatge TS.
- Conèixer que és ES6 i ES7 i la seva relació amb TS.
- Provar aplicacions amb el playground.
- Realitzar transpilacions amb TS.

- TypeScript és un llenguatge de programació lliure i de codi obert desenvolupat i mantingut per Microsoft.

- És un superconjunt de JavaScript, que essencialment afegeix tipatge estàtic i objectes basats en classes. Anders Hejlsberg, dissenyador de C # i creador de Delphi i Turbo Pascal, ha treballat en el desenvolupament de TypeScript.



- Typescript pot ser usat per a desenvolupar aplicacions JavaScript que s'executaran en el costat del client o del servidor (NODE.JS).

- TypeScript estén la sintaxi de JavaScript, per tant qualsevol codi JavaScript existent hauria de funcionar sense problemes.

- Typescript pot ser usat per a desenvolupar aplicacions JavaScript que s'executaran en el costat del client o del servidor (NODE.JS).

- TypeScript estén la sintaxi de JavaScript, per tant qualsevol codi JavaScript existent hauria de funcionar sense problemes.

- Està pensat per a grans projectes, els quals a través d'un compilador de TypeScript es tradueixen a codi JavaScript original.

- El compilador de TypeScript està escrit així mateix en TypeScript, compilat a JavaScript i amb Llicència Apache 2.

LET VS VAR

- Normalment utilitzarem la paraula clau let en lloc de la paraula clau var de JavaScript

- La paraula clau let és en realitat una nova construcció de JavaScript que funciona amb JS.

- Permet solucionar el problema d'accés a una variable definida en un bloc, que és accessible fóra del lloc on s'ha definit

EXEMPLE LET VS VAR

```
for(var i in quelcom) {
 // i es accessible

}
```

```
for(let i in quelcom) {
 // i es accessible
}

// i NO es accessible
```

INFERÈNCIA DE TIPUS

- La declaració de tipus en TS és opcional. Normalment TS pot determinar correctament el tipus de dada, encara que ni s'hagi declarat de manera explícita.

- Quan no és possible TS assigna ANY com a tipus de dada.

- És una millor pràctica fer-ho de manera explícita.

- Booleà

```
let fet: boolean = false;
```

- nombre, Igual que en JavaScript, tots els números són valors de punt flotant.

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

- Cadenes, amb cometes simples o dobles

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

- Expressions encastades, equivalents a les concatenacions amb operador +. Aquest format prové de ES6

```
let nom: string = `Bob Bobbington`;
let edat: number = 37;
let cadena: string = `Hola, sóc ${ nom }.
tindré ${ edat + 1 } el proper any.`
```

+ Les matrius han de contenir valors del mateix tipus a menys que el tipus de dada sigui any

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

+ Les tuples representen dades en forma [clau, valor]

```
let x: [string, number];
x = ["sergi", 10]; // OK
x = [sergi, "hello"]; // Error
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' no té
‘substr'
x[3] = "Joan"; // OK
console.log(x[5].toString()); // OK
x[6] = true; // Error, 'boolean' no és 'string | number'
```

- Una enumeració és un conjunt de valors únics que es poden accedir mitjançant un nom senzill de llegir

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
```

```
enum Color {Red = 1, Green, Blue};
let c: Color = Color.Green;

enum Color {Red = 1, Green = 2, Blue = 4};
let c: Color = Color.Green;

enum Color {Red = 1, Green, Blue};
let nomColor: string = Color[2];

alert(nomColor);
```

S’utilitza com a tipus que sigui compatible amb biblioteques diverses, fent que sigui compatible amb qualsevol tipus

```
let desconegut: any = 4;
desconegut = "sergi";
desconegut = false; // ok
```

```
let desconegut: any = 4;
desconegut.prova(); // ok
desconegut.toFixed(); // ok

let obj: Object = 4;
obj(); // Error: ‘toFixed'no existeix en ‘Object'.

let llista: any[] = [1, true, "free"];
llista[1] = 100;
```

VOID, NULL I UNDEFINED

- Void s'utilitza per a indicar l'absència de tipus, i normalment està associat a funcions que no retornen cap valor

```
function avisar(): void {
    alert("avís");
}

let noGaireUtil: void = undefined; // o null

let u: undefined = undefined;
let n: null = null;

//com son subclasses de la resta es poden assignar a
number, string, etc
```

+ representa un tipus de valor que mai ocorre.

```
function error(missatge: string): never {
    throw new Error(missatge);
}

// s'obté never per inferència
function fail() {
    return error("problema");
}

function infiniteLoop(): never {
    while (true) {
    }
}
```

ASSERCIONS

- les assercions són com els castings d'altres llenguatges. Es pot expressar de dues maneres.

```
let variable: any = "cadena";
let n: number = (<string>variable).length;
```

```
let variable: any = "cadena";
let n: number = (variable as string).length;
```

- Les funcions en TS no són molt diferents que en JS, excepte pel fet que es poden utilitzar tipus estàtics i que permeten indicar al compilador la signatura d'una funció en la seva crida.

- Les funcions poden ser anònimes com en JS

```
function saludar(nom:string):string{
  return ‘hola ‘+ nom;
}

var saludar = function(nom:string):string{
  return ‘hola ‘+ nom;
}
```

FUNCIONS

+ Utilitzar funcions anònimes convencionals tenen problemes per que la inferència no permet recordar el tipus de la funció.

+ Podem utilitzar lambdas

```
var saludar = function(nom:string):string{
  return ‘hola ‘+ nom;
}

var saludar: (nom:string)=> string =
function(nom:string):string {
  return‘hola ‘+ nom;
}
```

PARÀMETRES EN FUNCIONS

- Podem definir paràmetres opcionals, a la dreta dels paràmetres convencionals

```
function saludar(nom:string, salutacio?:string):string{
 if(!salutacio){
  salutacio='Hola';
 }
 return salutacio+','+nom;
}
```

- Podem definir valor per defecte per als paràmetres, a la dreta dels paràmetres convencionals

```
function saludar(nom:string,
salutacio:string='hola'):string{
 return salutacio+','+nom;
}
```

PARÀMETRES EN FUNCIONS

- Podem utilitzar paràmetres variadics

```
function saludar(salutacio:string, … noms:string[]):string
{
 return salutacio+’,’+noms.join(‘ i ‘);
}
```

- De la mateixa manera que JS no disposem de sobrecàrrega de funcions.

- De la mateixa manera que en Java, C#, Python i C++ disposem de lambas, en ES6 disposem de funcions “fletxa”.

- Permet una expressió més minimalista.

- Solucionen problemes de no accés a variables que es poden amb funcions anònimes en callbacks de JS.

```
var doble = x => x*2;
var sumar = (x,y) => x+y;
var reduir = [1,2,3,4].reduce((a,b)=>a+b,0);
var sumarIDoblar = (x,y) => {
  var suma=x+y;
  return suma*2;
}
```

```
function saludarEndarrerit(nom):void{
  this.nom=nom;
  this.salutacio=function(){
    setTimeout(function(){
      alert('hola'+this.nom);
    }, 0);
  }
}
var salutar= new saludarEndarrerit('sergi');
saludar.salutacio();// 'hola undefined'


function saludarEndarrerit(nom):void{
  this.nom=nom;
  this.salutacio=function(){
    setTimeout(() => alert('hola'+this.nom), 0);
  }
}
```

- Les classes són un dels elements claus de Angular2

- És una paraula reservada de JS, però en JS no s'utilitza POO basada en classes sinó dinàmic i basat en prototipus

ESTRUCTURA D'UNA CLASSE

DAWM12 GP2 APPS HÍBRIDES EN ENTORNS MÒBILS sergi.grau@fje.edu

```
class Cotxe {
    private distanciaRecorreguda: number = 0;
    color: string;

    constructor(public esHibrid: boolean, color: string = 'vermell') {
        this.color = color;
    }

    conduir(distancia: number): void {
        this.distanciaRecorreguda += distancia;
    }

    static sonarClaxon(): string {
        return 'meeec';
    }

    get distancia(): number {
        return this.distanciaRecorreguda;
    }
}
 let cotxe : Cotxe = new Cotxe(false);
console.log(cotxe.distancia);
```

CLASSES

- Les classes són un dels elements claus de Angular2
- És una paraula reservada de JS, però en JS no s'utilitza POO basada en classes sinó dinàmic i basat en prototipus.
- Disposem de variables membre (atributs), constructors (amb les mateixes característiques que les funcions), mètodes, mètodes i atributs de classe i accessors de propietat (amb get i set)

EXEMPLE D'INTERFICIES

```
interface IException{
  missatge:string;
  id?:number;
}

interface IExceptionArrayItem{
  [index:number]:IException;
}

interface IErrorHandler {
  exceptions:IExceptionArrayItem[];
  logExeption(missatge:string, id?:number):void;
}

class ErrorHandler implements IErrorHandler {
  exceptions: IExceptionArrayItem[];
  . . .
```

HERÈNCIA DE CLASSES

- Les classes són un dels elements claus de Angular2
- És una paraula reservada de JS, però en JS no s'utilitza POO basada en classes sinó dinàmic i basat en prototipus.
- Disposem de variables membre (atributs), constructors (amb les mateixes característiques que les funcions), mètodes, mètodes i atributs de classe i accessors de propietat (amb get i set)

HERÈNCIA DE CLASSES

- Les subclasses hereten de la superclasse amb la paraula clau extends, que permet heretar totes les operacions i atributs de la superclasse.

- Només es permet l'herència simple, no la múltiple.

- Les subclasses poden fer una reescriptura dels mètodes o invocar als mètodes de la superclasse amb super, per a ampliar el seu comportament.

```
class Familiar extends Cotxe{
 model:string;
 constructor(fer:string, model:string){
  super(fer);
  this.model=model;
 }
}
```

DECORADORS / DECORATORS

- Els decoradors són una eina molt potent de TS, que va crear Google amb AtScript al 2015 i que formen part de ES7.

- Ens permeten afegir metadades a les declaracions d'una classe per al seu us en injecció de dependència o en les directives de compilació.

- Utilitzen la notació @decorador, i es col·loquen abans del element a "decorar", incloent les crides a mètodes.

- Hi ha decoradors de classe, de propietat, de mètode i de paràmetre.

DECORADORS DE CLASSE

- Els decoradors de classe ens permeten ampliar una classe o realitzar operacions sobre els seus membres.

- El decorador es crida abans que la instanciació de la classe.

- Crear el nostre propi decorador és senzill. Cal crear una funció que serà cridat pel constructor de la classe que “decora”.

```
declare type Decorador = <TFunction extends
Function>(Target:TFunction) => Function | void;
```

DECORADORS DE CLASSE

```
function saludador(target:Function):void{
    target.prototype.salutacio = function(): void{
        console.log('hola');
    }
}
```

```
@saludador
class Saluda {
    constructor() {

    }
}

var laMevaSalutacio = new Saluda();
laMevaSalutacio.salutacio();
```

PERSONALITZACIÓ DE DECORADORS

- Podem personalitzar els nostres decoradors. Ho fem sobrecarregant la funció que defineix el decorador.

```
function saludador(missatge:string){

 return function (target:Function){
    target.prototype.salutacio = function(): void{
        console.log(missatge);
    }
  }
}

@saludador('sergi')
class Saluda {
    constructor() {

    }
}
```

- S’apliquen a propietats i atributs. La signatura de la funció decoradora té el target (el prototipus de la classe que es vol decorar) i key (el nom de la propietat a decorar)

- Es pot utilitzar per a fer logging, o controlar quan canvia una les dades d’una propietat.

DECORADORS DE PROPIETAT

DAWM12 GP2 APPS HÍBRIDES EN ENTORNS MÒBILS sergi.grau@fje.edu

```
function LogCanvis(target: Object, key: string) {
    var valorPropietat: string = this[key];
    if (delete this[key]) {
        Object.defineProperty(target, key, {
            get: function () {
                return valorPropietat;
            }
            set: function (nouValor) {
                valorPropietat = nouValor;
                console.log(`${key} ara val ${valorPropietat}`);
            }
        });
    }
}

class Fruita{
    @LogCanvis
    nom: string;
}

var fruita = new Fruita();
fruita.nom = 'poma';
fruita.nom = 'pera';
```

PERSONALITZACIO DE DECORADORS DE PROPIETAT


```
function LogCanvis(callback: any): Function {
    return function (target: Object, key: string): void {
        var valorPropietat: string = this[key];
        if (delete this[key]) {
            Object.defineProperty(target, key, {
                get: function () {
                    return valorPropietat;
                },
                set: function (nouValor) {
                    valorPropietat = nouValor;
                    callback.onchange.call(this.valorPropietat);
                }
            });
        }
    }
}

class Fruita{
    @LogCanvis({
        onchange: function (nouValor: string): void {
            console.log(`la fruita ara es ${nouValor}`);
        }
    });
    nom: string;
}

var fruita = new Fruita();
fruita.nom = 'poma';
fruita.nom = 'pera';
```

- Aquests decoradors poden detectar, anotar i intervenir quan un mètode és executat.

- Només cal definir una funció MethodDecorator amb els paràmetres target (un objecte que representa el mètode a decorar), key (cadena que té el métode real a decorar) i value (que és una propietat, amb un apuntador al mètode)

DECORADORS DE MÈTODE


```
function Log(target: Function, key: string, descriptor: any) {
    var metodeOriginal = descriptor.value;
    var nouMetode = function (...args: any[]):any {
        var resultat: any = metodeOriginal.apply(this, args);
        if (!this.sortidaLog) {
            this.sortidaLog = new Array<any>();
        }
        this.sortidaLog.push({
            method: key,
            parameters: args,
            output: resultat,
            timestamp:new Date()
        })
        return resultat;

    }

    descriptor.value = nouMetode;
}
```

DECORADORS DE MÈTODE

```
class Calculador{
    @Log
    doblar(num: number): number{
        return num * 2;
    }
}

var calc = new Calculador();
calc.doblar(2);
console.log(calc.sortidaLog);
```

DECORADORS DE PARÀMETRE

- Estan associats a la funció ParameterDecorator.
- No estan destinats a canviar els paràmetres en la crida, sino en mirar i verificar els seus valors.
- La funció té els arguments (target: objecte prototipus on la funció decora als paràmetres), key (nom de la funció que conté els paràmetres) i parameterIndex (index en el array de paràmetres on el decorador s'aplica.

```
function Log(target: Function, key: string, parameterIndex: number) {
    var funcioLogada = key || target.prototype.constructor.name;
    console.log(`
    el paràmetre en posicio {parameterIndex} en la
    funcio {funcioLogada} ha estat decorat`);

}

class Salutacio {
    salut: string;
    constructor( @Log frase: string) {
        this.salut = frase;
    }
}
```

- L'organització d'aplicacions amb TS en fa mitjançant mòduls.
- Els mòduls poden ser interns o externs.

MÒDULS INTERNS

- Els mòduls interns són embolcalls “singleton” contenint un conjunt de classes, funcions, objectes o variables d’àmbit intern.

- Podem exposar els continguts d’un mòdul amb la paraula clau export. Per exemple aquestes dues classes són accessibles des del exterior.

```
module Salutacions {
export class Salutacio{
  . . .
}

export class SalutacioOficial {
  . . .
}
```

+ Per importar un mòdul es fa servir la paraula clau import

```
import SalutacioOficial = Salutacions.SalutacioOficial;
```

+ Per indicar on es troba un mòdul es fa servir

```
/// <reference path='salutacions/SalutacioOficial.ts'
```

MÒDULS EXTERNS

- Acostumen a ser una millor solució per al creixement de projectes.

- Cada fitxer és un mòdul, no cal posar la paraula module, només desar-ho en un fitxer amb el nom del mòdul.

- S'utilitza la paraula require que no está suportada en ES5

```
import salutacions = require('Salutacions');
var SalutacioOficial= salutacions.SalutacioOficial();
var salutacioOficial = new SalutacioOficial();
```

GENÈRIQUES

- Una part important de l'enginyeria de programari són els components de construcció que no només tenen APIs ben definides i coherents, sinó que també són reutilitzables. Això donarà capacitats més flexibles per a la construcció de grans sistemes de programari.

- En llenguatges com C # i Java, una de les principals eines per crear components reutilitzables són les classes parametritzades o genèriques, és a dir, ser capaç de crear un component que pot treballar sobre una varietat de tipus en lloc d'una sola.

- Això permet als usuaris que consumeixen aquests components i utilitzar els seus propis tipus.

```
function indentitat(arg: number): number {
    return arg;
}
// també podriem tenir…
function indentitat(arg: any): any {
    return arg;
}

// en format de funció genèrica

function indentitat<T>(arg: T): T {
    return arg;
}

let output = indentitat<string>("sergi");  // el tipus de output és
string

i per inferència
let output = indentitat("sergi");  // el tipus de output és string
```

```
class NombreGeneric<T> {
    valorZero: T;
    afegir: (x: T, y: T) => T;
}

let gn = new NombreGeneric<number>();
gn.valorZero = 0;
gn.afegir = function(x, y) { return x + y; };

let gs = new GenericNumber<string>();
gs.valorZero = "";
gs.afegir = function(x, y) { return x + y; };

alert(gs.afegir(gn.valorZero, "test"));
```

+ Juntament amb les jerarquies tradicionals OO, una altra forma popular de la construcció de les classes és a partir de components reutilitzables que es poden construir mitjançant la combinació de les classes parcials simples.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }
}

// Activable Mixin
class Activable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}
```

```
class Classe implements Disposable, Activable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " +
this.isDisposed), 500);
    }

    interact() {
        this.activate();
    }

    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(Classe, [Disposable, Activable]);

let objecte = new Classe();
setTimeout(() => objecte.interact(), 1000);
```

TIPUS D'INTERSECCIÓ

+ Un tipus d'intersecció combina diversos tipus en un de sol. Això permet afegir tipus existents en conjunt per obtenir un únic tipus que té totes les característiques que es necessiten.

TIPUS D'INTERSECCIÓ

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Persona {
    constructor(public nom: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}
var sergi = extend(new Persona("Sergi"), new ConsoleLogger());
var n = sergi.nom;
sergi.log();
```

TIPUS D'UNIÓ

+ Els tipus avançats d'unió estan estretament relacionats amb la intersecció, però s'utilitzen de manera molt diferent. De tant en tant, ens cal passar un paràmetre que pot ser un nombre o una cadena.

```
interface Ocell {
    volar();
    posarOus();
}

interface Peix {
    nedar();
    posarOus();
}

function obtenirMascota(): Peix | Ocell {
    // ...
}

let mascota = obtenirMascota();
mascota.posarOus(); // ok
mascota.nedar();    // errors
```

+ https://www.typescriptlang.org/

+ Learning Angular 2, Pablo Deeleman, ed. Packt