TypeScript

Summary

- Problematiche di Javascript
- Introduzione a Typescript
  - Motivazioni
  - Funzionalità
  - Compilazione
  - Integrazione con Javascript

Problematiche di Javascript

- Nessuna compilazione
- Tipizzazione "dinamica"
- Variabili globali
- Abstract Equality Comparison Algorithm
- Scope Inconsistencies
- Prototype based
- Difficile organizzazione architetturale
- …

Ma ha anche aspetti positivi

- Setup immediato
- Ecosistema enorme

- Ma soprattutto…
- ***È l'unico modo per fare computazione lato client!***

Perché Javascript

- I browser sono stati progettati per eseguire solo Javascript.
- Nel tempo Javascript è diventato lo standard indiscusso per la programmazione basata su browser.
- I browser sono dotati di un motore runtime Javascript, basato su ECMA Script.
- Non si è mai arrivati (e non arriveremo mai?) al punto in cui si potrebbe lasciar perdere Javascript per passare a qualcos'altro, migliore o meglio progettato.
- Nel tempo si è affermato anche nello sviluppo di app per device mobili e per computazione lato server.

Idea ...

- Per superare difetti e limitazioni di Javascript, si sono diffusi molti linguaggi di programmazione alternativi che ***compilano*** in Javascript.
- Vision: *JavaScript is assembly language for the web*
- Esempi:
  - CoffeeScript
  - **TypeScript**
  - ClojureScript
  - Haxe
  - Scala
  - Dart (di Google)
  - …

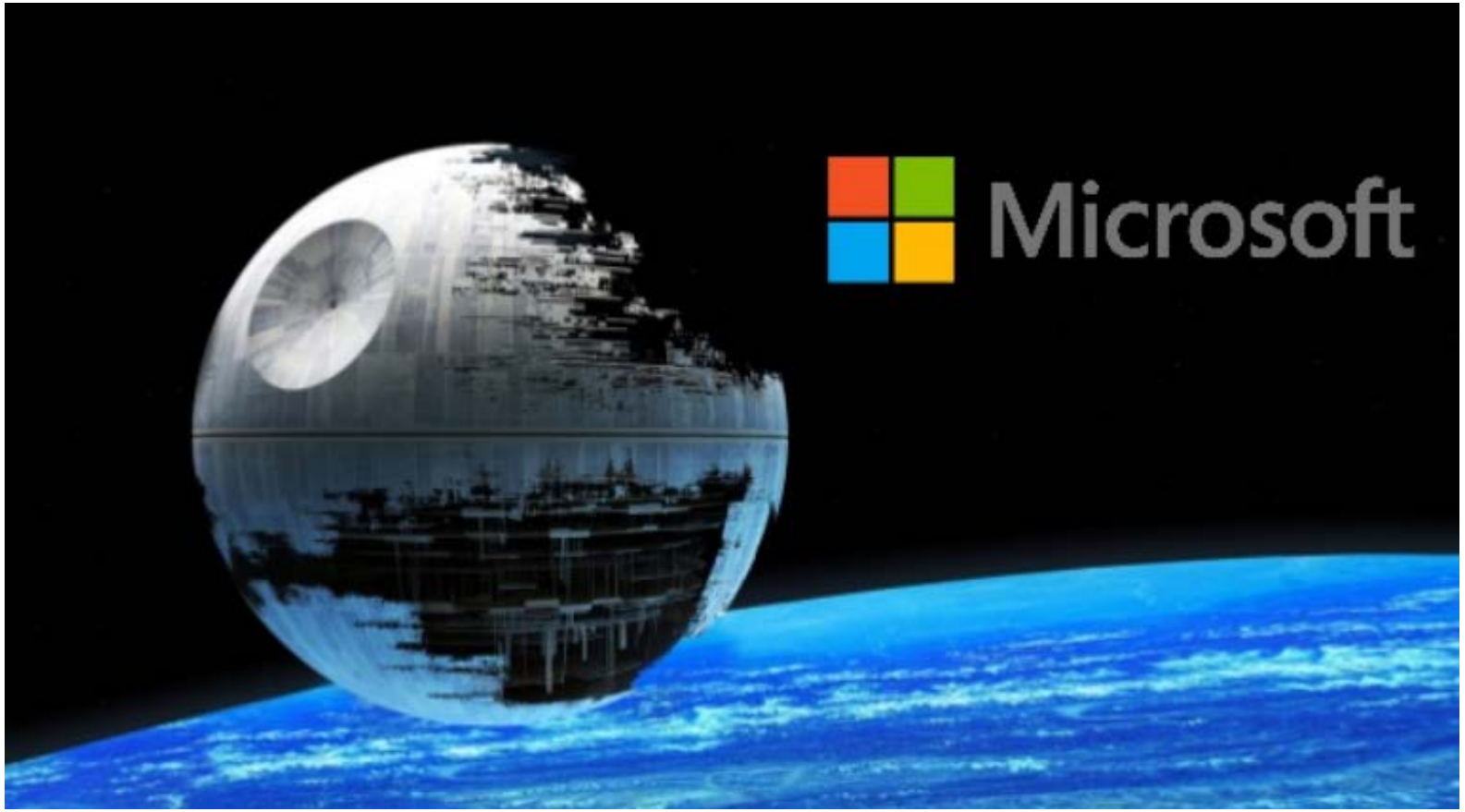

TypeScript
JavaScript that scales.

TypeScript

- *"The goal was to strengthen JavaScript with things like classes, modules, and static typing, without sacrificing the advantage of its being open-standards and cross-platform; the result was a language for application scale JavaScript development, built as a super set of the language“*

Anders Hejlsberg, co-fondatore di TypeScript
Microsoft

TypeScript



TypeScript

- «TypeScrit is a typed ***superset*** of JavaScript that compiles to plain JavaScript.»
- «Any browser. Any host. Any OS. **Open source.**»
- https://github.com/Microsoft/TypeScript

Linguaggi Super-Set

- Lo scopo principale dei linguaggi nati come super-set di altri linguaggi è aggiungere feature e migliorare la potenza espressiva
- Questo è possibile grazie alla compilazione di codice sorgente, scritto nel linguaggio super-set (che ammette queste feature) in altro codice sorgente scritto nel linguaggio di partenza
- Compilazione ***source-to-source*** (chiamata ***transpiling***), per distinguerla dalla compilazione classica ***source-to-binary***
- Il linguaggio super-set può avere caratteristiche semantiche o sintattiche diverse dal linguaggio originale: il ***transpiler*** (il compilatore che esegue il ***transpiling***) avrà il duplice compito di controllo degli errori e della generazione del nuovo codice sorgente

Funzionalità

- Tipizzazione statica
- Funzioni
- Classi
- Interfacce
- Moduli

Q&A

- Ci sono domande?

Funzionalità

- Tipizzazione statica
- Funzioni
- Classi
- Interfacce
- Moduli

Annotazione del tipo opzionale

- TypeScript consente di definire il tipo di una variabile. Esempio:
```
var age: number = 40;
```
- È una funzionalità puramente a design time. Nessun codice aggiuntivo è inserito nel codice Javascript prodotto dal compilatore. Esempio:
```
var age = 40;
```
- Vantaggi?

Annotazione del tipo opzionale

- TypeScript consente di definire il tipo di una variabile. Esempio:
  ```
  var age: number = 40;
  ```
- È una funzionalità puramente a design time. Nessun codice aggiuntivo è inserito nel codice Javascript prodotto dal compilatore. Esempio:
  ```
  var age = 40;
  ```
- Vantaggi?
  ```
  var age:number = 'forty';
  ```
  Error: Cannot convert 'string' to 'number'

Static Types

- Primitive
- Array
- Enum
- Any

Primitive Types

- Boolean

```
var exists:boolean = true // or false
```

- Number
  - Non c'è separazione tra integer e float/double
  - Tutti i numeri sono valori floating point

```
var x:number = 55
var y:number = 123.4567
```

- Attenzione! Quanto fa 0.1 + 0.2?

Primitive Types

- Boolean
```
var exists:boolean = true // or false
```
- Number
  - Non c'è separazione tra integer e float/double
  - Tutti i numeri sono valori floating point
```
var x:number = 55
var y:number = 123.4567
```
- Attenzione! Quanto fa 0.1 + 0.2?

```
console.log(0.1 + 0.2);
//output
//0.30000000000000004

console.log((0.1 + 0.2) ==.3);
//output
//false
```

Primitive Types (2)

- String
  - Possono essere usate indifferentemente singoli o doppi apici
  - Non esiste un tipo char

```
var msg1 = 'hello world'
var msg2 = "hello world"
```

- Void: Usato principalmente per indicare che una funzione che non restituisce valori
- Null
- Undefined

Array

```
var cities: string[] =
['Cesena','Bologna','Ravenna'];

var primes: number[] = [2,3,5,7,11,13,17,23];

var bools: booleans[] = [true, true, false,
true];
```

Enum

- Simili all'Enum di C#.
- Modo per definire un insieme di elementi.
- Esempio:

```
enum Color{Red, Green, Blue}
var b:Color = Color.Red;
```

- 0-indexed ma è possibile specificare l'indice per ogni elemento (ad esempio indicando '=1'):

```
enum Color{Red = 1, Green, Blue}
var b:Color = Color.Red;
```

Any

- Utile per descrivere il tipo di variabili che potremmo non sapere quando scriviamo l'applicazione.
- Esempi:
  - Input dell'utente
  - Libreria di terze parti
- Consente di rinunciare al type-checking.
- ```
  var notSure: any;
  notSure = 2;
  notSure = 'hello';
  ```
- ```
  var list: any[] = [1, true, 'hi'];
  list[2] = 42;
  ```

Type Inference

- TypeScript cerca di inferire il tipo di una variabile.

```
var x = 175
```

- Quale tipo?

Type Inference

- TypeScript cerca di inferire il tipo di una variabile.

```
var x = 175
```

- Quale tipo? **Number**

```
x = ‘hello’;
```

- Cosa succede?

Type Inference

- TypeScript cerca di inferire il tipo di una variabile.

```
var x = 175
```

- Quale tipo? **Number**

```
x = 'hello';
```

- Cosa succede?

Cannot convert 'string' to 'number'

Type Inference (2)

- Ci sono quattro modi per dichiarare una variabile

1. Dichiarare il suo tipo e il suo valore in un'unica istruzione
```
var message1:string = 'hello world';
```
2. Dichiarare il suo tipo ma non il valore, che sarà settato a undefined
```
var message2:string;
message2 = 'hello world';
```
3. Dichiarare il valore ma non il tipo. Il tipo potrebbe essere inferito in base al valore
```
var message3 = 'hello world';
```
4. Non dichiarare tipo e valore. La variabile sarà di tipo Any e il suo valore sarà undefined
```
var message4;
```

Q&A

- Ci sono domande?

Funzionalità

- Tipizzazione statica
- Funzioni
- Classi
- Interfacce
- Moduli

Overview Funzioni Javascript

- Le funzioni sono una componente fondamentale di qualsiasi applicazione in Javascript.
- Javascript supporta le First-class function.
- Typescript aggiunge alcune nuove funzionalità alle funzioni standard di Javascript:
  - Tipi per parametri e tipo di ritorno
  - Parametri opzionali e di default
  - Function overload

Tipi per parametri e tipo di ritorno

```
function add(x:number, y:number):number{
      return x+y;
}
add(2,3);
add(2,'Hello');
add('hello','world');
return 'hello';
```

Tipi per parametri e tipo di ritorno

```
function add(x:number, y:number):number{
      return x+y;
}
add(2,3); //ok
add(2,'Hello'); //errore
add('hello','world'); //errore
return 'hello'; //errore
```

Parametri opzionali e di default

- I parametri opzionali dovrebbero avere un valore di default da usare se il valore non è specificato nell'invocazione della funzione.

```
function display(msg:string,
user:string = 'du'){
    alert(msg+' '+user);
}
display('hello');
display('hello','world');
```

Parametri opzionali e di default

- I parametri opzionali dovrebbero avere un valore di default da usare se il valore non è specificato nell'invocazione della funzione.

```
function display(msg:string,
user:string = 'du'){
    alert(msg+' '+user);
}
display('hello'); //hello du
display('hello','world');  //hello world
```

Function overload

```
function add(var1:string, var2:string):string;
function add(var1:number, var2:number):number;

function add(var1, var2):any{
    if (typeof var1 == "number" && typeof var2 == "number"){
        return var1 + var2;
    }
    if (typeof var1 == "string" && typeof var2 == "string"){
        return var1 + ' ' + var2;
    }
}

var r1 = add(10,20);
var r2 = add('hello','world');

alert(r1); // 30
alert(r2); // hello world
```

Q&A

- Ci sono domande?

Funzionalità

- Tipizzazione statica
- Funzioni
- Classi
- Interfacce
- Moduli

Field e Property

```
class Employee{
    name:string // public field
    private _hiddenField:string // private field

    private _salary:number // private backing field

    // public property
    get Salary():number{
        return this._salary
    }
    set Salary(value:number){
        if (value <= 0){
            throw "salary can not be less than 0"
        }
        else{
            this._salary = value
        }
    }
}
```

Field e Property (2)

```
var emp: Employee = new Employee()
try{
    emp.Salary = -100
}
catch(error){
    alert(error)
}
emp.Salary = 50
alert(emp.Salary)
```

Costruttore

- Definibile usando la keyword **`constructor`**
- **`public`** di default, non può essere **`private`**

```
class Employee{
    private name:string
    private basic:number
    private allowance:number

    constructor(name:string, basic:number, allowance:number){
        this.name = name
        this.basic = basic
        this.allowance = allowance
    }
```

public and private

```
1  class Employee{
2      public public_method(){
3          alert('public_method called')
4      }
5      private private_method(){
6          alert('private_method called')
7      }
8  }
9
10 var emp: Employee = new Employee()
```

Metodi Statici

```
class Employee{
    instanceMethod(){
        alert('Employee.instanceMethod called')
    }

    static staticMethod(){
        alert('Employee.staticMethod called')
    }
}

new Employee().instanceMethod();
Employee.staticMethod();
```

Ereditarietà

- TypeScript supporta l'ereditarietà delle classi attraverso la keyword **`extends`**.

```
1  class Person{
2      constructor(public name:string, public age:number){
3      }
4      showInfo(){
5          alert("Name: " + this.name + " Age: " + this.age);
6      }
7  } // Base class
8
9  class Employee extends Person{
10     constructor(name, age, public salary:number){
11         super(name,age); // super calls the constructor of base class
12     }
13     showInfo(){
14         alert("Name:" + this.name + " Age:" + this.age + " Salary:" + this.salary);
15     }
16 } // Class that inhertits from Base class
17
18 var per:Person = new Person('Aniruddha',40);
19 per.showInfo(); // calls showInfo of Person class
20
21 var emp:Person = new Employee('Bill',55,100);
22 emp.showInfo(); // calls showInfo of Employee class
```

Q&A

- Ci sono domande?

Funzionalità

- Tipizzazione statica
- Funzioni
- Classi
- Interfacce
- Moduli

Interfacce

- Le interfacce sono definite con la keyword **`interface`**.
- Come altre funzionalità, sono considerate solo a *Design-time*. Se compilate, producono un file vuoto.

```
1 interface Employee{
2     FirstName:string;
3     LastName:string
4 }
5
6 var emp:Employee = {FirstName:"Bill",LastName:"Gates"};
7 alert(emp.FirstName);
8
```

Proprietà Opzionali

- Proprietà opzionali possono essere dichiarate per un'interfaccia, usando ?

```
interface Employee{
    FirstName:string;
    LastName?:string // LastName is declared as optional property using ?
    Age:number
}

// ShowEmployeeDetails expects an Employee object with FirstName, LastName and Age property
function ShowEmployeeDetails(emp:Employee){
    alert('hello ' + emp.FirstName + ' ' + emp.LastName + '. Your age is ' + emp.Age);
}

var emp1:Employee = {FirstName:'Bill',LastName:'Gates',Age:50};
var emp2 = 'is this an employee?'
var emp3 = {FirstName:'Bill',Age:50};

ShowEmployeeDetails(emp1); // Works as expected
ShowEmployeeDetails(emp2); // not an Employee object, shows undefined
ShowEmployeeDetails(emp3); // Since LastName is now optional no error is shown
```

Funzionalità

- Tipizzazione statica
- Funzioni
- Classi
- Interfacce
- Moduli

Moduli

- I moduli sono definiti con la keyword **`module`**.
- Possono contenere:
  - Moduli
  - Classi
  - Interfacce
  - Enum
- Non possono contenere funzioni
- Classi e interfacce possono essere esposte usando la keyword **`export`**.

Moduli (2)

```
module Utils{
    export class Math{
        public add(x:number, y:number): number{
            return x+y
        }
    }
    // not accessible out side the module
    class Helper{
        help(){}
    }
}

// can not access the class without referencing Module
var m = new Utils.Math
m.add(20,30)
```

Q&A

- Ci sono domande?

Perché TypeScript

- A livello sintattico, TypeScript non è particolarmente innovativo.
- Uno dei vantaggi principali è la combinazione di:
  - ***Intellisense***: sistema di suggerimenti e auto completamento che l'IDE fornisce allo sviluppatore durante la scrittura del codice.
  - ***Type-checking statico***: un linguaggio si dice staticamente tipato quando il tipo di una variabile è conosciuto a tempo di compilazione.
  - ***Compilazione***

Compilazione

- TypeScript dispone di un compilatore che produce codice JavaScript a partire da codice TypeScript.
- Tipi di compilazione:
  - Compilazione 1 a 1 di file (da *.ts* a *.js*)
  - Compilare tutti i file internamente a una cartella
  - Compilare un file sorgente e tutti i file referenziati da esso

tsc e tsconfig

- La compilazione è svolta dall'eseguibile *tsc* (*TypeScript Compiler*) e può avvenire manualmente o tramite IDE.
- La fase di compilazione è configurata da:
  - file *tsconfig.json*
  - argomenti in ingresso all'eseguibile *tsc*
- Documentazione completa *tsconfig* https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Esempio – Codice TypeScript

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

let greeter = new Greeter("world");

let button =
document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

Esempio – Codice TypeScript

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

let greeter = new Greeter("world");

let button =
document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

```
var Greeter = /** @class */
(function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet =
function () {
        return "Hello, " +
this.greeting;
    };
    return Greeter;
}());

var greeter = new Greeter("world");
var button =
document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};

document.body.appendChild(button);
```

Integrazione con Js

- È possibile utilizzare JavaScript puro in TypeScript?
  - Sì, il compilatore TypeScript rileva il codice JavaScript presente nei sorgenti e lo riporta tale e quale nel file JavaScript compilato.
- È possibile mantenere i vantaggi di TypeScript anche utilizzando librerie JavaScript pure?
  - Dipende!

Typings

- ***Typings*** (https://github.com/typings/typings) è un manager che gestisce le definizioni di TypeScript.
- È nato proprio per risolvere il problema precedente.
- Typings mantiene un archivio (più d'uno ...) di definizioni.

Typings

- Gli archivi di definizioni sono mantenuti dalla comunità (è un progetto GitHub), quindi è possibile trovare:
  - definizioni assenti
  - definizioni errate (che è peggio)
  - definizioni non allineate con la latest build della libreria originale (occhio)
  - più definizioni in archivi diversi (quale usare?)

Typings

- In generale, per le librerie/framework principali (es: JQuery) le definizioni sono corrette.
- Per librerie di nicchia o con poco seguito su GitHub, è possibile avere problemi.
- Nel caso in cui non fossero disponibili sorgenti TypeScript nativi, e Typings non fosse d'aiuto, come fare?
  - Scrivere le definizioni manualmente (!!)
  - Utilizzare ***any***

Installazione

- È possibile installare TypeScript e Typings utilizzando npm
- **`npm install -g typescript`**
- **`npm install -g typings`**

Q&A

- Ci sono domande?

Riferimenti

- Documentazione ufficiale TypeScript (https://www.typescriptlang.org/)
- Playground: https://www.typescriptlang.org/play/index.html
- Aniruddha Chakrabarti. TypeScript (2015). https://www.slideshare.net/aniruddha.chakrabarti/typescript-44668095
- Giacomo Dradi. Progettazione e sviluppo di applicazioni ibride: un case study basato su Single Page Application (2016). [Laurea magistrale], Università di Bologna, Ingegneria e scienze informatiche