

Umberto Emanuele

TYPESCRIPT

Gennaio 2023



TypeScript è un linguaggio di programmazione open source sviluppato da Microsoft. Si tratta di un **Super-set** di **JavaScript** che basa le sue caratteristiche su **ECMAScript 6**.

Il linguaggio estende la sintassi di JavaScript in modo che qualunque programma scritto in JavaScript sia anche in grado di funzionare con TypeScript senza nessuna modifica. È stato progettato per lo sviluppo di grandi applicazioni ed è destinato a essere compilato in JavaScript per poter essere interpretato da qualunque web browser o app.

TypeScript nasce dal crescente bisogno di un linguaggio **front-end** per lo sviluppo di applicazioni JavaScript su larga scala e dalla necessità di **sicurezza e robustezza**, sia da parte di sviluppatori interni a Microsoft sia da parte di clienti e sviluppatori indipendenti.

TypeScript non può essere aperto dai browser senza essere prima compilato e trasformato in javascript.

PRO:

- TypeScript permette con la sua sintassi di scrivere codice più pulito e meno soggetto ad errori
- TypeScript converte automaticamente il codice in JavaScript ottimizzato, con le best practises e cosa più importante permette di specificare i DATA TYPES
- Il tuo editor ora può controllare il tuo codice molto più a fondo, evidenziando incongruenze, riassegnazioni strane e bugs

CONTRO:

- È necessario un po' di tempo per abituarsi alla nuova sintassi (in ogni caso ben speso)
- TypeScript non è ancora supportato dai browser o dalle LTS di Node.js, ha ancora bisogno di essere traspilato in JavaScript per la sua esecuzione (a questo ci pensa il compilatore integrato chiamato tsc)

- <https://www.typescriptlang.org/>

Get TypeScript

Node.js

The command-line TypeScript compiler can be installed as a Node.js package.

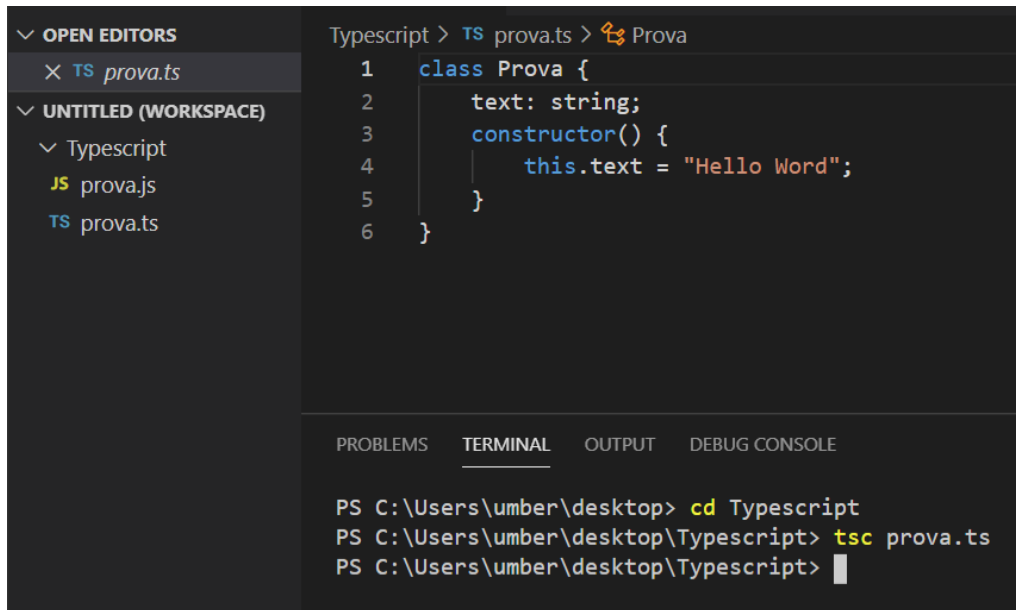
INSTALL

```
npm install -g typescript
```

COMPILE

```
tsc helloworld.ts
```

Come transpilare codice Typescript in Javascript



The screenshot shows a VS Code editor interface. On the left, the Explorer sidebar shows a workspace with a folder named 'Typescript' containing two files: 'prova.js' (JavaScript) and 'prova.ts' (TypeScript). The main editor area displays the content of 'prova.ts', which defines a class 'Prova' with a 'text' property and a 'constructor' that sets 'this.text' to 'Hello Word'. Below the editor, the TERMINAL panel is active, showing the command prompt with the following commands and output:

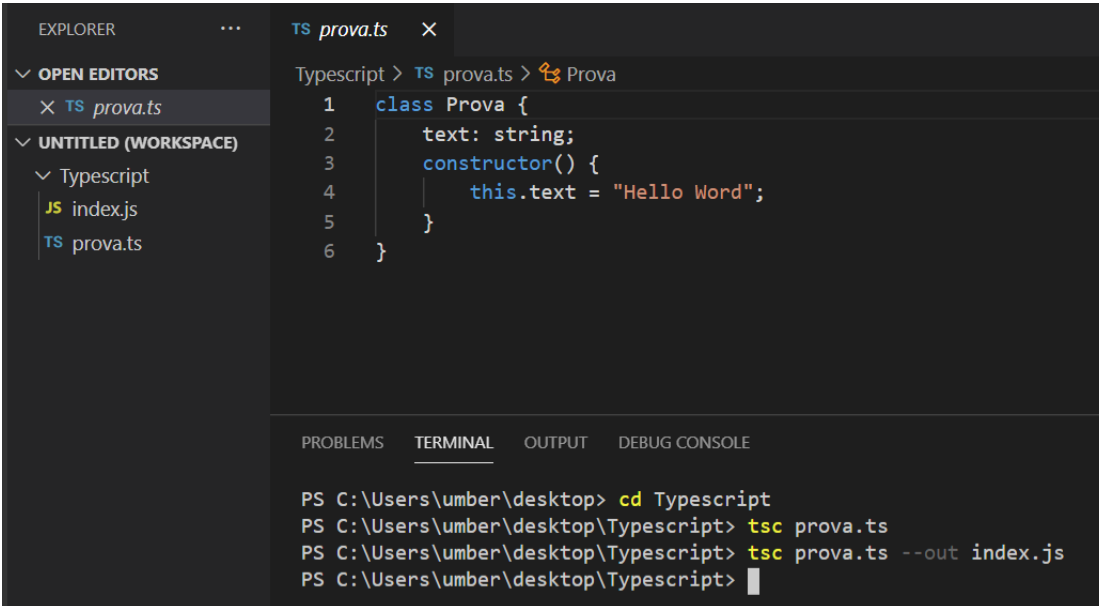
```
Typescript > TS prova.ts
1 class Prova {
2     text: string;
3     constructor() {
4         this.text = "Hello Word";
5     }
6 }
```

```
PS C:\Users\umber\desktop> cd Typescript
PS C:\Users\umber\desktop\Typescript> tsc prova.ts
PS C:\Users\umber\desktop\Typescript>
```

Come transpilare codice
TypeScript in un file
Javascript di nome diverso

`--out`

`--outFile`



The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows a workspace with two files: `index.js` (JavaScript) and `prova.ts` (TypeScript). The main editor displays the content of `prova.ts`, which contains a class definition:

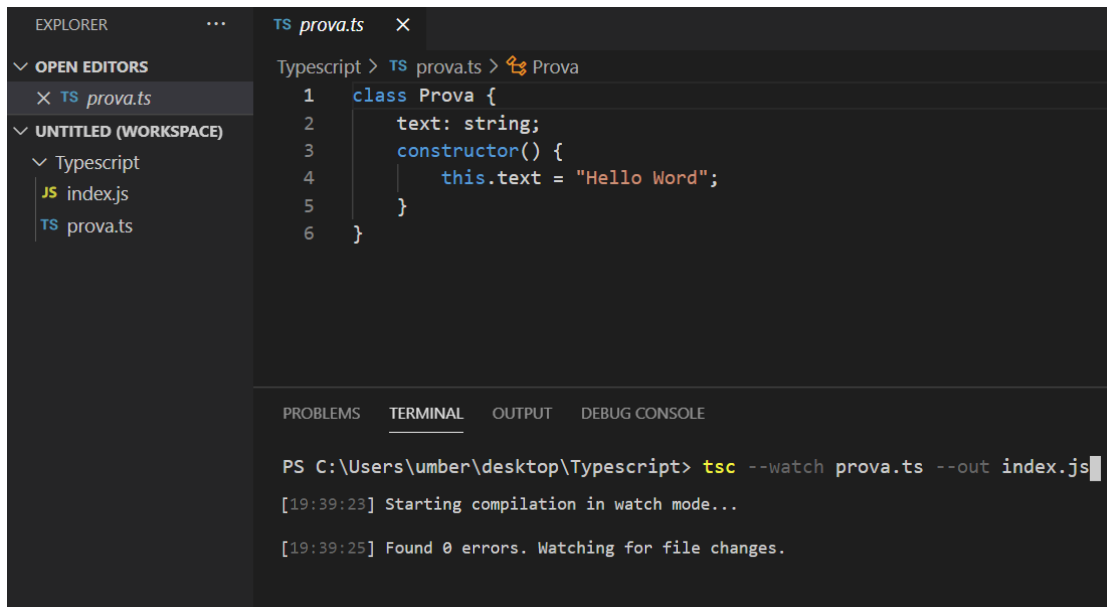
```
1 class Prova {  
2   text: string;  
3   constructor() {  
4     this.text = "Hello Word";  
5   }  
6 }
```

At the bottom, the Terminal panel shows the following commands and output:

```
PS C:\Users\umber\desktop> cd Typescript  
PS C:\Users\umber\desktop\Typescript> tsc prova.ts  
PS C:\Users\umber\desktop\Typescript> tsc prova.ts --out index.js  
PS C:\Users\umber\desktop\Typescript> 
```


Come transpilare codice TypeScript in Javascript in automatico con Watch

--watch



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows a project named 'Typescript' containing two files: 'index.js' and 'prova.ts'. The main editor area displays the contents of 'prova.ts', which is a TypeScript class definition for 'Prova'. The code is as follows:

```
1 class Prova {  
2   text: string;  
3   constructor() {  
4     this.text = "Hello Word";  
5   }  
6 }
```

Below the editor, the TERMINAL panel is active, showing the command to run the TypeScript compiler in watch mode:

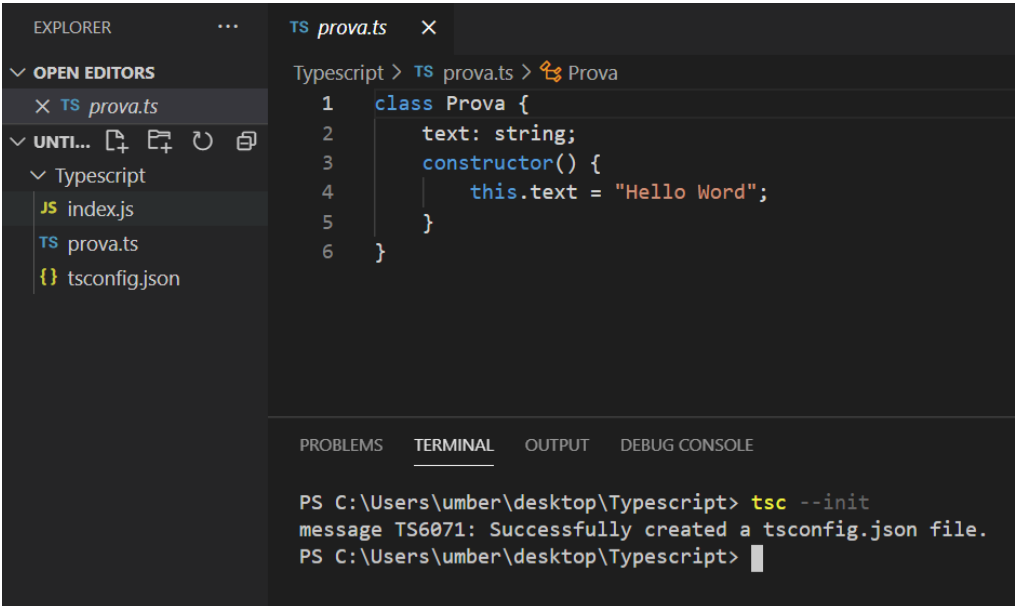
```
PS C:\Users\umber\desktop\Typescript> tsc --watch prova.ts --out index.js
```

The terminal output shows the compiler starting in watch mode and reporting that it found 0 errors:

```
[19:39:23] Starting compilation in watch mode...  
[19:39:25] Found 0 errors. Watching for file changes.
```

Creare un file di configurazione
`tsconfig.json` per
autocompilare il codice.

`--init`



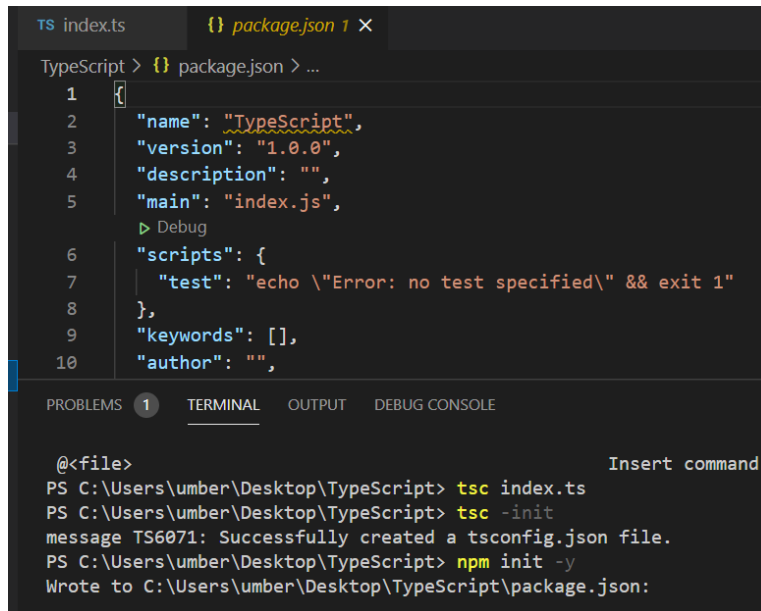
The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows a project named 'Typescript' containing 'index.js', 'prova.ts', and 'tsconfig.json'. The main editor displays 'prova.ts' with the following code:

```
1 class Prova {  
2     text: string;  
3     constructor() {  
4         this.text = "Hello Word";  
5     }  
6 }
```

At the bottom, the Terminal panel shows the command `tsc --init` being executed, resulting in the message: `message TS6071: Successfully created a tsconfig.json file.`

Se non abbiamo TypeScript
installato globalmente o per
distribuire il nostro progetto,
possiamo installare localmente
il modulo di Typescript

Creare il package.json
npm init -y



```
TS index.ts  {} package.json 1 x
TypeScript > {} package.json > ...
1  {
2    "name": "TypeScript",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",

```

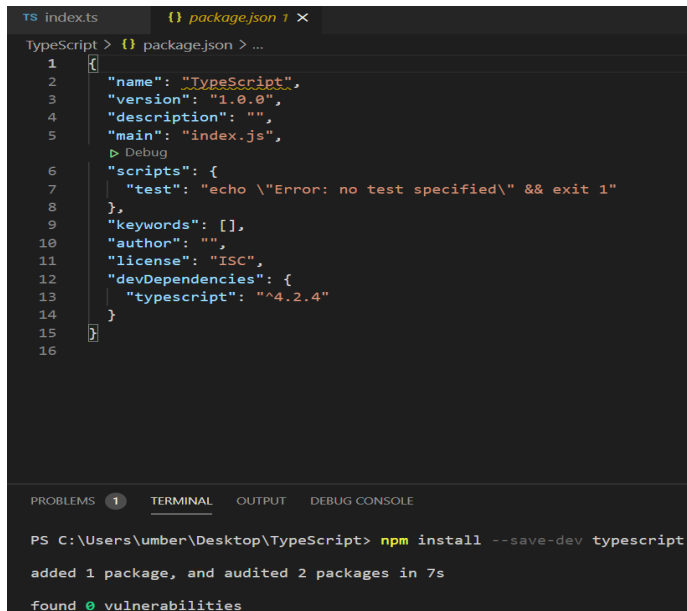
PROBLEMS 1 TERMINAL OUTPUT DEBUG CONSOLE

@<file> Insert command

```
PS C:\Users\umber\Desktop\TypeScript> tsc index.ts
PS C:\Users\umber\Desktop\TypeScript> tsc -init
message TS6071: Successfully created a tsconfig.json file.
PS C:\Users\umber\Desktop\TypeScript> npm init -y
Wrote to C:\Users\umber\Desktop\TypeScript\package.json:
```

Installare il modulo Typescript

**npm install --save-dev
typescript**



```
TS index.ts  () package.json 1 x
TypeScript > () package.json > ...
1 {
2   "name": "TypeScript",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "devDependencies": {
13    "typescript": "^4.2.4"
14  }
15 }
16

PROBLEMS 1  TERMINAL  OUTPUT  DEBUG CONSOLE

PS C:\Users\umber\Desktop\TypeScript> npm install --save-dev typescript
added 1 package, and audited 2 packages in 7s
found 0 vulnerabilities
```

- Differenze principali con Javascript
- <https://www.typescriptlang.org/docs/handbook/basic-types.html>

Tipizzazione delle variabili e delle costanti

```
var myVar: boolean; //boolean – string - number  
myVar = 'Name'; //Errore
```

```
let myAge = 40; //viene tipizzato con il valore iniziale  
  (number) di default  
myAge = 'Name'; //Errore
```

```
const num: number = 3,14;  
num = 5; //Errore
```

Array

Un array è una raccolta omogenea di valori dello stesso tipo di dati. Come per javascript, in un array è possibile aggiungere, rimuovere, ordinare gli elementi attraverso metodi come **push-pop-sort,shift-unshift-reverse-length-indexof**.

```
let myPc = ['a', 'b', 'c']; //viene tipizzato string di default
```

```
// let myPc: string[] = ['a', 'b', 'c'];
```

```
myPc = [1, 2, 3]; //Errore
```

```
let myCar: Array<T> = ['a', 'b', 'c']; //Utilizzando i Generics
```

```
// let myCar: Array<string> = ['a', 'b', 'c'];
```

Tuple

I tipi tupla consentono di esprimere un array con un numero fisso di elementi i cui tipi sono noti, ma non devono necessariamente essere gli stessi.

// Declare a tuple type

let x: [string, number];

// Initialize it

x = ["hello", 10]; // OK

// Initialize it incorrectly

x = [10, "hello"]; // Error

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
//x = [10, "hello"]; // Error

function func(params:[string, number]) {
    console.log(params[0] + ' ' + params[1]);
}

func(x);
```


Enum

Un'utile aggiunta al set standard di tipi di dati da JavaScript è il file enum. Come in linguaggi come C #, un enum è un modo per dare nomi più descrittivi a set di valori numerici. Le enumerazioni consentono di definire un insieme di costanti

enum Color {Red, Green, Blue}; // indici 0,1,2

let c: Color = Color.Green; // 1

let c: Color = Color[1]; // Green

enum Color {Red = 1, Green, Blue}; // indici 1,2,3

let c: Color = Color.Green; // 2

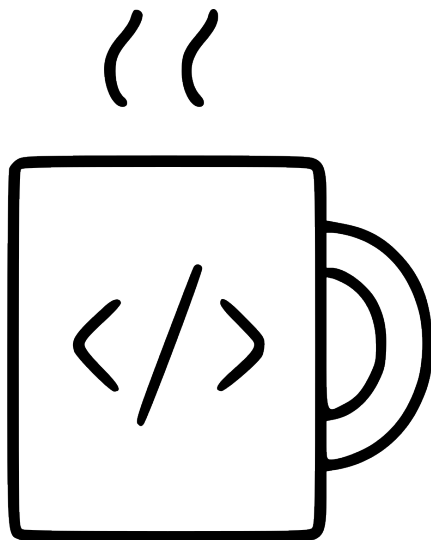
enum Color {Red = 1, Green = 2, Blue = 4}; // negli indici 1 – 2 – 4

let c: Color = Color.Green; // 2

```
enum Color {Red, Green, Blue};

function funColor(color: Color) {
  console.log(color);
}

funColor(Color.Blue);
```



PAUSA

Ci vediamo alle ore 11.25

Any

In alcune situazioni, non tutte le informazioni sul tipo sono disponibili o la sua dichiarazione richiederebbe uno sforzo inadeguato. In questi casi, potremmo voler rinunciare al controllo del tipo. A tal fine, etichettiamo questi valori con any

```
// non definisco il tipo di dato
```

```
let notSure: any = 4;
```

```
notSure = "maybe a string instead";
```

```
notSure = false; // okay, definitely a Boolean
```

```
notSure = [] //ok
```

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

```
list[1] = 100;
```

```
list = 4 //Errore, non è un array
```

```
function funcXyz(x: any): any {  
    return x;  
}
```

Void

// Una funzione che non ritorna un valore(Undefined)

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

//Dichiarare le variabili di tipo void non è utile perché puoi solo assegnarle undefined o null

```
let unusable: void = undefined;
```

Never

// Una funzione che non ritorna nulla

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

Custom Type

Attraverso il custom type è possibile referenziare più tipi primitivi anche **union** ad un tipo unico custom ed essere utilizzato su proprietà o altro.

```
type mioType = string | number;  
  
let x: mioType;  
x = "ciao";  
x = 5;
```

Assertion Type

L'**asserzione del tipo** consente di impostare e comunicare al compilatore il tipo di un valore. Quando tu, come programmatore, potresti avere una migliore comprensione del tipo di una variabile rispetto a ciò che TypeScript può dedurre da solo.

```
let txt = document.querySelector("#txt") as HTMLInputElement ;  
let testo = <HTMLInputElement> document.querySelector("#testo");  
console.dir(txt, testo);
```

Operatori aritmetici

+ addizione
- sottrazione
* prodotto
/ divisione
% modulo

Operatore ternario

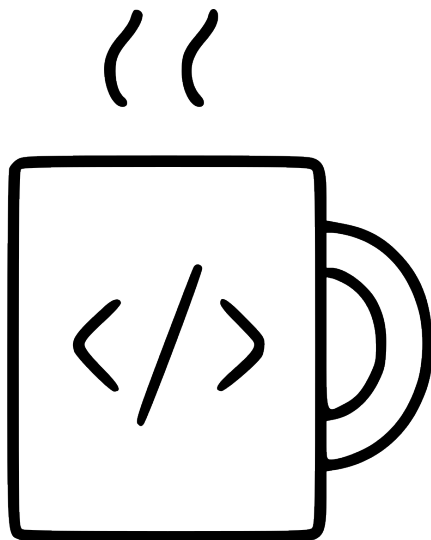
let exp = 5 < 3 ? true : false

Operatori di confronto

<	minore
>	maggiore
<=	minore uguale
>=	maggiore uguale
!=	diverso da
==	uguale a

Operatori logici

&&	AND
	OR
!	NOT



PAUSA

Ci vediamo alle ore 14.00

- Utilizzo delle Interface in TypeScript
- <https://www.typescriptlang.org/docs/handbook/interfaces.html>

La dichiarazione di un interfaccia è un altro modo per denominare un tipo di oggetto:

```
function showUser (user: { firstName: string, lastName: string }) {  
    console.log(user.firstName+ ' , ' + user.lastName);  
}
```

//Definisco una interface che posso riutilizzare nel programma

```
Interface IUser { firstName: string, lastName: string }
```

```
function showUser2 (user: IUser) {  
    console.log(user.firstName+ ' , ' + user.lastName);  
}  
  
let myUser = {firstName : 'Mario', lastName : 'Bianchi'};  
showUser(myUser);  
showUser2(myUser);
```

Proprietà opzionali

//Definisco una interface con proprietà opzionali

```
Interface IUser2 {  
    firstName: string,  
    lastName: string,  
    age?: number } // ? Sta ad indicare la proprietà opzionale
```

```
function showUser3 (user: IUser2) {  
    console.log(user.firstName+ ' , ' + user.lastName);  
}
```

```
let myUser = {firstName : 'Mario', lastName : 'Bianchi'};  
showUser3(myUser);
```

Proprietà opzionali

//Definisco una interface con un numero di proprietà indefinito

Interface IUser3 {

firstName: string,

lastName: string,

age?: number, // ? Sta ad indicare la proprietà opzionale

[propName: string]: any; } // ? Sta ad indicare che potrebbero esserci N ulteriori proprietà

function showUser4 (user: IUser3) {

console.log(user.firstName+ ' , ' + user.lastName);

}

let myUser = {firstName : 'Mario', lastName : 'Bianchi', address: 'via po'};

showUser4(myUser);

- Le funzioni sono la componente fondamentale di qualsiasi applicazione in JavaScript
- <https://www.typescriptlang.org/docs/handbook/functions.html>

Function

Per ricapitolare rapidamente quali sono gli approcci in JavaScript:

// Named function

```
function add(x, y) { return x + y; }
```

// Anonymous function

```
let myAdd = function(x, y) { return x + y; };
```

// Arrow function

```
let myArrowAdd = (x, y) => { return x + y; };
```

Function

Aggiungiamo tipi ai paramentri della funzione e il ritorno

```
function add(x: number, y: number): number {  
    return x + y;  
}
```

```
let myAdd = function(x: number, y: number): number { return x + y; };
```


Function

//Dichiaro una variabile a cui posso assegnare una funzione con i tipi di parametri e il valore di ritorno definito

```
let myAdd: (x: number, y: number) => number;
```

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

```
myAdd(2,3); //5
```

Function Overload

TypeScript fornisce il concetto di sovraccarico di funzioni. È possibile avere più funzioni con lo stesso nome ma tipi di parametri e tipo restituito diversi.

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

add("Hello ", "Steve");
add(25, 35);
add(25, "Steve");
add("Steve", 25);
```

- Gli oggetti in typescript sono definiti da una costante o variabile assegnandone il tipo dove all'interno l'oggetto sarà caratterizzato dalla proprietà e valore.

mode javascript

```
const user = {name:'Mario', lastname:'Rossi', age: 45};  
console.log(user.name)
```

mode typescript

```
const user: {name:string, lastname:string, age: number} = {name:'Mario', lastname:'Rossi', age: 45};  
console.log(user.name)
```

array destructuring

```
let arrStr: [string, string, string] = ["primo", "secondo", "terzo"]  
  
/* let str1: string = arrStr[0];  
let str2: string = arrStr[1];  
let str3: string = arrStr[2]; */  
  
let [str1, str2, str3] = arrStr;  
console.log(str1);
```

object destructuring

```
let obj = {id: 1, nome: "Mario", cognome: "Rossi"}  
let { nome, cognome } = obj  
console.log(cognome);
```

Spread operator

Lo spread operator consente di prelevare i singoli elementi da un oggetto iterabile come un array o una stringa.

```
let arrStr: [string, string, string] = ["primo", "secondo", "terzo"]
console.log(arrStr)
console.log(arrStr[0], arrStr[1], arrStr[2])
console.log(...arrStr)

function testFunc(a: string, b: string, c: string) {
    console.log(a, b, c);
}

testFunc(...arrStr);

let newStrArr = [...arrStr, "quarto"]
let altroArr = [...arrStr, ...newStrArr]
let pushArr = [];
pushArr.push(...arrStr);
```

- Le funzioni sono la componente fondamentale di qualsiasi applicazione in JavaScript
- <https://www.typescriptlang.org/docs/handbook/classes.html>

Classe in Javascript

//in Javascript definisco una classe nel seguente modo

```
function Greeter(message){  
    this.message = message;  
}  
Greeter.prototype.getName = function(){  
    return this.message;  
}  
var msg = new Greeter('Ciao');  
Console.log(msg.getName);
```


Classe in Typescript

//in Typescript definisco una classe nel seguente modo

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

Modificatori di accesso

Public

Private

Protected

```
Typescript > TS prova.ts > ...
1  class Prova {
2      public text1: string
3      protected text2: string
4      private text3: string
5      constructor() {
6          this.text1 = "Hello Word -";
7          this.text2 = "Hello Word --";
8          this.text3 = "Hello Word ---";
9      }
10     getText3(){ return this.text3 }
11 }
12
13 let p = new Prova();
14 console.log(p.text1); //public
15 //console.log(p.text2); //protected
16 //console.log(p.text3); //private
17 console.log(p.getText3) // tramite get
```

Static

Proprietà e metodi **static** hanno caratteristiche differenti rispetto a proprietà e metodi di istanza:

- ✓ Possono essere invocati solo dalla classe
- ✓ Possono collaborare solo con elementi definiti static

```
class Persona {  
    public static count = 0;  
    public name: string;  
    private lastname: string;  
    protected city: string;  
  
    constructor(name: string, lastname: string, city: string) {  
        Persona.count++;  
        this.name = name;  
        this.lastname = lastname;  
        this.city = city;  
    }  
  
    public getLastName(): string { ...  
    }  
  
    public setLastName(lastname: string): void { ...  
    }  
}
```

I paradigmi dell'OOP

Incapsulamento

Il concetto dell'incapsulamento è quello di mantenere riservate le informazioni solo all'interno della classe che li definisce attraverso il modificatore di accesso **private**.

Tutte le proprietà o metodi con modificatore **private** non potranno avere accesso da una classe esterna.

Ereditarietà

E' possibile ereditare tutte le caratteristiche (proprietà o metodi definiti **public**) appartenenti ad una data classe padre(superclasse) utilizzando la keyword **extends** su di una nuova classe figlia (sottoclasse).

Polimorfismo

Si intende '**polimorfico**' la ridefinizione di un metodo appartenente alla superclasse mediante l'utilizzo dell'ereditarietà nella sottoclasse.

Get e Set in Typescript

```
class Animal {  
    name: string;  
    type: string;  
  
    getName() {  
        return this.name;  
    }  
  
    setName(name: string = 'Bobby') {  
        this.name = name;  
    }  
}
```

Estendere una Classe in Typescript

```
class Animal {  
    move(distanceInMeters: number = 0) {  
        console.log(`Animal moved ${distanceInMeters}m.`);  
    }  
}  
  
class Dog extends Animal {  
    bark() { console.log('Woof! Woof!'); }  
}
```

```
const dog = new Dog();  
dog.bark();  
dog.move(10);  
dog.bark();
```

Costruttori in Typescript

```
class Animal {  
  name: string;  
  
  constructor(theName: string) { this.name = theName; }  
  
  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); Override del metodo della classe padre  
  }  
}
```

```
let sam = new Snake("Sammy the  
Python");
```

```
sam.move();  
tom.move(34);
```

Variabili Parametriche

È possibile dichiarare le variabili nei parametri del costruttore e in automatico il costruttore andrà ad impostare le variabili nella classe

```
class Car {  
  constructor (  
    private name: string,  
    protected model: string,  
    public age: number ) {  
  }  
  getAge(){ return this.age; }  
}
```

```
let car = new Car('Fiat 500', 'auto', 5)  
car.getAge(); //5
```



```
class Car {  
  private name: string;  
  protected model: string;  
  public age: number  
  constructor (name: string, model: string,  
    age: number ) {  
    this.name = name;  
    this.model = model;  
    this.age = age;  
  }  
  getAge(){ return this.age; }  
}  
let car = new Car('Fiat 500', 'auto', 5)  
car.getAge(); //5
```


TypeScript getter / setter

```
class Employee {  
    private _fullName: string;  
    get fullName(): string {  
        return this._fullName;  
    }  
    set fullName(newName: string) {  
        this._fullName = newName;  
    }  
}
```

```
let employee = new Employee();  
employee.fullName = "Bob Smith";  
if (employee.fullName) {  
    console.log(employee.fullName);  
}
```

Proprietà readonly

È possibile creare proprietà in sola lettura utilizzando la parola chiave **readonly**. Le proprietà di sola lettura devono essere inizializzate nella loro dichiarazione o nel costruttore.

```
class Octopus {  
    readonly name: string; // Dichiarazione  
    readonly numberOfLegs: number = 8; // Dichiarazione e inizializzazione  
    constructor (theName: string) {  
        this.name = theName; // Inizializzazione nel costruttore  
    }  
}  
  
let dad = new Octopus("Man with the 8 strong legs");  
dad.name = "Man with the 3-piece suit"; // error! name is readonly. }  
}
```

Proprietà Static in TypeScript

```
Class MathCalc{  
    static readonly euroDollaroRate = 1.16;  
    static readonly euroSterlinaRate = 0,89;  
  
    static calcEuroToDollar(euro: number){  
        return MathCalc.euroDollaroRate * euro;  
    }  
    calcEuroToSterlina(euro: number){  
        return MathCalc. euroSterlinaRate * euro;  
    }  
}
```

```
let calc = new MathCalc();  
console.log(MathCalc.euroDollaroRate);  
console.log(MathCalc.euroSterlinaRate);
```

```
console.log(MathCalc.calcEuroToDollar(10));  
console.log(calc.calcEuroToSterlina(10));
```

//Accedo ad un membro **static** tramite il nome della classe piuttosto che tramite un oggetto.

Interface

L'interfaccia è un modello così come una classe ma con specifiche più restrittive assomigliando ad una classe **astratta**. Così come nella classe astratta una interfaccia può avere solo la definizione dei metodi, le proprietà sono costanti nell'interfaccia e non può essere istanziata

L'interfaccia si definisce con la keyword `interface`. Un'interfaccia può estendere n interfacce e può essere implementata da una classe. La classe che implementerà un'interfaccia dovrà implementarne tutte le sue caratteristiche ovvero i metodi definiti sull'interfaccia.

Interface che estende una Classe

```
class Point {  
  x: number;  
  y: number;  
}  
  
interface Point3d extends Point {  
  z: number;  
}  
  
let point3d: Point3d = {x: 1, y: 2, z: 3};
```

Classe che estende una Interface

```
class Point {  
  x: number;  
  y: number;  
}  
  
interface Point3d extends Point {  
  z: number;  
}  
  
class Square implements Point3d {  
  z: number = 10;  
}
```

Interface che estende una Classe

Una interface che estende una classe prende la signature della classe, senza l'implementazione dei metodi e l'inizializzazione delle variabili.

```
class MyLogger {  
  log(msg: string): void { console.log(msg); }  
  generateId():number { return Math.round(Math.random()*100); }  
}  
  
interface MyLog extends MyLogger {  
  email: string;  
}  
  
class MyMailLogger implements MyLog {  
  email: string  
  log() {}  
  generateId() { return 1 }  
}
```

Abstract in TypeScript

Una classe astratta è una classe che ha almeno un metodo astratto. Non si può istanziare direttamente ma deve essere estesa da una classe che deve implementare tutti i metodi astratti definiti.

```
abstract class Animal {  
    abstract makeSound(): void;  
    move(): void {  
        console.log("roaming the earth...");  
    }  
}  
  
class Dog extend Animal {  
    makeSound(): void {  
        console.log("Grrr...");  
    }  
}
```

```
let dog = new Dog();  
dog.makeSound();  
dog.move();
```


Generics in TypeScript

I generics permettono di definire funzioni, classi o interfacce che sono in grado di lavorare con diversi tipi di dato. Per esempio, è possibile creare una funzione in cui il tipo dei parametri e del valore di ritorno non viene specificato fino al momento in cui la funzione viene invocata.

```
function stampa<T, K>(x: T, y:K): void {  
    console.log(x, y);  
}
```

```
stampa<string, string>("ciao", "abc");  
stampa<number, string>(52, "abc");  
stampa<number, boolean>(52, true);  
stampa<User, IUser>(cu, iu1);
```

```
class Test<T, K> {  
    txt?: T;  
    arr?: K[];  
}
```

Generics in TypeScript

Rest è un'architettura software che consente di erogare servizi web.

Rest definisce come creare le URL dei servizi e come utilizzare i metodi HTTP (GET per il recupero di informazioni, POST per inviare dati, PUT sostituisce la risorsa corrente, PATCH per modifiche parziali, DELETE per eliminare una risorsa).

Il principio fondamentale di REST sono le risorse, accessibili tramite una URI.

```
let URLapi = "https://jsonplaceholder.typicode.com/users";
let xhr = new XMLHttpRequest();
xhr.open('GET', URLapi);
xhr.send();
xhr.onreadystatechange = function () {
    if(xhr.readyState === 4 && xhr.status === 200) {
        let obj = JSON.parse(xhr.responseText);
        console.log(obj)
    }
}

let pr = fetch(URLapi).then(response => response.json())
console.log(pr);
pr.then(dato => stampaHtml(dato))

function stampaHtml(dati: any) {
    //stampo HTML
}
```

- Il modulo è un file di Javascript il cui contenuto è isolato, non va a sporcare l'ambiente globale di Javascript.
- <https://www.typescriptlang.org/docs/handbook/modules.html>

Moduli

I moduli vengono eseguiti all'interno del proprio ambito, non nell'ambito globale; questo significa che le variabili, le funzioni, le classi, ecc. dichiarate in un modulo non sono visibili al di fuori del modulo a meno che non vengano esportate esplicitamente.

Al contrario, per consumare una variabile, una funzione, una classe, un'interfaccia, ecc. Esportati da un modulo diverso, è necessario importarli.

Esportare una dichiarazione

Qualsiasi dichiarazione (come una variabile, un'array, una funzione, una classe, un alias di tipo o un'interfaccia) può essere esportata aggiungendo la parola chiave **export**. L'importazione di una dichiarazione esportata viene effettuata utilizzando **import**.

```
//book.ts
```

```
export class Book {
```

```
  title: string
```

```
  author: string
```

```
  content: string
```

```
  year: number
```

```
}
```

```
export const Admin = 'Admin'
```

```
//school.ts
```

```
import { Book, Admin } from './book'
```

```
let book = new Book();
```

```
book.title = 'My Story';
```

```
book.author = 'abc';
```

```
book.content = 'Lorem Ipsum';
```

```
book.year = 2018;
```

```
console.log(book);
```

```
console.log(Admin)
```

Esportare una dichiarazione

Qualsiasi dichiarazione (come una variabile, un'array, una funzione, una classe, un alias di tipo o un'interfaccia) può essere esportata aggiungendo la parola chiave **export**. L'importazione di una dichiarazione esportata viene effettuata utilizzando **import**.

```
//book.ts
```

```
export class Book {
```

```
  title: string
```

```
  author: string
```

```
  content: string
```

```
  year: number
```

```
}
```

```
export const Admin = 'Admin'
```

```
//school.ts
```

```
Import * as books from './book'
```

```
Let book = new books.Book()
```

```
book.title = 'My Story';
```

```
book.author = 'abc';
```

```
book.content = 'Lorem Ipsum';
```

```
book.year = 2018;
```

```
console.log(book);
```

```
console.log(books. Admin)
```

Esportare una dichiarazione di default

Qualsiasi dichiarazione (come una variabile, un'array, una funzione, una classe, un alias di tipo o un'interfaccia) può essere esportata aggiungendo la parola chiave **export**. L'importazione di una dichiarazione esportata viene effettuata utilizzando **import**.

```
//book.ts
```

```
Export default class Book {  
  title: string  
  author: string  
  content: string  
  year: number  
}
```

```
export const Admin = 'Admin'
```

```
//school.ts  
import MyBook from './book'  
let book = new MyBook();  
book.title = 'My Story';  
book.author = 'abc';  
book.content = 'Lorem Ipsum';  
book.year = 2018;  
console.log(book);
```

NameSpace

Man mano che aggiungiamo moduli, vorremo avere una sorta di schema organizzativo in modo da poter tenere traccia dei nostri tipi e non preoccuparci delle collisioni di nomi con altri oggetti. Invece di inserire molti nomi diversi **namespace** globale, racchiudiamo i nostri oggetti in un **namespace**.

<https://www.typescriptlang.org/docs/handbook/namespaces.html>

NameSpace

TypeScript > Es2 > TS testNamespace.ts > ...

```
1 namespace test {  
2   export class x {  
3     txt: string = 'class test'  
4   }  
5  
6   export const y = () => 'func test'  
7  
8   export const z = 'const test'  
9 }
```

```
/// <reference path="testNamespace.ts" />
```

```
let obj = new test.x();  
let func = test.y();  
let cos = test.z;
```

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>Document</title>  
    <script>var exports = { "__esModule": true };</script>  
    <script src="testNamespace.js"></script>  
    <script src="index.js"></script>  
  </head>
```

JSX

JSX è una sintassi simile a XML incorporabile. È pensato per essere trasformato in JavaScript valido, sebbene la semantica di tale trasformazione sia specifica dell'implementazione. JSX è diventato popolare con il framework React , ma da allora ha visto anche altre implementazioni.

TypeScript supporta l'incorporamento, il controllo del tipo e la compilazione di JSX direttamente in JavaScript.

JSX

Per utilizzare JSX devi fare due cose.

- Assegna un'estensione ai file **.tsx**
- Abilita l'opzione **jsx** nel **tsconfig.json**

```
/* Basic Options */  
// "incremental": true,  
"target": "es5",  
"module": "commonjs",  
// "lib": [],  
// "allowJs": true,  
// "checkJs": true,  
"jsx": "preserve",
```

JSX

Modalità	Ingresso	Produzione	Estensione del file di output
preserve	<code><div /></code>	<code><div /></code>	<code>.jsx</code>
react	<code><div /></code>	<code>React.createElement("div")</code>	<code>.js</code>
react-native	<code><div /></code>	<code><div /></code>	<code>.js</code>
react-jsx	<code><div /></code>	<code>_jsx("div", {}, void 0);</code>	<code>.js</code>
react-jsxdev	<code><div /></code>	<code>_jsxDEV("div", {}, void 0, false, {...}, this);</code>	<code>.js</code>



shaping the skills of tomorrow

challengenetwork.it

