

TYPESCRIPT 3

```
// ***** COS'E' TYPESCRIPT?

/**
 * TypeScript è un superset di Javascript, il che significa che aggiunge
funzionalità extra al normale JavaScript,
 * pur rimanendo compatibile con esso e le librerie esistenti. La caratteristica
principale è la tipizzazione statica,
 * che aiuta a trovare errori nel codice mentre lo scriviamo, prima ancora di
eseguirlo. TypeScript non può essere eseguito direttamente,
 * deve essere prima compilato in JavaScript.
 */

// ***** INSTALLAZIONE E COMPILAZIONE

/**
 * Per installare TypeScript, avendo già Node.js e npm, basta eseguire questo
comando :
 * npm install -g typescript
 * Una volta installato, si può compilare un file TypeScript(.ts) in
Javascript(.js) usando il comando:
 * tsc nomefile.ts
 * O individuando la directory esatta:
 * tsc/src/folder/file.ts
 */

// ***** TIPI DI DATO PRINCIPALI

/**
 * Typescript ci permette di definire il tipo delle variabili :
 */
let nome: string = "Nello"; // stringa
let età: number = 23; // numero
let isProgrammer: boolean = true; // booleano
let qualsiasiTipo: any; // disattiva di fatto i controlli di tipo
let numeri: number[] = [1, 2, 3]; // array di numeri
let tuplaArr: [string, number] = ["Nello", 23]; // TUPLA: Array di valori ben
definiti

// TYPE INFERENCE
// anche se non si definisce esplicitamente il tipo di una variabile, TypeScript
può inferirlo in automatico :
let numero = 10; // sa che è un number

// ***** FUNZIONI

/**
 * Le funzioni possono avere parametri tipizzati e definire il tipo del valore
di ritorno :
 */
function somma(a: number, b: number): number {
    return a + b;
}
```

```

somma(10, 10); // si aspetta 2 parametri di tipo number che ritornino un number

/**
 * Si possono avere anche parametri opzionali o con valori predefiniti :
 */
function saluto(nome: string = "Utente"): string {
    return `Ciao ${nome}`;
}

// ***** INTERFACCE

/**
 * Le interfacce permettono di definire la struttura di un oggetto o le regole
 * che una classe deve eseguire :
 */
interface Person {
    nome: string;
    saluta(): void;
}

interface Write {
    scrivi(): void;
}

class Student implements Person, Write {
    nome: string;

    constructor(nome: string) {
        this.nome = nome;
    }

    scrivi(): void {
        throw new Error("Method not implemented.");
    }

    saluta(): void {
        console.log("Scrivi Student class");
    }
}

// ***** CLASSI

/**
 * TypeScript supporta le classi in stile orientato agli oggetti, con
 * modificatori di accesso come
 * public, private e protected :
 */
class Persona {
    public nome: string; // public è di default
    public età: number;

    constructor(nome: string, età: number) {
        this.nome = nome;
        this.età = età;
    }
}

```

```

saluta() {
    console.log(`Ciao Persona class, sono ${this.nome} e ho ${this.età} anni`);
}

presenta(persona: Persona){
    console.log(`Piacere di conoscerti ${persona.nome}`);
}
}
let persona1: Persona = new Persona("Nello", 23);
let persona2: Persona = new Persona("Antonio", 15);
persona1.saluta();
persona2.presenta(persona1);

// è possibile dichiarare le variabili di una classe specificando il modificatore
di accesso stesso nel costruttore :
class Persone{
    constructor(public cognome: string, public età: number) {}

    greeting() {
        console.log(`Sono ${this.cognome} e ho ${this.età}` );
    }
}
let persone: Persone = new Persone("Esposito", 45);
persone.greeting();

// le classi possono estendere altre classi, ma hanno delle regole, ovvero che
oltre il costruttore che riceverà
// le variabili estese avrà anche un super(variabili estese) con le variabili
della classe estesa
class Studenti extends Persone {
    constructor(cognome: string, età: number, private scuola: string){
        super(cognome, età);
    }
}
let students: Studenti = new Studenti("Esposito", 10, "elementari");
students.greeting();

// le classi possono essere anche astratte e mettere a disposizione variabili,
metodi e funzioni di
// default che un'altra classe è obbligata ad utilizzare se la estendono
// abstract class..

// ***** DECORATOR & DECORATOR FACTORY
// sono funzioni accessibili tramite la '@' e sono assegnabili alle classi :
function Logger(constructor: any) {
    console.log("manda a schermo");
    console.log(constructor);
}

@Logger
class Schermo {
    constructor() {

```

```

        console.log("Classe schermo");
    }
}

function LoggerFactory(messaggio: string) {
    return function (constructor: any) {
        console.log(messaggio);
        console.log(constructor);
    };
}

@LoggerFactory("Sono il decorator factory")
class SchermoFactory {
    constructor() {
        console.log("factory");
    }
}

// ***** PATTERN SINGLETON

/**
 * Il Singleton è un pattern che permette di avere una sola istanza di una
 * classe:
 */
class President {
    private static instance: President;

    private constructor(public nome: string) {}

    static getInstance() {
        if (!President.instance) {
            President.instance = new President("Marco");
        }
        return President.instance;
    }

    saluta() {
        console.log(`Buongiorno, sono il preside ${this.nome}`);
    }
}
President.getInstance().saluta();

// ***** UNIONI DI TIPI & INTERSEZIONI

/**
 * Le unioni permettono di usare più tipi per una variabile:
 */
let codice: string | number;
codice = 123;
codice = "ABC";

/**
 * Le intersezioni combinano più tipi in uno:
 */

```

```

interface Son{
    nome: string;
}

interface Father{
    surname: string;
}

let fam: Father & Son = {nome: "Nello", surname: "Casolla"};

// ***** TYPE ALIAS

/**
 * Un type alias permette di dare un nome a un tipo complesso :
 */
type ID = string | number;
let stringId: ID = "pmx";
let codeId: ID = 98;

// ***** TYPE GUARDS
/**
 * Esse aiutano typescript a perfezionare i tipi in base ai controlli di runtime
(tempo in cui viene eseguito) :
 */
function printId(id: string | number){
    if(typeof id === "string") {
        console.log(`ID is a string: ${id}`);
    } else {
        console.log(`ID is a number: ${id}`);
    }
}

// ***** DISCRIMINATED UNIONS
/**
 * Gestiscono più tipi di oggetti, soprattutto con le interfacce :
 */
interface Circle{
    kind: "circle";
    radius: number;
}
interface Square {
    kind: "square";
    sideLength: number;
}
type Shape = Circle | Square;

function getArea(shape: Shape){
    if(shape.kind === "circle"){
        return Math.PI * shape.radius ** 2;
    } else {
        return shape.sideLength ** 2;
    }
}

```

```
// ***** ENUMS
/**
 * Si usano per definire insiemi di costanti nominate:
 */
enum Direction {
    Up, Down, Left, Right
}
let move: Direction = Direction.Down;

// ***** MAPPED TYPES
/**
 * Sono utili per creare dinamicamente nuovi tipi basati su quelli dichiarati in anticipo:
 */
// oggetto che definisce 2 variabili
type Human = {
    name: string;
    age: number;
};
// è come Human ma le proprietà mappate sono di sola lettura e quindi non modificabili
type ReadOnlyHuman = {
    readonly [K in keyof Human]: Human[K];
};
// esempio
let person: ReadOnlyHuman = {
    name: "alice",
    age: 30
}
// person.age = 5; ERRORE la proprietà è di sola lettura

// ***** GENERICS
/**
 * I Generics permettono di scrivere codice riutilizzabile con tipi dinamici. Un esempio
 * è con la creazione di funzioni o classi che può ritornarci qualsiasi tipo:
 */
function identity<T>(valore: T): T {
    return valore;
}

let number = identity<number>(10);
let string = identity<string>("Parola");

// possono essere anche più complessi, quindi avere una stessa funzione che accetta e ritorna più tipi:
function superIdentity<T, U, V>(arg: T, arg2: U, arg3: V): [T, U, V] {
    return [arg, arg2, arg3];
}
let first = superIdentity<boolean, string, number>(false, "ciao", 3);
console.log(first);
let second = superIdentity<string, boolean, number>("arrivederci", true, 2);
console.log(second);
```

```
// ***** IMPORTAZIONE ED ESPORTAZIONE DI MODULI
```

```
/**
 * In un progetto TypeScript, si può dividere il codice in file separati e
 importare o
 * esportare funzioni, classi o variabili:
 */
```

```
// file1.ts
export const greet = "Ciao";
```

```
// file2.ts
/**
 * import {greet} from "./file1";
 * console.log(greet); // 'Ciao'
 */
```

```
// per poter importare i file in questo modo bisogna configurare "module": "es6"
// e aggiungere allo <script> nell'html il 'type'="module" quindi sarà:
// <script type="module" src="..directory"></script>
```

```
// ***** STRUMENTI & ECOSISTEMA
```

```
/**
 * TypeScript è ampiamente supportato, specialmente in React. Si possono creare
 componenti React in
 * TypeScript per avere un typing migliore e individuare errori prima di eseguire
 il codice:
 *
```

```
import React from "react";
interface Props{
  titolo: string,
  sottotitolo?: string,
}
```

```
let Titolo: React.FC<Props> = ({titolo, sottotitolo}) => (
  <h1>{titolo} {sottotitolo && <small>{sottotitolo}</small>}</h1>
);
```

```
FC è un acronimo di Functional Component
*/
```

```
// ***** CONFIGURAZIONE DI TYPESCRIPT & COMPILATORE
```

```
/**
 * Per generare il file 'tsconfig.json' bisogna digitare 'tsc --init'
 * Per progetti più grandi, si può usare il file tsconfig.json per configurare
 TypeScript.
 * Ad esempio, compilando automaticamente tutti i file .ts nel progetto con il
 comando :
```

```
* tsc --watch oppure tsc -w
* In questo modo compilerà tutti i file '.ts' nel programma e li aggiorna
quando ci sono delle modifiche
* E' possibile includere o escludere dei file con le configurazioni "include" o
"exclude", indicando la directory
* Per poter leggere file ',ts' nel browser bisogna attivare la funzione
"sourceMap"
* Per poter indicare la directory che vogliamo compilare bisogna attivare
"rootDir"
*     indicando la directory interessata, cos' compila solo i file
all'interno di essa
* Per poter dividere i file '.ts' che compilano dai file '.js', bisogna
attivare "outDir": "./dist"
*     in questo modo ci genererà tutti i file '.js' in questa cartella "dist"
così da tenerli separati
*/
```

```
// ***** VANTAGGI DI UTILIZZARE TYPESCRIPT
```

```
/**
* - Meno errori: grazie alla tipizzazione statica, troviamo gli errori prima di
eseguire il codice.
* - Codice più chiaro: sapere il tipo di ogni variabile rende più semplice
capire e mantenere il codice.
* - Supporto per le tecnologie moderne: TypeScript funziona bene con framework
come REACT,
*     rendendo più sicuro e semplice lo sviluppo
* - In un grande progetto rende la gestione delle relazioni tra classi, oggetti
o funzioni più semplice
*     e mantiene traccia di tutti i tipi di dati. Aumentando la scalabilità(il
potenziale di crescita)
* - Aggiunge un livello di sicurezza e robustezza sul codice
*/
```

```
// ***** SVANTAGGI DI UTILIZZARE TYPESCRIPT
```

```
/**
* - Progetti piccoli: potrebbe non essere necessario per script rapidi o
applicazioni molto semplici
* - Curva di apprendimento: richiede un pò di tempo per abituarsi, specialmente
provenendo da JS
*/
```