



TYPESCRIPT: BASI

Mercoledì 22 Ottobre



TYPESCRIPT

TypeScript è un superset di JavaScript e viene compilato in JavaScript

TypeScript (TS) è un linguaggio di programmazione di alto livello, che **estende** il JavaScript, aggiungendo un sistema di **tipi statici** e funzionalità avanzate

JavaScript è un linguaggio debolmente tipato, mentre i tipi di TypeScript vengono verificati a compile-time

TYPE SYSTEM

Le regole sull'applicazione dei tipi
e sulla loro conversione sono rigide
(type safety)

FORTE

TypeScript

Il tipo delle variabili viene
verificato a compile-time

DINAMICO

STATICO

Il tipo delle variabili viene
verificato a runtime

JavaScript

```
"5" - 1 = 4
"5" + 1 = "51"
```

DEBOLE

Le regole sull'applicazione dei tipi sono
meno rigide e sono consentite conversioni
implicite tra tipi diversi



BENEFICI DI TYPESCRIPT

- Controllo statico dei tipi → previene bug ed errori
- Autocompletamento e intellisense
- Code refactoring e scalabilità
- Self-documenting
- Supporto per nuove feature del linguaggio

UN PO' DI STORIA

1995

JavaScript per Netscape

1997

ECMAScript (ES) – ES1

1998-99

ES2 – ES3

2009

ES5 + Node.js

2012

2014

2015

ES6 / ES2015

2016

ES2016

2018

⋮

2020

2023

ES2023

TypeScript 0.8

TypeScript 1.0

TypeScript 2.0

TypeScript 3.0

TypeScript 4.0

TypeScript 5.0

ES6+

let e const

```
let r = 3  
const PI = 3.14;
```

arrow functions

```
const hello = () => { console.log("Hello world") };
```

```
const squareArray = myArray.map(x => x * x);
```

```
const fib = (n) => {  
  if (n < 1) throw new Error("Invalid input");  
  return (n <= 2 ? n : fib(n-1)+fib(n-2));  
}
```

ES6+

classi

template
literals

Si istanzia con: `new Person(...)`

```
class Person {  
  name;  
  surname;  
  #fiscalCode;  
  
  constructor(name, surname, fiscalCode) {  
    this.name=name;  
    this.surname=surname;  
    this.#fiscalCode=fiscalCode;  
  }  
  
  greet() {  
    console.log(`Hello, my name is ${this.name} ${this.surname}`);  
  }  
  
  #getGender() {  
    return (this.#fiscalCode[9] < 4 ? "M" : "F");  
  }  
}
```

Proprietà pubbliche

Proprietà privata

Costruttore

Metodo pubblico


Metodo privato

Delimitato da backtick,
espressioni con `${}`

ES6+

destructuring

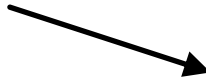
```
const {name, surname} = person;
```



```
const name = person.name;  
const surname = person.surname;
```

```
const {name: firstName, surname: lastName} = person;
```

```
const [first, , third] = myArray;
```



```
const first = myArray[0];  
const third = myArray[2];
```


ES6+

spread operator

```
const myArray = [1, 2, 3];
```

```
const myArray2 = [...myArray];
```

```
const myArray2 = [  
  myArray[0],  
  myArray[1],  
  myArray[2],  
];
```

Shallow copy

```
const book = {  
  title: "Metamorfosi",  
  author: "Franz Kafka"  
};
```

```
const book2 = {...book};
```

```
const book2 = [  
  title: book.title,  
  author: book.author,  
];
```

ES6+

spread operator

```
const printing = { year: 2014, publisher: "Einaudi", ...book};
```

```
const a = [1, 2, 3], b = [4, 5, 6];  
const all = [...a, ...b];
```

rest operator



spread operator
+
destructuring

```
const { year, ...edition } = printing;
```

```
function sumAll(...nums) {  
  return nums.reduce((a, b) => a + b, 0);  
}  
  
console.log(sumAll(1, 2, 3));           // = 6
```

ES6+

promise

```
const p = new Promise((resolve, reject) => {  
  resolve(42);  
});
```

```
p.then(res => console.log(res))  
  .catch(err => console.log(err.message));
```

```
const pSum = (...nums) => {  
  return new Promise((resolve, reject) => {  
    if (nums.every(n => n > 0))  
      resolve(nums.reduce((a, b) => a + b, 0));  
    reject(new Error("Only positive numbers are accepted!"));  
  });  
};
```

```
pSum(1, -2, 3).then(console.log).catch(e => console.log(e.message));  
pSum(1, 2, 3).then(console.log).catch(e => console.log(e.message));
```

Only positive numbers
are accepted!

⓪

ES6+

async/await

```
const handleSum = async (...nums) => {  
  try {  
    console.log(await pSum(...nums));  
  } catch (err) {  
    console.log(err.message);  
  }  
};
```

Converte l'output in una promise

resolve

reject

```
const pSum = async (...nums) => {  
  if (nums.every(n => n > 0))  
    return nums.reduce((a, b) => a + b, 0);  
  throw new Error("Only positive numbers are accepted!");  
};
```

ES6+

moduli

math.js

```
export const PI = 3.14;  
export const square = x => x * x;  
  
export default function multiply(a, b) {  
  return a * b;  
};
```

named exports

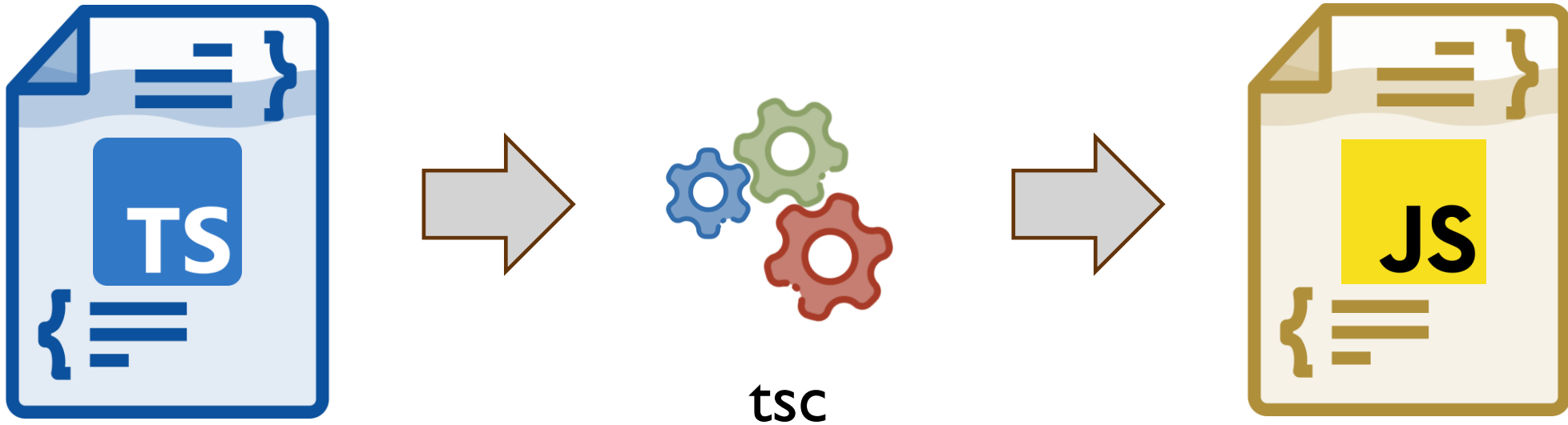
default export

Nome assegnato al default export

I named exports possono essere rinominati

```
import mul, { PI, square as sq } from "./math.js";  
  
console.log(mul(sq(5), PI));
```

ENTERING TYPESCRIPT



INIZIALIZZARE UN PROGETTO TYPESCRIPT

```
npm install --save-dev typescript
```



```
npx tsc
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2022",
    "module": "es2022",
    "rootDir": "src",
    "outDir": "dist",
    "strict": "true"
  },
  "include": ["src/**/*.ts"],
  "exclude": ["**/*.spec.ts"],
  "extends": "../tsc.base.json"
}
```

- versione ES di riferimento per la generazione del codice
- sistema di gestione dei moduli da usare nel codice finale
- directory di base contenente la struttura del codice
- directory in cui saranno salvati i file prodotti
- abilita controlli più rigorosi sul codice TypeScript
- indicano i file da includere e (opzionalmente) da escludere
- estende un altro file tsconfig

TYPE ANNOTATION

variable: type

```
let name: string;
```

*dice al compilatore che la variabile
name è di tipo stringa*

tipo del parametro name

tipo di ritorno della funzione

```
function greet(name: string): string {  
  return `Hello ${name}!`;  
}
```

```
const greet = (name: string): string => {  
  return `Hello ${name}!`;  
}
```


I TIPI PRIMITIVI

string

"Hello world!"

number

42

10.5

boolean

true

false

null

null

undefined



any

"Hello world!"

42

unknown

"Hello world!"

42

never

void

bigint

42n

symbol

Symbol("mySymbol")

I TIPI PRIMITIVI

```
let name: string;
name = "Marco";
name = 42;      // Error: Type 'number' is not assignable to type 'string'
```

```
function repeat(text: string, times: number): string {
  return text.repeat(times);
}

console.log(repeat("Hello", 3));
console.log(repeat("Hello"));      // Error: Expected 2 arguments, but got 1
console.log(repeat("Hello", "3")); // Error: Argument of type 'string' is not
                                   // assignable to parameter of type 'number'
```

ARRAY & TUPLE

array

`type[]`

`Array<type>`

```
const strings: Array<string> = ["a", "b", "c"];
```

```
const nums: number[] = [1, 2, 3];
```

```
nums.push(4);
```

```
nums.push("5"); // Error: Argument of type 'string' is not  
                assignable to parameter of type number
```

tuple

`[type, type, ..., type]`

```
let user: [number, string] = [1, "Matteo"];
```

```
const id = user[0];
```

```
const username = user[1];
```

```
user = ["Marco", 2];
```

```
// Error: Type 'string' is not  
assignable to type 'number'
```

FUNZIONI

default ← `= "World"` *parametro opzionale
(può essere string o undefined)* → `surname?: string`

```
function greet(name: string = "World", surname?: string): string {  
    return `Hello ${name} ${surname ?? ""}!`;  
}
```

```
console.log(greet());  
console.log(greet("Francesco"));  
console.log(greet("Francesco", "Rossi"));
```

```
function helloWorld(): void {  
    console.log("Hello, world!");  
}  
  
function throwError(): never {  
    throw new Error("error");  
}
```

*il rest operator
corrisponde a un array* → `...nums: number[]`

```
function sumAll(...nums: number[]): number {  
    return nums.reduce((a, b) => a + b, 0);  
}
```

FUNZIONI

il secondo argomento deve essere una funzione che prende in input un number e restituisce un number

funzione

(arg: type, ...)

=> type

```
function mapper(arr: number[], fn: (el: number) => number): number[] {  
    return arr.map(fn);  
}
```

il nome dei parametri è obbligatorio

```
console.log(mapper([1, 2, 3], (x: number) => x * 2));  
console.log(mapper([1, 2, 3], (x: number) => x * x));
```

equivale a number:any

~~(number) => number~~



(el: number) => number

FUNZIONI

function overloads

```
function greet(): string;
function greet(name: string): string;
function greet(name?: string): string {
    return `Hello, ${name || "World"}!`;
}
```

le firme (senza implementazione)
sono visibili dall'esterno

la firma che contiene il body deve essere
compatibile con tutte quelle precedenti e
non è visibile dall'esterno

call signatures

(arg: type, ...): type

```
{
  (name: string): string;
}
```

construct signatures

new (arg: type, ...): type

```
{
  new (name: string): Greet;
}
```

OGGETTI

oggetto

{ name: type; ...}

```
let book: { title: string; author: string; pages: number };  
book = { title: "1984", author: "George Orwell", pages: 328 };
```

non si può assegnare

proprietà opzionale

```
let user: { readonly id: number; username?: string };
```

```
user = { id: 1 };  
user.username = "Luca";
```

```
user = { id: 2, username: "Giovanni" };  
user.id = 2; // Error: Cannot assign to 'id' because it is a read-only property
```

OGGETTI

index signature:
indica che le proprietà di questo oggetto sono di tipo string

```
let vocabulary: { [key: string]: string } = {};  
  
vocabulary.apple = "A fruit";  
vocabulary.banana = "Another fruit";  
vocabulary.number = 42;           // Error: Type 'number' is not assignable  
                                   to type 'string'
```


UNION TYPES

union type

type | type | ...

```
function welcomePeople(x: string[] | string) {  
  if (Array.isArray(x)) {  
    console.log("Hello, " + x.join(" and "));  
  } else {  
    console.log("Hello, " + x);  
  }  
}
```

*x può essere una stringa o
un array di stringhe*

```
function greet(name: string = "World", surname?: string): string {  
  return `Hello ${name} ${surname ?? ""}!`;  
}
```

*nel corpo della funzione il tipo di
surname equivale a string | undefined*

LITERAL TYPES

```
let color: "red";  
color = "red";  
color = "blue"; // Error: Type '"blue"' is not assignable to type '"red"'
```

```
let primaryColor: "red" | "green" | "blue";  
  
primaryColor = "red";  
primaryColor = "green";  
primaryColor = "blue";  
primaryColor = "yellow"; // Error: Type '"yellow"' is not assignable  
                           to type '"red" | "green" | "blue"'.
```

```
let day: 1 | 2 | 3 | 4 | 5 | 6 | 7;
```

TYPE ALIASES

type alias

type name = def

il tipo della variabile è a
tutti gli effetti string

```
type MyString = string;  
let a: MyString = "Hello world!";
```

```
type Coordinate = [number, number];  
let point: Coordinate = [100, 200];
```

```
type Direction = "N" | "NE" | "E" | "SE" | "S" | "SW" | "W" | "NW";  
type Wind = {  
  direction: Direction;  
  speed: number;  
};  
let wind: Wind = { direction: "NE", speed: 100 };
```

INTERSECTION TYPES

intersection type

type & type & ...

```
type LandAnimal = {  
  legs: number;  
  walk: () => void;  
};
```

```
type AquaticAnimal = {  
  swim: () => void;  
};
```

```
type Amphibian = LandAnimal & AquaticAnimal;
```



```
const frog: Amphibian = {  
  legs: 4,  
  walk: () => {  
    console.log("Frog is walking...");  
  },  
  swim: () => {  
    console.log("Frog is swimming...");  
  }  
};
```

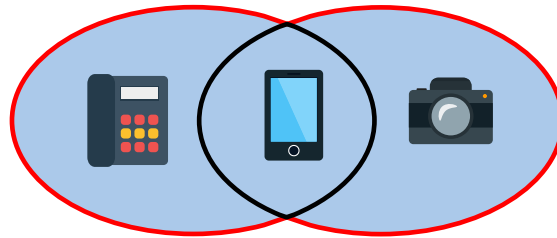
UNION & INTERSECTION

```
type Phone = {  
  phoneNumber: string;  
  turnOn: () => void;  
  makeCall: () => void;  
};
```

```
type Camera = {  
  resolution: string;  
  turnOn: () => void;  
  takePhoto: () => void;  
};
```

union

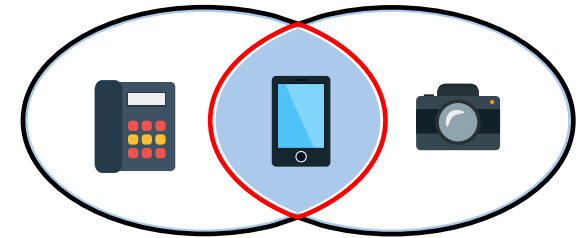
```
type Device = Phone | Camera;
```



```
turnOn: () => void;
```

intersection

```
type SmartPhone = Phone & Camera;
```



```
phoneNumber: string;  
resolution: string;  
turnOn: () => void;  
makeCall: () => void;  
takePhoto: () => void;
```

CLASSI

una classe definisce un tipo

gli attributi hanno un tipo, come le variabili

gli attributi e i metodi possono avere access modifiers (default public)

- public
- private
- protected

Don't try this at home!

```
actor["name"]
```

```
class Actor {  
  private name: string = "";  
  private costume: string = "";  
  
  constructor(name: string) {  
    this.name = name;  
  }  
  
  public setCostume(newCostume: string): void {  
    this.costume = newCostume;  
  }  
  
  public play(): void {  
    console.log(` ${this.name} enters the stage,  
                wearing ${this.costume || "no costume"} `);  
    console.log(` ${this.name}: To be or not to be... `);  
  }  
}
```

gli attributi vanno dichiarati e inizializzati (strict mode)

nei metodi i tipi si dichiarano come nelle funzioni

CLASSI

```
class Person {  
    protected name: string = "";  
    private static counter: number = 0;  
  
    constructor(name: string) {  
        this.name = name;  
        Person.counter++;  
    }  
  
    public static getCount(): number {  
        return Person.counter;  
    }  
}
```

i metodi e gli attributi statici si
accedono senza istanziare la classe:

`Person.getCount()`

gli attributi e i metodi statici seguono
le stesse regole di ereditarietà

```
class Actor extends Person {  
    private costume: string = "";  
  
    constructor(name: string) {  
        super(name);  
    }  
  
    public setCostume(newCostume: string): void {  
        this.costume = newCostume;  
    }  
}
```

la classe figlia eredita attributi e
metodi dalla classe padre

invoca il costruttore del padre

CLASSI

```
abstract class Animal {  
  abstract legs: number;  
  abstract makeSound(): void;  
  
  describe(): void {  
    console.log(`I have ${this.legs} legs.`);  
  }  
}
```

```
class Cat extends Animal {  
  legs = 4;  
  makeSound(): void {  
    console.log("Meow!");  
  }  
}
```

```
class Canary extends Animal {  
  legs = 2;  
  makeSound(): void {  
    console.log("Tweet!");  
  }  
}
```

```
const animals: Animal[] = [new Cat(), new Canary()];  
  
animals.forEach((a) => {  
  a.describe();  
  a.makeSound();  
});
```


INTERFACCE

```
interface Vehicle {  
    wheels: number;  
    drive(): void;  
}
```

non ci sono access modifier
perché i metodi e gli
attributi di un'interfaccia
possono essere solo public

```
class Bike implements Vehicle {  
    wheels: number = 2;  
  
    drive(): void {  
        console.log("Riding a bike");  
    }  
}
```

```
class Car implements Vehicle {  
    wheels: number = 4;  
  
    drive(): void {  
        console.log("Driving a car");  
    }  
}
```

le classi che
implementano
l'interfaccia devono
inizializzare tutti
gli attributi e
implementare i
metodi definiti
nell'interfaccia

INTERFACCE VS TYPE ALIAS

interfaccia

```
interface Vehicle {  
  wheels: number;  
  drive(): void;  
}  
  
const myCar : Vehicle = {  
  wheels: number = 4;  
  
  drive(): void {  
    console.log("Driving a car");  
  }  
}
```

type alias

```
type Vehicle = {  
  wheels: number;  
  drive(): void;  
}  
  
const myCar : Vehicle = {  
  wheels: 4,  
  
  drive(): void {  
    console.log("Driving a car");  
  }  
}
```

INTERFACCE VS TYPE ALIAS

interfaccia

- Le interfacce si applicano solo agli oggetti
- Le interfacce si possono estendere tramite ereditarietà o declaration merging

```
interface Animal {  
    species: string;  
}  
  
interface Mammal extends Animal {  
    feedingMonths: number;  
}
```

type alias

- Gli alias si applicano a qualsiasi tipo
- Gli alias si possono estendere tramite intersection

```
type Animal = {  
    species: string;  
}  
  
type Mammal = Animal & {  
    feedingMonths: number;  
}
```

ENUMS

numeric enums

```
enum ServiceStatus {  
  Active, —————→ 0  
  Inactive, —————→ 1  
  Pending, —————→ 2  
}
```

```
let currentStatus: ServiceStatus = ServiceStatus.Active;  
let statusName: string = ServiceStatus[currentStatus];  
console.log(`Current service status is ${statusName}`);
```

string enums

```
enum Color {  
  Red = "red",  
  Green = "green",  
  Blue = "blue",  
}
```

```
let backgroundColor: Color = Color.Red;  
console.log(`Bgcolor is ${backgroundColor}`);
```

Meglio usare
un oggetto!



MODULI

File TS	File JS	Modulo	Target	Import
utils.mts	utils.mjs	ESM	Qualsiasi	import utils from "../utils.mjs"
utils.cts	utils.cjs	CJS	Qualsiasi	import utils from "../utils.cjs"
utils.ts	utils.js	CJS	Node (default)	import utils from "../utils"
		ESM	Node (type="module")	import utils from "../utils.js"
		ESM	Bundler	import utils from "../utils"

DEFINITELY TYPED



1.12.2 • Public • Published 6 days ago

npm i axios

```
axios.get();  
  
(method) Axios.get<T = any, R = {  
  data: T;  
  status: number;  
  statusText: string;  
  headers: AxiosResponseHeaders | Partial<RawAxiosHeaders & {  
    Server: AxiosHeaderValue;  
    "Content-Type": AxiosHeaderValue;  
    "Content-Length": AxiosHeaderValue;  
    "Cache-Control": AxiosHeaderValue;  
    "Content-Encoding": AxiosHeaderValue;  
  } & {  
    "set-cookie": string[];  
  }>;  
}>;
```



19.1.1 • Public • Published 2 months ago

npm i react
npm i -D @types/react

```
React.Component  
  
class Component<P = {}, S = {}, SS = any> extends ComponentLifecycle<P, S, SS> {  
  static contextType?: Context<any> | undefined;  
  static propTypes?: any;  
  constructor(props: P);  
  constructor(props: P, context: any);  
  context: unknown;  
  setState<K extends keyof S>(state: S | ((prevState: Readonly<S>, props: Readonly<P>) =>  
Pick<S, K> | S | null) | Pick<S, K> | null, callback?: () => void): void;  
  forceUpdate(callback?: () => void): void;  
  render(): ReactNode;  
  readonly props: Readonly<P>;  
  state: Readonly<S>;  
}
```

daisyui

5.1.13 • Public • Published 3 days ago

npm i daisyui

any
daisyui.init();



RIPASSO: I TIPI SEMPLICI

tipi primitivi	string, number, boolean, null, undefined, any, unknown, never, void
type annotations	<code>const</code> uname: string = "marco";
array e tuple	<code>const</code> nums: number[] = [1, 2, 3]; <code>const</code> xy: [number, number] = [1, 5];
funzioni	<code>function</code> exec(fn: () => string): string { return fn(); }
oggetti	<code>const</code> book: { title: string; author: string } = { title: "1984", author: "George Orwell", };

type alias	<code>type</code> Animal = { species: string; };
union e intersection	<code>type</code> Dirs = "N" "S" "O" "W"; <code>type</code> LandA = Animal & {legs: number};
interfacce	<code>interface</code> Person { name: string; surname: string; }
classi	<code>class</code> Programmer <code>implements</code> Person { name: string; surname: string; constructor(n: string, s: string) { ... } }