



PUNTO  
CONFINDUSTRIA

**TYPESCRIPT**

**18/12/2019**

**Paolo Cargnini**



# Cos'è TypeScript



Un Super-set di JavaScript

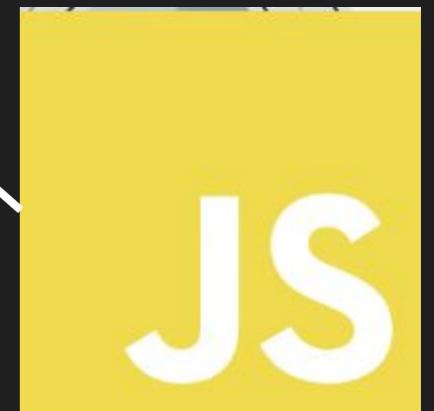
Un linguaggio che estende la  
sintassi di JavaScript

Aggiunge  
features e  
vantaggi a JS

Il browser non  
può eseguirlo!!



Compila  
in



# Come fa ad aggiungere features se l'output è JavaScript?

Di fatto, semplifica una sintassi che  
altrimenti sarebbe difficile da leggere e/o  
usare

**...Permettendoci quindi di  
trovare gli errori prima e  
evitare alcuni errori in runtime**

# Features

- Tipizzazione
- Compilazione in versioni più vecchie di ECMAScript

**Non è stato il primo superset  
che aggiungeva features**



1. Migliora la sintassi
2. Aggiungeva le classi
3. Aggiungeva la concatenazione di stringhe



Tutte funzioni che sono state aggiunte in  
ECMAScript 6

**Con TS però, si parla di  
“metodo di utilizzo”**

**... Che è come parlare di religioni**

The logo consists of the letters 'TS' in a bold, white, sans-serif font, centered within a solid blue square.

**La tipizzazione non  
piace a tutti.**

The word 'BABEL' in a large, black, hand-drawn style font, centered within a solid yellow square.

# Perché TypeScript?

Il problema più vecchio del mondo...

```
function add(num1,num2){  
    return num1 + num2  
}  
  
console.log(add("2","3")) // -> "23"
```

# Bisogna....

```
function add(num1,num2){  
  if (typeof num1 === "number" && typeof num2 === "number"){  
    return num1 + num2  
  }  
  return +num1 + +num2;  
}  
  
console.log(add("2","3"))//-> "23"
```

- Assicurarsi che si stia parlando di numeri
  - Se così non fosse, tradurli
- Mettere d'accordo tutti gli sviluppatori
- Mantenere aggiornata la documentazione (e i commenti...)
- E ancora qualcuno continuerà a mandare stringhe a quella funzione!

# Primo Esempio

```
const button = document.querySelector("button");
const input1 = document.getElementById("num1");
const input2 = document.getElementById("num2");

function add(num1, num2) {
  return num1 + num2;
}

button.addEventListener("click", function() {
  console.log(add(input1.value, input2.value));
});
```

I

\***element.value**, in JS, è sempre una stringa

# Metodo JS di fare il check

```
const button = document.querySelector("button");
const input1 = document.getElementById("num1");
const input2 = document.getElementById("num2");

function add(num1, num2) {
  if (typeof num1 === "number" && typeof num2 === "number") {
    return num1 + num2;
  } else {
    return +num1 + +num2;
  }
}

button.addEventListener("click", function() {
  console.log(add(input1.value, input2.value));
});
```

# TypeScript ci aiuterà con:

**Tipizzazione!**

**Funzionalità di tipizzazione avanzate non presenti in JS  
(Interface e Generics)**

**Possibilità di configurare a nostro piacimento la soluzione e le modalità di compilazione**

**JS di ultima generazione  
(compilato anche nelle versioni più vecchie)**

**(proprio come Babel)**

**Funzionalità avanzate di meta-programmazione  
(Decorators)**

**Integrazione smart con i moderni IDE  
Sintax highlight, alert, autocomplete ecc ecc**

# Tappe del corso

**Setup dell'ambiente di sviluppo**

**Classi e Interface**

**Lavorare con Namespaces e modules**

**TypeScript Base**

**Tipi avanzati e funzionalità aggiuntive di TypeScript**

**Webpack**

**Configurazione del compilatore**

**Generics**

**Librerie di terze Parti e integrazione con TS**

**Sguardo alle funzionalità di ECMAScript6**

**Decorators**

# Setup dell’ambiente di sviluppo

Visual Studio Code con:

- ❖ Estensione “ESLINT”  
(che comprende anche TSLINT)
- ❖ Estensione Material Icon  
(Perché è bella)
- ❖ Estensione Path Intellisense
- ❖ Estensione Prettier

# Dipendenze e organizzazione dei file.

- Installazione di NodeJs
- Installazione globale di TypeScript Compile
- Come trovare il codice di queste slides
- Setup del progetto base per provare
- Deep look al package.json

# Installazione di NodeJs

NVM è l'unica via corretta.  
(diffidate dalle altre)

<https://github.com/coreybutler/nvm-windows>

VSCODE ha il terminale, ma se non vi piacesse...

<https://cmder.net/>

# Installazione globale di TypeScript Compile

`npm i -g typescript`

per usarlo:

`tsc`

# Come trovare il codice di queste slides

<https://github.com/BoomPcargnin/CORSOTypeScript>

# Diamo un'occhiata al package.json

# Usare i tipi

Sintassi core e funzionalità

# Core types

number

1,2,5,4.3, -10, NaN

Tutti i numeri, nessuna differenza tra interi e con la virgola

string

"Abc", 'abc', `abc`

Tutti i valori testuali

boolean

true, false

Solo true e false, non i valori “truthy” (“0”, ‘string’ ecc ecc) o falsy (0,null, undefined, ecc ecc)

Object

{age: 27}

qualsiasi oggetto JS, ci sono tipi più specifici.

Array

[1,2]

qualsiasi array JS, si può essere flessibili o severi.

Tuple

[1,2]

Aggiunta di TypeScript: Array super severi

enum

enum {NEW,OLD}

Aggiunta di TypeScript: oggetti che mappano valori

# Core types principali

number

1,2,5,4.3, -10, NaN

Tutti i numeri, nessuna differenza tra interi e con la virgola

string

"Abc", 'abc', `abc`

Tutti i valori testuali

boolean

true, false

Solo true e false, non i valori "truthy" ("0", 'string' ecc ecc) o falsy (0,null, undefined, ecc ecc)

# TS capisce il tipo delle variabili

```
const number1 = 5; // 5.0
const number2 = 2.8;
const printResult = true;
const resultPhrase = 'Result is: ';
```

```
let a = 4
```

```
let a: number
Type '"ciao"' is not assignable to type 'number'. ts(2322)
Peek Problem No quick fixes available
a = 'ciao'
```

# Lavorare con le funzioni

```
function add(n1: number, n2: number) {  
    return n1 + n2;  
}
```

```
const number1 = '5';  
const number2 = 2.8;  
  
const result = add(number1, number2);  
console.log(result);
```

app.ts:8:20 - error TS2345: Argument of type '"5"' is not assignable to parameter of type 'number'.

```
function add(n1: number, n2: number, showResult: boolean, phrase: string) {
    const result = n1 + n2;
    if (showResult) {
        console.log(phrase + result);
        // console.log(` ${phrase} ${result}`); // -> Stessa cosa
    } else {
        return result;
    }
}

const number1 = 5;
const number2 = 2.8;
const printResult = true;
const resultPhrase = 'Il risultato è: ';
const result = add(number1, number2, printResult, resultPhrase);
console.log(result);
```

# Questo significa che

- TS, Non cambia JS
- ci aiuta a scrivere e capire meglio il codice
- di default compila il codice lo stesso  
(Anche quando trova degli errori logici che abbiamo compiuto).

# TypeScript types vs JavaScript Types

I controlli sui tipi di variabili possono essere scritti anche in typescript poiché TS compila poi in JS

```
function add(n1: number, n2: number) {
  if (typeof n1 !== 'number' || typeof n2 !== 'number') {
    throw new Error('Incorrect input!');
  }
  return n1 + n2;
}

const number1 = '5';
const number2 = 2.8;

const result = add(number1, number2);
console.log(result);
```

**Il concetto è che**

**Js Usa tipi dinamici**

**Ts Usa tipi statici**

# Piccola nota

in TypeScript si lavora con tipi di variabili  
come “string” e “number”, non “String” e  
“Number”

# Inizializzazione variabili

```
const number1 = 5;
const number2 = 2.8;
const printResult = true;
const resultPhrase = 'Il risultato è: ';
const result = add(number1, number2, printResult, resultPhrase);
```

JS capisce il tipo fin dall'inizializzazione. Sta noi decidere se è abbastanza.

```
const number1 : number = 10
let numberOrName : (number | string) = 10
numberOrName = 'Paolo'
```

# Inizializzazione variabili

```
let a = 4
```

```
let a: number
```

```
Type '"ciao"' is not assignable to type 'number'. ts(2322)
```

```
Peek Problem No quick fixes available
```

```
a = 'ciao'
```

Con TS. Non possiamo cambiare tipo.

# Inizializzazione variabili

```
let number1: number = 5; // 5.0
```

```
let number1 = 5; // 5.0
```

- **Essere specifici è quindi inutile per TS**
- **Venendo ignorato da JS, possiamo comunque specificarlo in onore della leggibilità**

# Quindi...

I tipi di JS sono controllati durante?

I tipi di TS sono controllati durante?

# Quindi...

I tipi di JS sono controllati durante il **runtime**, e possono generare errori all'utente finale

I tipi di TS sono controllati durante la **compilazione**, e non generano errori all'utente finale.

# Iniziamo ad approfondire con gli altri tipi principali

Object

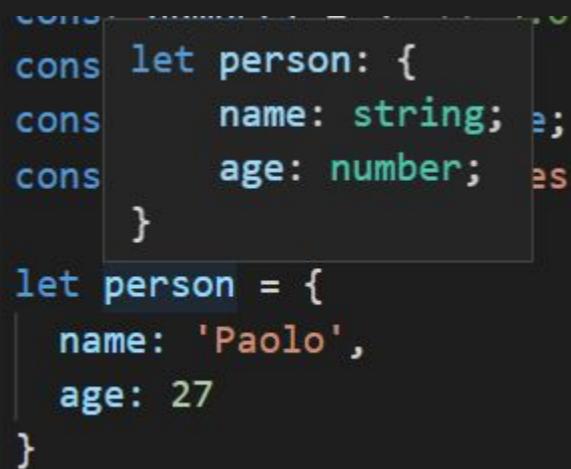
{age: 27}

Qualsiasi oggetto JS, ci sono tipi più specifici.

# Anche qui, TS fa tutto da solo

```
let person = {  
    name: 'Paolo',  
    age: 27  
}
```

```
console.log(person.nickname) // ERROR. Nickname doesn't exist  
person.surname = 'Cargnini' // ERROR. surname doesn't exist
```



A screenshot of a code editor showing a TypeScript code snippet. A tooltip is displayed over the word 'surname' in the second line of code, showing its type definition: 'string'. The code is as follows:

```
const let person: {  
    name: string;  
    age: number;  
}  
let person = {  
    name: 'Paolo',  
    age: 27  
}
```

# Questo, non è un oggetto JS.

---

```
const person: {  
    name: string;  
    age: number;  
}
```

è un Object type

# Si può comunque essere più descrittivi

```
let person: object = {  
  name: 'Paolo',  
  age: 27  
}
```

```
let person: {  
  name: string,  
  age: number,  
} = {  
  name: 'Paolo',  
  age: 27  
}
```

**Di norma. Non si specifica solo il tipo, ma anche il modello.**

# Object type in un object type

```
const product : {  
    id: string;  
    price: number;  
    tags: string[],  
    details: {  
        title: string;  
        description: string;  
    }  
} = {  
    id: 'abc1',  
    price: 12.99,  
    tags: ['offerte-50', 'nuove-uscite'],  
    details: {  
        title: 'Tappeto rosso',  
        description: 'Tappeto bellissimo!'  
    }  
}
```

# Chiudiamo la lista delle variabili semplici

Array

[1,2]

Qualsiasi array JS, si può essere flessibili o severi.

# Come sempre, riconosce il tipo in automatico

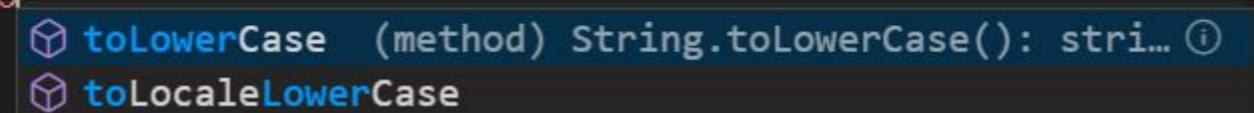
```
let person = {  
    name: 'Paolo'  
    (property) hobbies: string[]  
    hobbies: ['Videogames','gatti']  
}
```

## Inizializzazione variabili:

```
let hobbies: string[] = ['Sports','Cooking']  
let hobbiesWithNumbers: (string|number)[] = ['Sports','Cooking']  
let hobbiesWithAny: any[] = ['Sports','Cooking']
```

# Aiuta l'autocomplete

```
let hobbies: string[] = ['Sports', 'Cooking']
for (const hobby of hobbies){
    hobby.toLowerCase()
}
```



A screenshot of a code editor showing a tooltip for the `toLowerCase` method. The tooltip shows two options: `toLowerCaseCase` (method) `String.toLowerCase(): string` and `toLocaleLowerCase`.

```
let hobbies = ['S' any
for (const hobby o Property 'map' does not exist on type 'string'. ts(2339)
    console.log(hobb Peek Problem No quick fixes available
    console.log(hobby.map())
}
```

# Tipi di variabili non semplici

- Rendono il codice ancora più descrittivo
  - Ancora più error-proof
  - Devono essere specificate dallo sviluppatore, altrimenti TS utilizzerà di default le variabili semplici già viste

# TS core types

Tuple

[1,2]

Aggiunta di TypeScript: Array super severi

# Array a struttura costante

```
const person = {  
    name: 'Maximilian',  
    age: 30,  
    hobbies: ['Sports', 'Cooking'],  
    role: [2, 'author']  
};
```

E va per forza specificato

```
const person:{  
    name:string,  
    age:number,  
    hobbies:string[],  
    role: [number,string]  
} = {  
    name: 'Maximilian',  
    age: 30,  
    hobbies: ['Sports', 'Cooking'],  
    role: [2, 'author']  
};
```

# Tipi di eccezioni generabili

```
numbers [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
(property) role: [number, string]
Type '10' is not assignable to type 'string'. ts(2322)
Peek Problem No quick fixes available
person.role[1] = 10
```

# Domanda...

```
person.role.push('admin');
```

```
person.role = [2, 'author', 'admin']
```

Sono la stessa cosa?

Generano eccezioni?

```
person.role.push('admin');
```

**Eccezione  
consentita**

```
person.role = [2,'author','admin']
```

**Genera  
un'eccezione**

**Non ci sono differenze logiche.  
E’ “severo” solo con:**

- L’assegnazione di nuovi array
- La gestione dei tipi all’interno dell’array

```
role = [2]
role = []
role = ['2','admin','hi']
```

# Perchè usare i tuple

- Più severità  
*(che poi è anche lo scopo di TS)*
- Più precisione
- Più documentazione non scritta  
*(Capire con che dati lavori e con che dati ti aspetti)*

# Ts Core types (+ introduzione ai custom type)

## Tipi enum

enum

enum {NEW,OLD}

Aggiunta di TypeScript: oggetti che mappano valori

# Anti-behavior in JS

```
const person = {  
    name: 'Maximilian',  
    age: 30,  
    hobbies: ['Sports', 'Cooking'],  
    role: 'READ ONLY USER'  
};
```

```
if (person.role === 'READ-ONLY-USER') {  
    console.log('is read only');  
}
```

# Pattern più comune

```
const ADMIN = 0;
const READ_ONLY = 1;
const AUTHOR = 2;

const person = {
  name: 'Maximilian',
  age: 30,
  hobbies: ['Sports', 'Cooking'],
  role: ADMIN
};
```

# Però..

1. Dobbiamo ricordarci il nome delle variabili
2. Usando il semplice type *number* potremmo anche inserire valori non validi ( $> 2$ )

# Utilizziamo enum

```
enum Role { ADMIN, READ_ONLY, AUTHOR };
```

è un custom type perché:

- Sostituisce i più comuni “var”, “let” e “const”
- Trasforma quanto scritto in del codice JS più complesso

```
var Role;
(function (Role) {
    Role[Role["ADMIN"] = 0] = "ADMIN";
    Role[Role["READ_ONLY"] = 1] = "READ_ONLY";
    Role[Role["AUTHOR"] = 2] = "AUTHOR";
})(Role || (Role = {}));
```

# Focus nel compilato

```
var Role;
(function (Role) {
    Role[Role["ADMIN"] = 0] = "ADMIN";
    Role[Role["READ_ONLY"] = 1] = "READ_ONLY";
    Role[Role["AUTHOR"] = 2] = "AUTHOR";
})(Role || (Role = {}));
```

- è una funzione (che di fatto è un oggetto)
  - Assegna a Role[0] il valore Admin ...
  - Di default i valori degli indici sono 0,1,2...

# Modi di assegnare enum

```
// ADMIN = 1
// READ_ONLY = 2
// AUTHOR = 3

enum Role { ADMIN, READ_ONLY, AUTHOR }

// const ADMIN = 1;
// const READ_ONLY = 2;
// const AUTHOR = 3;

enum Role { ADMIN = 1 , READ_ONLY, AUTHOR  };

// const ADMIN = "admin";
// const READ_ONLY = "read_only";
// const AUTHOR = "author";

const getSomeValue = () => {
  return 2
}

enum E [
  A = getSomeValue(),
  B = 4, // Error! Enum member must have initializer.
]
```

# Un po' di eccezioni:

- 1. Usare stringhe come valori è un anti-pattern**
- 2. Se si usa una funzione, quest'ultima deve tornare un numero**
- 3. Se si valorizza il primo numero con qualcosa di diverso di un numero, bisogna valorizzare tutti i valori**

# Utilizzare gli enum

```
const person = [
  name: 'Maximilian',
  age: 30,
  hobbies: ['Sports', 'Cooking'],
  role: Role.ADMIN
];
if (person.role === Role.AUTHOR) {
  console.log('is author');
}
```

# Sono quindi ottimi per:

- Variabili mappate con dietro un valore da mappare
  - Rendere le variabili leggibili

# L'ultimo tipo, il fratellino sfigato

any

\*

Può essere qualsiasi cosa

# Vantaggi

- Non verrà eseguito nessun controllo
  - TS non ti dirà mai niente se lo usi
  - Ti permette di essere flessibile

**Da usare quando non si ha assoluta certezza del dato che si va ad elaborare**

# Svantaggi

- Rende TS meno leggibile
  - E' di fatto, inutile
- Trasforma l'esperienza di sviluppo  
uguale a quella di JS

**Da usare il meno possibile!**

# Altri Tipi di variabili

Union

Literal

# Union

unione di più tipi

Sintassi:

```
function combine(input1: (number|string), input2: string|number ) {  
  const result = input1 + input2;  
  return result;  
}  
  
const combineAges = combine(20,25)  
const combineNames = combine("Ciao"," Paolo")
```

# Mhhhh

è solo un “any” ma con degli extra step?

**NO!**

```
basics-08-enums > Operator '+' cannot be applied to types 'string | number' and  
1   function c  
2     | 'string | number'. ts(2365)  
3     | Peek Problem  No quick fixes available  
4     | const result = input1 + input2;  
5     | return result;  
6   }
```

Nelle funzioni, ci aiuta a documentarle  
meglio

# In questo modo TS riesce a darci i giusti suggerimenti

```
function combine(input1: number | string, input2: number | string) {  
  let result;  
  if (typeof input1 === 'number' && typeof input2 === 'number') {  
    result = input1 + input2;  
  } else {  
    result = input1.toString() + input2.toString();  
  }  
  return result;  
}
```

E, di conseguenza, riusciamo a gestire tutte le varie casistiche delle nostre variabili

# Da tenere in considerazione

- Non sempre si riesce ad elaborare diversi tipi di variabile così semplicemente
- Dipende strettamente dalla logica che stai scrivendo
- In generale, meglio usare union types piuttosto che “any”

# Literal Types

“letteralmente il valore che ti dico”

# Non sempre si vuole un comportamento “standard” dalle funzioni che si scrivono

```
// Voglio combinare due numeri sommandoli
// voglio che ritorni un numero
const combinedAges = combine(30, 26, 'as-number');
console.log(combinedAges);

// Voglio combinare due numeri concatenandoli
// voglio che ritorni un numero
const combinedStringAges = combine('30', '26', 'as-number');
console.log(combinedStringAges);

// Voglio combinare due stringhe concatenandole
// Voglio che ritorni una stringa
const combinedNames = combine('Max', 'Anna', 'as-text');
console.log(combinedNames);
```

# Ottimizziamo il codice

```
)] {
  let result;
  ✓ if (typeof input1 === 'number' && typeof input2 === 'number' ) {
    | result = +input1 + +input2;
  ✓ } else {
    | result = input1.toString() + input2.toString();
  }
  ✓ if (resultConversion === 'as-number') {
    | return +result;
  }
  return result.toString();
}
```

Da usare, possibilmente,  
insieme agli union types

```
function combine(
  input1: number | string,
  input2: number | string,
  resultConversion: 'as-number' | 'as-text'
) {
  let result;
  if (typeof input1 === 'number' && typeof input2 === 'number' || resultConversion === 'as-number') {
    | result = +input1 + +input2;
  } else {
    | result = input1.toString() + input2.toString();
  }
  return result;
}
```

Ovviamente, possono essere usate anche nei modi più semplici, con un solo valore:

```
const A = 'B'  
// è uguale a  
const C : 'B' = 'B' ;
```

(Sarebbero solo “cosnt” con extra steps)

# Aliases / Custom types

Riutilizzare e modificare tipi già esistenti

# Esempio semplice

```
type Combinable = number;

function combine(
  input1: Combinable | string,
  input2: Combinable | string,
  resultConversion: 'as-number' | 'as-text'
) {
```

# Ok, ma così...

- Ha poco senso
- Rende il codice meno leggibile

# Ha senso usarli con, per esempio, le variabili union

```
type Combinable = number | string ;
type ConvesionDescriptior = 'as-number' | 'as-text';

function combine(
  input1: Combinable,
  input2: Combinable,
  resultConversion: ConvesionDescriptior
) {
```

# ...o con gli oggetti

```
type Person = {  
    name: string,  
    age: number,  
    hobbies: string[]  
}  
const person: Person = {  
    name: 'Maximilian',  
    age: 30,  
    hobbies: ['Sports', 'Cooking']  
};
```

ottimo per le funzioni:

```
const printName = (person:Person) => {  
    return person.name  
}
```

# Funzioni che ritornano tipi

```
function add(n1: number, n2: number): number
function add(n1: number, n2: number){
    return n1 + n2
}
```

Per rendere più severe le nostre funzioni possiamo descrivere cosa ritorneranno

```
function add(n1: number, n2: number):number{  
    return n1 + n2  
}
```

# è utile

- Per la leggibilità
- Per la documentazione
- Per la validazione

```
1 function add(n1: number, n2: number):string{  
2  
3  
4  
5  
6  
7     return n1 + n2  
8 }
```

Type 'number' is not assignable to type 'string'. ts(2322)

Peek Problem No quick fixes available

**Di default, typescript capisce  
la funzione che tipo ritorna**

**Ma non sempre lavoriamo con gli stessi  
tipi quindi, questa sintassi, può aiutarci**

# Che tipo ritorna questa funzione?

```
function add(n1: number, n2: number){  
    console.log( n1 + n2)  
}
```

# void !

```
basics-05-objec function add(n1: number, n2: number): void
1   function add(n1: number, n2: number){
2     console.log( n1 + n2)
3   }
4
```

un tipo di variabile che non esiste in JS,  
ma hanno inserito in TS

# In JS, questa funzione ritorna “undefined”

**Undefined è un tipo  
riconosciuto da TS**

```
function add(n1: number, n2: number): undefined{  
    console.log( n1 + n2)  
    return;  
}
```

**Quasi mai utilizzato**

**Ma essendo un tipo  
anche per JS, per  
ritornare un undefined  
bisognerebbe:**

```
let a:undefined;
```

**Senza senso**

# Tipo function

- Scopo descrittivo (per parametri e per cosa ritornano)
- Quando lavoriamo con le funzioni, è molto utile usare la stessa logica usata finora in TS

# Sintassi:

```
let a = Function;
```

```
let a : [(a: number, b:number)] => number
```

```
type CombineFunctions = (a:number,b:number) => number  
let a : CombineFunctions
```

# Appunti sulle callbacks

```
function addAndHandle(n1: number, n2: number, cb: (num: number) => void) {
  const result = n1 + n2;
  cb(result);
}

addAndHandle(3, 3, (result, b) => {
  console.log(result)
})
```

Il void, nell callbacks, viene ignorato (tradotto in “any”)

Ci segnala dove sbagliamo

```
function addAndHandle(n1: number, n2: number, cb: (num: number) => void) {
  const result = n1 + n2;
  cb(result);
}

addAndHandle(3, 3, (result) => {
  console.log(result)
  return result + 10
})
```

# Ultimi due tipi che possono essere utili

- Unknow
- Never

# Unknow

Come any ma:

- Ti forza a fare del type checking
- Non puoi utilizzare cose che non sai

```
let a : unknown
let b : any

let output: string

a = 5
b = 6

output = a; // No! mi devi assicurare che output sia una stringa
output = b; // OK, any può anche essere tutto...

// Questo il modo corretto usando unknown
if (typeof a === 'string'){
    output = a;
}

a.method() // No! Non sapendo niente. non possiamo chiamare un metodo
b.method() // Ok, tanto any potrebbe esserlo...
```

# Never type

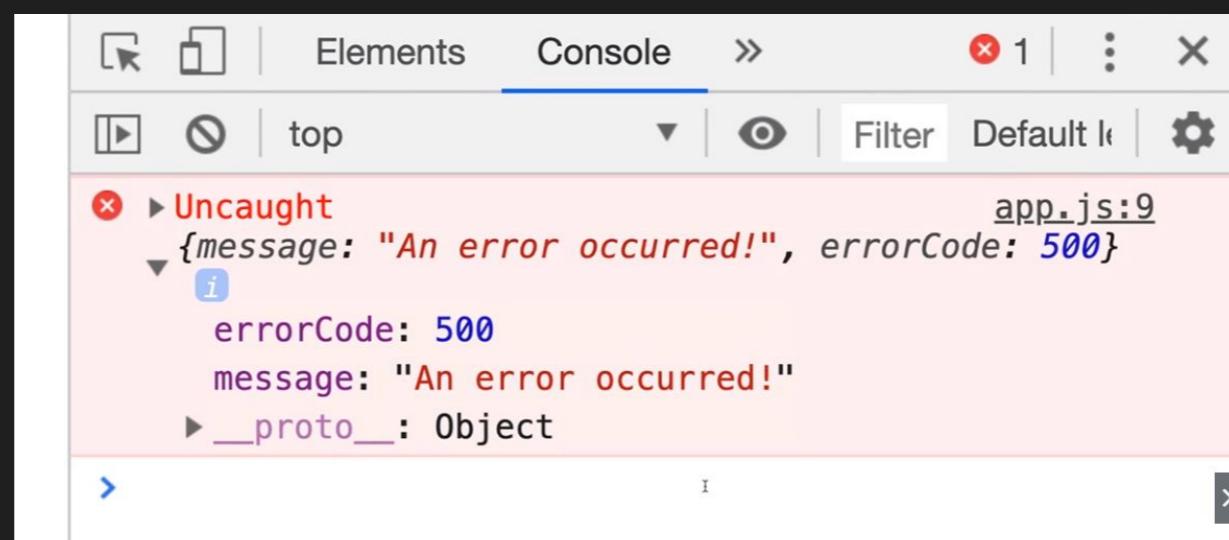
- ritornato dalle funzioni
- usato nelle funzioni che non ritornano  
niente. Mai.

```
function generateError(message: string, code: number): never {
  throw { message: message, errorCode: code };
  // while (true) {} // Altro tipo di funzione che non ritorna mai niente
}

generateError('An error occurred!', 500);

// psssst... Il corretto utilizzo della funzione qui sarebbe:

try{
  generateError('An error occurred!', 500);
} catch(e){
  // qui tornerebbe il valore che abbiamo gettato ("throw")
}
```



# Il compilatore di TS

Fino ad ora abbiamo visto le cose basice

- Configuriamo il compilatore
- Personalizziamo l'esperienza di sviluppo

# Inizializzare il progetto

```
tsc --init
```

Crea un

```
{...} tsconfig.json
```

# By default

- Eseguendo “tsc” o “tsc -w” compilerà tutti i file della cartella nel progetto
- “-w” è il watcher, così lo fa in automatico ad ogni salvataggio

# Escludere alcuni file dal progetto

```
  ...
  "exclude": [
    "fileNotToCompile.ts",
    "*.dev.ts",
    "node_modules" // Escluso di default
  ]
}
```

```
"exclude" : [
  "node_modules" // Valore di default
]
```

# Altre opzioni

- Include (funziona al contrario di exclude)
  - Files (per selezionare solo determinati files)

# Opzione target:

```
"target": "es5",
```

```
let i = 3  
console.log(3)          "use strict";  
var i = 3;  
console.log(3);
```

```
"target": "es6",
```

```
let i = 3  
console.log(3)          "use strict";  
let i = 3;  
console.log(3);
```

## Browser Support for ES5 (2009)

Browser	Version	From Date
Chrome	23	Sep 2012
Firefox	21	Apr 2013
IE	9*	Mar 2011
IE / Edge	10	Sep 2012
Safari	6	Jul 2012
Opera	15	Jul 2013

## Browser Support for ES6 (ECMAScript 2015)

Browser	Version	Date
Chrome	51	May 2016
Firefox	54	Jun 2017
Edge	14	Aug 2016
Safari	10	Sep 2016
Opera	38	Jun 2016

[https://www.w3schools.com/js/js\\_versions.asp](https://www.w3schools.com/js/js_versions.asp)

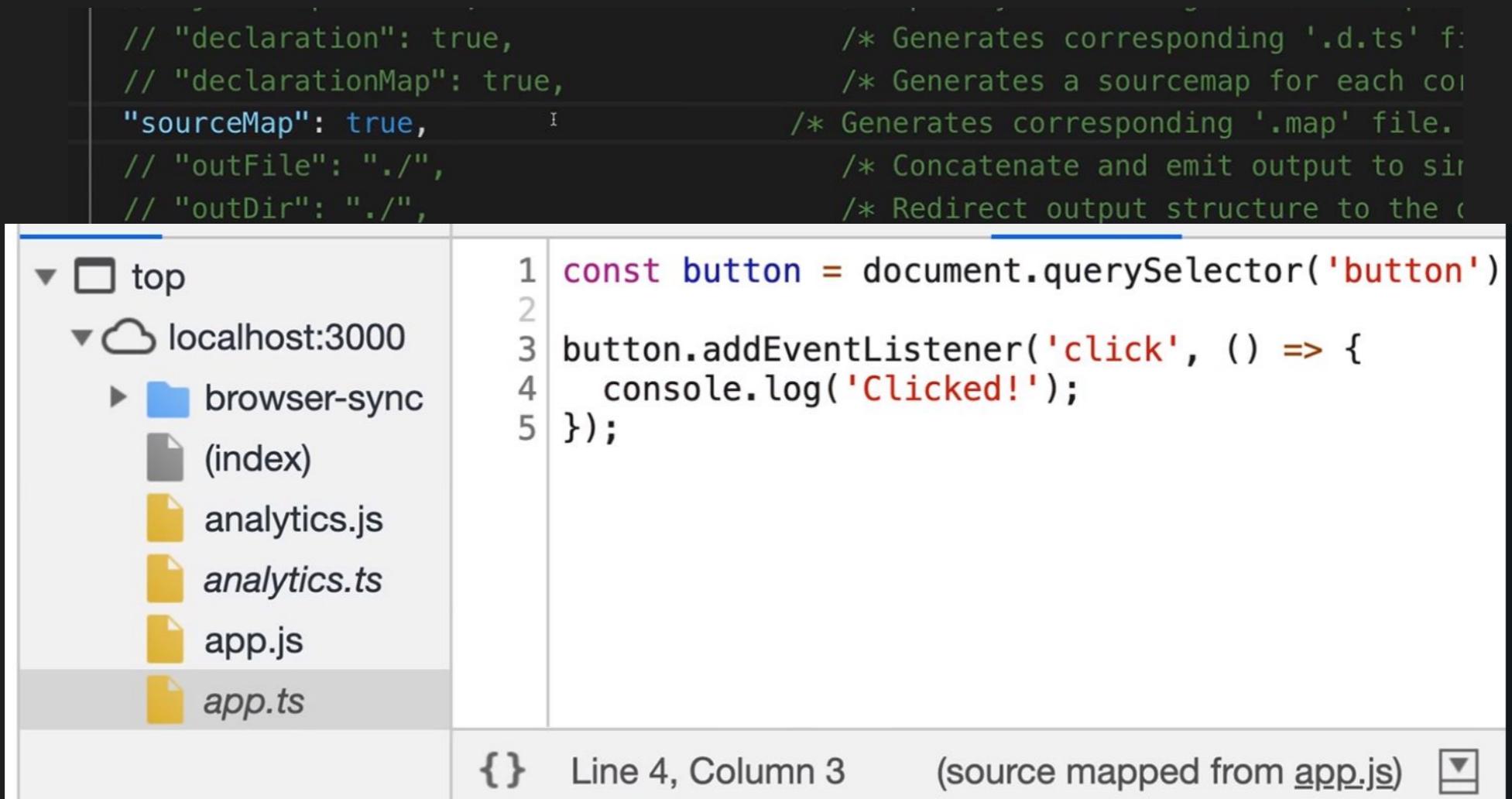
# Lib option

```
const btn = document.querySelector('.button')!;

btn.addEventListener('onclick', (ev)=>{
  console.log('clicked')
})
```

Di default, carica le librerie del target scelto.

# SourceMaps



The screenshot shows a browser developer tools interface with a sidebar and a main content area. The sidebar on the left lists files under 'localhost:3000': 'browser-sync', '(index)', 'analytics.js', 'analytics.ts', 'app.js', and 'app.ts'. The file 'app.ts' is currently selected, indicated by a grey background. The main content area displays two snippets of code. The top snippet is a configuration object with comments explaining the properties:

```
// "declaration": true,          /* Generates corresponding '.d.ts' file
// "declarationMap": true,       /* Generates a sourcemap for each compilation
"sourceMap": true,             /* Generates corresponding '.map' file.
// "outFile": "./",             /* Concatenate and emit output to single file
// "outDir": "./",              /* Redirect output structure to the directory

```

The bottom snippet is a block of JavaScript code:

```
1 const button = document.querySelector('button')
2
3 button.addEventListener('click', () => {
4   console.log('Clicked!');
5 });

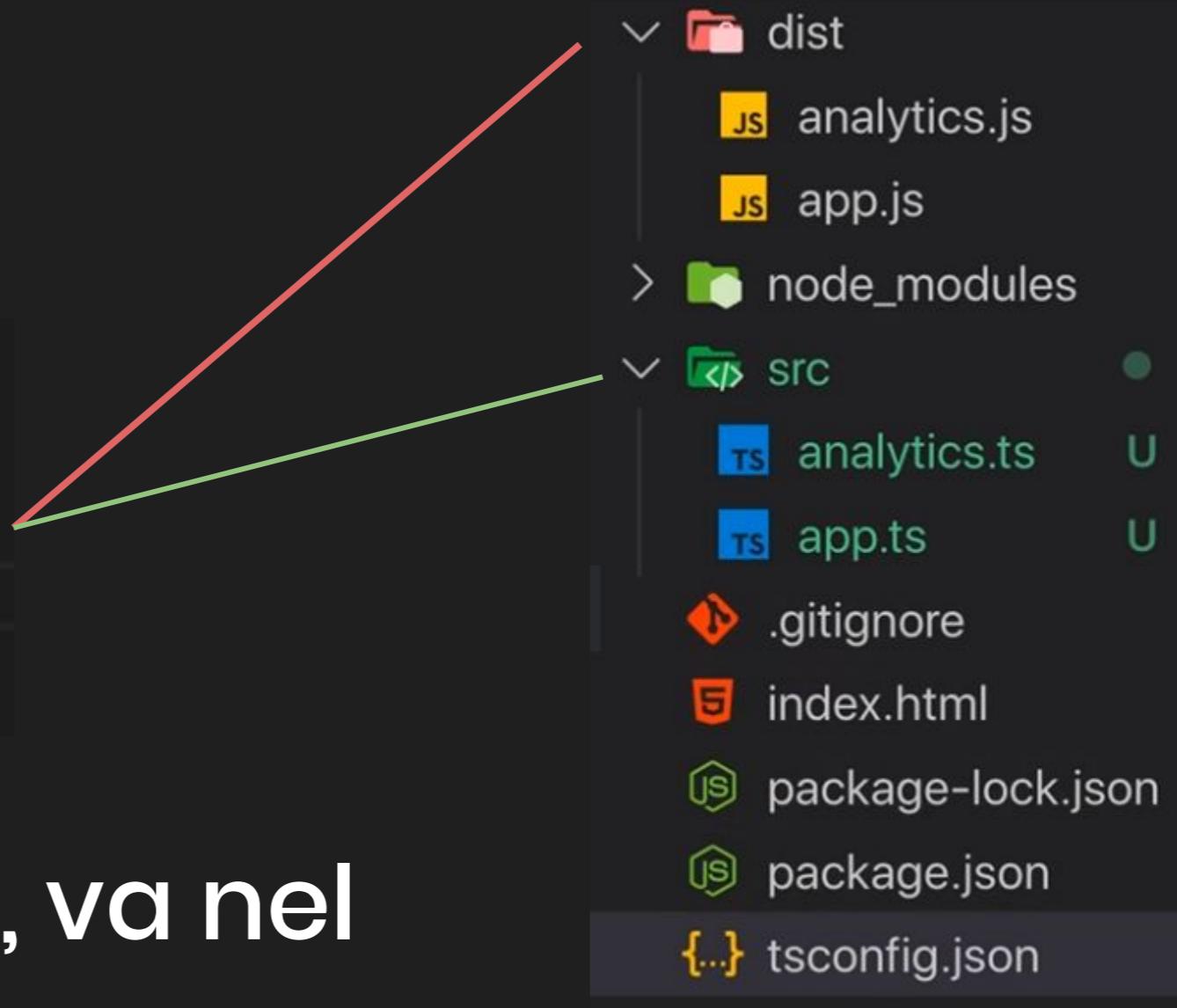
```

At the bottom of the main area, there is a status bar with the text '{ } Line 4, Column 3 (source mapped from app.js) ▾'.

Da rimuovere in produzione

# Out dir & root Dir

```
// "declaration": true,  
// "sourceMap": true,  
// "outFile": "./",  
"outDir": "./dist",  
"rootDir": "./src",  
// "composite": true,  
// "removeComments": true
```



dist, solitamente, va nel  
gitignore

# Non buildare se ci sono errori

```
// "downlevelIteration": true,  
// "isolatedModules": true,  
"noEmitOnError": true,  
  
/* Strict Type-Checking Options */  
"strict": true,
```

1. By default, compila lo stesso
2. Va aggiunto, tsconfig non ce l'ha

# Altre risorse utili

Documentazione tsconfig.json:

<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

Guida Visual Studio Code Typescript Debugger:

<https://code.visualstudio.com/docs/typescript/typescript-debugging>

# Usare le ultime funzionalità di JavaScript

TypeScript compila l'ultima versione di ECMAScript nella versione “target” che scegliamo, vediamo le funzioni più comuni

# Documentation table:

<https://kangax.github.io/compat-table/es6/>

# let & const

```
const person = "Paolo";
person = "Marco";
let age = 30;
age = 29;
```

# Perché let e non var?

*let* è inizializzata solo nello scope in cui viene usata.

Ecco la differenza:

```
if ( age > 20 ){
  let isOld = true
}
console.log(isOld) // -> Stampa undefined
```

```
if ( age > 20 ){
  var isOld = true
}
console.log(isOld) // -> Stampa true
```

Stesso vale per le funzioni

# Arrow Functions

```
// Standard syntax
const add = (a:number, b: number = 1 ) => {
    const result = a + b;
    return result
}

// Se ho solo il return da fare, posso omettere le graffe
const add2 = (a:number, b:number) => a + b

// un solo argomento, ometto le parentesi
// (TS ci segnala che vorrebbe il tipo)
const add3 = a => a + 2

// Se non ho parametri, uso la () o _ (questa la sanno in pochi)
const button = document.querySelector('button')
button?.addEventListener('click',_ => console.log('pressed'))
// oppure... (TS sa il tipo dell'argomento "event")
button?.addEventListener('click', event => console.log(event))
```

# Differenza con le funzioni standard?

```
// ES5
var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(function() {
      console.log(this.id);
    }.bind(this), 1000);
    // ho bindato (this) all'oggetto padre,
    // Altrimenti this.id sarebbe stato undefined
  }
};
```

```
// ES6
var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(() => {
      console.log(this.id);
    }, 1000);
    // Con le arrow function, non serve
  }
};
```

le arrow function  
hanno il this  
*bindato* allo  
scope che le  
contengono

# Non sempre è un bene

```
//Context dinamico, this non è quello corretto
var button = document.getElementById('press');
button?.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

```
// Tocca usare le functions classiche
// TS ovviamente, aiuta.
var button = document.getElementById('press');
button?.addEventListener('click', function() {
  this.classList.toggle('on');
});
```

# Spread operator

```
const hobbies = ['Sport','Cooking'];
const activeHobbies = ['Hiking'];

// Old way:
activeHobbies.push(hobbies[0],hobbies[1])
// New way
activeHobbies.push(...hobbies) // Tutto quello che è dentro hobbies
// La mia sintassi preferita
const myActiveHobbies = ['hiking',...hobbies]
```

# Funziona anche con gli oggetti

```
const person = {  
    name: 'Paolo',  
    age : 27,  
}  
  
const newPerson = [  
    ...person,  
    hobbies: ['Sport']  
]
```

```
const execSomething = function (options:object){  
    var opt = {  
        defaultOptions: true,  
        ...options,  
        // Le opzioni passate come parametro  
        // vinceranno su quelle di deafult  
    }  
    console.log(opt)  
}
```

# Rest Parameters

```
const sum = (...numbers: number[]) => {
  return numbers.reduce( (curResult, curValue) => {
    return curResult + curValue
  }, 0)
}

// In passato, c'era la variabile "arguments"
// Array con tutti gli argomenti di una funzione
// Ma non poteva essere tipizzata

console.log(sum(2,42,51,4))
```

# Array e oggetti destrutturati

```
const hobbies = ['Sport','Hiking'];

// old way
const hoby1= hobbies[0];
const hoby2= hobbies[2];
// new way
const [hobby1,hobby2,...remaningHobbies] = hobbies
```

```
const person = {
  firstName: 'Paolo',
  age: 27
}
// Valorize firstName variable and yearsOld Variable
const {firstName, age: yearsOld } = person
```

# String template (tick symbol)

```
(` Name of the employee: ${employee}, age: ${parseInt(Math.Random())}`)
```

Per il setup della tastiera...

<https://www.microsoft.com/en-us/download/details.aspx?id=22339>

# Class

1. Funzionalità
2. Classi e ereditarietà
3. Interface

# Class

```
class Department {  
    name: string = "DEFAULT"  
  
    constructor(n: string) {  
        // Runned at the creation  
        this.name = n  
    }  
  
    describe(){  
        console.log(this.name)  
    }  
}  
  
const itDep = new Department('IT')  
itDep.describe()
```

# **this keyword example**

```
class Department {
    name: string = "DEFAULT"

    constructor(n: string) {
        // Runned at the creation
        this.name = n
    }

    describe(this:Department){
        console.log(this.name)
    }
}

const itDep = new Department('IT')

const itDepCopy = {
    name: 'Account',
    describe: itDep.describe
}
itDepCopy.describe()
```

```
class Department {  
    name: string = "DEFAULT"  
    private employee: string[] = []  
  
    constructor(n: string) {  
        // Runned at the creation  
        this.name = n  
    }  
  
    addEmployee(employee:string) {  
        // Validation etc  
        this.employee.push(employee)  
    }  
  
    describe(this:Department){  
        console.log(this.name)  
    }  
}  
const itDep = new Department('IT')  
itDep.emp  
const itDep addEmployee (method) Dep
```

# Private and public (Solo con TS!)

Controllo al build e non in runtime

Private e public sono stati aggiunti da poco in JS ma con un'altra sintassi

# Trick per i parametri al constructor

```
class Department {  
    // name: string = "DEFAULT"  
    private employee: string[] = []  
  
    constructor(public readonly name: string = "Default") {  
        // Possiamo omettere le variabili prima di constructor  
        // E possiamo omettere le righe per valorizzare  
        // le properties delle classi  
        // passate come parametro al constructor  
        // this.name = name  
    }  
}
```

# readOnly

```
public readonly name: string = "Default")  
    amo ammattone le variabili prima di constra
```

Solo in TS, controllo alla compilazione e  
non in runtime

# extends e super()

```
class Department {
    // name: string = "DEFAULT"
    private employee: string[] = []

    constructor(public readonly id:String, name: string = "Default") {
        // Possiamo omettere le variabili prima di constructor
        // E possiamo omettere le righe per valorizzare
        // le properties delle classi
        // passate come parametro al constructor
        // this.name = name
    }
}

class ITDepartment extends Department{
    constructor(id: string){
        super(id,'IT')
    }
}
```

JS

# protected vs private

```
class Department {
    // name: string = "DEFAULT"
    private employee: string[] = []

    constructor(public readonly id:String, name: string = "Default") {
    }
}

class ITDepartment extends Department{
    construct (property) Department.employee: string[]
        super
    } Property 'employee' is private and only accessible within class
        'Department'. ts(2341)
    addITEmpl Peek Problem No quick fixes available
        this.employee.push(name)
    }
}
```

```
class Department {
    // name: string = "DEFAULT"
    protected employee: string[] = []

    constructor(public readonly id:String, name: string = "Default") {
    }
}

class ITDepartment extends Department{
    constructor(id: string){
        super(id,'IT')
    }

    addITEmployee(name:string){
        this.employee.push(name)
    }
}
```

Ovviamente come private, protected è una feature di TS

# getters

```
class Department {  
    // name: string = "DEFAULT"  
    protected employee: string[] = []  
  
    get lastEmployee(){  
        return this.employee[this.employee.length - 1]  
    }  
  
    constructor(public readonly id:String, name: string = "Default") {  
    }  
}  
  
class ITDepartment extends Department{  
    constructor(id: string, documentations: string[]){  
        super(id,'IT')  
    }  
  
    addITEmployee(name:string){  
        this.employee.push(name)  
    }  
}  
  
const itDep = new ITDepartment('it001',['Angular Docs'])  
// const lastEmployee = itDep.lastEmployee  
// Better:  
const {lastEmployee} = itDep;
```

JS

# setters

```
set lastEmployee(value:string){  
    if (!value){  
        throw new Error( "You must pass a correct value")  
    }  
    this.addEmployee(value)  
}
```

```
const itDep = new ITDepartment('it001',['Angular Docs'])  
const {lastEmployee} = itDep;  
itDep.lastEmployee = 'Marco' //set new employee
```

JS

# Static methods

```
class Department {  
    static fiscalYear = 2020;  
  
    constructor(public readonly id:string, name: string = "Default")  
        // Non posso accedere alle static da qui  
        console.log(this.fiscalYear)  
    }  
  
    static yearsFromTheAperture() {  
        // Da metodo statici,  
        // posso accedere a variabili statiche  
        return this.fiscalYear - 1980  
    }  
  
}  
  
// Method più famosi nella classe Math:  
  
Math.random()  
Math.pow(12,2)
```

JS

# abstract classes & methods

```
abstract class Department {  
    static fiscalYear = 2020;  
    abstract describe(): void;  
  
    class ItDepartment  
}  
Non-abstract class 'ItDepartment' does not implement inherited  
abstract member 'describe' from class 'Department'. ts(2515)  
    Peek Problem Quick Fix...  
class ItDepartment extends Department {  
}
```



# Le abstract non possono essere inzializzate

```
static fiscalYear = 2020;
abstract describe(): void;
constructor Department(id: string, _name: string): Department
constructor(param
}
}
Cannot create an instance of an abstract class. ts(2511)
Peek Problem No quick fixes available
const AccountDep = new Department('account-001', 'Acc Dep')
```



# Tipo interface

Custom types per le classi

# ... Oper gli oggetti

```
interface Person {  
    name: string;  
  
    greet(phrase: string): void;  
}  
  
const user : Person = {  
    name: 'Paolo',  
    greet(prashe: string){  
        console.log(` ${prashe} ${this.name}`)  
    }  
}
```

# Perché non usare type?

```
type Person = {  
    name: string;  
  
    greet(phrase: string): void;  
}  
  
const user : Person = {  
    name: 'Paolo',  
    greet(prashe: string){  
        console.log(` ${prashe} ${this.name}`)  
    }  
}
```

- Con le interface puoi solo specificare strutture di oggetti (o classi, stessa cosa)
- Con i type puoi usare qualsiasi altro tipo (union, String ecc ecc),

# Differenze tra usare interface per gli oggetti o per le classi

- Con **gli oggetti**, devo essere preciso
- Con **le classi**, di fatto “implemento” le interface, ergo si possono estendere

```
interface Greetable {  
  name: string;  
  greet(phrase: string): void;  
}  
  
const user : Greetable = [  
  name: 'Paolo',  
  age: 27,  
  greet(prashe: string){  
    console.log(`$ {prashe} ${this.name}`)  
  }  
]
```

```
class Person implements Greetable {  
  name: string;  
  age = 30;  
  
  constructor(n: string) {  
    this.name = n;  
  }  
  
  greet(phrase: string) {  
    console.log(phrase + ' ' + this.name);  
  }  
}
```

# Puoi assegnare interface e Classes come tipi di variabili

```
let user1: Person;  
user1 = new Person('Max');  
user1.greet('Hi there - I am');  
console.log(user1);
```

```
let user1: Greetable;  
user1 = new Person('Max');  
user1.greet('Hi there - I am');  
console.log(user1);
```

# Funzionalità

- Non si può definire cosa sarà public o private
- Si può definire cosa sarà readonly

```
interface Greetable {  
    name: string;  
    // private age: number; non possibile  
    readonly id : string;  
    greet(phrase: string): void;  
}
```

# Definendo il readonly nelle interface, lo assegniamo di default alla classe

```
class Person implements Greetable {
    name: string;
    age = 27;
    id = '001';

    constructor(n: string) {
        this.name = n;
    }
    greet(phrase: string) {
        console.log(` ${phrase} ${this.name}`);
    }
    (property) Greetable.id: string
}
let user1 = new Person('John');
user1.id = '002';
Cannot assign to 'id' because it is a read-only
property. ts(2540)
user1.id = '002' No quick fixes available
```

# Ereditarietà delle interface

## 1° metodo

```
interface Named {
  name: string;
}

interface Greetable {
  // private age: number; non possibile, pro
  readonly id : string;
  greet(phrase: string): void;
}

class Person implements Greetable,Named {
```

## 2° metodo

```
interface Named {
  name: string;
  age:number,
}

interface Greetable extends Named {
  // private age: number; non possibile, pro
  readonly id : string;
  greet(phrase: string): void;
}

class Person implements Greetable {
```

- Somma le interface,
- in caso di attributi con tipi diversi, TS segna un errore durante l'utilizzo dell'attributo dentro la classe

- Le interface estendono altre Interface
- in caso di attributi con tipi diversi, TS segnala un errore durante l'utilizzo dell'attributo dentro l'interface figlia
- Si possono avere diverse interface separate dalla virgola, proprio come il primo metodo

**Strutturalmente, meglio il  
secondo metodo**

# In JS, le funzioni sono degli oggetti

Quindi, qualche matto usa le interface per definire le funzioni

```
interface addFn {  
    (a: number, b:number): number;  
}  
type addFnWithType = (a:number,b:number) => number
```

# Attributi opzionali

Semplicemente aggiungendo “?”  
nell’attributo.

```
interface Named {  
  name: string;  
  age?: number;  
}
```

In questo modo, anche se l’attributo è  
opzionale, sarà tipizzato

# Questa sintassi, si può usare anche nelle classi, e in generale, ovunque.

```
class Person implements Greetable {  
    name?: string;  
    age = 27;  
    id = '001';  
  
    constructor(n?: string) {  
        if (n){  
            this.name = n;  
        }  
    }  
}
```

Bisogna però essere consistenti, non si può metterla opzionale nell'interface e non nella classe

# Ricordarsi che:

JS non sa niente delle interface! Serve solo agli sviluppatori. In runtime, non vengono nemmeno compilate.



# Tipi avanzati

Intersection

Discriminated Unions

Type Guards

Type Casting

Function Overloads

# intersection (&)

```
type Admin = {  
    name: string;  
    privileges: string[];  
};  
  
type Employee = {  
    name: string;  
    startDate: Date;  
};  
type ElevatedEmployee = Admin & Employee;
```

# intersection (&)

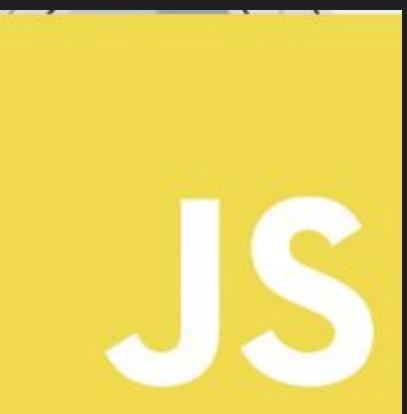
- Funziona come l'ereditarietà per le interface
  - Può essere usato tra due union
  - Se l'intersezione ritorna nessun tipo, allora il tipo sarà “never”

```
type Combinable = string | number;
type Numeric = number | boolean;
| type Universal = number
type Universal = Combinable & Numeric;
```

```
type NeverType = never
type NeverType = string & number;
```

# Type Guards

- `typeof`
- *property* in object
- instance of



# typeof

```
function add(a: Combinable, b: Combinable) {  
  if (typeof a === 'string' || typeof b === 'string')  
  {  
    return a.toString() + b.toString();  
  }  
  return a + b;  
}
```



Ritorna il tipo JS di una variabile in quel momento

“string” | “object” | “number” | “function” | “boolean” |

```
type Admin = {  
    name: string;  
    privileges: string[];  
};  
  
type Employee = {  
    name: string;  
    startDate: Date;  
};  
  
type UnknownEmployee = Employee | Admin;  
  
function printEmployeeInformation(emp: UnknownEmployee) {  
    console.log('Name: ' + emp.name);  
    if ('privileges' in emp) {  
        console.log('Privileges: ' + emp.privileges);  
    }  
    if ('startDate' in emp) {  
        console.log('Start Date: ' + emp.startDate);  
    }  
}
```

**property in  
object**  
*Il vecchio  
(item.property !== undefined)*

Utile quando lavoriamo con  
oggetti e union Types



```
class Truck {  
  drive() {  
    console.log('Driving a truck...');  
  }  
  
  loadCargo(amount: number) {  
    console.log('Loading cargo ...' + amount);  
  }  
}  
  
type Vehicle = Car | Truck;  
  
const v1 = new Car();  
const v2 = new Truck();  
  
function useVehicle(vehicle: Vehicle) {  
  vehicle.drive();  
  // if ( 'loadCargo' in vehicle ) è buono lo stesso  
  if (vehicle instanceof Truck) {  
    vehicle.loadCargo(1000);  
  }  
}  
useVehicle(v2);
```

**instanceOf**

ritorna un  
Booleano,

ci dice se la  
**variabile fa parte**  
di quella specifica  
classe



# Utile soprattutto per le classi native

```
const date = new Date()  
if (date instanceof Date){  
    // yes  
}
```

# Discriminated unions

`instanceOf` ma con i tipi di TypeScript



# step 0 - Realizzare che *instanceof* funziona solo con le Classi

```
interface Bird {  
    type: 'bird';  
    flyingSpeed: number;  
}  
  
interface Horse {  
    type: 'horse';  
    runningSpeed: number;  
}  
  
type Animal = Bird | Horse;  
  
const animal : Animal = {type:'bird',flyingSpeed:10}  
  
if (animal instanceof Bird){  
}  
}
```

# Step 1 - aggiungiamo un literal type alle nostre interface

```
interface Bird {  
    type: 'bird';  
    flyingSpeed: number;  
}
```

```
interface Horse {  
    type: 'horse';  
    runningSpeed: number;  
}
```

# Step 2 - controlliamo il literal type dove ci serve

```
function moveAnimal(animal: Animal) {  
    let speed;  
    switch (animal.type) {  
        case 'bird':  
            speed = animal.flyingSpeed;  
            break;  
        case 'horse':  
            speed = animal.runningSpeed;  
    }  
    console.log('Moving at speed: ' + speed);  
}
```

# Step 3 - TypeScript ci aiuterà a chiamare le funzioni

```
switch (animal.type) {
    case 'bird':
        speed = animal (property) Horse.runningSpeed: number
        break;
    case 'horse':
        speed = animal } is not assignable to parameter of type 'Animal'.
    }
    console.log('Movin' Object literal may only specify known properties, and
}                                         'runningSpeed' does not exist in type 'Bird'. ts(2345)
                                         Peek Problem  No quick fixes available

moveAnimal({type: 'bird', flyingSpeed: 10});
moveAnimal({type: 'bird', runningSpeed: 10});
```

# Perché sono utili e perché sono la soluzione migliore

- Mantenere le Classi più pulite,dato che le Interface non vengono usate in runtime
  - Mantenere la sintassi di TypeScript costante in tutta la soluzione
  - Avere suggerimenti logici da TypeScript

# Type Casting

Utilizzare le librerie caricare da TypeScript

# Le librerie caricare da TypeScript sono molto intelligenti (e molto utili)

```
// Specificando il tipo di elemento html
// TypeScript sa che possiamo accedere a "value"
const firstInputElement = document.querySelector('input')!;
const {value : val} = firstInputElement;
```

# Non sempre sappiamo il tipo di elemento HTML

```
const secondInputElement = document.querySelector('#input');
                                ^any
Property 'value' does not exist on type 'Element | null'. ts(2339)
Peek Problem  No quick fixes available
const {value : val2} = secondInputElement;
```

I casting type servono a specificare le funzionalità di un specifico elemento della libreria

```
const userInputElement =  
<HTMLInputElement>document.getElementById('user-input')!;
```

# 2 tipi di sintassi possibili:

```
const userInputElement1 =  
<HTMLInputElement>document.getElementById('user-input')!;
```

```
const userInputElement2 = document.getElementById('user-input') as  
HTMLInputElement;
```

# Il type casting può essere usato nelle funzioni

```
function getValue({value: val}:HTMLInputElement){  
    console.log(val)  
}
```

# Sintassi per il controllo

```
const userInputElement3 = document.getElementById('user-input')!;  
if (userInputElement3) {  
  (userInputElement3 as HTMLInputElement).value = 'Hi there!';  
}
```

- Utile, in questo caso, quando si lavora con il DOM
- Come vedremo, sarà utile con tutte le librerie scritte (bene) in TypeScript
- Possiamo far sì che le nostre librerie siano utili agli altri sviluppatori\*
- Autocomplete in VSC e negli altri IDE

# Punto Esclamativo dopo l'assegnazione delle variabili

```
const firstInputElement = document.querySelector('input')!;
```

- Per comunicare a TS che quell'elemento non sarà mai null

# Index properties

```
interface ErrorContainer {  
  id: string,  
  // I soli tipi accettati sono  
  // number & string  
  [prop: string]: string;  
}  
  
const errorBag: ErrorContainer = {  
  id: 'form-newsletter-error',  
  email: 'Not a valid email!',  
  // Se specifico string come tipo,  
  // Posso usare anche number (ma non viceversa)  
  2: 'Not a valid email!',  
  username: 'Must start with a capital character!'  
};
```

Per quando  
lavoriamo con degli  
oggetti incerti

# function overloads

Cercare di essere più specifici possibili con TS ci permette di:

- Essere più leggibili
- Commettere meno errori
- Farci aiutare il più possibile con l'autocomplete e le altre funzionalità

# In questa funzione, TypeScript non è certo su che tipo sarà ritornato

```
function add(a: Combinable, b: Combinable) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}

const result = add('Max', ' Schwarz');

any
Property 'split' does not exist on type 'Combinable'.
  Property 'split' does not exist on type 'number'. ts(2339)
Peek Problem  No quick fixes available
result.split(' ');
```

# le function overloads chiariscono le varie possibilità

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: string, b: number): string;
function add(a: number, b: string): string;
function add(a: Combinable, b: Combinable) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}

const result = add('Max', ' Schwarz');
result.split(' ');
```

# Optional chaining ovvero, “?”

Assumiamo di ricevere dati non tipizzati correttamente in TS

```
const fetchedUserData = {  
  id: 'u1',  
  name: 'Max',  
  job: { title: 'CEO', description: 'My own company' }  
};
```

# Con JS faremmo così:

```
console.log(fetchedUserData && fetchedUserData.job.title);
```

# Con TS facciamo così:

```
console.log(fetchedUserData?.job?.title);
```

*Possiamo farlo in automatico per tutte le  
sub-properties!*

# Generics

iniziamo ad usare TypeScript in maniera professionale.

# Strutture complesse, richiedono maggiori dettagli di tipizzazione

```
const names: Array<string> = []; // string[]
names[0].split(' ');
```

(Esempio semplice)

```
const names: Array<string> = []; // string[]
names[0].sp
    ⚭ split  (method) String.split(se
```

# Esempio:

## Le chiamate asincrone in JS sono gestite da delle Promise (promesse)

```
const promise: Promise<string> = new  
Promise((resolve) => {  
    setTimeout(() => {  
        resolve('Ciao');  
    }, 2000);  
});  
  
promise.then(data => {  
    data.split(' ');  
})
```

Possiamo usare la sintassi dei Generics per gestire quello che viene tornato

# Uso per un fetch http

```
interface Car {  
    model: string,  
    licensePlate: string,  
    daysLog: number,  
    maxSpeed: number,  
    kmsToNextCharge: (speed:number, kWh: number) => number  
}
```

```
const responseCarInfo: Promise<Car>  
= axios  
    .get('/car-info/sdf9876ds')
```

# Generic function

```
function merge<T extends object, U extends object>(objA: T, objB: U)  
{  
    return Object.assign(objA, objB);  
}  
  
const mergedObj = merge({ name: 'Paolo ', hobbies: ['Sports'] }, {  
age: 30 });
```

Possiamo estendere  
anche con interface

**Uniamo tutto insieme**

# Decorators

- Cosa sono
  - Sintatti
- Come vengono utilizzati dai framework moderni
- Esempio di utilizzo reale

# Cosa sono

Funzioni ausiliarie da eseguire prima di determinati eventi nelle classi

“uncini” con il quale gli altri sviluppatori possono alterare, controllare, gestire le classi

# Sintassi

# Come possiamo usarli in altri framework (come angular)

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
  `,
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

# I decorators

Vengono eseguiti sequenzialmente

# Possiamo usare i decorator anche quando:

- Inizializziamo una classe  
(standard decorator)
- Inizializziamo una property di una classe  
(Property decorator)
- Inizializziamo una funzione di una class  
(Method decorator)
  - Prima di un getter o un setter  
(access decorator)

# Ciclo di vita:

# Manipolare la classe

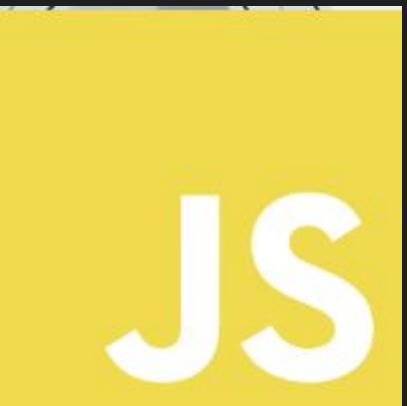
```
function WithTemplate(template: string, hookId: string) {  
    console.log('TEMPLATE FACTORY');  
    return function<T extends { new (...args: any[]): { name: string } }>(  
        originalConstructor: T  
    ) {  
        return class extends originalConstructor {  
            constructor(..._: any[]) {  
                super();  
                console.log('Rendering template');  
                const hookEl = document.getElementById(hookId);  
                if (hookEl) {  
                    hookEl.innerHTML = template;  
                    hookEl.querySelector('h1')!.textContent = this.name;  
                }  
            }  
        };  
    };  
}
```

# Possibilità di utilizzo

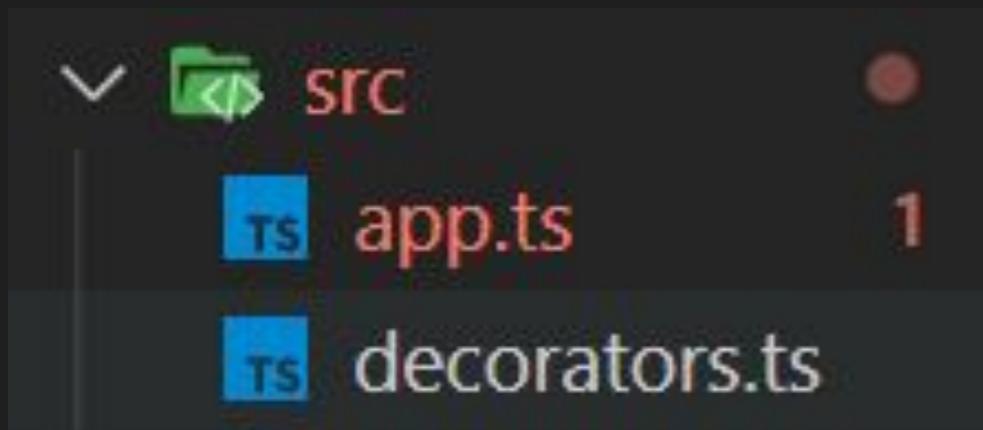
- **bindare una classe con un elemento del DOM**
  - Eseguire logiche di validazione
  - Eseguire logiche di tipizzazione
    - Sky is the limit!

# MODULES e namespaces

- come organizzare i file
  - Import e export



# Come organizzare i file



Esempio:

- Inseriamo una libreria
- Spostiamo tutti i decorator su un altro file.

# decorator.ts

```
export const Logger = function(logString: string) {[...]}  
export const WithTemplate = function(template: string, hookId: string) {[...]}  
export const Log = function(target: any, propertyName: string | Symbol) {[...]}  
export const Log2= function(target: any, name: string, descriptor: PropertyDescriptor) {[..]}  
export const Log3= function(  
    target: any,  
    name: string | Symbol,  
    descriptor: PropertyDescriptor  
) {[...]}  
export const Log4= function(target: any, name: string | Symbol, position: number) {[...]}
```

# App.ts

```
import "moment"
import {Logger, WithTemplate, Log, Log2, Log3, Log4} from './decorators'

@Logger('LOGGING')
@WithTemplate('<h1>My Person Object</h1>', 'app')
```

da una parte esporto nel file principale  
importo

# Altra possibile sintassi

```
import "moment"
import * as decorators from './decorators'

@decorators.Logger('LOGGING')
@decorators.WithTemplate('<h1>My Person Object</h1>', 'app')
```

importo tutti gli export e li inserisco dentro  
un oggetto.

# Il più utilizzato di tutti: Export default

L'utilità di organizzare i file,  
è far sì che essi svolgano una sola attività  
principale.

# Esempio. Spostiamo la classe Person in un file separato:

```
import * as decorators from './decorators'

@decorators.Logger('LOGGING')
@decorators.WithTemplate('<h1>My Person Object</h1>', 'app')
class Person {
    name = 'Paolo';

    constructor() {
        console.log('Creating person object...');

    }
}

export default Person
```

# Ora possiamo importarlo semplicemente con

```
import Person from './Person'
```

```
const pers = new Person();
```

# Caratteristiche

- Se un file viene importato da più file (esempio: decorators.ts), nel codice compilato esso viene importato una sola volta, la prima. E poi riutilizzato
- L'idea è quella di avere un unico file da compilare, e molti file che fungano da dipendenze.

**2 brutte notizie.**

1

Avere molti file in produzione  
però Abbassa le performance

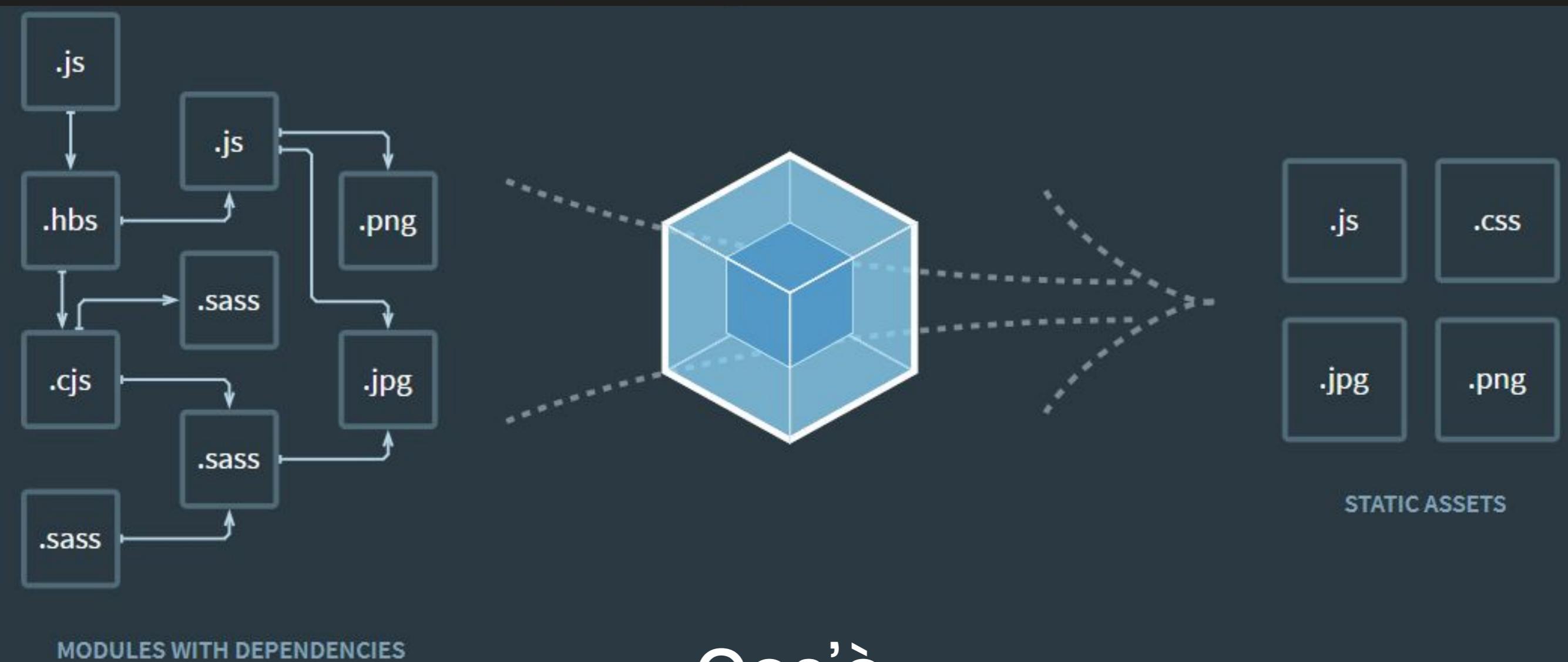
2

Funziona solo con il target  
“es6”  
(altrimenti es5 non sa come usare gli export)

# webpack!

- Utilizza la stessa sintassi dei modules
- Gestisce non solo i file js, ma tutti gli assets (img,css, scss etc etc)
- Ci permette di customizzare l'output finale

# Introduzione a webpack



- Cos'è
- Come funziona
- Perché è diventato uno standard.

# Cos'è

## bundler di assets configurabile in JavaScript

**Setup senza webpack**

Creare nuovi file “pesa”  
sulla soluzione finale

Codice non ottimizzato

**Setup con webpack**

Non ci preoccupiamo del  
numero di file

Codice ottimizzato,  
minifizzato

# Come funziona

3 dipendenze principali per gli sviluppatori:

1. webpack
  2. webpack-cli
  3. webpack-dev-server
- + Tutte le dipendenze per la tua soluzione!  
(TS, SCSS, img optimizer ecc ecc)

# Come funziona

Ogni dipendenza, funziona a sé, webpack  
si occupa di:

Eseguirla quando necessario

Unirla agli altri file una volta eseguita

**Questo significa**

**il tsconfig.json è ancora valido,**

**Non siamo obbligati a tenere il target “es6”  
(è webpack a gestire i moduli)**

**Se una libreria ha delle funzionalità da  
aggiungere per lo sviluppatore, può farlo  
(es. Angular CLI, Vue.js Cli ecc ecc)**

# File di configurazione base

`webpack.config.js`

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: './src/app.ts',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    publicPath: 'dist'
  },
  devtool: 'inline-source-map',
  module: {
    rules: [
      {
        test: /\.ts$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js']
  }
};
```

# Esempio base.

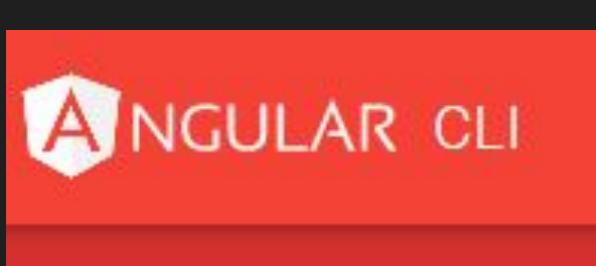
# webpack.config.prod.js

```
const path = require('path');
const CleanPlugin = require('clean-webpack-plugin');

module.exports = {
  mode: 'production',
  entry: './src/app.ts',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  devtool: 'none',
  module: {
    rules: [
      {
        test: /\.ts$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js']
  },
  plugins: [
    new CleanPlugin.CleanWebpackPlugin()
  ]
};
```

# Sapere come configurare il webpack.config.js è utile.

Ma non lo userete mai.



```
npm install -g cli-react
```



# Ogni bundle, come TypeScript

- Ha il suo file di configurazione
  - i suoi metodi di utilizzo
- e, di conseguenza, un boilerplate standard di utilizzo.

# Usare librerie esterne con TypeScript

- Librerie “normali” e come usarle con TS
- Librerie scritte in TS e come integrarle

# Aggiungere una libreria “normale”

1° Step (da terminale):

**npm i -s lodash\***

(modifica del package.json in automatico)

2° step:

`import _ from 'lodash';`

3° step:

Utilizziamo la libreria.

# Ok, però...

- Che fine ha fatto la tipizzazione?
- Come possiamo controllare i suoi metodi?

**Di default, non si può fare  
niente.**

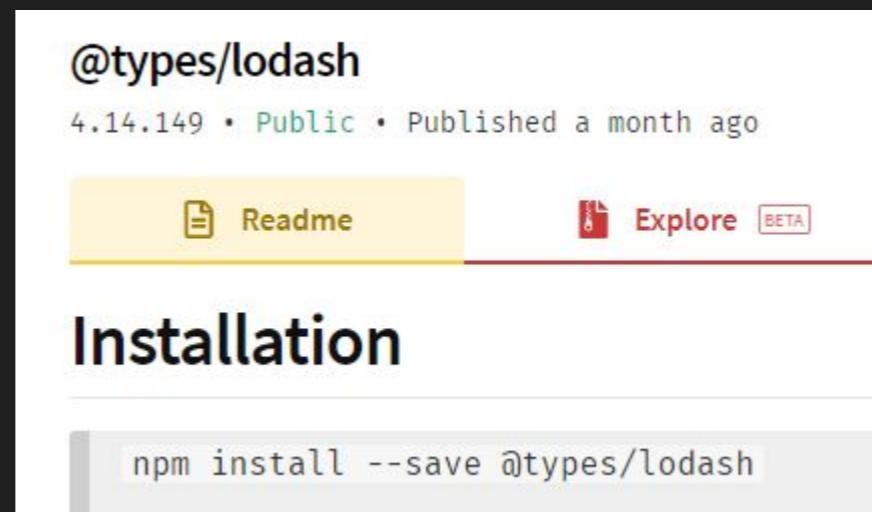
Per fortuna esiste...

**DefinitelyTyped**

<https://github.com/DefinitelyTyped/DefinitelyTyped>

**Repository centralizzato  
(In continuo aggiornamento)**  
Con tutte le librerie più  
utilizzate  
tradotte da JS a TS

# La soluzione quindi è...



```
npm install --save @types/lodash
```

# I vantaggi di questa implementazione:

1. L'autocomplete documentato con tutte le funzioni di un pacchetto
2. TypeScript conosce Custom Types, Interface, classes e tutto il resto di quella libreria

# Possono ancora esserci delle eccezioni

- Codice JS in altri file all'interno della nostra applicazione
- Libreria che , sfortunatamente, non è stata tradotta.

# Declare

```
declare var GLOBAL : any;
```

*any* ci torna d'aiuto,  
TS non ci chiederà più niente su quella  
variabile e possiamo usarla in pace

# npm package che può tornare utile:

## 1. class-transformer

(ci evita di fare il map e in automatico transforma gli oggetti ricevuti dal server in classi nostre)

## 2. class-validator

(ci aiuta a creare Decorators di validazione)