TYPESCRIPT

JAVASCRIPT È TYPESCRIPT

(valido)

- Typescript è un *superset* di javascript
- Codice javascript corretto = codice typescript corretto
- Typescript aggiunge *static type checking* a javascript

- estensione dei file javascript: *.js* / *.jsx*
- estensione dei file typescript: *.ts* / *.tsx*

```
npm install typescript
```

TYPESCRIPT NON VIENE ESEGUITO DIRETTAMENTE

```
tsc hello.ts
```

Genera un file `hello.js` che può essere eseguito in un browser o in Node.js (o altro ambiente javascript).

TSC: TYPESCRIPT COMPILER

Durante il “transpiling” di un file, typescript controlla che il codice sia coerente e riporta errori e warning se trova qualche errore.

TIPI SEMPLICI

```
const nome: string = "Mario";
const today: Date = new Date();
const eta: number = 23;
const presente: boolean = false;
const classe: "mouse" | "samba" = "mouse"; // string literal u
const preferenze: string[] = ["react", "htnl"] // array di str
```

NULL E UNDEFINED

- `null`: è un valore, di tipo nullo
- `undefined`: non è un valore, viene restituito quando una variabile non è presente

NULL

```
const nickName: string | null = null;
```

UNDEFINED

```
type Person = {
  name: string,
  nickName: string | undefined // same as: nickName?: string
}
```

ANY E UNKNOWN

```
function f1(a: any) {
  a.b(); // OK
}

function f2(a: unknown) {
  a.b();
// error: 'a' is of type 'unknown'.
}
```

Entrambi i tipi accettano qualsiasi cosa, ma con *unknown* non si può fare nulla

USO DI UNKNOWN

```
interface Box {
  contents: unknown;
}

let x: Box = {
  contents: "hello world", // accetta qualsiasi tipo
};
```

TIPI MENO SEMPLICI

TUPLE

Un array di cui è conosciuto il numero di elementi e il tipo di ciascuno di essi

```
type SetStateFn = (v: string) => string;
// semplificazione, in realtà il tipo di React è più complesso

const [state, setState]: [string, SetStateFn] = useState("init
```

UNION TYPES

```
const dataDiNascita: string | Date = "10/03/2001";
// potrebbe anche essere: new Date(2001, 3, 10);

const id: string | number = "15"; // stringa
// sarebbe valido usare anche il numero 15

const preferenze: string | string[] = "giallo";
// anche ["giallo", "blu"] è valido
```

UNION TYPES DI STRINGHE

```
const alignment: "left" | "center" | "right" = "center";
```

Utile quando il valore stringa deve essere ridotto a un set finito di opzioni

GENERICS

```
interface Box<Type> {
  contents: Type;
}

let box1: Box<string> = { contents: "hello world" };
let box2: Box<number> = { contents: 20 };
```

Invece di usare *unknown*, usiamo un tipo generico che accetta un parametro per descrivere una proprietà interna

OGGETTI

```
const coordinates: {
  latitude: number,
  longitude: number
} = {
  latitude: 0.543,
  longitude: 10.345
}
```

OGGETTI

```
const coordinates: {
  latitude: number,
  longitude: number
} = {
  latitude: 0.543,
  longitude: 10.345
}
```

OGGETTI

```
const coordinates: {
  latitude: number,
  longitude: number
} = {
  latitude: 0.543,
  longitude: 10.345
}
```

TYPE ALIASES

Un modo più leggibile per rappresentare l'oggetto precedente:

```
type Coord = {latitude: number, longitude: number};

const coordinates: Coord = {
  latitude: 0.543,
  longitude: 10.345
}
```

PROPRIETÀ OPZIONALI

```
type Person = {
  firstName: string,
  lastName: string,
  nickName?: string // questo parametro è opzionale (può esser
}
```

INTERSECTION TYPES

```
type Person = {
  firstName: string,
  lastName: string,
}

type Student = Person & {
  school: string
}

const Mario: Student = {
  firstName: "Mario",
  lastName: "Rossi",
  school: "ITS"
}
```

FUNZIONI: TIPIZZAZIONE DEI PARAMETRI

```
// Parameter type annotation
function greet(name: string) {
  console.log("Hello, " + name.toUpperCase() + "!!");
}
```

FUNZIONI: RETURN TYPE

```
function getFavoriteNumber(): number {
  return 26;
}
```

FUNZIONI ASINCRONE

```
async function getFavoriteNumber(): Promise<number> {
  return 26;
}
```

ARROW FUNCTIONS

```
type MultiplyFn = (a: number, b: number) => number;

const multiply: MultiplyFn = (a, b) => {
  return a * b;
}

const result = multiply(2, 3); // result is a number
```

GENERICS

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}

// s is of type 'string'
const s = firstElement(["a", "b", "c"]);

// n is of type 'number'
const n = firstElement([1, 2, 3]);

// u is of type undefined
const u = firstElement([]);
```

Il tipo di valore ritornato non è noto a priori, dipende dall’input

REST PARAMETERS & ARGUMENTS

```
function multiply(n: number, ...m: number[]) {
  return m.map((x) => n * x);
}

const a = multiply(10, 2, 3, 4, 5); // n = 10; m = [2, 3, 4, 5
// 'a' = [20, 30, 40, 50]
```

TIPI DI REACT

```
npm install —D @types/react @types/react-dom
```

React non contiene le definizioni dei tipi, devono essere installate a parte

COMPONENT

```tsx
type AppProps = {
  greet: string
}

const App: React.FunctionComponent<AppProps> = ({ greet })
  return <div>Hello {greet}</div>
}

// Il tipo FunctionComponent ha un alias FC,
// che viene usato più di frequente:
const App = React.FC<AppProps> = (...) => {...}
```

STATE

```
const [user, setUser] = useState<User | null>(null);
```

CONTEXT

```tsx
import { createContext } from "react";

type ThemeContextType = "light" | "dark";

const ThemeContext = createContext<ThemeContextType>("light");
```

DOM ELEMENTS

```
const divRef = useRef<HTMLDivElement>(null);
```

DOM EVENTS

```
function handleChange(event: React.ChangeEvent<HTMLInputElemen
  setValue(event.currentTarget.value);
}
```

CHILDREN

```
type ModalRendererProps = {
  title: string;
  children: React.ReactNode; // accetta copmonenti, stringhe
}
```

```
type ModalRendererProps = {
  title: string;
  children: React.ReactElement; // accetta solo componenti R
}
```

STYLE PROPS

```
type MyComponentProps = {
  style: React.CSSProperties;
}
```

APPROFONDISCI

https://react.dev/learn/typescript