Typescript

Typescript è un linguaggio di programmazione open source sviluppato da Microsoft. Nello specifico è un superset di Javascript al quale va ad aggiungere tipi, interfacce e moduli. Può essere considerato come un'estensione di Javascript.
Uno dei superpoteri di Typescript è sicuramente lo Static Type Checker. Grazie ad esso è possibile individuare errori di tipo (tipo di dato) ancor prima di avviare il programma, esempio:

```
const rectangleData = {width: 20, height: 7};
const area = rectangleData.width * rectangleData.heigt;
```

Questo codice darebbe una scritta di errore dicendo che non esiste il tipo "heigt" nel tipo "{width:20, height:7}" e potrebbe anche suggerirti il tipo di dato che stavi cercando.

Se per esempio vogliamo specificare una variabile di tipo numero, in Javascript non potremmo farlo, al contrario di Typescript in cui invece è incentivato. Ciò farà anche in modo di impedire a quella stessa variabile di essere riassegnata con un valore che non sia un numero, esempio:

```
// JavaScript
let num = 42
num = "Faccio quello che voglio 🤓"
```

```
// Typescript
let num: number = 42
num = "NON faccio quello che voglio 😢"
```

Nel secondo caso, ancora prima di avviare il programma vedremmo questo errore nella console:
Type 'string' is not assignable to type 'number'.

Ma come si fa funzionare un file Typescript? Risposta breve: non si fa.

Risposta lunga: Typescript ha un compiler che compila il codice da Typescript a Javascript, sarà poi quest'ultimo ad essere avviato nei soliti modi (esempio con node.js o per una pagina web con un index.html), questo perchè NON si può usare direttamente Typescript negli ambienti di runtime.
Il compiler non fa altro che cancellare tutta la tipizzazione statica aggiunta da Typescript e convertire tutto in un Javascript già fatto e finito che basterà solamente avviare.
Per fare ciò solitamente da riga di comando si scrive:

```
tsc nomeFile.ts
```

e premendo invio avviene la magia.

Tornando alla tipizzazione statica, Typescript mette a disposizione oltre ai tipi primitivi di Javascript (number, string, boolean, bigint, symbol, undefined, null):

- any (permetti qualunque cosa, come il normale Js)
- unknown (fa in modo che chi userà questo tipo di dato, dichiarerà che dato sia)
- never (un tipo di dato che NON può accadere)
- void (una funzione che ritorna undefined o non ha alcun return)

Grazie a Typescript si possono anche creare tipi di dato complessi, come le Unions, i Generics e le Tuple:

- Con le Unions è possibile dichiarare che un tipo di dato possa essere uno tra più tipi, esempio:

```
type doorStates = "open" | "closed"
```

Le Unions danno modo di gestire anche tipi di dati differenti, esempio quando abbiamo una funzione a cui dobbiamo passare un array (composto solo di stringhe grazie a Typescript) o una stringa:

```
function getDimension(house: string | string[]) {
 return house.length;
}
```

- Con le Tuple possiamo creare un tipo di dato che contenga due tipi (uguali o diversi), esempio:

```
let coordinates: [string, string];
coordinates = ["124143251", "23432545"]
```

Se al posto di una delle due coordinate avessi messo un dato ti dipo diverso da string, mi avrebbe dato errore poichè nella dichiarazione della tupla abbiamo definito che entrambi i dati debbano essere di tipo string.

- Con le Generics possiamo dare delle variabili ai tipi di dato:

```
function doSomething<T>(arg: T): T {
  return arg
}

doSomething<string>("ciao")
doSomething<number>(42)
doSomething<Array<number>>([12, 32, 42, 33]);
```

Nella funzione di esempio qui sopra, il T dentro al “diamante” funziona da placeholder, che andrà poi sostituito da chi farà le invocazioni della funzione, potendo dunque mettere uno dei tipi validi in Typescript.

Mentre nell’esempio sotto vediamo come anche in un'interfaccia possa essere usato il Generic per definire il comportamento di quest’ultima e lasciare a chi la invoca la scelta del tipo.
Ciò ci permette di ridurre la possibilità di introdurre bug grazie ai nuovi vincoli aggiunti da Typescript.

```
interface Basket<T> {
  add: (obj: T) => void;
```

```
  get: () => T;
  remove: (obj: T) => obj.name
}



// shortcut per dire a Ts che c'è una costante di
// nome basket (che al suo interno contiene una
interfaccia tipo di dato)
declare const basket = Basket<string>
// armadilloPeluche risulterà un string
const armadilloPeluche = basket.get()

// darà errore perchè basket ha tipo string, perciò non
si può passare un oggetto.
basket.add({peluches: 12})
```

Quella che abbiamo visto all'inizio di questo codice è un'interfaccia.
Le interfacce sono simili agli alias "type", solo che viene incentivato l'utilizzo delle prime poiché restituiscono migliori messaggi d'errore e sono aperte, rispetto agli aliases che sono chiusi. Per aperte si intende che le interface possono essere estese dichiarandole nuovamente, al contrario degli aliases type che possono essere dichiarate solo una volta:

```
interface Dog {
  weight: "110kg"
}

interface Dog {
  fur: "a lot"
}
```