# Writing strongly typed code in TypeScript

Giulio Canti

June 3, 2018

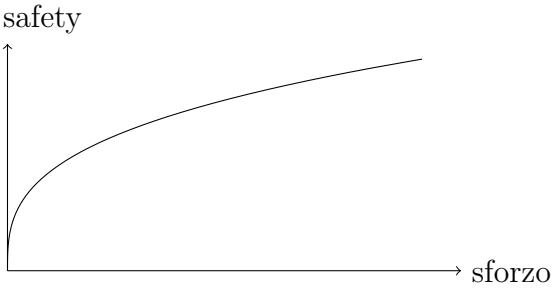
## Contents

# 1 Introduzione

Questo corso mira ad esporre una serie di tecniche per sfruttare al massimo la *type safety* che offre il linguaggio TypeScript.

> Type safe usually refers to languages that ensure that an operation is working on the right kind of data at some point before the operation is actually performed. This may be at compile time or at runtime.

**Obbiettivo** (ambizioso): eliminare gli errori a runtime




## 1.1 Prerequisiti

- `node e npm`
- `typescript@2.8.3`
- `typings-checker@2.0.0`
- `tsconfig.json` con il flag `strict:true`
- `ts-node@6.0.4`
- (consigliato) `prettier@1.11.1+`
- (consigliato) `VS Code`

```
npm i -g typescript@2.8.3 ts-node@6.0.4
git clone https://github.com/gcanti/typescript-course.git
cd typescript-course
npm install
```

## 1.2 Come eseguire i test

```
npm test -- src/<file>
```

## 1.3 Il type system di TypeScript è strutturale

> TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible. - Documentazione ufficiale

**Esempio 1.3.1.** Due classi sono compatibili se sono compatibili i loro campi

```ts
class A {}

class B {}

class C {
  constructor(public value: number) {}
}

declare function f(a: A): void

f(new A())
f(new B())
f(new C(1))
f({})
f(f)

declare function g(c: C): void

g(new C(1))
g(new A()) // error
```

## 1.4 Funzioni parziali

**Definizione 1.1.** Una funzione *parziale* $f : X \to Y$ è una funzione che non è definita per tutti i valori del suo dominio $X$ ($Y$ è chiamato il codominio).

Viceversa una funzione definita per tutti i valori del dominio è detta *totale*.

**Esempio 1.4.1.**

$$f(x) = \frac{1}{x}$$

La funzione $f : number \to number$ non è definita per $x = 0$.

**Esempio 1.4.2.** La funzione `head`

```
// il codominio di questa funzione dovrebbe
// essere number | undefined ma il type-checker
// non mi avverte!
const head = (xs: Array<number>): number => {
  return xs[0]
}

const x: number = head([]) // no error
```

**Esempio 1.4.3.** La funzione `readFileSync`

```
import * as fs from "fs"

// should return `string`
fs.readFileSync("", "utf8")
// throws "no such file or directory" instead
```

Una funzione parziale $f : X \to Y$ può essere sempre ricondotta ad una funzione totale $f'$ aggiungendo un valore speciale, chiamiamolo $None \notin Y$, al codominio e associandolo ad ogni valore di $X$ per cui $f$ non è definita

$$f' : X \to Y \cup None$$

Chiamiamo $Option(Y)$ l'insieme $Y \cup None$.

$$f' : X \to Option(Y)$$

Torneremo a parlare del tipo `Option` più avanti.

| |
|---|
| Quando possibile, cercate di definire funzioni totali |

## 1.5 Strutture dati immutabili

In TypeScript usare strutture dati **mutabili** può condurre ad errori a runtime

```
const xs: Array<string> = ["foo", "bar"]
const ys: Array<string | undefined> = xs
ys.push(undefined)
xs.map(s => s.trim())
// runtime error:
// Cannot read property 'trim' of undefined
```

> Quando possibile, cercate di usare strutture dati immutabili

## 1.6 I tipi `any` e `never`

Se pensiamo ai tipi come insiemi, allora gli *abitanti* di un tipo sono gli elementi di quell'insieme.

```
// gli abitanti sono tutte le stringhe
type A = string

// gli abitanti sono tutti i numeri
type B = number

// questo è un "literal type" e contiene un solo abitante:
// la stringa "foo"
type C = "foo"

// quanti abitanti ha questo tipo?
type D = 0 | 1
```

**Definizione 1.2.** Un tipo `A` si dice *sottotipo* di un tipo `B` se ogni abitante di `A` è abitante di `B`.

Si dice *supertipo* se vale la proprietà inversa.

**Esempio 1.6.1.** Il tipo `C` è sottotipo del tipo `string`. Il tipo `number` è supertipo del tipo `D`

**Esercizio 1.6.1.** In che relazione sono i seguenti tipi?

```
type E = { a: string }
type F = { b: number, a: string }
```

**Definizione 1.3.** Un tipo `X` si dice *bottom type* se è sottotipo di ogni altro tipo

Il tipo `never` non contiene abitanti ed è un *bottom type*.

**Definizione 1.4.** Un tipo `X` si dice *top type* se è supertipo di ogni altro tipo

Il tipo `any` è sia *top type* sia *bottom type*.

> Cercate di utilizzare `any` solo nelle implementazioni e non nelle firme

**Osservazione 1.6.1.** Un problema di `any` è che non è del tutto adatto a rappresentare input non validati

**Esempio 1.6.2.** `JSON.parse` è unsafe

```
const payload = `{"foo":"bar"}`
const x = JSON.parse(payload)
// `x` è di tipo `any`
x.bar.trim() // runtime error:
// Cannot read property 'trim' of undefined
```

Una possibile soluzione è utilizzare la seguente unione

```
type unknown =
  | object // vedi sezione seguente
  | number
  | string
  | boolean
  | symbol
  | undefined
  | null
```

In `typescript@3` ci sarà un tipo `unknown` ufficiale.

## 1.7 Il tipo `object`

Il tipo `object` rappresenta tutti i valori meno quelli primitivi

```
const x1: object = { foo: "bar" }
const x2: object = [1, 2, 3]
const x3: object = 1 // error
const x4: object = "foo" // error
const x5: object = true // error
const x6: object = null // error
const x7: object = undefined // error
```

# 2 Tour delle feature avanzate

Facciamo un tour delle feature avanzate che offre il type system di TypeScript.

## 2.1 Inline declarations

Le dichiarazione all'interno del codice invece che nei definition file sono particolarmente utili quando si stia esplorando una soluzione e per fare velocemente delle prove di type checking.

**Esempio 2.1.1.** Costanti, variabili, funzioni e classi

```
// costanti
declare const a: number
declare const g: (x: number) => void

// variabili
declare let b: number

// funzioni
declare function f(x: string): number

// classi
declare class Foo {
  public value: string
  constructor(value: string)
}
```

## 2.2 Overloading

Gli overloading servono a rendere più precise le firme delle funzioni.

**Esempio 2.2.1.** Vediamo un esempio pratico.

- la funzione `f` deve restituire un numero se l'input è una stringa
- la funzione `f` deve restituire una stringa se l'input è un numero

Usare un'unione non è soddisfacente

```
declare function f(x: string | number): number | string

// x1: string | number
const x1 = f("foo")
// x2: string | number
const x2 = f(1)
```

Definendo due overloading possiamo rendere preciso il comportamento

```
declare function g(x: number): string
declare function g(x: string): number
declare function g(
  x: string | number
): number | string

// x3: number
const x3 = g("foo")
// x4: string
const x4 = g(1)
```

La terza firma di `g` serve a guidare l'implementazione e **non comparirà nel definition file** generato da TypeScript se `declaration = true` nel `tsconfig.json`.

Gli overloading possono essere definiti anche per i metodi di una classe.

**Esercizio 2.2.1.** Tipizzare la funzione `compose`

## 2.3 Polimorfismo

> Parametric polymorphism refers to when the type of a value contains one or more (unconstrained) type variables, so that the value may adopt any type that results from substituting those variables with concrete types.

Una funzione viene detta *polimorfica* se può gestire diversi tipi parametrizzati da uno o più *type parameter*, *monomorfica* altrimenti.

```ts
// una funzione monomorfica
declare function head(xs: Array<number>): number | undefined

// una funzione polimorfica
declare function head<A>(xs: Array<A>): A | undefined
```

Le funzioni polimorfiche favoriscono una implementazione corretta e fanno emergere le assunzioni nascoste

```ts
// compila
const head = (xs: Array<number>): number | undefined => {
  return 1
}

// non compila
const head = <A>(xs: Array<A>): A | undefined => {
  return 1
}
```

Per esempio la funzione identità ha una e una sola implementazione possibile

```ts
const id = <A>(a: A): A => a
```

> Quando possibile, cercate di definire funzioni polimorfiche

**Esercizio 2.3.1.** Date le firme delle seguenti funzioni, cosa possiamo dire del loro comportamento?

```ts
declare function f(xs: Array<number>): Array<number>
declare function g<A>(xs: Array<A>): Array<A>
```

## 2.4 Custom type guards

Le custom type guard servono a *raffinare i tipi*. Un raffinemento di un tipo `A` è un sottoinsieme `B` di `A` tale che per ogni elemento vale un *predicato*.

**Definizione 2.1.** Un *predicato* è una funzione con la seguente firma

```
type Predicate<A> = (a: A) => boolean
```

**Esempio 2.4.1.** In TypeScript la sintassi per definire un predicato non è sufficiente per raffinare un tipo

```
const isString = (x: any): boolean => {
  return typeof x === "string"
}

const f = (x: string | number): number => {
  if (isString(x)) {
    // qui x non è raffinato
    return x.length // error
  } else {
    return x // error
  }
}
```

Invece viene utilizzata questa sintassi (che definisce una type custom guard)

```
type Refinement<A, B extends A> = (a: A) => a is B
```

Notate che `B` **deve essere assegnabile** ad `A`

**Esempio 2.4.2.** Una semplice custom type guard: `isString`

```ts
export const isString = (x: any): x is string => {
  return typeof x === "string"
}

const f = (x: string | number): number => {
  if (isString(x)) {
    // qui x è di tipo string
    return x.length
  } else {
    // qui x è di tipo number
    return x
  }
}
```

**Esercizio 2.4.1.** Definire una custom type guard che raffina un valore qualsiasi in un `Array<number>`

## 2.5 Lifting di un valore: l'operatore `typeof`

I valori e i tipi vivono in mondi separati, però è possibile passare dal mondo dei valori a quello dei tipo sfruttando l'operatore `typeof`.

**Esempio 2.5.1.** Ricavare il tipo di un valore

```ts
const x = {
  foo: "foo",
  baz: 1
}

type X = typeof x
/* same as
type X = {
  foo: string;
  baz: number;
}
*/
```

## 2.6 Immutabilità: il modificatore `readonly`

Il modificatore `readonly` rende immutabili i campi di un record

**Esempio 2.6.1.** Rendere immutabile una interfaccia

```
interface Person {
  readonly name: string
  readonly age: number
}

declare const person: Person

person.age = 42 // Cannot assign to 'agÈ because
// it is a constant or a read-only property
```

È possibile rendere immutabile anche una *index signature*

```
interface ImmutableDictionary {
  readonly [key: string]: number
}

declare const dict: ImmutableDictionary

dict["foo"] = 1 // Index signature in type
// 'ImmutableDictionary' only permits reading
```

Per rendere immutabile un tipo già definito è possibile usare il tipo predefinito `Readonly` [1]

[1] Per l'implementazione di `Readonly` si veda la sezione Mapped types

```ts
interface Point {
  x: number
  y: number
}

type ImmutablePoint = Readonly<Point>
/* same as
type ImmutablePoint = {
    readonly x: number;
    readonly y: number;
}
*/
```

Per i field delle classi è possibile esprimere il modificatore `readonly` direttamente nel costruttore

```ts
class Point2D {
  constructor(readonly x: number, readonly y: number) {}
}
```

È anche possibile rendere immutabile un array con il tipo predefinito `ReadonlyArray`.

**Esercizio 2.6.1.** Rendere immutabile la seguente interfaccia

```ts
interface Person {
  name: {
    first: string
    last: string
  }
  interests: Array<string>
}
```

## 2.7 Index types

### 2.7.1 Index type query operator: `keyof`

Così come è possibile, dato un oggetto, ricavarne le chiavi tramite la funzione `Object.keys`

```
const point = { x: 1, y: 2 }
const pointKeys = Object.keys(point)
// [ "x", "y" ]
```

così è possibile ricavare il tipo delle chiavi di un oggetto (come unione) usando l'operatore `keyof`.

**Esempio 2.7.1.** Estrarre le chiavi di un record

```
interface Point {
  x: number
  y: number
}

type PointKeys = keyof Point
/* same as
type PointKeys = "x" | "y"
*/
```

`keyof` può operare anche sugli array

```
type ArrayKeys = keyof Array<number>
/* same as
type ArrayKeys = "length" | "toString" | "toLocaleString" |
"push" | "pop" | "concat" | "join" | "reverse" | "shift" |
"slice" | "sort" | "splice" | "unshift" | "indexOf" |
"lastIndexOf" | "every" | "some" | "forEach" | "map" |
"filter" | "reduce" | "reduceRight" | "entries" | "keys"
| "values" | "find" | "findIndex" | "fill" | "copyWithin"
*/
```

e le tuple

```
type TupleKeys = keyof [string, number]
/* same as
type TupleKeys = "0" | "1" | "length" | "toString" |
"toLocaleString" | "push" | "pop" | "concat" | "join" |
"reverse" | "shift" | "slice" | "sort" | "splice" |
"unshift" | "indexOf" | "lastIndexOf" | "every" |
"some" | "forEach" | "map" | "filter" | "reduce" |
"reduceRight" | "entries" | "keys" | "values" |
"find" | "findIndex" | "fill" | "copyWithin"
*/
```

**Esercizio 2.7.1.** Rendere type safe la seguente funzione `translate`

### 2.7.2 Indexed access operator `[]`

Così come è possibile, dato un oggetto, ricavare il valore di una sua proprietà usando l'accesso per indice

```
const person = { name: "Giulio", age: 44 }
const name = person["name"]
```

così l'operatore `T[K]` permette di estrarre il tipo del campo `K` dal tipo `T`

**Esempio 2.7.2.** Estrarre il tipo di una chiave di un record

```ts
interface Person {
  name: string
  age: number
}

type Name = Person["name"]
/* same as
type Name = string
*/

type Age = Person["age"]
/* same as
type Age = number
*/

Person["foo"] // error
```

**Esercizio 2.7.2.** Ricavare il tipo del campo `baz` estraendolo dalla seguente definizione

```ts
interface Foo {
  foo: {
    bar: {
      baz: number
      quux: string
    }
  }
}
```

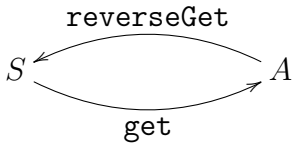
**Esercizio 2.7.3.** Ricavare il tipo delle chiavi del campo `bar` di `Foo`

**Esercizio 2.7.4.** Tipizzare la funzione `get`

**Esercizio 2.7.5.** Aggiungere degli alias a delle proprietà di una classe.

Un *isomorfismo* $f : S \to A$ è una funzione invertibile, ovvero esiste una funzione $f^{-1} : A \to S$ tale che

$$f \circ f^{-1} = f^{-1} \circ f = identity$$




```ts
/**
 * Rappresenta un isomorfismo tra gli insiemi S e A
 */
class Iso<S, A> {
  constructor(
    readonly get: (s: S) => A,
    readonly reverseGet: (a: A) => S
  ) {}
}
```

- aggiungere gli alias `unwrap` e `to` per `get`
- aggiungere gli alias `wrap` e `from` per `reverseGet`

**Esercizio 2.7.6.** Estrarre il tipo di una componente di una tupla

Come è possibile estrarre l'unione dei tipi di una tupla?

```ts
type X = [string, number]

// type ValuesOfX = string | number
type ValuesOfX = X[number]
```

**Esercizio 2.7.7.** Tipizzare la funzione `set`

## 2.8 Mapped types

TypeScript fornisce un modo per creare nuovi tipi basati su tipi già definiti, i *mapped types*. La formula generale di un mapped type è la seguente

$$\{[\texttt{K in U}] : f(\texttt{K})\}$$

ove

- `K` è una variabile
- `U` è una unione
- $f$ è una funzione di `K`

**Esempio 2.8.1.** Creare un *option object*

```
type Flag = "option1" | "option2" | "option3"

type Options = { [K in Flag]: boolean }
/* same as
type Options = {
    option1: boolean;
    option2: boolean;
    option3: boolean;
}
*/
```

Come soluzione è possibile anche usare il tipo predefinito `Record` (per la sua definizione vedi oltre)

```
type Options = Record<Flag, boolean>
```

**Esercizio 2.8.1.** Creare una mappa di predicati

**Esercizio 2.8.2.** Dato lo string literal type

```
type Key = "foo"
```

derivare il tipo

```
type O = {
  foo: number
}
```

Vediamo ora qualche tipo predefinito definito grazie a questa feature

**Esempio 2.8.2.** `Partial<T>`

```
/**
 * Make all properties in T optional
 */
type Partial<T> = { [P in keyof T]?: T[P] }
```

**Esempio 2.8.3.** `Readonly<T>`

```
/**
* Make all properties in T readonly
*/
type Readonly<T> = { readonly [P in keyof T]: T[P] }
```

**Esempio 2.8.4.** `Pick<T, K>`

```
/**
* From T pick a set of properties K
*/
type Pick<T, K extends keyof T> = { [P in K]: T[P] }
```

**Esempio 2.8.5.** `Record<K, T>`

```
/**
 * Construct a type with a set of properties K of type T
 */
type Record<K extends string, T> = { [P in K]: T }
```

**Esercizio 2.8.3.** Rendere type safe la funzione `pick`

## 2.9 Subtyping e type parameter

La keyword `extends` viene usata per estendere una classe

```
class Cat extends Animal {}
```

ma può essere usata anche per esprimere una relazioni dei type parameter. Vediamo un esempio in cui questa feature risulta utile

**Esempio 2.9.1.** Definire un getter generico

```
interface Person {
  name: string
  age: number
}

const getName = (p: Person): string => p.name
```

La funzione `getName` è fin troppo restrittiva, accetta in input un tipo `Person` ma, data l'implementazione, potrebbe lavorare su qualsiasi record che contiene un campo `name` di tipo `string`

```
const getName = <T extends { name: string }>(x: T): string =>
  x.name
```

**Esercizio 2.9.1.** Generalizzare `getName` in modo che lavori con qualsiasi record che abbia una proprietà `name`, anche se non è una stringa

**Esercizio 2.9.2.** `getName` è una funzione che lavora su un campo specifico (`name`), definire una funzione `getter` che, dato il nome di un campo, restituisce il getter corrispondente

```
const getName = getter("name")
```

## 2.10 Module augmentation

**Esempio 2.10.1.** Riaprire una classe modificandone il `prototype`

```ts
// foo.ts
class Foo {
  doSomething(): string {
    return "foo"
  }
}

// augment.ts
import { Foo } from "./foo"

declare module "./Foo" {
  interface Foo {
    doSomethingElse(): number
  }
}

Foo.prototype.doSomethingElse = function() {
  return this.doSomething().length
}

new Foo().doSomethingElse() // ok
```

## 2.11 Conditional type

I *conditional type* hanno la seguente sintassi

```ts
T extends U ? X : Y
```

Questa scrittura significa

se `T` è assegnabile a `U` allora il tipo risultante è `X`, altrimenti è `Y`

**Esempio 2.11.1.** Tipizzare l'operatore `typeof`

```
type TypeName<T> =
    T extends string ? "string" :
    T extends number ? "number" :
    T extends boolean ? "boolean" :
    T extends undefined ? "undefined" :
    T extends Function ? "function" :
    "object";

type T0 = TypeName<string>;  // "string"
type T1 = TypeName<'a'>;  // "string"
type T2 = TypeName<true>;  // "boolean"
type T3 = TypeName<() => void>;  // "function"
type T4 = TypeName<string[]>;  // "object"
```

I conditional type *distribuiscono* le unioni, ovvero se per esempio

```
T extends U ? X : Y
```

è istanziato con `T = A | B | C` , allora il conditional type è risolto in

```
  (A extends U ? X : Y)
| (B extends U ? X : Y)
| (C extends U ? X : Y)
```

Vediamo qualche tipo built-in che sfrutta i conditional type

**Esempio 2.11.2.** `Exclude`

```
/**
 * Exclude from T those types that are assignable to U
 */
type Exclude<T, U> = T extends U ? never : T;
```

Vediamo come funziona con un esempio concreto

```ts
// voglio una copia di ‘Person‘ tranne il campo ‘age‘
export interface Person {
  firstName: string
  lastName: string
  age: number
}

type NotAge = Exclude<keyof Person, 'age'>

type Explanation =
  | ("firstName" extends "age" ? never : "firstName")
  | ("lastName" extends "age" ? never : "lastName")
  | ("age" extends "age" ? never : "age")

type Result = Pick<Person, NotAge>
/* same as
type Result = {
    firstName: string;
    lastName: string;
}
*/
```

**Esempio 2.11.3.** `Extract`

```ts
/**
 * Extract from T those types that are assignable to U
 */
type Extract<T, U> = T extends U ? T : never;
```

**Esempio 2.11.4.** `NonNullable`

```ts
/**
 * Exclude null and undefined from T
 */
type NonNullable<T> = T extends null | undefined ? never : T;
```

Nell clausola `extends` di un conditional type è possibile utilizzare la key-

word `infer` che introduce una type variable da far inferire al type checker. Queste type variable possono essere poi utilizzate nel ramo positivo del conditional type.

**Esempio 2.11.5.** `ReturnType`

```
/**
 * Obtain the return type of a function type
 */
type ReturnType<T extends (...args: any[]) => any> =
  T extends (...args: any[]) => infer R ? R : any;
```

**Esempio 2.11.6.** `InstanceType`

```
/**
 * Obtain the return type of a constructor function type
 */
type InstanceType<T extends new (...args: any[]) => any> =
  T extends new (...args: any[]) => infer R ? R : any;
```

**Esempio 2.11.7.** `Equals` e `AssertEquals`

```
type Equals<A, B> = [A] extends [B] ?
  ([B] extends [A] ? "T" : "F") : "F"

type AssertEquals<A, B, Bool extends Equals<A, B>> = [A, Bool]
```

Per ulteriori esempi, si veda:
https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-8.html

**Esercizio 2.11.1.** Ricavare le chiavi di `X` che hanno valori di tipo `string`

```
interface X {
  a: string
  b: number
  c: string
}
```

**Esercizio 2.11.2.** Tipizzare la funzione `remove`

**Esercizio 2.11.3.** Manipolazione di unioni taggate: estrarre o escludere dei membri

**Esercizio 2.11.4.** Tipizzare la funzione `omit`

# 3 Definition file

Un *definition file* contiene solo dichiarazioni di tipi e servono a descrivere le API pubbliche di una package.

Tipicamente il nome di un definition file termina con `.d.ts`.

## 3.1 Il flag `declaration`

È possibile far generare a TypeScript i definition file dei moduli scritti in TypeScript impostando nel `tsconfig.json` il flag `declaration: true`.

## 3.2 Un problema serio: le API JavaScript

Le API delle librerie JavaScript sono pensate per essere ergonomiche e consumate da JavaScript, aggiungere un definition file a posteriori è spesso problematico.

In più spesso i definition file ufficiali non sono del tutto soddisfacenti.
Possibili soluzioni

- cambiare libreria
- definire un custom definition file
- definire una funzione wrapper con una tipizzazione sana
- module augmentation / declaration merging

**Esempio 3.2.1.** Correggere lodash (49.000.000 di download / mese)

```ts
// lodash@4.17.10
// @types/lodash@4.14.109
import * as _ from "lodash"

const f = (a: number, b: string): number =>
  a + b.trim().length

/*

  La funzione ‘flip‘ è definita con questa tipizzazione

  flip<T extends (...args: any[]) => any>(func: T): T;

*/
const g = _.flip(f)

g(1, "a") // esplode a runtime: b.trim is not a function
```

Una possibile soluzione: module augmentation / declaration merging

```ts
declare module "lodash" {
  interface LoDashStatic {
    flip<A, B, C>(f: (a: A, b: B) => C): (b: B, a: A) => C
    flip<A, B>(f: (a: A) => B): (a: A) => B
    flip<A>(f: () => A): () => A
  }
}
```

**Esercizio 3.2.1.** Correggere la funzione `_.get`

# 4 TDD (Type Driven Development)

> "Type driven development" is a technique used to split a problem into a set of smaller problems, letting the type checker suggest the concrete implementation, or at least helping us getting there.

**Esempio 4.0.1.** Reimplementare `Promise.all`.

```
declare function sequence<T>(
  promises: Array<Promise<T>>
): Promise<Array<T>>
```

live coding...

# 5 ADT (Algebraic Data Types)

Un *Algebraic Data Type* (o ADT) è un tipo composto da product e/o sum types, anche innestati.

## 5.1 Product types

**Definizione 5.1.** Un product type è una collezione di tipi $A_i$ indicizzati da un insieme $I$.

**Definizione 5.2.** Il *prodotto cartesiano* di due insiemi $A$ e $B$ (indicato con $A \times B$) è l'insieme delle coppie $(a, b)$ tale che $a \in A$ e $b \in B$.

Un product type è isomorfo[2] al prodotto cartesiano $\prod_i A_i$.

Esponenti notevoli di questa famiglia sono le $n$-tuple, ove $I$ è un intervallo non vuoto di numeri naturali [3]

**Esempio 5.1.1.** Tuple come product type

```
type Tuple1 = [string]
type Tuple2 = [string, number]
type Tuple3 = [string, number, boolean]
```

e i record, ove $I$ è una collezione di label [4]

**Esempio 5.1.2.** Record come product type

```
type Person = {
  name: string,
  age: number
}
```

`Tuple2` e `Person` sono isomorfi tra loro

[2] Due insiemi $A$ e $B$ sono isomorfi se esiste una funzione $f : A \to B$ iniettiva e suriettiva, ovvero se esiste una funzione $f^{-1} : B \to A$, detta *funzione inversa* di $f$, tale che $f \circ f^{-1} = identity$

[3] $\{0\}$ per `Tuple1`, $\{0, 1\}$ per `Tuple2`, $\{0, 1, 2\}$ per `Tuple2`

[4] $\{"name", "age"\}$ per `Person`

$$f : \texttt{Tuple2} \to \texttt{Person}$$

$$f(\texttt{[name, age]}) = \{\ \texttt{name, age}\ \}$$

$$f^{-1} : \texttt{Person} \to \texttt{Tuple2}$$

$$f^{-1}(\{\ \texttt{name, age}\ \}) = \texttt{[name, age]}$$

L'isomorfismo, almeno in una direzione, risulta evidente se si implementa `Person` con una classe

```
class Person {
  constructor(readonly name: string, readonly age: number) {}
}
```

in cui `constructor` realizza la funzione $f$.
Si può reificare l'isomorfismo costruendo una istanza di `Iso`

**Esempio 5.1.3.** Isomorfismo come valore

```
const iso = new Iso<[string, number], Person>(
  ([name, age]) => ({ name, age }),
  ({ name, age }) => [name, age]
)
```

**Perchè si chiamano product types?** Se indichiamo con $\|A\|$, detta *cardinalità* o *ordine* di $A$, il numero di elementi dell'insieme $A$ è facile convincersi che vale la seguente formula

$$\|A \times B\| = \|A\| * \|B\|$$

ovvero la cardinalità del prodotto cartesiano è il prodotto delle cadinalità.

**Esempio 5.1.4.** Calcolare il numero di abitanti di un product type

```
type Hour = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
type Period = "AM" | "PM"
type Clock = [Hour, Period]
```

Il tipo `Clock` ha $12 * 2 = 24$ abitanti.

## 5.2 Sum types

Così come i product types sono analoghi ai prodotti cartesiani di insiemi, i sum types sono analoghi alle unioni di insiemi disgiunti

```
type Action =
  | { type: "INCREMENT" }
  | { type: "DECREMENT" }
```

Product types e sum types possono essere mischiati e innestati

**Esempio 5.2.1.** Sum type e product type insieme

```
export type Action =
  | { type: "ADD_TODO"; text: string }
  | {
      type: "UPDATE_TODO"
      id: number
      text: string
      completed: boolean
    }
  | { type: "DELETE_TODO"; id: number }
```

Il tipo `Array<A>`, con una buona misura di fantasia, può essere interpretato come sum type

```
type Array<A> = [] | [A] | [A, A] | [A, A, A] | ...
```

I sum type possono essere ricorsivi

**Esempio 5.2.2.** Linked lists

```
type List<A> =
  | { type: "Nil" }
  | { type: "Cons", head: A, tail: List<A> }
```

**Esempio 5.2.3.** Binary trees

```
type BinaryTree<A> =
  | { type: "Leaf" }
  | {
      type: "Node"
      left: BinaryTree<A>
      value: A
      right: BinaryTree<A>
    }
```

**Perchè si chiamano sum types?** È facile convincersi che la cadinalità di un sum type è la somma delle cardinalità dei suoi membri

$$\|A|B\| = \|A\| + \|B\|$$

**Esempio 5.2.4.** Calcolare il numero di abitanti di un sum type

```
type Option<A> = { type: "None" } | { type: "Some"; value: A }
```

Il tipo `Option<boolean>` ha $1 + 2 = 3$ abitanti.

I sum type sono particolarmente utili per modellare record con campi dipendenti tra loro.

**Esercizio 5.2.1.** Definire il tipo `JSON`

## 5.3 Exaustivity checking

Nel caso dei sum type TypeScript offre un servizio prezioso: controlla che ogni caso del sum type sia stato gestito.

Se nella funzione `reducer` qui sotto non vengono gestiti tutti i casi contenuti in `Action` il compilatore ci avverte, così come ci avverte se il tipo `Action` viene cambiato, per esempio aggiungendo o togliendo un caso.

**Esempio 5.3.1.** Exaustivity checking in azione

```
type Action =
  | { type: "ADD_TODO"; text: string }
  | {
      type: "UPDATE_TODO"
      id: number
      text: string
      completed: boolean
    }
  | { type: "DELETE_TODO"; id: number }

interface Todo {
  id: number
  text: string
  completed: boolean
}

interface State {
  todos: Array<Todo>
}

declare const getNextId: (s: State) => number

const reducer = (s: State, a: Action): State => {
  switch (a.type) {
    case "ADD_TODO":
      return {
        todos: [
          ...s.todos,
          { id: getNextId(s), text: a.text, completed: false }
        ]
      }
    case "UPDATE_TODO":
      return {
        todos: s.todos.map(todo =>
          (todo.id === a.id ? a : todo))
      }
    case "DELETE_TODO":
      return {
        todos: s.todos.filter(({ id }) => id !== a.id)
      }
  }
}
```

Cercate di non usare `default` in uno `switch` relativo ad sum type, altrimenti depotenziate l'exaustivity checking

**Esercizio 5.3.1.** Supponiamo di dover modellare la seguente struttura dati (chiamiamola `Selection`)

una lista di utenti di cui uno è considerato la selezione corrente

Un vostro collega propone questo modello

```
interface User {
  name: string
}

interface Selection {
  items: Array<User> // <= la lista degli utenti
  current?: number   // <= l'indice della selezione corrente,
                     // può essere undefined
}
```

Avete una proposta migliore?

# 6 Error handling funzionale

Consideriamo la funzione

```
const inverse = (x: number): number => 1 / x
```

Tale funzione si dice *parziale* perchè non è definita per tutti i valri del dominio ($x = 0$). Come possiamo gestire questa situazione?

Una soluzione potrebbe essere lanciare un'eccezione

```
const inverse = (x: number): number => {
  if (x !== 0) return 1 / x
  throw new Error('cannot divide by zero')
}
```

ma così la funzione non sarebbe più da considerarsi pura [5].

Un'altra possibile soluzione è restituire `null`

```
const inverse = (x: number): number | null => {
  if (x !== 0) return 1 / x
  return null
}
```

Sorge però un nuovo problema quando si cerca di comporre la funzione `inverse` così modificata con un'altra funzione

```
const double = (n: number): number => n * 2

// calcola l'inverso e poi moltiplica per 2
const doubleInverse = (x: number): number => double(inverse(x))
```

L'implementazione di `doubleInverse` non è corretta, cosa succede se `inverse(x)` restituisce `null`? Occorre tenerne conto

[5] Le eccezioni sono considerate un side effect inaccettabile perchè modificano la normale esecuzione del codice e violano la trasparenza referenziale

```
const doubleInverse = (x: number): number | null => {
  const y = inverse(x)
  if (y === null) return null
  return double(y)
}
```

Appare evidente come l'obbligo di gestione del valore speciale `null` si propaghi in modo contagioso a tutti gli utilizzatori di `inverse`.

Questo approccio ha alcuni svantaggi

- molto boilerplate
- le funzioni non compongono facilmente

## 6.1 Il tipo `Option`

La soluzione funzionale ai problemi illustrati precedentemente è l'utilizzo del tipo `Option`, eccone la definizione

```
type Option<A> = None<A> | Some<A>

class None<A> {
  // sum type, si noti la type annotation esplicita
  // altrimenti nel definition file avremmo
  // readonly type: string;
  readonly type: "None" = "None"
  map<B>(f: (a: A) => B): Option<B> {
    return this as any
  }
  chain<B>(f: (a: A) => Option<B>): Option<B> {
    return this as any
  }
}

class Some<A> {
  readonly type: "Some" = "Some"
  constructor(readonly value: A) {}
  map<B>(f: (a: A) => B): Option<B> {
    return new Some(f(this.value))
  }
  chain<B>(f: (a: A) => Option<B>): Option<B> {
    return f(this.value)
  }
}

const none: Option<never> = new None()

const some = <A>(a: A): Option<A> => new Some(a)
```

Ridefiniamo `inverse` sfruttando `Option`

```
const inverse = (x: number): Option<number> =>
  x === 0 ? none : some(1 / x)
```

Possiamo interpretare questa modifica in termini di successo e fallimento: se viene restituita una istanza di `Some` la computazione di `inverse` ha avuto successo, se viene restituita una istanza di `None` essa è fallita.

> Il tipo `Option` codifica l'*effetto* di una computazione che può fallire

Ora è possibile definire `doubleInverse` senza boilerplate

```
const doubleInverse = (x: number): Option<number> =>
  inverse(x).map(double)

doubleInverse(2) // Some(1)
doubleInverse(0) // None
```

Inoltre è facile concatenare altre operazioni

```
const inc = (x: number): number => x + 1

inverse(0)
  .map(double)
  .map(inc) // None
inverse(4)
  .map(double)
  .map(inc) // Some(1.5)
```

> `Option` mi permette di concentrarmi solo sul *path di successo* in una serie di computazioni che possono fallire

Inoltre è possibile concatenare varie operazioni che possono fallire tramite il metodo `chain`

```
const head = <A>(as: Array<A>): Option<A> => {
  return as.length > 0 ? some(as[0]) : none
}

head([]).chain(inverse) // None
head([1, 2, 3]).chain(inverse) // Some(0.5)
```

## 6.2 Branching tramite la funzione `fold`

Prima o poi dovrò affrontare il problema di stabilire cosa fare sia nel caso di successo che di fallimento. La funzione `fold` permette di gestire i due casi

```
class Some<A> {
  ...
  fold<R>(whenNone: () => R, whenSome: (a: A) => R): R {
    return whenSome(this.value)
  }
}

class None<A> {
  ...
  fold<R>(whenSome: () => R, whenNone: (a: A) => R): R {
    return whenSome()
  }
}

const whenNone = (): string => "cannot divide by zero"
const whenSome = (x: number): string => "the result is" + x

inverse(2).fold(whenNone, whenSome) // "the result is 0.5"
inverse(0).fold(whenNone, whenSome) // "cannot divide by zero"
```

Si noti come il branching è racchiuso nella definizione di `Option` e non necessita di alcun `if` e che l'utilizzo necessita solo di funzioni.

Inoltre le funzioni `f` e `g` sono generiche e riutilizzabili.

## 6.3 Interoperabilità

Per questioni di interoperabilità con codice che non usa `Option` possiamo definire alcune funzioni di utility

```
const fromNullable = <A>(
  a: A | null | undefined
): Option<A> => a == null ? none : some(a)

const toNullable = <A>(fa: Option<A>): A | null =>
  fa.fold(() => null, a => a)
```

## 6.4 Il tipo `Either`

Il tipo `Option` è utile quando c'è un solo modo evidente per il quale una computazione può fallire, oppure ce ne sono diversi ma non interessa distinguerli.

Se invece esistono molteplici ragioni di fallimento ed interessa comunicare al chiamante quale si sia verificata, oppure se si vuole definire un errore personalizzato, è possibile impiegare il tipo `Either`. Eccone la definizione

```
type Either<L, A> = Left<L, A> | Right<L, A>

class Left<L, A> {
  readonly type: "Left" = "Left"
  constructor(readonly value: L) {}
  map<B>(f: (a: A) => B): Either<L, B> {
    return this as any
  }
  chain<B>(f: (a: A) => Either<L, B>): Either<L, B> {
    return this as any
  }
}

class Right<L, A> {
  readonly type: "Right" = "Right"
  constructor(readonly value: A) {}
  map<B>(f: (a: A) => B): Either<L, B> {
    return new Right(f(this.value))
  }
  chain<B>(f: (a: A) => Either<L, B>): Either<L, B> {
    return f(this.value)
  }
}

const left = <L, A>(l: L): Either<L, A> =>
  new Left(l)

const right = <L, A>(a: A): Either<L, A> =>
  new Right(a)
```

Per convenzione `Left` rappresenta il caso di fallimento mentre `Right` quello di successo.

Ridefiniamo la funzione `inverse` in funzione del tipo `Either`

```
const inverse = (x: number): Either<string, number> =>
  x === 0 ? left("cannot divide by zero") : right(1 / x)
```

Ancora una volta è possibile definire `doubleInverse` senza boilerplate

```
const doubleInverse = (x: number): Either<string, number> =>
  inverse(x).map(double)

doubleInverse(2) // Right(1)
doubleInverse(0) // Left('cannot divide by zero')
```

ed è facile comporre insieme altre operazioni

```
inverse(0)
  .map(double)
  .map(inc) // Left('cannot divide by zero')
inverse(4)
  .map(double)
  .map(inc) // Right(1.5)
```

Anche per il tipo `Either` è possibile definire una funzione `fold`

```
class Left<L, A> {
  ...
  fold<R>(
    whenLeft: (l: L) => R,
    whenRight: (a: A) => R
  ): R {
    return whenLeft(this.value)
  }
}

class Right<L, A> {
  ...
  fold<R>(
    whenLeft: (l: L) => R,
    whenRight: (a: A) => R
  ): R {
    return whenRight(this.value)
  }
}
```

**Esempio 6.4.1.** Tipizzare le callback nelle API di node.js

Un esempio tipico di una API in node.js è `readFile`

```
declare function readFile(
  path: string,
  callback: (err: Error | undefined, data?: string) => void
): void
```

Il problema di questa tipizzazione è che può rappresentare anche situazioni che dovrebbero essere escluse, in particolare quella in cui sia `err` che `data` sono presenti.

Una migliore tipizzazione può essere realizzata sfruttando il tipo `Either`

```
import { Either } from "./Either"

declare function betterReadFile(
  path: string,
  callback: (result: Either<Error, string>) => void
): void
```

**Esempio 6.4.2.** Tipizzare le `Promise`

La tipizzazione standard delle `Promise` non prevede un tipo per gli errori.

```
declare function readFile(path: string): Promise<string>
```

Anche in questo caso una migliore tipizzazione può essere realizzata sfruttando il tipo `Either`.

```
import { Either, right, left } from './Either'

declare function betterReadFile(
  path: string
): Promise<Either<Error, string>>

export const attempt = <L, A>(
  promise: Promise<A>,
  onrejected: (reason: {}) => L
): Promise<Either<L, A>> => {
  return promise.then(
    a => right<L, A>(a),
    reason => left<L, A>(onrejected(reason))
  )
}
```

# 7 Make impossible states irrepresentable

Vediamo un'altra tecnica per ottenere type safety, questa volta addirittura per costruzione.

Sappiamo che la funzione `head` è parziale

```
const head = <A>(xs: Array<A>): A => xs[0]
```

e che per renderla totale occorre modificare il codominio

```
const head = <A>(xs: Array<A>): Option<A> =>
  xs.length > 0 ? some(xs[0]) : none
```

Tuttavia questo ci obbliga ad usare `Option`.

Un'altra opzione è quella di cambiare il dominio invece che estendere il codominio

## 7.1 Il tipo `NonEmptyArray`

```
class NonEmptyArray<A> {
  constructor(readonly head: A, readonly tail: Array<A>) {}
}

const head = <A>(fa: NonEmptyArray<A>): A => fa.head
```

## 7.2 Il tipo `Zipper`

Supponiamo di dover modellare la seguente struttura dati

> una lista non vuota di elementi di cui uno è considerato la selezione corrente

Un modello semplice potrebbe essere questo

```
type Selection<A> = {
  items: Array<A>
  current: number
}
```

Tuttavia questo modello ha diversi difetti

- la lista può essere vuota
- l'indice può essere out of range

Possiamo migliorare il modello usando `NonEmptyArray`

```
type Selection<A> = {
  items: NonEmptyArray<A>
  current: number
}
```

Tuttavia l'indice può essere ancora out of range.
Uno `Zipper` invece è un modello perfetto e type safe per il problema

```
type Zipper<A> = {
  prev: Array<A>
  current: A
  next: Array<A>
}
```

## 7.3 Slaying a UI Antipattern with TypeScript

**The problem.** This problem is presented in "Slaying a UI Antipattern in Fantasyland"[6] by Stefan Oestreicher which in turn is based on "How Elm Slays a UI Antipattern"[7] by Kris Jenkins.

The problem they present is a very common one. You are loading a list of things but instead of showing a loading indicator you just see zero items. In JavaScript your data model may look like this

```
{
  loading: true,
  items: []
}
```

But of course it's easy to forget to check the loading flag. What about using `null` in order to represent the "not loaded" case? Both Stefan and Kris wisely discourage it

> Long experience will have taught you that setting a property to null may be correct, but it's just asking for runtime exceptions

Fortunately, this concern evaporates when using TypeScript

```
type Model = {
  things: Array<Thing> | undefined | null
}
```

Now if you try to use an instance of this model incorrectly

```
const SomeView = ({ things }: Model) => {
  return <div>
    { things.map(thing => String(thing)) }
  </div>
}
```

TypeScript will complain

[6] https://medium.com/javascript-inside/slaying-a-ui-antipattern-in-fantasyland-907cbc322d2a

[7] http://blog.jenkster.com/2016/06/how-elm-slays-a-ui-antipattern.html

```
[ts] Object is possibly 'null' or 'undefined'.
```

However we can go even further, as Kris says "we can be much more sophisticated".

**The solution.** HTTP requests have one of four states

- we haven't asked yet
- w've asked, but we haven't got a response yet
- we got a response, but it was an error
- we got a response, and it was the data we wanted

With TypeScript we can easily define a type that represents these four states

```
type RemoteData<E, D> =
  | { type: "NotAsked" }
  | { type: "Loading" }
  | { type: "Failur"; error: E }
  | { type: "Success"; data: D }

type Model = {
  things: RemoteData<HttpError, Array<Thing>>
}
```

As Kris observes

> The nice thing about this data model is, the type checker will now force you to write the correct UI code. It will keep track of the possibility of "things not loaded" and errors, and force you to handle them all in the UI.

```
// will raise: Property 'data' does not exist
// on type 'RemoteData<string, Thing[]>'
const SomeView = ({ things }: Model) => {
  return <div>{things.data.map(thing => {})}</div>
}
```

TypeScript will force you to handle all cases

```
const SomeView = ({ things }: Model) => {
  switch (things.type) {
    case "NotAsked":
      return (
        <div>Please press the button to load the things</div>
      )
    case "Loading":
      return <div>Loading things...</div>
    case "Failure":
      return <div>An error has occurred {things.error}</div>
    case "Success":
      return <div>{things.data.map(thing => { ... })}</div>
  }
}
```

Or, using a more functional style, let's define a `fold` function that can be re-utilised in more use cases

```
function fold<E, D, R>(
  rd: RemoteData<E, D>,
  fs: {
    NotAsked: () => R
    Loading: () => R
    Failure: (error: E) => R
    Success: (data: D) => R
  }
): R {
  switch (rd.type) {
    case "NotAsked":
      return fs.NotAsked()
    case "Loading":
      return fs.Loading()
    case "Failure":
      return fs.Failure(rd.error)
    case "Success":
      return fs.Success(rd.data)
  }
}
```

Now `SomeView` can be defined as

```
const SomeView = ({ things }: Model) =>
  fold(
    things,
    {
      NotAsked: () => (
        <div>Please press the button to load the things</div>
      ),
      Loading: () => <div>Loading things...</div>,
      Failure: error => <div>An error has occurred {error}</div>,
      Success: data => <div>{data.map(thing => {})}</div>
    }
  )
```

## 7.4 Finite state machines

> First, we should have a clear understanding of what a finite-state machine is. There are many variations and definitions, and I'm sure you, especially if coming from an engineering background, have some relation to state machines.
>
> In general, a finite-state machine can be described as an abstract machine with a finite set of states, being is in one state at a time. Events trigger state transitions; that is, the machine changes from being in one state to being in another state. The machine defines a set of legal transitions, often expressed as associations from the current state and an event to another state.[8]

Gli stati e gli eventi possono essere modellati con dei sum type, rispettivamente `S` e `E`.

Il modello di una finite state machine è allora

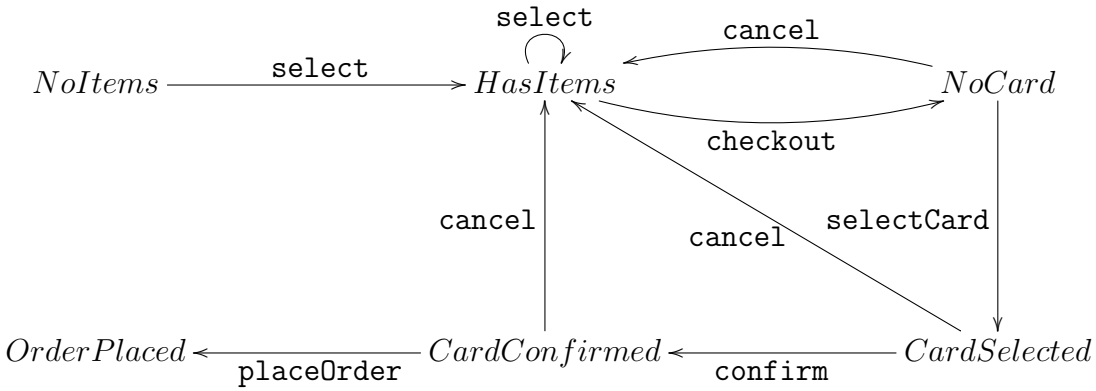
```
type PureFSM<S, E> = (s: S, e: E) => S
```

Siccome però è probabile che per produrre il nuovo stato occorra generare dei side effect, un modello più realistico è il seguente

```
type FSM<S, E> = (s: S, e: E) => Promise<S>
```

**Esercizio 7.4.1.** Modellare il seguente grafo (shopping cart checkout) usando degli ADT sia per gli stati sia per gli eventi

[8] https://wickstrom.tech/finite-state-machines/2017/11/10/finite-state-machines-part-1-modeling-with-haskell.html


select
cancel
NoItems
select
HasItems
NoCard
checkout
cancel
cancel
selectCard
OrderPlaced
placeOrder
CardConfirmed
confirm
CardSelected

# 8 Come migliorare la type inference delle funzioni polimorfiche

Consideriamo la generica definizione di funzione

$$\texttt{const f} = \underbrace{(a_1, \ldots, a_n)}_{A} \texttt{ => } \underbrace{(b_{n+1}, \ldots, b_m)}_{B} \texttt{ => } \ldots \texttt{ => } \{ \ldots \}$$

Chiamiamo $A$, $B$, ... col nome di *gruppi di argomenti* della funzione `f`.

Quando la funzione definita è polimorfica, ogni gruppo di argomenti è sede di inferenza per TypeScript.

**Esempio 8.0.1.** La seguente funzione ha un solo gruppo di argomenti e TypeScript è in grado di inferire correttamente tutti i type parameter coinvolti

```
const map = <A, B>(
  f: (a: A) => B,
  fa: Array<A>
): Array<B> => fa.map(f)

// map: <string, number>
map(s => s.length, ['foo'])
```

se consideriamo la versione curried di `map` le cose cambiano

```
const mapCurried = <A, B>(f: (a: A) => B) => (
  fa: Array<A>
): Array<B> => fa.map(f)

// mapCurried: <{}, any>
mapCurried(s => s.length)(['foo'])
// error: Property 'length' does not exist on type '{}'
```

TypeScript non è in grado di inferire il type parameter `A` nel primo gruppo di argomenti e gli assegna il tipo {}. Lo sviluppatore perciò deve correggere la situazione aggiungendo una type annotation alla callback

```
// mapCurried: mapCurried: <string, number>(f:
// (a: string) => number) => (fa: string[]) => number[]
mapCurried((s: string) => s.length)(['foo'])
```

oppure specificando i type parameter

```
mapCurried<string, number>(s => s.length)(['foo'])
```

È però possibile migliorare la type inference scambiando l'ordine dei gruppi di argomenti

```
const mapCurriedFlipped = <A>(fa: Array<A>) => <B>(
  f: (a: A) => B
): Array<B> => fa.map(f)

// mapCurriedFlipped: <string>(fa: string[]) =>
// <B>(f: (a: string) => B) => B[]
mapCurriedFlipped(['foo'])(s => s.length)
```

# 9 Simulazione dei tipi nominali

È possibile simulare i tipi nominali aggiungendo una proprietà fittizia (detta *phantom property*) che non viene mai valorizzata a runtime ma che viene comunque presa in considerazione dal compilatore.

**Esempio 9.0.1.** Implementazione con le classi

```
class A {
  readonly type!: "A"
}

class B {
  readonly type!: "B"
}

declare function f(a: A): void

f(new A())
f(new B())
/*
[ts]
Argument of type 'B' is not assignable to parameter of type 'A'.
  Types of property '_tag' are incompatible.
    Type '"B"' is not assignable to type '"A"'.
*/
```

Si noti l'uso del *definite property assignment assertion operator*.

# 10 Refinements e smart constructors

Consideriamo la funzione `inverse`

```
const inverse = (x: number): Option<number> =>
  x === 0 ? none : some(1 / x)
```

Un altro modo per ottenere lo stesso grado di type safety senza avere una funzione parziale è l'utilizzo degli *smart constructors*.

In pratica si fa in modo che `Option` non compaia a valle, nel codominio di `inverse`, ma a monte, in fase di creazione dell'input `x`.

Supponiamo di volere rappresentare il tipo "numero diverso da zero", possiamo implementarlo con un wrapper nominale

```
class NonZero {
  readonly type: "NonZero" = "NonZero"
  constructor(readonly value: number) {}
}
```

Il problema è che non ho controllo sulla creazione

```
const x = new NonZero(0)
```

Posso allora definire uno *smart constructor*, ovvero una funzione che ha come codominio `Option<NonZero>` e che effettua il controllo a runtime

```
export const create = (n: number): Option<NonZero> =>
  n === 0 ? none : some(new NonZero(n))
```

Nella pratica uno smart constructor rappresenta un controllo a runtime a livello dei tipi.

Si noti che `NonZero` non è esportato mentre dal modulo è esportata la sola funzione `create`, in questo modo si mantiene il controllo totale sulla creazione di istanze di `NonZero` che non sempre valide.

Tuttavia ci può essere una complicazione se `declaration = true` nel `tsconfig.json` (caso che può capitare se per esempio se si sta scrivendo una libreria). In questo caso TypeScript costringe ad esportare anche il tipo `NonZero`, ma con esso **anche il costruttore**.

Si può rimediare con il seguente workaround

- rendere privato il costruttore di `NonZero`
- spostare la definizione di `creare` come funzione statica di `NonZero`

```
class NonZero {
  readonly type: "NonZero" = "NonZero"
  static create(n: number): Option<NonZero> {
    return n === 0 ? none : some(new NonZero(n))
  }
  private constructor(readonly value: number) {}
}

// inverse adesso è totale!
const inverse = (x: NonZero): number => 1 / x.value
```

In questo modo spesso si spingono i controlli a runtime là dove dovrebbe essere il loro posto naturale: ai confini del sistema, dove vengono fatte tutte le validazioni dell'input.

# 11 Phantom types

A *phantom type* is a parametrised type whose parameters do not all appear on the right-hand side of its definition, e.g

```
type Phantom<M> = { value: number }
```

Here `Phantom` is a phantom type and `M` is a *phantom type parameter*, because the `M` parameter doesn't appear in its implementation.

Since TypeScript is a structural type system we must choose a different encoding

**Esercizio 11.0.1.** Perchè?

e.g. using a `class` and a phantom property

```
class Phantom<M> {
  readonly M!: M
  constructor(readonly value: number) {}
}
```

## 11.1 Validating user input

Say you have a

```
declare function use(input: string): void
```

function (the return type doesn't really matter for our example), perhaps it saves the input to a database, or calls an internal API.

Now you want to enforce that, before being called, the input has been validated.

And maybe you also don't want to waste CPU cycles (let's assume the process of validating is expensive) so you want to ensure that the validation happens only once, what would you do?

Let's start with a few definitions

```
class Data<M> {
  readonly M!: M
  constructor(readonly input: string) {}
}
```

The `Data` class looks strange since at first it seems the type parameter is unused and could be anything, without affecting the value inside. Indeed, one can write

```
const changeType = <A, B>(data: Data<A>): Data<B> =>
  new Data(data.input)
```

to change it from any type to any other.

However, if the constructor is not exported then users of the library that defined `Data` can't define functions like the above, so the type parameter can only be set or changed by library functions.

So we might do

```
export type Unvalidated = "Unvalidated"
export type Validated = "Validated"
export type State = Unvalidated | Validated

declare function myvalidation(s: string): boolean

export class Data<M extends State> {
  readonly M!: M
  private constructor(readonly input: string) {}

  /**
   * since we don't export the constructor itself,
   * users with a string can only create Unvalidated values
   */
  static make(input: string): Data<Unvalidated> {
    return new Data(input)
  }

  /**
   * returns none if the data doesn't validate
   */
  static validate(
    data: Data<Unvalidated>
  ): Option<Data<Validated>> {
    return myvalidation(data.input)
      ? some(data as any)
      : none
  }
}

/**
 * can only be fed the result of a call to validate!
 */
export function use(data: Data<Validated>): void {
  console.log("using " +  data.input)
}
```

Now let's try to use the library incorrectly

```
import { Data, use } from './data'

const data = Data.make('hello')

use(data) // called without validating the input
```

If you run TypeScript you get the following error

```
[ts]
Argument of type 'Data<"Unvalidated">' is not assignable
  to parameter of type 'Data<"Validated">'.
  Type '"Unvalidated"' is not assignable to type '"Validated"'.
```

If you call `validate` instead

```
const data = Data.make('hello')
Data.validate(data).map(use)
```

everything is fine.

The beauty of this is that we can define functions that work on all kinds of `Data`, but still can't turn unvalidated data into validated data

```
static toUpperCase<M extends State>(data: Data<M>): Data<M> {
  return new Data(data.input.toUpperCase())
}
```

One last thing, what happens if you try to validate the input **twice**?

```
const data = Data.make('hello')
Data.validate(data).chain(validated =>
  Data.validate(validated)
)
```

TypeScript complains!

```
[ts]
Argument of type 'Data<"Validated">' is not assignable
  to parameter of type 'Data<"Unvalidated">'.
  Type '"Validated"' is not assignable to type '"Unvalidated"'.
```

This technique is perfect for validating user input to a web application. We can ensure **with almost zero overhead** that the data is validated **once and only once** everywhere that it needs to be, or else we get a compile-time error.

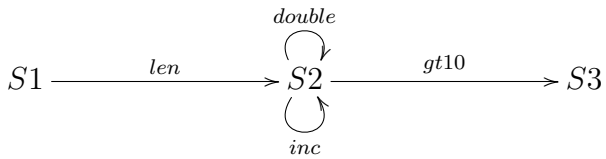
## 11.2 Finite state machines

Vediamo un'altra implementazione di una macchina a stati finiti.

L'implementazione si basa su questi due principi

- gli stati sono rappresentati da una struttura dati `Node` contenente un valore e una indicazione dello stato corrente
- gli eventi / transizioni sono rappresentate da funzioni

Si consideri la seguente macchina a stati finiti




```
type State = "S1" | "S2" | "S3"

class Node<S extends State, A> {
  readonly S!: S
  constructor(readonly value: A) {}
}

const start = (value: string): Node<"S1", string> =>
  new Node(value)
```

Ora definiamo una operazione per ogni arco del grafo

```
const len = (
  input: Node<"S1", string>
): Node<"S2", number> => new Node(input.value.length)

const double = (
  input: Node<"S2", number>
): Node<"S2", number> => new Node(input.value * 2)

const inc = (
  input: Node<'S2', number>
): Node<'S2', number> => new Node(input.value + 1)

const gt10 = (
  input: Node<"S2", number>
): Node<"S3", boolean> => new Node(input.value > 10)
```

La corretta successione delle operazioni ora è assicurata staticamente

```
double(start("foo")) // error
double(len(start("foo"))) // ok
```

Volendo le operazioni possono anche essere definite come metodi, il che rende più comodo concatenarle, sfruttando la possibilità di annotare `this`.

Inoltre posso imporre uno stato iniziale rendendo il costruttore privato

```
class Node2<S extends State, A> {
  readonly S!: S
  static start(value: string): Node2<"S1", string> {
    return new Node2(value)
  }
  private constructor(private readonly value: A) {}
  len(this: Node2<"S1", string>): Node2<"S2", number> {
    return new Node2(this.value.length)
  }
  double(this: Node2<"S2", number>): Node2<"S2", number> {
    return new Node2(this.value * 2)
  }
  inc(this: Node2<"S2", number>): Node2<"S2", number> {
    return new Node2(this.value + 1)
  }
  gt10(this: Node2<"S2", number>): Node2<"S3", boolean> {
    return new Node2(this.value > 10)
  }
}
```

Ancora una volta la corretta successione delle operazioni è assicurata staticamente

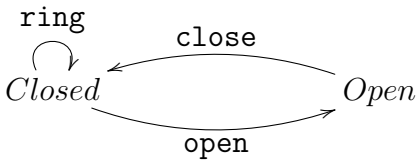
```
Node2.start("foo").double()
// static error: Type '"S1"' is not assignable to type '"S2"'

Node2.start("foo")
  .len()
  .double() // ok
```

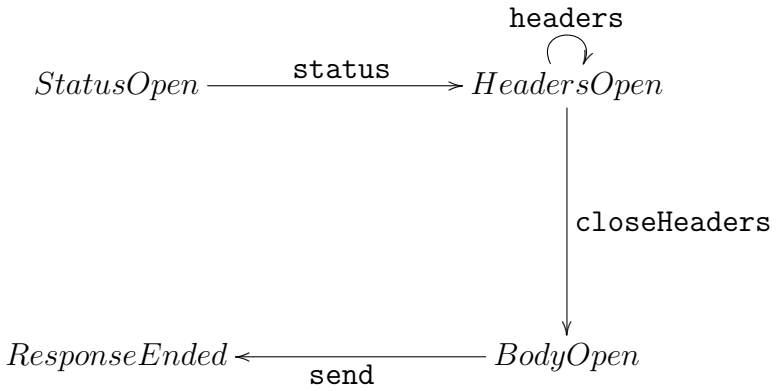
Posso anche richiedere un esplicito stato finale

```
const final: Node2<"S3", boolean> = Node2.start("foo")
  .len()
  .gt10()
```

**Esercizio 11.2.1.** Modellare il seguente grafo (operazioni su una `Door`), mantenendo il conteggio delle volte in cui il campanello è stato suonato




**Esercizio 11.2.2.** Modellare il seguente grafo (operazioni su una `Response`)




## 11.3 Un `EventEmitter` type safe

Supponiamo di dover modellare un `EventEmitter`

```
interface EventEmitter {
  emit: (name: string, data: any) => void
  listen: (name: string, handler: (data: any) => void) => void
}
```

Questo modello è insoddisfacente

- non c'è relazione tra l'evento e i dati associati
- non c'è relazione tra l'emissione dell'evento e i relativi listener

Si può rimediare definendo un phantom type `Event` che immagazzina il tipo di dato relativo ad un evento

```
class Event<D> {
  readonly D!: D
  constructor(readonly name: string) {}
}

interface EventEmitter {
  emit: <D>(evt: Event<D>, data: D) => void
  listen: <D>(evt: Event<D>, handler: (data: D) => void) => void
}

const evt1 = new Event<string>('evt1')
const evt2 = new Event<number>('evt2')

declare const ee: EventEmitter

ee.emit(evt1, 'foo') // ok
ee.emit(evt1, 1) // static error
ee.emit(evt2, 1) // ok

ee.listen(evt1, data => console.log(data.trim()))
ee.listen(evt2, data => console.log(data * 2))
```

## 11.4 Estrarre i tipi da mappe eterogenee

I phantom type sono utili anche per manipolare mappe i cui valori sono di tipo eterogeneo

```
type EventMap = { [key: string]: Event<any> }

const map = {
  evt1,
  evt2
}

type Handlers<EM extends EventMap> = {
  [K in keyof EM]: (data: EM[K]["D"]) => void
}

type MyHandlers = Handlers<typeof map>
/* same as
type MyHandlers = {
    evt1: (data: string) => void;
    evt2: (data: number) => void;
}
*/
```

# 12 Newtypes

Non sempre i tipi riescono a modellare in modo soddisfacente un sistema, si consideri la seguente funzione

```
declare function getPost(a: string, b: string): string
```

Cosa sono `a` e `b`? Usiamo dei nomi più parlanti per gli argomenti

```
declare function getPost(
  postId: string,
  facebookToken: string
): string
```

e anche per i tipi

```
type PostId = string
type FacebookToken = string
type PostContents = string

declare function getPost(
  postId: PostId,
  facebookToken: FacebookToken
): PostContents
```

ma il type system vede ancora `(string, string) -> string` (e anche voi in VSCode),

Posso per errore scambiare l'ordine di `PostId` e `FacebookToken` essendo tutte e due stringhe, il type checker non mi avverte.

> A common programming practice is to define a type whose representation is identical to an existing one but which has a separate identity in the type system.

Un esempio tipico sono i valori espressi in una qualche unità di misura

```
type Celsius = number
type Fahrenheit = number

const celsius2fahrenheit = (celsius: Celsius): Fahrenheit =>
  celsius * 1.8 + 32

const c: Celsius = 1

celsius2fahrenheit(c)

const f: Fahrenheit = 33.8

celsius2fahrenheit(f) // oops...
```

I type alias danno qualche beneficio in termini di documentazione ma non offrono nessun vantaggio dal punto di vista della type safety.

## 12.1 Phantom type wrapper

Una soluzione è quella di creare dei wrapper simulando i tipi nominali

```
class Newtype<M, A> {
  readonly M!: M
  constructor(readonly value: A) {}
}

class Celsius extends Newtype<"Celsius", number> {}
class Fahrenheit extends Newtype<"Fahrenheit", number> {}

const celsius2fahrenheit = (celsius: Celsius): Fahrenheit =>
  new Fahrenheit(celsius.value * 1.8 + 32)

const f = new Fahrenheit(33.8)

celsius2fahrenheit(f)
// static error: Type '"Fahrenheit"' is not assignable
// to type '"Celsius"'
```

Un `newtype` tuttavia ha la caratteristica di *non modificare la rappresentazione a runtime* cosa che chiaramente non succede con un wrapper. È possibile implementare in TypeScript un nozione equivalente?

## 12.2 Implementazione tramite `Iso`

Dato che newtype di un tipo `A` è *isomorfo* ad `A` possiamo usare un `Iso` per passare da uno all'altro.

```
interface Newtype<M, A> {
  readonly M: M
  readonly A: A
}

const unsafeCoerce = <A, B>(a: A): B => a as any

const anyIso = new Iso<any, any>(unsafeCoerce, unsafeCoerce)

const iso = <S extends Newtype<any, any>>(): Iso<
  S,
  S["A"]
> => anyIso

interface Celsius extends Newtype<"Celsius", number> {}
const celsiusIso = iso<Celsius>()

interface Fahrenheit extends Newtype<"Fahrenheit", number> {}
const fahrenheitIso = iso<Fahrenheit>()

const celsius2fahrenheit = (celsius: Celsius): Fahrenheit =>
  fahrenheitIso.from(celsiusIso.to(celsius) * 1.8 + 32)

const f: Fahrenheit = fahrenheitIso.from(33.8)

celsius2fahrenheit(f)
// static error: Type '"Fahrenheit"' is not
// assignable to type '"Celsius"'
```

Sfruttando il subtyping è possible codificare comportamenti interessanti

```
interface NonZero
  extends Newtype<{ NonZero: true }, number> {}
interface Positive

  extends Newtype<
      { NonZero: true; Positive: true },
      number
    > {}

declare function inverse(nz: NonZero): NonZero
declare function mult(a: Positive, b: Positive): Positive

declare const nonZero: NonZero
declare const positive: Positive

inverse(nonZero)
inverse(positive)
mult(positive, nonZero)
// error: Property 'Positive' is missing
// in type '{ NonZero: true; }'
```

# 13 Validazione a runtime

TypeScript ci aiuta ad avere type safety all'interno del sistema ma alla sua frontiera occorre validare i dati in ingresso. Scrivere manualmente le validazioni è noioso e prono ad errori, vediamo una soluzione più economica: definire un runtime type system che collabori con lo static type system.

live coding...

# 14 Covarianza e controvarianza

TypeScript 2.6 ha introdotto un nuovo flag `strictFunctionTypes` (contenuto nel gruppo `strict`) che abilita il controllo della *varianza* sulle funzioni.

È perciò necessario capire cosa sia la varianza e come questa influisce sul design delle API e suigli errori che il compilatore può emettere.

**Definizione 14.1.** Within the type system of a programming language, a typing rule or a type constructor `T` is:

- *covariant* if it preserves the ordering of types ($\leq$), which orders types from more specific to more generic
- *contravariant* if it reverses this ordering
- *bivariant* if both of these apply (i.e., both `T<A>` $\leq$ `T<B>` and `T<B>` $\leq$ `T<A>` at the same time)
- *invariant* if neither of these applies

## 14.1 Array

In teoria

- Read-only data types (sources) can be covariant
- write-only data types (sinks) can be contravariant
- Mutable data types which act as both sources and sinks should be invariant.

In TypeScript gli array invece sono **covarianti**, ovvero
`Array<A>` $\leq$ `Array<B>` se `A` $\leq$ `B`

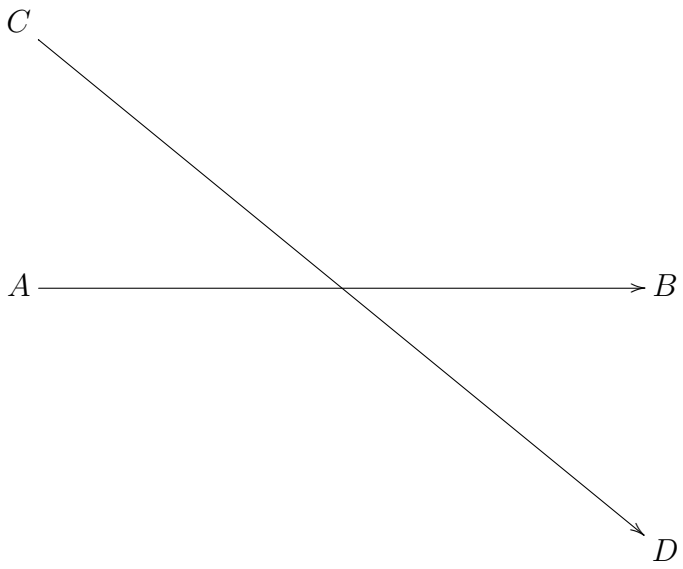
```
const xs: Array<number> = [1, 2, 3]
const ys: Array<number | undefined> = xs // ok
```

Attenzione però, questo comportamento è *sound* solo se gli array sono **immutabili**

```
ys.push(undefined) // ok :(
```

## 14.2 Functions

Le funzioni sono **controvarianti** in input e **covarianti** in output, ovvero `(a:A) => B` $\leq$ `(c:C) => D` se `A` $\leq$ `C` e `D` $\leq$ `B`



# 15 React

## 15.1 Default props

### 15.1.1 Il problema

### 15.1.2 Una soluzione

```
// default props
type DP = { foo: string }

// non default props
type NDP = { bar: number; baz: boolean }

class MyComponent extends React.Component<DP & NDP> {
  // type-checked defaultProps
  static defaultProps: DP = { foo: 'foo' }

  render() {
    // default props appear as required within the component
    const { foo, bar, baz } = this.props
    return (
      <div>
        {foo.trim()} {baz ? bar * 2 : 0}
      </div>
    )
  }
}

// make default props optional
default MyComponent as React.ComponentClass<
  Partial<DP> & NDP
>
```

**Features**

- the value `defaultProps` is type-checked

- default props appear as required within the component but optional to the consumer
- if you forget the last cast the result is even more type-safe and the fix is backward compatible

## 15.2 Componenti polimorfiche

Se si usa JSX le componenti polimorfiche sono supportate solo se sono implementate tramite funzioni (stateless components). Per le componenti definite come classi un workaround possibile è la specializzazione prima dell'utilizzo. In quest'ultimo caso di consiglia di aggiungere un default `never` ai type parameter per evitare un utilizzo non corretto della componente

```tsx
interface Props<T> {
  a: T
  b: T[]
}

declare const Comp1: <T>(
  props: Props<T>
) => React.ReactElement<any>

declare class Comp2<T> extends React.Component<Props<T>> {
  render(): React.ReactElement<any>
}

// specializzazione
declare class MyComp2 extends Comp2<number> {}

const props = {
  a: '',
  b: [2]
}
;<Comp1 {...props} /> // error
;<Comp2 {...props} /> // no error

// specializzazione
declare class MyComp2 extends Comp2<number> {}

;<MyComp2 {...props} /> // error

// usare 'never' come default per evitare un cattivo uso
declare class Comp3<T = never> extends React.Component<
  Props<T>
> {
  render(): React.ReactElement<any>
}

;<Comp3 {...props} /> // error
```