# Writing strongly typed code in TypeScript

Giulio Canti

November 25, 2018

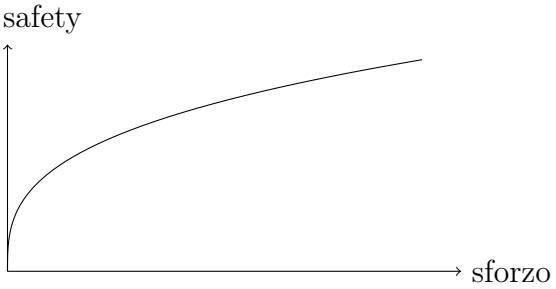
## Contents

# 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@3.1.6+`
- `typings-checker@2.0.0+`
- `tsconfig.json`
  - `"strict": true`
  - (consigliato) `"noImplicitAny": true`
- `ts-node@6.0.4+`
- (consigliato) `prettier@1.15.2+`
- (consigliato) `VS Code`

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

## 1.2 Come eseguire gli esercizi del corso

Gli esercizi sono contenuti nella cartella `src`. Una volta svolto un esercizio è possibile controllarne il risultato eseguendo il seguente comando da terminale

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

per gli esercizi che riguardano la programmazione type level, oppure

```
ts-node src/<file>
```

per gli altri.

Se non viene sollevata alcuna eccezione la soluzione è da considerarsi corretta.

**Osservazione 1.2.1.** Le soluzioni fornite nel repository `typescript-course` costituiscono solo un punto di referimento, in generale **gli esercizi possono ammettere più di una soluzione corretta**.

## 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

```
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`

```ts
// 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`

```ts
import * as fs from "fs"

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

**Osservazione 1.4.1.** Il problema più grosso nel lanciare eccezioni è che questo fatto non si riflette nella firma della funzione, quindi il comportamento reale della funzione non è codificato a livello del type system di TypeScript.

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 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
```

## 1.7 I tipi `any`, `never` e `unknown`

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

```ts
// 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 1.7.1.** Il tipo `C` è sottotipo del tipo `string`. Il tipo `number` è supertipo del tipo `D`

**Esercizio 1.7.1.** In che relazione sono i seguenti tipi?

```ts
type T1 = { a: string }
type T2 = { b: number, a: string }
```

- ☐ `T1` è sottotipo di `T2`
- ☐ `T2` è sottotipo di `T1`
- ☐ nessuno dei due

```ts
type T3 = { a: string, b: boolean }
type T4 = { b: number, a: string }
```

- ☐ `T3` è sottotipo di `T4`
- ☐ `T4` è sottotipo di `T3`

☐ nessuno dei due

**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*.

```
const raise = (message: string): never => {
  throw new Error(message)
}

const absurd = <A>(x: never): A => {
  return raise("absurd")
}
```

**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*. Viene usato per "disabilitare" il type-checker (a volte risulta necessario). Usatelo con parsimonia.

**Osservazione 1.7.1.** Un problema di `any` è che non è adatto a rappresentare *input non validati*

**Esempio 1.7.2.** `JSON.parse` è unsafe dato che il tipo di ritorno è `any`.

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

Dalla `3.0.1` una possibile soluzione è utilizzare il tipo `unknown`.
Il tipo `unknown` è un top type ma **non è** un bottom type.
Per poter utilizzare un valore di tipo `unknown` occorre *raffinarlo*

**Esempio 1.7.3.** Un `JSON.parse` type safe (o quasi [1])

[1] può lanciare eccezioni

```ts
export const parse: (input: string) => unknown = JSON.parse

const payload = `{"bar":"foo"}`
const x = parse(payload)
x.bar // static error: Object is of type 'unknown'
```

**Osservazione 1.7.2.** Ridefinire il tipo di una funzione può essere fatto in modo che non ci sia alcun costo a runtime.

*Raffinare* un valore vuol dire scrivere del codice specifico che permette di provare al type checker alcune caratteristiche sul tipo del valore

```ts
if (typeof x === 'object') {
  // x ha tipo object | null
  if (x !== null) {
    // x ha tipo object
    const bar = (x as { [key: string]: unknown }).bar
    if (typeof bar === 'string') {
      // bar ha tipo string
      console.log(bar.trim())
    }
  }
}
```

Vedremo più avanti come sia possibile eliminare il boilerplate utilizzando le *custom type guard*.

# 2 Tour delle feature avanzate

## 2.1 Inline declarations

Le dichiarazione all'interno del codice invece che nei **definition file** [2] 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

// variabili
declare let b: number

// funzioni
declare function f(x: string): number
declare const g: (x: string) => number

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

In molti degli esercizi sfrutteremo questa feature lavorando solo sulle dichiarazioni invece di preoccuparci dell'implementazione.

## 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

[2] I definition file costituiscono un ponte tra il mondo untyped di JavaScript e quello types di TypeScript

- 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 tipo della funzione

```
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.

```
class G {
  g(x: number): string
  g(x: string): number
  g(x: string | number): number | string {
    ...
  }
}
```

**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.

```
// 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

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

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

Esempio notevole è la funzione identità che ammette una sola implementazione possibile

```
const identity = <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?

```
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* (sul tipo `A`) 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: unknown): 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 custom type guard)

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

**Osservazione 2.4.1.** Notate che `B` **deve essere assegnabile** ad `A`. O in altre parole `B` deve essere un **sottotipo** di `A`.

> La keyword `extends` può essere usata per **mettere in relazione** due type parameter

In particolare `extends` viene spesso usata per definire dei vincoli su di un type parameter che siano basati su un altro type parameter.

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

```
export const isString = (x: unknown): 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
  }
}
```

**Esempio 2.4.3.** Riprendiamo l'esempio del capitolo precedentemente

```
const payload = `{"bar":"foo"}`
const x = parse(payload)

/*
if (typeof x === 'object') {
  // x ha tipo object | null
  if (x !== null) {
    // x ha tipo object
    const bar = (x as { [key: string]: unknown }).bar
    if (typeof bar === 'string') {
      // bar ha tipo string
      console.log(bar.trim())
    }
  }
}
*/

const isObject = (x: unknown): x is object =>
  typeof x === 'object' && x !== null

const isDictionary = (
  x: unknown
): x is { [key: string]: unknown } => isObject(x)

const isString = (x: unknown): x is string =>
  typeof x === 'string'

if (isDictionary(x)) {
  if (isString(x.bar)) {
    console.log(x.bar.trim())
  }
}
```

**Esempio 2.4.4.** Alcune custom type guard sono predefinite, un esempio notevole è `Array.isArray`

```
const payload = `{"bar":[1,2,3]}`
const x = parse(payload)

if (Array.isArray(x)) {
  // x ha tipo Array<any>
  console.log(x.keys)
}
```

Notate però che `Array.isArray` raffina a `Array<any>`.

**Esercizio 2.4.1.** Definire una versione di `Array.isArray` più type-safe

**Esercizio 2.4.2.** Definire una custom type guard che raffina un valore qualsiasi in un `Array<number>`

**Esercizio 2.4.3.** È possibile generalizzare la soluzione precedente?

## 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`.

**Osservazione 2.5.1.** Attenzione, in questo caso non stiamo parlando dell'omonimo operatore `typeof` di JavaScript, che lavora value-level, ma dell'operatore `typeof` di TypeScript, che lavora type-level.

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

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

// value-level
const X = typeof x
// "object"

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

## 2.6 Immutabilità: il modificatore `readonly`

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

**Esempio 2.6.1.** Rendere immutabile i campi di una interfaccia

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

declare const person: Person

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

È possibile rendere immutabile anche una *index signature*

```ts
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` [3]

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

type ImmutablePoint = Readonly<Point>
/*
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) {}
}
```

**Esempio 2.6.2.** È anche possibile rendere immutabile un array con il tipo predefinito `ReadonlyArray`.

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

```
const x: ReadonlyArray<number> = [1, 2, 3]
x.push(4) // error: Property 'push' does
// not exist on type 'ReadonlyArray<number>'
```

Ci sono interfacce analoghe per `Map` e `Set`, rispettivamente `ReadonlyMap` e `ReadonlySet`.

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

```
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, ricavare **il valore** delle 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 il tipo delle chiavi di una interfaccia

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

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

`keyof` può operare anche sugli array

```ts
type ArrayKeys = keyof Array<number>
/*
type ArrayKeys = number | "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

```ts
type TupleKeys = keyof [string, number]
/*
type TupleKeys = number | "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`

```
const translations = {
  when: "Quando",
  where: "Dove"
}

export declare function translate(key: string): string
```

### 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 una interfaccia

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

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

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

type Unknown = Person["foo"] // error
```

**Esercizio 2.7.2.** Ricavare il tipo del campo `baz` della seguente interfaccia

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

**Esercizio 2.7.3.** Ricavare il tipo delle chiavi del campo `bar` dell'interfaccia `Foo` definita nel precedente esercizio.

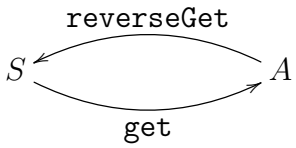
**Esercizio 2.7.4.** Tipizzare la seguente funzione `get` in modo che gli argomenti e il tipo di ritorno siano i più precisi possibile

```ts
declare function get(key: string, obj: unknown): unknown
```

**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 della prima e della seconda componente della seguente tupla

```
type Tuple = [number, string, boolean]

// estrarre la prima componente
type Num = ?
// estrarre la seconda componente
type Str = ?
```

**Osservazione 2.7.1.** Come è possibile estrarre l'**unione** dei tipi di una tupla? Accedendo con l'indice generico `number`

```
type X = [string, number]

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

**Esercizio 2.7.7.** Tipizzare la seguente funzione `set` in modo che gli argomenti e il tipo di ritorno siano i più precisi possibile

```
declare function set(
  k: string,
  v: unknown,
  o: unknown
): unknown
```

**Osservazione 2.7.2.** Si noti che `Object.keys` è tipizzato così

```
Object.keys: (o: {}) => string[]
```

**Esercizio 2.7.8.** Perchè? Quale alternativa potremmo definire?

## 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 }
/*
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.** Derivare un record di predicati dalla seguente interfaccia

```
interface X {
  a: string
  b: number
  c: boolean
}
/*
Risultato atteso:
{
  a: (x: string) => boolean
  b: (x: number) => boolean
  c: (x: boolean) => boolean
}
*/
```

**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] }
```

**Osservazione 2.8.1.** Il modificatore `?` rende opzionali tutti i campi.

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

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

**Osservazione 2.8.2.** Il modificatore `-?` rende obbligatori tutti i campi.

**Esempio 2.8.4.** `Readonly<T>`

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

**Osservazione 2.8.3.** Il modificatore `readonly` rende tutti i campi in sola lettura.

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

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

**Esempio 2.8.6.** `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`

```
declare function pick(ks: Array<string>, o: never): never
```

## 2.9 Subtyping e type parameter

Come abbiamo gà visto la keyword `extends` viene usata per estendere una classe

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

ma è spesso usata anche esprimere una relazioni tra type parameter. Vediamo un altro 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 tipo che abbia una proprietà `name`, indipendentemente dal suo tipo

**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
```

Un caso comune di utilizzo di questa feature è quello di "correggere" i typings di una libreria esterna.

## 2.11 Conditional types

I *conditional types* 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 types *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 types

**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

```
// 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`

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

**Esempio 2.11.4.** `NonNullable`

```
/**
 * 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.** `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`

## 2.12 Mapped tuples

Fino alla versione `3.0.x` i rest parameter erano tipizzabili esclusivamente con `Array`

**Esempio 2.12.1.** La funzione `Math.max`

```
/**
 * Returns the smaller of a set of supplied numeric
 * expressions.
 */
min(...values: Array<number>): number
```

Nel caso di `min` tipizzare con `Array` è corretto dato che i parametri sono omogenei (tutti di tipo `number`), ma come è possibile gestire parametri con tipi diversi?

**Esempio 2.12.2.** La funzione di utility `tuple`. Per capire perchè dovremmo aver bisogno di una tale funzione di utility si consideri il seguente

```
// x: (string | number)[]
const x = [1, "foo"]
```

Qui vorremmo che TypeScript inferisse `x: [number, string]` ma di default il type checker inferisce un valore di tipo `Array`.

Per ovviare a questo problema possiamo definire una funzione di utility `tuple`

```
function tuple<A, B>(a: A, b: B): [A, B] {
  return [a, b]
}

// x: [number, string]
const x = tuple(1, "foo")
```

E se volessimo una tupla con 3 componenti? Possiamo definire degli overloading ma è scomodo e il numero di overloading è arbitrario

```
function tuple<A, B, C>(a: A, b: B, c: C): [A, B, C]
function tuple<A, B>(a: A, b: B): [A, B]
function tuple(...t: Array<any>): Array<any> {
  return t
}

// y: [number, string, boolean]
const y = tuple(1, "foo", true)
```

Oppure possiamo gestire un numero variabile di argomenti sfruttando le nuove capacità della versione `3.1.x`

```
function tuple<T extends Array<any>>(...t: T): T {
  return t
}

// x: [number, string]
const x = tuple(1, "foo")
// y: [number, string, boolean]
const y = tuple(1, "foo", true)
```

È possibile mappare le tuple così come abbiamo visto fare con i mapped type

```
declare function toPairs<T extends Array<any>>(
  ...t: T
): { [K in keyof T]: [K, T[K]] }

// const x: [["0", number], ["1", string], ["2", boolean]]
const x = toPairs(1, "s", true)
```

**Esercizio 2.12.1.** Tipizzare la seguente funzione `all` in modo che preservi le tuple

```
function all(ps: Array<Promise<unknown>>): Promise<unknown>
```

# 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
- castare ad una tipizzazione sana
- module augmentation / declaration merging

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

```ts
// lodash@4.17.11
// @types/lodash@4.14.118
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)

https://github.com/gcanti/talks/blob/master/adt/adt.md

**Esercizio 5.0.1.** Modellare il tipo `JSON`. Si può considerare un ADT?

**Esercizio 5.0.2.** Modellare un albero binario

**Esercizio 5.0.3.** Modellare una struttura dati (chiamiamola `These`) che rappresenta alternativamente

- un successo di tipo `A`
- un errore bloccante di tipo `L`
- un successo di tipo `A` e un errore non bloccante di tipo `L`[4]

**Esercizio 5.0.4.** Definire una funzione `match` che simuli il pattern matching per `These`

**Esercizio 5.0.5.** Tennis Kata: modellare il punteggio di un game

[4]Tratto da https://package.elm-lang.org/packages/joneshf/elm-these/1.2.0/
There are a few ways to interpret the both case:

- You can think of a computation that has a non-fatal error.
- You can think of a computation that went as far as it could before erroring.
- You can think of a computation that keeps track of errors as it completes.

# 6 Error handling funzionale

Consideriamo la funzione

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

Come abbiamo visto nell'introduzione tale funzione si dice *parziale* perchè non è definita per tutti i valori del dominio (in particolare non è definita per $x = 0$).

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` (oppure un altro valore *sentinella*)

```
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))
```

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

> An expression is said to be *referentially transparent* if it can be replaced with its corresponding value without changing the program's behavior

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

```
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

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

class None<A> {
  readonly _tag: "None" = "None"
  map<B>(f: (a: A) => B): Option<B> {
    return none
  }
  chain<B>(f: (a: A) => Option<B>): Option<B> {
    return none
  }
}

class Some<A> {
  readonly _tag: "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)
head([0, 1, 2]).chain(inverse) // None
```

**Esercizio 6.1.1.** Definire la funzione `index` che, dato un array e un indice, restituisce l'elemento a quell'indice (se esiste)

**Esercizio 6.1.2.** Definire la funzione `updateAt` che, dato un array, un indice e una funzione, restituisce un nuovo array con la funzione applicata all'`i`-esimo suo elemento

## 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 None<A> {
  ...
  fold<R>(whenNone: R, whenSome: (a: A) => R): R {
    return whenNone
  }
  // variante "lazy"
  foldL<R>(whenNone: () => R, whenSome: (a: A) => R): R {
    return whenNone()
  }
}

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

Vediamo un esempio di utilizzo

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

inverse(2).foldL(whenNone, whenSome)
// "the result is 0.5"
inverse(0).foldL(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, identity)

const toUndefined = <A>(fa: Option<A>): A | undefined =>
  fa.fold(undefined, identity)
```

## 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 _tag: "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 _tag: "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)
  }
}
```

**Esercizio 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`

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

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

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

**Esercizio 6.4.2.** Definire un wrapper della seguente funzione `readFile` in modo che utiizzi `Either` nel tipo di ritorno

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

# 7 Finite 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**. [6]

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

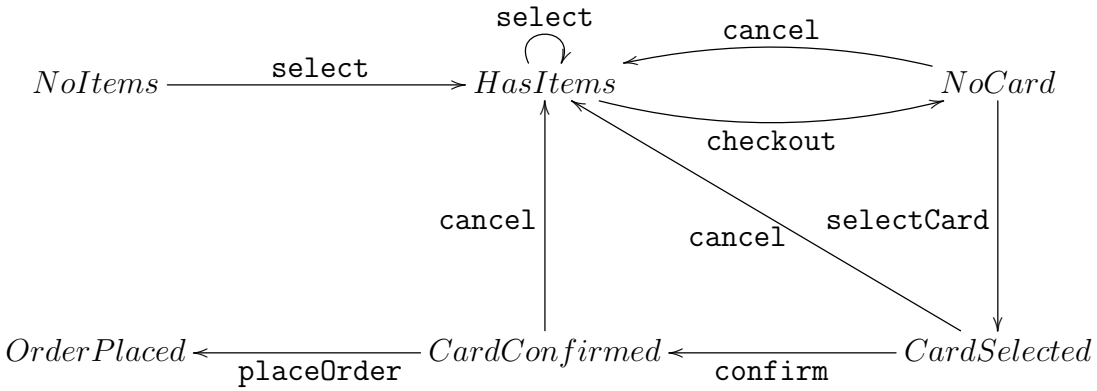
Il modello di una finite state machine è allora

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

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

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

**Esercizio 7.0.1.** Modellare la seguente macchina a stati finiti (shopping cart checkout) usando degli ADT sia per gli stati sia per gli eventi




[6] https://wickstrom.tech/finite-state-machines/2017/11/10/finite-state-machines-part-

# 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)
```

1-modeling-with-haskell.html

# 9 Simulazione dei tipi nominali

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

**Esempio 9.0.1.** Implementazione con le classi

```
class USD {
  private readonly brand!: symbol
  constructor(readonly value: number) {}
}

class EUR {
  private readonly brand!: symbol
  constructor(readonly value: number) {}
}

declare function f(usd: USD): void

f(new USD(1))
f(new EUR(1))
/*
[ts]
Argument of type 'EUR' is not assignable to parameter
of type 'USD'. Types have separate declarations of a
private property 'brand'
*/
```

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 {
  private readonly brand!: symbol
  constructor(readonly value: number) {}
}
```

Il problema è che non ho controllo sulla creazione

```
const x = new NonZero(0) // oops!
```

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 trasporta 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 {
  private readonly brand!: symbol
  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 si possono spingere 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

Un *phantom type* è un tipo polimorfico i cui type parameter non appaiono tutti alla destra della sua definizione

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

Qui `Phantom` è un phantom type e `M` è un *phantom type parameter*, dato che il parametro `M` non appare nella sua implementazione.

Dato che TypeScript ha un type system strutturale dobbiamo però scegliere una diversa implementazione.

**Esercizio 11.0.1.** Perchè?

per esempio una classe con una phantom property

```
class Phantom<M> {
  private 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> {
  private 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

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

const isNonEmptyString = (s: string): boolean => s.length > 0

export class Data<M extends State> {
  private 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 isNonEmptyString(data.input)
      ? some(new Data(data.input))
      : 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!

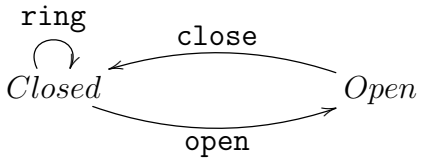
```
[ts]
Argument of type 'Data<"Validated">' is not assignable
  to parameter of type 'Data<"Unvalidated">'.
  Type '"Validated"' is not assignable to type '"Unvalidated"'.
```

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 come i phantom type possano essere sfruttati per definire una macchina a stati finiti.

**Esempio 11.2.1.** Modellare la seguente macchina a stati finiti (operazioni su una `Door`), mantenendo il conteggio delle volte in cui il campanello è stato suonato




L'implementazione si basa su questi due principi

- gli stati sono rappresentati da una struttura dati `Door` contenente il conteggio e una indicazione dello stato corrente
- le transizioni sono rappresentate da funzioni

```
type Open = 'Open'
type Closed = 'Closed'
type State = Open | Closed

class Door<S extends State> {
  private readonly S!: S
  constructor(readonly count: number) {}
}

const start = (): Door<Closed> => new Door(0)
```

Ora definiamo una operazione per ogni arco del grafo

```
const close = (door: Door<Open>): Door<Closed> =>
  new Door(door.count)

const open = (door: Door<Closed>): Door<Open> =>
  new Door(door.count)

const ring = (door: Door<Closed>): Door<Closed> =>
  new Door(door.count + 1)
```

La corretta successione delle operazioni ora è assicurata staticamente

```
close(start()) // error

ring(open(start())) // error

open(ring(close(open(ring(start()))))) // ok
```

**Esempio 11.2.2.** Volendo le operazioni possono anche essere definite come metodi, il che rende più comodo concatenarle.

Per farlo possiamo sfruttare la possibilità di annotare `this`.

Inoltre posso imporre uno stato iniziale rendendo il costruttore privato

```
class Door<S extends State> {
  private readonly S!: S
  static start(): Door<Closed> {
    return new Door(0)
  }
  private constructor(readonly count: number) {}
  open(this: Door<Closed>): Door<Open> {
    return new Door(this.count)
  }
  close(this: Door<Open>): Door<Closed> {
    return new Door(this.count)
  }
  ring(this: Door<Closed>): Door<Closed> {
    return new Door(this.count + 1)
  }
}
```

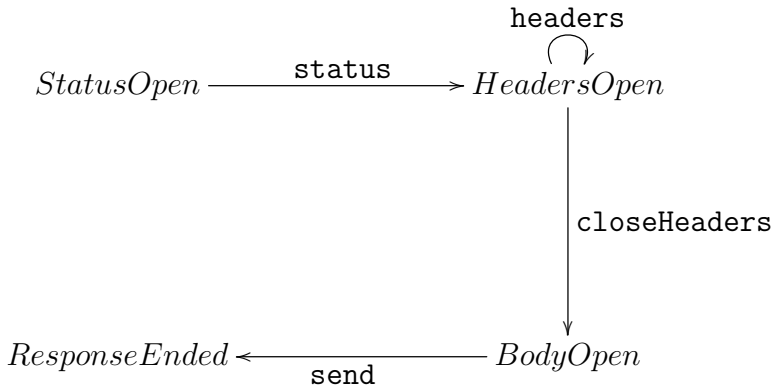
Ancora una volta la corretta successione delle operazioni è assicurata staticamente

```
Door.start().close() // error

Door.start()
  .open()
  .ring() // error

Door.start()
  .ring()
  .open()
  .close()
  .ring()
  .open() // ok
```

Posso anche richiedere un esplicito stato finale

```
const x: Door<Closed> = Door.start()
  .ring()
  .open() // error
const y: Door<Closed> = Door.start()
  .ring()
  .open()
  .close() // ok
```

**Esercizio 11.2.1.** Modellare la seguente macchina a stati finiti (ciclo di vita di una `Response` di `express`)




L'obbiettivo principale è evitare staticamente che vengano aggiunti degli header una volta che si è mandato il body.

**Esercizio 11.2.2.** Rendere type-safe il metodo `status` di `Middleware`

## 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'evento e il relativo listener

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

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

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

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

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<unknown> }

const map = {
  evt1,
  evt2
}

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

Si noti che per poter accedere al campo `D` devo modificare la sua visibilità nella definizione di `Event`

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

I tipi dei diversi eventi possono ora essere estratti in modo indipendente

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

# 12 Newtypes

Non sempre i tipi predefiniti, in particolare quelli primitivi, 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.

Un altro 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 cel: Celsius = 1

celsius2fahrenheit(cel)

const far: Fahrenheit = 33.8

celsius2fahrenheit(far) // oops...
```

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

> 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.

Tali tipi sono comunemente chiamati *newtype*.

## 12.1 Phantom type wrapper

Come abbiamo già visto una possibile soluzione è quella di creare dei wrapper simulando i tipi nominali

```ts
class Newtype<M, A> {
  private 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 far = new Fahrenheit(33.8)

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

Un "vero" **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 un newtype `S` basato su tipo `A` è *isomorfo* ad `A` possiamo usare un `Iso` per passare da uno all'altro.

```ts
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.wrap(celsiusIso.unwrap(celsius) * 1.8 + 32)

const far: Fahrenheit = fahrenheitIso.from(33.8)

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

Sfruttando il subtyping è possible codificare comportamenti interessanti

```ts
interface NonZero
  extends Newtype<
      { readonly NonZero: unique symbol },
      number
    > {}

interface Positive
  extends Newtype<
      NonZero['M'] & {
        readonly Positive: unique symbol
      },
      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) // ok
inverse(positive) // ok
mult(positive, nonZero)
// error: Property 'Positive' is missing
// in type '{ readonly NonZero: unique symbol; }'
```

**Esercizio 12.2.1.** Definire `Integer` e `PositiveInteger`

**Esercizio 12.2.2.** È possibile definire un `Iso` per `Positive`?

**Esercizio 12.2.3.** Definire delle funzioni di conversione tra `number` e il *raffinamento* `Positive`

# 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`