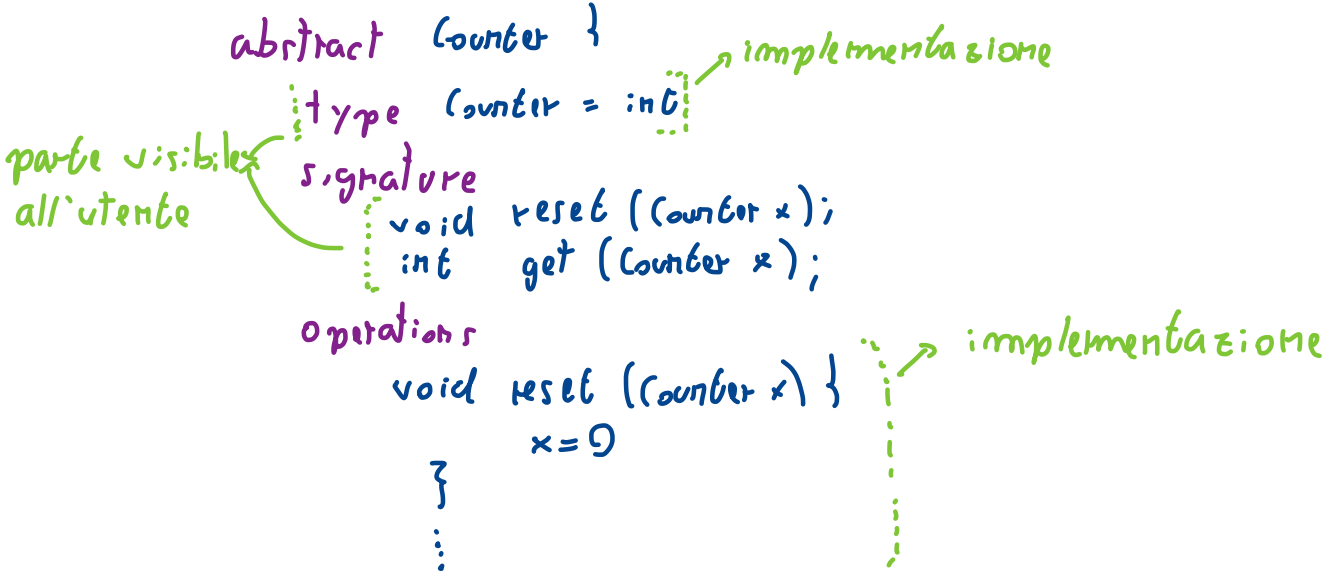
PARADIGMI ORIENTATI AGLI OGGETTI

Problema affrontato: garantire
- incapsulamento
- occultamento informazione

Tipi di dato astratti (ADT)

- Dati e modi per manipolarli in unico costrutto



Problemi con ADT

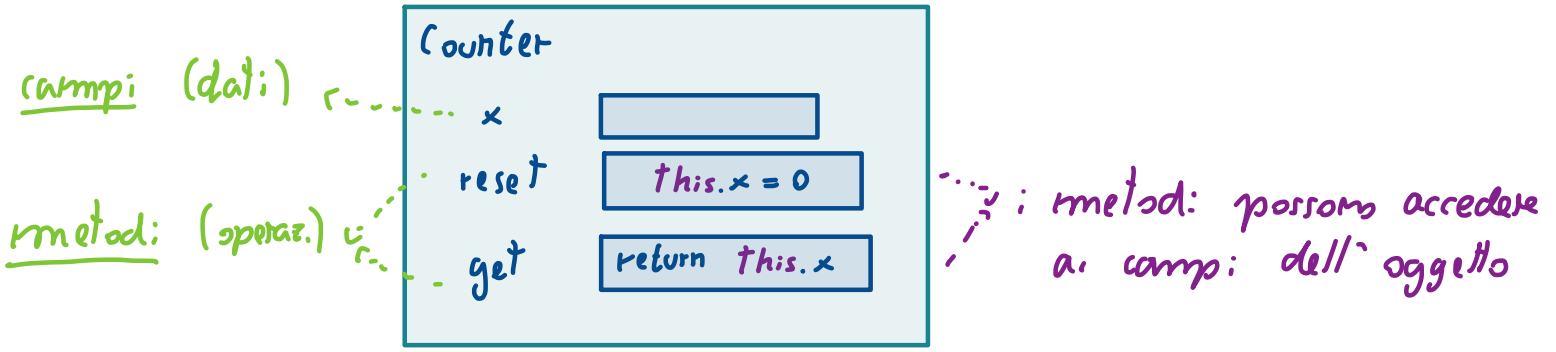
- difficili da modificare
- difficili da estendere
- non ci sono relazioni di compatibilità tra ADT

Paradigma ad oggetti

astrazione

- incapsulamento e occultamento informazione
- ereditarietà (cf. estensioni)
- relazioni di compatibilità (sottotipaggio)

Costrutto principale: oggetto: "capsula" contenente dati e operazioni



Per invocare un'operazione su un oggetto gli si manda un messaggio

computazione = comunicazione

myCounter.reset

classe = meccanismo per raggruppare oggetti con la stessa struttura

Una classe stabilisce

- quali sono i campi/dati degli oggetti contenuti nella classe (quanti, di quale tipo, con quale visibilità)
- nome, segnatura, visibilità, e implementazione dei metodi

se l'implementazione manca, si parla di: classi astratte

```
class Counter {
    private int x
    public void reset () {
        x = 0
    }
    public int get () {
        return x
    }
}
```

```
Counter c = new Counter ()
```

ISTANZE

Esiste una relazione naturale tra una classe e gli oggetti che sono istanza di quella classe

Ogni classe induce un tipo, e diciamo che un oggetto è istanza della classe se ha il tipo indotto dalla classe

Classe ⟶ Tipi ↗ Relazione di compatibilità

$\forall \alpha . \alpha \to \alpha$

$int \to int$

- T comp S [se ogni valore di tipo T è ammesso in un qualsiasi contesto in cui sarebbe richiesto un valore di tipo S]

Es. $\forall \alpha . T$ comp $T[S/\alpha]$?

$T[S/\alpha]$ comp $\forall \alpha . T$?

Sottotipaggio T subtype S $\iff$ ogni messaggio compreso da oggetti di tipo S, è compreso anche da oggetti di tipo T

T subtype S $\Rightarrow$ T ha tutti i campi di S (e magari altri)

campi pubblici:
- tipo
- nome

$\rightsquigarrow$ interfaccia
- campi pubblici
- segnatura dei metodi pubblici
  - nome
  - tipo
  - nome e tipo parametri

NB. T subtype S $\iff$ interfaccia(T) $\supseteq$ interfaccia(S)

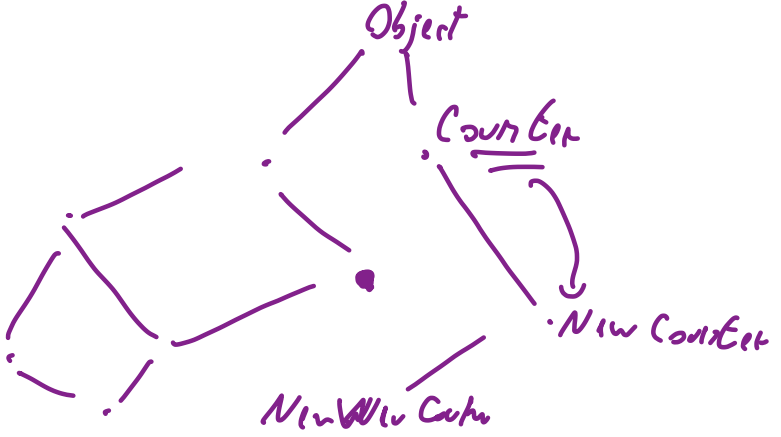
Es. In Java abbiamo sottotipaggio nominale; in OCaml strutturale

```
class Counter {
  public String name
  public void foo() {...}
  private void bar() {...}
}

class NameCounter extends Counter {
      public String name
      public void foo() {...}
      public string get_name() {
                  return name }
   }
```

→ non appartiene all'interfaccia

metodo aggiuntivo



NewCounter <: Counter

Istanze (NewCounter) ≤ # Istanze (Counter)

Interfaccia (NewCounter) ≥ Interfaccia (Counter)

$\mathbb{R} \cong \mathbb{Z}$

Float <: Int

Istanze (Float) ≤ # Istanze (Int)

Interfaccia (Float) ≥ Interfaccia (Int)

(Float → Bool) <: (Int → Bool) → Int

Float

$F : Type \to Type$

- Covariant:

$$T <: S \implies F(T) <: F(S)$$

- Contravariant

$$T <: S \implies F(S) <: F(T)$$

$$F_A(T) = T \to A$$

$$F_{int}(T) = T \to int$$

$$F_{Bool \to Int}(T) = T \to (Bool \to Int)$$

$S <: T$ $\qquad$ $T \to A <: S \to A$ $\qquad$ $f : S \to A$

$g : T \to A$

T :p: come Proposizioni

$A \to B$

$T \vdash S \implies T \to A \vdash S \to A$

$T \to A, S \vdash A$

piove ⊢ ombrello piove → ombrello ⊢ ombrello

$x_1 : \tau_1, \ldots, x_m : \tau_m \vdash E : \tau$

int → int * int