Linguaggi di Programmazione

Ivano Salvo

OOP2: Introduzione agli Oggetti
FP3: programmazione su liste

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **4**, 20 ottobre 2020

Lezione 4a:

Ereditarietà e Subtyping

Cosa distingue Oggetti da ADT?

Abbiamo visto l'esempio dei **numeri razionali**: abbiamo visto il costrutto `class` che assomiglia a una `struct`: per ora aggiunge **restrizioni di visibilità** (**information hiding**) e la nozione di **metodo** che permette di definire le funzioni sul tipo di dato definito (**incapsulamento**).

Anche se programmati in Linguaggio Orientato agli Oggetti, di fatto si tratta semplicemente di un **tipo di dato astratto**.

Gli ingredienti **in più** dell'OOP sono due:

- **Ereditarietà:** è possibile **estendere il comportamento di una classe**, riutilizzando il codice già scritto (senza bisogno di duplicarlo).
- **Sottotipaggio:** in un contesto in cui mi aspetto un tipo (ad esempio il parametro di un metodo o di una funzione) posso accettare qualunque **oggetto di un sottotipo**.

Sottotipaggio ed Ereditarietà sono spesso legati.

Punti, Punti Colorati e Bidimensionali

Prima di capire come effettivamente si utilizzano nella progettazione dei programmi, vedremo questi meccanismi in un esempio canonico: la gerarchia di **punti** e **punti colorati** e **punti bidimensionali**.

Benché non molto eccitante, si tratta dell'esempio minimale canonico di ereditarietà.

- I **punti** hanno una coordinata sulla retta (per semplicità sarà intera), e un metodo `move` per sportarli di un certo `dx`.
- I **punti colorati** hanno in più un colore, e quando sono troppo distanti dall'origine diventano sempre neri.
- I **punti bidimensionali** sono colorati e hanno una seconda coordinata.

Vedremo come il codice si può scrivere in modo **incrementale**.

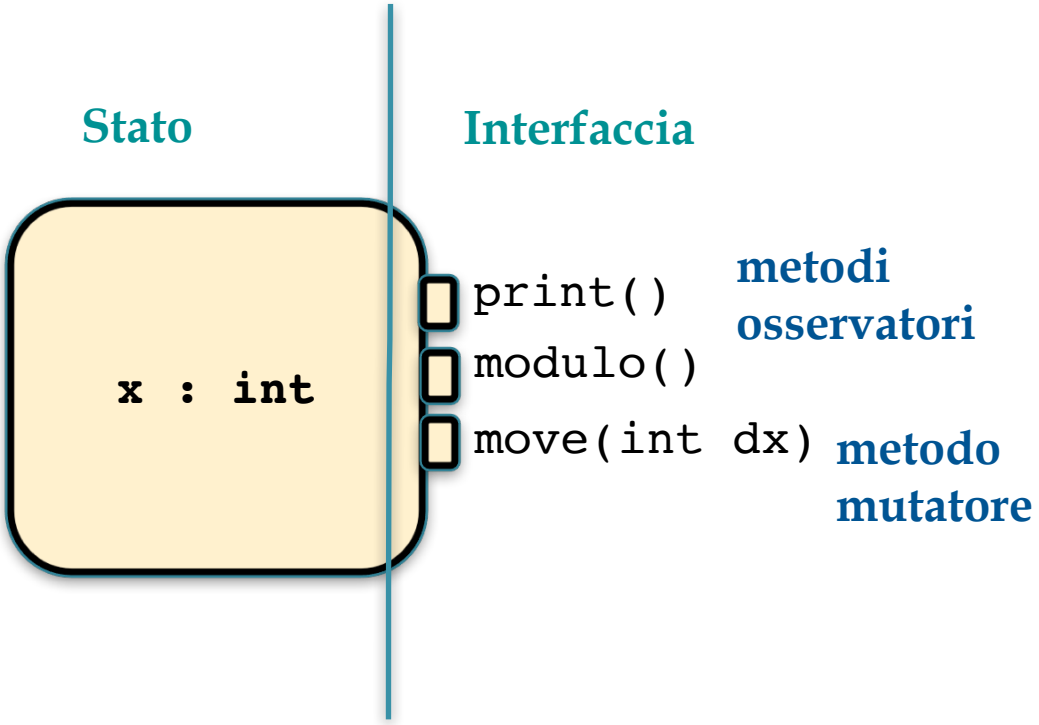
Punti mobili: codice

```
class Punto {
    int x;
  public:
    Punto(int v);
    Punto();

    void move(int dx);
    virtual int modulo();
    virtual void print();
}; /* questo in punti.h */

  /* costruttori */
Punto::Punto(int v) { x=v;}
Punto::Punto() { x=0; }

 /* codice dei metodi */
void Punto::move(int dx){x += dx;}
int Punto::modulo(){return abs(x);}
void Punto::print(){
     printf("pos: %d ",x);
} /* questo in punti.cpp */
```

Cosa abbiamo definito?

Possiamo tornare alla nostra metafora degli oggetti come piccoli automi con stato nascosto.

I costruttori servono a creare questi automi (**produttori**)



Punti mobili colorati (1)

```
class PuntoCol : Punto {
    Colore c;
  public:
    PuntoCol();
    PuntoCol(int x);
    PuntoCol(int x, Colore c);

    void move(int dx);
    virtual void print();
};

  /* costruttori */
PuntoCol::PuntoCol(){
    Punto();
    c.setWhite();
}

PuntoCol::PuntoCol(int v){
    Punto(v);
    c.setWhite();
}
```

I punti colorati sono punti ed ereditano dalla classe Punto. Aggiungono solo l'info colore

occorre ridichiarare i metodi che verranno ridefiniti (overriden). Il metodo **`modulo`** *viene ereditato*

si può chiamare il costruttore della superclasse. Anzi, si deve... se non lo fate chiamerà automaticamente il costruttore senza parametri

La variabile `x` *di* `Punto` *è privata e non è visibile in* `PuntoCol`

Punti mobili colorati (2)

```
/* costruttori - cntd*/
PuntoCol::PuntoCol(int v, Colore col){
    Punto(v);
    c=col;
}

/* metodi*/
PuntoCol::move(int dx){
    Punto::move(dx);
    if (this.modulo() > 3) then c.setBlack();
}

PuntoCol::print(){
    Punto::print();
    printf("col: ");
    c.print();
}
```

Anche i metodi possono chiamare i super-metodi

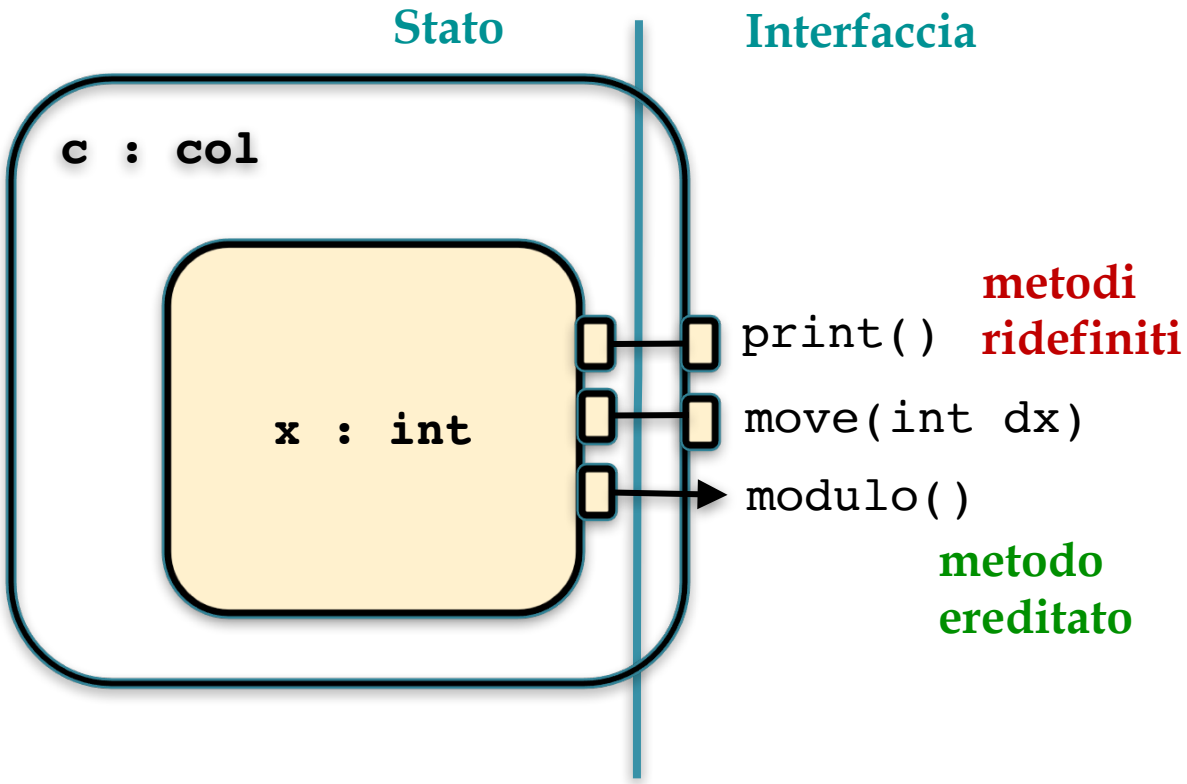
Il metodo modulo viene ereditato dalla superclasse Punto

Tipica tecnica di programmazione: delegation: un oggetto chiede a un altro di fare qualcosa

Cosa abbiamo definito?

Abbiamo usato i Punti (**classe base** o **superclasse**) per definire incrementalmente i punti colorati (**classe derivata** o **sottoclasse**).

Alcuni metodi vengono ridefiniti (**overriding**) altri vengono semplicemente ereditati.



Subtyping

`PuntoCol` è **sottotipo** di `Punto`. Possiamo usare un `PuntoCol` in ogni contesto in cui si prevede `Punto`.

Le seguenti funzioni e main **compilano correttamente**.

`&p` è una **reference**: si tratta di una forma **edulcorata di pointer**: le reference possono essere **solo indirizzi di oggetti** sullo stack. Dopodiché però non si usano le notazioni dei pointers (ad esempio `->`), ma quelle delle variabili (.).

```
whichPointRef(Punto &p){
   p.print();
   p.move(3);
   p.print();
}

whichPoint(Punto p){
   p.print();
   p.move(3);
   p.print();
}

void main{
   Puntocol pc(1);
   whichPointRef(pc);
   whichPoint(pc);
}
```

Dynamic Lookup (1)

Ma quali metodi verranno eseguiti su p? Quelli di Punto o di PuntoCol?

```
whichPointRef(Punto &p){
   printf("WPRef\n");
   p.print();
   p.move(3);
   p.print();
}
void main{
   PuntoCol pc(1);
   whichPointRef(pc);
   whichPoint(pc);
}
```

```
whichPoint(Punto p){
   printf("WP\n");
   p.print();
   p.move(3);
   p.print();
}
```

```
Output:
WPRef:
pos: 1 col: Bianco
pos: 4 col: Bianco
WP:
pos 1
pos 4
```

Quando **p è una reference** e il **metodo è virtual** (come `print`) il metodo eseguito è di `PuntoColorato` (**Lookup Dinamico** = dipende **dall'oggetto puntato**, **non dal tipo della variabile**).

`move` **non è virtual** e il punto rimane Bianco. **Sperimentare!**

Dynamic Lookup (2)

Lo stesso effetto si ottiene con i pointer: per **subtyping posso assegnare un `PuntoCol*` a una variabile `Punto*`**. Dopodiché i metodi virtual dipenderanno dall'oggetto ricevente.

Notare il **ruolo attivo** dell'**oggetto ricevente** che lo distingue nettamente da un parametro di una funzione!

Immaginiamo stavolta, di avere **una versione virtual** del metodo `move`, `moveVirtual`: in questo caso **viene eseguito il metodo di `PuntoCol`** che colora di nero i punti troppo lontani dall'origine.

```
void main{
   Punto *p = new PuntoCol(1);
   p->print();
   p->moveVirtual(3);
   p->print();
}
```

Output:

```
pos: 1 col: Bianco
pos: 4 col: Nero
```

Punti bidimensionali (1)

```
class PuntoBid : PuntoCol {
    int y;
  public:
    PuntoBid(int v, int u,
        Colore col);
    PuntoBid();

    void move(int dx, int dy);
    virtual void print();
    virtual int modulo();
};

  /* costruttori */
PuntoBid::PuntoBid(){
   PuntoCol();
   y = 0;
}
PuntoBid::PuntoBid(int v, int u, Colore col){
   PuntoCol(v, c);
   y = 0;
}
```

Questa `move` *non ridefinisce quelle precedenti: ha due parametri invece di uno: fa overloading. La vecchia* `move` *viene ereditata da* `PuntoCol`

viene viceversa ridefinito il metodo `modulo` *che era stato definito nella classe* `Punto`.

Punti bidimensionali (2)

```
   /* metodi */
PuntoBid::move(int dx, int dy){
    y += dy;
    PuntoCol::moveVirtual(dx);
}

PuntoBid::modulo(){
    return max(abs(y), Punto::modulo());
}

PuntoBid::print(){
    PuntoCol::print();
    printf("posy: %d ", y);
}
```

Osservate come il codice riusi i metodi definiti nelle superclassi.
Sorge ora il problema di capire quali metodi vengano effettivamente invocati

Dynamic Lookup (3)

La `move` a due parametri di `PuntoBid` chiama la `moveVirtual` definita in `PuntoCol`: questa chiama il metodo `modulo`. Per dynamic lookup sarà eseguito quello di `PuntoBid` (**un metodo che ancora non esisteva quando è stata definita** `moveVirtual`).

Il punto si colora di nero perché si è allontanato dall'origine sulla seconda coordinata.

Il metodo `move` ha cambiato comportamento (**specializzato**) perché si è specializzato `modulo`: **la ricerca del metodo da eseguire riparte sempre dal primo oggetto ricevente**!

```
void main{
    /* creiamo un punto Bianco nell'orgine x=0, y=0 */
   Punto *p = new PuntoBid();
   p->print();
   p->move(0, 4);
   p->print();
}
```

Output:

```
pos: 0 col: Bianco posy: 0
pos: 0 col: Nero posy: 4
```

Dynamic Lookup: osservazioni

Nella comunità di C++, il **dynamic lookup** (aka: **late binding**) è chiamato **polimorfismo**, la parola che (**attenzione!**) la comunità funzionale di Haskell riserva ai tipi contenenti variabili di tipo.

Il dynamic lookup rende **flessibili** metodi anche **verso il futuro**: spesso basta specializzare un metodo chiamato per specializzare un metodo più complesso.

Slogan SmallTalk: **"Write Stupid Methods!"** (max 7 righe 🤣): scrivere metodi che chiamano altri metodi.

Al contrario dei linguaggi OOP puri, in C++ il dynamic lookup **non è lo standard**, ma va dichiarato dal programmatore con la keyword `virtual` (ragioni di **efficienza**)

```
doSomething(…){
   doThis(…);
   doThat(…);
   doOtherThings(…)
}
```

Subtyping: osservazioni

Il **subtyping** è una relazione d'ordine tra i tipi (riflessiva, transitiva) e usualmente si scrive <.

Il subtyping in C++ nasce prevalentemente dall'**ereditarietà**: il **tipo** definito dalla **classe derivata** (**sottoclasse**) **è sottotipo** del tipo originato dalla **classe base** (**superclasse**).

Più in generale, un sottotipo ha **un'interfaccia più ricca** di un suo supertipo.

Il subtyping introduce una **flessibilità sui tipi**: se un metodo/funzione tipa assumendo un parametro di tipo *T*, funzionerà correttamente se passo un oggetto di tipo *T'*<*T* (**gli oggetti dei sottotipi rispondono a più metodi!**).

Quindi in ogni contesto in cui ho una variabile di tipo *T** o *T*&, questa può correttamente puntare un oggetto di tipo *T'* con *T'*<*T*: **non accadrà mai di invocare un metodo su un oggetto che non sa rispondere** (**message-not-understood**)

Lezione 4d:

Programmazione su Liste in Haskell

Tipi predefiniti e Costruttori di Tipo

Abbiamo già visto alcuni tipi predefiniti (o **tipi base**) come `Bool` e `Integer`. Ce ne sono altri come `Char`, `String`, `Float`, `Double` con significato spero chiaro.

Osserviamo che alcuni valori (ad esempio costanti intere come 42) possono avere tutti i tipi numerici (Integer, Float, Double).

Abbiamo già visto due costruttori di tipo: le **coppie** (e più in generale le **tuple**) e le **funzioni**.

```
>:t (False, True)
(False, True) :: (Bool, Bool)
 -- le tuple hanno dimensione arbitraria
 -- e possono essere non omogenee
>:t (“LdpMat”, 42, True, ‘E’)
(“LdpMat”,42,True,‘E’)::(String,Integer,Bool,Char)
```

Liste

Nella tradizione dei Linguaggi Funzionali, c'è predefinito il costruttore di tipo **lista**: `[]`.

Il **cons** (inserzione in testa) è un operatore infisso e si scrive `:`.

La scrittura `[1,2,3]` è zucchero sintattico per `1:2:3:[]`.

```
 -- la costante lista vuota ha tipo polimorfo
>:t []
[] :: [t]
 -- il cons si scrive : (infisso)
>:t (:)
(:) :: t -> [t] -> [t]
 -- notazioni abbreviate per le liste
> if 1:2:3:[]==[1,2,3] then 1 else 2
1
```

Funzioni su Liste

Tutte le liste hanno la forma `x:xs`, `x` **testa** e `xs` **coda**: gli Haskellioti usano un nome che finisce in s (plurale) per denotare liste. Possiamo usare questo fatto per scrivere funzioni per **pattern matching**.

Attenzione! l'applicazione di **funzione associa sempre più di tutto** e quindi `f x:xs` viene inteso `(f x):xs` e non `f (x:xs)` Occorre mettere le parentesi se necessario.

```
 -- testa e coda si scrivono facilmente
testa (x:_) = x
coda (_:xs) = xs
> testa [1,2,3]
1
>:t testa
testa :: [t] -> t
> coda [1,2,3]
[2,3]
>:t coda
coda :: [t] -> [t]
```

Pattern matching non esaustivi

Testa e coda **non sono definite** su lista vuota: significa che ci sono dei **casi che sfuggono al pattern matching**.

In questi casi viene sollevata **un'eccezione**, cioè un errore

… che può eventualmente essere **catturato** e **gestito**.

```
 -- xs fa matching con lista vuota sulla lista [42]
> coda [42]
[]
 -- testa e coda non sono definite su lista vuota
> testa []
*** Exception: lezione2.hs:2:1-16: Non-exhaustive
patterns in function testa
 -- la funzione predefinita head si comporta in
 -- modo leggermente diverso
> head []
*** Exception: Prelude.head: empty list
 -- l'eccezione è intercettata e trasformata nel suo
 -- significato `logico'
 -- In questo caso eentrambe le funzioni si bloccano
```

Annidamento dei costruttori di tipo

Le liste si possono annidare: è facile avere le **liste di liste**.

Scriviamo una funzione che sostituisce ogni elemento di una lista con la lista contenente quell'elemento…

… oppure la lista dei suffissi [`xs@(h:txs)` permette di riferire a destra di = sia la lista (`xs`) che le sue componenti (`h` e `txs`)].

```
> :t [[]]
[[]] :: [[t]]

lol (x:xs) = [x]:lol xs
lol [] = [[]]
> :t lol
lol::[t]->[[t]]
> lol [1,2,3]
[[1],[2],[3]]

suffissi xs@(_:txs) = xs:suffissi txs
suffissi     []     = [[]]
> suffissi [1,2,3]
[[1,2,3],[2,3],[3],[]]
```

Schemi di programmi (1)

Molti programmi hanno una struttura comune: ad esempio iterano una stessa funzione su una lista.

Un funzionale può generalizzare questa forma di ricorsione: è chiamato spesso **reduce**, ma in Haskell si chiama `foldl` (vedremo altre versioni simili, come `foldr`).

```
 -- La funzione length
myLength (_:xs) = 1 + myLength xs
myLength   []   = 0
 -- La funzione sum è simile a length
mySum (x:xs) = x + mySum xs
mySum   []   = 0
 -- La funzione produttoria
myProd (x:xs) = x * mySum xs
myProd   []   = 1

 -- Possiamo generalizzare...aka reduce
myFold f g (x:xs) = f x (myFold f g xs)
myFold f g []     = g
myFold :: (t -> t1 -> t1) -> t1 -> [t] -> t1
```

Schemi di programmi (2)

C'è un gusto tutto funzionalista di scrivere funzioni come composizione di altre funzioni (one-liner) senza fare ricorsione esplicita decomponente liste o altre strutture dati.

Può a volte aiutare anche il compositore, che esiste già predefinito e si chiama (.)

```
-- e quindi...
> myLength' = myFold (\x y->y+1) 0
> mySum' = myFold (+) 0
> myProd' = myFold (*) 1
 -- ma anche...
> myLength'' = myFold (\x -> (+1)) 0

 -- il funzionale composizione di funzioni
c f g x = f (g x)
c :: (t1 -> t) -> (t2 -> t1) -> t2 -> t
 -- oppure infisso predefinito
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Schemi di programmi (3)

Un famoso funzionale è l'**apply-to-all**, noto soprattutto come `map`, che applica una funzione a tutti gli elementi di una lista.

C'è una versione binaria: `zipWith`.

Volendo c'è pure quella *n*-aria. Vedremo come generalizzare.

```
 -- map applica f a tutti gli elementi di una lista:
myMap f (x:xs) = f x : myMap f xs
myMap f [] = []
myMap :: (t -> t1) -> [t] -> [t1]
 -- ad esempio:
> myMap (+1) [41,36,72]
[42,37,73]
> myMap (\x->[]) [42,37,73]
[[42],[37],[73]] -- myMap (\x->[x]) . (+1) [41,36,72]

 -- speso è utile una versione 'binaria' di map
myZipWith f (x:xs)(y:ys)=f x y: myZipWith f xs ys
myZipWith f [] _ = []
myZipWith f _ [] = []
myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Schemi di programmi (4)

Tra i casi particolari, molto famoso è `zip`, o **cerniera** che accoppia ordinatamente gli elementi di una lista.

Può essere ottenuto facilmente da `zipWith`.

Possiamo anche immaginare di applicare una lista di funzioni a una lista di argomenti…

```
 -- speso è utile una versione 'binaria' di map
myZip (x:xs)(y:ys)=f x y: myZipWith f xs ys
myZip [] _ = []
myZip _ [] = []
myZip :: [a] -> [b] -> [(a,b)]

 -- ma ovviamente
myZip' = myZipWith (\x y ->(x,y))

applyList (f:fs)(x:xs)=f x: applyList fs xs
applyList [] _ = []
applyList _ [] = []
applyList :: [t -> t1] -> [t] -> [t1]
```

Una famosa applicazione…

John Backus (progettista del Fortran) nel 1978 scrisse un celeberrimo articolo che rilanciò la programmazione funzionale ponendo l'accento sulle sue **virtù composizionali**.

Fece l'esempio del **prodotto scalare**.

```
 -- prodotto scalare con myFold e myZipWith:
prodottoScalare xs ys =
   myFold (+) 0 (myZipWith (*) xs ys)

 -- ma anche con map e applyList
prodottoScalare' xs ys =
   myFold (+) 0 (applyList (myMap (*) xs) ys)

 -- ma a ben vedere
 -- giocando con la composizione
prodottoScalare'' xs =
     mySum . ((applyList . (myMap (*))) xs)
```

Lezione 4

That's all Folks…

Grazie per l'attenzione…

…Domande?