Programmazione a Oggetti
Lezione 3

Il linguaggio SmallTalk

Perché Smalltalk?

- Primo linguaggio Object-Oriented
- Object-Oriented Puro (everything is an object)
- Molte nuove idee (allora) ancora popolari (oggi)

Smalltalk: caratteristiche 1

- Core language molto conciso
  - pensato attorno un'unica metafora di astrazioni (oggetti)
- Ricca gerarchia (almost standard) di classi predefinite
- L'object orientation favorisce una naturale integrazione (riuso) e adattabilita' delle classi predefinite (via specializzazione)

Smalltalk: caratteristiche 2

- Ambiente di Sviluppo
  - La programmazione e' guidata da un'interfaccia grafica (system browser)
- Smalltalk genera un bytecode interpretato da una virtual machine (portabilita')
- Smalltalk e' tipato dinamicamente
- Gestione della memoria implicita (garbage collector)

Good News: free books!

All'URL:

http://www.iam.unibe.ch/~ducasse/WebPages/FreeBooks.html

trovate libri in pdf da scaricare. In particolare:

- John Hunt "Smalltalk and Object-Orientation: an introduction" **(il piu' aderente ai temi trattati nel corso)**
- Alex Sharp "Smalltalk by example"
- Ivan Tomek "The Joy of Smalltalk"
- Kaheler & Patterson "A taste of Smalltalk" **(divertente, da leggere)**

Motto:everything is an object

```
tmp := 2 + 5
```

Sintassi: simile ad altri LdP

Semantica: invia il messaggio **+** all'oggetto **2**, con parametro **5** e crea un nuovo oggetto col risultato dell'applicazione del metodo da assegnare a **tmp**

Formato Messaggi 1

- Messaggi Unari
  - **`5 even`**

    Invoca il metodo **`even`** sull'oggetto 5, fornendo come risposta l'oggetto **`FALSE`**

    5 e' il ricevente e il messaggio invoca un metodo
  - **`myPoint := Point new`**

    Il messaggio **`new`** inviato alla classe **`Point`** genera una nuova istanza (**`Point`** e' una classe)
- Parsing: left-to-right
  - **`Point new getX`**

    Si crea una istanza e si ispeziona la posizione

Formato Messaggi 2

- Messaggi Binari

  Sintassi simile alle usuali operazioni aritmetiche

  ```
  tmp := 5 + 2
  ```

- Parsing: left-to-right

  ```
  2 + 3 * 5
  ```

  invia il messaggio + all'oggetto 2 con parametro l'oggetto 3, genera un nuovo oggetto, il numero 5, che riceve il messaggio * con parametro 5

  Attenzione: no precedenza tra operatori

- Usuali operazioni aritmetiche e di confronto

Formato Messaggi 3

- Keyword messages

```
myPoint:= BiDimPoint new move: 3 and: 2
```

move: and: e' il selettore del metodo mentre 3 e 2 sono i parametri

- Usare parentesi per risolvere ambiguita'

Assegnamento

- L'assegnamento e' "by reference"

```
myObject := newObject := oldObject
```

crea tre referenze allo stesso oggetto

Ogni statement restituisce un valore

Definizione di Classi

```
Object subclass: #Point
    instanceVariableNames: 'xcoord'
    classVariableNames: 'howMany'
    poolDictionary: ''
    category: 'Graphics'
```

Object e' il nome della superclasse (**Object** e la root class)

xcoord mantiene la posizione del punto, ed ogni istanza creata ha la sua propria copia

howMany contiene il numero totale di punti creati. Esiste una sola copia di questa variabile

Definizione di Metodi

```
xcoord
  ^xcoord.
xcoord: aNumber
  xcoord := aNumber.
move: dx
  |tmp|
  tmp:= self xcoord + dx.
  self xcoord: tmp.
distOrig ^self xcoord abs.
```

Return statement

Ritorna di default l'oggetto ricevente

Variabile locale

e self?

Self

- pseudovariabile che indica l'oggetto che riceve il metodo
- Viene legata dinamicamente
- Attenzione puo' essere legata a oggetti di classi diverse da quella che sto definendo (tipicamente istanze di sottoclassi)

Metodi di inizializzazione

```
initialize
   self xcoord: 0.
   howMany:=howMany + 1.

new
   ^super new initialize.
```

Metodo di classe

Chiama il metodo **new** della superclasse **Object**

Senza ^ torna self, cioe' la classe!!

e super?

Super

- pseudovariabile che indica di selezionare un metodo a partire dalla superclasse
- viene legata staticamente
- serve tipicamente, nella ridefinizione di un metodo, a sfruttare il codice gia' scritto

- evita cicli infiniti (presto vediamo il method lookup in dettaglio)

Cosa abbiamo definito?

Class: Point	
Variabili di Istanza: xcoord **Metodi**: xcoord, xcoord:, move:, distOrig	**Variabili di classe**: howMany **Metodi di classe**: new

Definizione di Sottoclassi

```
Point subclass: #ColoredPoint
instanceVariableNames: 'color'
classVariableNames: ''
poolDictionary: ''
category: 'Graphics'
```

La classe ColoredPoint eredita
le caratteristiche della classe **Point** e la estende
con una nuova variabile di istanza

color mantiene il colore del punto

Specializzazione di Metodi

```
color
  ^color.
color: aString
  xcoord := aString.
move: dx
  super move dx
  (self distOrig > 3)
     ifTrue: [self color: 'red']
     ifFalse: [self color: 'black']
```

Ridefinisce (specializza) il metodo move

Punti bidimensionali

```
ColoredPoint subclass:
  #BidimensionalPoint
instanceVariableNames: 'ycoord'
classVariableNames: ''
poolDictionary: ''
category: 'Graphics'
```

Late Binding

```
movex: dx
   super move: dx.
movey: dy
   ycoord := ycoord + dy.
distOrig
   ^super distOrig max: (ycoord abs).
move: dx and: dy
   (self movey: dy) movex: dx.
```

Come viene risettato il colore per effetto di **move**?

E se inverto **movex** e **movey**?

Late Binding

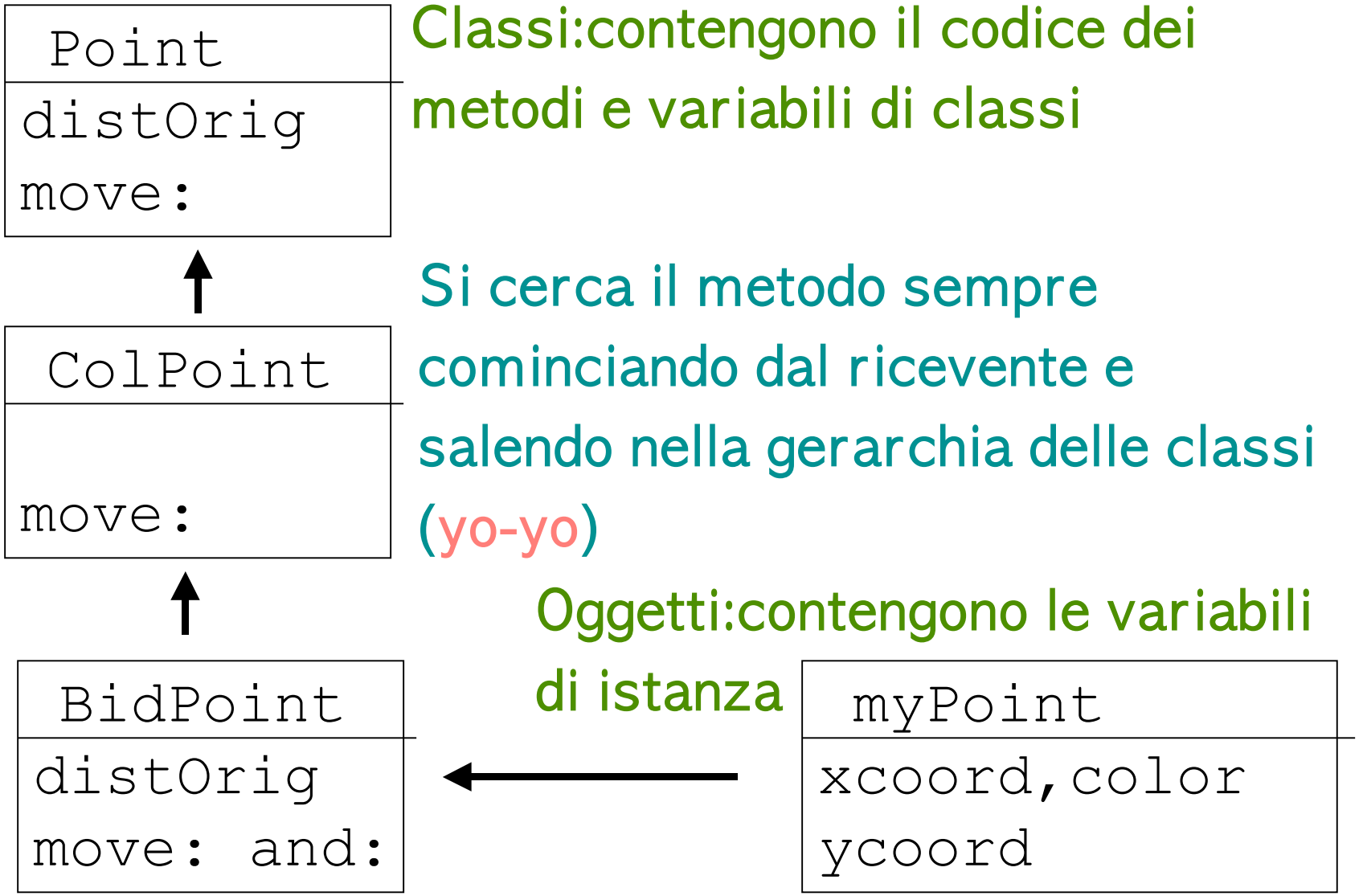
```
myPoint:= (BidimensionalPoint new move: 1
  and: 3) color.
```

Il risultato sara' black oppure red ?

Quale sequenza di invocazioni di metodi viene generata?

E chi sono i riceventi?

Method Lookup



Inheritance/Subtyping

- Osservate che tra BidimensionalPoint e Point non c'e' una relazione di tipo IS-A
- Non c'e' neanche la relazione intuitiva di sottoinsieme tra l'insieme dei punti monodimensionali e quelli bidimensionali
- BidimensionalPoint e' sottoclasse di Point perche' ha piu attributi (specializzazione procedurale)

Istanziazione di Oggetti 1

- Osservate che e' sufficiente riscrivere il metodo `initialize`. Nella classe radice

```
new
^self basicNew initialize.
```

- Nelle classi figlie:

```
initialize
super initialize.
 do somethingElse.
```

- A volte e' utile/opportuno richiedere dei valori iniziali alla creazione degli oggetti

Istanziazione di Oggetti 2

- Non e' sempre opportuno affidarsi alla buona volonta' del cliente della classe

```
myPoint := Point new.
(myPoint xcoord: 1) ycoord: 1
```

- Utile definire metodi inizializzatori appositi e "cancellare" new

```
xcoord: initX ycoord: initY
^super new move: initX and: initY
new self error 'use initX: initY'
```

Regole di Visibilità

- Metodi pubblici, variabili private

- Non e' possibile definire metodi privati
  - Si ricorre di solito a convenzioni sui nomi

    `myMethod`
- Le classi sono analoghe a variabili globali (come le variabili globali iniziano con una maiuscola)

Accesso alle variabili di istanza

- Via accessors methods (*getters* and *setters*)
  - Usati sistematicamente, violano l'information hiding
  - Attiva method lookup per compiti banali
  - Potrebbe essere utile quando cambia implementazione per mantenere l'interfaccia
  - Evita confusioni tra variabili locali e di istanza
- Diretto
  - Piu' efficiente
  - Salvaguarda l'information hiding
  - Evita proliferazione di codice poco significativo computazionalmente

Questioni di Stile...

- Usare nomi significativi
- Nomi formati da piu' parole, cominciano con minuscola e usano maiuscole per separare le parole
- Scrivere metodi dal codice corto (7 linee in media!)
- Metodi che usano solo i parametri e non variabili di istanza o self sono probabilmente nel posto sbagliato
- Evitare classi con troppi metodi