Polimorfismo (cenni), classi astratte, interfacce, classi interne

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

Programmazione, lezione 19
19 aprile 2004

Dove siamo

- Programmazione strutturata
- Strutture di controllo
  - Sequenza, selezione, iterazione
- Array
- Metodi
- TDA
- Oggetti, scambio messaggi, ereditarietà

Prossime 6 lezioni/esercitazioni

- Fine OO: polimorfismo (cenni)
- OO in Java
  - Classi astratte, interfacce, classi interne
- Rassegna API
  - Uso package, `Object`, `String`, …
  - File (eccezioni)
  - GUI: AWT
  - Cenni computabilità

Calendario

- Lezioni (con me)
  - 19 - Lun. 19/4
  - 20 - Mer. 21/4
  - 21 - Mer. 28/4
  - 22 - Mer. 5/5
  - 23 - Mer. 12/5
  - 24 - Mer. 19/5
- Esercitazioni (Lucio Ieronutti)
  - 19 - Mar. 20/4
  - 20 - Mer. 28/4
  - 21 - Mer. 5/5
  - 22 - Mer. 12/5
  - 23 - Mar. 18/5
  - 24 - Mer. 19/5
- (Laboratorio comunque a vostra disposizione…)

Esame

- Traccia esecuzione: condizione necessaria
- Scritto + progetto (facoltativo) + orale
- Voto max. senza progetto: 27
- Progetto dà incremento di 0-3 punti
  - Solo se voto scritto ≥ 21!!
  - Se voto <21, non consegnato, non presentato ⇒ progetto annullato (come non fatto)
- Se voto scritto ≤ 10 ⇒ -5 all'appello succ.!!

Oggi

- Polimorfismo (cenni)
- OO in Java:
  - Classi astratte
  - Interfacce
  - Classi interne (cenni)

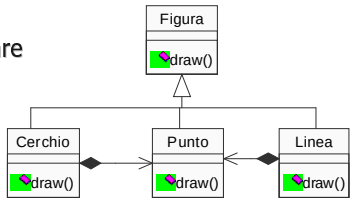
Polimorfismo (1/2)

- Programma di grafica
- Struttura dati per tutte le figure
- Tanti array…
- Scomodo!


```
Punto[] punti;
Cerchio[] cerchi;
...
if (x instanceof Punto)
  punti[i] = x;
else if (x instanceof Cerchio)
  cerchi[i] = x;
else ...
for (int i = ...){
  punti[i].draw();
  cerchi[i].draw();
  ...
}
```


Polimorfismo (2/2)

- È più comodo parlare alla classe base!
- Si può fare!

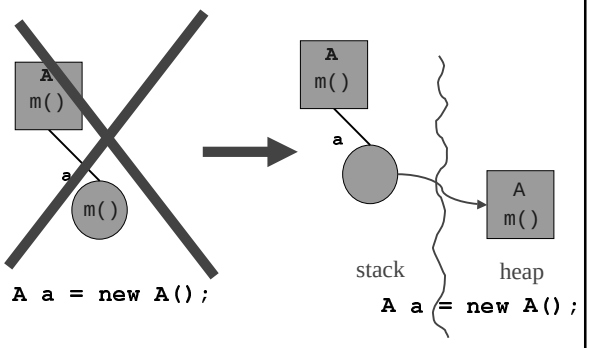



```
Figura[] figure = new Figura[100];
figure[i] = new Punto();
figure[j] = new Cerchio();
for (int k = ...)
  figure[k].draw();
```

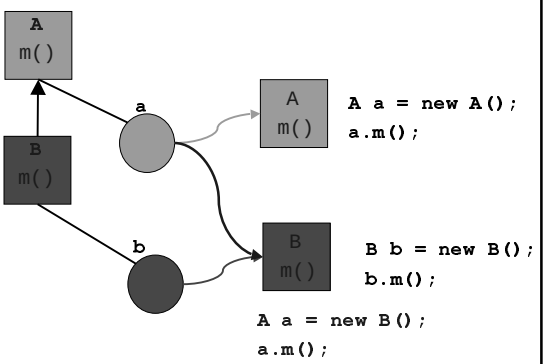

Perché il polimorfismo funziona

- N.B. Polimorfismo va combinato con (dipende da, è basato su):
  - eredità (**`Figura`** è sopraclasse) e
  - sovrascrittura (**`draw()`** è sovrascritto)
- Maniglie
  - Le variabili non contengono gli oggetti
  - Le variabili contengono il riferimento (la maniglia) agli oggetti

Le maniglie



Maniglie ed eredità

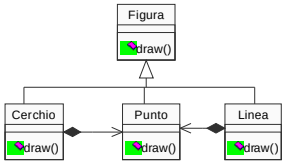


Terminologia

- Late binding, run-time binding
  - (durante l'esecuzione)
- Non early binding, compile-time binding
  - (durante la compilazione)
- Dynamic method lookup
- Sovrascrittura, sovraccarico
- Eckel:
  "If it isn't late binding, it isn't poliymorphism"

Il codice Java

- 4 + 1 classi
  - `Punto.java`
  - `Cerchio.java`
  - `Figura.java`
  - `(Linea.java)`
  - `UsaFigura.java`



Punto.java, Cerchio.java

```
class Punto extends Figura {
  //tutto come prima
  public void draw() {
    ...
  }
}

class Cerchio extends Figura {
  //tutto come prima
  public void draw() {
    ...
  }
}
```


Figura.java

- Una figura generica non sa disegnarsi…

```
class Figura {
  public void draw() {}
}
```


UsaFigura.java

```
class UsaFigura {
  public static void main (String[] args) {
    Figura[] figure = new Figure[4];
    figure[0] = new Punto();
    figure[1] = new Cerchio();
    figure[2] = new Punto();
    figure[3] = new Cerchio();
    for (int i = ...)
      figure[i].draw(); // Il draw() "in basso"
  }
}
```


Esercizio: cosa visualizza?

```
class A {
  protected void m(){
    System.out.println("A");
  }
}
class B extends A {
  protected void m(){
    System.out.println("B");
  }
}
class P {
  public static void main (String[] args) {
    A a = new A();
    B b = new B();
    a.m();
    b.m();
    a = b;
    a.m();
  }
}
```

```
>java P
A
B
B
```

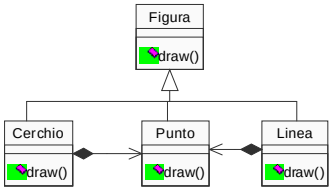

Riassunto: cos'è la OOP?

- TDA
  - Incapsulamento, interfaccia, implementazione
  - Composizione, uso
- Scambio messaggi
  - Oggetti attivi che si "parlano"
  - Metodi/attributi d'istanza e di classe
- Ereditarietà
  - Estensione, sovrascrittura, sottotipo, …
- Polimorfismo
  - Maniglie, late binding, "talk to the base class"

Scaletta

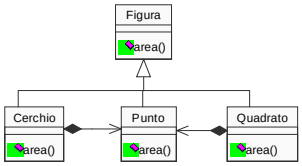
- Polimorfismo (cenni)
- OO in Java:
  - Classi astratte
  - Interfacce
  - Classi interne (cenni)

Figure...



`Figura` con `area()`?

- Aggiungiamo un metodo `area()`
  - Però: cerchi, punti e quadrati sanno calcolare la propria area
  - E una figura generica?
  - Uhm...



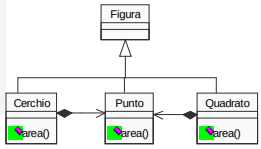

```
Cerchio c = new Cerchio();
Figura f = new Cerchio();
System.out.println(c.area());
System.out.println(f.area());
```


`Figura` senza `area()`?

- Una figura generica non sa calcolare la propria area ⇒ niente `area()` in `Figura`!
- Ma se non metto `area()` in `Figura`...

```
Figura c = new Cerchio();
Figura f = new Figura();
System.out.println(c.area());
System.out.println(f.area());
System.out.println(
          (Figura)c.area());
```

```
cannot resolve symbol
symbol  : method area  ()
location: class Figura
    f.area();
      ^
```

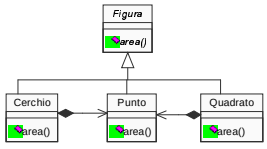


Problema: `Figura` e `area()`

- Se metto `area()` in `Figura`
  - Non so cosa deve fare
- Se non metto `area()` in `Figura`
  - No polimorfismo, perché non posso essere sicuro che le istanze di `Figura` (o sottoclasse) hanno `area()`
- Soluzione: classi e metodi astratti

`Figura` e `area()` astratti

- Soluzione:
  - `area()` in `Figura` è astratto (non invocabile)
  - anche `Figura` è astratta (non istanziabile)
- Astratto = non specificato

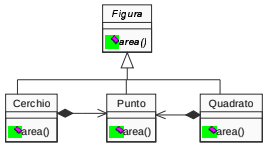


In Java

- Se dichiaro

```
abstract class Figura{
  abstract double area();
}
```

poi posso scrivere




```
Cerchio c = new Cerchio();
Figura f = new Cerchio();
System.out.println(c.area());
System.out.println(f.area());
```

```
f = new Figura();
System.out.println(f.area());
System.out.println((Figura)c.area());
```


Classi e metodi astratti

- Una classe astratta non può essere istanziata
- Metodo astratto: lasciato non specificato
- Se una classe contiene metodi astratti ⇒ deve essere astratta (cosa succederebbe invocando un metodo astratto?)
- N.B. Non è detto che una classe astratta contenga metodi astratti
  - Se una classe è astratta ~~≠~~ deve contenere metodi astratti

Utilità delle classi astratte

- Scomposizione problema: alcune classi della mia gerarchia sono lasciate non specificate (e verranno poi specializzate)
- Impedisce la creazione di istanze di quella classe
- Permette il polimorfismo

Postille

- Metodi **`static`** in classe **`abstract`**: OK (e si possono usare)
- Metodo **`abstract static`**: NO
- Come si può impedire la creazione di istanze di una classe?
  - Con costruttore privato e classe **`final`**

Scaletta

- Polimorfismo (cenni)
- OO in Java:
  - Classi astratte
  - Interfacce
  - Classi interne (cenni)

Interfacce

- "Classi"
  - completamente astratte (non specificate)
  - senza attributi
  - solo metodi d'istanza astratti (e costanti)

  `interface I {...}`
- Una classe **`A`** può implementare **`I`**:

  `class A implements I {...}`
- **`A`** definisce tutti i metodi di **`I`**

Figure scalabili (1/3)

```
interface Scalabile {
  public void riduci (int scala);
}

abstract class Figura {
   ...//tutto come prima...
}

class Punto extends Figura {
   ...//tutto come prima...
}
```

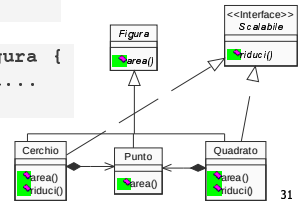


Figure scalabili (2/3)

```
class Cerchio extends Figura implements Scalabile {
  ...//tutto come prima...
  public void riduci (int scala) {
    raggio = raggio * scala / 100;
  }
}

class Quadrato extends Figura
        implements Scalabile {
  ...//tutto come prima...
  public void riduci (int scala)
  {
    ...
  }
}
```

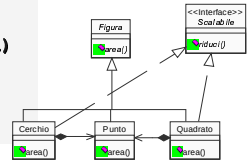
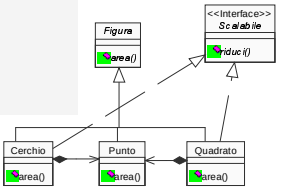


Figure scalabili (3/3)

- Ora posso scrivere

```
Scalabile s1 = new Quadrato();
Scalabile s2 = new Cerchio();
s1.riduci(50);
Scalabile[] s = new Scalabile[10];
...
s[i] = new Cerchio();
s[j] = new Quadrato();
...
s[i].riduci(50);
s[j].riduci(20);
```



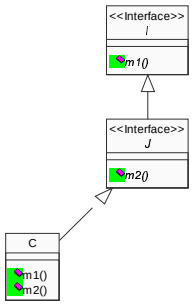
Interfacce

- Oltre alle classi: interfacce
- Classi che implementano interfacce devono implementarne i metodi:
  - se `C implements I` siamo sicuri (anche il compilatore!) che gli oggetti di `C` hanno i metodi di `I`
  - (oppure `C` è astratta)

```
abstract class NonSoComeRidurmi implements Scalabile{
  // senza riduci()
}
```

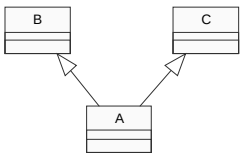

Interfacce ed eredità

- Un'interfaccia `J` può ereditare da un'interfaccia `I`
- Una classe (non astratta) `C` che implementa `J` deve implementare i metodi anche di `I`



Eredità singola e multipla

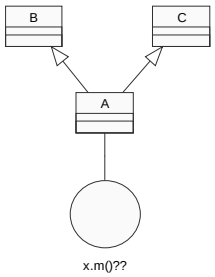
- Eredità singola
  - Ogni classe è sottoclasse diretta di un'unica superclasse
- Eredità multipla
  - Le classi possono avere più superclassi dirette



Eredità multipla: pro…

- Gerarchie nel mondo reale: non sempre a eredità singola
- **`Cerchio`** sottoclasse di
  - **`Figura`**
  - **`Scalabile`** (oggetti che possono ridursi)
- **`Cerchio`** eredita i metodi sia di **`Figura`** sia di **`Scalabile`**

…e contro



- Da chi viene ereditato **`m()`**?
- Quale **`m()`** va eseguito alla chiamata **`x.m()`**?

Eredità multipla in Java?

- In Java: eredità singola
- In C++: eredità multipla
- Però in Java l'eredità multipla "rientra dalla finestra"

```
class A extends B, C {
  ...
}
```


Eredità multipla con interfacce

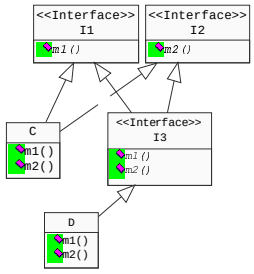
```
interface I1 {...}

interface I2 {...}

class C implements I1, I2 {
  ...
}

interface I3 extends I1, I2 {...}

class D implements I3 {...}
```



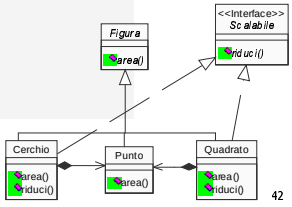
Eredità multipla con interfacce

- Quindi:
  - Un'interfaccia può estendere più interfacce
  - Una classe può implementare più interfacce
- (interfacce ≠ classi astratte)
- Perché?
  - No eredità multipla dell'implementazione

Utilità delle interfacce (1/3)

- Un'altra possibilità per il polimorfismo
- Es.: array di oggetti scalabili

```
Scalabile[] s = new Scalabile[10];
s[i] = new Cerchio();
s[j] = new Quadrato();
...
s[i].riduci(50);
s[j].riduci(20);
```



Utilità delle interfacce (2/3)

- Definizione costanti
- Oltre a metodi d'istanza astratti, anche attributi!
- Però **`static`** e **`final`** (ossia: costanti)
- Più comodo di **`CostantiUtili.PI`**

```
interface CostantiUtili{
  public static final double PI = 3.14159;
  public static final double E = 2.71828;
}
```

```
class C implements CostantiUtili{
  ... PI ... E ...
}
```


Utilità delle interfacce (3/3)

- Marcatori di oggetti, per sapere se un oggetto è istanza di una classe che implementa un'interfaccia
- Ad esempio la clonabilita è gestita con

```
interface Cloneable { }
```

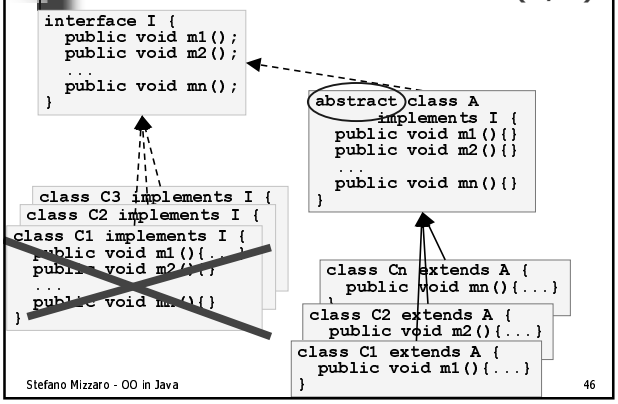
- Per vericare se una classe è clonabile:

```
if (x instanceof Cloneable) {
  ... operazioni di duplicazione ...
} else System.out.println("Non clonabile");
```


Utilità classi astratte - bis (1/2)

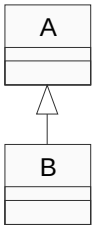
- "Pattern" comune:
  - Classi astratte che forniscono implementazioni parziali o banali di un'interfaccia
- Esempio:
  - Classe C che deve implementare un'interfaccia **`I`** con tanti metodi…
  - … ma è interessata solo a uno di questi (gli altri vuoti)
  - Se c'è una classe "adattatore"…

Utilità classi astratte - bis (2/2)



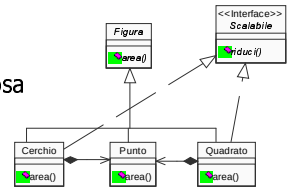
Conversioni di tipo e cast

- Promozione (implicita): da **`B`** ad **`A`**
- Cast (esplicito): da **`A`** a **`B`**



Cast rivisitato: upcast

- Upcast/Promozione:
  - Di solito implicita
  - Ok perché non pericolosa




```
Figura[] f = new Figura[…];
f[i] = new Punto();
```

```
f[i] = (Figura)new Punto();
```

Cast rivisitato: downcast

- Esplicito e pericoloso
- Se so che **`f[i]`** è un **`Punto`**, posso chiamare i suoi metodi, previo cast
- A volte downcast non fino "in fondo" alla gerarchia (array di **`Object`**, contenente una **`Figura`**…)

```
Figura[] f = new Figura[…];
f[i] = new Punto();
f[j] = new Cerchio();
f[i].setX(2.0);
f[j].setX(2.0);
```

```
(Punto)f[i].setX(2.0);
(Punto)f[j].setX(2.0);
```


Scaletta

- Polimorfismo (cenni)
- OO in Java:
  - Classi astratte
  - Interfacce
  - Classi interne (cenni)

Classi interne

- Classi dentro a un'altra classe
- Classi membro
- Classi locali (a un metodo)
  - Classi anonime

Esempio

```
class Classe {
  class Membro extends A {
    ...
  }
  public void m() {
    class Locale extends A {
      ...
    }
    Membro m = new Membro();
    Locale l = new Locale();
    A x = (new A(){
      ...
    });
  }
}
```

- Anche se **`A`** è un'interfaccia! (**`extends`**→**`implements`**)

Riassunto

- Polimorfismo (cenni)
- OO in Java:
  - Classi astratte
  - Interfacce
  - Classi interne (cenni)