OO in Java: 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 20
26 gennaio 2006

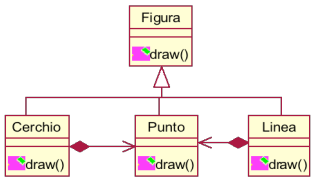
Riassunto: cos'è la OOP?

- TDA
  - Incapsulamento, interfaccia, implementazione
  - Composizione, uso
- Scambio messaggi
  - Oggetti (istanze) attivi che si "parlano"
  - Metodi/attributi d'istanza e di classe
  - Consente (ereditarietà e) polimorfismo
- Ereditarietà
  - Estensione, sovrascrittura, sottotipo, **`extends`**, **`super`**, relazioni fra classi…
- Polimorfismo
  - Maniglie, late binding, "talk to the base class"

Scaletta

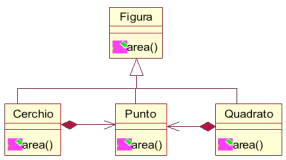
- OO in Java:
  - Classi astratte
  - Interfacce
  - Classi interne (cenni)

Figure…



`Figura` con `area()`?

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

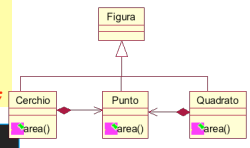



```
Cerchio c = new Cerchio();
Figura f = new Cerchio();
System.out.println(c.area());
System.out.println(f.area());
```


`Figura` senza `area()`?

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

```
Cerchio c = new Cerchio();
Figura c1 = new Cerchio();
Figura f = new Figura();
System.out.println(c.area());
System.out.println(f.area());
System.out.println(c1.area());
```



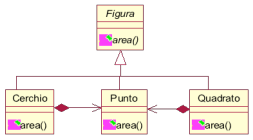

```
cannot resolve symbol
symbol  : method area  ()
location: class Figura
    f.area();
      ^
```

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



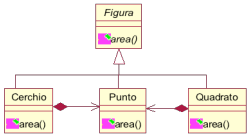
In Java

- 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

- 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...
}
```

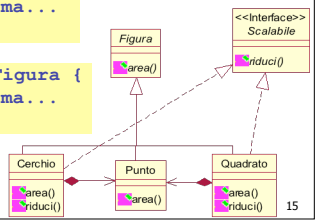


Figure scalabili (2/3)

```
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){
    ...
  }
}
```

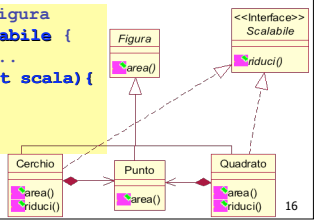
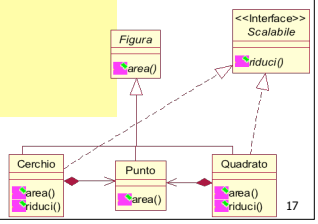


Figure scalabili (3/3)

- 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);
```



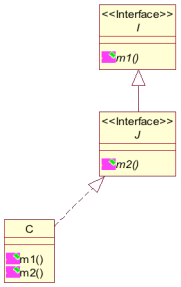
Interfacce

- 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()
}
```

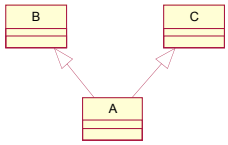
Interfacce ed eredità

- 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

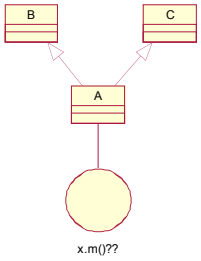
- Eredità singola
  - Ogni classe è sottoclasse diretta di un'unica superclasse
- Eredità multipla
  - Le classi possono avere più superclassi dirette



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()`**?

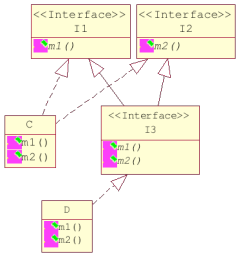
Eredità multipla in Java?

- 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 {...}
```



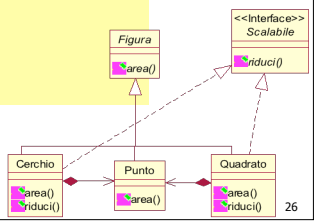
Eredità multipla con interfacce

- 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)

```
interface I {
  public void
  m1();
  public void m2();
  ...
  public void mn();
}
```

```
abstract class A
      implements I {
  public void m1(){}
  public void m2(){}
  ...
  public void mn(){}
}
```

```
class C3 implements I {
class C2 implements I {
class C1 implements I {
  public void m1(){...}
  public void m2(){}
  ...
  public void mn(){}
}
```

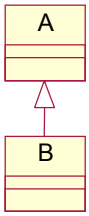
```
class Cn extends A {
  public void mn(){...}
}
```

```
class C2 extends A {
  public void m2(){...}
```

```
class C1 extends A {
  public void m1(){...}
}
```

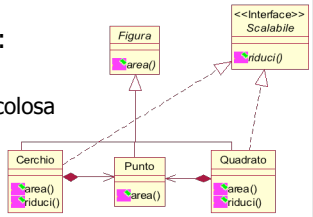
Conversioni di tipo e cast

- Con i tipi predefiniti:
  - Promozione (implicita): da `byte` ad `int`
  - Cast (esplicito): da `int` a `byte`
- Con le classi
  - Promozione (implicita): da `B` ad `A`
  - Cast (esplicito): da `A` a `B`
- Implicito: non crea problemi



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
  - Ma se poi `f[i]` non è un `Punto`, errore in esecuzione
- A volte downcast non fino "in fondo" alla gerarchia (array di `Object`, contenente una `Figura`…)
- Ex.: completare e sperimentare

```
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

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

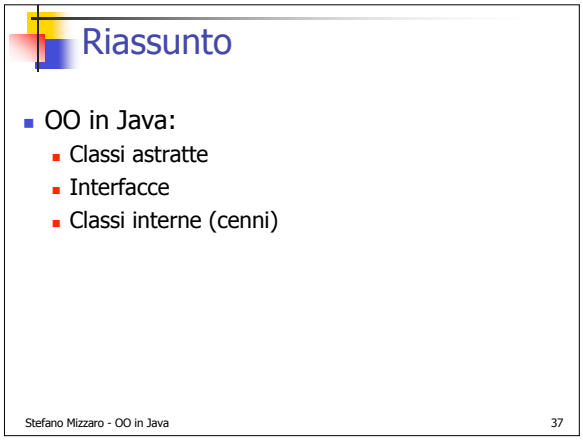
Classi interne

- Classi dentro a un'altra classe
- Classi membro
- Classi locali (a un metodo)
  - Classi anonime
- Le uniche differenze sono sulla visibilità
- Consiglio: per ora non usatele…

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
OO in Java:
Classi astratte
Interfacce
Classi interne (cenni)
Stefano Mizzaro - OO in Java
37