La programmazione OO

Stefano Mizzaro

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

Programmazione, lezione 19
25 gennaio 2006

Prossime lezioni

- Aula (8 lez.)
  - 18/1, 19/1,
  - 25/1, 26/1, 1/2, 8/2, 15/2, 22/2
- Laboratorio (9 lez.)
  - 16/1, 23/1,
  - 25/1, 30/1, 1/2, 6/2, 13/2, 20/2, 27/2
- (salvo contrattempi/variazioni)
- Ricevimento: Martedì 10:00 – 12:00

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.!!
- Scelta obbligatoria fra 1o scritto e 2o compitino
  - FATE ATTENZIONE QUANDO VI ISCRIVETE SU SINDY!!!!!

Riassunto

- "Mattoni"
- Sequenza, Selezione, Iterazione
- Array
- Sottoprogrammi (metodi)
- Ricorsione
- TDA
- TDA -> OO (Scambio messaggi)

Oggi

- Puntualizzazioni
  - Il `this` e il `toString` rivisitati
  - Funzionale, procedurale, OO
- Ereditarietà
- Polimorfismo

Funzionale, procedurale, OO (1/2)

- Funzionale:

```
public static Pila push (Pila p, int e) {
  if (!piena(p))
    p.elementi[p.numElementi++] = e;
  return p;
}
...
pi = Pila.push(p1,5);
```

- Procedurale:

```
public static void push (PilaP p, int e){
  if (!piena(p))
    p.elementi[p.numElementi++] = e;
}
...
Pila.push(p1,5);
```

- OO:

```
public void push (int e) {
  if (!piena())
    elementi[numElementi++] = e;
}
...
pi.push(5);
```

Funzionale, procedurale, OO (2/2)

- Programma funzionale in esecuzione =
  - funzioni che si chiamano a vicenda passandosi come parametri gli argomenti e restituiscono valori al chiamante
- Programma procedurale in esecuzione =
  - procedure che si chiamano a vicenda passandosi come parametri sia gli argomenti sia le variabili in cui memorizzare i risultati
- Programma OO in esecuzione =
  - oggetti che si scambiano messaggi, eventualmente passandosi come parametri altri oggetti

Di classe e d'istanza

	Attributi	Metodi
Di classe (`static`)	Informazioni sulla classe. Costanti	Approccio TDA
D'istanza	Informazioni sulle istanze	Approccio OO

Esercizi

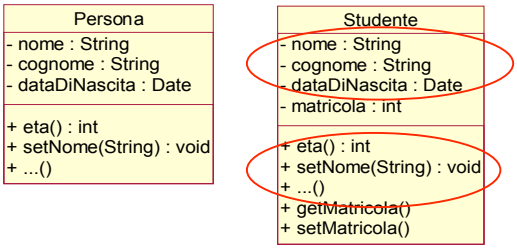
- **`Punto`, `Cerchio`** e **`Quadrato`** con approccio OO
  - Usando **`this`** e **`toString`**
  - POI confrontare con il codice sul testo
- **`Insieme`** con approccio OO
  - Usando **`this`** e **`toString`**
  - POI confrontare con il codice sul testo
- (fine cap. 8)

Scaletta

- Puntualizzazioni
  - Il **`this`** e il **`toString`** rivisitati
  - Funzionale, procedurale, OO
- Ereditarietà
- Polimorfismo

Terzo ingrediente: ereditarietà

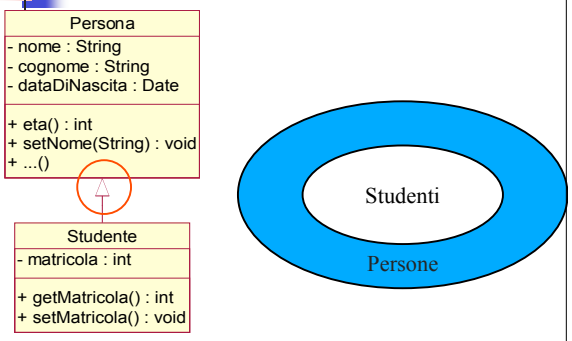
- La classe **`Persona`**...
- ... e la classe **`Studente`**



Un problema

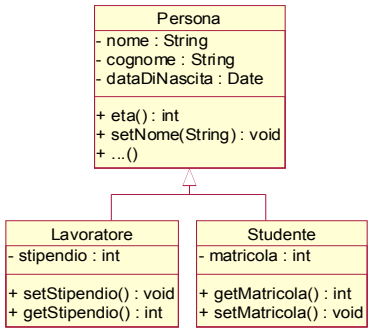
- Duplicazione di codice
  - Fatica inutile
  - Pericolo di incoerenze
- Soluzione: ereditare
  - **`Studente`** eredita da **`Persona`** attributi e metodi (e ne aggiunge, sovrascrive, ...)

Persona e Studente



Stefano Mizzaro - OO 13

... e Lavoratore



Stefano Mizzaro - OO 14

Ereditarietà in Java

- Parola riservata **`extends`**
- **`class Studente extends Persona {`**
  - ... e **`Studente`** ha metodi e attributi di **`Persona`**
- Sovrascrittura: nella sottoclasse posso specializzare il comportamento
  - N.B. Sovrascrittura (*overriding*) ≠ sovraccarico (*overloading*)
    - Sovraccarico: nome =, firme ≠
    - Sovrascrittura: nome =, firme =
- N.B. Ereditarietà troppo enfatizzata...

Stefano Mizzaro - OO 15

La classe `Persona`

```
class Persona {
  private String nome;
  private String cognome;
  private Data nascita;

  public void setNome(String nome) { ... }
  public String getNome() { ... }
  public void setNascita( ... ) { ... }
  public int eta() { ... }
  public String toString() { ... }
  ...
}
```

- ... una classe come un'altra...

Stefano Mizzaro - OO 16

La classe `Studente`

```
class Studente extends Persona {
  private int matricola;

  public int getMatricola() { ... }
  public void setMatricola( ... ) { ... }
  ...
}
```

- Oltre a quello che c'è in **`Studente`**, anche quello che c'è in **`Persona`**
- ... ma ciò che è privato in **`Persona`** non è visibile in **`Studente`**

Stefano Mizzaro - OO 17

Uso

```
...
Persona p1 = new Persona( ... );
Studente s2 = new Studente( ... );
...
p1.setNome("Gino");
s2.setMatricola(1234);
s2.setNascita( ... );
...
System.out.println(p1.getNome());
System.out.println(s2.eta());
...
```

- Uno studente è una persona (!)

Stefano Mizzaro - OO 18

Quindi, l'ereditarietà

- Consente a una classe di ereditare da altre classi, evitando duplicazione di codice
- Consente di vedere un'istanza di una sottoclasse "come se" fosse un'istanza della sopraclasse
  - Vedo uno studente come se fosse una persona (!)
- **`extends`** in Java

Ereditarietà e `private`

- Si eredita tutto (anche attributi e metodi **`private`**) ma non tutto è visibile (solo ciò che non è **`private`**)

```
class A {
  private int x;
  public int getX(){
    return x;
  }
}
```

```
class B extends A {
  public void m(){
    System.out.println(x);
  }
}
```

```
class B extends A {
  public void m() {
    System.out.println(getX());
  }
}
```


La sovrascrittura (overriding)

- Un metodo di una sottoclasse può sovrascrivere un metodo con la stessa firma in una sopraclasse
- Es.: visualizziamo anche la matricola di uno studente (e quindi il **`toString`** va ridefinito/sovrascritto nella sottoclasse)

```
class Studente extends Persona {
  private int matricola;
  ...
  public String toString() {
    return ...//COPIO il toString di Persone
           + matricola; // e aggiungo la matricola
  }
}
```

- **`System.out.println(p2)`**, con **`p2`** istanza di **`Studente`**, visualizza anche la matricola

Sovrascrittura ≠ Sovraccarico

- Sovrascrittura (firme =) ≠ sovraccarico (firme ≠)

```
class A {
  public void m1() { ... }
  public void m2() { ... }
  public void m2(.) { ... }
  public void m3() { ... }
}

class B extends A {
  public void m1() { ... }
  public void m3(.) { ... }
}
```


Il `super` (1/2)

- Duplicazione di codice…@#$}[%&>~?
- Con **`super`** si fa riferimento alla sopraclasse
- Esempio:

```
class Studente extends Persona {
  private int matricola;
  ...
  public String toString() {
    return
       super.toString() // chiamo toString di Persone
        + matricola;    // E aggiungo la matricola
  }
}
```


Il `super` (2/2)

- Il **`super`** nel costruttore

```
public B() {
  super();
  ...;
}
```

- Il costruttore di default:

```
public B() { super(); }
```

- **`super`** (riferimento alla sopraclasse) simile al **`this`** (riferimento alla classe)

Relazioni fra classi (1/2)

- **È-un** (is-a)
  - Relazione fra sopraclassi e sottoclassi
  - Il trapezio è un quadrilatero, che è un poligono, che è una figura geometrica bidimensionale, che è una figura geometrica; l'automobile è un veicolo, ecc. ecc.
  - È la relazione che corrisponde all'eredità
- **Parte-di** (part-of, o has-a, o ha-un)
  - Relazione fra un oggetto e le sue componenti
  - Il punto è una parte del cerchio (è il suo centro); il pistone è parte del motore, che è parte dell'automobile
  - Relazione che corrisponde all'operazione di composizione.
- **Usa**
  - Relazione fra le classi che implementano TDA e le classi che li usano: classi di prova.

Relazioni fra classi (2/2)

- [**Istanza-di** (instance-of, membro-di, member-of)]
  - Relazione fra una classe e i suoi elementi (istanze, oggetti, esemplari), o fra un insieme e i suoi elementi:
  - Il cerchio `c1` è un'istanza della classe `Cerchio`; la mia automobile è un'istanza della classe delle automobili; ecc.
- Oppure (?):
  - `Cerchio` è-un `Punto` (cerchio = punto + raggio)
  - `Studente` parte-di `Persona` (in `toString()` di `Studente` invoco `toString()` di `Persona` per visualizzare la "parte dello studente che è una persona")…
  - Buon senso, "regola del sottoinsieme", …

Scaletta

- Puntualizzazioni
  - Il `this` e il `toString` rivisitati
  - Funzionale, procedurale, OO
- Ereditarietà
- Polimorfismo

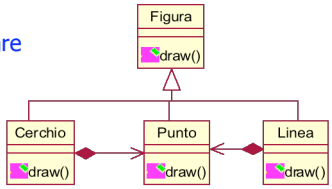
Polimorfismo (1/2)

- 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 if ...
for (int i = ...){
  punti[i].draw();
  cerchi[i].draw();
  ...
}
```


Polimorfismo (2/2)

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

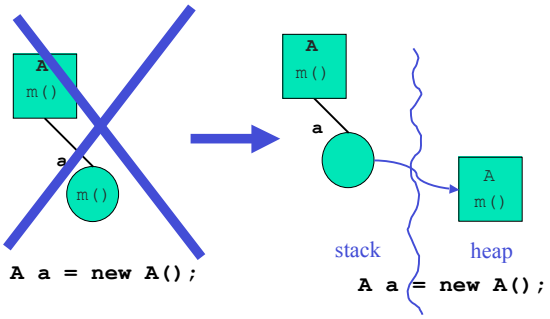



```
Figura[] figure = new Figura[100];
figure[i] = new Punto();
figure[j] = new Cerchio();
for (int k = ...)
  figure[k].draw();
```

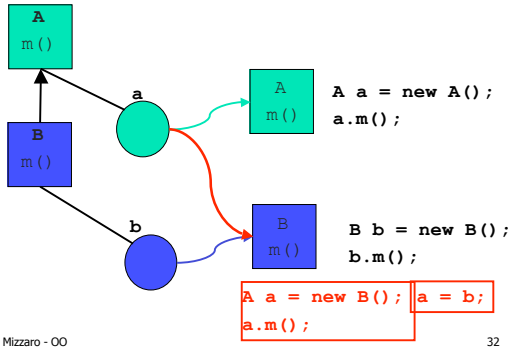

Perché il polimorfismo funziona

- 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

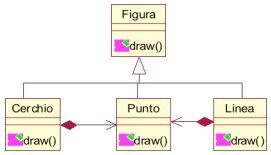


Terminologia

- 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 {
  public void m(){
    System.out.println("A");
  }
}
class B extends A {
  public 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
```


Riassumendo: 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, **`extends`**, **`super`**, relazioni fra classi…
- Polimorfismo
  - Maniglie, late binding, "talk to the base class"

Riassunto

- TDA
- Scambio messaggi
- Ereditarietà (cennI)
- Polimorfismo (cenni)
- Prossima lezione
  - OO in Java