Programmazione a Oggetti

Ereditarieta'

Sommario

Come definire sottoclassi

Costruttori

Abstract Classes

Final

Ereditarietà: promemoria

Strumento tipico dell'OOP per riusare il codice e creare una gerarchia di astrazioni

1. ***Generalizzazione****: Una superclasse generalizza una sottoclasse fornendo un comportamento condiviso dalle sottoclassi*
2. ***Specializzazione:*** *Una sottoclasse specializza (concretizza) il comportamento di una superclasse*

Permette di rappresentare relazioni di tipo IS-A (o IS-KIND-OF) oppure ***IS-LIKE*** *(****specializzazione procedurale****)*

Ereditarietà: a cosa serve?

In un certo senso, ***a niente****!*

Replicando opportunamente il codice…
(attenzione al method lookup!)

Localizza *la scrittura di alcune procedure rendendole comune a tutte le classi;*

Permette di specializzare il comportamento di una classe, prevedendo nuove funzionalità, mantendendo le vecchie e quindi ***senza influenzare codice cliente già scritto****.*

Ereditarietà in Java

Una sottoclasse si definisce usando la parola chiave **`extends`**:

```
class Impiegato extends Persona
    {. . .}
```

Osservate che l'ereditarietà in Java è ***semplice***.

Una classe può implementare più interfacce, ma estendere ***solo*** *una superclasse*.

Perché?

Ereditarietà in Java

```
class Persona {
  private String nome;
  private int eta;
  Persona(String s, int e)
     {nome=s; eta=e;}
  public int eta() {return eta;}
}

class Impiegato extends Persona{
  private static int salario;
  private int ufficio;
  Impiegato(??){??}
  public static void aumento(int a)
     {salario = salario + a;}
```

Ereditarietà in Java

La classe `Impiegato` *eredita* ***implicitamente*** *i campi definiti dalla classe* `Persona`.

Osservate che nella classe `Impiegato` ***non sono visibili*** *le variabili e metodi dichiarati private. Quindi fanno parte dello stato degli oggetti istanziati, ma non sono riferibili (direttamente) dai nuovi metodi.*

Importante: *Il tipo* `Impiegato` *è* ***sottotipo*** *del tipo* `Persona`.

E il costruttore?

Tipicamente deve accedere ***alle variabili di istanza****, anche della* ***superclasse****.*

Soluzioni*:*

definire i ***metodi di accesso*** *(getters e setters)*

definire le variabili di istanza sempre **`protected`**

Insoddisfacenti*:*

rilassano l'incapsulazione (l'implementatore della sottoclasse potrebbe non rispettare proprietà desiderate nella superclasse)

non favoriscono il riuso del codice

Si deve poter riferire il costruttore della superclasse

Costruttore

```
class Impiegato extends Persona
{private static int salario;
 private int ufficio;
 Impiegato(
 public static void aumento(int a)
 {salario = salario + a;}
 Impiegato(String n, int e, int u)
 { super(n,e);
    ufficio = u;
 }
}
```

Costruttore "super"

Attenzione! *Se non c'è nella sottoclasse una chiamata esplicita al costruttore della superclasse, il complilatore* ***aggiunge automaticamente*** *il codice:*

```
super()
```

che invoca il default constructor o un costruttore senza parametri. Se non fosse presente, genererebbe un errore in compilazione.

Ordine di esecuzione dei costruttori:

prima i costruttori delle superclassi, poi eventuali inizializzatori, infine il costruttore.

Esercizio

Scrivere una gerarchia di 3 classi, **`Nonno`**, **`Padre`** *e* **`Nipote`** *che in output* ***evidenzi l'ordine di esecuzione dei costruttori****.*

(max 20 righe di codice…)

Il programma principale, sarà semplicemente:

```
public class testOrderConstruction
{ Nipote n = new Nipote();
  public void static main(String arg[]){}
}
```

Metodi

*La sottoclasse **eredita i metodi** dalla superclasse.*

*La sottoclasse può ridefinire (**override**) un metodo (stesso tipo e livello di **visibilità maggiore**).*

*Il **dynamic dispatch** garantirà che venga eseguita la versione più specifica del metodo (sulla base del **tipo dinamico** del ricevente).*

Attenzione**: non confondere **overriding** e **overloading

Dynamic Method Lookup

***Attenzione**: Il lookup dei metodi **non è sempre dinamico**. Esistono due significative eccezioni:*

*- chiamate di **metodi privati** (in fondo non sono accessibili alle sottoclassi, quindi non si può fare overriding).*

*- chiamate attraverso **super***

*E' **sempre statico** anche il binding dei metodi di classe (beh, si chiamano static) e delle variabili (sia di istanza che di classe)*

Esercizio

Scrivere una gerarchia di 2 classi, **`Padre`** *e* **`Figlio`**
che in output ***evidenzi le regole di lookup*** *dei metodi sopra viste*

(max 15 righe di codice…)

Il programma principale, sarà semplicemente:

```
public class testMethodLookup
{ Figlio f = new Figlio();
  public void static main()
     {f.m();}
}
```

Method Lookup: esempio

```
class Punto
{ private int xcoord;

Punto(int x) {xcoord=x;}
public void move(int dx)
   {xcoord += dx;}
protected int modulo()
   {return Math.abs(xcoord);}
public void printXcoord()
   {System.out.println(""+xcoord);}
}
```

Method Lookup: esempio

```
class PuntoColorato extends Punto
{ private Color c;

PuntoColorato(int x, Color c)
   {super(x); this.c=c;}
public void move(int dx)
   {super.move(dx);
    if (modulo()>5)c.setGreen();}
public void printColor()
   {c.printColor();}
}
```

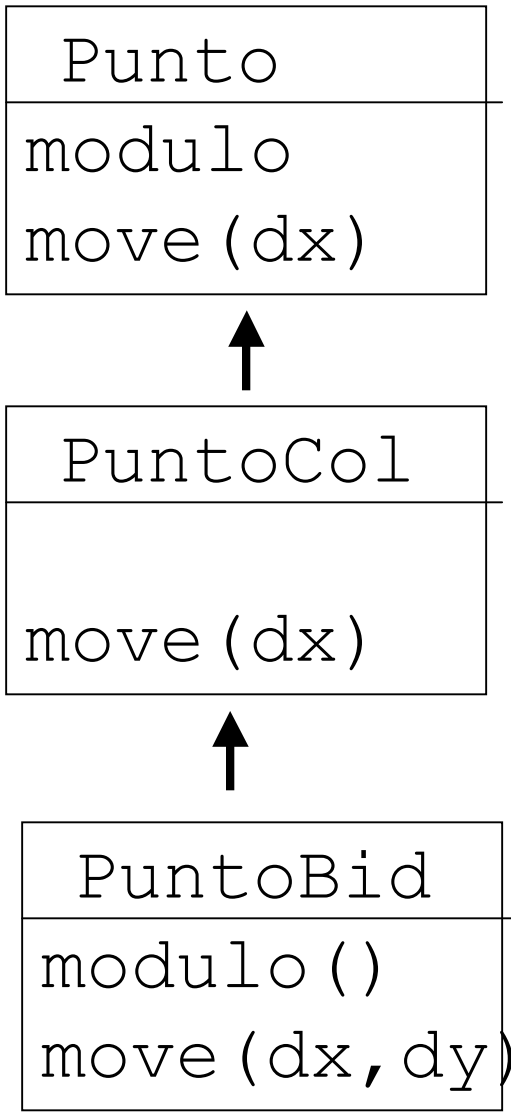
Method Lookup: esempio

```
class PuntoBid extends PuntoColorato
{ private int ycoord;
PuntoBid(int x, int y, Color c)
   {super(x,c); this.y=y;}
public void move(int dx, int dy)
   {ycoord+=dy; super.move(dx);}
protected int modulo(){
 return Math.max(Math abs(ycoord),
   super.modulo());}
public void printYcoord()
   {System.out.println(“”+yccord);}
}
```

Method Lookup: esempio

```
public class TestPunti
{ public static void main(String[] args)
  { PuntoBid p = new PuntoBid(3,3, new
    Color("red"));
     p.printColor();  /* stampa red */
     p.move(2,5);
     p.printXcoord(); /* stampa 5 */
     p.printYcoord(); /* stampa 8 */
     System.out.println(""+p.modulo());
     p.printColor();  /* stampa green */
  }
}
```

Method Lookup



Classi:contengono il codice dei metodi e variabili di classi

Si cerca il metodo sempre cominciando dal ricevente e salendo nella gerarchia delle class (yo-yo)

Oggetti:contengono le variabilidi istanza

Template Methods

Buona abitudine: *scomporre il comportamento di un metodo in sottotask (**write stupid methods**)*

```
public void doSomething()
      { doThis();
        doThat();
           }
```

Logica: *ritardare finchè possibile l'implementazione.*

Vantaggio: *si guadagna **flessibilità.** Magari per specializzare il metodo in una sottoclasse è sufficiente fare overriding di* `doThis()` *.*

Problema: `doThis()` *e* `doThat()` *non sono logicamente parte dell'interfaccia.*

Soluzione: *dichiararli protected!*

Final

Un metodo dichiarato `final` ***non può venire ridefinito*** *in una sottoclasse (ha senso per i* `public` *e* `protected`*)*

La parola chiave `final` *può essere usata anche per* ***campi e classi****:*

```
private final int pi = 3.14
```
Costanti

```
final class NoSub{…}
```
non si può estendere

Motivazioni:sicurezza

```
public boolean validatePassword(String s)
```

Attenzione*: se il metodo final contiene chiamate a metodi non final il suo comportamento può essere modificato dalle sottoclassi;*

Ottimizzazione*: i metodi final sono chiamati seguendo* ***static binding****.*

Classi Astratte

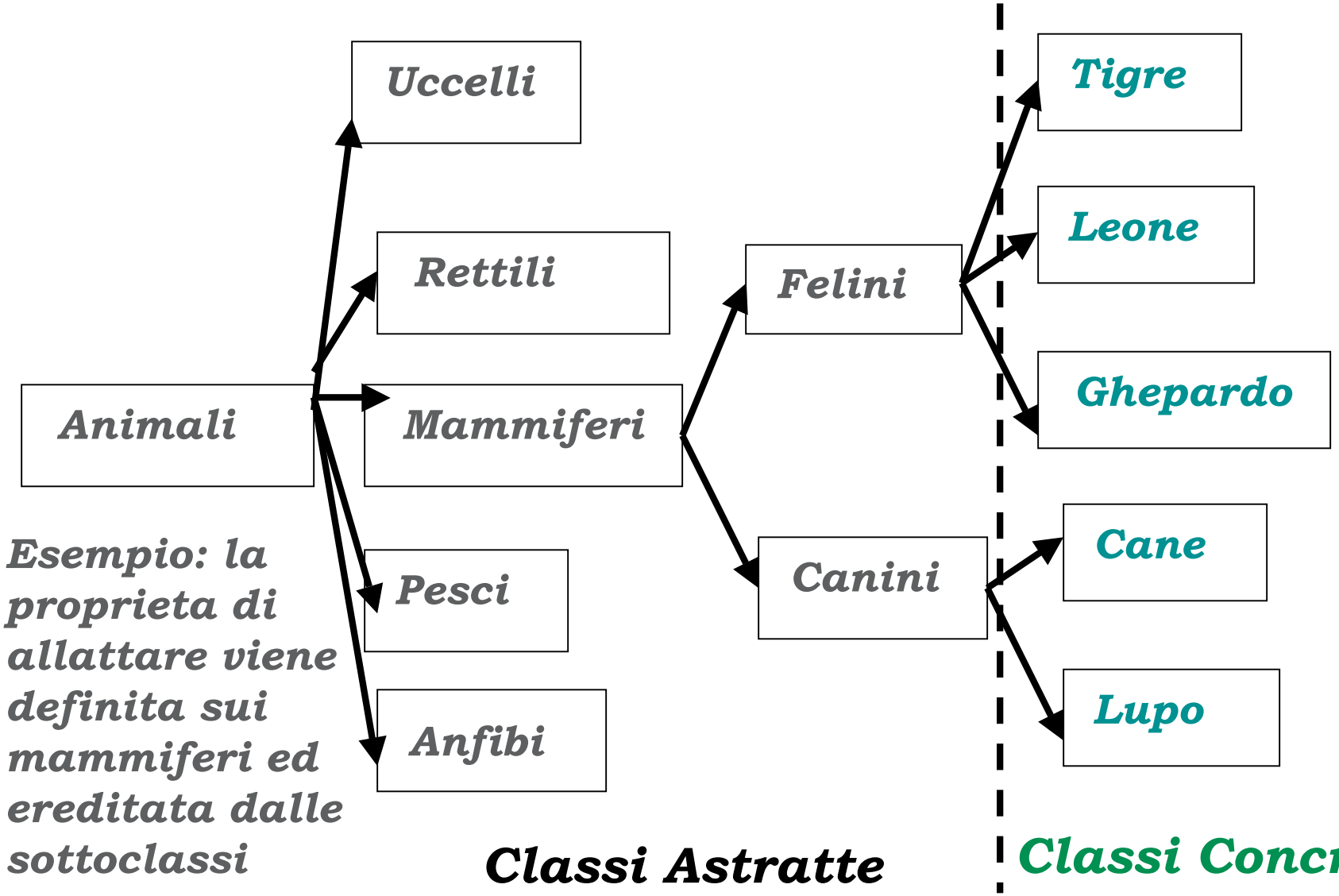
Non** sono intese per **generare istanze

***Raggruppano caratteristiche** comuni di classi concrete*

*Spesso **definiscono metodi non implementati** che verranno definiti dalle sottoclassi*

Non è possibile generare oggetti da classi astratte

Classi Concrete e Astratte



Esempio: la proprieta di allattare viene definita sui mammiferi ed ereditata dalle sottoclassi

Metodi e Classi Astratte

Un metodo dichiarato **`abstract`** *può non venire implementato.*

Una classe che contiene un metodo astratto ***deve essere*** *dichiarata anch'essa* **`abstract`**`.`

Da una classe astratta non si possono generare istanze.

Una sottoclasse può diventare concreta se implementa ***TUTTI*** *i metodi pubblici* **`abstract`** *della superclasse astratta. Altrimenti è anch'essa una classe astratta.*

Classi astratte vs Interfacce: riflettete

Classe Object

Tutte le classi ereditano implicitamente dalla root class **`Object`**.

Fornisce un insieme di metodi ereditati da TUTTE le classi. Ad esempio il metodo **`toString()`** .

I metodi definiti da **`Object`** *possono essere* ***ridefiniti***.

Esperimento: *cercate sul JAVA SUN TUTORIAL i servizi offerti dalla classe* **`Object`**.

Esempio: cella di memoria

Scriviamo una classe che implementa una cella di memoria:

```
class Cell{
{ private int val;
  public Cell(int v) {val=v;}
  public int getVal() {return val;}
  public void setVal(int v){val=v;}
  public void clear(){setVal(0);}
}
```

Esempio: cella con backup

Ogni volta che modifichiamo il valore memorizzato, salviamo il vecchio valore.

Definiamo una funzione **`restore()`** *che permette di ripristinare il valore precedente:*

```
class BackupCell extends Cell{
{ private int backup;
  public BackupCell(int v)
     {super(v); backup=0;}
  public void setVal(int v)
     {backup=getVal(); super.setVal();}
  public int restore()
     {super.setVal(backup); return
    backup;}
}
```

Esempio: `setVal()`

Chiama il metodo ereditato

```
public void setVal(int v)
{backup=getVal(); super.setVal();}
```

Invoca il metodo della superclasse.

E' overridden, ma visibile.

Non ci sono altri modi di settare `val`

E se invece...

dimenticassimo super?

```
public void setVal(int v)
{backup=getVal(); setVal();}
```

Che succede?

```
Cell c = new Cell(0);
c.setVal(1);
c = new BackupCell(1);
c.setVal(4);
```

restore()

e se dimenticassimo `super`?

```
public void setVal(int v)
{super.setVal(backup);
 return backup;}
```

Qui potremmo scrivere semplicemente

```
setVal(backup);
```

Cosa cambierebbe?