Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria

CORSO DI

FONDAMENTI DI INFORMATICA C
Linguaggio Java: Classi Astratte e Interfacce

Prof. Franco Zambonelli – Ing. Giacomo Cabri

Lucidi realizzati in collaborazione con
Ing. Enrico Denti - Univ. Bologna

Anno Accademico 2001/2002

CLASSI ASTRATTE

L'ereditarietà porta a riflettere sul rapporto fra *progetto* e *struttura*.
In particolare,

- **una classe può limitarsi a *definire l'interfaccia e le proprietà base di una classe di entità*...**
- **..lasciando "in bianco" alcune operazioni...**
- **...che verranno poi implementate dalle classi derivate.**

Una classe con queste caratteristiche:

- **fattorizza, *dichiarandole*, operazioni comuni a tutte le sue sottoclassi, e *definisce* dati comuni...**
- **... ma non *definisce* (implementa) le operazioni**

Quindi...

- **di una classe siffatta *è impossibile definire istanze*,** perché esistono metodi lasciati in bianco" (a cui queste istanze non saprebbero come rispondere)

–

È una CLASSE ASTRATTA

Una *classe astratta*:

- non viene creata **per derivarne *istanze***...
- ma **per derivarne *altre classi***, che dettaglieranno (specificandole) le operazioni qui solo dichiarate

ESEMPIO: IL REGNO ANIMALE

Tutti abbiamo un'idea di cosa sia un animale: in particolare

- ogni animale ha *un qualche verso*
- ogni animale si muove *in qualche modo*
- ogni animale vive *in un qualche ambiente* (aria, acqua, terraferma...)

...ma proprio per questo, ***non esiste "il generico animale"!***

Ogni animale reale:

- ha *uno* ***specifico*** *verso*
- si muove *in uno* ***specifico*** *modo*
- vive *in uno* ***specifico*** *ambiente*

Ad esempio:

- il leone *ruggisce*, si muove *avanzando a quattro zampe*, e vive *sulla terraferma*
- la rana *gracida*, si muove *saltando*, e vive *negli stagni*
- l'uomo *parla*, si muove *camminando*, e vive *sulla terra*

Allora, *perché introdurre una tale classe "animali"?*

- appunto **per *fattorizzare aspetti comuni***
- che inducono una ***classificazione*** **degli oggetti del mondo**

In effetti, non esistono neanche i *pesci*, i *mammiferi*, gli *uccelli*...
... ma disporre di queste categorie concettuali è comunque utile!

SPECIFICA DI CLASSI ASTRATTE

Come specificare una classe come "astratta" ?

In Java, basta etichettare la classe come **`abstract`**

ESEMPIO

```
public abstract class Animale {
  public abstract String verso();
  public abstract String si_muove();
  public abstract String vive();
}
```

Il concetto così espresso è che

- *ogni animale "reale" può fare un verso, può muoversi, e può dire in che ambiente vive*
- ma ***non si può, in generale, precisare come***.

NOTE

- una classe avente anche solo un metodo `abstract` è astratta, e deve essere dichiarata `abstract` essa stessa (altrimenti si ha errore)
- una classe astratta può però anche non avere metodi dichiarati `abstract` (ma resta comunque astratta, e quindi è impossibile istanziarla)
- una sottoclasse di una classe astratta è anch'essa astratta, se non ridefinisce *tutti* i metodi che erano astratti nella classe base.

ESEMPIO: TASSONOMIA ANIMALE

L'assunto alla base di questo esempio è che ogni animale "concreto" sia caratterizzato (compiutamente, almeno ai fini dell'esempio stesso) dalle due seguenti proprietà:

- l'ambiente in cui vive;
- il modo in cui si muove.

IPOTIZZIAMO CHE:

- **ogni animale sappia rispondere a un messaggio, `chi_sei()`,** che restituisce una breve descrizione dell'animale stesso;
- **tutti gli animali siano rappresentabili nello stesso modo** (nel senso che esiste almeno un denominatore comune alle loro rappresentazioni)
- **ogni animale sappia rispondere al messaggio `mostra()`,** la cui definizione è *indipendente* dallo specifico animale a cui si applica, perché basata sugli altri metodi.

Poiché *ogni* animale:

- *vive* in un certo ambiente (ma *quale* ambiente dipende dallo specifico animale considerato)
- *si muove* in un certo modo (ma *come* esattamente varia da caso a caso)

i due metodi `vive()` e `si_muove()` hanno tutte le caratteristiche dei metodi astratti
→ è opportuno che vengano dichiarati nella classe-base `animale` (perché tipici di ogni possibile animale reale)

COSTRUTTORI E CLASSI ASTRATTE

La classe-base atratta deve avere un costruttore?

Essendo astratta, forse non serve.... o invece sì ?

RIFLESSIONE

- le classi derivate che rappresentano animali concreti **avranno senz'altro un costruttore**, che *per prima cosa* invocherà un costruttore della classe-base
- **se esiste anche solo un campo dati *privato* nella classe-base, *per inizializzarlo servirà un costruttore della classe base***, visto che le classi derivate non ne hanno visibilità diretta

Naturalmente, **tale costruttore non sarà mai usato direttamente** (perché non esisteranno mai istanze di "generici" animali), ma sarà chiamato dai costruttori delle classi derivate

→ può essere `public` o anche solo `protected`

```
public abstract class Animale {
  private String nome;
  protected String verso;
  public Animale(String s) { nome=s; }
  public abstract String si_muove();
  public abstract String vive();
  public abstract String chi_sei();
  public void mostra() {
    System.out.println(nome + ", " + chi_sei() +
        ", " + verso + ", si muove " +
        si_muove() + " e vive " + vive() );  } }
```

Per procedere nella costruzione della tassonomia, occorre ora *decidere i criteri con cui operare la classificazione.*

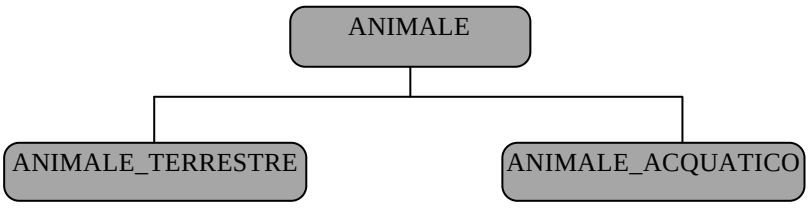
CRITERI DI CLASSIFICAZIONE

- in linea di principio, si possono pensare sottoclassi di animali caratterizzati da un certo colore della pelle, dal metodo di riproduzione, o da qualunque altro criterio
- ma *queste scelte non sono, ora, le più naturali*
- infatti, ***tipicamente i criteri di classificazione si basano sulle caratteristiche corrispondenti ai metodi astratti,*** **perché sono esattamente le proprietà ritenute <u>rilevanti</u> fin dalla progettazione della classe-base**

↓

Due "criteri naturali":
- l'ambiente in cui l'animale vive
- il modo di muoversi

Scegliendo, *per fare una prima ripartizione*, il criterio dell'ambiente di vita, una possibile scelta può essere distinguere fra



animali ***terrestri*** e *animali* ***acquatici***.

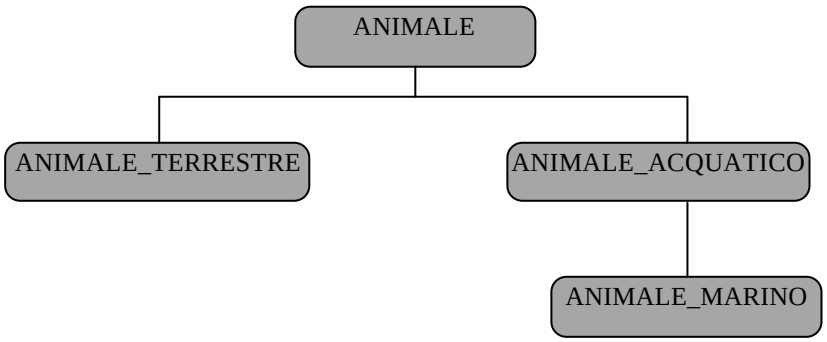
<u>Queste due nuove sottoclassi sono ancora classi astratte</u>:
- infatti, nulla è ancora stato detto sul movimento
  → impossibile definire il metodo `si_muove()`.

LE PRIME DUE SOTTOCLASSI (astratte)

```
public abstract class AnimaleTerrestre
           extends Animale {
  public AnimaleTerrestre(String s) { super(s); }
  public String vive() {
       return "sulla terraferma"; }
  public String chi_sei() {
       return "un animale terrestre"; }
}
```

```
public abstract class AnimaleAcquatico
       extends Animale {
  public AnimaleAcquatico(String s) { super(s); }
  public String vive() { return "nell'acqua"; }
  public String chi_sei() {
       return "un animale acquatico"; }
}
```

Un'ulteriore classificazione (meno scontata della precedente) può portare a introdurre la classe (ancora astratta) degli *animali marini*, come specializzazione degli animali acquatici.



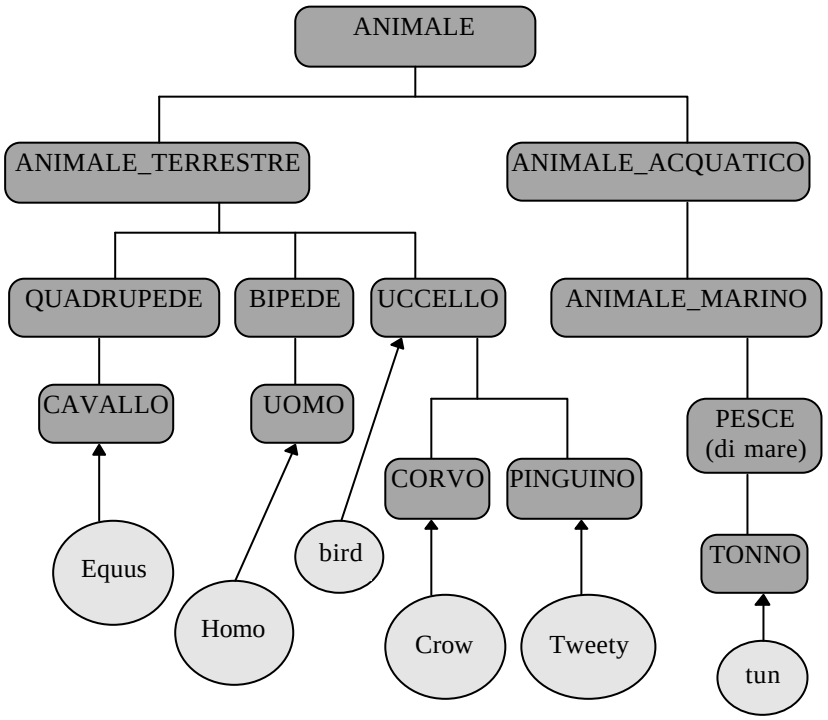
In questo caso il criterio usato non ha nulla a che fare con le caratteristiche inizialmente evidenziate (ambiente di vita, movimento e, in misura minore, verso), ma rispecchia semplicemente una suddivisione che esiste nella realtà.

La dichiarazione della nuova classe ha l'aspetto seguente:

```
public abstract class AnimaleMarino
       extends AnimaleAcquatico {
  public AnimaleMarino(String s) { super(s); }
  public String vive()  { return "in mare"; }
  public String chi_sei() {
       return "un animale marino"; }
}
```

LA TASSONOMIA COMPLETA

La tassonomia completa mette in gioco anche l'altro criterio "naturale", quello relativo al movimento: su questa base, gli animali terrestri vengono suddivisi in quadrupedi, bipedi e uccelli.



Ognuna di queste categorie dà la sua definizione per il metodo `si_muove()`, perciò *queste classi rappresentano classi di animali reali* (non più astratte), di cui si possono creare istanze.

LE CLASSI “CONCRETE”

```
public class PesceDiMare extends AnimaleMarino {
  public PesceDiMare(String s) { super(s);
        verso="non fa versi"; }
  public String si_muove() { return "nuotando"; }
  public String chi_sei() {
        return "un pesce (di mare)"; }
}
```

```
public class Uccello extends AnimaleTerrestre {
  public Uccello(String s) { super(s);
        verso="cinguetta"; }
  public String si_muove() { return "volando"; }
  public String chi_sei() { return "un uccello";}
  public String vive() {
        return "in un nido su un albero"; }
}
```

```
public class Bipede extends AnimaleTerrestre {
  public Bipede(String s) { super(s); }
  public String si_muove() {
         return "avanzando su 2 zampe"; }
  public String chi_sei() {
        return "un animale con due zampe"; }
}
```

```
public class Quadrupede extends AnimaleTerrestre
{
  public Quadrupede(String s) { super(s); }
  public String si_muove() {
        return "avanzando su 4 zampe"; }
  public String chi_sei() {
        return "un animale con quattro zampe"; }
}
```

ULTERIORI CLASSI PIÙ SPECIFICHE

- l'uomo si muove sì "avanzando su due zampe", ma più esattamente "cammina su due gambe" → ridefinisce `si_muove()`
- il pinguino, pur essendo un uccello (e in generale gli uccelli volano), non sa volare → ridefinisce `si_muove()`

```
public class Cavallo extends Quadrupede {
  public Cavallo(String s) { super(s);
        verso="nitrisce"; }
  public String chi_sei() { return "un cavallo";
}
}
```

```
public class Uomo extends Bipede {
  public Uomo(String s) { super(s);
        verso="parla"; }
  public String si_muove() {
        return "camminando su 2 gambe"; }
  public String chi_sei() {
        return "un homo sapiens"; }
  public String vive() { return "in condominio";
}
}
```

```
public class Corvo extends Uccello {
  public Corvo(String s) { super(s);
        verso="gracchia"; }
  public String chi_sei() { return "un corvo"; }
}
```

```
public class Pinguino extends Uccello {
  public Pinguino(String s) { super(s);
        verso = "non fa versi"; }
  public String chi_sei() { return "un
pinguino";}
  public String si_muove() {
        return "ma non sa volare"; }
}
```

ULTERIORI CLASSI PIÙ SPECIFICHE (segue)

```
public class Tonno extends PesceDiMare {
  public Tonno(String s) { super(s); }
  public String chi_sei() { return "un tonno"; }
}
```

L' ESEMPIO COMPLESSIVO

Un possibile main, che costruisce un "mondo di animali":

```
public class MondoAnimale {
  public static void main(String args[]) {
    Cavallo c = new Cavallo("Furia del West");
    Uomo    h = new Uomo("Johnny");
    Corvo   w = new Corvo("Il corvo dell'uva");

    Tonno   t = new Tonno("Palmera");
    // AnimaleMarino p = new AnimaleMarino("x");
    // ERRATO: classe astratta!
    Uccello u = new Uccello("Gabbiano");
    Pinguino p = new Pinguino("Tweety");

    c.mostra();     h.mostra();
    w.mostra();     t.mostra();
    u.mostra();     p.mostra();
  }
}
```

Output del programma:

```
Furia del West, un cavallo, nitrisce, si muove
avanzando su 4 zampe e vive sulla terraferma.
Johnny, un homo sapiens, parla, si muove
camminando su 2 gambe e vive in un condominio.
Il corvo dell'uva, un corvo, gracchia, si muove
volando e vive in un nido su un albero.
...
```

ARRAY DI ANIMALI

Grazie alla compatibilità fra tipi, **è possibile pensare a strutture-dati (in particolare, array) di *generici animali***, in cui però potranno trovare posto *specifici animali (concreti)* : un uccello, un cavallo, un tonno, etc.

ESEMPIO

```
public class Zoo {
  public static void main(String args[]) {
    Animale fauna[] = new Animale[6];
    fauna[0] = new Cavallo("Furia del West");
    fauna[1] = new Uomo("Johnny");
    fauna[2] = new Corvo("Il corvo dell'uva");
    fauna[3] = new Tonno("Palmera");
    fauna[4] = new Uccello("Gabbiano");
    fauna[5] = new Pinguino("Tweety");

    for(int i=0; i<6; i++) fauna[i].mostra();
  }
}
```

Da notare il ruolo fondamentale del polimorfismo!

EREDITARIETÀ E CLASSIFICAZIONE

Riprendiamo l'esempio degli animali, e consideriamo una classificazione secondo *due criteri*:

- ambiente in cui vivono (acquatici o terrestri)
- categoria zoologica (mammiferi, rettili,...)

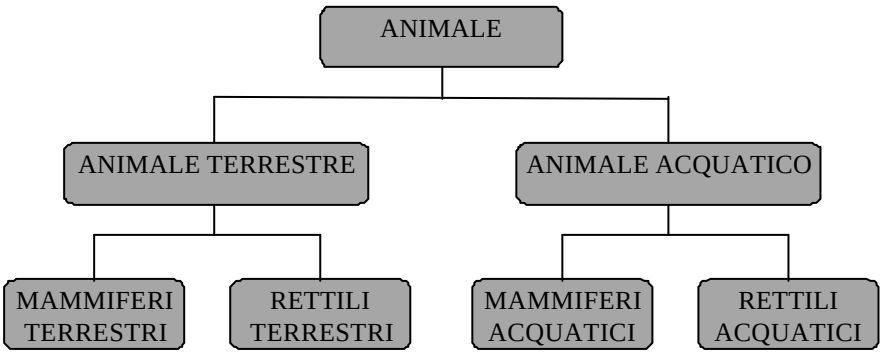
In forma tabellare, il tutto potrebbe presentarsi così :

	animali acquatici	animali terrestri
mammiferi		
rettili		

Supponiamo di voler riportare questa classificazione in una *tassonomia*, mediante *ereditarietà.*

Tale approccio richiede di *scegliere innanzitutto un criterio*, da usare per la creazione delle due sottoclassi "di primo livello".

Supponendo di optare per l'ambiente in cui vivono come sottoclassi di primo livello, e applicando poi l'altro criterio alle sottoclassi così ottenute, si ha la seguente tassonomia:

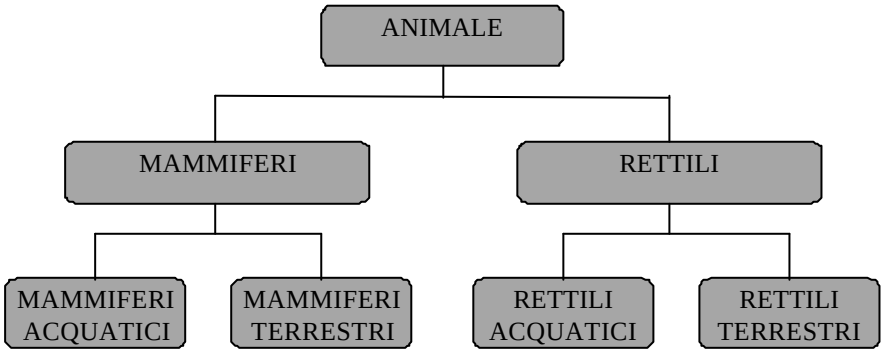


EREDITARIETÀ E CLASSIFICAZIONE (II)

Rispetto alla rappresentazione tabellare, *qualcosa è andato perso:* ***non esistono più i concetti di "mammiferi" e "rettili"***, che si sono *dispersi nelle sottoclassi*.

Per recuperare il concetto di "mammifero", si può solo osservare che esistono "animali acquatici mammiferi" e "animali terrestri mammiferi", ognuno con le sue proprie caratteristiche, ma non esiste la categoria dei "mammiferi" *con le peculiarità che la contraddistinguono*.
Abbiamo quindi perso la possibilità di denotare le *caratteristiche comuni a tutti* (e soli) i mammiferi.

In alternativa avremmo potuto adottare mammiferi e rettili come classificazione di primo livello:



Il questo caso, al contrario, avremmo perso la rappresentazione pura di "animale acquatico" e "animale terrestre".

EREDITARIETÀ E CLASSIFICAZIONE (III)

Attraverso il meccanismo di ereditarietà, è come se, nella rappresentazione tabellare precedente, fosse possibile selezionare *intere colonne*, accedendo poi *tramite esse* alle singole celle, *ma non intere righe*.

Questo è accaduto perché *si è stabilita una gerarchia fra i due criteri di classificazione, che in partenza erano ortogonali (sullo stesso piano),* e la ragione di ciò sta nel meccanismo di ereditarietà *singola* utilizzato per implementare la tassonomia.

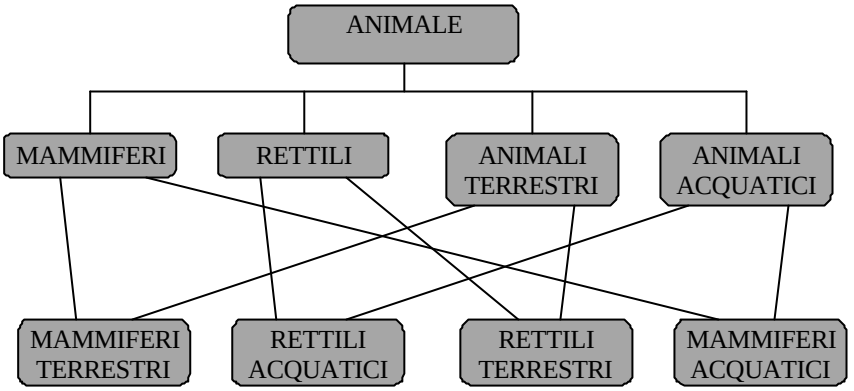
QUINDI: *l'ereditarietà singola è uno strumento molto potente, ma che non appare adatto a modellare tutte le situazioni*.

Sarebbe utile poter derivare le sottoclassi (mammiferi terrestri, rettili acquatici, etc) *a partire da più classi-base*, corrispondenti ciascuna all'applicazione di un criterio di classificazione.

In questo modo, i due criteri sarebbero sullo stesso piano e restrebbero ortogonali.

EREDITARIETÀ MULTIPLA

Quando è possibile derivare una classe facendole ereditare le proprietà di più classi base si parla di ***ereditarietà multipla.***
L'ereditarietà multipla è un meccanismo molto potente, *ma molto*



discusso, perché introduce ambiguità non banali da risolvere.
Poiché la sottoclasse è sottotipo di tutte le sue classi base ne unisce in sé i *dati* e i c*omportamenti* (metodi), ***MA COSA SUCCEDE SE:***

- nelle classi base vi sono campi dati o metodi omonimi? Per esempio, con riferimento alla classe *mammiferi terrestri*, cosa succede se c'e' un metodo *print()* sia in *mammiferi* che in *animali terrestri*? Quale dei due viene ereditato? Entrambi? Uno Solo? Da quale classe base?
- una classe eredita, direttamente o indirettamente, più volte da una stessa classe base? I campi dati di quest'ultima sono replicati o no? I metodi si considerano ambigui o no? Per esempio, *mammiferi terrestri* eredita indirettamente DUE VOLTE dalla classe *animale*. Cosa succede alla varibile nome?

Per questi motivi **Java non supporta l'ereditarietà multipla** (disponibile, invece - con tutti i problemi del caso - in C++).

Java introduce invece un diverso concetto più semplice da gestire: l' *interfaccia*.

INTERFACCE

Una interfaccia (`interface`) è analoga a una classe, ma, a differenza di questa, *costituisce una pura specifica di comportamenti*. Come tale:

- si limita a *dichiarare* i metodi, *senza implementarli* (tutti i metodi di un'interfaccia sono implicitamente `abstract`)
- in più, può solo definire costanti (cioè variabili `static final`), *ma non variabili.*

ESEMPIO

```
public interface AnimaleTerrestre {
   public deambula();
}
```

Le interfacce possono essere estese per ereditarietà, esattamente come le classi:

```
public interface Primate extends Mammifero {
   public void manipola();
}
```

A differenza delle classi, però, **le interfacce possono essere strutturate anche in gerarchie di ereditarietà *multipla.***

```
public interface SuperPippo
        extends Mammifero, SuperEroe {
   ...
}
```

Il motivo è che le interfacce, *non fornendo implementazioni*, evitano alla radice i problemi dell'ereditarietà multipla "classica" fra classi, consentendo una ereditarietà multipla "controllata".

USO DELLE INTERFACCE

Una interfaccia viene usata da (una o più) classi, che **dichiarano di *implementare l'interfaccia.*** Esempio:

```
public class MammiferoTerrestre
            implements AnimaleTerrestre {
   public void deambule() {DEFINIZIONE METODO!}
}
```

Una classe può ovviamente implementare una (o più) interfacce e contemporaneamente ereditare da una classe:

```
public class MammiferoTerrestre extends Mammifero
        implements AnimaleTerrestre {
   public void deambule() {DEFINIZIONE!}
}
```

È anche possibile definire strutture dati (es. array) di un tipo interfaccia, esattamente come si farebbe con una classe astratta: ovviamente, le istanze ivi memorizzate saranno di una classe (concreta) che implementi l'interfaccia:

```
public class Esempio {
   public static void main(String args[]) {
  AnimaleTerrestre a[] = new
AnimaleTerrestre[20];
      a[1] = new MammiferoTerrestre();
}}
```

In varie situazioni, Java definisce delle *interfacce vuote*: tali sono, ad esempio, `Serializable` e `Cloneable`. Il loro scopo è *fare da "marcatori"*, obbligando le classi che vogliono sfruttare certe funzionalità a indicarlo in modo esplicito, dichiarando di implementare tali interfacce.

INTERFACCE E METODOLOGIA

Le interfacce introducono un ***diverso modo di concepire il progetto.***

<u>Prima</u> si definiscono le interfacce che occorrono e si stabilisce in che relazione devono essere fra loro

<u>Poi</u> si definiscono le classi che le implementano

<u>Vantaggi:</u>

- la gerarchia delle interfacce, che riflette il progetto, è separata da quella delle classi, che riflette scelte implementative
- è possibile scegliere a ogni livello di progetto ciò che è più appropriato

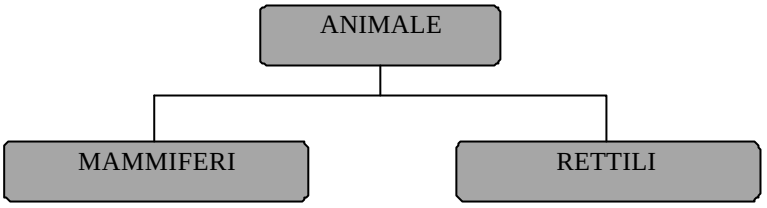
Ad esempio, l'interfaccia B può estendere l'interfaccia A, ma questo non implica che la classe Y, che implementa B, debba estendere la classe X che implementa A…

Anzi, le due classi potrebbero essere in relazione opposta, o anche non avere alcuna relazione fra loro!

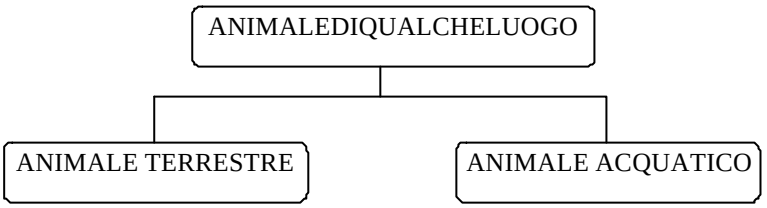
ESEMPIO: IL REGNO ANIMALE

Due classificazioni ortogonali, una di classi e una di interfacce.

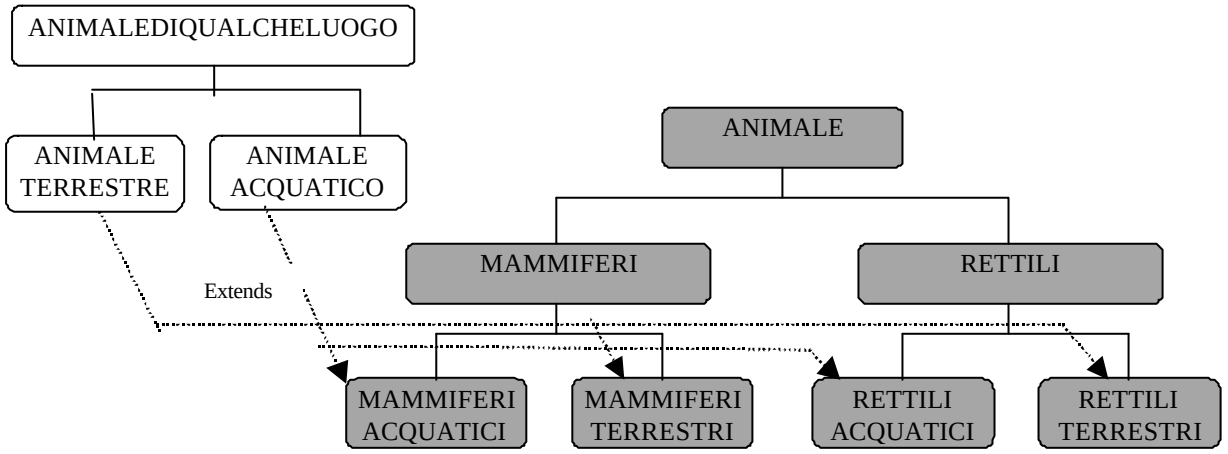
Gerarchia di classi:



Gerarchia di Interfacce:



Sistema Finale:



Si poteva pensare di invertire i ruoli delle gerarchie di interfacce e di classe, ma è certamente più naturale pensare a mammiferi e rettili come classi (con metodi e proprietà che specificano le caratteristiche peculiari di mammiferi e rettili), e quindi pensare al posto dove vivono gli animali in termini di interfacce (con metodi che specificano come gli animali si muovono…)

ESEMPIO: NUMERI COMPLESSI

In un linguaggio a oggetti che offra solo classi (C++):

- se si dispone già di una classe che rappresenta i numeri reali, viene spontaneo definire la classe dei complessi come sottoclasse dei reali, in quanto, *operativamente*, si tratta di aggiungere un campo-dati (la parte immaginaria)
- ma un tale modo di procedere darebbe però *risultati assurdi*, perché porterebbe a un modello del mondo in cui i complessi sono un sottotipo dei reali, *il che è contraddetto dalla realtà.*
- occorre dunque *invertire il rapporto fra le classi*, definendo i reali come sottoclasse dei complessi
- tuttavia, questo comporta *inefficienza*, in quanto ogni reale ha anche una parte immaginaria (che dovrà valere sempre 0).

In un linguaggio a oggetti con interfacce, invece:

- *il rapporto fra le interfacce* può essere quello suggerito della realtà da modellare (qui: reali che derivano da complessi)
- *ma il rapporto fra le classi che le implementano può essere diverso*, in particolare, ispirato a criteri di efficienza (qui: la classe dei complessi può avere due campi dati, quella dei reali uno solo)

NOTA FINALE: METODOLOGIE DI ANALISI E PROGETTO

La programmazione ad oggetti ha un forte impatto sul processo di produzione del software perchè

INVITA FORTEMENTE AD ANALIZZARE E PROGETTARE UN SISTEMA SOFTWARE PRIMA DI REALIZZARLO

Fase di Analisi:

- identificare che cosa il sistema deve fare e in che ambiente applicativo si colloca
- identificare che oggetti il sistema deve modellare, intendendo sia oggetti come astrazioni di ogget reali, sia oggetti concepiti come pure entità software
- identificare le proprietà e i servizi che devono essere serviti dagli oggetti

Fase di Progetto:

- definire le classi del sistema
- analizzare le caratteristiche degli oggetti identificati nella fase di analisi, estrarre le proprietà comuni
- definire le gerarchie di ereditarietà, possibilmente usando classi astratte, e sfruttando le interfacce per estrarre proprietà comuni che escono dalla classificazione della gerarchia di ereditarietà
- eventualmente ri-usare classi e grarchie precedentemente definite o disponibili in librerie

Fase di Realizzazione:

- scrivere le classi del sistema e i metodi