Dipartimento di Informatica e Sistemistica
Antonio Ruberti

“Sapienza” Università di Roma

Classi Astratte

Corso di Tecniche di Programmazione

Laurea in Ingegneria Informatica

(Canale di Ingegneria delle Reti e dei Sistemi Informatici - Polo di Rieti)

Anno Accademico 2007/2008

Prof. Paolo Romano

Si ringrazia il Prof. Enrico Denti per aver reso
disponibile il materiale didattico sul quale si basano queste slides

ENTITÀ ASTRATTE

- **Moltissime entità che usiamo per descrivere il mondo *non sono reali***
- **sono *pure categorie concettuali,* ma sono comunque *utili per esprimersi***

ESEMPIO: GLI ANIMALI

- **parlare di “animali” ci è molto utile, ma a ben pensarci *non esiste “il generico animale”!***
- **nella realtà esistono solo *animali specifici***
  - **cani, gatti, lepri, serpenti, pesci, …**

CLASSI ASTRATTE: L'IDEA

Da qui nasce il concetto di *CLASSE ASTRATTA*

- **una classe che rappresenta una categoria concettuale astratta**
  - **ad esempio, `Animale`**
- **di cui quindi *non possono esistere istanze***
  - **perché non esistono "animali qualsiasi"**
- **Le istanze apparterranno invece a sue *sottoclassi concrete***
  - **ad esempio `Cane`, `Gatto`, `Corvo`, ...**

CLASSI ASTRATTE: SINTASSI

- **Una *CLASSE ASTRATTA* potrebbe essere realizzata da una normale classe**
  - **con la convenzione di non crearne mai istanze**
- **ma è meglio poterla qualificare esplicitamente**
  - **in modo che la sua *natura astratta* sia *evidente***
  - **e che il compilatore possa *verificare* l'assenza di istanze**
- **Per questo Java introduce la parola chiave `abstract` per etichettare**
  - **sia la classe in quanto tale**
  - **sia uno o più metodi (metodi astratti)**

CLASSI ASTRATTE: UN ESEMPIO

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

Esprime il fatto che ogni animale *si muove* in *qualche modo* e *vive da qualche parte*, *ma in generale non si può dire come*, perché varia da un animale all'altro.

Tecnicamente:

- **i metodi astratti *non hanno corpo,* c'è solo un "`;`"**
- **se *anche solo un metodo* è `abstract`, la classe intera dev'essere `abstract` – altrimenti, ERRORE**

CLASSI ASTRATTE: COME

La classe astratta è un potente strumento per modellare gli aspetti comuni di molte realtà

Operativamente, una classe astratta:

- **lascia "in bianco" uno o più metodi, dichiarandoli *senza però definirli***
- **tali metodi verranno prima o poi *implementati da qualche classe derivata (concreta)***
  - **per essere concreta, una classe deve disporre di una implementazione per *tutti* i metodi ex-astratti.**
  - **se ne implementa solo alcuni, rimane astratta.**

CLASSI DERIVATE ASTRATTE..

- **Una classe derivata può definire uno o più metodi che erano astratti nella classe base**
  - **Se *anche solo un metodo* rimane astratto, la classe derivata è comunque *astratta***
    (e deve essere qualificata come tale)

ESEMPIO

```
public abstract class AnimaleTerrestre
                              extends Animale {

 public String vive() { // era abstract
   return "sulla terraferma"; }
 ...
}
```

AnimaleTerrestre definisce concretamente UNO dei due metodi astratti, ma NON L'ALTRO
→ È anch'essa una classe astratta

..E CLASSI DERIVATE CONCRETE

- **Una classe derivata concreta possiede una implementazione di tutti i metodi ex-astratti**
  - **o perché li implementa lei**
  - **o perché eredita qualche implementazione da una classe superiore**

ESEMPIO:

```
public class Gatto extends AnimaleTerrestre {
  // vive() era già stato implementato sopra
  public String siMuove() { // era abstract
   return "saltando"; }
}
```

`Gatto` implementa ANCHE L'ALTRO metodo lasciato astratto da `AnimaleTerrestre`
→ `Gatto` non è più astratta, è concreta

UN ESEMPIO COMPLETO (1/7)

Completiamo il mini-esempio precedente:

- **ogni animale risponde a tre metodi che restituiscono una stringa descrittiva:**
  - `chiSei()` **fornisce il NOME dell'animale**
  - `vive()` **indica DOVE VIVE l'animale**
  - `siMuove()` **indica COME SI MUOVE l'animale**
- **un metodo `mostra()` *indipendente* dallo specifico animale stampa a video i dati dell'animale**
- **tutti gli animali hanno la stessa rappresentazione interna, data dal loro *nome* e dal loro *verso.***

UN ESEMPIO COMPLETO (2/7)

```
public abstract class Animale {
  private String nome;
  protected String verso;

  public Animale(String s) { nome=s; }

  public abstract String siMuove();
  public abstract String vive();
  public abstract String chiSei();

  public void mostra() {
   System.out.println(nome + ", " +
     chiSei() + ", " + verso +
     ", si muove " + siMuove() +
     " e vive " + vive() );  }
}
```

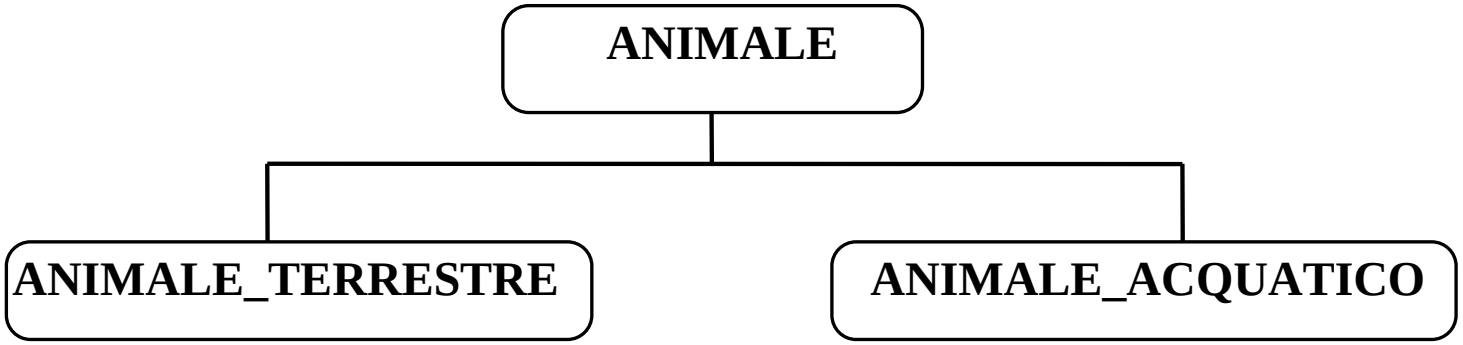
rappresentazione interna uguale per tutti

metodi astratti

usa i metodi astratti !

UN ESEMPIO COMPLETO (3/7)

Una possibile classificazione:



Sono ancora classi astratte:

- nulla si sa del movimento
- quindi è impossibile definire il metodo **`siMuove()`**

UN ESEMPIO COMPLETO (4/7)

```
public abstract class AnimaleTerrestre
      extends Animale {

   public AnimaleTerrestre(String s) {
      super(s); }

   public String vive() {
      return "sulla terraferma"; }

   public String chiSei() {
      return "un animale terrestre"; }
}
```

necessario per inizializzare il nome, che è privato nella classe base

Due metodi astratti su tre sono ridefiniti, ma *uno è ancora astratto*
→ la classe è ancora astratta

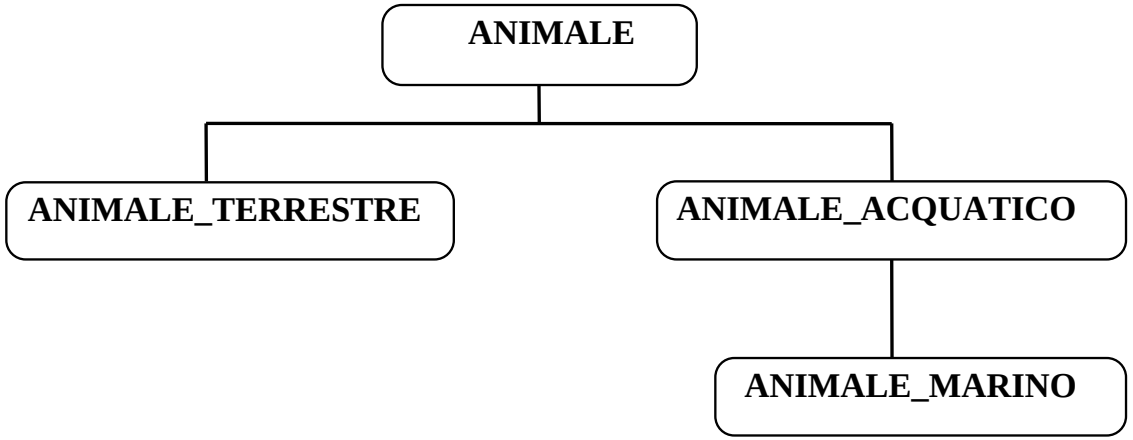
UN ESEMPIO COMPLETO (5/7)

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

Due metodi astratti su tre sono ridefiniti, ma *uno è ancora astratto*
→ la classe è ancora astratta

UN ESEMPIO COMPLETO (6/7)

Una possibile specializzazione:



Perché introdurre l'animale marino?

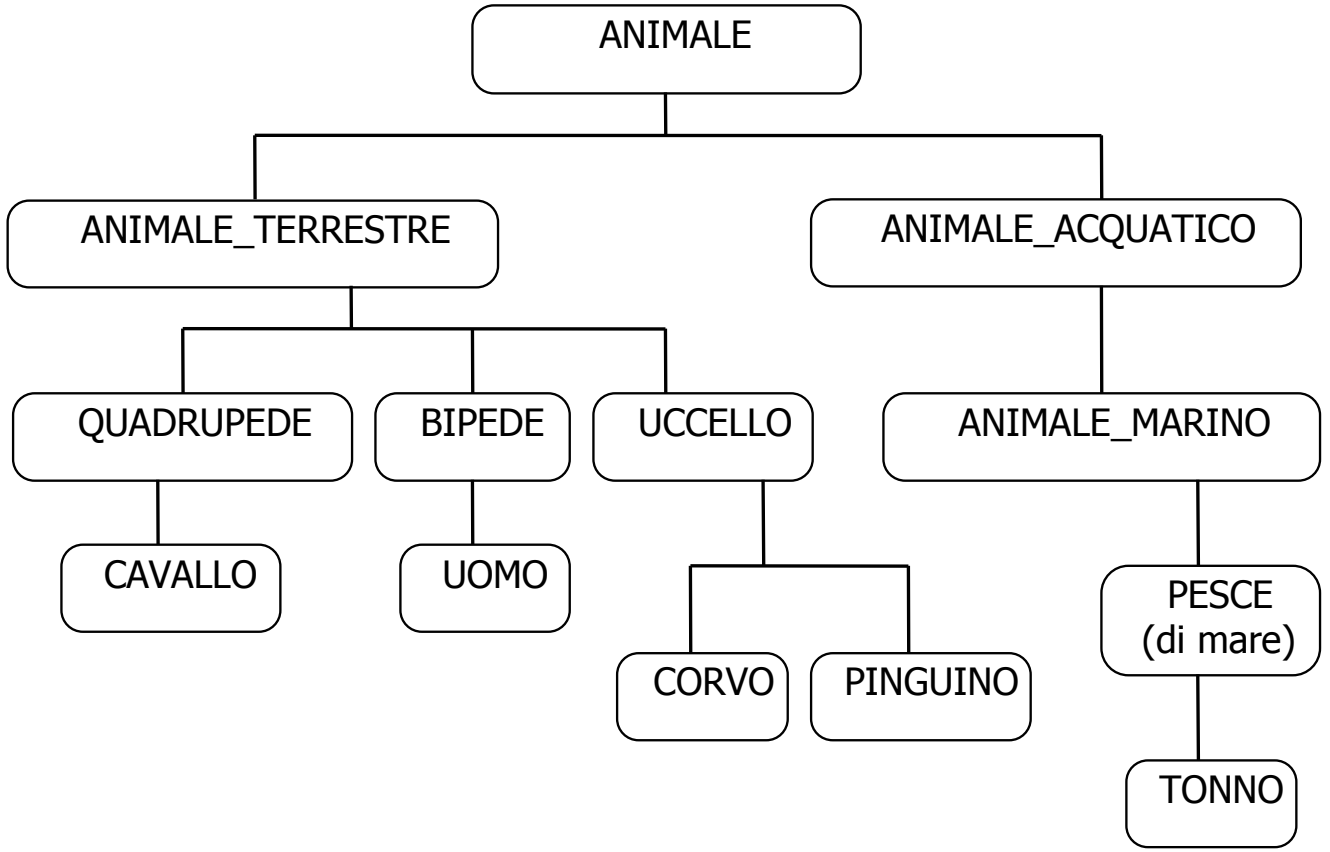
- non è correlato ad ambiente di vita o movimento
- rispecchia semplicemente una realtà che ci interessa.

UN ESEMPIO COMPLETO (7/7)

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

Specializza i metodi **`vive()`** e **`chiSei()`**,
ma non definisce il terzo, `siMuove()`
→ la classe è ancora astratta

LA TASSONOMIA COMPLETA



LE CLASSI CONCRETE (1/4)

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

possibile inizializzare direttamente il verso, perché protected nella classe base

Definisce l'ultimo metodo astratto rimasto, siMuove() → la classe non è più astratta

LE CLASSI CONCRETE (2/4)

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

LE CLASSI CONCRETE (3/4)

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

LE CLASSI CONCRETE (4/4)

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

ALTRE CLASSI PIÙ SPECIFICHE

```
public class Cavallo extends Quadrupede {
  public Cavallo(String s) {
      super(s);verso = "nitrisce"; }
  public String chiSei() { return "un cavallo"; }
}
public class Corvo extends Uccello {
  public Corvo(String s) {
      super(s);   verso = "gracchia"; }
  public String chiSei() { return "un corvo"; }
}
public class Tonno extends PesceDiMare {
  public Tonno(String s) { super(s); }
  public String chiSei() {
      return "un tonno"; }
}
```

ALTRE CLASSI PIÙ SPECIFICHE

```
public class Uomo extends Bipede {
  public Uomo(String s) { super(s); verso = "parla"; }
  public String siMuove() {
      return "camminando su 2 gambe"; }
  public String chiSei() {
      return "un homo sapiens"; }
  public String vive() { return "in condominio"; }
}
public class Pinguino extends Uccello {
  public Pinguino(String s) {
      super(s); verso = "non fa versi"; }
  public String chiSei() {    return "un pinguino"; }
  public String siMuove() { return "ma non sa volare";
  }
}
```

UN MAIN “ZOO”

```
public class Zoo {
 public static void main(String[] args) {
  Animale[] zoo = new Animale[6];
  zoo[0] = new Cavallo("Varenne");
  zoo[1] = new Uomo("John");
  zoo[2] = new Corvo("Crowy");
  zoo[3] = new Tonno("TonTon");
  zoo[4] = new Uccello("Tweety");
  zoo[5] = new Pinguino("Penguin");
  for(int i=0; i<6; i++) zoo[i].mostra();
 }
```

Possibile usare anche il nuovo `for`

POLIMORFISMO: il metodo `mostra()` si comporta in modo *specifico per ogni oggetto!*

UN MAIN “ZOO” col nuovo for

```
public class Zoo {
 public static void main(String[] args) {
  Animale[] zoo = new Animale[6];
  zoo[0] = new Cavallo("Varenne");
  zoo[1] = new Uomo("John");
  zoo[2] = new Corvo("Crowy");
  zoo[3] = new Tonno("TonTon");
  zoo[4] = new Uccello("Tweety");
  zoo[5] = new Pinguino("Penguin");
  for(Animale a : zoo) a.mostra();
 }
}
```

UN MAIN “ZOO”

.. e il suo output:

```
Varenne, un cavallo, nitrisce, si muove
avanzando su 4 zampe e vive sulla
terraferma.

John, un homo sapiens, parla, si muove
camminando su 2 gambe e vive in un
condominio.

Crowy, un corvo, gracchia, si muove
volando e vive in un nido su un albero.

...
```

Il metodo `mostra()` è polimorfo in quanto monta (in modo fisso) i "pezzi" forniti da *metodi polimorfi*

Un esempio più complesso:
Forme geometriche

L'OBIETTIVO

Definire una tassonomia di *forme geometriche*

- **non esiste la "generica forma geometrica"!**
- **esistono *triangoli, quadrilateri, pentagoni, ...***

Forme* può ben essere una *classe astratta

Quali caratteristiche comuni a tutte le forme?

- **ogni "forma geometrica" possiede un'*area* e un *perimetro*** *(… e dei lati? o no? il cerchio…?)*
- **ogni "forma geometrica" deve potersi *stampare* *(o visualizzare…)***

QUALE CLASSIFICAZIONE?

Se partizioniamo le forme per *numero di lati*:

- **triangoli**
  - **ulteriore criterio di classificazione: *lati uguali?***
    ⇒ scaleno, isoscele, equilatero
  - **"scaleno" = triangolo qualsiasi o con 3 lati diversi?**
- **quadrilateri**
  - **quadrilatero qualsiasi**
  - **ulteriore criterio di classificazione: *lati paralleli?***
    ⇒ trapezio, parallelogrammo, rombo
  - **ulteriore criterio di classificazione: *angoli retti?***
    ⇒ trapezio rettangolo, rettangolo, quadrato

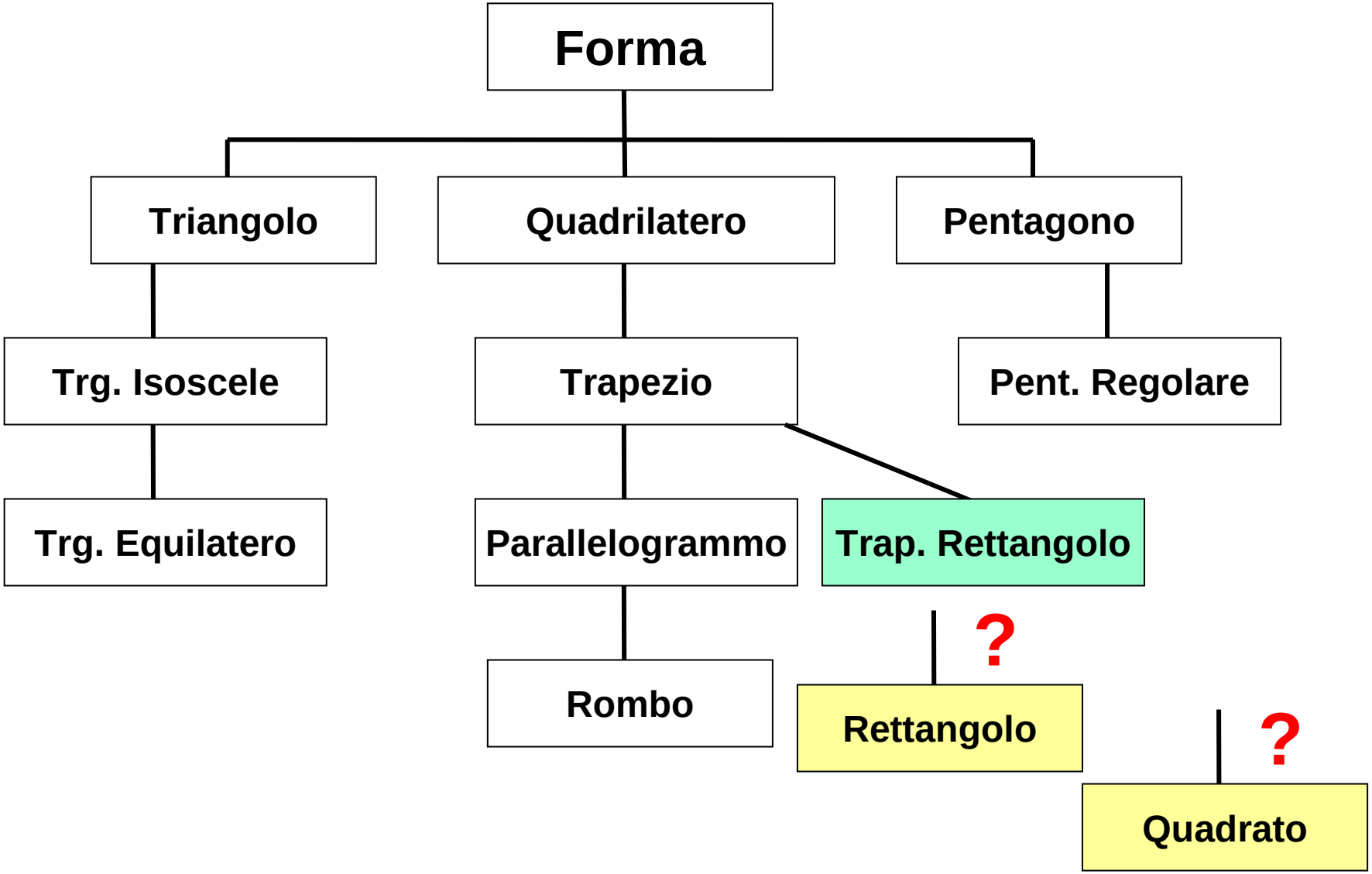
…e il triangolo rettangolo..?

CLASSIFICAZIONE: UNA IPOTESI

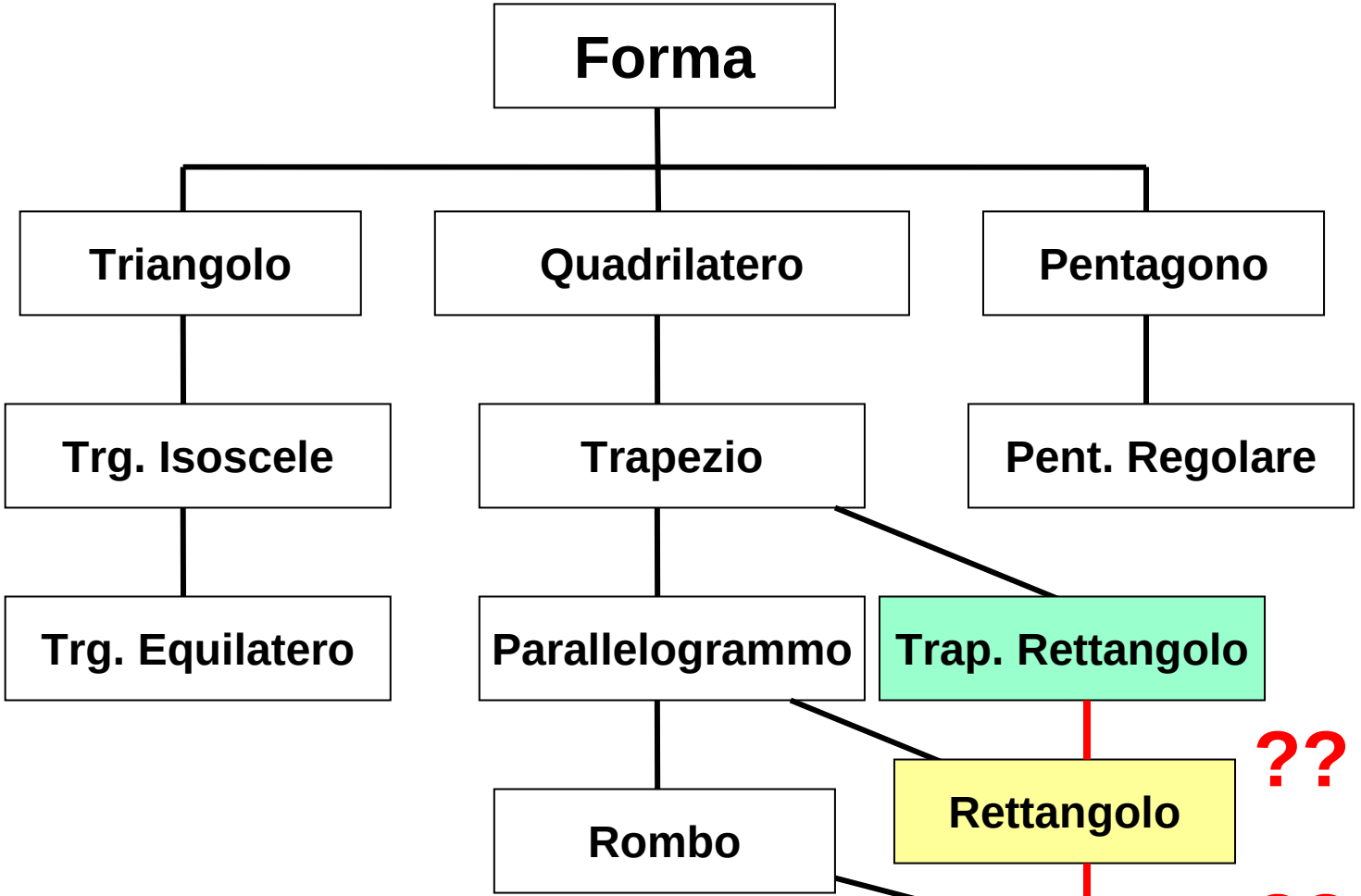
- **triangoli**
  - **scaleno, isoscele, equilatero**
  - **"scaleno" = *triangolo con i tre lati diversi***
- **quadrilateri**
  - **quadrilatero qualsiasi**
  - **criterio fondamentale: *lati paralleli***
    ⇒ trapezio, parallelogrammo, rombo
  - **eventuale ulteriore criterio: *angoli retti***
    ⇒ trapezio rettangolo, rettangolo, quadrato

Come definire la tassonomia in modo coerente?

UNA POSSIBILE TASSONOMIA



UNA POSSIBILE TASSONOMIA

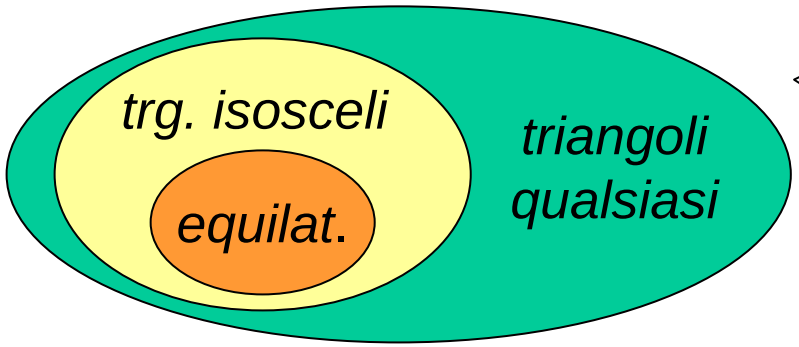


Dove porre Rettangolo e Quadrato?

CLASSIFICAZIONE: IL PROBLEMA

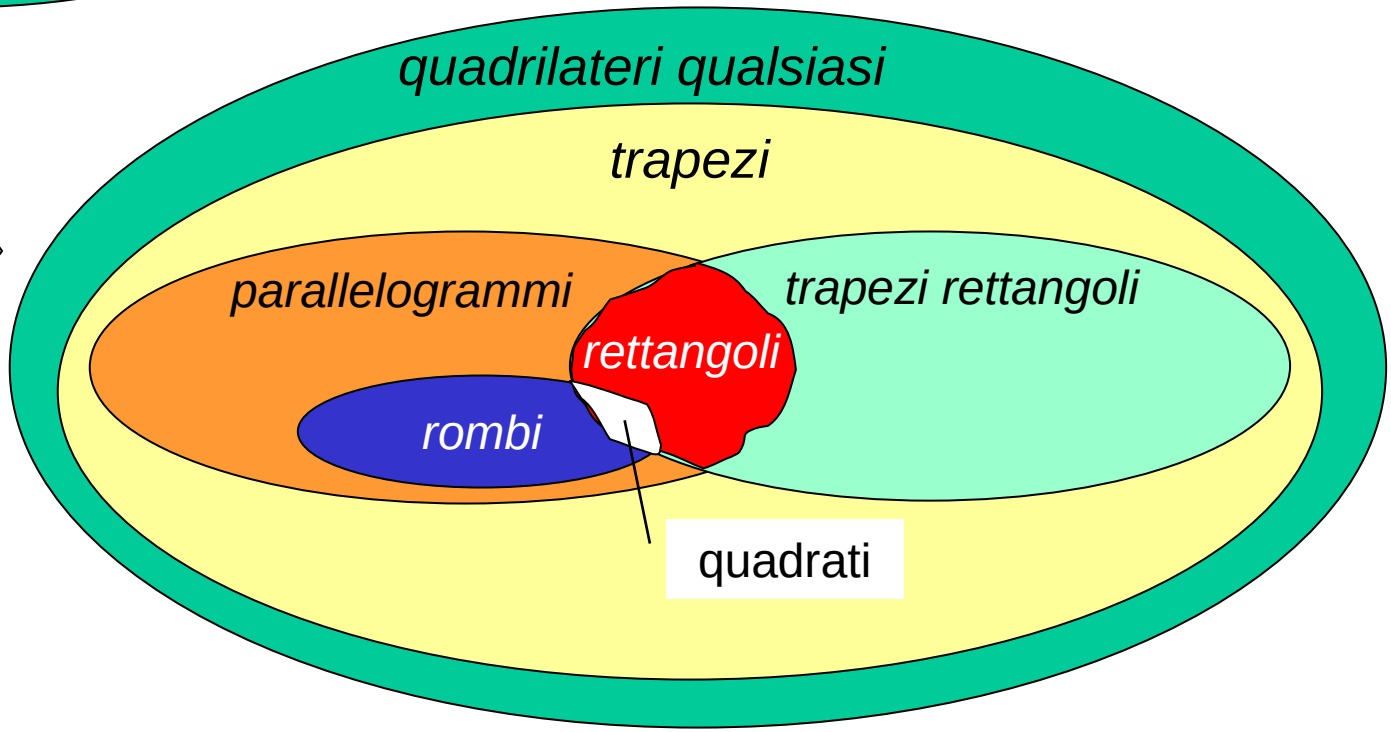
- **L'ereditarietà (singola) classifica *definendo sottoinsiemi***
- **Permette di applicare *un solo criterio di classificazione* per volta**
  - **un insieme non può essere sottoinsieme *di due insiemi diversi* (se uno non è già incluso nell'altro)**
- triangoli: esiste una relazione di inclusione
- quadrilateri:*non esiste* una relazione di inclusione perché i diversi quadrilateri sono caratterizzati da proprietà diverse e non correlate l'una all'altra (rettangolo: angoli retti; parallelogrammo: lati paralleli a due a due; etc)

RELAZIONI FRA INSIEMI



Fra le categorie di triangoli esiste una relazione di inclusione → *ben modella-te* dall'ereditarietà (singola) - se non consideriamo i *triangoli rettangoli*…!!

Fra quadrilateri la situazione è più complessa → *problemi di modellazione* con la sola ereditarietà singola



LA CLASSE BASE (astratta)

```
public abstract class Forma {
  public abstract double area();
  public abstract double perimetro();
  public abstract String nome();
  public String toString(){
    return nome() + " di area " +
           area() + " e perimetro " +
           perimetro();
  }
}
```

Servono dei costruttori?

Rappresentiamo qui i lati…?

UNA CLASSE CONCRETA

```
public class Triangolo extends Forma {
  protected double lato1, lato2, lato3;
   public String nome() {
       return "Triangolo qualsiasi"; }
  public double perimetro() {
       return lato1 + lato2 + lato3; }
  public double area() {
       // formule trigonometriche! }
  // altri metodi? magari per avere i lati?
}
```

QUALI E QUANTI COSTRUTTORI?
Con quali e quanti parametri?

I lati.. protetti o meglio privati?
Cosa rappresenta un triangolo? E' un valore o un contenitore? Si può *cambiare* qualcosa?

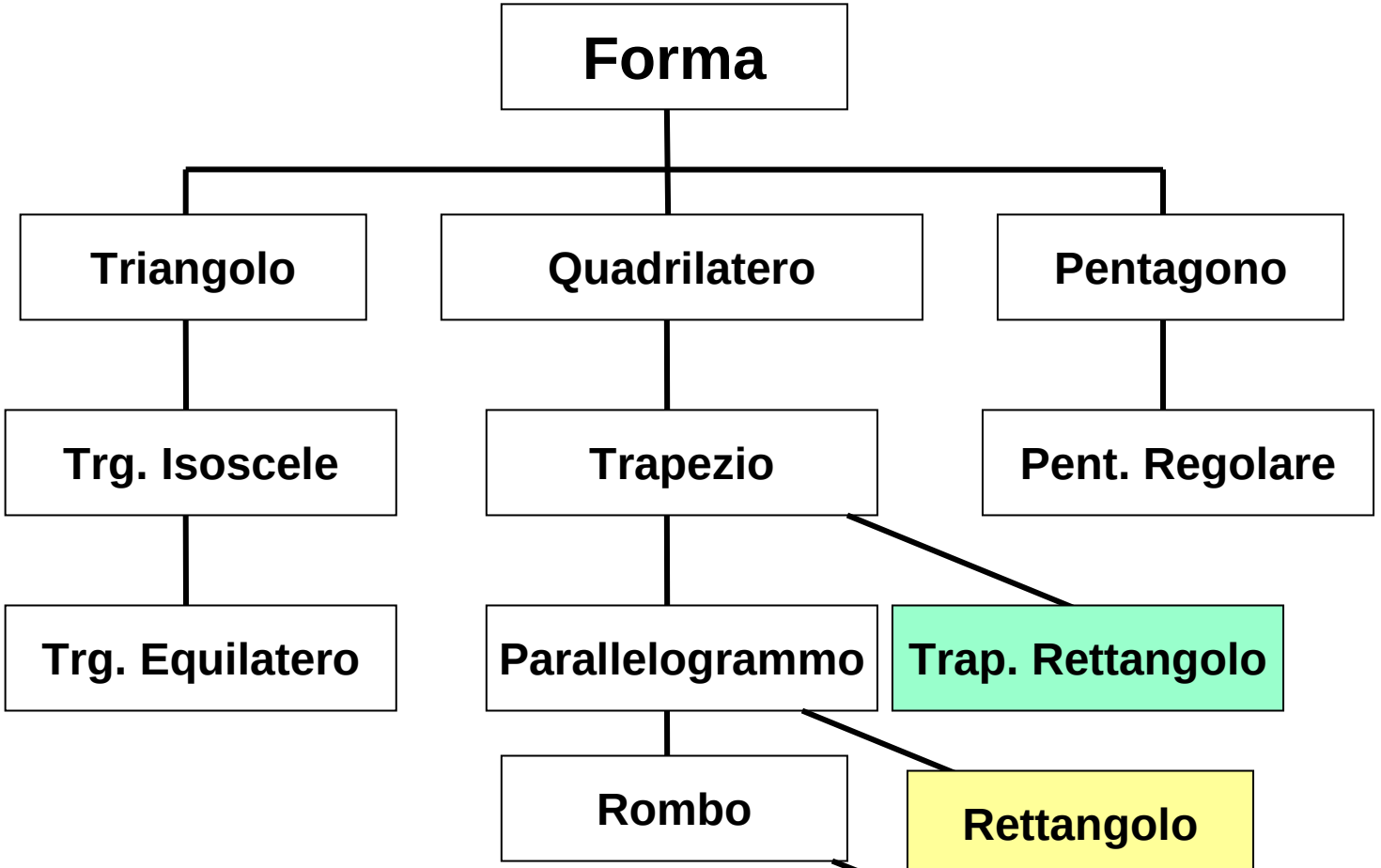
UNA CLASSE CONCRETA

```
public class TriangoloIsoscele extends Triangolo {
   public String nome() { // ridefinita
       return "Triangolo isoscele"; }
   // perimetro va già bene!!
   // area va bene ma può convenire ridefinirla
   // nuovi metodi specifici? base()? lato()?
}
```

La particolarità del triangolo isoscele giustifica l'introduzione di nuovi metodi specifici?

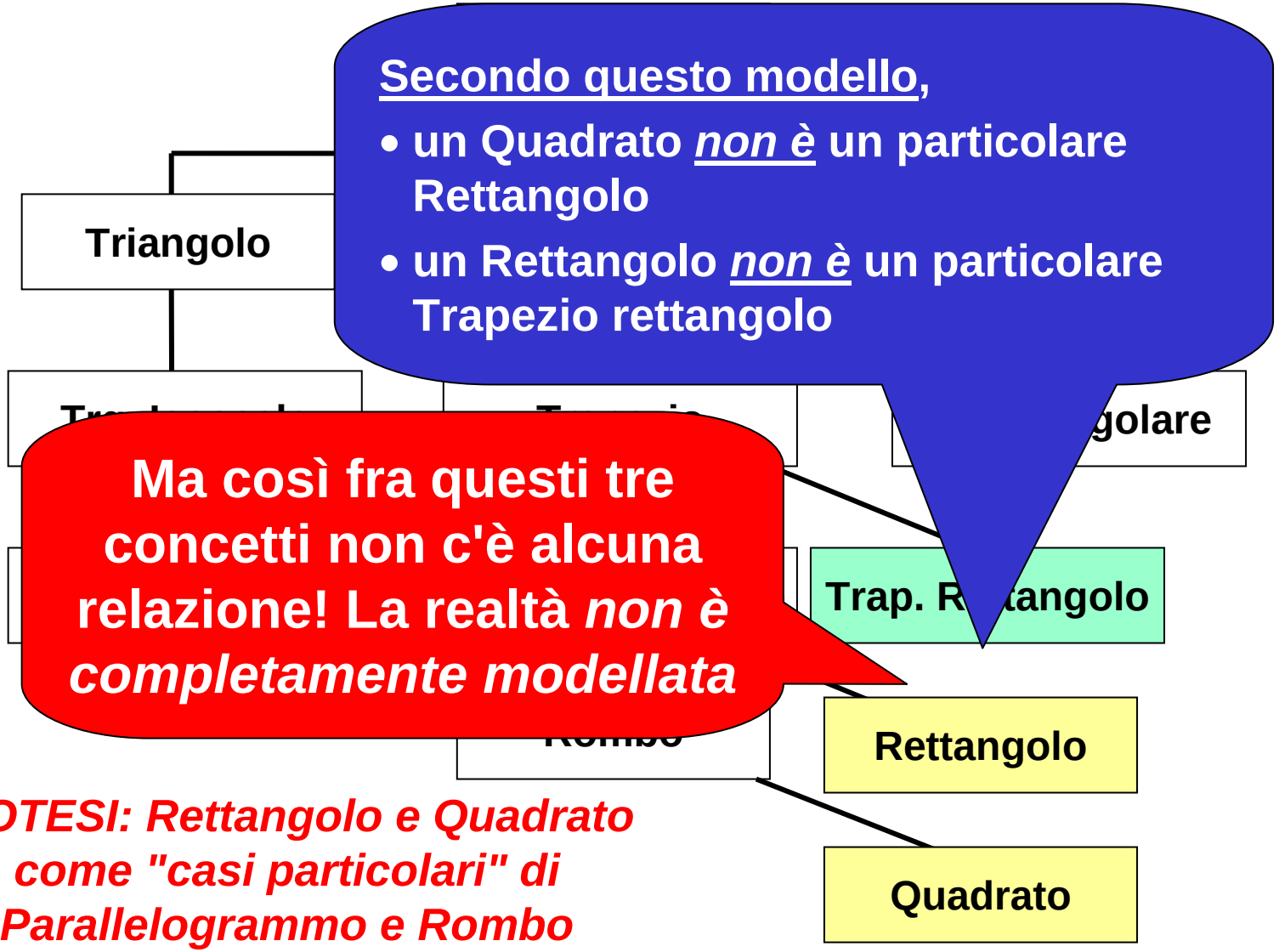
QUALI E QUANTI COSTRUTTORI? Con quali e quanti parametri? ***A cosa si appoggiano?***

UNA POSSIBILE TASSONOMIA



IPOTESI: Rettangolo e Quadrato come "casi particolari" di Parallelogrammo e Rombo

UNA POSSIBILE TASSONOMIA



IPOTESI: Rettangolo e Quadrato come "casi particolari" di Parallelogrammo e Rombo

UN MONDO DI FORME

```
public class TanteForme {
 public static void main(String args[]) {
  Forma forme[] = new Forma[10];
  forme[0] = new Triangolo(...);
  forme[1] = new TriangoloIsoscele(...);
  forme[2] = new TriangoloEquilatero(...);
  forme[3] = new Rettangolo(...);
  forme[4] = new Quadrato(...);
  for(Forma f : forme) System.out.println(f);
 }
}
```

DOMANDE & PROBLEMI

- **Il triangolo equilatero richiede aggiustamenti?**
  - **Quali e quanti costruttori ha?**
  - **Ha altri metodi particolari suoi propri?**
- **Rappresentare il triangolo con i tre lati *è la scelta migliore? C'erano alternative?***
  - **E se volessimo aggiungere i *triangoli rettangoli* (scaleni, isosceli.. ma mai equilateri!)?**
- **Come si rappresenta un *trapezio rettangolo*?**
  Possiamo assegnare un *Quadrato* a un *Rettangolo?*

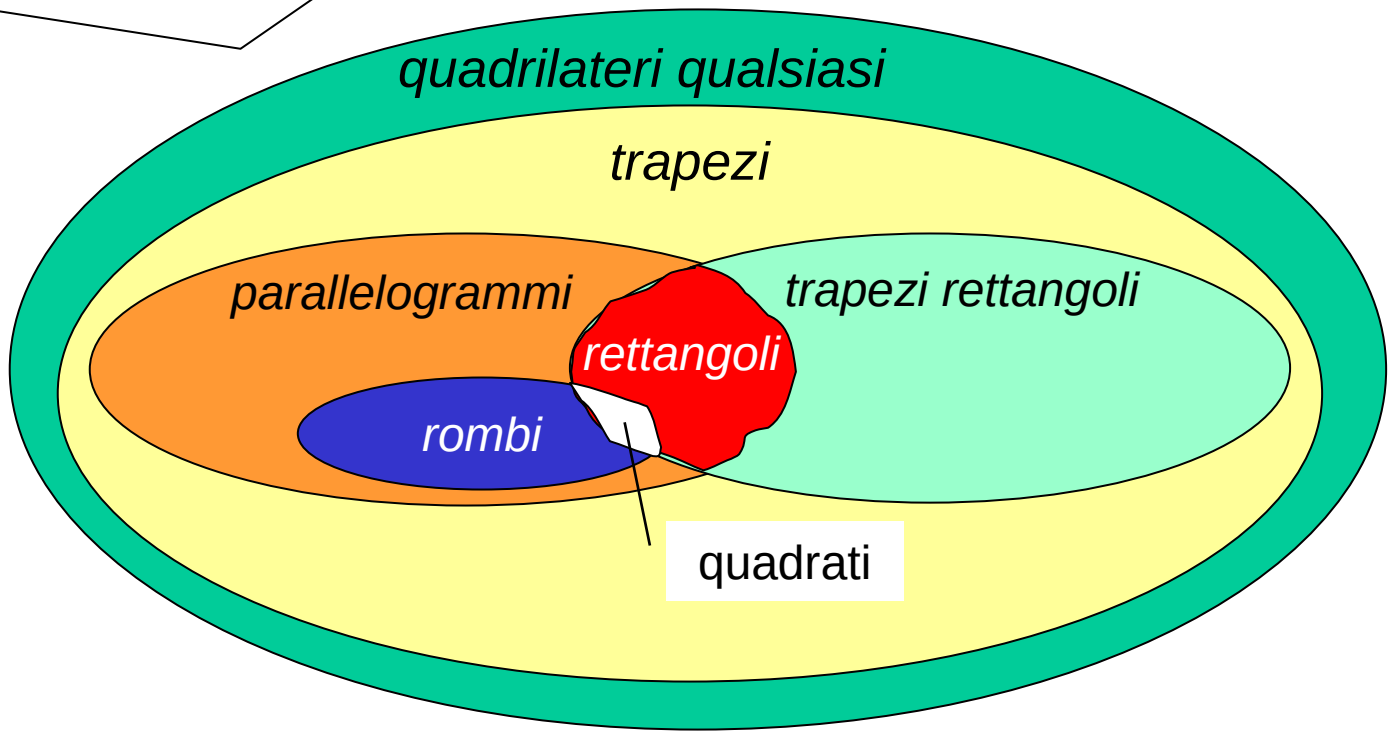
In breve:

La tassonomia regge il confronto con la realtà?

Manca qualcosa alla nostra possibilità di esprimerci?

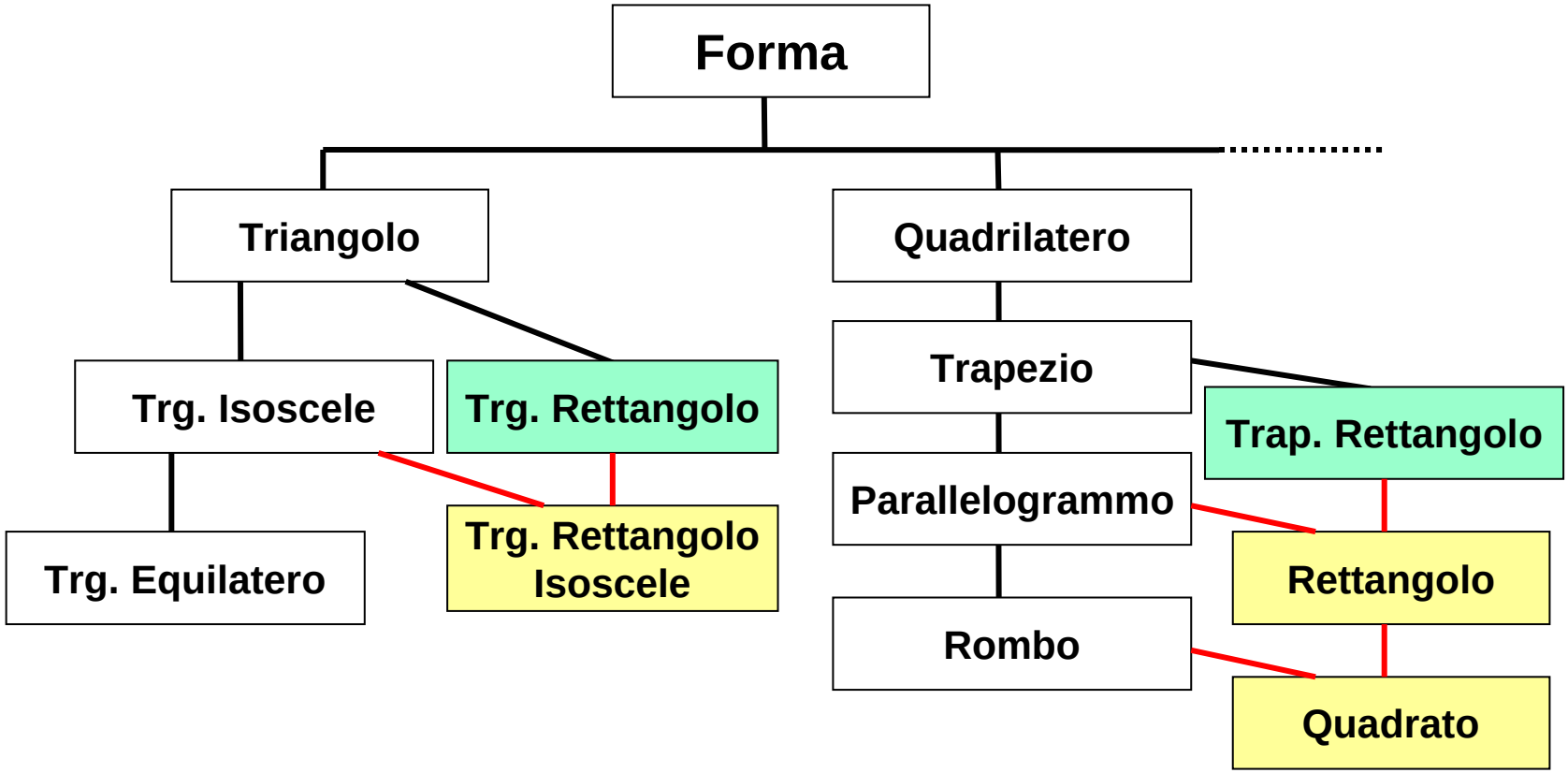
IL PUNTO

- **L'ereditarietà singola modella bene le relazioni insieme/sottoinsieme**
- **ma la nostra realtà prevede *intersezioni di insiemi,* che l'ereditarietà singola non può esprimere**



VERSO UN NUOVO OBIETTIVO

- **Disporre di un concetto di *ereditarietà multipla***
- **con cui catturare situazioni di *intersezione insiemistica***



EREDITARIETÀ MULTIPLA?

- **L'ereditarietà *multipla* è un strumento concettuale fondamentale, che tuttavia *comporta non pochi problemi pratici* se usata *fra classi***
  - **l'esperienza del C++ ha insegnato molto: dati ereditati duplicati, versioni di metodi in conflitto..**
- **Per questi motivi, in Java l'ereditarietà multipla esiste, MA *NON FRA CLASSI***
- **Per esprimerla in modo pulito, evitando i problemi, *verrà introdotto il concetto di INTERFACCIA***