Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Classi Astratte

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

ENTITÀ ASTRATTE

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

ESEMPI

- Parlare di «animali», «rocce» o «piante» ci è molto utile, ma a ben pensarci… *non esiste il generico «animale», la generica «roccia» o la generica «pianta»!*
- Nella realtà esistono solo *oggetti specifici*
  - Animali specifici: cani, gatti, lepri, serpenti, pesci, …
  - Rocce specifiche: granito, quarzo, basalto, …
  - …

CLASSI ASTRATTE: L'IDEA

Da qui nasce il concetto di *CLASSE ASTRATTA*

- per rappresentare una *categoria concettuale*
  - ad esempio **`Animale`**, **`Roccia`**, **`Finestra`**…
- di cui quindi *non possono esistere istanze*
  - perché non esistono "animali qualsiasi", o "rocce qualsiasi", o "finestre qualunque"

Gli oggetti effettivamente usati nell'applicazione apparterranno a specifiche *sottoclassi concrete*

- ad esempio **`Cane`**, **`Gatto`**, **`Corvo`**, **`...`**

CLASSI ASTRATTE: SINTASSI

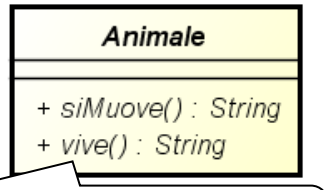
- Una classe astratta potrebbe essere realizzata da una normale classe, con la convenzione di non crearne istanze
- …ma è meglio poterla *qualificare esplicitamente come tale*
  - in modo che la sua *natura astratta* sia *evidente*
  - e che il compilatore possa verificare l'assenza di istanze
- Nuova parola chiave **`abstract`** per etichettare
  - sia la *classe* in quanto tale — Java, C#, Scala, Kotlin
  - sia alcuni *metodi* (metodi astratti) — Java, C#, Kotlin
  - sia, talora, alcuni *dati* (proprietà astratte) — Kotlin

CLASSI ASTRATTE: UN ESEMPIO

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

Java

C#



Classe e metodi astratti sono *in corsivo* in UML

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 il modo esatto dipende *dallo specifico animale*.

OPERATIVAMENTE:

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

Java

C#

Kotlin

Scala: i metodi astratti sono tali semplicemente perché non hanno corpo. Non si etichettano `abstract`.

Scala

CLASSI ASTRATTE: COME

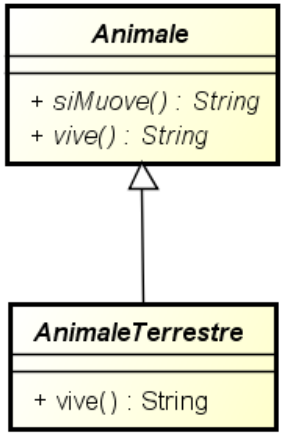
- Le classi astratte sono un *potente strumento* per *modellare gli aspetti comuni* di molte realtà.

Operativamente, una classe astratta:

- lascia “in bianco” uno o più metodi, dichiarandoli *senza però definirli*
  - al posto del corpo del metodo c'è un "`;`"
- tali metodi verranno prima o poi *implementati da qualche classe derivata (concreta)*
  - per essere concreta, una classe deve implementare *tutti* i metodi ex-astratti
  - se ne implementa solo alcuni, rimane astratta.

CLASSI DERIVATE ASTRATTE..

- Una classe derivata può implementare 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)
  - se invece tutti i metodi risultano definiti, la classe derivata è concreta (se ne potranno creare istanze)



ESEMPIO
La classe **`AnimaleTerrestre`** definisce il metodo **`vive`**, ma non **`siMuove`**: quindi, è ancora astratta

C#: sostituire `extends` con `:`

```
public abstract class AnimaleTerrestre extends Animale {

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

}
```

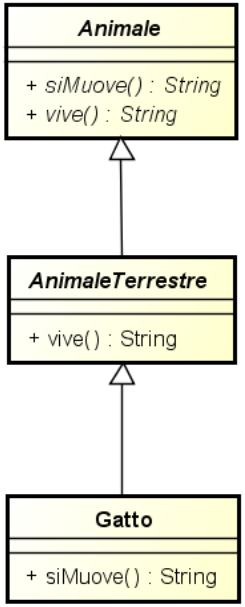
C#: aggiungere `override`

Java

~C#

..E CLASSI DERIVATE CONCRETE

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



ESEMPIO
La classe **`Gatto`** definisce il metodo **`siMuove`**, che era l'ultimo ancora astratto: quindi, è concreta.

C#: sostituire `extends` con `:`

```
public class Gatto extends AnimaleTerrestre {
   public String siMuove(){ return "saltando"; }
}
```

Java

C#: aggiungere `override`

~C#

RECAP (1/3)

- La parola chiave **`abstract`** si usa per etichettare
  - sia la *classe* in quanto tale — Java C# Scala Kotlin
  - sia alcuni *metodi* (metodi astratti) — Java C# Kotlin
  - sia, talora, alcuni *dati* (proprietà astratte) — Kotlin

Java / C#

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

Scala

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

Scala: i metodi astratti sono tali perché non hanno corpo. Non si usa **`abstract`**.

Kotlin

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

Kotlin: classe e metodi astratti sono implicitamente **`open`**.

RECAP (2/3)

Java

```
public abstract class AnimaleTerrestre extends Animale {
 public String vive() { return "sulla terraferma"; }
}
```

C#

```
public abstract class AnimaleTerrestre : Animale {
 public override String vive() { return "sulla terraferma"; }
}
```

Scala

```
abstract class AnimaleTerrestre extends Animale {
  override def vive() : String = { return "sulla terraferma"; }
}
```

Kotlin

```
public abstract class AnimaleTerrestre : Animale {
  public override fun vive() : String { return "sulla terraferma"; }
}
```

RECAP (3/3)

Java

```
public class Gatto extends AnimaleTerrestre {
   public String siMuove() { return "saltando"; }
}
```

C#

```
public class Gatto : AnimaleTerrestre {
 public override String siMuove() { return "saltando"; }
}
```

Scala

```
class Gatto extends AnimaleTerrestre {
  override def siMuove() : String = { return "saltando"; }
}
```

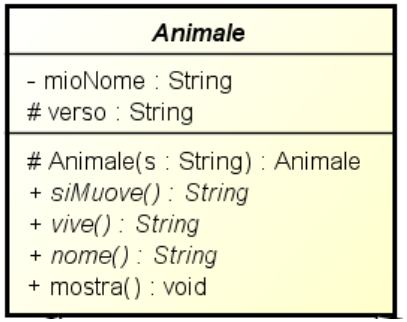
Kotlin

```
public class Gatto : AnimaleTerrestre {
  public override fun siMuove() : String { return "saltando"; }
}
```

Kotlin: classe e metodi concreti NON sono più implicitamente **open** → questa classe non sarà ulteriormente estendibile

UN ESEMPIO COMPLETO (1/9)

Generalizziamo il mini-esempio precedente:



- ogni animale dispone per ipotesi di tre metodi che restituiscono una stringa descrittiva:
  - **`nome`** fornisce il NOME/CATEGORIA 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 una descrizione completa dell'animale e del suo comportamento
- tutti gli animali hanno, per ipotesi, la stessa rappresentazione interna:
  - *mioNome* (privato): il nome proprio di quell'animale (es. "Fido")
  - *verso* (protected): il verso dell'animale
    → *poco realistico, ma utile a fini didattici in questo esercizio*
  - essendoci un campo privato, occorre un costruttore che lo inizializzi

UN ESEMPIO COMPLETO (2/9)

Java

~C#

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

rappresentazione interna uguale per tutti gli animali

metodi astratti

Il metodo **mostra** non è astratto esso stesso, ma usa i metodi astratti: *polimorfismo!*

un solo codice che però si sostanzia in *molte forme* diverse → *polimorfismo*

Vogliamo in output una frase del tipo:

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

mioNome | nome() | verso | siMuove() | vive()

UN ESEMPIO COMPLETO (3/9)

Scala

```
abstract class Animale(private val mioNome:String) {
   protected val verso : String;
   def siMuove() : String;
   def vive() : String;
   def nome() : String;
   def mostra() : Unit = {
    println(mioNome + ", " + nome() + ", " + verso +
        ", si muove " + siMuove() + " e vive " + vive() );  }
}
```

Dato astratto perché non inizializzato

metodi astratti

Kotlin

```
public abstract class Animale(private val mioNome:String) {
   protected abstract val verso : String;
   public fun siMuove() : String;
   public fun vive() : String;
   public fun nome() : String;
   public fun mostra() : Unit {
    println(mioNome + ", " + nome() + ", " + verso +
        ", si muove " + siMuove() + " e vive " + vive() );  }
}
```

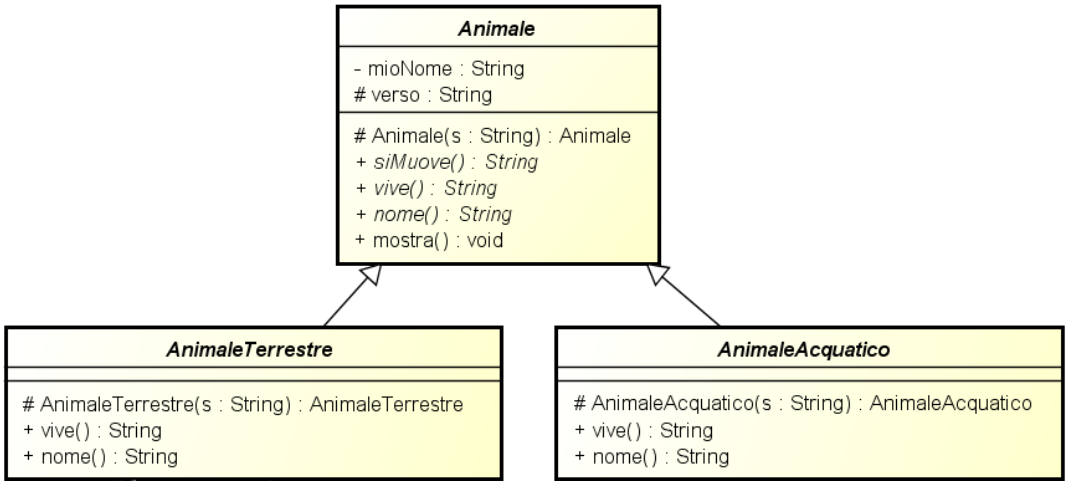
Dato astratto perché non inizializzato

In Kotlin richiede la keyword **abstract**

metodi astratti

UN ESEMPIO COMPLETO (4/9)

Una possibile classificazione:



- **`AnimaleTerrestre`** e **`AnimaleAcquatico`** sono ancora astratte, perché *nulla si sa del movimento*: quindi è impossibile in questo momento implementare il metodo **`siMuove`**

UN ESEMPIO COMPLETO (5/9)

Java

```
public abstract class AnimaleTerrestre extends Animale {
  protected AnimaleTerrestre(String s) { super(s); }

  public String vive() {
        return "sulla terraferma"; }
  public String nome() {
        return "un animale terrestre"; }
}
```

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

Due metodi astratti su tre sono concretizzati, ma *uno (siMuove) è ancora astratto:* perciò, queste due classi sono ancora astratte.

Java

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

UN ESEMPIO COMPLETO (6/9)

C#

```
public abstract class AnimaleTerrestre : Animale {
  protected AnimaleTerrestre(string s) : base(s) {}
  public override string vive() {
        return "sulla terraferma"; }
  public override string nome() {
        return "un animale terrestre"; }
}
```

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

Due metodi astratti su tre sono concretizzati, ma *uno (siMuove) è ancora astratto:* perciò, queste due classi sono ancora astratte.

C#

```
public abstract class AnimaleAcquatico : Animale {
  protected AnimaleAcquatico(string s) : base(s) {}
  public override string vive() {
        return "nell'acqua"; }
  public override string nome() {
        return "un animale acquatico"; }
}
```

UN ESEMPIO COMPLETO (7/9)

Scala

```
abstract class AnimaleTerrestre(mioNome:String)
                        extends Animale(mioNome) {
 override def vive() : String = {return "sulla terraferma"; }
 override def nome() : String = {return "un animale terrestre"; }
}
```

Kotlin

```
public abstract class AnimaleTerrestre(mioNome:String)
                        : Animale(mioNome) {
 public override fun vive():String {return "sulla terraferma"; }
 public override fun nome():String {return "un animale terrestre";
   }
}
```

UN ESEMPIO COMPLETO (8/9)

Una possibile specializzazione: l'animale marino

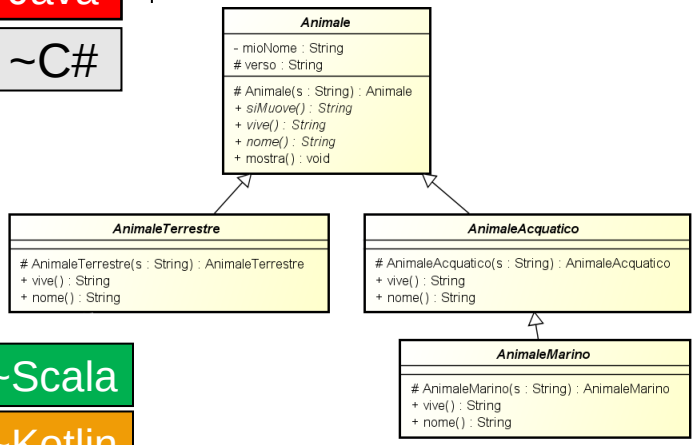
```
public abstract class AnimaleMarino
   extends AnimaleAcquatico {
 protected AnimaleMarino(String s){
    super(s); }
  public String vive() {
        return "in mare"; }
  public String nome() {
        return "un animale marino"; }
}
```

Java

~C#

~Scala

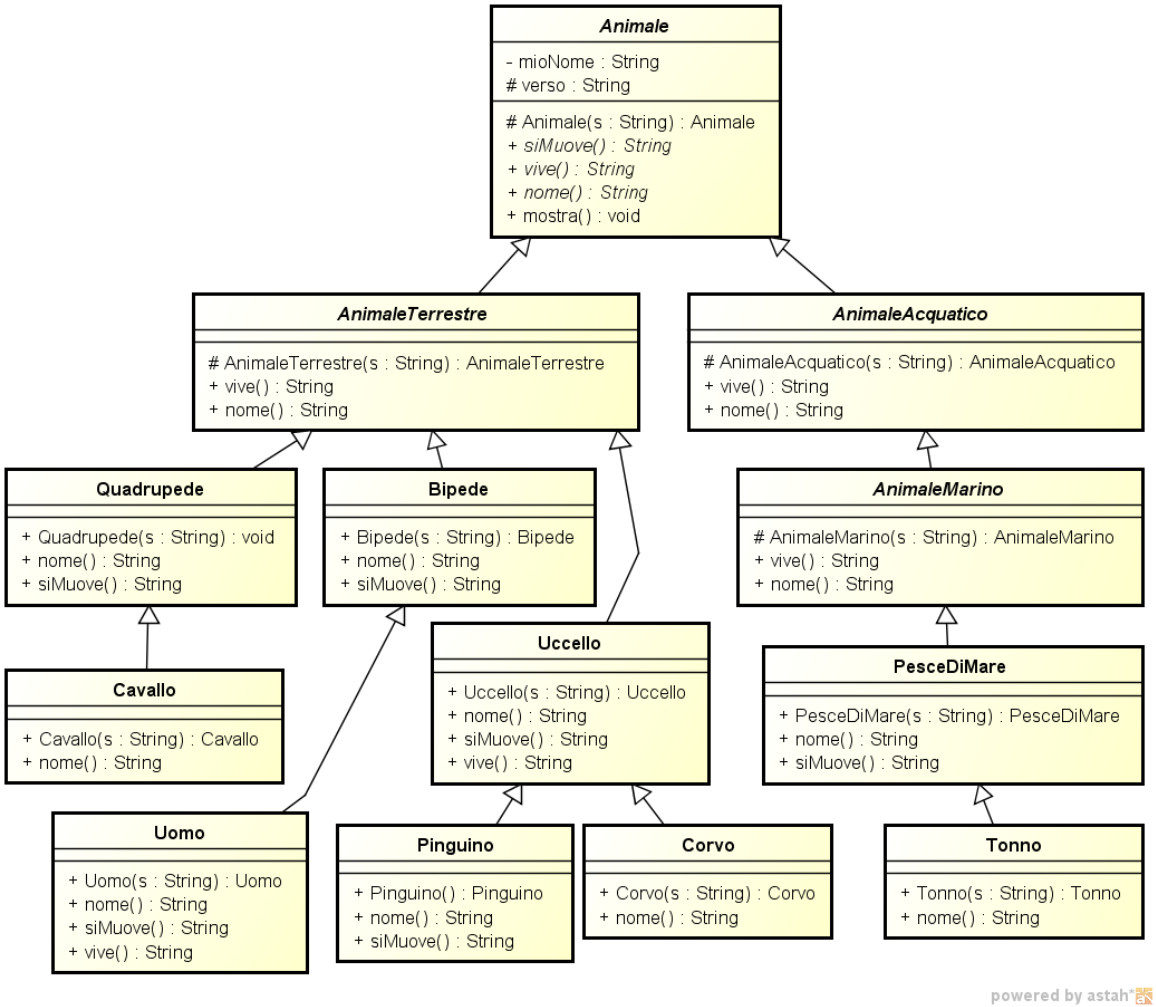
~Kotlin



Perché introdurre l'animale marino?

- non è correlato ad ambiente di vita o movimento: rispecchia semplicemente una realtà che ci interessa
- ridefinisce ulteriormente **`vive`** e **`nome`**, ma non **`siMuove`**: quindi, la classe è *ancora astratta*

UN ESEMPIO COMPLETO (9/9)



LE CLASSI CONCRETE (1/6)

Java

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

possibile inizializzare direttamente il verso, perché *protected* nella classe base

MA.. una proprietà che non c'è..? Progetto adeguato??

C#:
```
PesceDiMare(String s): base(s){
    verso = "non fa versi"; }
```

~C#

Java

~C#

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

Entrambe queste classi definiscono l'ultimo metodo astratto rimasto, **`siMuove()`**, e inizializzano la proprietà **`verso`**: perciò, non sono più astratte

LE CLASSI CONCRETE (2/6)

Scala

```
class PesceDiMare(mioNome:String) extends AnimaleMarino(mioNome){
  protected override val verso = "non fa versi";

  override def nome() : String = { return "un pesce (di mare)"; }
  override def siMuove() : String = { return "nuotando"; }
}
```

Per inizializzare il verso, si fa *override del dato astratto* definito nella classe base

Scala

```
class Uccello(mioNome:String) : AnimaleTerrestre(mioNome){
  protected override val verso = "cinguetta";

  override def vive() : String = { return "in un nido su un albero"; }
  override def nome() : String = { return "un uccello"; }
  override def siMuove() : String = { return "volando"; }
}
```

LE CLASSI CONCRETE (3/6)

Kotlin

```
public open class PesceDiMare(mioNome:String) : AnimaleMarino(mioNome){
 // In Kotlin, questa classe deve specificare open, altrimenti
 // non essendo abstract non sarebbe più estensibile

  protected override val verso = "non fa versi";
  public override fun nome() : String { return "un pesce (di mare)"; }
  public override fun siMuove() : String { return "nuotando"; }
}
```

Per inizializzare il verso, si fa *override del dato astratto* definito nella classe base

Kotlin

```
public open class Uccello(mioNome:String) : AnimaleTerrestre(mioNome){
  protected override val verso = "cinguetta";
  public override fun vive() : String {return "in un nido su un albero"; }
  public override fun nome() : String {return "un uccello"; }
  public override fun siMuove() : String { return "volando"; }
}
```

LE CLASSI CONCRETE (4/6)

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

Java

~C#

In Java e C# queste classi, pur non richiedendo più *formalmente* di essere astratte perché implementano l'ultimo metodo astratto rimasto, **`siMuove()`**, in realtà *non inizializzano la proprietà* ***`verso`***, che resta indefinitia: ciò le rende incoerenti e pericolose! MEGLIO SAREBBE DEFINIRLE *COMUNQUE ASTRATTE!*

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

Java

~C#

LE CLASSI CONCRETE (5/6)

Scala

```
abstract class Bipede(nome:String) extends AnimaleTerrestre(nome) {
    // A differenza di Java e C#, Bipede deve essere astratta
    // perché non inizializza il verso (giusto: era un design flaw!)
  override def vive() : String = { return "avanzando su 2 zampe"; }
  override def nome() : String = { return "un animale con 2 zampe"; }
}
```

Scala e Kotlin rimediano al problema, perché queste classi *devono* essere *astratte* in quanto *hanno ancora la proprietà astratta* ***verso***: si evita la pericolosità e si evidenzia il design flaw! ☺

Kotlin

```
public abstract class Bipede(nome:String) : AnimaleTerrestre(nome) {
    // A differenza di Java e C#, Bipede deve essere astratta
    // perché non inizializza il verso (giusto: era un design flaw!)
  public override fun vive() : String { return "avanzando su 2 zampe"; }
  public override fun nome() : String { return "un animale con 2 zampe"; }
}
```

LE CLASSI CONCRETE (6/6)

```
public class Cavallo extends Quadrupede {
  public Cavallo(String s) { super(s); verso = "nitrisce"; }
  public String nome() { return "un cavallo"; }
}
public class Corvo extends Uccello {
  public Corvo(String s) { super(s); verso = "gracchia"; }
  public String nome() { return "un corvo"; }
}
public class Tonno extends PesceDiMare {
  public Tonno(String s) { super(s); }
  public String nome() { return "un tonno"; }
}
public class Uomo extends Bipede {
  public Uomo(String s) { super(s); verso = "parla"; }
  public String siMuove() { return "camminando su 2 gambe"; }
  public String nome() { 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 nome() { return "un pinguino"; }
  public String siMuove() { return "camminando ma senza volare"; }
  public String vive() { return "sul ghiaccio"; }
}
```

Java

~C#

Queste classi inizializzano tutte il verso, quindi sono concrete

(NB: **`Tonno`** eredita l'inizializzazione di **`PesceDiMare`**)

~Scala

~Kotlin

UN “PICCOLO ZOO”

```
public class Zoo {
 public static void main(String[] args) {
  Animale[] zoo = {
    new Cavallo("Varenne"), new Uomo("John"),
    new Corvo("Crowy"),     new Tonno("TonTon"),
    new Uccello("Tweety"),  new Pinguino("Peng") };
  for(Animale a : zoo) a.mostra();
 }
}
```

Java

~C#

~Scala

~Kotlin

Scala, Kotlin: minime differenze nell’array

C#: `foreach(Animale a in zoo)`…

Scala: `for(a:Animale : zoo)`…

Kotlin: `for(a:Animale in zoo)`…

POLIMORFISMO: il metodo **`mostra()`** è unico, ma nonostante ciò si comporta in modo *specifico per ogni oggetto*, *montando in modo fisso parti specifiche*

```
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 condominio
Crowy, un corvo, gracchia, si muove volando e vive in un nido su un albero
TonTon, un tonno, non fa versi, si muove nuotando e vive in mare
Tweety, un uccello, cinguetta, si muove volando e vive in un nido su un albero
Peng, un pinguino, non fa versi, si muove camminando ma senza volare e vive sul ghiaccio
```

Un esempio più complesso: Forme geometriche

L'OBIETTIVO

Definire una tassonomia di *forme geometriche*

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

`Forma` può ben essere una classe astratta

Quali requisiti comuni?

- ogni “forma geometrica” possiede un'*area* e un *perimetro*
  (… e dei lati? o no? il cerchio…?)
- ogni “forma geometrica” deve potersi *mostrare*
- consideriamo forme geometriche *immodificabili*, ovvero entità definite e fissate in fase di costruzione → no rischi di classificazione

QUALE CLASSIFICAZIONE?

Se partizioniamo le forme per *numero di lati*:

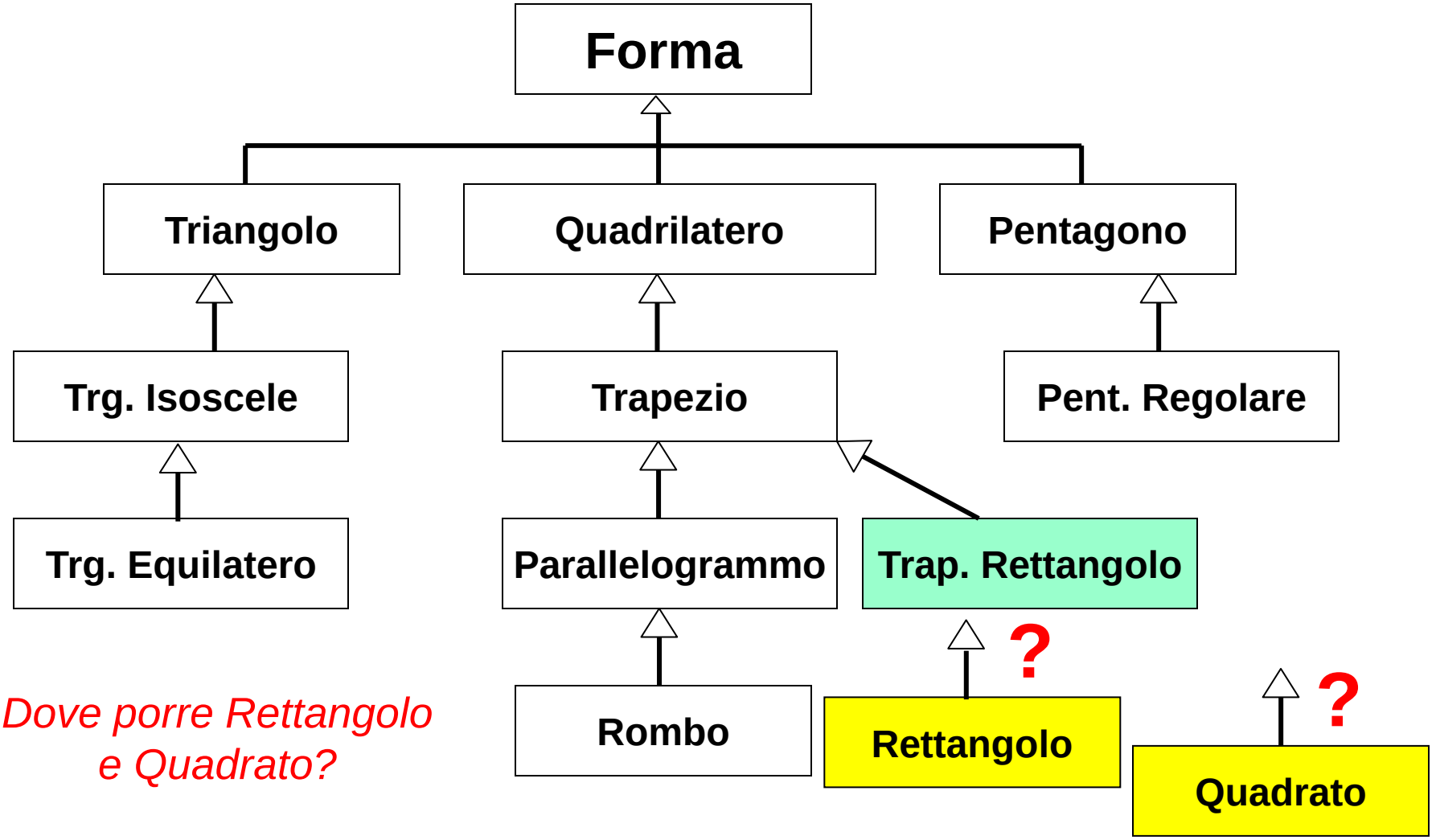
- triangoli
  - ulteriore criterio di classificazione: *lati uguali?*
    $\Rightarrow$ scaleno, isoscele, equilatero
    [dubbio: e i triangoli rettangoli..? mmhhh..]
- quadrilateri
  - quadrilatero qualsiasi
  - ulteriore criterio di classificazione: *lati paralleli?*
    $\Rightarrow$ trapezio, parallelogrammo, rombo
  - ulteriore criterio di classificazione: *angoli retti?*
    $\Rightarrow$ trapezio rettangolo, rettangolo, quadrato

CLASSIFICAZIONE: UNA IPOTESI

- triangoli
  - scaleno, isoscele, equilatero
  - non consideriamo l'eventuale angolo retto
    $\Rightarrow$ il concetto di "triangolo rettangolo" *non esiste in questo modello*
- quadrilateri
  - quadrilatero qualsiasi
  - criterio fondamentale: *lati paralleli*
    $\Rightarrow$ trapezio, parallelogrammo, rombo
  - [eventuale ulteriore criterio: *angoli retti*]
    $\Rightarrow$ trapezio rettangolo, rettangolo, quadrato

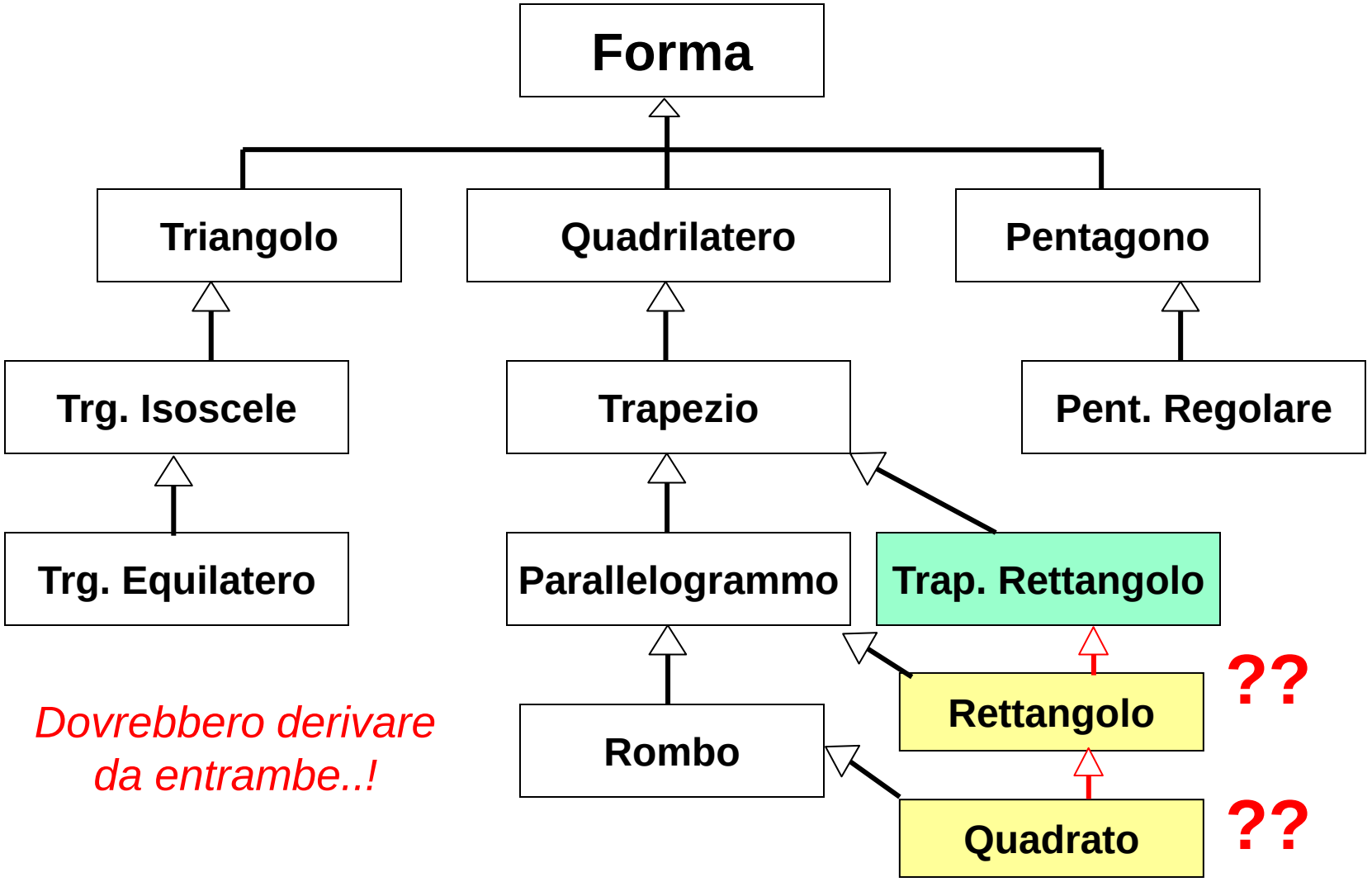
Come definire una tassonomia coerente?

UNA POSSIBILE TASSONOMIA



Dove porre Rettangolo e Quadrato?

UNA POSSIBILE TASSONOMIA

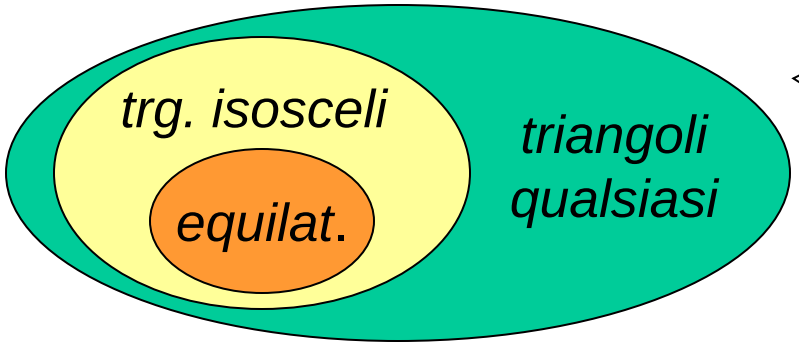


Dovrebbero derivare da entrambe..!

CLASSIFICAZIONE: IL PROBLEMA

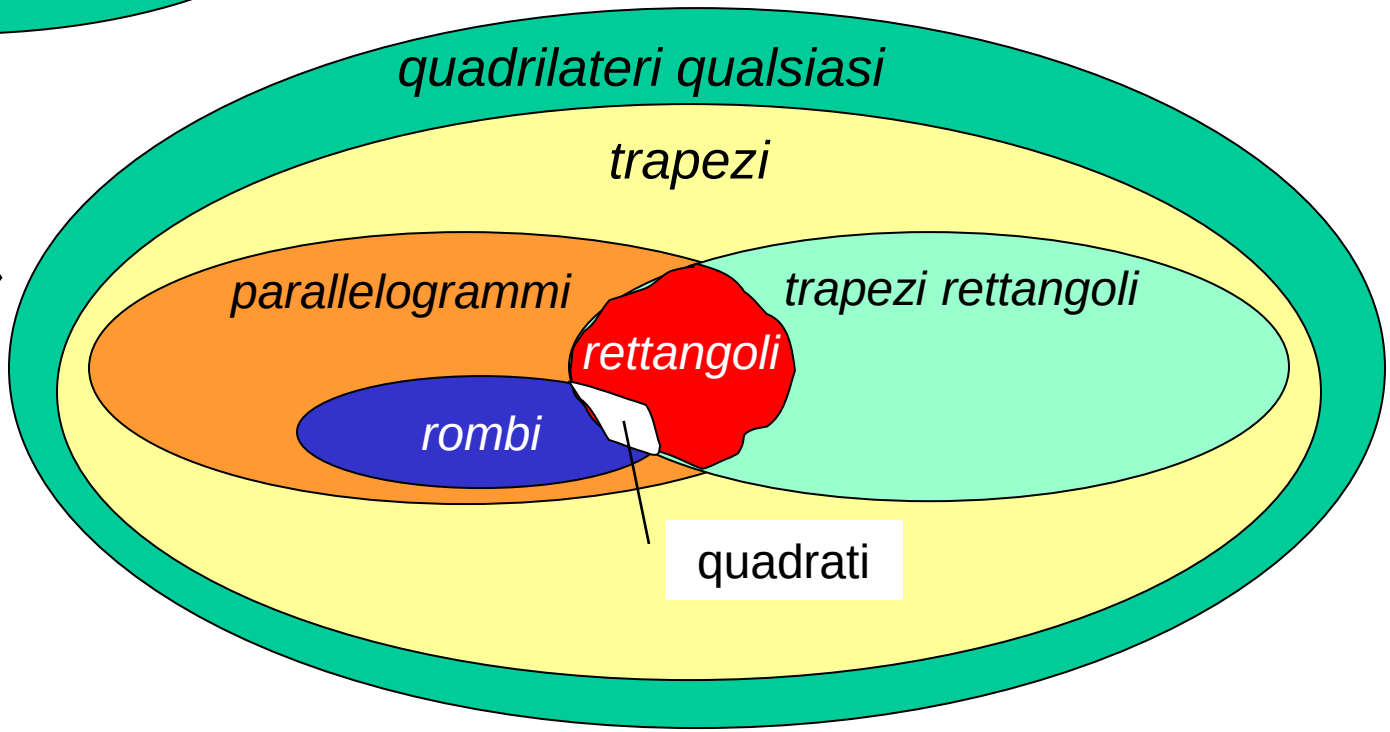
- L'ereditarietà come la conosciamo finora classifica *definendo sottoinsiemi*
- perciò, applicare *un solo criterio per volta*
  - perché un insieme non può essere sottoinsieme *di due insiemi diversi* (se uno non è già incluso nell'altro)
- nei triangoli, esiste una relazione di inclusione
  - perché abbiamo omesso i triangoli rettangoli: altrimenti..
- nei 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

RELAZIONI FRA INSIEMI



Fra le categorie di triangoli esiste una relazione di inclusione → *ben modellate* dall'ereditarietà singola *(omettendo i triangoli rettangoli…)*

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



LA CLASSE BASE (astratta)

Java

```
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…?

C#

```
public abstract class Forma {
  public abstract double Area();
  public abstract double Perimetro();
  public abstract String Nome();
   public override string ToString(){ return Nome() +
        " di area " + Area() + " e perimetro " + Perimetro();
   }
}
```

LA CLASSE BASE (astratta)

Scala

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

Kotlin

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

UNA CLASSE CONCRETA

Java

```
public class Triangolo extends Forma {
  private 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*?
Ricorda l'ipotesi: forme immodificabili…

C#

```
public class Triangolo : Forma {
  private double lato1, lato2, lato3;
  public override string Nome() { return "Triangolo qualsiasi"; }
  public override double Perimetro() { return lato1 + lato2 + lato3;
   }
  public override double Area() { /* formule trigonometriche */ }
  // altri metodi? magari per avere i lati?
}
```

UNA CLASSE CONCRETA

Scala

```
class Triangolo(val lato1:Double, val lato2:Double,
                val lato3:Double) extends Forma() {
  override def nome() : String = { return "Triangolo qualsiasi"; }
  override def perimetro() : Double = {
                                return lato1 + lato2 + lato3; }
  override def area() : Double = { /* formule trigonometriche */ }
  // altri metodi? magari per avere i lati?
}
```

Kotlin

```
public open class Triangolo(val lato1:Double,
   val lato2:Double, val lato3:Double) : Forma() {
  public override fun nome() : String {
                                return "Triangolo qualsiasi"; }
  public override fun perimetro() :Double {
                                return lato1 + lato2 + lato3; }
  public override fun area() : Double { /* formule trig. */ }
  // altri metodi? magari per avere i lati?
}
```

UN'ALTRA CLASSE CONCRETA

Java

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

Quali e quanti costruttori?
Con quali e quanti argomenti?
A cosa si appoggiano?

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

C#

```
public class TriangoloIsoscele : Triangolo {
   public override string Nome() { return "Triangolo isoscele"; }
   // Perimetro va già bene!!
   // Area va bene ma può convenire ridefinirla
   // nuovi metodi specifici? Base()? Lato()?
}
```

UN'ALTRA CLASSE CONCRETA

Scala

```
class TriangoloIsoscele(val base : Double, val lato : Double)
                        extends Triangolo(base,lato,lato) {
  override def nome() : String = { return "Triangolo isoscele"; }
}
```

Kotlin

```
public class TriangoloIsoscele(val base : Double, val lato : Double)
                        : Triangolo(base,lato,lato) {

 public constructor(base:Number, lato:Number)
                        : this(base.toDouble(), lato.toDouble());

 public override def nome() : String { return "Triangolo isoscele"; }
}
```

Costruttore ausiliario di cortesia, che accetta *ogni tipo di numeri* (=anche interi) e non solo Double: utile perché Kotlin è molto rigido sui tipi e in mancanza accetterebbe *solo* dei Double !

UN MONDO DI FORME

Java

```
public class TanteForme {
 public static void main(String[] args) {
  Forma[] forme = {
        new Triangolo(2,3,4), new TriangoloIsoscele(2,3),
        new TriangoloEquilatero(3),
        new Rettangolo(4,5), new Quadrato(6), … };
  for(Forma f : forme) System.out.println(f);
 }
}
```

C#

```
public class TanteForme {
 public static void Main(string[] args) {
  Forma[] forme = {
        new Triangolo(2,3,4), new TriangoloIsoscele(2,3),
        new TriangoloEquilatero(3),
        new Rettangolo(4,5), new Quadrato(6), … };
  for(Forma f in forme) Console.WriteLine(f);
 }
}
```

UN MONDO DI FORME

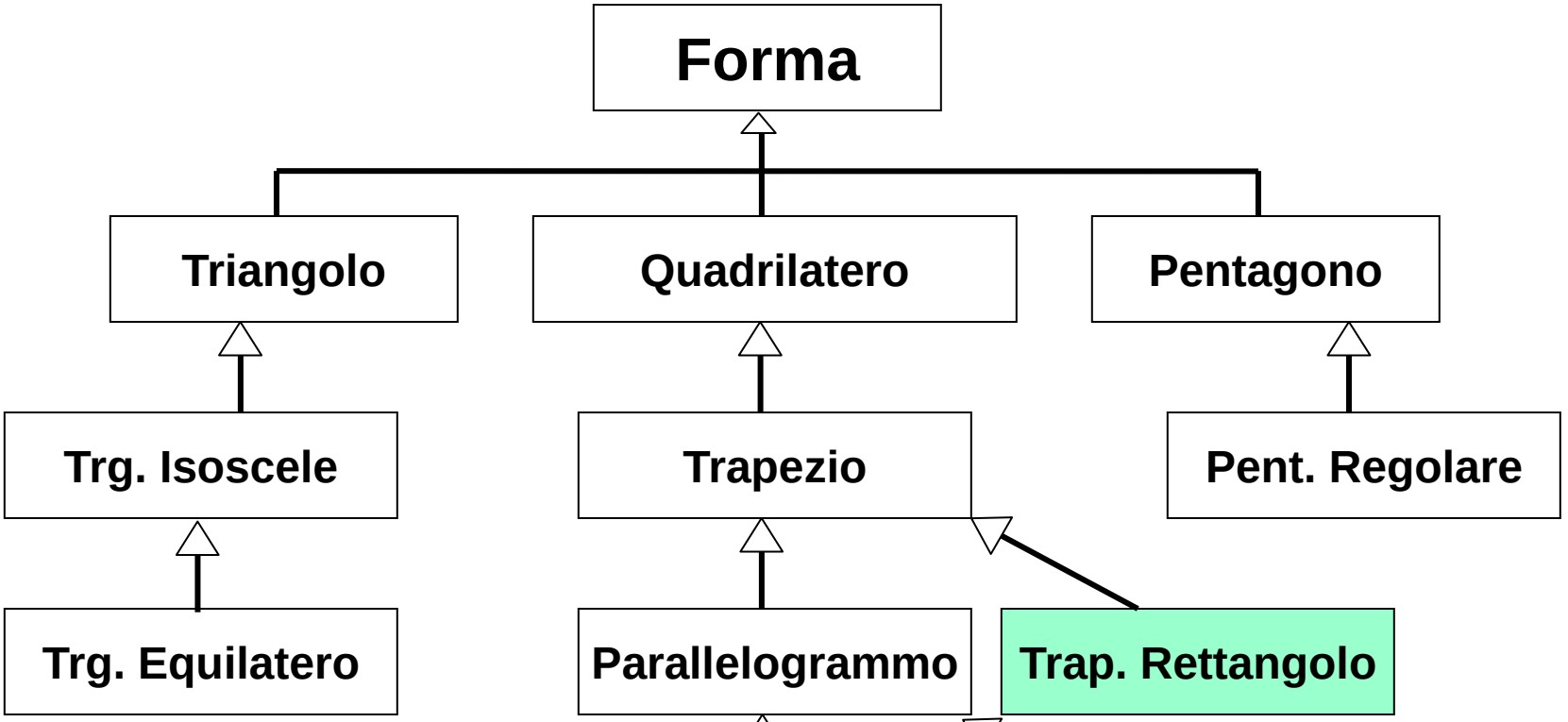
Scala

```
object TanteForme {
 def main(args: Array[String]) : Unit = {
  val forme = Array(
                new Triangolo(2,3,4),
                new TriangoloIsoscele(2,3),
                new TriangoloEquilatero(3), … );
  for(forma <- forme) println(forma);
 }
}
```

Kotlin

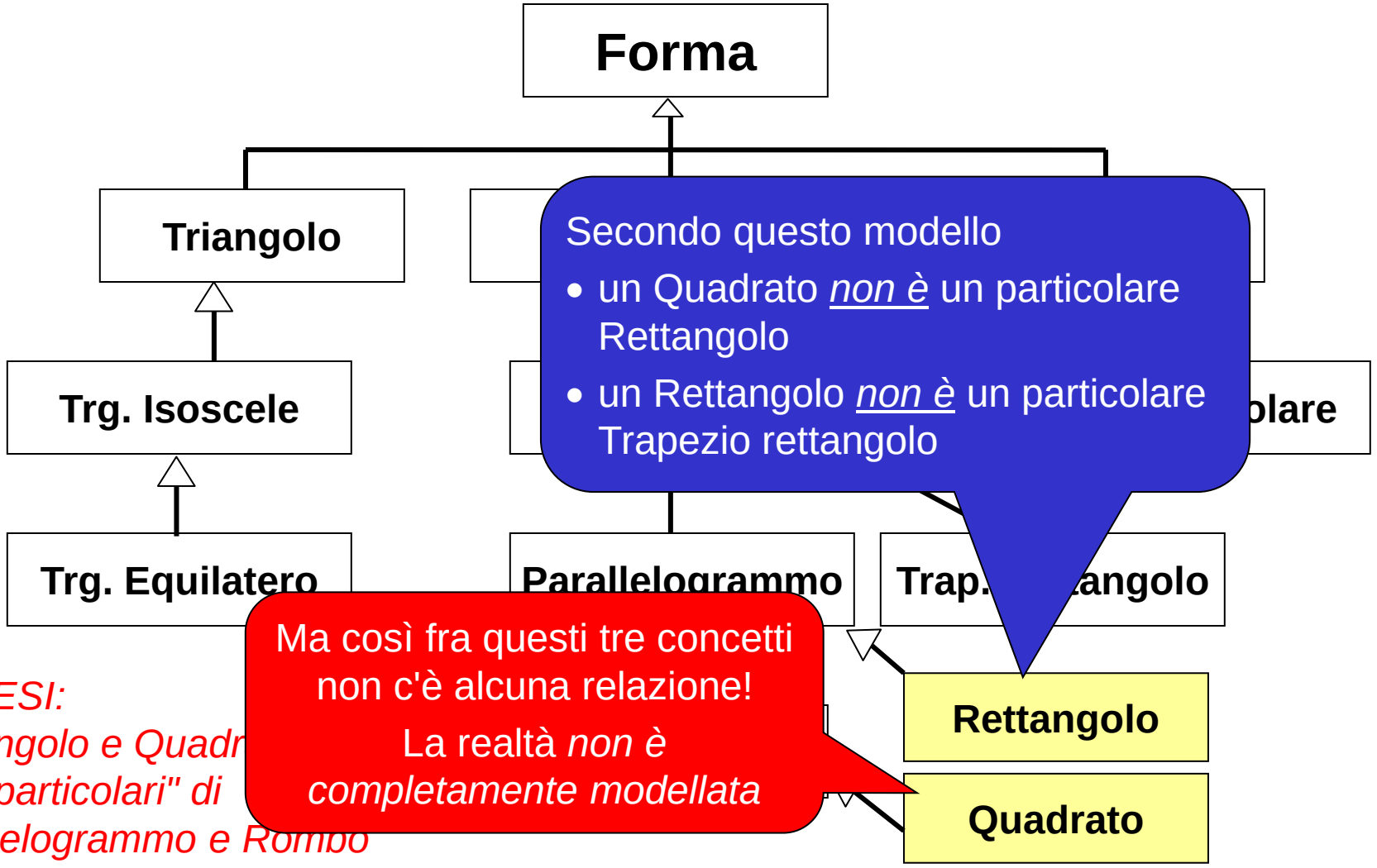
```
public fun main(args: Array<String>) : Unit {
 val forme = arrayOf(
                Triangolo(2,3,4), TriangoloIsoscele(2,3),
                TriangoloEquilatero(3), … );
 for(forma in forme) println(forma);
}
```

UNA POSSIBILE TASSONOMIA



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

UNA POSSIBILE TASSONOMIA



IPOTESI:
Rettangolo e Quadr
"casi particolari" di
Parallelogrammo e Rombo

LA QUESTIONE FONDAMENTALE

- Cosa comporta il fatto che
  - un Quadrato *non sia* un particolare Rettangolo?
  - un Rettangolo *non sia* un particolare Trapezio rettangolo?
- Non è solo un "fastidio": *è una violazione della realtà!*
  - tanto per cominciare, *non si può assegnare un quadrato a un* (riferimento a) *rettangolo, perché sono incompatibili!*
- In geometria, se una proprietà vale per i rettangoli, vale *a maggior ragione* anche per i quadrati
  - ergo, se quella proprietà/operazione è espressa/calcolata da una funzione/metodo, *ciò che vale per i rettangoli dovrebbe sempre essere applicabile anche ai quadrati*
    - invece, in questo modello, *una funzione che accetti un Rettangolo NON accetta un Quadrato!*

UN MONDO DI RETTANGOLI

```
public class RectWorld {
 public static void main(String args[]) {
  Rettangolo rett[] = new Rettangolo[10];
  rett[0] = new Rettangolo(...);
  rett[1] = new Quadrato(...);
 }
}
```

Java

~C#

~Scala

~Kotlin

In geometria avrebbe perfettamente senso,
MA nel nostro modello dà *errore di compilazione!*

OPERARE SUI RETTANGOLI

```
public class DrawLib {
 public static void disegna(Rettangolo r) {
   // disegna il rettangolo dato
 }
}
public class RectWorld {
 public static void main(String args[]) {
  DrawLib.disegna(new Quadrato(...));
 }
}
```

Java

~C#

~Scala

~Kotlin

Di nuovo, in geometria avrebbe perfettamente senso, MA nel nostro modello dà *errore di compilazione!*

ALTRE DOMANDE

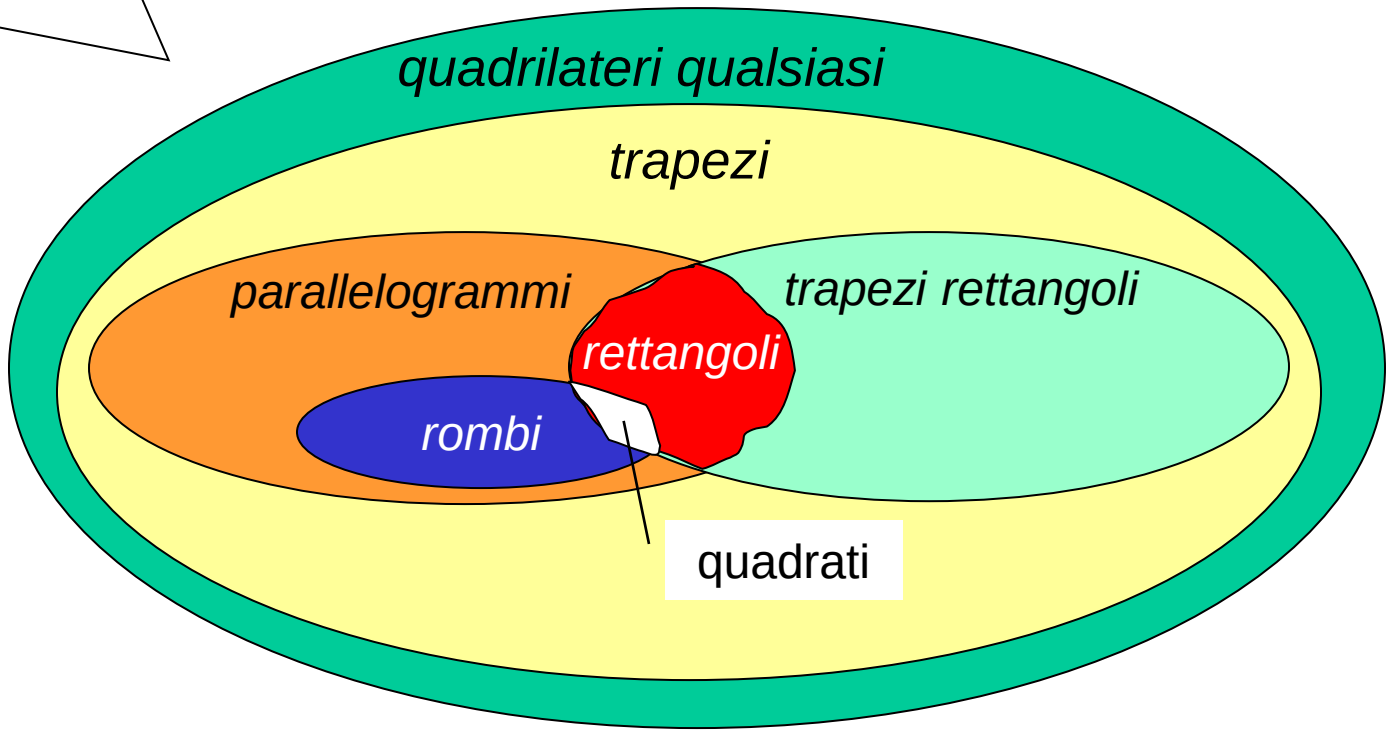
- Rappresentare il triangolo coi tre lati è la scelta migliore?
  - C'erano alternative?
- Il triangolo equilatero richiede aggiustamenti?
  - Quali e quanti costruttori ha?
  - Ha altri metodi particolari suoi propri?
- E se volessimo aggiungere i triangoli rettangoli ?
  - scaleni, isosceli, ma… mai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

In breve:

La tassonomia regge il confronto con la realtà?
Manca qualcosa alla nostra possibilità di esprimerci?

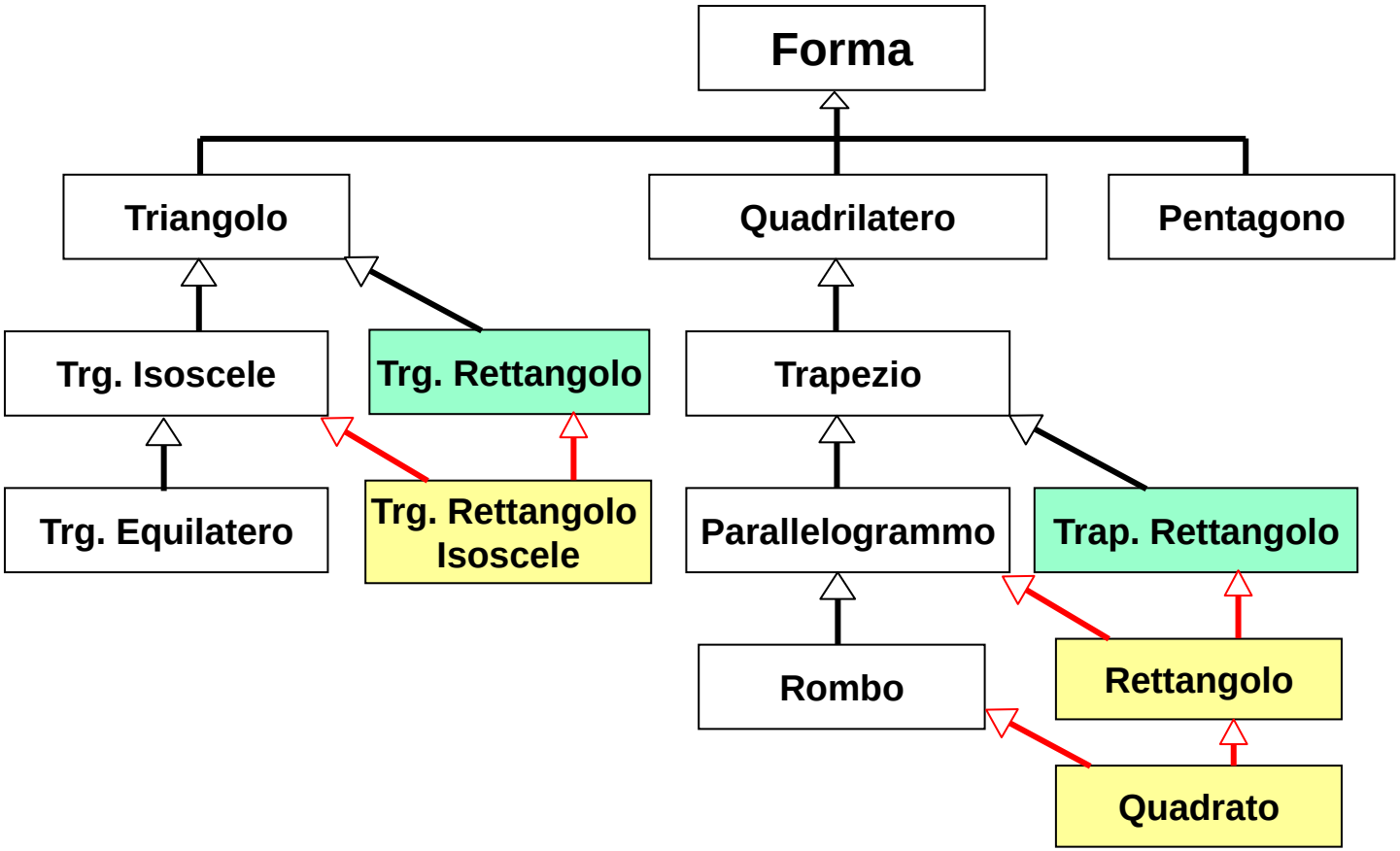
IL PUNTO

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



VERSO UN NUOVO OBIETTIVO

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



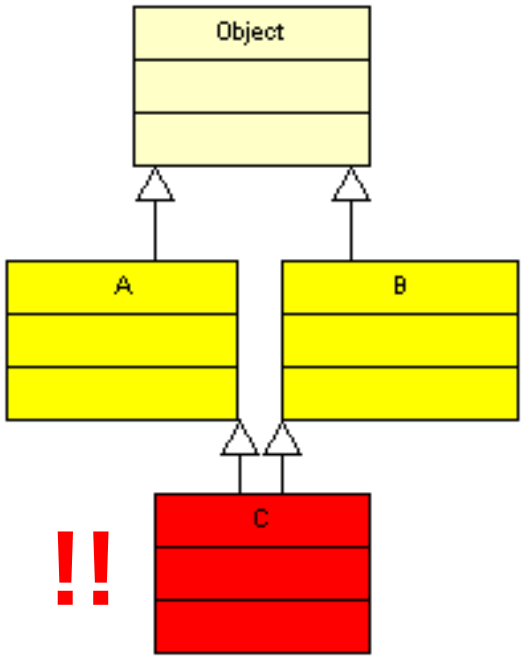
EREDITARIETÀ MULTIPLA…?

- 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 questo in Java e C# l'ereditarietà multipla non esiste *fra classi*
- Per esprimerla in modo pulito, evitando i problemi, viene introdotto il concetto di *interfaccia,* che separa:
  - l'idea dell'ereditarietà multipla in sé (giusta e utile)
  - dalla sua implementazione tramite classi (sbagliata)

… FRA CLASSI ?

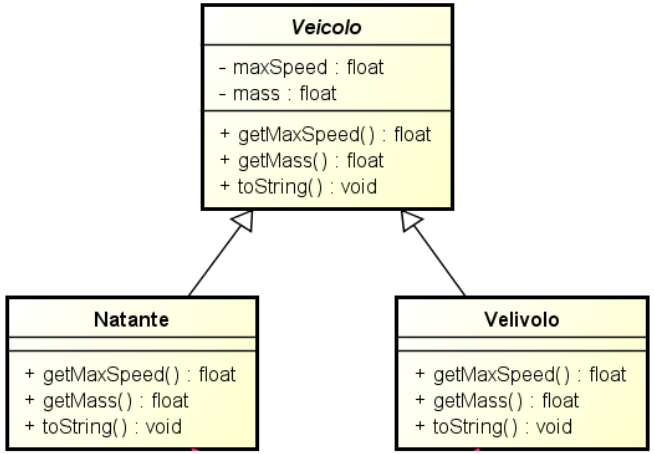
- Perché non ammettere l'ereditarietà *multipla ?*
- Perché nascono subito *problemi critici:*
  - la classe C unisce i *DATI* di A e di B
    - come si fa con le omonimie?
    - i dati della classe base sono replicati?
  - la classe C unisce *i METODI* di A e B
    - che si fa con definizioni replicate?
    - cosa si eredita nelle sottoclassi?

Il linguaggio C++ la consentiva, ma l'esperienza non è stata del tutto positiva.



PERCHÉ NO (1)

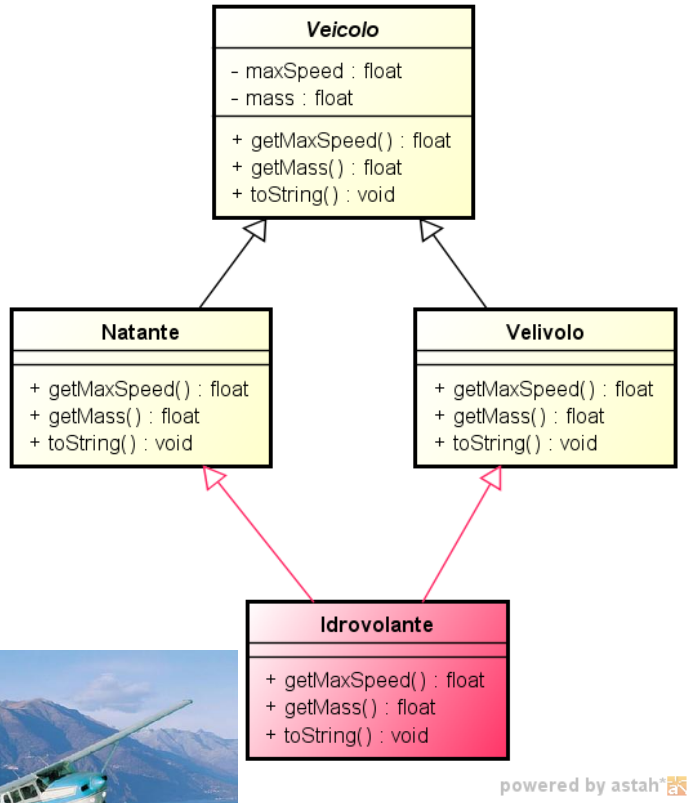
- ESEMPIO: *tassonomia di veicoli*
  - non esiste il "generico veicolo": è un'astrazione
  - dunque, è ragionevole che **`Veicolo`** sia una *classe astratta*
    - proprietà: `mass` (massa), `maxSpeed` (velocità max)
    - metodi: `getMass`, `getMaxSpeed`, `toString`
  - sottoclassi concrete: **`Velivolo`** e **`Natante`**
    - nessun problema
    - ogni velivolo ha una massa e una velocità max
    - ogni natante ha una massa e una velocità max



PERCHÉ NO (2)

- MA.. che succede con l'idrovolante?
  - È sia Velivolo sia Natante:
    dovrebbe *ereditare da entrambe*
    → ereditarietà multipla

- Problema: cosa eredita?
  - quante copie dei dati ereditati ha?
    - ha una velocità max, o due?
    - ha una massa, o due?
  - quanti versioni di metodi ha?
    - se ne ha due, cosa fanno?
      se ne ha uno, qual è?
    - quando si chiama un metodo,
      quale viene eseguito?



QUANTE COPIE/VERSIONI DI DATI E METODI EREDITARE?

- La risposta varia da caso a caso
  - ad esempio, l'idrovolante, come ogni oggetto fisico (e ogni veicolo), ha una massa sola, che parrebbe giusto *ereditare in copia singola*
  - ma al contempo ha due velocità massime (quando vola e quando galleggia), che quindi sarebbe corretto *ereditare entrambe*
- MORALE: dipende dalla semantica di ciascun dato
  - C++ permetteva di specificare se duplicare o no ogni elemento.. .. *ma obbligava a specificarlo nella classe base*
  - peccato che al momento di progettare Veicolo, l'idrovolante potesse essere del tutto imprevisto! Così cade la progettazione incrementale
- Per questo è stata scelta una strada diversa.

UN NUOVO APPROCCIO

- Java e derivati prendono atto che l'ereditarietà multipla è un formidabile strumento concettuale per unire *astrazioni,* non *implementazioni*
  - *astrazione* e *realizzazione* sono due *piani diversi*
  - *unire astrazioni è utile* per esprimere *intersezioni di insiemi*, una situazione presente e frequente nel mondo reale
  - *unire implementazioni è invece causa di problemi*, perché le cose diventano contorte e presto incomprensibili.

Nel costrutto **`class`** *astrazione e implementazione viaggiano sempre insieme: è la sua caratteristica.. e il suo limite*

Le problematiche legate all'ereditarietà multipla suggeriscono di *separarle più nettamente*. Per questo servirà un nuovo costrutto: *l'interfaccia.*