Dipartimento di Informatica e Sistemistica
Antonio Ruberti

“Sapienza” Università di Roma

Dalle Classi Alle Interfacce

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

CLASSI: LIMITI

- ***Definire*** **una classe implica essere in grado di specificare *i dettagli* del suo *funzionamento***
  - ossia, non soltanto dire QUALI operazioni ci sono
  - ma anche dire COME esse siano fatte DENTRO

Due problemi:

- **a volte *non si è in grado* di dare tali dettagli**
- **manca il supporto *all'ereditarietà multipla***

IL PRIMO PROBLEMA

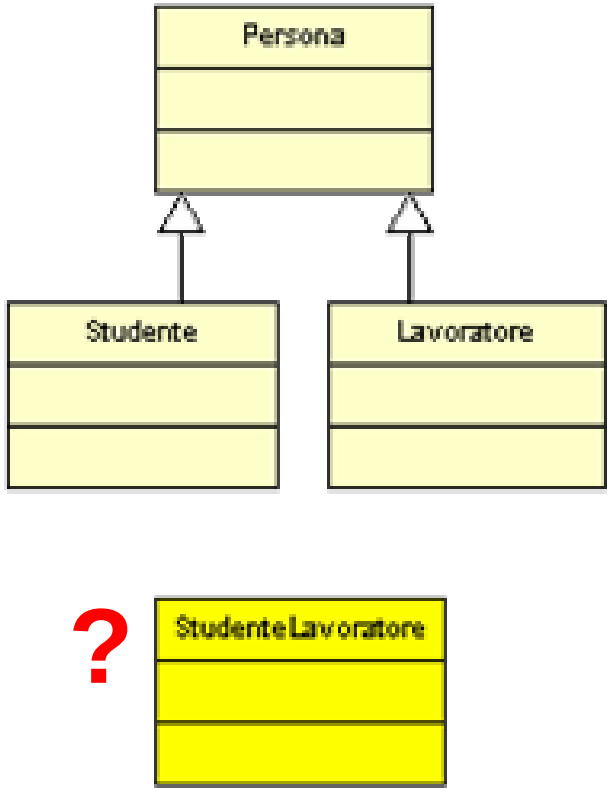
- **Se non si è in grado di specificare tutti *i dettagli* del *funzionamento* di una classe, si può definire una classe astratta…**
  - permette proprio di dire QUALI operazioni ci sono
  - ma senza doverle implementare
- **…però, *ciò non è del tutto soddisfacente***
  - **introduce un vincolo: per implementare le operazioni lasciate in bianco, bisogna essere *una sottoclasse..*** ma ciò potrebbe essere *impossibile*, se la classe doveva stare altrove nella gerarchia!
  - **È una soluzione troppo legata all'ereditarietà,** che oltre tutto in Java è solo singola!

IL SECONDO PROBLEMA

In Java, una classe può ereditare da *una sola* superclasse, ma ciò può creare *seri problemi nel modello della realtà:*

- **se `Studente` e `Lavoratore` estendono `Persona`, *dove si colloca `StudenteLavoratore`?***
- **idem per le forme geometriche (`Rettangolo`, `Quadrato`,...)**

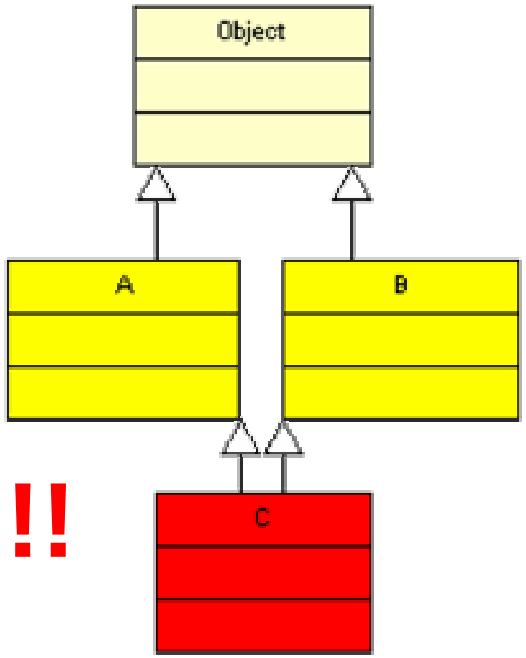
Perché non ammettere l'ereditarietà *multipla*?



EREDITARIETÀ MULTIPLA FRA CLASSI?

Se si ammette l'ereditarietà *multipla fra classi*, come in C++, **nascono *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?**



!!

***Per questi motivi,* l'ereditarietà multipla *fra classi* crea più problemi di quanti ne risolva.**

Occorre *salvaguardare l'idea* ma *in modo più pulito.*

OBIETTIVI

1°OBIETTIVO:

SEPARARE la fase in cui si specifica la *vista esterna* di un componente software dalla fase in cui si dettaglia la sua *realizzazione interna,* MA senza dipendere dall'ereditarietà come succede con le classi astratte.

2°OBIETTIVO:

SUPERARE il meccanismo di ereditarietà singola, che *non permette di comporre funzionalità di classi diverse,* rendendo quindi *molto complesso* o a volte *impossibile* definire la "giusta" tassonomia di classi (cf. forme geometriche) MA evitando meccanismi poco chiari.

Interfacce in Java

INTERFACCE

- **L'INTERFACCIA è un *nuovo costrutto,* simile alla classe astratta**
  - **infatti, dichiara i metodi senza implementarli**
- **ma slegato dalla *gerarchia di ereditarietà (singola)* delle classi**
  - **di cui, quindi, non subisce i vincoli**
- **e perciò in grado di supportare *ereditarietà multipla senza introdurre problemi e criticità***
  - **si prende l'idea, ma la si adatta / perfeziona in modo da evitare le problematiche**

IL CONCETTO DI INTERFACCIA

- **Un' INTERFACCIA dichiara metodi** e **costanti**

MA

- **non definisce *né variabili* né *metodi***
  ossia *non implementa assolutamente niente !*

- **Non avendo implementazioni**, **elimina alla radice il rischio di collisione fra *METODI* o DATI omonimi**
  → ***può supportare l'ereditarietà multipla***
  in modo pulito e senza rischi

- **È analoga a una classe astratta, ma non mischia l'idea di *fornire una specifica di metodi* con l'ereditarietà**
  → non obbliga chi realizza le operazioni a ereditare da essa

IL COSTRUTTO `interface`

- **Una *interfaccia* è introdotta dalla parola chiave `interface` anziché `class`**
- **Contiene solo *dichiarazioni* di metodi** (ed eventualmente costanti) **ma *nessuna implementazione***

```
public interface Comparable {
  public int compareTo(Object x);
}
```

- **Come per le classi, in Java il *NOME dell' interfaccia* deve *coincidere col nome del file `.java`***
- **Le *interfacce sono SEMPRE PUBBLICHE* e dichiarano sempre e solo *metodi PUBBLICI* e *costanti PUBBLICHE***

INTERFACCE vs. CLASSI ASTRATTE

Attraverso interfacce, si definiscono astrazioni di dato in termini di *comportamento osservabile:*

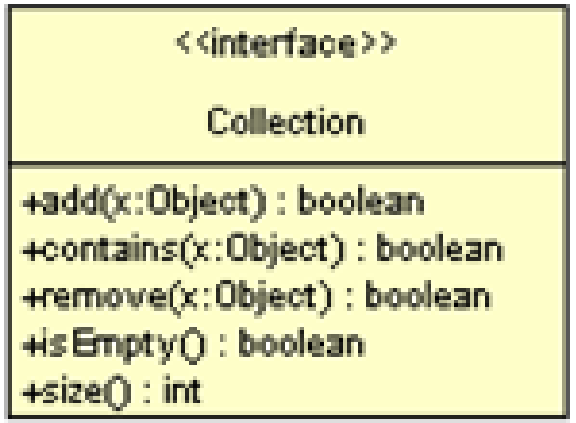
- **si specifica *"cosa ci si aspetta"* che entità con quel nome sappiano fare…**
- **.. MA si rinvia ad altri la loro *realizzazione pratica***

La differenza con le classi astratte è che

- **nel caso delle classi astratte, l'implementazione dei metodi "lasciati in bianco" dovrà essere fornita *da una sottoclasse***
- **nel caso delle interfacce, invece, tale implementazione potrà essere fornita *da una classe qualunque*.**

ESEMPIO: COLLEZIONI

```
public interface Collection {
 public boolean add(Object x);
 public boolean contains(Object x);
 public boolean remove(Object x);
 public boolean isEmpty();
 public int size();
 ...
}
```



Definisce il concetto di Collection come *qualunque entità avente queste cinque proprietà,* *indipendentemente da quali classi la realizzeranno*

Qualunque classe che affermi di implementare Collection dovrà garantire di fornire questi cinque metodi.

DALL'ASTRAZIONE ALL'IMPLEMENTAZIONE

Una interfaccia definisce un *comportamento osservabile,* ma *non implementa nulla.*

Qualcuno dovrà prima o poi *implementare la astrazione* definita dall'interfaccia.

A questo fine, una *classe* può *implementare (una o più) interfacce*

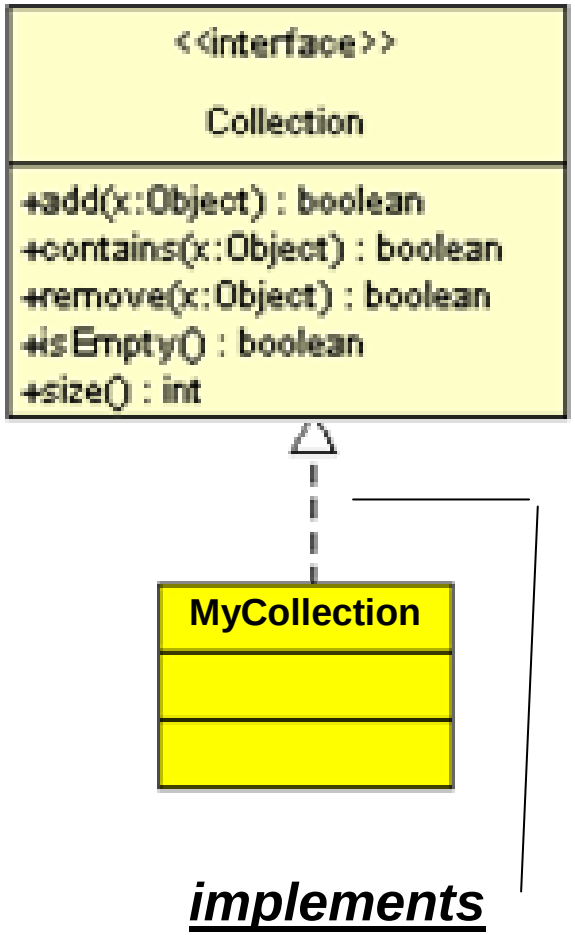
- **le interfacce specificano le *dichiarazioni* dei metodi**
- **la classe *definisce (implementa)* tutti i metodi delle interfacce che si impegna a implementare**
- **Nasce dunque la *relazione "IMPLEMENTS"***

CLASSI CHE IMPLEMENTANO INTERFACCE

```
public class MyCollection
     implements Collection {
...
}
```

Poiché la classe `MyCollection` afferma di implementare l'interfaccia `Collection`, la classe DEVE definire TUTTI i metodi esposti dall'interfaccia.

Se non lo fa, il compilatore dà ERRORE.



ESEMPIO: UN COUNTER "comparable"

Per rendere comparabile (confrontabile) un Counter con altri `Counter`, occorre:

- **riportare nell'intestazione della classe la dichiarazione `implements Comparable`**
  - **è un'interfaccia che dichiara il metodo**
    ```
    public int compareTo(Object x);
    ```
- **implementare un metodo pubblico avente esattamente la signature**
  ```
  public int compareTo(Object x);
  ```
  specificata dall'interfaccia.

IL COUNTER “Comparable"

```
public class Counter implements Comparable {
   ...
   public int compareTo(Object x) {
      Counter otherCounter = (Counter)x;
      if (val < otherCounter.val) return -1;
      if (val > otherCounter.val) return +1;
      /* else */ return 0;
   }
}
```

Formalmente è un `Object`, in realtà dev'essere un `Counter` per poter confrontare il valore

Deve rispettare la semantica che accompagna **`Comparable`**:

- **`compareTo`** deve restituire 0 se i due oggetti sono uguali
- **`compareTo`** deve restituire -1 se l'oggetto corrente precede quello passato come parametro
- **`compareTo`** deve restituire +1 se vale viceversa

UN MAIN per confrontare Counter

```
public class Test {
  public static void main(String args[]) {
   Counter c1 = new Counter(10);
   Counter c2 = new Counter(10);
   System.out.println("c1 vs c2: " +
                      c1.compareTo(c2)); // dà 0
   c1.inc();
   System.out.println("c1 vs c2: " +
                      c1.compareTo(c2)); // dà +1
   System.out.println("c2 vs c1: " +
                      c2.compareTo(c1)); // dà -1
}
```

Che succede se l'oggetto passato non è un `Counter` ?

PERCHÉ un Counter “Comparable”?

- **Per sfruttare la proprietà di confronto!**
- **Il package `java.util` offre una classe `Arrays` che contiene funzioni di utilità sugli array**
  - **ricerca binaria → `binarySearch`**
  - **rimpimento di valori → `fill`**
  - **ordinamento con QuickSort → `sort`**
- **La funzione `sort` permette proprio di ordinare un array di oggetti *comparabili*, ossia che implementino tutti l'interfaccia `Comparable`**
- **Quindi, ora possiamo ordinare gratis anche array di Counter!**

ORDINARE UN ARRAY DI COUNTER

```
public class Test {
  public static void main(String args[]) {
   Counter[] myCounterArray = new Counter[4];
   myCounterArray[0] = new Counter(11);
   myCounterArray[1] = new Counter(10);
   myCounterArray[2] = new Counter(3);
   myCounterArray[3] = new Counter(5);
   java.util.Arrays.sort(myCounterArray);
   for(int k=0; k<myCounterArray.length;k++)
     System.out.println(myCounterArray[k]);
  }
}
```

```
Counter di valore 3
Counter di valore 5
Counter di valore 10
Counter di valore 11
```

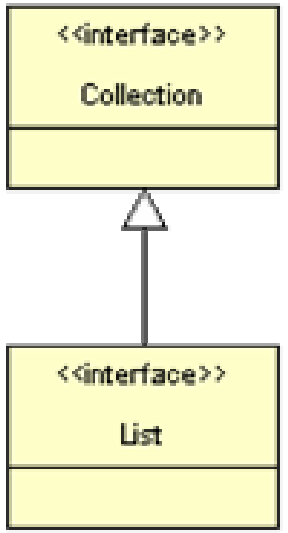
TASSONOMIE DI INTERFACCE

Le interfacce possono dare luogo a *gerarchie di ereditarietà , proprio come le classi:*

```
public interface List extends Collection {
  ...
}
```

Come in ogni gerarchia, anche qui le interfacce derivate:

- **possono aggiungere nuove *dichiarazioni di metodi***
- **possono aggiungere nuove *costanti***
- ***non possono* eliminare nulla**



TASSONOMIE DI INTERFACCE vs. TASSONOMIE DI CLASSI

La tassonomia delle interfacce è *separata* da quella delle classi che le implementano.

Come tale:

- **è *slegata* dagli aspetti implementativi** (che riguardano solo le classi)
- ***non deve per forza assomigliare alla tassonomia delle classi*** – anzi, può essere anche molto diversa
- **esprime le *relazioni concettuali* fra le entità esistenti nella realtà da modellare**
- **con ciò, *guida il progetto.***

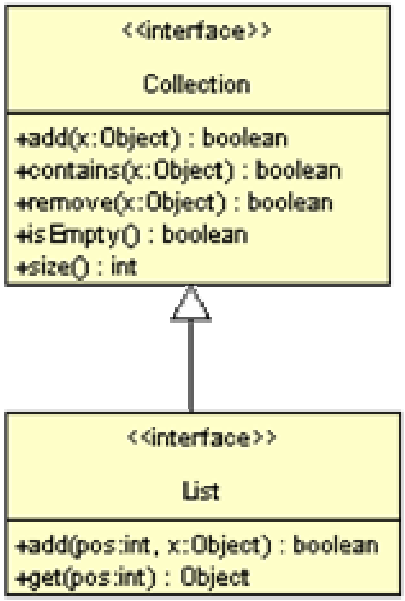
TASSONOMIE DI INTERFACCE: ESEMPIO

L'interfaccia `List` deriva da `Collection`

Significato: ***"Ogni lista è anche una collezione"***

- **ogni lista può interagire col mondo come una collezione** *(magari in modo specializzato)..*
- **ma** ***in più ha un concetto di sequenza*** (esiste un *I°elemento* , un *II°* ,etc.) **che porta a dichiarare due nuovi metodi:**

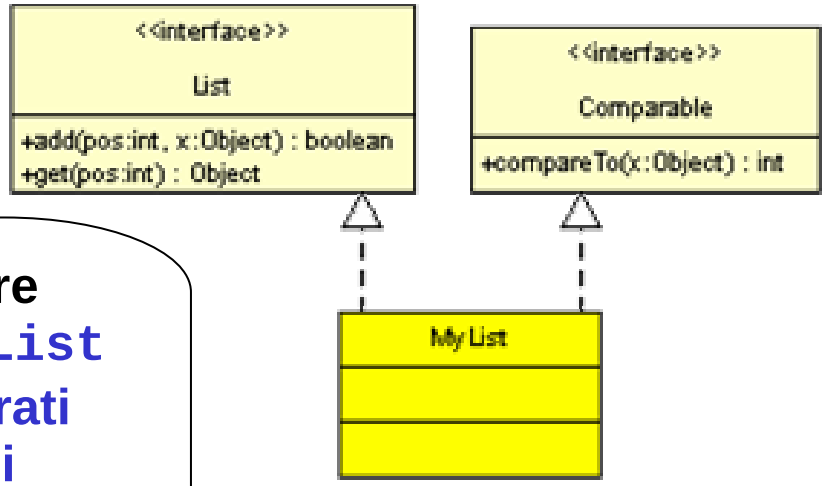
```
public interface List extends Collection {
 public boolean add(int pos, Object x);
 public Object get(int pos);
 ...
}
```



CLASSI CHE IMPLEMENTANO INTERFACCE

Una classe può implementare *più interfacce*:

```
public class myList
   implements List, Comparable {
...
}
```



Essendosi impegnata a implementare entrambe le interfacce, la classe `MyList` DEVE definire TUTTI i metodi dichiarati dall'interfaccia `List` più TUTTI quelli dichiarati dall'interfaccia `Comparable`.

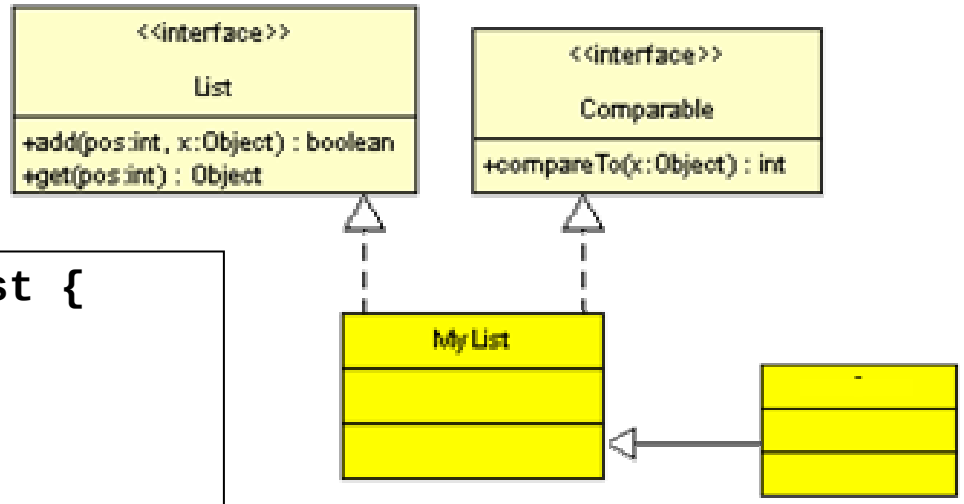
Se non lo fa, ERRORE.

CLASSI CHE IMPLEMENTANO INTERFACCE

Se una classe implementa una interfaccia, *anche tutte le sue sottoclassi automaticamente la implementano*

Non è necessario ripetere *implements*

```
class MyList2 extends MyList {
// SOTTINTESO implements
   List,Comparable
...
}
```



RIFERIMENTI a INTERFACCE

- **Il nome di una interfaccia può essere usato come *identificatore di tipo* per *riferimenti* e *parametri formali* di metodi e funzioni.**
- **Tuttavia, poiché un'interfaccia non è una classe, non può essere istanziata → quali oggetti??**
- **Gli oggetti referenziati devono essere istanze di *classi che implementano l'interfaccia***

Nomi di interfacce

Esempi

```
Collection c = new ArrayList();
List l1 = new LinkedList();
List l2 = new LinkedList(c);
```

Nomi di classi che implementano le interfacce

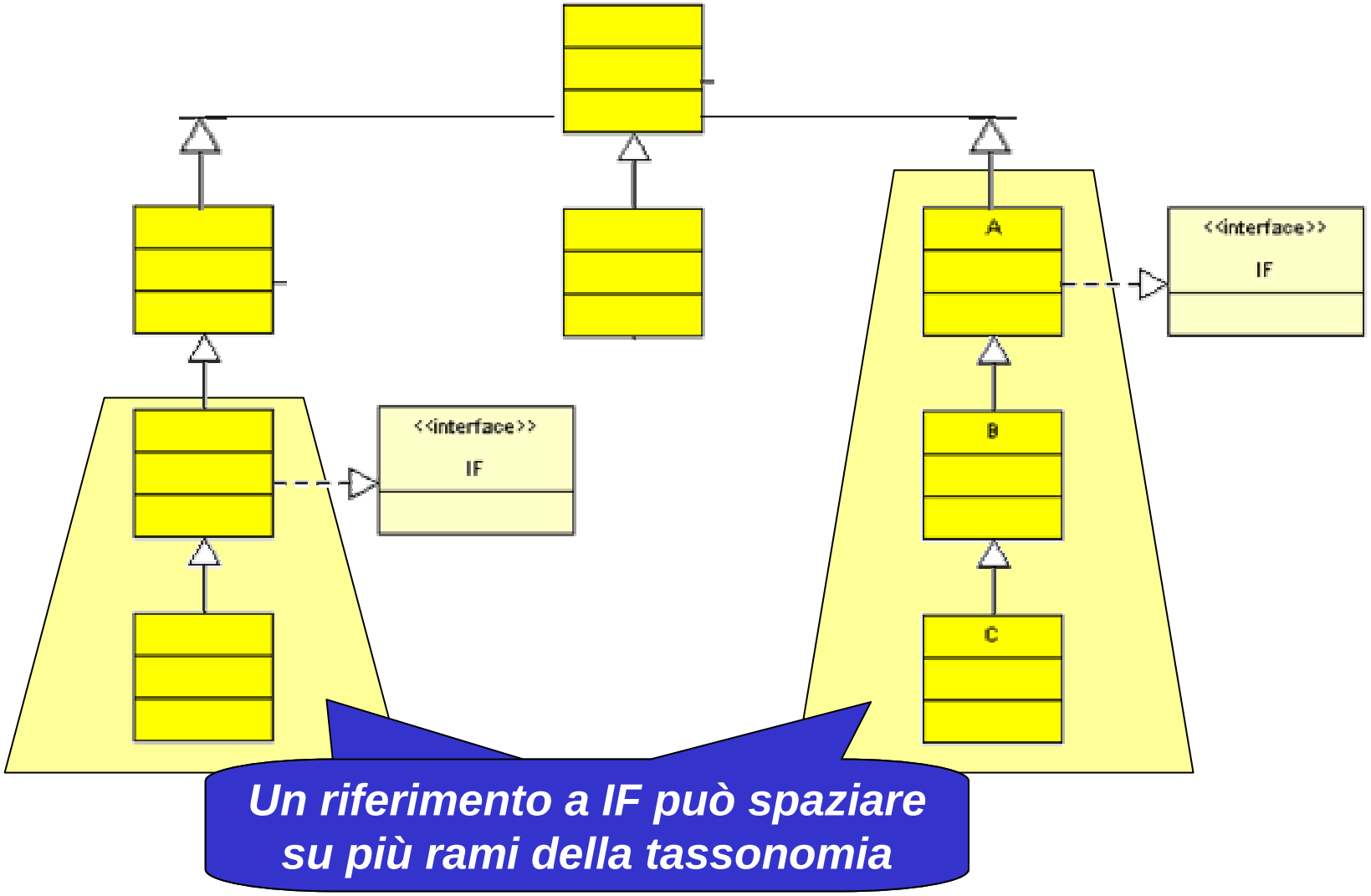
RIFERIMENTI a INTERFACCE: USO

I riferimenti a interfaccia sono *più flessibili* dei riferimenti a classi

- **un riferimento a classe può puntare a un oggetto di quella classe o di una sua sottoclasse**
  → è limitato a *quel ramo* della tassonomia

- **un riferimento a interfaccia può puntare a un oggetto *di una qualunque classe che implementi l'interfaccia***
  → può spaziare su *più rami* della tassonomia

RIFERIMENTI a INTERFACCE e TASSONOMIE di CLASSI



INTERFACCE E PROGETTO

Le interfacce inducono un diverso
modo di concepire il progetto

- **spesso, conviene prima definire le interfacce**
  - **la gerarchia delle interfacce riflette *scelte di progetto (pulizia concettuale)***
- **poi si realizzano le classi che le implementano**
  - **la gerarchia delle classi riflette tipicamente *scelte implementative (efficienza ed efficacia)***

INTERFACCE VUOTE: il pattern MARKER

Il ruolo delle interfacce *è così essenziale* da portare a definire, a volte, *interfacce vuote*

- **non dichiarano nuove funzionalità**
- **fungono da marcatori *(marker)* per le classi che asseriscono di implementarle**
- **La loro utilità sta proprio nel mettere a disposizione un *nome* usabile come *tipo***
  - **esempi: `Cloneable`, `Serializable`, etc**
  - **ogni classe che le implementa *"afferma"* di essere *clonabile*, *serializzabile*, etc**
  - **è una forma di auto-documentazione, che *sfrutta il compilatore per trovare incongruenze***

ESEMPIO: un COUNTER "clonabile"

Per essere clonabile, il contatore deve:

- **implementare un metodo pubblico `clone()` appoggiandosi sul metodo protetto `clone()` ereditato da `Object` → si usa `super`**
- **riportare nell'intestazione della classe la dichiarazione `implements Cloneable`**
  - **non comporta di dover scrivere alcun codice!**
  - **è però indispensabile farlo, perché *l'interprete Java rifiuterà di clonare oggetti che non siano "clonabili"*, scatenando un errore a run-time**

Usa il compilatore per scovare eventuali incongruenze

IL COUNTER "clonabile"

```
public class Counter implements Cloneable {
  protected int val;
  public Counter() { val = 1; }
  public Counter(int v) { val = v; }
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() { return val;}
  public Object clone()
      throws CloneNotSupportedException {
            return super.clone(); }
}
```

In pratica, si limita a "rendere pubblico" il servizio già fornito dalla classe **`Object`**.

IL COUNTER "clonabile": UN MAIN

```
public class Test {
  public static void main(String args[])
       throws CloneNotSupportedException {
   Counter c1 = new Counter(10);
   Object obj = c1.clone();
   Counter c2 = (Counter)obj; // cast
   c1.inc();
   System.out.println("c1 = " + c1.getValue());
   System.out.println("c2 = " + c2.getValue());
 }
}
```

c1 viene incrementato e diventa 11, ma c2 rimane fermo a 10 → sono oggetti distinti

CONTRO-TEST

```
public class Counter {
  ...
  public Object clone()
      throws CloneNotSupportedException {
          return super.clone(); }
}
```

se si toglie dal Counter la frase **implements Cloneable** …

… lo **stesso main** di poco fa **scatena ora un errore:**

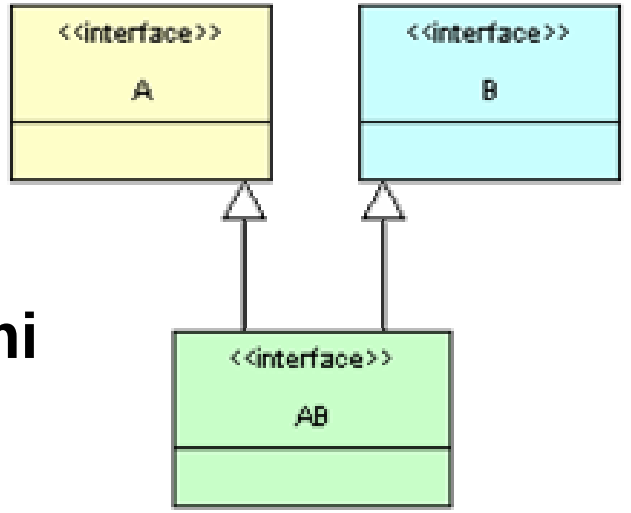
```
---------- Java Run ----------
java.lang.CloneNotSupportedException: Counter
      at java.lang.Object.clone(Native Method)
      at Counter.clone(Test.java:19)
      at Test.main(Test.java:4)
```

Interfacce ed ereditarietà multipla

EREDITARIETÀ MULTIPLA

Java la supporta fra interfacce

- **una interfaccia contiene *solo dichiarazioni* di metodi**
- **non ha implementazioni**
  → nessun problema di collisione fra *METODI* omonimi
- **non ha variabili**
  → nessun problema di collisione fra *DATI* omonimi



È un potente strumento di modellazione

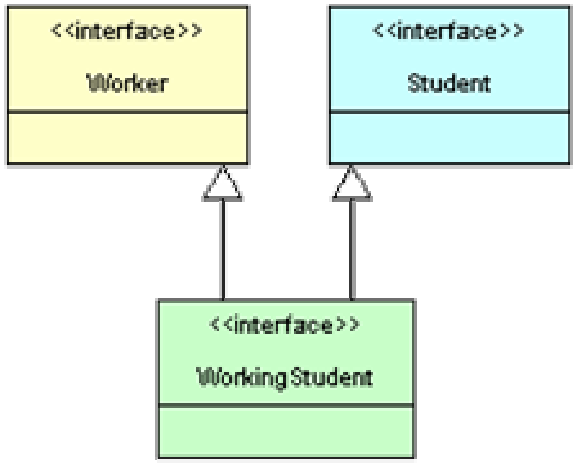
EREDITARIETÀ MULTIPLA

Esempio

```
public interface Worker {
...
}

public interface Student {
...
}

public interface WorkingStudent
   extends Worker, Student {
...
}
```



Dopo **extends** può esservi un elenco di *più interfacce*

Tassonomia di Forme Geometriche: modello con ereditarietà multipla tramite interfacce

TASSONOMIA di FORME GEOMETRICHE

La tassonomia di *forme geometriche* modellata tramite classi ha mostrato *limiti espressivi*

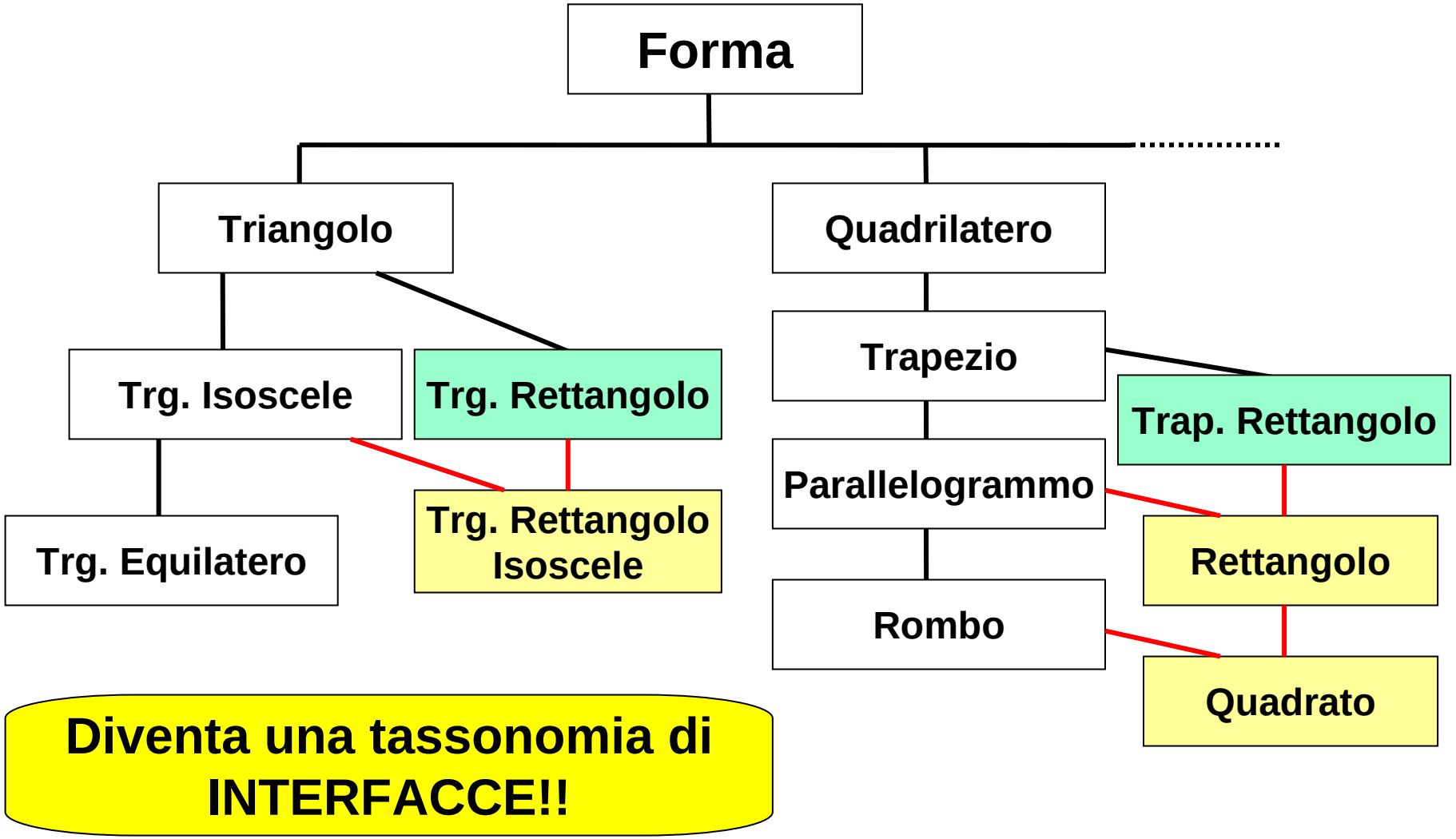
- **l'ereditarietà singola consente di classificare solo "per sottoinsiemi"**
- **si classifica usando "un solo criterio" per volta**
- ***ma la realtà delle forme geometriche è più variegata:* prevede *due criteri ortogonali fra loro* (lati paralleli, angoli retti)**
- **i limiti dell'ereditarietà singola non permettono di collocare bene *rettangoli, quadrati, etc.***

FORME GEOMETRICHE MODELLATE CON INTERFACCE

Le *interfacce* supportano *ereditarietà multipla*

- **consentono di esprimere "intersezioni" di insiemi → *migliore espressività***
- **si possono *applicare più "criteri di classifica-zione"* e comporli in modo naturale**
- ***adatta a rappresentare la realtà multiforme delle forme geometriche*:**
  - ***criterio dei lati paralleli***
  - ***criterio degli angoli retti***

TASSONOMIA *DI INTERFACCE*

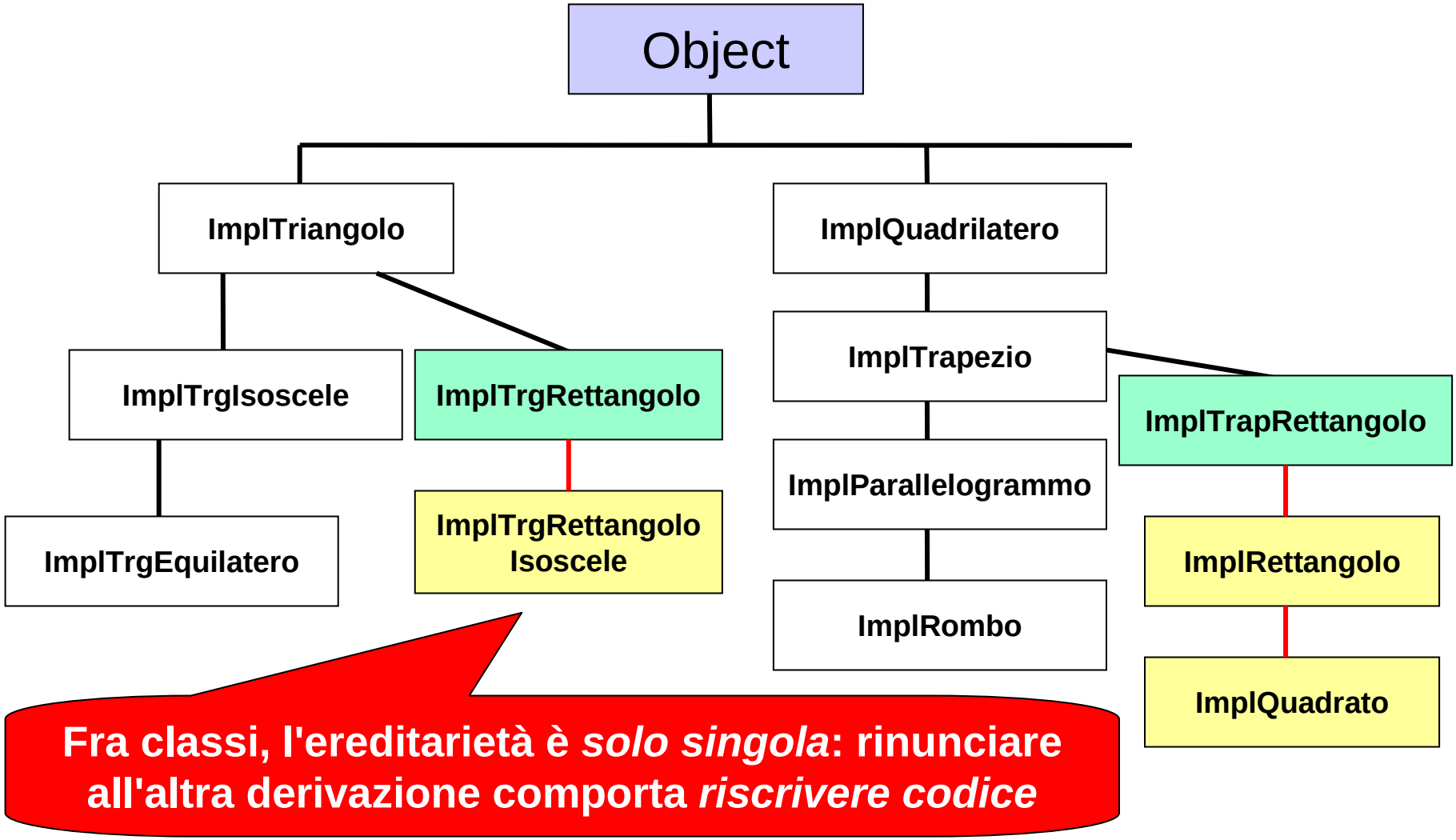


DALL'INTERFACCIA ALL'IMPLEMENTAZIONE

Una volta definita la tassonomia di *interfacce:*

- **come si decide la tassonomia di *classi* che le implementano ? Per le classi *non si può più usare l'ereditarietà multipla!***
  - **si privilegia la "linea di derivazione" che permette di riusare più codice e di riscriverne meno**
- **possono *coesistere* due tassonomie (una di interfacce, una di classi) *diverse*, o dev'esserci qualche relazione fra esse?**
  - **possono coesistere… ma non conviene abusare.**

TASSONOMIA *DI CLASSI*



Dipartimento di Informatica e Sistemistica
Antonio Ruberti

“Sapienza” Università di Roma

Il caso di studio Complessi /Reali: modello con interfacce

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

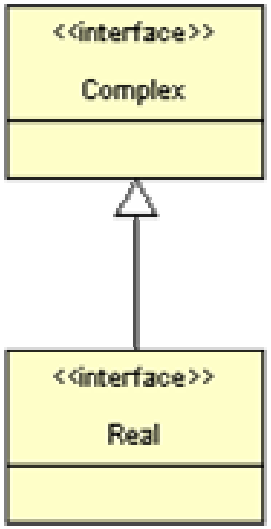
Si ringrazia il Prof. Enrico Denti per aver reso
disponibile il proprio materiale didattico sul quale si basano queste slides

Il caso di studio Complessi /Reali: modello con interfacce

IL CASO DI STUDIO Reali / Complessi

- **Usiamo *interfacce* per definire le *proprietà osservabili* degli ADT e le *relazioni tra loro***

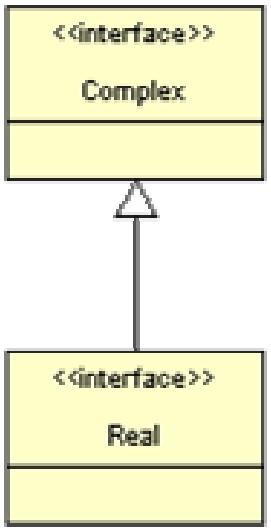
- **Poiché nella realtà i reali sono un sottoinsieme dei complessi, stabiliamo le seguenti relazioni *fra interfacce*:**
  - **`Complex` è l'interfaccia-base**
  - **`Real` la specializza**



L' INTERFACCIA Complex

```
public interface Complex {
 public double getReal();
 public double getIm();
 public double module();
 public Complex cgt();
 public Complex divByFactor(double x);
 public Complex sum(Complex z);
 public Complex sub(Complex z);
 public Complex mul(Complex z);
 public Complex div(Complex z);
 public String toString();
}
```

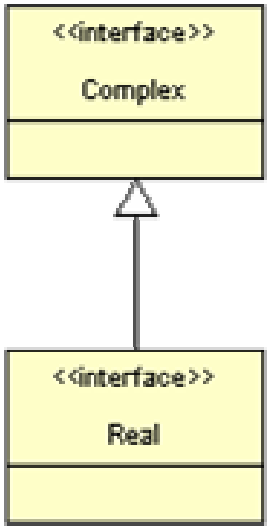
Non ci sono dati, ma è bene prevedere metodi di accesso alle informazioni caratterizzanti (anche modulo e arg .. ?)



L' INTERFACCIA Real

```
public interface Real extends Complex {
 public Real sum(Real x);
 public Real sub(Real x);
 public Real mul(Real x);
 public Real div(Real x);
}
```

Nota: non compare **`toString()`**, in quanto la sua signature è identica a quella presente in **`Complex`**.

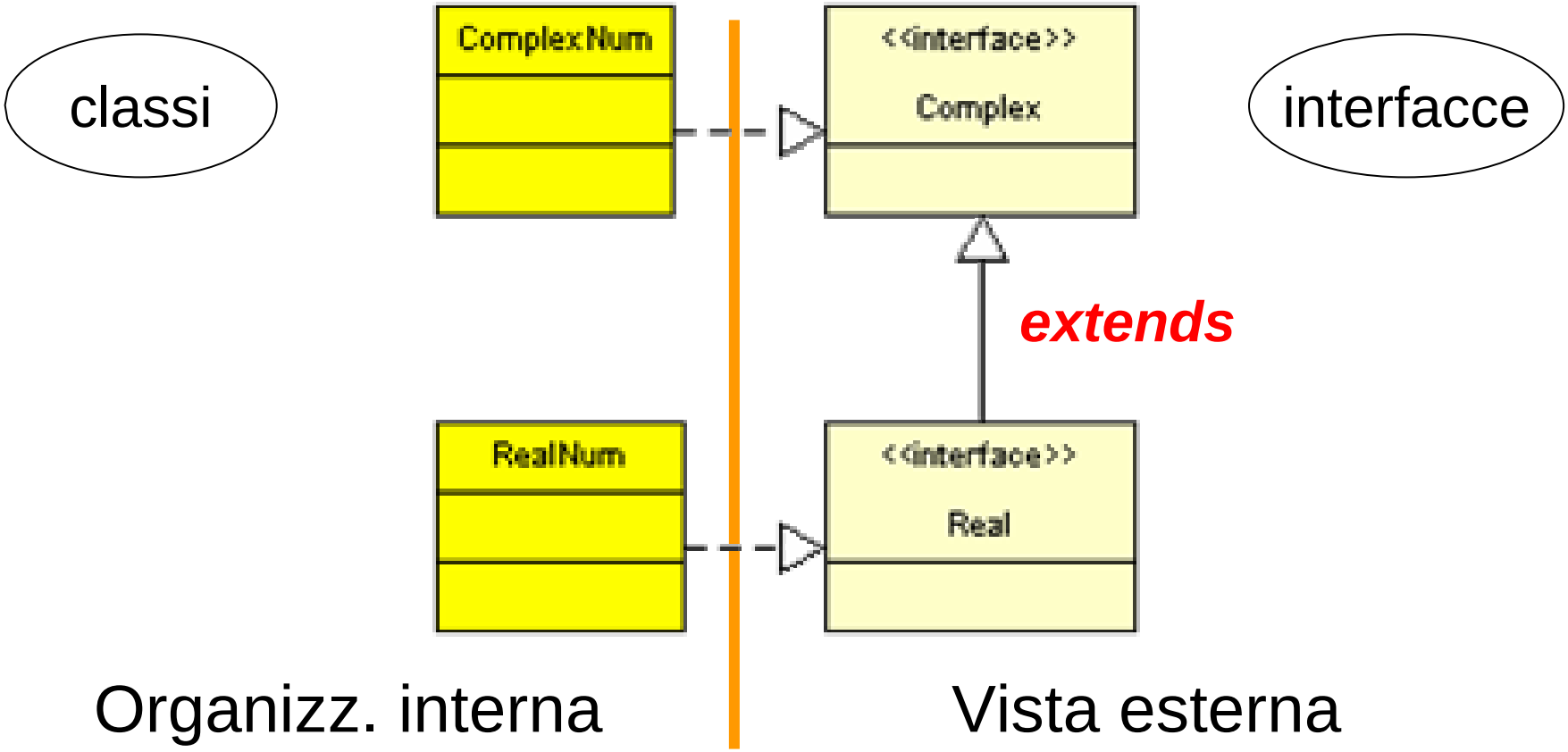


IL PROGETTO DELLE CLASSI

- **Usiamo *classi* per implementare le interfacce *in modo efficiente* dal punto di vista dell'uso delle risorse**
  - **`RealNum` implementa `Real`**
  - **`ComplexNum` implementa `Complex`**
- ***Non è detto che `RealNum` e `ComplexNum` siano in una qualche relazione fra loro*: anzi, potrebbero benissimo essere *due classi del tutto indipendenti***
- … oppure anche derivate una dall'altra.

REALI E COMPLESSI: IL PROGETTO

L'architettura complessiva:



LA CLASSE RealNum

```
public class RealNum implements Real {
 protected double re;
 public RealNum() { re=0; }
 public RealNum(double x) { re=x;  }
 //
 public double getReal() { return re; }
 public double getIm() { return 0; }
 //
 public double module() { return re<0 ? -re : re; }
 // operazioni con entrambi operandi reali
 public Real sum(Real x){return new RealNum(re + x.getReal());}
 public Real sub(Real x){return new RealNum(re - x.getReal());}
 public Real mul(Real x){return new RealNum(re * x.getReal());}
 public Real div(Real x){return new RealNum(re / x.getReal());}
 ...
```

LA CLASSE RealNum

```
 ...
 // operazioni con this reale, secondo operando Complex
 public Complex sum(Complex z) {
   return new ComplexNum( re+z.getReal(), z.getIm() ); }
 public Complex sub(Complex z) {
   return new ComplexNum( re-z.getReal(), z.getIm() ); }
 public Complex mul(Complex z) {
   return new ComplexNum( re*z.getReal(), re*z.getIm() ); }
 public Complex div(Complex z) {
   return new ComplexNum( re/z.getReal(), re/z.getIm() ); }
 public Complex cgt() { return this; }
 public Complex divByFactor(double x) {
   return new RealNum(re/x); }
 public String toString() {
   return Double.toString(re); }
}
```

Il coniugato di un Real è il Real stesso

Il risultato è in effetti un Real, ma l'interfaccia prevede un generico Complex

LA CLASSE ComplexNum

```
public class ComplexNum implements Complex {
 protected double re, im;
 public ComplexNum() { re = im = 0; }
 public ComplexNum(double x) { re = x; im = 0; }
 public ComplexNum(double x, double y) { re = x; im = y; }
 public double getReal() { return re; }
 public double getIm() { return im; }
 public double module(){return Math.sqrt(re*re+im*im); }
 public Complex cgt() { return new ComplexNum(re, -im); }
 public Complex divByFactor(double x) {
   return new ComplexNum(re/x, im/x); }
 public Complex sum(Complex z) {
   return new ComplexNum( re+z.getReal(), im+z.getIm()); }
 public Complex sub(Complex z) {
   return new ComplexNum( re-z.getReal(), im-z.getIm()); }
 ...
```

LA CLASSE ComplexNum

```
 ...
 public Complex mul(Complex z) {
   return new ComplexNum(
       re*z.getReal()-im*z.getIm(),
       re*z.getIm()+im*z.getReal()); }
 public Complex div(Complex z) {
   double mod = z.module();
   return mul(z.cgt()).divByFactor(mod*mod);
 }
  public String toString() { // stampa di un ComplexNum
   String res;
   if (re==0.0 && im==0.0) return "0";
   if (re==0.0) res = ""; else
   { res = Double.toString(re); if (im>=0.0) res += "+"; }
   res += (im==1 || im==-1 ? "" : Double.toString(im)) + "i";
   return res;
 }
}
```

UN PRIMO COLLAUDO

```
public class Prova {
 public static void main(String args[]){
   Real    r1 = new RealNum(18.5),       r2 = new RealNum(3.14);
   Complex c1 = new ComplexNum(-16, 0), c2 = new ComplexNum(3, 2),
           c3 = new ComplexNum(0, -2);
   Real    r = r1.sum(r2);
   Complex c = c1.sum(c2);
   System.out.println("r1 + r2 = " + r);     // il reale 21.64
   System.out.println("c1 + c2 = " + c);     // il complesso -13+2i
   c = c.sum(c3);
   System.out.println("c + c3 = " + c);      // il complesso -13+0i
   c = r;
   System.out.println("c = r; c = " + c);    // Qui c è reale
   // POLIMORFISMO: scatta la toString dei reali --> 21.64
 }
}
```

```
C:\esercizi>java Prova
r1 + r2 = 21.64
c1 + c2 = -13.0+2.0i
c + c3 = -13.0+0.0i
c = r; c = 21.64
```

UNA RIFLESSIONE

- **Nel programma precedente, si usano *quasi sempre* solo interfacce…**
  tranne che all'atto della creazione degli oggetti:

```
Real    r2 = new RealNum(3.14);
Complex c2 = new ComplexNum(3,2);
Real    r = r1.sum(r2);
Complex c = c1.sum(c2);
```

- **Dover specificare le classi *toglie flessibilità* al progetto, perché *ingessa la scelta di usare quell'implementazione..***
- **…quando tutto il resto sarebbe indipendente!**

UN APPROCCIO ALTERNATIVO

- **Sarebbe utile che il programma cliente usasse *sempre e solo* interfacce, nascondendo la fase di creazione degli oggetti**

```
Real    r2 = il reale di valore 3.14
Complex c2 = il complesso di valore 3+2i
Real    r = r1.sum(r2);
Complex c = c1.sum(c2);
```

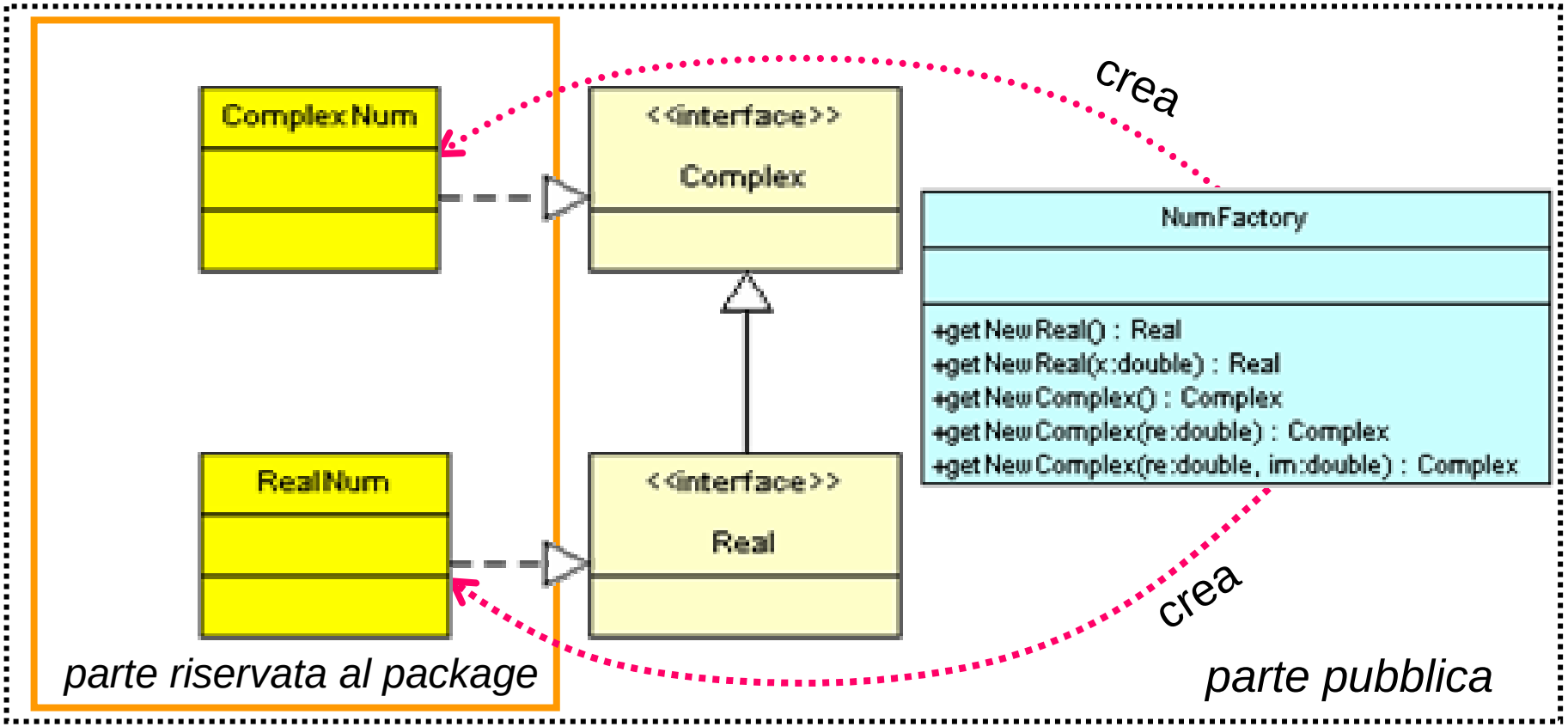
- **Così si potrebbe *cambiare l'implementazione in ogni momento*, senza impatto sui clienti.**
- **Ciò si ottiene *delegando la creazione dell'oggetto a una FABBRICA (FACTORY)***

IL PATTERN "FABBRICA" (factory)

- **La fabbrica nasconde agli utenti la costruzione degli oggetti**
- **Dunque, l'utente può non sapere di che classe è l'oggetto: basta che ne conosca *l'interfaccia!***
  - **le classi NON sono più PUBBLICHE!**
  - **la funzione statica che restituisce l'oggetto richiesto, già pronto per l'uso, specifica nella sua signature solo l'INTERFACCIA**

IL NUOVO PROGETTO con fabbrica

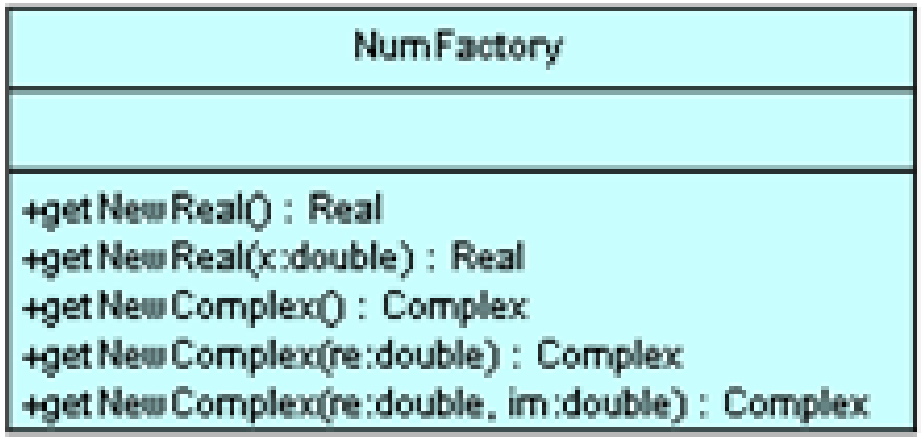
L'architettura complessiva (package)



LA CLASSE "FACTORY"

La classe "*factory*" `NumFactory`

- **è l'unica *classe* pubblica del package**
- **fornisce funzioni statiche che *incapsulano la new delle istanze di `ComplexNum` e `RealNum`***



`NumFactory` *restituisce formalmente riferimenti a* ***`Real`*** *o* ***`Complex`****:*
le classi `RealNum` o `ComplexNum` *fuori non si vedono mai !*

LA CLASSE "FACTORY"

```
package myNumbers;

public class NumberFactory {
   public static Real getNewReal() {
        return new RealNum(); }
   public static Real getNewReal(double x) {
        return new RealNum(x); }
   public static Complex getNewComplex() {
        return new ComplexNum(); }
   public static Complex getNewComplex(double x) {
        return new ComplexNum(x); }
   public static Complex getNewComplex(double x, double y) {
        return new ComplexNum(x,y); }
}
```

NumFactory *restituisce formalmente riferimenti a* ***Real*** *o* ***Complex****:*
le classi RealNum o ComplexNum *fuori non si vedono mai !*

LA CLASSE DI PROVA

```
import myNumbers.*;
public class Prova {
 public static void main(String args[]){
   Real    r1 = NumFactory.getNewReal(18.5),
           r2 = NumFactory.getNewReal(3.14);
   Complex c1 = NumFactory.getNewComplex(-16, 0),
           c2 = NumFactory.getNewComplex(3, 2),
           c3 = NumFactory.getNewComplex(0, -2);
   Real    r = r1.sum(r2);    Complex c = c1.sum(c2);
   System.out.println("r1 + r2 = " + r);     // il reale 21.64
   System.out.println("c1 + c2 = " + c);     // il complesso -13+2i
   System.out.println("c1 + c2 -i = " + c.sub(new Complex(0,1)));
   c = c.sum(c3);
   System.out.println("c + c3 = " + c);      // il complesso -13+0i
   c = r;
   System.out.println("c = r; c = " + c);    // Qui c è reale
 }
}
```

Creazione indiretta
di oggetti

```
C:\esercizi>java Prova
r1 + r2 = 21.64
c1 + c2 = -13.0+2.0i
c + c3 = -13.0+0.0i
c = r; c = 21.64
```