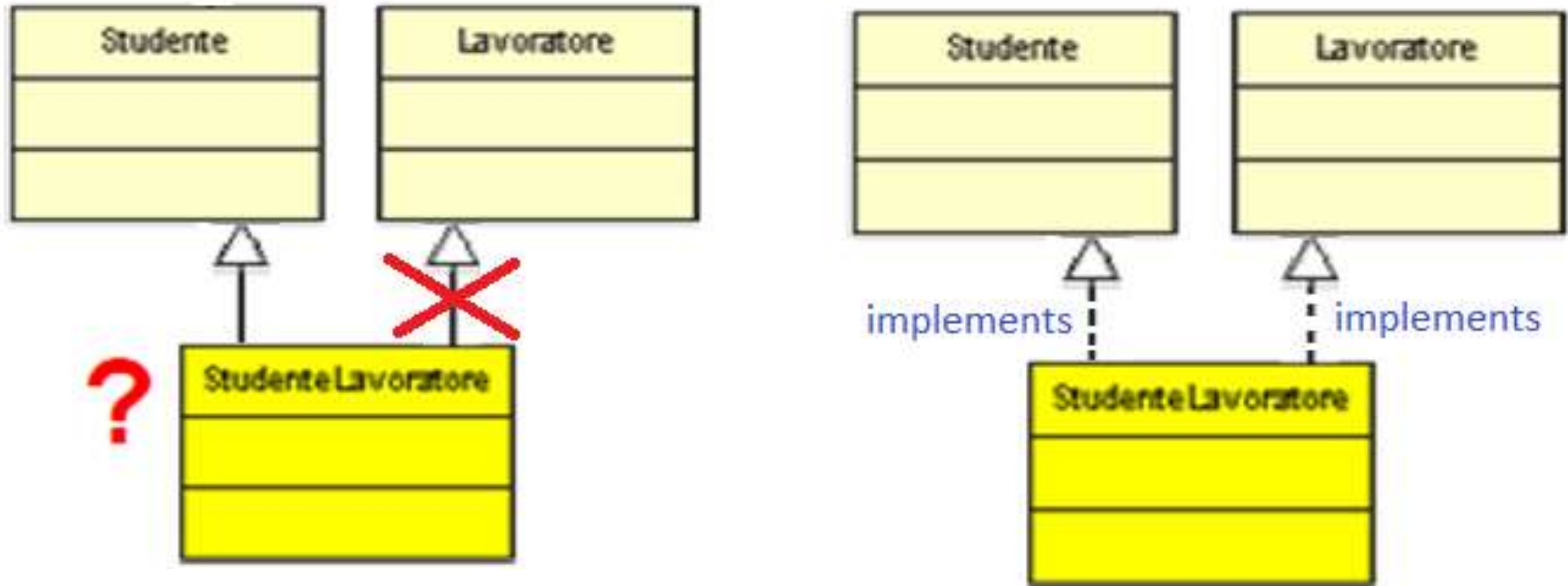
Interfacce in Java



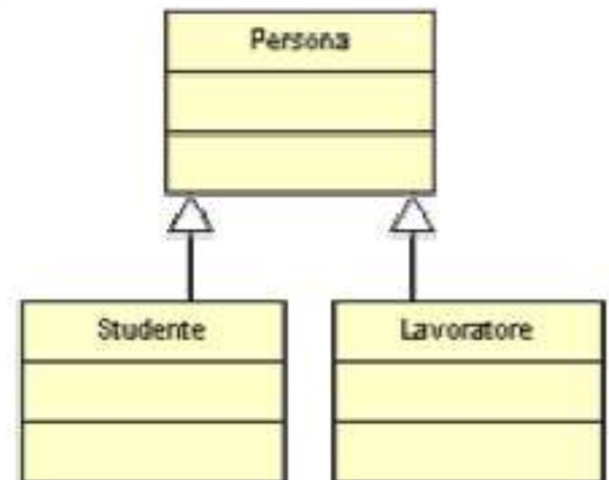
Superare il meccanismo dell'ereditarietà singola evitando ambiguità ma rendere disponibile il contratto di metodi comuni

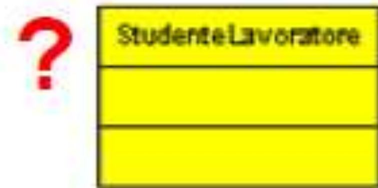
Interfacce in Java

tratto da risorsa [online]

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`,...)



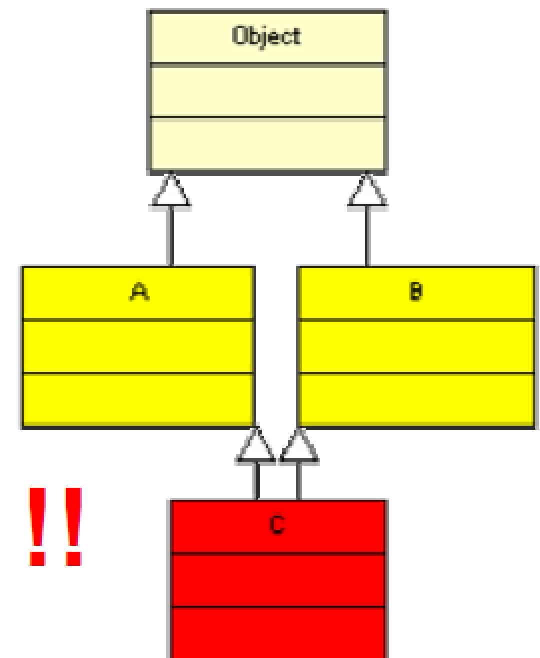


Perché non ammettere l'ereditarietà *multipla*?

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.

INTERFACE

- 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

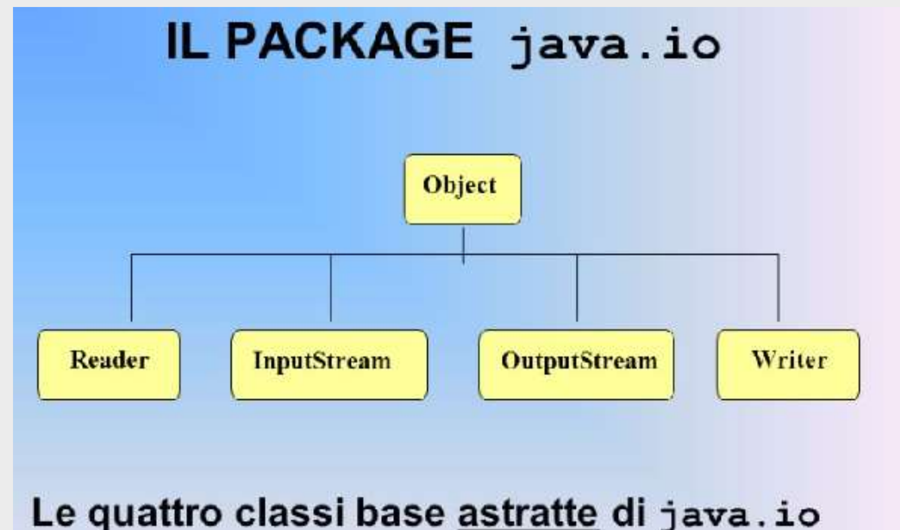
- **nel caso delle classi astratte, l'implementazione dei metodi "lasciati in bianco" dovrà essere fornita *da una sottoclasse***
- **nel caso delle interfacce, invece, tale implementazione potrà essere fornita *da una classe qualunque*.**

CLASSI ASTRATTE

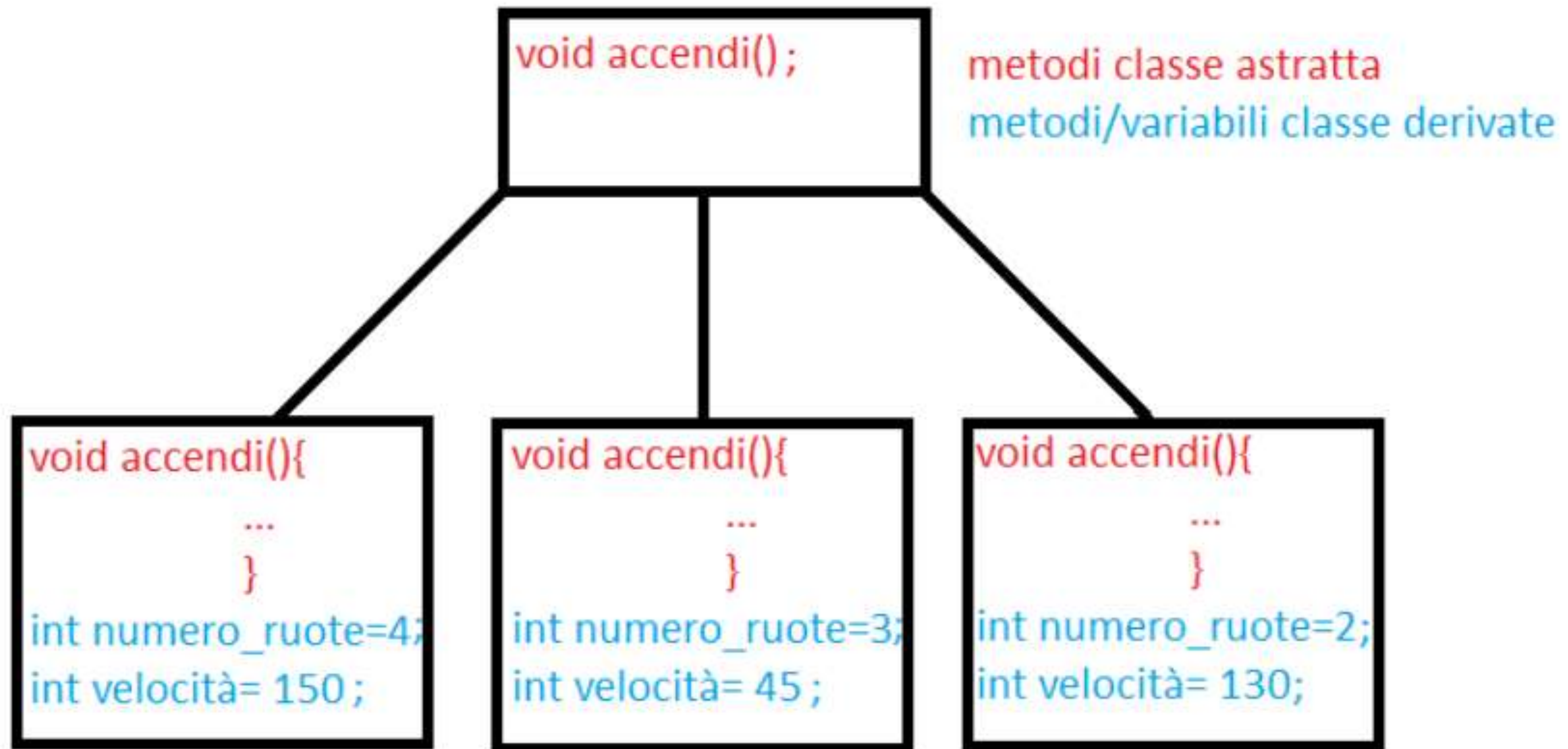
Una classe che può definire anche metodi astratti (cioè senza corpo).

Usata per poter dichiarare caratteristiche comuni fra classi di una determinata gerarchia.

La classe astratta non può essere instanziata.



Trasmissione di metodi da una classe astratta alle classi derivate



INTERFACCE vs CLASSI ASTRATTE

	Interfacce	Classi astratte
Istanziabile	no	no
Fields	solo `static final`	sì
Costruttore	no	sì
Metodi statici	Java8+	sì
Dichiarazione metodi (virtual)	no	sì
Implementazione metodi	java8+ (con il qualificatore `default`)	sì

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*

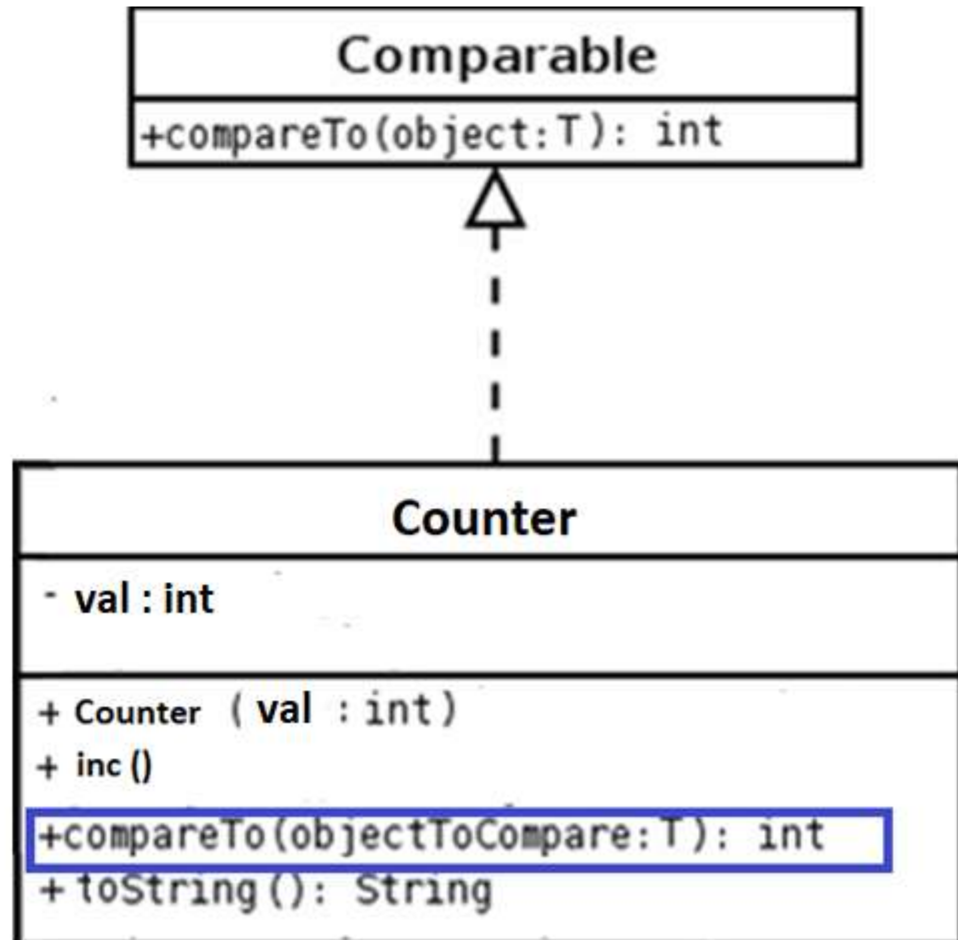
- le interfacce specificano le *dichiarazioni* dei metodi
- la classe *definisce (implementa)* tutti i metodi delle interfacce che si impegna a implementare
- Nasce dunque la *relazione "IMPLEMENTS"*

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(T x);`
- **implementare un metodo pubblico avente esattamente la signature**
  `public int compareTo(T x);`
  specificata dall'interfaccia.

La classe Counter



IL COUNTER *comparable*

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

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` ?

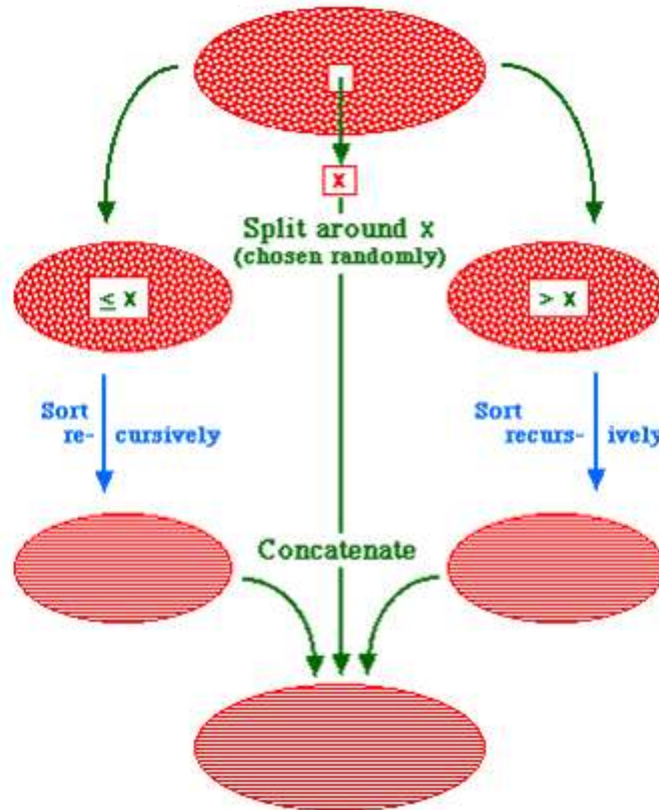
Perché un COUNTER *comparable* ?

- 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 oggetti

coi metodi sort() della classe Arrays

Quick sort:
divide et impera



ORDINARE UN ARRAY DI COUNTER

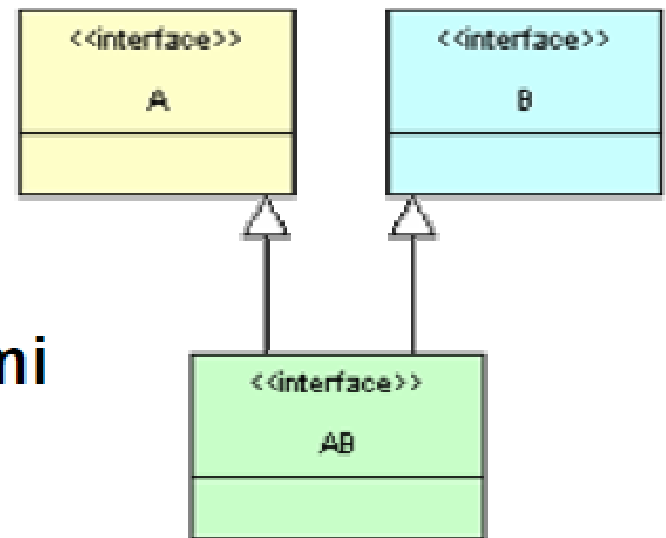
```
public class Test {
  public static void main(String args[]) {
   Counter[] myCounterArray = new Counter[4];
   myCounterArray[0] = new Counter(11);
   myCounterArray[1] = new Counter(10);
   myCounterArray[2] = new Counter(3);
   myCounterArray[3] = new Counter(5);
   java.util.Arrays.sort(myCounterArray);
   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
```

EREDITARIETA' 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



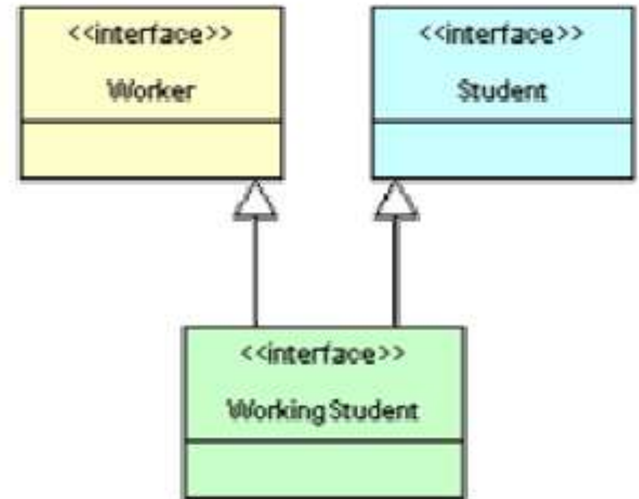
È un potente strumento di modellazione

UN ESEMPIO

```
public interface Worker {
...
}

public interface Student {
...
}

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



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

Definizioni

Un **metodo astratto** è un metodo senza corpo, con un "**;**" dopo l'intestazione.

Una **interfaccia (interface)** in Java ha una struttura simile a una classe, ma può contenere SOLO **metodi d'istanza astratti** e **costanti** (attributi *static final*) quindi non può contenere *costruttori*, *variabili* e *metodi statici*.

Si può dichiarare che una classe implementa (***implements***) una data interfaccia: in questo caso **deve realizzare tutti i suoi metodi astratti**, fornendo dei metodi con la stessa intestazione (e con il corpo, naturalmente).

La realizzazione di un metodo deve rispettare la specifica del corrispondente metodo astratto.

Le **interfacce possono essere utilizzate**:

• come **contratto** tra chi implementa una classe e chi la usa: le due parti possono essere sviluppate e compilate separatamente

• per evidenziare **funzionalità comuni a più classi**, sopperendo alle limitazioni dell'ereditarietà singola (come nell'esempio di Comparable)

Tratto da risorsa [online]