OLTRE LE CLASSI

Nella sua parte non statica, una classe fornisce la *definizione di un ADT*

- **parte visibile esternamente (public)**
- ***implementazione***
  - **dati privati, protetti, o visibili nel package**
  - **metodi privati, protetti, o visibili nel package**

In questo modo, però, diventa impossibile *dare una pura specifica di interazione di un ADT, senza definirlo*

OLTRE LE CLASSI

Alcune domande

- **Perché *dare una pura specifica di interazione di un ADT senza definirlo?***
- ***Non bastano, se mai, le classi astratte?***

OLTRE LE CLASSI

Alcune domande

- **Perché *dare una pura specifica di interazione di un ADT senza definirlo?***
- *Non bastano, se mai, le classi astratte?*

***Dare una pura specifica di interazione di un ADT senza definirlo* è essenziale per separare due aspetti:**
- **come interagisce un componente con gli altri**
- **come è realizzato un componente**

***Perché mai* per specificare il primo aspetto dovremmo essere obbligati a specificare sempre anche l'altro?**

OLTRE LE CLASSI

Alcune domande

- ***E perché* non bastano le classi astratte?**
- *Non permettono proprio di dare la specifica rinviando l'implementazione alle classi derivate?*

Perché le classi astratte dipendono dal meccanismo di ereditarietà,* che non è sufficientemente *flessibile
- **Java supporta solo *ereditarietà singola***
  Come specificare un componente le cui funzionalità sono specificate in varie classi diverse? Come "unirle"?
- **Definire la "giusta" tassonomia di classi potrebbe diventare *molto complesso* o al limite *impossibile***

OLTRE LE CLASSI

Possiamo anche avere classi e ereditarietà multipla

- **una classe** può derivare da **più di una classe esistente**

Nuova relazione tra classi

- implementazione difficile per possibili conflitti
- difficile gestione della gerarchia di ereditarietà delle classi, sia uso, sia supporto

OLTRE LE CLASSI

Può essere utile disporre di un *nuovo costrutto*

- **simile alla (parte non statica di una) classe,** nel senso di consentire la definizione del "modo di interagire" di un'entità...
- **… ma non tenuto a fornire *implementazioni*…**
- **… né legato alla *gerarchia di ereditarietà* delle classi, con i relativi vincoli**

INTERFACCE

INTERFACCE

Una *interfaccia* costituisce una *pura specifica di interfaccia*

- **contiene *solo dichiarazioni* di metodi**
- **ed eventualmente costanti**
- **ma non contiene né variabili né *definizioni* di metodi**

INTERFACCE

Praticamente, una *interfaccia*

- **è strutturalmente *analoga alla parte di interfaccia di una classe...***
- **ma è introdotta dalla parola chiave `interface` anziché `class`**
- **contiene solo *dichiarazioni* di metodi**

Esempio:

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

INTERFACCE E PROGETTO

Le interfacce inducono un diverso *modo di concepire il progetto*

- **prima si definiscono le interfacce delle entità che costituiscono il sistema**
  - **in questa fase si giocano *scelte di progetto (pulizia concettuale)***
- **poi si realizzeranno le classi che "implementeranno" tali interfacce**
  - **in questa fase entreranno in gioco *scelte implementative (efficienza ed efficacia)***

UN ESEMPIO di PURA SPECIFICA

Definizione dell'astrazione "Collezione"

- **Cosa si intende per "Collezione"?** ossia
- ***Come ci si aspetta di poter interagire* con un'entità qualificata come "Collezione"?**

***Indipendentemente da qualsiasi scelta o aspetto implementativo,* una "Collezione" è tale perché**

- **è un contenitore → è possibile chiedersi se è vuota e quanti elementi contiene**
- **vi si possono aggiungere e togliere elementi**
- **è possibile chiedersi se un elemento è presente o no**
- **...**

UN ESEMPIO

Una "Collezione" è dunque una *qualsiasi entità che si conformi a questo "protocollo di accesso"*

- **è possibile chiedersi se è vuota**
- **è possibile chiedersi quanti elementi contiene**
- **vi si possono aggiungere e togliere elementi**
- **è possibile chiedersi se un elemento è presente o no**
- **...**

È possibile (e utile!) definire questo concetto *prima ancora di iniziare a pensare* come sarà realmente realizzata una "Collezione"!

un ESEMPIO di ASTRAZIONE PURA

Si definiscono così astrazioni di dato in termini di *comportamento osservabile*, ossia di

- **"cosa ci si aspetta" da esse**
- **"cosa si pretende che esse sappiano fare"**

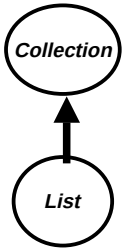
rinviando a tempi successivi la realizzazione pratica di ADT (classi) che rispettino questa specifica.

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

GERARCHIE DI INTERFACCE

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

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



La gerarchia delle interfacce:

- **è una gerarchia separata da quella delle classi**
- **è slegata dagli aspetti implementativi**
- **esprime le *relazioni concettuali* della realtà**
- **guida il progetto del *modello della realtà.***

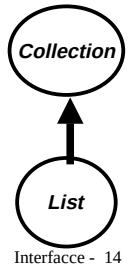
GERARCHIE: ESEMPIO

Come in ogni gerarchia, anche qui le interfacce derivate:

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

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

- **ogni lista può interagire col mondo come farebbe una collezione** *(magari in modo specializzato)***…**
- **… ma può avere *proprietà peculiari al concetto di lista*,** che non valgono per una "collezione" qualsiasi.

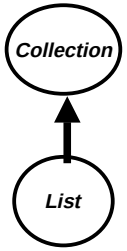


GERARCHIE: ESEMPIO

Ad esempio, una "Lista" ha un concetto di *sequenza,* di *ordine* fra i suoi elementi

- **esiste quindi un *primo elemento*, un secondo, …**
- **quando si aggiungono nuovi elementi bisogna dire *dove* aggiungerli** (ad esempio, in coda)
- **è possibile recuperare un elemento *a partire dalla sua posizione*** (il primo, il decimo,…)

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



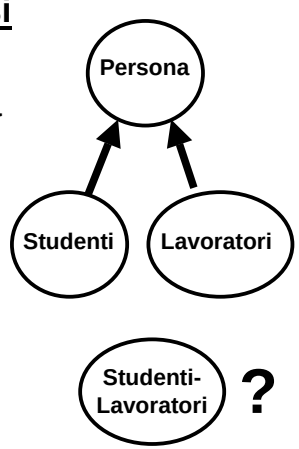
INTERFACCE: UN ALTRO ASPETTO

In Java, l'ereditarietà fra classi può essere solo *singola*

- **una classe può ereditare da *una sola superclasse***

Questo può essere limitativo in alcune circostanze:

- **se `Studenti` e `Lavoratori` estendono `Persona`...**
- ***...dove collocare la classe `StudentiLavoratori`?***



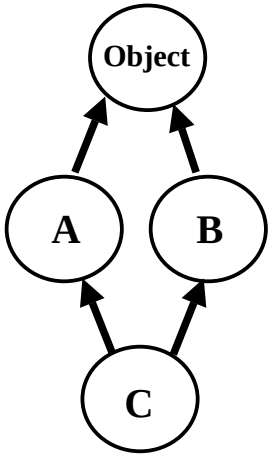
INTERFACCE: UN ALTRO ASPETTO

Altri linguaggi supportano l'ereditarietà *multipla*

- **una classe può ereditare da *più superclassi***

che però comporta *non pochi problemi:*

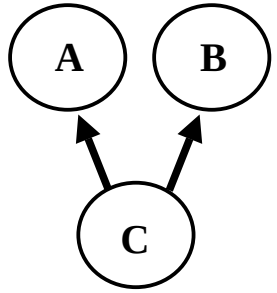
- **concettuali** (duplicazioni di dati e proprietà)
- **tecnico-pratici** (omonimie di dati e/o metodi)



EREDITARIETÀ MULTIPLA

Con l'ereditarietà *multipla fra classi:*

- **la classe C unisce *sia i dati di A sia quelli di B***
  - **che si fa con le omonimie?**
  - **come si distinguono?**
- **la classe C unisce *i metodi di A con quelli di B***
  - **e se ci sono definizioni doppie?**
  - **e nelle classi successive?**

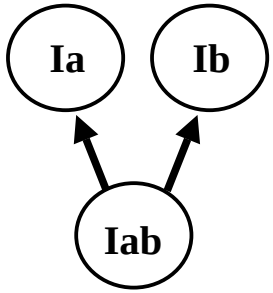


Java non supporta* l'ereditarietà multipla *fra classi

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 modellizzazione

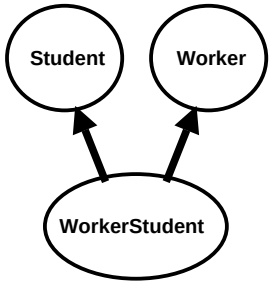
EREDITARIETÀ MULTIPLA

Esempio

```
public interface Worker {
...
}

public interface Student {
...
}

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



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

DALL'ASTRAZIONE ALL'IMPLEMENTAZIONE

Una interfaccia definisce una astrazione di dato in termini di *comportamento osservabile*

Per sua natura, però, *non implementa nulla*

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

DALL'ASTRAZIONE ALL'IMPLEMENTAZIONE

Una interfaccia definisce una astrazione di dato in termini di *comportamento osservabile*

Per sua natura, però, *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 contengono *dichiarazioni* di metodi**
- **la classe *definisce (implementa)* tutti i metodi delle interfacce che si impegna a implementare**

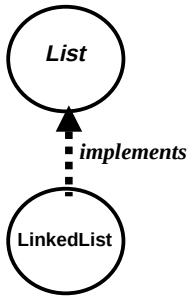
DALL'ASTRAZIONE ALL'IMPLEMENTAZIONE

Esempio:

```
public class LinkedList implements List {
...
}
```

Essendosi impegnata a implementare tale interfaccia, la classe `LinkedList` DEVE definire TUTTI i metodi dichiarati dall'interfaccia `List`

Se non lo fa, il compilatore Java segnala ERRORE



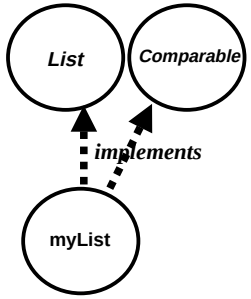
DALL'ASTRAZIONE ALL'IMPLEMENTAZIONE

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, il compilatore Java segnala ERRORE



INTERFACCE E PROGETTO

Le interfacce inducono un diverso *modo di concepire il progetto*

- **prima si definiscono le interfacce che servono, e si stabiliscono le relazioni tra loro**
  - **la gerarchia delle interfacce riflette *scelte di progetto* *(pulizia concettuale)***
- **poi si realizzano le classi che implementano tali interfacce**
  - **la gerarchia delle classi riflette *scelte implementative* *(efficienza ed efficacia)***

INTERFACCE E PROGETTO

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

- **non dichiarano funzionalità**
- **fungono da marcatori per le classi che asseriscono di implementarle**
  - **esempi: `Cloneable`, `Serializable`**
- **una classe che le implementa non è tenuta a fornire alcuna funzionalità, ma *"parla di sé"***
  - **afferma di essere clonabile, serializzabile..**
  - **è una forma di auto-documentazione**
  - ***sfrutta il compilatore per trovare incongruenze nel "modello del mondo"***

INTERFACCE: USO

Il nome di una interfaccia può essere usato come *identificatore di tipo* per *riferimenti* e *parametri formali* di metodi e funzioni.

```
public static void main(String s[]){
  Collection c;
  ...
  List l1;
  List l2;
  ...
}
```


INTERFACCE: USO

A tali riferimenti si possono assegnare *istanze di una qualunque classe che implementi l'interfaccia*

```
public static void main(String s[]){
  Collection c = new Vector();
  c.add(new Point(3,4));
  ...
  List l1 = new LinkedList();
  List l2 = new LinkedList(c);
}
```

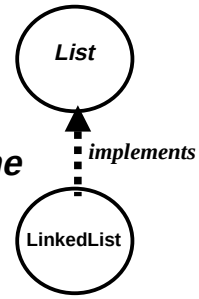
Per maggiori dettagli si veda il package `java.util`

- Le classi **`Vector`** e **`LinkedList`** implementano entrambe l'interfaccia **`List`**, che a sua volta estende **`Collection`**
- Il costruttore di default di **`Vector`** e **`LinkedList`** crea una collezione di oggetti vuota

INTERFACCE: USO

In effetti, quando una classe implementa una interfaccia, le *dichiarazioni dei metodi* contengono *il nome dell'interfaccia...*

```
public List copy(List z){
   return new LinkedList(z);
}
```



***...* ma in realtà tali metodi ricevono e *manipolano istanze di una qualche classe* che implementi l'interfaccia richiesta.**

USO

- **I riferimenti che introduciamo devono *riflettere la vista esterna***
  → **`Collection`, `List`, ...**
- **Gli oggetti che creiamo devono necessariamente essere istanze di *classi concrete***
  → **`LinkedList`, `Vector`, ...**

Esempi

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


Uso Interfacce per ESTENSIONI

- **Le interfacce che introduciamo possono *servire a descrivere comportamenti condivisi***
  → **`Serializable`, `Runnable`, ...**
- **Una classe può dichiarare molte interfacce e differenziate, se ne implementa i metodi**

Esempio

```
Public Class ActiveWindows implements
   Runnable, Serializable, ... {
}
```