Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Modulo di Laboratorio di Algoritmi e Strutture Dati

Java: Le interfacce

Le interfacce

- Ad uno stesso problema algoritmico possono corrispondere diverse soluzioni algoritmiche caratterizzate da prestazioni differenti
- In un progetto sw vorremmo potere utilizzare una qualunque implementazione a “scatola chiusa” e in modo interscambiabile, senza dovere modificare l’interfaccia verso l’applicazione chiamante
  - Oltre le classi astratte….

Richiami: il problema dei duplicati

```
public static boolean verificaDupList (List S){
  for (int i=0; i<S.size(); i++){
      Object x=S.get(i);
      for (int j=i+1; j<S.size(); j++){
            Object y=S.get(j);
            if (x.equals(y)) return true;
      }
   }
   return false;
}
```

Richiami: il problema dei duplicati

```
public static boolean verificaDupOrdList (List S){
   Collections.sort(S);
   for (int i=0; i<S.size()-1; i++)
      if (S.get(i).equals(S.get(i+1))) return true;
   return false;
}
```

Richiami: il problema dei duplicati

```
public static boolean verificaDupArray (List S){
  Object[] T = S.toArray();
 for (int i=0; i<T.length(); i++){
     Object x=T[i];
     for (int j=i+1; j<T.length; j++){
          Object y=T[j];
          if (x.equals(y)) return true;
     }
   }
   return false;
}
```

Richiami: il problema dei duplicati

```
public static boolean verificaDupOrdArray (List S){
 Object[] T = S.toArray();
 Arrays.sort(T);
 for (int i=0; i<T.length(); i++){
      if (T[i].equals(T[i+1])) return true;
      }
  }
  return false;
}
```

Le interfacce

- `verificaDupList, verificaDupOrdList verificaDupArray,verificaDupOrdArray` hanno stesso parametro e stesso tipo restituito, ma nomi diversi:

```
public static boolean <nome_m> (List S)
```

- Modificare un progetto sw per utilizzare una diversa implementazione comporta la sostituzione di ogni occorrenza del nome del metodo

Le interfacce

- Vorremmo utilizzare lo **stesso nome** di metodo rimanendo liberi di scegliere in seguito ed in modo indipendente l'implementazione più adatta allo specifico scenario applicativo senza costose modifiche
- Il meccanismo del **polimorfismo** dei metodi ci viene in aiuto…
  - definendo un'**interfaccia** Java che specifica l'intestazione del metodo **`verificaDup`** che risolve il problema dei duplicati
  - definendo per ogni diversa realizzazione una classe opportuna che implementa l'interfaccia data

Le interfacce

- Un'**interfaccia** è un insieme di metodi astratti e costanti, senza campi e senza alcuna definizione di metodo
- In ogni interfaccia tutti gli identificatori di metodi e di costanti sono pubblici
- Le interfacce non contengono costruttori

Le interfacce

- Quando una classe fornisce le definizioni dei metodi di un'interfaccia si dice che **implementa** o **realizza** l'interfaccia
- Le interfacce non contengono costruttori perché i costruttori sono sempre relativi ad una classe
- La classe può anche definire altri metodi

Le interfacce: Il problema dei duplicati

```
public interface AlgoDup {
  public boolean verificaDup(List S);
}
public class VerificaDupList implements AlgoDup {
  public boolean verificaDup (List S)
    { <corpo di verificaDupList> }
}
public class VerificaDupOrdList implements AlgoDup {
  public boolean verificaDup (List S)
    { <corpo di verificaDupOrdList> }
}
```

… così via per le realizzazioni delle classi `VerificaDupArray` e `VerificaDupOrdArray`

Le interfacce: Il problema dei duplicati

- In questo modo, anziché 4 metodi con nomi diversi, abbiamo:
  - uno stesso metodo **`verificaDup`**
  - differenti realizzazioni in 4 diverse classi

Le interfacce: Il problema dei duplicati

- L'implementazione dell'interfaccia obbliga il programmatore a rispettare l'intestazione del metodo **`verificaDup`** nelle varie classi
- I metodi verranno invocati nella forma generica

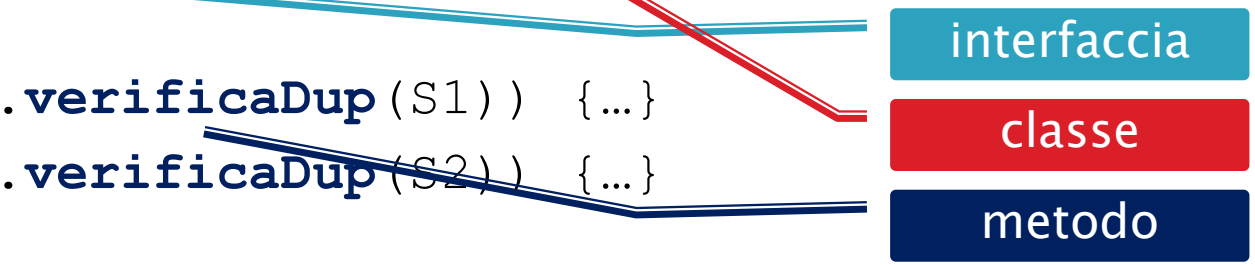
```
v.verificaDup(S)
```

- dove **`v`** è il riferimento ad un oggetto di una classe che implementa l'interfaccia **`AlgoDup`**

Le interfacce: Il problema dei duplicati

- Decidendo la classe dell'oggetto `v` si controlla la particolare implementazione che si intende usare
- Per decidere quale algoritmo utilizzare basta modificare la prima linea del seguente blocco di codice. Tutte le occorrenze di **`v.verificaDup`** restano invariate al variare della classe scelta nella linea 1.

```
AlgoDup v=new VerificaDuplist();//scelta algoritmo
…
If (v.verificaDup(S1)) {…}
If (v.verificaDup(S2)) {…}
…
```



Un altro esempio: L'interfaccia `Figure`

- Supponiamo di volere creare classi per cerchi, rettangoli ed altre figure
- Ciascuna classe avrà metodi per disegnare la figura e spostarla da un punto dello schermo ad un altro
- Esempio: la classe `Circle` avrà un metodo `draw` ed un metodo `move` basati sul centro del cerchio e sul suo raggio

L'interfaccia Figure

```
public interface Figure {
  // costanti
  final static int MAX_X_COORD=1024;
  final static int MAX_Y_COORD=768;

  /**
  * Disegna questo oggetto di tipo Figure
  * centrandolo rispetto alle coordinate
  * fornite.
  *
  *@param x la coordinata X del punto centrale
  *         della figura da disegnare.
  *@param y la coordinata Y del punto centrale
  *         della figura da disegnare.
  */
  public void draw(int x, int y);
}
```

L'interfaccia `Figure`

```
  /**
  * Sposta questo oggetto di tipo Figure
  * nella posizione di cui vengono fornite
  * le coordinate.
  *
  *@param x la coordinata X del punto centrale
  *         della figura da spostare.
  *@param y la coordinata Y del punto centrale
  *         della figura da spostare.
  */
  public void move(int x, int y);
}
```

L'interfaccia Figure

```
Public class Circle implements Figure {
  // dichiarazione di campi
  private int xCoord, yCoord, radius;

  // costruttori che inizializzano x,y, e il raggio
  …
  /** (commenti javadoc)
  */
  public void draw(int x, int y){
      xCoord=x; yCoord=y;
      // … disegna il cerchio
  }
  public void move(int x, int y){
      // … definizione del metodo move
  }
} // classe Circle
```

L'interfaccia Volante

```
public class Veicolo {
        public void accelera() { ... }
        public void decelera() { ... }
}

public class Aereo extends Veicolo {
        public void decolla() { ... }
        public void atterra() { ... }
        public void accelera() {
                // override del metodo ereditato ... }
        public void decelera() {
                // override del metodo ereditato ... }
        ...
}
```

L'interfaccia Volante

```
public class Automobile extends Veicolo {
  public void accelera() { // override del metodo ereditato ...}
  public void decelera() { // override del metodo ereditato ...}
  public void innestaRetromarcia() { ... }
  ...
}

public class Nave extends Veicolo {
  public void accelera() { // override del metodo ereditato ...}
  public void decelera() { // override del metodo ereditato ...}
  public void gettaAncora() { ... }
  ...
}
```

L'interfaccia Volante

```
public interface Volante {
  void atterra();
  void decolla();
}

public class Aereo extends Veicolo implements Volante {
  public void atterra() { ... ... }
  public void decolla() { ... ... }
  public void accelera(){ // override del metodo di Veicolo ... }
  public void decelera(){ // override del metodo di Veicolo ... }
}
```

L'interfaccia `Volante`

Qual è il vantaggio di tale scelta?

- Ovvia risposta: possibilità di utilizzo del polimorfismo. Infatti, sarà legale scrivere

```
Volante volante = new Aereo();
```

- oltre ovviamente a

```
Veicolo veicolo = new Aereo();
```

- e quindi si potrebbero sfruttare parametri polimorfi, collezioni eterogenee e invocazioni di metodi virtuali, in situazioni diverse.
- Potremmo anche fare implementare alla classe Aereo altre interfacce…

L'interfaccia `Ordinabile`

- Consideriamo la seguente classe `Punto`, che serve per memorizzare dei punti di un piano mediante le coordinate x e y:

```
class Punto {
    int x;
    int y;
    public Punto (int x, int y) {
        this.x = x;
        this.y = y;
    }
    int leggiX() { return x;}
    int leggiY() { return y;}
    int distanza() {
        return (int) Math.sqrt(x * x + y * y);
    }
}
```

L'interfaccia `Ordinabile`

- Abbiamo già visto come usare una **classe astratta** (classe `VettoreOrdinato`) per scrivere un algoritmo utilizzabile per ordinare oggetti di una qualsiasi classe
- Con questo approccio è necessario dichiarare una classe specializzata per ogni classe che si intende ordinare, anche se tale classe contiene poco codice
  - es. classe `VettoreTempo`
  - es. classe `VettorePunto`

L'interfaccia `Ordinabile`

- Cosa hanno in comune la classe `Punto` e la classe `Tempo` ?
  - Gli oggetti di entrambe possono essere ordinati secondo un criterio univoco
- In effetti, si può desiderare di ordinare oggetti di un gran numero di classi, secondo criteri diversi
- Sarebbe comodo dire che una qualsiasi classe è ordinabile se ha un metodo **`maggioreDi`** che consenta di confrontare due istanze e di stabilire quale delle due deve precedere l'altra.

L'interfaccia `Ordinabile`

- In questo modo delineiamo una specie di classe trasversale che accomuna classi diverse la cui unica caratteristica comune è quella di avere un metodo con la stessa firma.
- Potremmo poi avere una classe che ordina questa classe trasversale, senza la necessità di avere classi specializzate.

L'interfaccia `Ordinabile`

Per esempio:

```
interface Ordinabile {
    public boolean maggioreDi (Ordinabile o);
}
```

- Ricorda: non si può creare un'istanza di un'interfaccia con l'istruzione `new`, visto che sarebbe vuota, per cui quando un oggetto è di un tipo corrispondente a un'interfaccia, significa che appartiene a una classe che *implementa* quell'interfaccia.

L'interfaccia `Ordinabile`

- Una classe può *implementare* una o più interfacce dichiarandole esplicitamente e implementando i metodi dichiarati nelle interfacce stesse.
- In tal caso, gli oggetti di questa classe saranno riconosciuti anche come oggetti che implementano l'interfaccia.

L'interfaccia Ordinabile

```
class Punto implements Ordinabile  {
   int x; int y;
   public Punto (int x, int y) { this.x = x; this.y = y; }
   int leggiX() { return x; }
   int leggiY() { return y; }
   int distanza() {
       return (int) Math.sqrt(x * x + y * y);
   }
   public boolean maggioreDi (Ordinabile o) {
   // criterio d'ordinamento la distanza dall'origine
       if (o instanceof Punto)
          return distanza() > ((Punto) o).distanza();
       else
          return false;
   }
}
```

L'interfaccia `Ordinabile`

- Modifichiamo la classe **`VettoreOrdinato`** vista precedentemente in modo che possa funzionare con le interfacce. Si può notare che dal punto di vista del trattamento, non c'è nessuna differenza tra istanze di classi e istanze di classi che implementano interfacce.

```
class VettoreOrdinato
{
   private Ordinabile vettore[];
   private int maxElementi;
   private int curElementi;

   public VettoreOrdinato (int maxElementi) {
      this.maxElementi = maxElementi;
      vettore = new Ordinabile[maxElementi];
      curElementi = 0;
   }
```

L'interfaccia Ordinabile

```
public boolean aggiungi (Ordinabile elemento) {
     if (elemento != null && curElementi < maxElementi) {
        vettore[curElementi++] = elemento;
        return true;
     } else return false;
  }

  public Ordinabile leggi (int indice) {
     if (indice >= 0 && indice < curElementi)
       return (vettore[indice]);
      else  return null;
  }

  public int maxElementi () { return maxElementi; }
  public int elementi () { return curElementi; }
```

L'interfaccia Ordinabile

```
public void ordina () {  // shell-sort
      int   s, i, j, num;
      Ordinabile temp
      num = curElementi;
      for (s = num / 2; s > 0; s /= 2)
         for (i = s; i < num; i++)
            for (j = i - s; j >= 0; j -= s)
               if (vettore[j].maggioreDi (vettore[j + s])) {
                  temp = vettore[j];
                  vettore[j] = vettore[j + s];
                  vettore[j + s] = temp;
               }
   }
}
```

L'interfaccia `Ordinabile`

- Verifica del funzionamento di quanto visto:

```
class TestOrdinamento2
{
   public static void main (String argv[]) {
      VettoreOrdinato vo = new VettoreOrdinato(10);
      vo.aggiungi (new Punto(30,40));
      vo.aggiungi (new Punto(300,400));
      vo.aggiungi (new Punto(3,4));
      vo.ordina();
      for (int i = 0; i < vo.elementi(); i++)
      System.out.println(((Punto)(vo.leggi (i))).distanza());
   }
}
```

- Il codice produce il risultato

```
5
50
500
```

Tipi di dati astratti e strutture dati (cenno)

- Un **tipo di dato astratto** è costituito da un insieme di valori e da un insieme di operazioni ad esse relative
- Nei linguaggi O.O. come Java, i tipi di dato astratti corrispondono alle **interfacce**, nel senso che per ogni classe che implementa un'interfaccia l'utilizzatore può:
  - Creare un oggetto della classe ("oggetto" corrisponde a "valore")
  - Invocare i metodi pubblici della classe ("metodo pubblico" corrisponde a "operazione")

Tipi di dati astratti e strutture dati

- Una **struttura dati** è la realizzazione concreta (o *implementazione*) di un tipo di dato astratto
- Nei linguaggi O.O. come Java un programmatore può progettare una **classe** che implementa un'interfaccia
- In altre parole valgono le associazioni:
  - Tipo di dato astratto – Interfaccia
  - Struttura dati – Classe

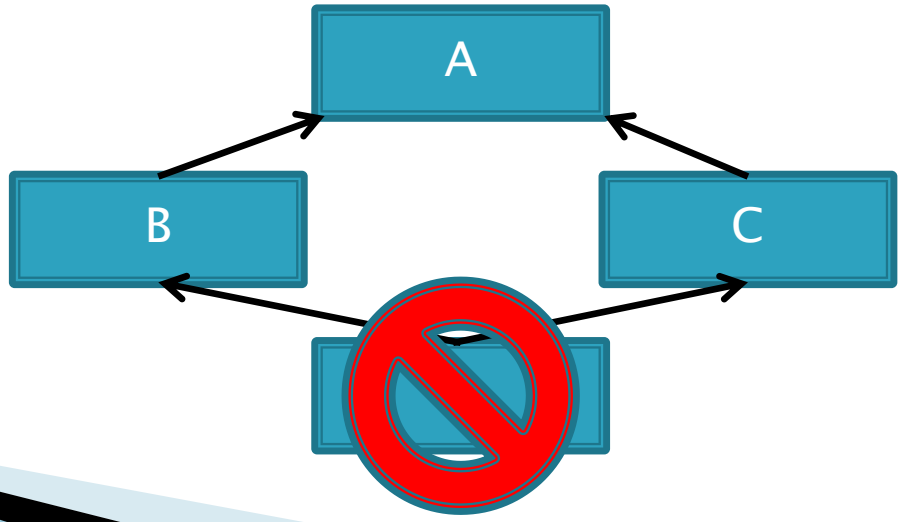
Ereditarietà multipla ed interfacce

- In Java non esiste la cosiddetta “ereditarietà multipla” (come in C++).
- Questa permette ad una classe di estendere più classi contemporaneamente.
- In pratica non è possibile scrivere:

```
public class Idrovolante extends Nave,Aereo
  {
  . . .
  }
```

Ereditarietà multipla ed interfacce

- L'ereditarietà multipla può essere causa di ambiguità.
- Ad esempio se due classi B e C ereditano dalla classe A e la classe D eredita sia da B che da C, se un metodo in D chiama un metodo definito in A, da quale classe viene ereditato?



Ereditarietà multipla ed interfacce

- Tale ambiguità prende il nome di **problema del diamante,** a causa della forma del diagramma di ereditarietà delle classi, simile ad un diamante.
- In Java per risolvere questo inconveniente si è adottato questo compromesso:
  - una classe può ereditare le **interfacce** da più di una classe base – cioè esporre all'esterno gli stessi metodi delle interfacce delle classi base
  - ma può ereditare i dati ed i metodi effettivi da una sola classe base.

Ereditarietà multipla ed interfacce

- In altre parole la differenza tra l'**implementare** un'interfaccia ed **estenderla** consiste nel fatto che, mentre possiamo estendere una sola classe alla volta, possiamo invece implementare infinite interfacce, simulando di fatto l'ereditarietà multipla, ma senza i suoi effetti collaterali negativi.

Classi astratte vs Interfacce

- Il vantaggio che offrono sia le classi astratte che le interfacce, risiede nel fatto che esse possono “obbligare” le sottoclassi ad implementare dei **comportamenti**
- Una classe che eredita un metodo astratto infatti, deve fare override del metodo ereditato oppure essere dichiarata astratta.
- Dal punto di vista della progettazione quindi, questi strumenti supportano l’astrazione dei dati.

Classi astratte vs Interfacce

- Una evidente differenza pratica è che possiamo simulare l'ereditarietà multipla solo con l'utilizzo di interfacce. Infatti, è possibile estendere una sola classe alla volta, ma implementare più interfacce.
- Tecnicamente la differenza più evidente è che un'interfaccia non può dichiarare né variabili ne metodi concreti, ma solo costanti statiche e pubbliche e metodi astratti.
- È invece possibile dichiarare in maniera concreta un'intera classe astratta (senza metodi astratti). In quel caso il dichiararla astratta implica comunque che non possa essere istanziata.

Classi astratte vs Interfacce

- Quindi una classe astratta solitamente non è altro che un'**astrazione troppo generica per essere istanziata** nel contesto in cui si dichiara.
- Un'interfaccia invece, solitamente non è una vera astrazione troppo generica per il contesto, ma semmai un'"**astrazione comportamentale**", che non ha senso istanziare in un certo contesto.

Classi astratte vs Interfacce

- Le classi astratte pure definiscono un legame più forte con la classe derivata poiché ne rappresentano il **tipo base** definendone il comportamento comune
- Le interfacce possono invece essere usate per definire un modello generico, che implementa un **comportamento comune a classi di vario genere e natura**