UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Laboratorio di Algoritmi e Strutture Dati
a.a. 2024/2025

RICHIAMI DI JAVA
CLASSI ASTRATTE ED INTERFACCE

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

La classe `Object`

- La classe **`Object`** è la **superclasse**, diretta o indiretta, di ciascuna classe in Java.
- Grazie al meccanismo dell'ereditarietà, i metodi della classe `Object` possono essere invocati su tutti gli oggetti.
- La classe `Object` definisce lo stato ed il comportamento base che ciascun oggetto deve avere e cioè l'abilità di:
  - verificare l'uguaglianza con un altro oggetto (`equals`)
  - convertirsi in una stringa (`toString`)
  - ritornare la classe dell'oggetto (`getClass`)
  - clonarsi (`clone`) – creare una nuova istanza della classe dell'oggetto corrente e inizializzare tutti i suoi campi con esattamente i contenuti dei campi corrispondenti di questo oggetto (*clonazione superficiale*)
  - …

La classe astratta

- È caratterizzata dalla parola chiave **`abstract`**, ed ha solitamente (non obbligatoriamente!) almeno un metodo astratto che è dichiarato ma non implementato (**`abstract`**).
- Una classe astratta fattorizza, dichiarandole, operazioni comuni a tutte le sue sottoclassi, ma non le definisce (implementa).
- In effetti, non viene creata per definire istanze (che non saprebbero come rispondere ai metodi "lasciati in bianco"), ma per derivarne altre classi, che dettaglieranno i metodi qui solo dichiarati.
- Una classe senza metodi astratti è definita «abstract» per non essere implementata, e costituire semplicemente una **categoria concettuale.**

Esercitazione

- Focus sul problema dell'ordinamento di array di oggetti
  - L'operazione di confronto tra coppie di oggetti dipende dal tipo degli oggetti
- Consideriamo una classe **`VettoreOrdinabile`** che serva da contenitore per degli oggetti generici, tale da poterli ordinare secondo criteri da stabilire (rif. **`VettoreOrdinabile`**)
- Osservate il seguente metodo **`ordina()`**. Quale parte del codice dovremmo adattare allo specifico tipo di oggetti contenuti nell'array?

La classe `VettoreOrdinabile` (1 di 4)

- La classe **`VettoreOrdinabile`** è una classe astratta in quanto, per poter funzionare, necessita di conoscere il criterio di ordinamento degli oggetti che deve contenere.
- Il metodo **`ordina`**`()` utilizza il metodo **`confronta()`** per stabilire l'ordinamento dei singoli oggetti.
- il metodo **`confronta()`** avrà implementazioni diverse per i diversi tipi di oggetti contenuti nell'array.

La classe `VettoreOrdinabile` (2 di 4)

- Il metodo `confronta()` è definito **astratto** in modo da obbligare la sottoclasse a implementare un metodo che svolga la funzione di confronto.
- Esso dovrà restituire:
  - un valore positivo se il primo argomento è maggiore del secondo (ovvero "segue" il secondo nella sequenza di ordinamento),
  - un valore negativo se il primo argomento è minore del secondo,
  - 0 altrimenti.

La classe `VettoreOrdinabile` (3 di 4)

```
Protected abstract int confronta (Object elemento1, Object elemento2);
 /* The protected modifier specifies that the member can only be accessed within its own package
  * (as with package-private) and, in addition, by a subclass of its class in another package
  */
public void ordina () {          //shell-sort
      int   s, i, j, num;
      Object 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 (confronta (vettore[j], vettore[j + s]) > 0) {
                  temp = vettore[j];
                  vettore[j] = vettore[j + s];
                  vettore[j + s] = temp;
               }
}
```

La classe `VettoreOrdinabile` (4 di 4)

Per testare il funzionamento della classe **`VettoreOrdinabile`** è necessario derivarne una sottoclasse che faccia riferimento a degli oggetti definiti:

- deriviamo quindi la classe **`VettorePunto`** destinata a contenere oggetti della classe **`Punto`**
- deriviamo anche la classe **`VettoreIntero`** destinata a contenere oggetti della classe **`Integer`**.

Ricapitoliamo

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

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Le interfacce (1 di 3)

- 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 (2 di 3)

- 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(3 di 3)

- 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.

Verso l'interfaccia `Ordinabile`

- Cosa hanno in comune la classe **`Punto`** e la classe **`Integer`** ?
  - 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 **`confronta`** che consenta di confrontare due oggetti della stessa classe.
  - Notare che su un insieme di questi oggetti è possibile definire una relazione di ordine totale
  - L'insieme può essere ordinato!

L'interfaccia `Ordinabile` (1 di 3)

- 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 oggetti di questa classe trasversale, senza la necessità di avere classi specializzate.

L'interfaccia `Ordinabile` (2 di 3)

Per esempio:

```
interface Ordinabile {
    public int confronta (Ordinabile obj);
}
```

- **Ricorda**: quando un oggetto è di un tipo corrispondente a un'interfaccia, significa che appartiene a una classe che implementa quell'interfaccia.

L'interfaccia `Ordinabile` (3 di 3)

- Modifichiamo la classe **`VettoreOrdinabile`** vista precedentemente in modo che possa funzionare con l'interfaccia **`Ordinabile`**.
- rif. **`Ordinabile`**

Ereditarietà multipla (1 di 3)

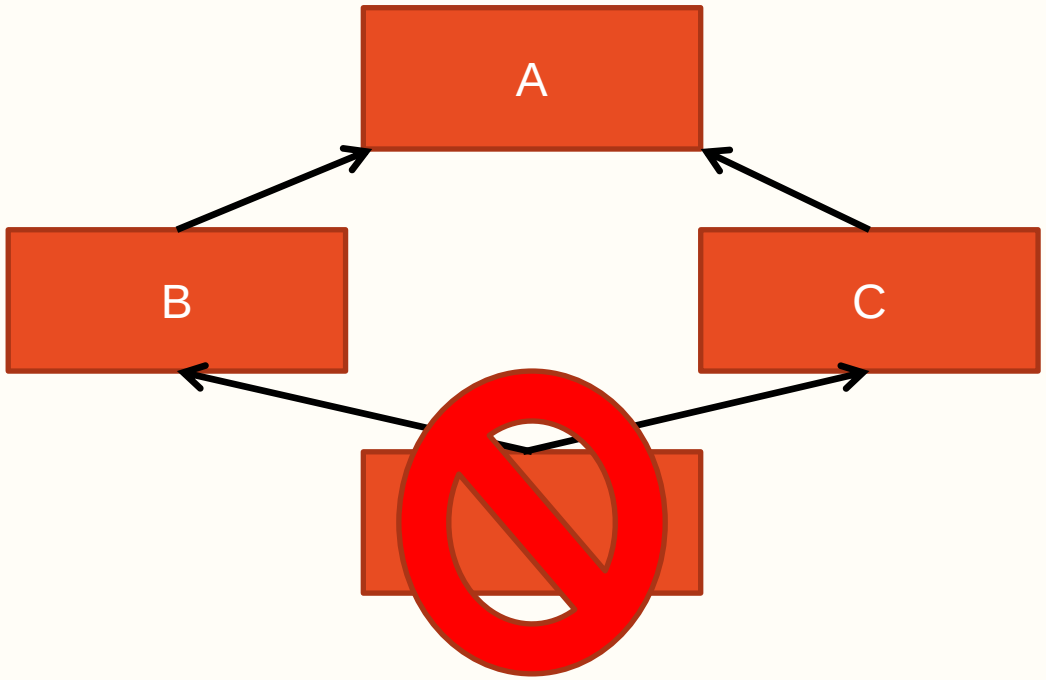
- In Java non esiste la cosiddetta “ereditarietà multipla” (come in C++)
- In pratica non è possibile scrivere:

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

- Questa permette ad una classe di estendere più classi contemporaneamente

Ereditarietà multipla (2 di 3)

- Problema del diamante



Ereditarietà multipla (3 di 3)

- L'ereditarietà multipla può essere causa di ambiguità:
  - due classi B e C ereditano dalla classe A
  - 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?
  - in particolare, cosa succede se B e C presentano due differenti implementazioni di uno stesso metodo?

Ereditarietà multipla ed interfacce

- Tale ambiguità prende il nome di **problema del diamante**, proprio 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 estendere una sola classe alla volta**, cioè ereditare i dati ed i metodi effettivi da una sola classe base
  - **può invece implementare infinite interfacce**, simulando di fatto l'ereditarietà multipla, ma senza i suoi effetti collaterali negativi.

Classi astratte vs Interfacce (1 di 4)

- 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 (2 di 4)

- Un'evidente differenza pratica è che possiamo simulare l'ereditarietà multipla solo con l'utilizzo di 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 definire 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 (3 di 4)

- 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 una "**astrazione comportamentale**", che non ha senso istanziare in un certo contesto.

Classi astratte vs Interfacce (4 di 4)

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

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Ancora sulle interfacce…

- (Come già sottolineato…) 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

Richiami: il problema dei duplicati (1 di 4)

```
public static boolean verificaDupList (LinkedList 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 (2 di 4)

```
public static boolean verificaDupOrdList (LinkedList 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 (3 di 4)

```
public static boolean verificaDupArray (LinkedList 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 (4 di 4)

```
public static boolean verificaDupOrdArray (LinkedList 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;
}
```

Verso l'interfaccia **`AlgoDup`** (1 di 4)

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

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

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

Verso l'interfaccia `AlgoDup` (2 di 4)

- 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 aiuta…
  - 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.

L'interfaccia **AlgoDup** (1 di 5)

```
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**

L'interfaccia `AlgoDup` (2 di 5)

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

L'interfaccia `AlgoDup` (3 di 5)

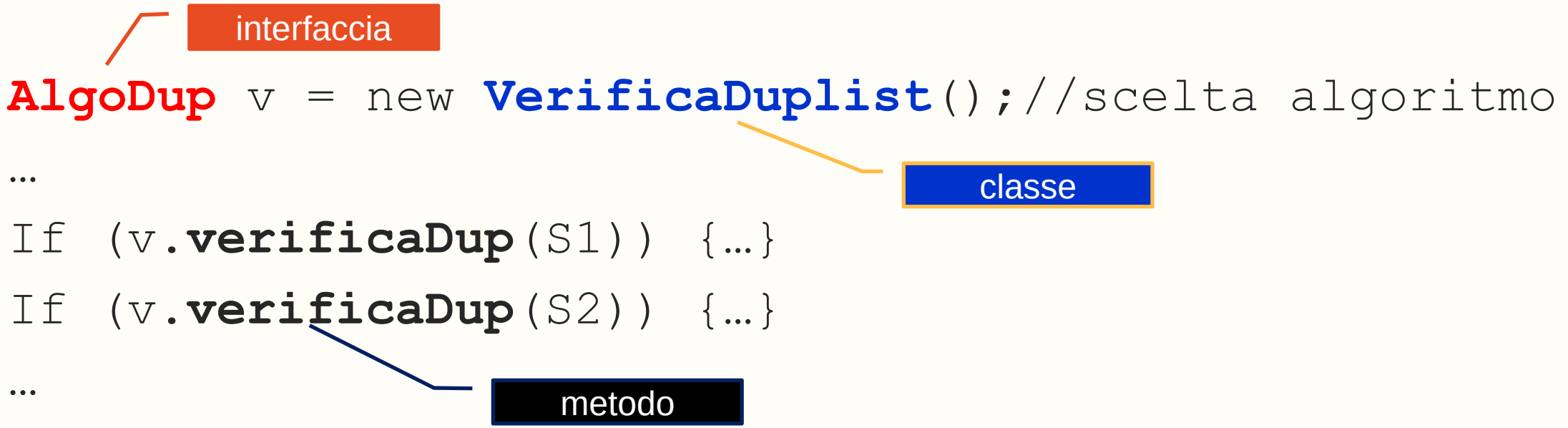
- L'implementazione dell'interfaccia obbliga il programmatore a rispettare l'intestazione del metodo **`verificaDup`** nelle varie classi
- I metodi verranno invocati nella forma generica

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

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

L'interfaccia **AlgoDup** (4 di 5)

- Decidendo la classe dell'oggetto **v**, si controlla la particolare implementazione che si intende usare



L'interfaccia **`AlgoDup`** (5 di 5)

- Per decidere quale algoritmo utilizzare basta modificare la prima linea del seguente blocco di codice:
  - `AlgoDup v = new VerificaDuplist();`
- Tutte le occorrenze di **`v.verificaDup`** restano invariate al variare della classe scelta.

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** (1 di 3)

```
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`** (2 di 3)

```
    /**
    * 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** (3 di 3)

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

    // costruttori che inizializzano x,y, e il raggio
  …
    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
```

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM