



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica



Laboratorio di Algoritmi e Strutture Dati a.a. 2022/2023

RICHIAMI DI JAVA
CLASSI ASTRATTE ED INTERFACCE

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Preliminare: la classe `Object` (1 di 2)

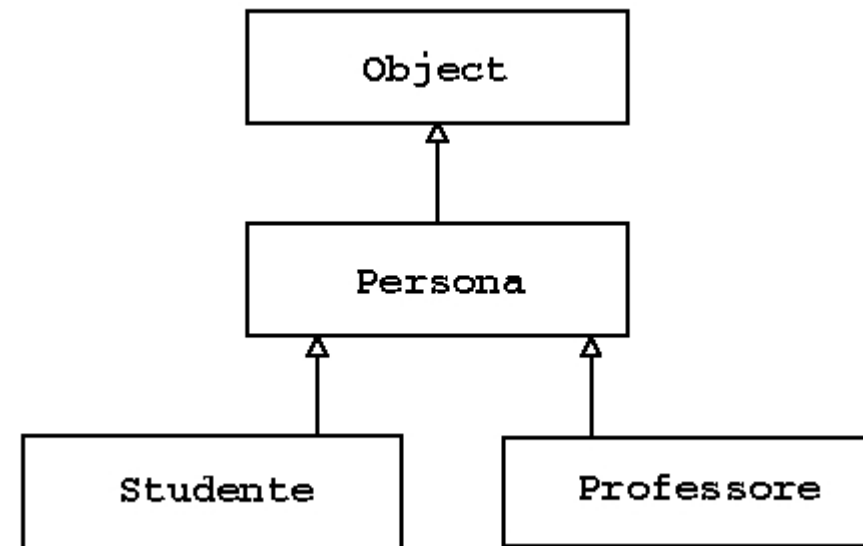
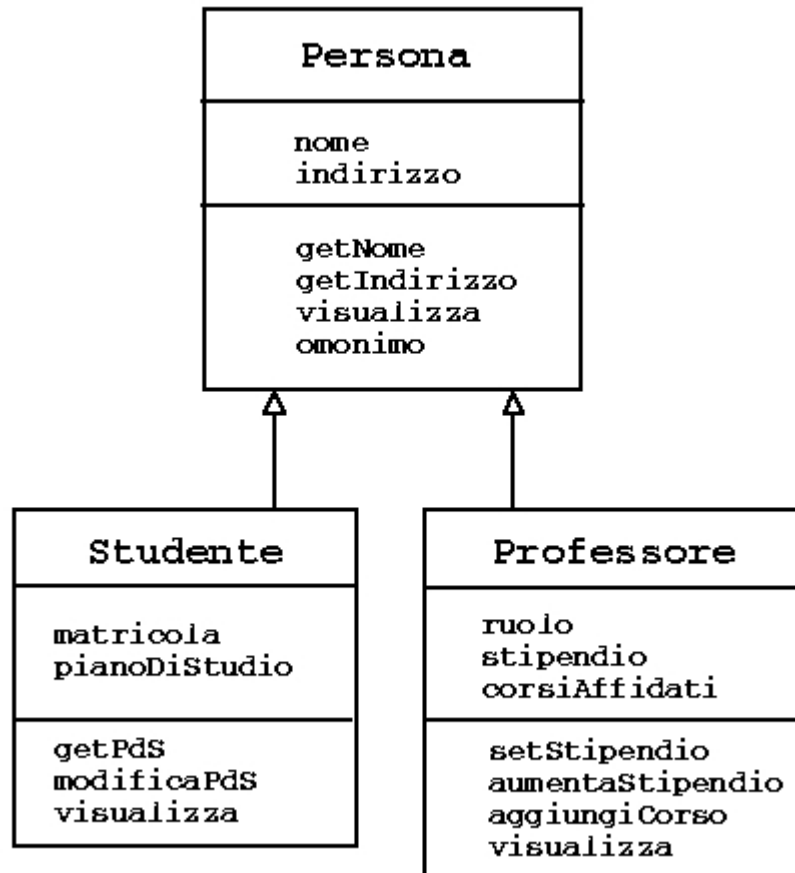
- In Java si possono definire delle gerarchie di classi arbitrariamente complesse.
- Ogni classe è una classe derivata, in modo diretto o indiretto, dalla classe `Object`.
 - Una classe che non estende un'altra classe estende automaticamente la classe `Object`.
 - La classe `Object` è la **superclasse**, 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` (2 di 2)

La classe `Object` definisce lo stato ed il comportamento base che ciascun oggetto deve avere e cioè l'abilità di:

- confrontarsi con un altro oggetto (`equals`)
- convertirsi in una stringa (`toString`)
- ritornare la classe dell'oggetto (`getClass`)
- clonarsi (`clone`) – notare che l'implementazione di default è detta *clonazione superficiale*, che semplicemente copia campo per campo
- ...

Ereditarietà: esempio (class diagram)





UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica



Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Classe astratta (1 di 3)

L'ereditarietà porta a riflettere sul rapporto fra progetto e struttura:

- Una classe può limitarsi a definire solo l'interfaccia, lasciando indefiniti uno o più metodi (**classe astratta**), che verranno poi implementati dalle classi derivate
- 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.

Classe astratta (2 di 3)

- Una **classe astratta** è simile a una classe regolare:
 - può avere attributi (tipi primitivi, istanze di oggetti, ...)
 - può avere metodi
 - è caratterizzata dalla parola chiave **abstract**, ed ha solitamente (non obbligatoriamente!) almeno un metodo che è dichiarato ma non implementato (**abstract**).

Classe astratta (3 di 3)

- **Una classe astratta può anche non avere metodi astratti**
 - in tal caso è definita astratta per non essere implementata, e costituire semplicemente una **categoria concettuale**, quindi l'imposizione della parola chiave **abstract** nella classe non implica che i metodi saranno astratti
- Se comunque almeno un metodo è **abstract**, la parola chiave **abstract** va inserita anche nella classe, pena errore di compilazione.

Esercitazione

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/vettore?

La classe VettoreOrdinabile (1 di 4)

```
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;
                }
    }
```

```
protected int confronta (Object elemento1, Object elemento2);
```

La classe `VettoreOrdinabile` (2 di 4)

- La classe `VettoreOrdinabile` è stata dichiarata 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.

La classe `VettoreOrdinabile` (3 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` (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`
 - aggiungiamo il metodo `maggiorDi` in modo da non dover trattare direttamente con le variabili d'istanza;
- deriviamo anche la classe `VettoreIntero` destinata a contenere oggetti della classe `Integer`.



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica



Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Oltre le classi astratte: le 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;  
}
```

Le interfacce (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

Le interfacce (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.

Le interfacce (3 di 4)

- 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 (4 di 4)

- 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 (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**

Le interfacce: Il problema dei duplicati (2 di 5)

- 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 (3 di 5)

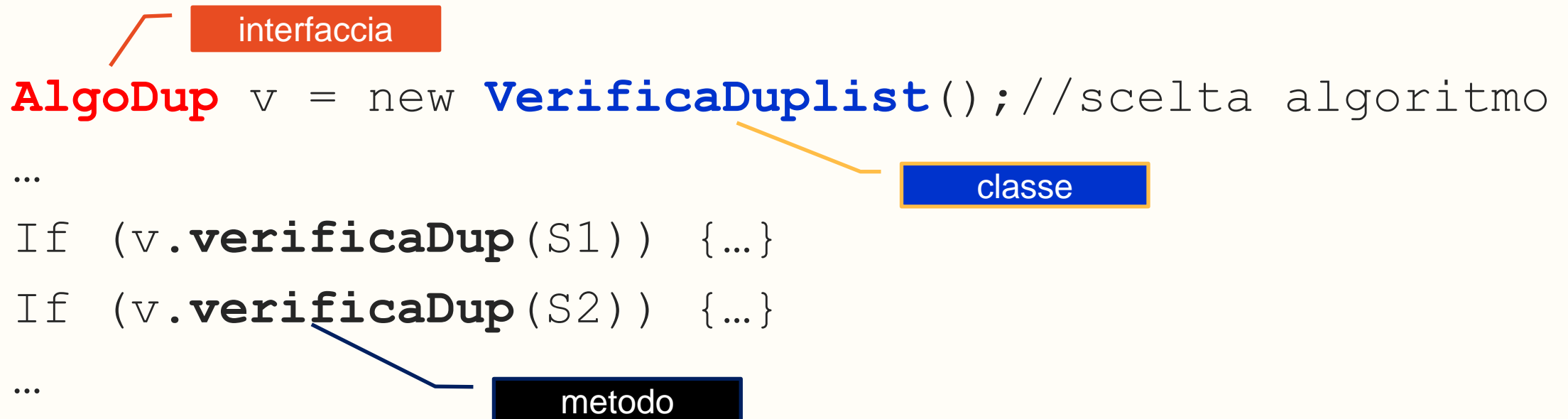
- L'implementazione dell'interfaccia obbliga il programmatore a rispettare l'intestazione del metodo **verificaDup** nelle varie classi
- I metodi verranno invocati nella forma generica

`v.verificaDup(S)`

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

Le interfacce: Il problema dei duplicati (4 di 5)

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


AlgoDup $v = \text{new } \text{VerificaDuplist}(); // \text{scelta algoritmo}$
...
If ($v.\text{verificaDup}(S1)$) { ... }
If ($v.\text{verificaDup}(S2)$) { ... }
...

Le interfacce: Il problema dei duplicati (5 di 5)

- 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.

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
```


L'interfaccia Volante (1 di 4)

```
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 (2 di 4)

```
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 (3 di 4)

```
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 (4 di 4)

Qual è il vantaggio di tale scelta?

- La 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...



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica



Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM