Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

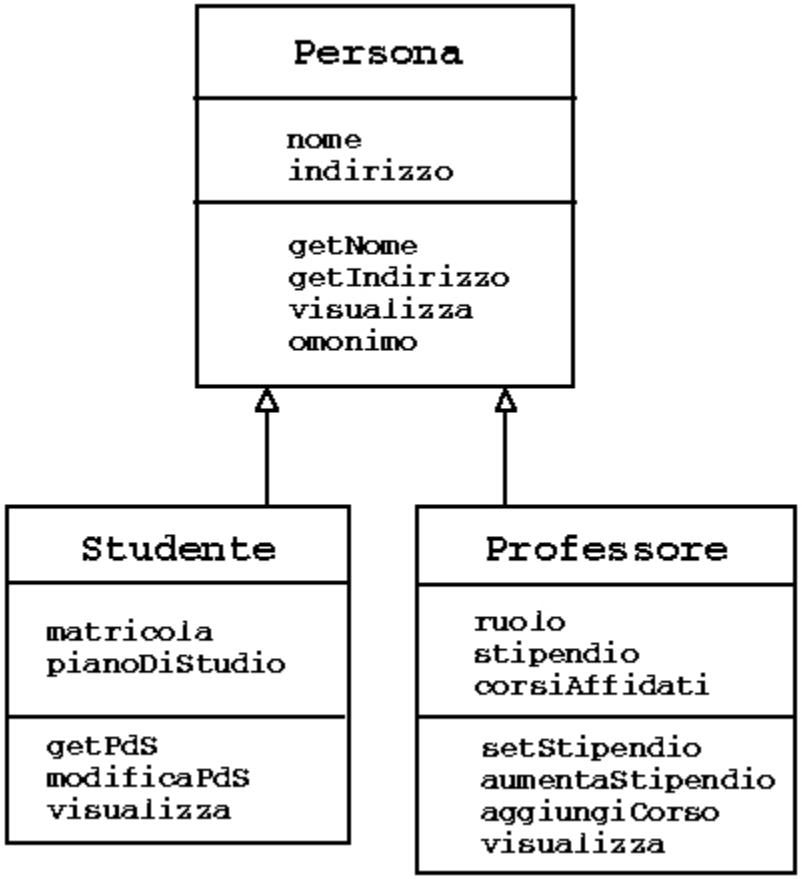
Modulo di Laboratorio di Algoritmi e Strutture Dati

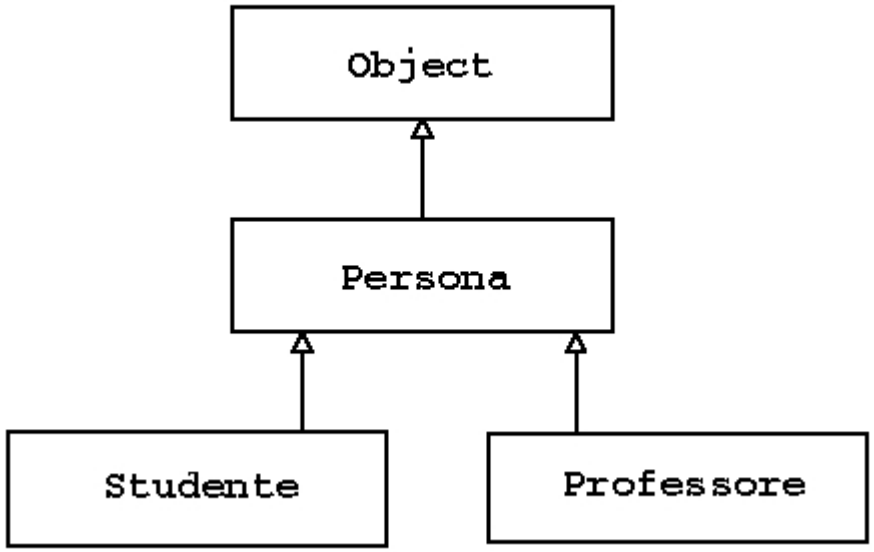
Java: Classi astratte

La classe `Object`

- La definizione di sottoclassi può essere iterata a piacere: si possono definire delle **gerarchie** di classi arbitrariamente complesse.
- In Java ogni classe è una classe derivata, in modo diretto o indiretto, dalla classe **`Object`**.
- Ciascuna classe che non estende un'altra classe estende automaticamente la classe **`Object`**.

Esempio: diagramma di classi





La classe `Object`

Ricorda che :

- la classe **`Object`** è la superclasse, diretta o indiretta, di ciascuna classe in Java
- grazie al meccanismo dell'ereditarietà, i suoi metodi possono essere invocati su tutti gli oggetti.

In particolare, alcuni metodi molto utili sono:

- **`toString()`**
- **`equals()`**

La classe `Object`

- **`public String toString()`**

  Restituisce una rappresentazione testuale dell'oggetto in forma di stringa: è molto utile ad esempio per le stampe.

- **`public boolean equals(Object obj)`**

  Verifica se l'oggetto su cui è invocato è uguale (equivalente) a quello passato per argomento.

Il metodo `toString()`

- Il metodo **`toString()`** restituisce una stringa che può essere considerata come la "**rappresentazione testuale**" dell'oggetto su cui è invocato (da usare ad esempio nella stampa).
- Poiché la classe **`Object`** non può conoscere la struttura dell'oggetto, la definizione default del metodo nella classe **`Object`** restituisce una stringa del tipo **`<classe>@<hashcode>`** dove
  - <**classe**> è il nome della classe dell'oggetto su cui il metodo è invocato
  - <**hashcode**> è la rappresentazione esadecimale del codice hash dell'oggetto (indirizzo in memoria dell'oggetto).

Il metodo `toString()`

- Il metodo **`toString()` deve quindi essere ridefinito in ogni classe che lo usa**, per ottenere un risultato significativo.
- Tipicamente, di un oggetto si vogliono stampare i valori delle variabili d'istanza, ed eventualmente una intestazione.
- Grazie all'esistenza del metodo **`toString()`**, in Java possiamo liberamente mettere una variabile di tipo riferimento in un contesto in cui ci dovrebbe essere una stringa:
  - Ad es: in un comando di stampa, oppure come argomento dell'operatore di concatenazione +
- L'oggetto verrà convertito in una stringa automaticamente invocando su di esso il metodo **`toString()`.**

Il metodo `equals()`

- Il metodo **`equals()`** implementa una **relazione d'equivalenza** sulle istanze di ogni classe, cioè una relazione **riflessiva**, **simmetrica** e **transitiva** (e soddisfa alcune altre proprietà).
- Nella pratica, l'invocazione **`o1.equals(o2)`** restituisce **`true`** quando gli oggetti **`o1`** e **`o2`**
  - sono istanze della stessa classe, e
  - hanno uguale contenuto, cioè valori equivalenti per ogni variabile d'istanza.

Il metodo `equals()`

- Nella classe **`Object`**, poichè non si può fare alcuna assunzione sulla struttura interna degli oggetti su cui viene invocato, il metodo **`equals`** è realizzato nel modo più restrittivo possibile:
  - due oggetti sono considerati equivalenti solo se sono lo stesso oggetto
- Quindi il confronto è basato sull'operatore ==

Il metodo `equals()`

- Poichè tutte le classi ereditano da **`Object`**, il metodo **`equals()`** può essere invocato su una istanza di una qualunque classe
- Se però nella classe in questione il metodo non è stato sovrascritto, verrà eseguito il metodo ereditato da **`Object`**, con risultati che potrebbero essere inattesi
- Quindi il metodo **`equals()`** deve essere ridefinito in tutte le classi in cui è necessario effettuare confronti, per ottenere risultati significativi.

Il metodo `equals()`

Osservazione:

- Quando in una classe si sovrascrive il metodo **`equals()`**, la firma del metodo **NON** può essere modificata. Il parametro del metodo **deve** essere necessariamente un **`Object`**.
- Nel corpo del metodo si dovrà quindi eseguire un **`cast`** sul parametro per convertirlo nel tipo della classe.
- Esempio: estendiamo le classi **`Persona`** e **`Studente`** sovrascrivendo il metodo **`equals()`** della classe **`Object`**.

Esempio: la classe `Persona`

```
public class Persona {
...
// override
public boolean equals(Object obj) {
  if (obj == null) return false;
  if (!(obj instanceof Persona)) return false;
  Persona p = (Persona) obj;
  return ( this.omonimo(p) &&
   this.indirizzo.equalsIgnoreCase(p.indirizzo)
   );
}
}
```

Esempio: la classe Studente

```
public class Studente extends Persona {
...
//  override
public boolean equals(Object obj) {
  if (!super.equals(obj)) return false;
  if (!(obj instanceof Studente)) return false;
  Studente s = (Studente) obj;
  return (this.pianoDiStudio.equalsIgnoreCase(
                                  s.pianoDiStudio)
    && this.matricola == s.matricola );
}
}
```

Classe astratta

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

- 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* almeno un metodo che è dichiarato ma non implementato (abstract)
- 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.

Classe astratta

- Una classe astratta **non può essere istanziata** ma può avere dei costruttori con lo scopo di inserire operazioni comuni che probabilmente saranno sfruttate dalle sottoclassi (i loro costruttori quindi chiameranno super(...))
- Una classe astratta suppone l'esistenza di almeno una sottoclasse
- Una classe che estende una astratta può fornire tutte le implementazioni necessarie; qualora ciò non avvenga, resta astratta anche se i suoi metodi non sono abstract.

Classe astratta

```
public abstract class Animale {
 public abstract String verso();
 public abstract String si_muove();
 public abstract String vive();
  ...
}

public abstract class AnimaleTerrestre
  extends Animale {
public String vive() {
return "sulla terraferma"; }
}
```

Esempio

- rif. **`VettoreOrdinato`**
- L'obiettivo è implementare una classe **`VettoreOrdinato`** che serva da contenitore per degli oggetti tale da poterli ordinare secondo criteri da stabilire.
- La classe **`VettoreOrdinato`** è stata dichiarata **astratta** in quanto, per poter funzionare, necessita di conoscere il criterio di ordinamento degli oggetti che deve contenere.

Ricorda:

- In Java le classi che non derivano esplicitamente da un'altra classe, derivano implicitamente dalla classe `Object`
- Ciò implica che qualsiasi classe Java deriva in maniera diretta o indiretta da `Object`

La classe VettoreOrdinato

```
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])) {
               temp = vettore[j];
               vettore[j] = vettore[j + s];
               vettore[j + s] = temp;
            }
}
abstract protected boolean confronta (Object elemento1,
                                      Object elemento2);
}
```

La classe `VettoreOrdinato`

- Il metodo **`ordina()`** utilizza il metodo **`confronta()`** per stabilire l'ordinamento dei singoli oggetti: questo metodo **è definito astratto in modo da obbligare la sottoclasse a implementare un metodo che svolga la funzione di confronto**.
- Esso dovrà restituire `true` se il primo argomento è maggiore del secondo (ovvero «segue» il secondo nella sequenza di ordinamento), `false` altrimenti.

La classe VettoreOrdinato

- Per testare il funzionamento della classe `VettoreOrdinato` è necessario derivarne una sottoclasse che faccia riferimento a degli **oggetti definiti**.
- Deriviamo quindi la classe `VettoreTempo` destinata a contenere oggetti della classe `Tempo`.
- Nella classe `Tempo` non abbiamo metodi che ci consentano di confrontare due tempi per stabilire un ordine. Aggiungiamo quindi il metodo **`maggioreDi`** in modo da non dover trattare direttamente con le variabili d'istanza.
- La classe Tempo finale diventa dunque la seguente:

La classe Tempo (rev)

```
// aggiungiamo alla classe Tempo il seguente metodo

public boolean maggioreDi (Tempo t) {
  if (ore > t.ore || ore == t.ore && minuti > t.minuti ||
          ore == t.ore && minuti == t.minuti && secondi >
                                                t.secondi)
         return true;
      else
          return false;
   }
} //end-classe-Tempo
```

La classe VettoreTempo

```
class VettoreTempo extends VettoreOrdinato {
  VettoreTempo () {super (10); }
  VettoreTempo (int elementi) { super (elementi);}

  protected boolean aggiungi (Object elemento) {
      return false; }
  protected boolean aggiungi (Tempo elemento) {
      return super.aggiungi(elemento); }

   protected boolean confronta (Object elemento1, Object
                                              elemento2) {
    return ((Tempo) elemento1).maggioreDi((Tempo)
                                              elemento2);
   }
}
```

La classe VettoreTempo

- In questa classe si è dichiarato un costruttore senza argomenti che dimensiona il contenitore per default a 10 elementi massimo e un costruttore che permette di stabilire la dimensione voluta.
- Per essere sicuri che questo contenitore contenga solo oggetti della classe Tempo, o oggetti di classi derivate, abbiamo sovrascritto il metodo `aggiungi(Object elemento)` in modo tale da impedire che vengano inseriti oggetti qualsiasi, e abbiamo definito il metodo `aggiungi(Tempo elemento)`.

La classe VettoreTempo

- Nel metodo `confronta` abbiamo dovuto dichiarare gli argomenti come object-id della classe `Object` anche se siamo sicuri che corrisponderanno sempre a istanze di `Tempo` poiché altrimenti il metodo non avrebbe corrisposto al metodo astratto dichiarato nella classe genitrice.
- Questo ci obbliga a usare dei cast che complicano un po' la lettura ma che possono essere usati con tranquillità senza bisogno di controlli aggiuntivi.

Test

```
class Test {
   public static void main (String argv[]) {
      VettoreTempo vt = new VettoreTempo (12);
      int ore, minuti, secondi;
  Tempo.separatore = ':';
  for (int i = 0; i < 12; i++) {
     ore = (int) (Math.random () * 24);
     minuti = (int) (Math.random () * 60);
     secondi = (int) (Math.random () * 60);
     vt.aggiungi (new Tempo (ore, minuti, secondi));
  }
  vt.ordina();
      for (int i = 0; i < vt.elementi(); i++)
        ((Tempo)vt.leggi (i)).visualizza(true);
   }
}
```