Fondamenti di Informatica C

Esercitazioni di Laboratorio / 3

http://polaris.ing.unimo.it/fiC/laboratorio.html

Ing. Francesco De Mola

demola.francesco@unimore.it

→ Ricevimento su appuntamento ←

DII, Modena – Via Vignolese
(lab. Dottorandi 1° piano)
Tel.: 059 205.61.42

DISMI, Reggio Emilia – Via
Amendola 2 (Pad. Morselli)
Tel.: 0522 52.26.60

Home Page:
http://www.agentgroup.ing.unimo.it/didattica/curriculum/francesco

Outline

- Javadoc: installazione e concetto di package
- Ereditarietà e polimorfismo
- Classi `abstract`, `interface` e classi normale con metodi vuoti
- Modificatori `protected`, `package`, `final`
- Esercizi: gerarchia di persone e di autovetture

Javadoc

http://java.sun.com/j2se/1.5.0/download.jsp

→ J2SE 5.0 Documentation (Javadoc)

- Scompattare in:
  JAVA_HOME\docs (es. c:\jdk1.5.0\docs)
- Accesso:
  JAVA_HOME\docs\api\index.html
- Configurazione Documentation in JCreator:
  Configure → Options → JDK Profiles → Edit

Package

Notazione puntata:

```
package.sottopackage.NomeClasse
```

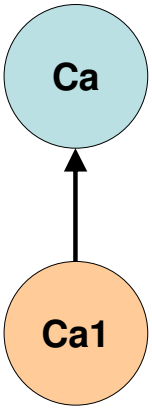
Dichiarazione di appartenenza:

```
package nomepackage;
```

Importazione package e classi:

```
import nomepackage.NomeClasse;
import nomepackage.*;
```

Ereditarietà




```
class Ca{
   int a;
   public Ca(int a){
      this.a=a;
   }
}
```

```
class Ca1 extends Ca{
   int b;
   public Ca1(int a,int b){
      super(a)
      this.b=b;
   }
}
```


Polimorfismo

```
class Ca{
   int a; ...
   op(){ a=a*2; }
}
```

Ca

```
class Ca1 extends Ca{
   int b; ...
   op(){ a=a+b; }
}
```

Ca1

```
class Ca2 extends Ca{
   int b; ...
   op(){ a=a-b; }
}
```

Ca2

```
Ca c = new Ca();
Ca1 c1 = new Ca1();
Ca2 c2 = new Ca2();
```

```
c.op();
c1.op();
c2.op();
```

```
c=c1;
c.op();
c2=c;
c2.op();
```

Classi astratte e interfacce

```
class Ca{
   int a;
   void met1(){}
   void met2(){}
}


Class Ca1
extends Ca{
   void met2(){
      a=a*2;
   }
}
```

```
abstract class Ca{
   int a;
   abstract void met1();
   void met2() {}
}


Class Ca1 extends Ca{
   void met1(){
      a=a*2;
   }
}
```

```
interface Ca{
   void met1();
   void met2();
}


Class Ca1
implements Ca{
   void met1(){
      a=a*2;
   }
   void met2(){
      a=a*2;
   }
}
```

→ `abstract` costringe la ridefinizione dei metodi (*override*)

Modificatori per l'ereditarietà

`protected`: consente **l'accesso diretto** a tutte le sottoclassi (implica che *ci si fida* di chi estende la classe base)

`private`: richiede la **definizione di metodi** appositi (implica che occorre sempre usare un metodo di accesso => *overhead*)

`final`: impedisce la ridefinizione (*override*) dei metodi nelle sottoclassi

Esercizio1: ***Gerarchia di persone (1)***

Si definisca una classe Java che rappresenta una persona.

La classe **Persona** deve possedere i seguenti campi (PROTETTI dall'esterno):

- **nome** (stringa)
- **cognome** (stringa)

Inoltre, mette a disposizione un metodo **info** per ritornare una stringa che riporta il nome e il cognome.

Si definiscano poi due classi che rappresentano rispettivamente studenti e lavoratori (continua...)

Esercizio1: ***Gerarchia di persone (2)***

La classe **Studente** estende la classe **Persona** e aggiunge i seguenti campi:

- **matricola** (intero)
- **corso di laurea** (stringa)

e ridefinisce il metodo info per riportare anche la matricola e il corso di laurea.

La classe **Lavoratore** estende la classe **Persona** e aggiunge i seguenti campi:

- **codice** (intero)
- **azienda** (stringa)

e ridefinisce il metodo info per riportare anche il codice e l'azienda in cui lavora.

Esercizio1: ***Gerarchia di persone (3)***

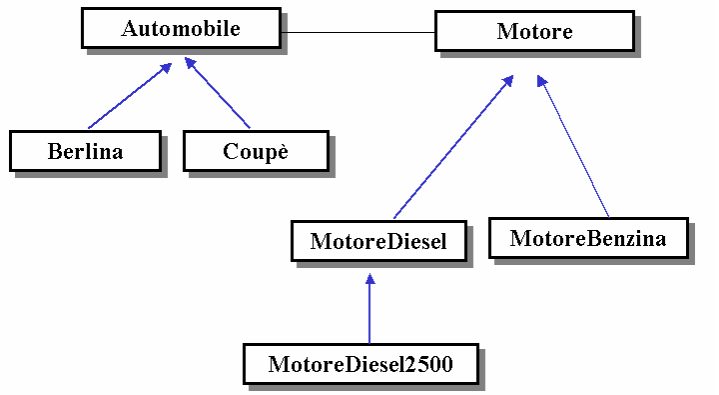
Infine si definisca un main che crea diversi oggetti di classe persona, Studente e Lavoratore, e ne mantiene i riferimenti in un array. Dopo, chiede le informazioni di ogni elemento dell'array e le stampa a video.

Esercizio2: ***Gerarchia di classi per la gestione di automobili e dei relativi motori (1)***

Si vuole implementare un'insieme di classi sufficiente a descrivere un set di **automobili,** con particolare riferimento a due proprietà fondamentali: il suo **motore** e l'insieme di **optional** con i quali viene fornita.

Non è nota a priori la *tipologia delle automobili* che verranno trattate dal programma, così come risulta sconosciuto il *tipo di motore* che un'automobile può montare.

Esercizio2: Gerarchia di classi per la gestione di automobili e dei relativi motori (2)



Esercizio2: Gerarchia di classi per la gestione di automobili e dei relativi motori (3)

Realizzare il package 'motori' con le classi riferite ai motori. Questi devono contenere informazioni su:
- cilindrata
- cilindri
- cavalli

La *potenza* (cavalli, calcolati in base ai primi due) e gli *RPM massimi* sono definiti in modo diverso in base al fatto che si tratti di un motore diesel o benzina.

Realizzare il package 'auto' con le classi riferite alle auto e gli optional. Ogni auto deve prevedere:
- targa
- marca
- modello
- motore
- numero variabile di optional

Un optional deve contenere codice, descrizione, valore e prevedere un metodo toString che restituisca i dati in un'unica stringa (per velocizzarne la stampa).

Si implementi un metodo main che testa l'utilizzo e il funzionamento di tali classi.