TEHNOLOGII JAVA
PENTRU DEZVOLTAREA APLICAȚIILOR
LUCRARE DE LABORATOR 8

Clase si metode abstracte in Java

I. SCOPUL LUCRĂRII

Lucrarea de față are rolul de a prezenta şi familiariza studentul cu noțiunea fundamentală a limbajului Java - clase abstracte, cu importanța şi utilizarea corectă a acestora.

La sfârşitul acestei lucrări, studentul va avea posibilitatea să scrie programe Java în care să utilizeze noțiunile învățate.

II. NOȚIUNI TEORETICE

1. Clase abstracte

Vezi Curs 8, pentru consultarea noțiunilor teoretice.

Cateodată avem nevoie să declarăm o clasă, dar nu ştim cum să definim toate metodele care aparțin clasei. De exemplu, sa încercăm să declarăm o clasă numită **Mamifer** în care să includem o metoda membru numită **MarcheazaTeritoriul()**. Oricum, nu ştim cum să scriem **MarcheazaTeritoriul**() pentru ca aceasta se întâmplă diferit în funcție de specia de Mamifer. Bineînțeles, o să rezolvăm acest lucru derivând subclase din clasa **Mamifer**, clase cum ar fi **Maimuta** şi **Om**. Dar ce cod să conțină funcția **MarcheazaTeritoriul**() din clasa **Mamifer**?

În Java funcția **MarcheazaTeritoriul()** se poate declara în clasa **Mamifer** ca fiind o metodă **abstract**. Făcând aceasta se permite declararea metodei fără a scrie cod pentru subclase. Se va scrie însă cod în subclase.

Dacă o metodă este declarată **abstract,** atunci şi clasa trebuie declarată **abstract**. Pentru **Mamifer** şi subclasele sale, acest lucru înseamnă că trebuie să arate după cum urmează:

```
abstract class Mamifer {
    abstract void MarcheazaTeritoriul();
}
public class Om extends Mamifer {
    public void MarcheazaTeritoriul() {
       //cod prin care se marcheaza teritoriul construind un gard
    }
}
```

```
public class UnMembruAlGastii extends Mamifer {
    public void MarcheazaTeritoriul() {
        //cod prin care se marcheaza teritoriul cu graffiti
    }
}
public class Maimuta extends Mamifer {
    public void MarcheazaTeritoriul() {
      //cod prin care se marcheaza teritoriul cu frunze si crengi
    }
}
```

În declarațiile precedente, clasa **Mamifer** nu conține nici un fel de cod pentru **MarcheazaTeritoriul()**. Clasa **Om** poate conține cod despre cum omul marchează teritoriul construind un gard în jurul lui, în timp ce clasa **UnMembruAlGastii** poate conține cod despre cum acesta îşi marchează teritoriul pictând cu spray graffiti. Clasa **Maimuta** işi va marca teritoriul cu ajutorul crengilor (sau altfel).

1.1. Clase abstracte: Interfețe

Tipic, o clasă abstractă va avea câteva metode declarate abstract şi altele care nu sunt declarate astfel. Dacă se declară o clasă care este în totalitate abstractă, atunci se declară ceea ce în Java este cunoscut sub numele de **interfață**. O interfață este o clasă abstractă în întregime. Se pot deriva clase dintr-o interfață într-o manieră complet analoagă cu aceea a derivării claselor din alte clase.

Ca exemplu, să presupunem că dezvoltăm o aplicație care trebuie să afişeze ora. Utilizatorii vor avea două opțiuni pentru a obține această informație. Pot să o preia dintr-un ceas electronic sau dintr-un ceas cu limbi. S-ar putea implementa ca mai jos:

```
interface Ceas {
    public String CitesteTimpul(int ora);
}
class CeasCuLimbi implements Ceas {
    public String CitesteTimpul(int ora) {
        StringBuffer str = new StringBuffer();
        for (int i=0; i < ora; i++)
            str.append("CeasCuLimbi ");
        return str.toString();
    }
}
class CeasElectronic implements Ceas {
    public String CitesteTimpul(int ora) {
        return new String("Este ora " + hour + ":00");
    }
}
```

In acest exemplu, **Ceas** este ***o interfață*** care furnizează o singură funcție, **CitesteTimpul()**. Aceasta înseamnă că orice clasă care este derivată (sau, în alte cuvinte, **implementată** - *implements*) din interfața Ceas trebuie să implementeze o funcție **CitesteTimpul()**. **CeasCuLimbi** este un exemplu de clasă care implementează **Ceas**, şi săi observăm că în loc de **class CeasCuLimbi extends Ceas**, sintaxa care ar fi trebuit

folosită în cazul în care **Ceas** ar fi fost o clasă abstractă, este folosita sintaxa **class CeasCuLimbi implements Ceas**.

Pentru că **CeasCuLimbi** implementează interfața **Ceas**, ea furnizează o funcție **CitesteTimpul**(). Clasa **CeasElectronic** implementează de asemenea **Ceas** şi furnizează o funcție **CitesteTimpul**().

Interfețele şi superclasele nu se exclud reciproc. O clasă nouă poate fi derivată dintr-o superclasă şi poate implementa una sau mai multe interfețe. Acest lucru se poate efectua ca mai jos, pentru o clasă care implementează doua interfețe şi are o superclasă:

```
class SubClasaMea extends SuperClasaMea implements
                          PrimaInterfata, ADouaInterfata
{
    // implementarea clasei
}
```

Pentru că este posibil ca o clasă să implementeze mai mult de o interfață, interfețele reprezintă o alternativă pentru moştenirea multiplă, care nu este permisă în Java.

1.2. Metode abstracte

Metodele abstracte sunt metode care nu au corp de implementare. Ele sunt declarate numai pentru a forța subclasele, care se vor instanția, să implementeze metodele respective.

Metodele abstracte trebuie declarate numai în interiorul claselor care au fost declarate abstracte. Altfel compilatorul va semnala o eroare de compilare. Orice subclasă a claselor abstracte care nu este declarată abstractă trebuie să ofere o implementare a acestor metode, altfel va fi generată o eroare de compilare.

Prin acest mecanism ne asigurăm că toate instanțele care pot fi convertite către clasa care conține definiția unei metode abstracte au implementată metoda respectivă dar, în acelaşi timp, nu este nevoie să implementăm în nici un fel metoda chiar în clasa care o declară, deoarece nu ştim pe moment cum va fi implementată.

O metodă statică nu poate fi declarată şi abstractă pentru că o metodă statică este implicit finală şi nu poate fi rescrisă.

III. MODUL DE LUCRU

1. Se editează codul sursă al programului Java folosind un editor de text disponibil(de ex., se poate utiliza Notepad).
2. Se salvează fişierul cu extensia **.java**. Fişierul trebuie salvat la următoarea locație: **c:\JBulider7\jdk1.3.1\bin**
3. Compilarea mini-aplicației Java se va face din linia de comandă. Se poate proceda astfel. Se deschide o fereastră MS-Dos: Start ->Run, se tipăreşte **command** în căsuța de text şi se apasă butonul OK. Printr-o schimbare de directoare şi subdirectoare se trece la locația: **c:\JBulider7\jdk1.3.1\bin.** Sau, se lansează WindowsCommander. Se trece la locația **c:\JBulider7\jdk1.3.1\bin.** Se deschide o fereastră MS-Dos: Commander ->Run Dos.

4. Pentru compilare, se tipăreşte la prompter **javac nume_fişier_sursă.java** şi se apasă Enter. De ex., dacă fişierul se numeşte Test.java, se va scrie **javac Test.java**. În cazul în care programul conține erori acestea vor fi semnalate şi afişate în fereastra MS-Dos, după care va apare iar prompter-ul. Dacă programul nu conține erori şi compilarea se face cu succes, atunci va apare numai prompter-ul.
5. Pentru rularea aplicației Java, se lansează interpretorul Java. Se tipăreşte la prompter următoarea comandă **java nume_clasă_care_conține_main** şi se apasă Enter. De ex., dacă clasa din program care conține metoda *main( )* se numeşte Test, se va scrie **java Test**.

IV. TEMĂ

1. Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic si explicând rezultatele obținute.
2. Implementați clasa **Mamifer** din cadrul exemplului de mai sus astfel:
   - scrieți codul pentru implementarea metodei: public void **MarcheazaTeritoriul**(), în fiecare din clasele: **Om**, **UnMembruAlGastii**, **Maimuta**;
   - încercați să apelați metoda: public void **MarcheazaTeritoriul**(), în cadrul claselor, şi din afara lor.

   Explicați, la fiecare caz în parte, rezultatele obținute.