Programare orientată pe obiecte

#14 JAVA Polimorfism

https://www.runceanu.ro/adrian/activitate-didactica/

Curs 14

Polimorfism

Polimorfism

1. Polimorfism
2. Legarea statica si legarea dinamica
3. Metode si clase finale
4. Metode si clase abstracte
5. Exemple de utilizare a polimorfismului, a claselor si metodelor finale si abstracte

Polimorfism

- **Polimorfismul** reprezinta capacitatea unui obiect de a aparea sub diferite forme.
- De exemplu, in lumea reala, **apa** apare:
  - sub forma solida
  - sub forma lichida
  - sau sub forma gazoasa
- In **Java**, polimorfismul inseamna ca *o singura variabila referinta x de tipul unei superclase poate fi folosita pentru a construi diferite obiecte (instante) ale claselor derivate, direct sau indirect din acea superclasa, in momente diferite ale executiei unui program*.

- Cand variabila referinta *x* este folosita pentru a apela o metoda a unui obiect apartinand unei clase derivate, metoda adecvata care va fi selectata depinde de tipul obiectului pe care variabila referinta *x* il indica in acel moment.

- Se spune ca **variabila referinta** este **polimorfica**.

- De exemplu, sa presupunem ca s-au definit clasele: *AnimalDeCasa*, *Pisica*, *Caine.*
- Se doreste sa se afiseze la ecran (programul Polimorfism.java) caracteristicile principale ale claselor *Pisica* si *Caine*, atunci cand instanţele lor sunt înfometate.

```
class AnimalDeCasa
{
    private boolean stareSuparare;
    public boolean flamand;
    protected void hranesc()
    {
        System.out.println("Nu se cunoaste");
    }
```

```
public void caracteristici()
  {
        System.out.println("Caracteristici
   necunoscute");
  }
 public boolean getStare()
 {
        return stareSuparare;
 }
 public void setStare(boolean stare)
 {
        stareSuparare = stare;
 }
}
```

```
class Pisica extends AnimalDeCasa
{
   public void caracteristici()
   {
         String starePisica;
         if (getStare() == true) starePisica = "miauna";
         else starePisica = "nu miauna";
         if (flamand == true)
            System.out.println("Pisica " + starePisica + ". Este
    flamanda.");
       else
            System.out.println("Pisica " + starePisica + ". Este
    satula.");
   }
```

```
public void hranesc()
  {
      if (flamand==true)
      {
          System.out.println("Pisica mananca lapte.");
         flamand = false;
         setStare(false);
      }
       else
           System.out.println("Pisica a mancat deja.");}
}
```

```
class Caine extends AnimalDeCasa
{
   public void caracteristici()
   {
         String stareCaine;
         if (getStare() == true) stareCaine = "latra";
         else  stareCaine = "nu latra";
         if (flamand == true)
          System.out.println("Cainele " + stareCaine + ". Este
    flamand.");
         else
          System.out.println("Cainele " + stareCaine + ". Este
    satul.");
   }
```

```
public void hranesc()
    {
        if (flamand==true)
        {
            System.out.println("Cainele mananca oase.");
          flamand = false;
          setStare(false);
            }
        else
            System.out.println("Cainele a mancat deja.");}
}
```

```
public class Polimorfism{
   public static void main(String args[]){
            AnimalDeCasa a = new Pisica();
            a.flamand = true;
            a.setStare(true);
            System.out.println("Caracteristicile primului animal de casa: ");
            a.caracteristici();
            a.hranesc();
            a.caracteristici();
            a.hranesc();
            a.caracteristici();
            a = new Caine();
            a.flamand = true;
            a.setStare(true);
            System.out.println("Caracteristicile celui de al doilea animal de casa: ");
            a.caracteristici();
            a.hranesc();
            a.caracteristici();
   }
}
```

Pe baza celor prezentate mai sus, programul va afisa urmatoarele rezultate:

Caracteristicile primului animal de casa:
Pisica miauna. Este flamanda.
Pisica mananca lapte.
Pisica nu miauna. Este satula.
Pisica a mancat deja.
Pisica nu miauna. Este satula.
Caracteristicile celui de al doilea animal de casa:
Cainele latra. Este flamand.
Cainele mananca oase.
Cainele nu latra. Este satul.

In concluzie:

- pe de o parte, **mostenirea** permite tratarea unui obiect ca fiind de *tipul propriu sau de tipul de baza* (din care este derivat tipul propriu).
- Aceasta caracteristica permite mai multor tipuri (derivate din acelasi tip de baza) sa fie tratate ca si cum ar fi un singur tip, ceea ce face ca **aceeasi secventa de cod sa fie folosita de catre toate aceste tipuri**.
- In cazul din exemplul prezentat, clasa *Pisica* si clasa *Caine* au doua atribute (*stareSuparare* si *flamand*) si doua metode *setStare()* si *getStare()* mostenite de la superclasa *AnimalDeCasa* din care deriva.

- pe de alta parte, **polimorfismul** permite unui tip de obiect sa exprime distinctia fata de un alt tip de obiect similar, atata timp cat amandoua sunt derivate din aceeasi superclasa.

- Distinctia este exprimata prin *redefinirea (suprascrierea) metodelor* care pot fi apelate prin intermediul unei variabile-referinta de tipul superclasei (in exemplul nostru este vorba despre metodele *caracteristici()* si *hranesc()*).

Polimorfism

1. Polimorfism
2. Legarea statica si legarea dinamica
3. Metode si clase finale
4. Metode si clase abstracte
5. Exemple de utilizare a polimorfismului, a claselor si metodelor finale si abstracte

Legarea statica si legarea dinamica

- Conectarea unui apel de metoda de un anumit corp de metoda poarta numele de **legare (binding).**
- Cand legarea are loc inainte de rularea programului respectiv (adica, in faza de compilare) spunem ca este vorba despre o **legare statica(early binding).**
- Termenul este specific programarii orientate pe obiecte.
- In programarea procedurala (limbajele C si Pascal) notiunea de legare statica nu exista, pentru ca toate legaturile se fac in mod static.

- Cand legarea are loc in faza de executie a programului respectiv, spunem ca este vorba despre o legare tarzie (late binding) sau **legare dinamica**.

- Legarea tarzie permite determinarea in faza de executie a tipului obiectului referit de o variabila referinta si apelarea metodei specifice acestui tip (in exemplul prezentat, metodele *caracteristici()* si *hranesc()* din clasa *Pisica* sau din clasa *Caine*).

- *Polimorfisul apare doar atunci cand are loc legarea tarzie a metodelor.*

Polimorfism

1. Polimorfism
2. Legarea statica si legarea dinamica
3. Metode si clase finale
4. Metode si clase abstracte
5. Exemple de utilizare a polimorfismului, a claselor si metodelor finale si abstracte

Metode si clase finale

- **Metodele finale** sunt *acele metode care nu pot fi redefinite niciodata intr-o subclasa*.
- Acestea sunt declarate folosind modificatorul ***final***.
- Metode finale sunt folosite pentru a mari viteza de executie a aplicatiei care le foloseste.
- In mod normal, atunci cand interpretorul Java apeleaza o metoda, el cauta metoda mai intai in clasa curenta, dupa aceea in superclasa si “urca” mai departe in ierarhie pana ce gaseste definitia acesteia.
- Prin acest proces se pierde din viteza in favoarea flexibilitatii si a usurintei in dezvoltare.

- In cazul metodelor finale legarea este statica, la compilare, si nu in timpul executiei aplicatiei.
- Astfel, compilatorul **Java** poate introduce codul executabil (bytecode-ul) al metodei in locul instructiunii de apel a acesteia in cadrul programelor care o apeleaza.

Nota:

- Metodele cu acces de tip *private* sunt implicit si de tip *final*, deoarece ele nu pot fi suprascrise in nici o situatie.

- **Clasele finale** sunt *acele clase din care nu se pot deriva subclase*.
- Acestea sunt declarate folosind modificatorul ***final***.
- Clasele finale sunt folosite pentru a mari viteza de executie a aplicatiei care le foloseste.
- Majoritatea claselor mai des folosite din bibliotecile de clase **Java** sunt clase finale, cum ar fi: *java.lang.String, java.lang.Math, java.lang.Integer* etc.

Nota: Toate metodele dintr-o clasa de tip final sunt automat finale.

Polimorfism

1. Polimorfism
2. Legarea statica si legarea dinamica
3. Metode si clase finale
4. Metode si clase abstracte
5. Exemple de utilizare a polimorfismului, a claselor si metodelor finale si abstracte

Metode si clase abstracte

- **O metoda abstracta** este o *metoda din superclasa care are sens doar pentru clasele derivate direct din superclasa*, *nu are implementare (nu are corp) ci numai antet, in superclasa si care in mod obligatoriu trebuie definita (completata cu corpul ei) in clasele derivate (altfel rezulta eroare de compilare).*

- O metoda abstracta este declarata cu modificatorul ***abstract***.

- *Intr-o ierarhie de clase, cu cat clasa se afla pe un nivel mai inalt, cu atat definirea sa este mai abstracta*.

- O clasa aflata ierarhic deasupra altor clase poate defini doar atributele si comportamentul comune celor aflate sub ea pe ramura respectiva a ierarhiei.

- In procesul de organizare a unei ierarhii de clase, se poate descoperi, uneori, cate o *clasa care nu se instantiaza direct* (adica din ea nu se pot crea direct obiecte).

- De fapt, aceasta serveste, doar ca loc de pastrare a unor metode si atribute pe care le folosesc in comun subclasele sale.

- Acesta clasa se numeste **clasa abstracta** si este creata folosind modificatorul ***abstract***.

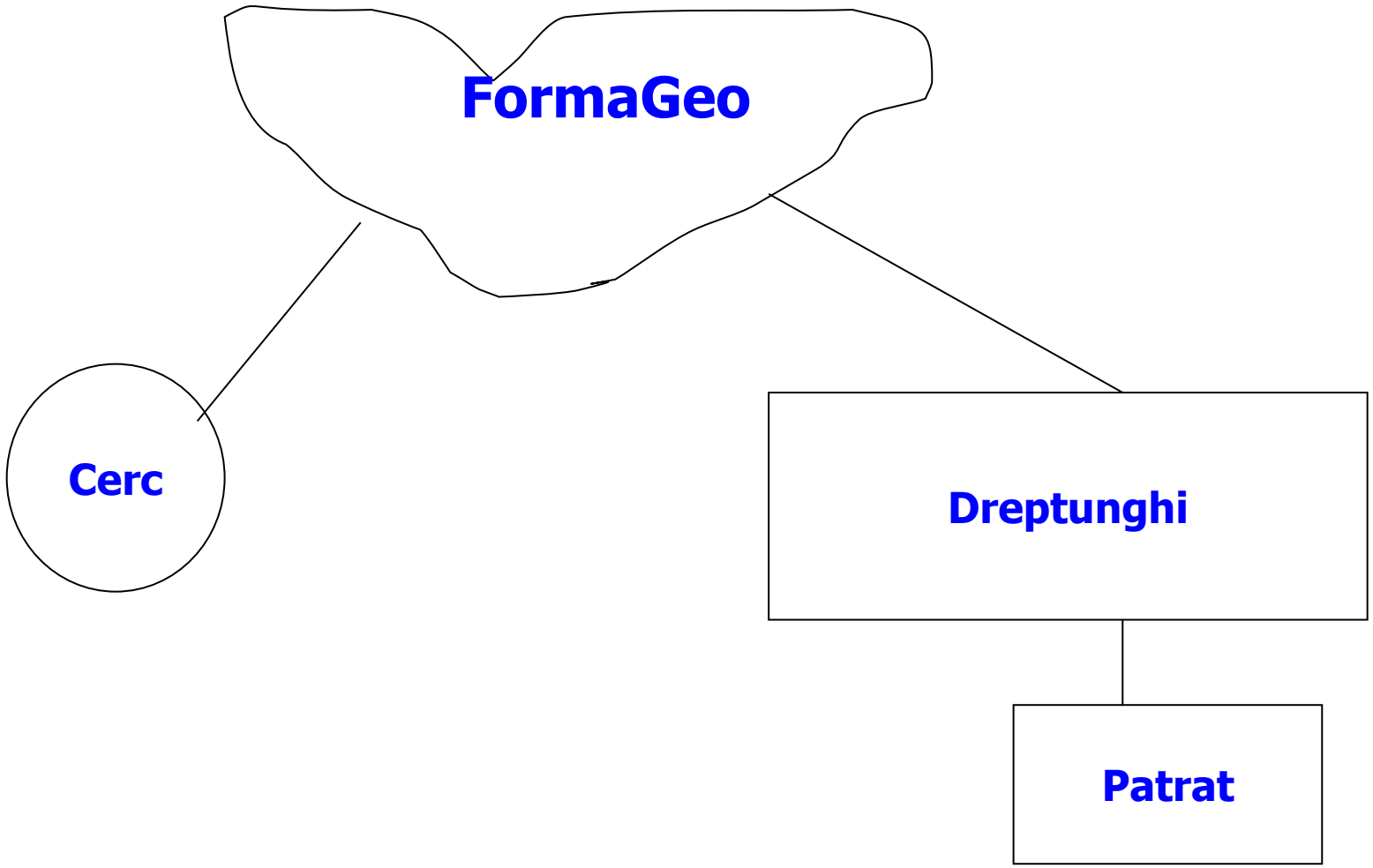
- *Clasele abstracte pot contine aceleasi tipuri de membri ca o clasa normala, inclusiv metodele-constructor, deoarece subclasele lor pot avea nevoie sa mosteneasca aceste metode-constructor*.
- Clasele abstracte pot contine, de asemenea, metode abstracte.
- *O clasa care are cel putin o metoda abstracta este o clasa abstracta*.
- Nu se poate declara o metoda abstracta intr-o clasa non-abstracta.

Polimorfism

1. Polimorfism
2. Legarea statica si legarea dinamica
3. Metode si clase finale
4. Metode si clase abstracte
5. Exemple de utilizare a polimorfismului, a claselor si metodelor finale si abstracte

Exemplu de folosire a mostenirii, a polimorfismului, a claselor si metodelor finale si abstracte:

- Un *exemplu simplu de clasa abstracta este clasa formelor geometrice*.
- Din clasa formelor geometrice pot deriva forme specifice cum ar fi: cerc, dreptunghi.
- Se poate, apoi, deriva clasa patratului ca un caz particular de dreptunghi.


FormaGeo
Cerc
Dreptunghi
Patrat

- Clasa *FormaGeo* poate sa aiba membri care sa fie comuni pentru toate subclasele, cum ar fi, de exemplu tipul formei geometrice, afisarea caracteristicilor formelor geometrice concrete (cerc, dreptunghi, etc).
- De asemenea, clasa poate defini metode care se aplica fiecarui obiect in parte, cum ar fi, calculul ariei unui obiect oarecare (dreptunghi, cerc, etc).
- In consecinta, metoda *arie()* va fi declarata ca abstracta.

- Clasa *FormaGeo* fiind abstracta nu se poate instantia.
- Deci, nu se va putea crea un obiect de tip *FormaGeo,* ci numai obiecte din clasele derivate.
- Totusi, o referinta de tip *FormaGeo* poate sa refere orice forma concreta derivata, cum ar fi: *Dreptunghi, Cerc, Patrat*.

- Fisierul-sursa urmator (FormaGeo.java) prezinta clasa abstracta *FormaGeo*.
- Constructorul acestei clase nu va fi apelat niciodata direct, deoarece *FormaGeo* este o clasa abstracta.
- Avem totusi nevoie de constructor, care sa fie apelat din clasele derivate pentru a initializa atributul *nume* de tip *private* care specifica tipul figurii geometrice.
- Metoda cu numele *arie()* este o metoda abstracta, deoarece nu putem furniza nici un calcul implicit al ariei pentru o clasa derivata care nu isi defineste propria metoda de calcul a ariei.

- Metoda *maiMic()* de comparatie a ariei unui obiect curent de tip *FormaGeo* cu un alt obiect de tip *FormaGeo* (preluat prin parametrul *rhs*) nu este abstracta, deoarece ea poate fi aplicata la fel pentru toate clasele derivate.
- De fapt, definirea ei este invarianta de-a lungul ierarhiei de aceea am declarat-o cu tipul *final*.
- Variabila *rhs* poate sa refere orice instanta a unei clase derivate din *FormaGeo* (de exemplu, o instanta a clasei *Dreptunghi*).
- Astfel, este posibil sa folosim aceasta metoda pentru a compara aria obiectului curent (care poate fi, de exemplu, o instanta a clasei *Cerc*) cu aria unui obiect de alt tip, derivat din *FormaGeo*.
- Acesta este un exemplu de folosire a polimorfismului.

Observatie:

- Vom face cateva precizari asupra folosirii metodei *toString()* in orice clasa in care se doreste afisarea starii instantelor unei clase.
- In general, afisarea starii unui obiect se face utilizand metoda *print()* din clasa *System.out*.
- Pentru a putea face acest lucru, trebuie ca in clasa obiectului care se doreste a fi afisat sa existe o metoda cu numele de *toString()*.
- Aceasta metoda trebuie sa intoarca o valoare de tip *String* (reprezentand starea obiectului) care poate fi afisata.
- Astfel, in cazul nostru, am definit o metoda *toString* care permite sa afisam un obiect oarecare de un tip derivat din clasa *FormaGeo* folosind instructiunea *System.out.println()*.
- Practic, compilatorul **Java** apeleaza automat toString() pentru fiecare obiect care se afiseaza cu metoda *print().*

```
/*
 Superclasa abstracta pentru forme FormeGeo

 CONSTRUIREA: nu este permisa, FormaGeo fiind
    abstracta.
 Constructorul cu un parametru este necesar ptr.
    Clasele derivate.
 ------------------ metode publice --------------------
 double arie()       --> Intoarce aria (abstracta)
 boolean maiMic     --> Compara doua forme dupa
    arie
 String toString     --> Metoda uzuala pentru afisare
 */
```

```
abstract class FormaGeo  {
    private String nume;
    abstract public double arie();
    public FormaGeo(String numeForma)  {
        nume = numeForma;
    }
    final public boolean maiMic(FormaGeo rhs)  {
        return arie() < rhs.arie();
    }
    final public String toString()  {
        return nume + ", avand aria " + arie();
    }
}
```

Pe baza clasei abstracte *FormaGeo* definite ne propunem sa rezolvam urmatoarea problema:

Se citesc N forme geometrice (patrate, dreptunghiuri, cercuri) de la tastatura. Sa se afiseze formele geometrice ordonate dupa arie.

Mai intai, trebuie sa definim subclasele *Cerc, Dreptunghi si Patrat*.

Definirea lor se face distinct in fisiere-sursa separate (Cerc.java, Dreptunghi.java si Patrat.java).

```
/* clasa Cerc derivata din FormaGeo
CONSTRUCTORI: cu raza ptr. cerc
------------------- metode publice --------------------
double arie()-->Implementeaza metoda abstracta din
   FormaGeo
*/
public class Cerc extends FormaGeo  {
    private double raza;
    public Cerc(double rad)  {
          super("Cerc");
          raza = rad;
    }
    public double arie()  {
          return Math.PI * raza * raza;}
}
```

```
/* clasa Dreptunghi; derivata din FormaGeo
 * CONSTRUCTORI:  cu lungime si latime ptr. dreptunghi
 * ------------------ metode publice --------------------
 * double arie()-->Implementeaza metoda abstracta din FormaGeo
*/
public class Dreptunghi extends FormaGeo {
      private double lungime;
      private double latime;
      public Dreptunghi(double lg, double lat)  {
              this(lg, lat, "Dreptunghi");
      }
      Dreptunghi(double lg, double lat, String nume)  {
             super(nume);
             lungime = lg;
             latime = lat;
      }
      public double arie()  {
              return lungime * latime;
      }
}
```

```
/* clasa Patrat; derivata din Dreptunghi
 * CONSTRUCTORI: cu latura ptr. patrat
 * ---------------- metode publice -----------------
 * double arie()-->Implementeaza metoda abstracta
   din FormaGeo
 */
public class Patrat extends Dreptunghi
{
    public Patrat(double latura)
    {
        super(latura, latura, "Patrat");
    }
}
```

Observatie:

- Clasa *Patrat* mosteneste de la clasa *Dreptunghi* metoda *arie()* si de aceea nu o mai defineste in interiorul ei.
- Pentru rezolvarea problemei ordonarii formelor geometrice dupa aria lor, se foloseste un tablou de tip *FormaGeo* cu numele *forme[]* cu o dimensiune citita de la tastatura.
- De retinut ca tablou *forme[]* este un tablou de referinte de tip *FormaGeo* pentru care se aloca zona de memorie.
- Acest tablou nu stocheaza obiecte din clasele derivate ale clasei *FormaGeo* (de tip *Cerc, Dreptunghi, Patrat*) ci numai referinte catre aceste obiecte.

- Fisierul-sursa (TestForma.java) care realizeaza ordonarea si punerea in executie a aplicatiei se prezinta in continuare.

- Metoda *citesteForma()* citeste de la tastatura atributele obiectelor de tipul *Cerc, Dreptunghi, Patrat*, pe baza unor optiuni care precizeaza tipul figurii, creaza un nou obiect de un tip derivat (*Cerc, Dreptunghi, Patrat*) din tipul *FormaGeo* si returneaza o referinta catre obiectul creat.

- In metoda *main(),* referinta la obiecte este apoi stocata in tabloul *forme[]*.

- Metoda *sortare()* este folosita pentru a sorta formele geometrice referite prin tabloul *forme[]* in functie de aria calculata a fiecarui tip de figura geometrica.

- Metoda *arie()* este apelata prin intermediul metodei *maiMic()* care la randul ei este apelata in metoda *sortare()*.

```
import java.io.* ;

class TestForma  {
    private static BufferedReader in;

    public static void main(String[] args) throws
    IOException {
        //Citeste numarul de figuri
        in = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.print("Numarul de figuri: ");
        int numForme = Integer.parseInt(in.readLine());
```

```
//citeste formele

FormaGeo[ ] forme = new FormaGeo[numForme];
for (int i = 0; i < numForme; ++i)
        {       forme[i] = citesteForma();        }

//sortare si afisare
sortare(forme);
System.out.println("Sortarea dupa arie: ");
for (int i = 0; i < numForme; ++i)
        {       System.out.println(forme[i]);   }
}
```

```
//creaza un obiect adecvat de tip FormaGeo. Functie de datele de intrare.
 //utilizatorul introduce 'c', 'p' sau 'd' pentru a indica forma, apoi introduce
     dimensiunile
 //in caz de eroare se intoarce un cerc de raza 0
    private static FormaGeo citesteForma() throws
    IOException  {
     double rad;
     double lg;
     double lat;
     String s;
     System.out.println("Introduceti tipul formei: ");
```

```
do
    {
      s = in.readLine();
 }  while (s.length() == 0);
 switch (s.charAt(0))
 {
            case 'c':
                        System.out.println("Raza cercului: ");
                        rad = Integer.parseInt(in.readLine());
                        return new Cerc(rad);
            case 'p':
                        System.out.println("Latura patratului: ");
                        lg = Integer.parseInt(in.readLine());
                        return new Patrat(lg);
```

```
 case 'd':
        System.out.println("Lungimea si latimea "+ "
   dreptunghiului pe linii separate: ");
        lg = Integer.parseInt(in.readLine());
        lat= Integer.parseInt(in.readLine());
        return new Dreptunghi(lg, lat);
default:
         System.err.println("Tastati c, p sau d: ");
         return new Cerc(0);
         }
   }
```

```
    //sortare
    private static void sortare(FormaGeo [ ] a)  {
    FormaGeo temp;
    for (int i = 0; i <= a.length - 2; i++)
        {for (int j = i+1; j <= a.length - 1; j++)
              {if (a[j].maiMic(a[i]))
                    {
                      temp =  a[i];
                      a[i] = a[j];
                      a[j] = temp;
                    }
              }
         }
    }
}
```

Dupa executia programului *TestForma.class* pe ecran se afiseaza:

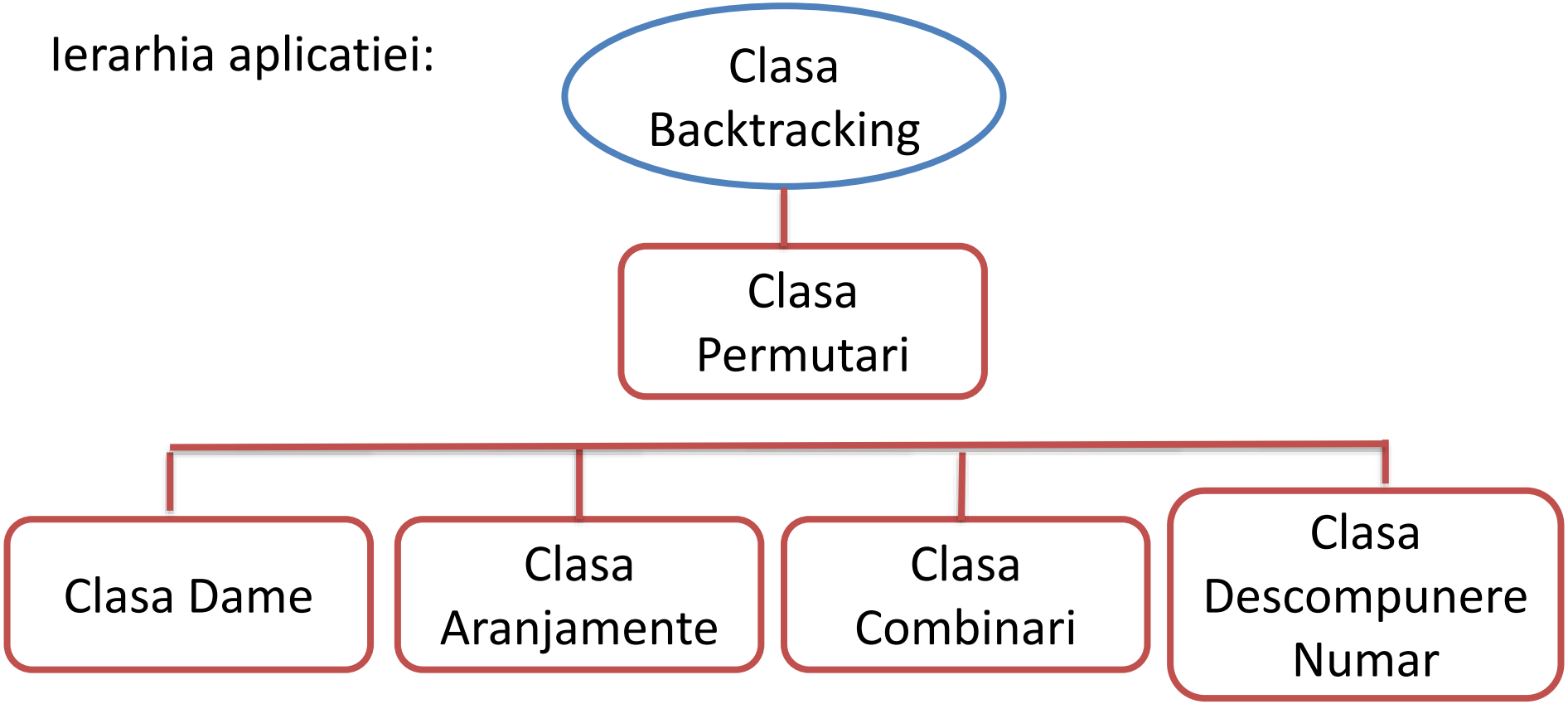
Sortarea dupa arie:
Cerc, avand aria 3.141592653589793
Dreptunghi, avand aria 6.0
Dreptunghi, avand aria 8.0
Dreptunghi, avand aria 24.0
Patrat, avand aria 36.0
Cerc, avand aria 78.53981633974483
Patrat, avand aria 100.0
Cerc, avand aria 314.1592653589793
Cerc, avand aria 452.3893421169302
Cerc, avand aria 452.3893421169302

Exemplul 2

Programul urmator (testBacktracking.java) prezinta o clasa abstracta (Backtracking), din care deriva clasa Permutari, din care deriva clasele Dame, Aranjamente, Combinari si DescompunereNumar, care se pot folosi (declarand obiecte din clasele respective) pentru rezolvarea problemelor cu ajutorul metodei de programare – Backtracking.

Exemplul 2

Ierarhia aplicatiei:



testBacktracking.java

```java
package testBacktracking;
import java.util.*;

abstract class Backtracking
{
    int[] st; int n,p;
    abstract void init(int k);
    abstract boolean Succesor(int k);
    abstract boolean Valid(int k);
    abstract boolean solutie(int k);
    abstract void tipar();
    void back()
    {
            boolean AS;
            int k=1;
            st = new int[n+1];
            init(k);
            while (k>0)
            {
                do { AS=Succesor(k); } while ( AS && !Valid(k) );
                if( AS )
                    if( solutie(k) )
                        tipar();
                    else{
                        k++; init(k);
                    }
                else k--;
            }//while
    }//back
}//class Backtracking
```

Clasa Backtracking

testBacktracking.java

```
31
32  class Permutari extends Backtracking
33  {
34      Permutari(int n)
35      {
36          this.n=n;
37      }//Permutari
38      void init(int k)
39      {
40          st[k]=0;
41      }//init;
42      boolean Succesor(int k)
43      {
44      if ( st[k]<n )
45      { st[k]++; return true; }
46      return false;
47      }//Succesor
48      boolean Valid(int k)
49      {
50          for(int i=1; i<k; i++)
51          if(st[i]==st[k])
52              return false;
53      return true;
54      }//Valid
55      boolean solutie(int k)
56      {
57          return k==n;
58      }//solutie
59      void tipar()
60      {
61          for(int i=1; i<=n; i++)
62          System.out.print(" "+st[i]);
63          System.out.println();
64      }//tipar
65  }//class Permutari
```

Clasa Permutari

testBacktracking.java

```
66
67  class Dame extends Permutari
68  {
69      Dame(int n)
70      {
71          super(n);
72      }//Dame
73      boolean Valid(int k)
74      {
75          for(int i=1; i<k; i++)
76          if( (st[i]==st[k]) || (Math.abs(st[k]-st[i])==Math.abs(k-i)) )
77              return false;
78          return true;
79      }//Valid
80  }//class Dame
```

Clasa Dame

testBacktracking.java

```
81
82  class Aranjamente extends Permutari
83  {
84      Aranjamente(int n, int p)
85      {
86          super(n);
87          this.p=p;
88      }//Aranjamente
89      boolean solutie(int k)
90      {
91          return k==p;
92      }//solutie
93      void tipar()
94      {
95          for(int i=1; i<=p; i++) System.out.print(" "+st[i]);
96          System.out.println();
97      }//tipar
98  }//class Aranjamente
99
```

Clasa Aranjamente

testBacktracking.java

```
 99
100 class Combinari extends Permutari
101 {
102     Combinari(int n, int p)
103     {
104         super(n);
105         this.p=p;
106     }//Combinari
107
108     boolean Valid(int k)
109     {
110         for(int i=1; i<k; i++)
111           if(st[i]==st[k]) return false;
112         if(k>1)
113             if(st[k]<st[k-1]) return false;
114         return true;
115     }//Valid
116     boolean solutie(int k)
117     {
118         return k==p;
119     }//solutie
120     void tipar()
121     {
122         for(int i=1; i<=p; i++) System.out.print(" "+st[i]);
123         System.out.println();
124     }//tipar
125 }//class Combinari
126
```

Clasa Combinari

testBacktracking.java

```
126
127 class DescompunereNumar extends Permutari
128 {
129     DescompunereNumar(int n)
130     {
131         super(n);
132     }//DescompunereNumar
133     void init(int k)
134     {
135         st[k]=0;
136     }//init;
137     boolean Succesor(int k)
138     {
139     if ( st[k]<n &&k<=n)
140     { st[k]++; return true; }
141     return false;
142     }//Succesor
143     boolean Valid(int k)
144     {
145         int s=0;
146         for(int i=1; i<=k; i++) s=s+st[i];
147         if(s>n) return false;
148         return true;
149     }//Valid
150     boolean solutie(int k)
151     {
152         int s=0;
153         for(int i=1; i<=k; i++) s=s+st[i];
154         if(s==n) return true;
155         else return false;
156     }//solutie
157     void tipar()
158     {
159         for(int i=1; i<=n; i++)
160             if(i!=n) System.out.print(st[i]+"+");
161             else System.out.print(st[i]);
162         System.out.println();
163     }//tipar
164 }//class DescompunereNumar
165
```

Clasa Descompunere Numar

testBacktracking.java

```
165
166 public class testBacktracking {
167
168     private static Scanner input;
169
170     public static void main(String[] args) {
171         System.out.print ("Introduceti numarul de elemente ale multimii A (n) =  ");
172         input = new Scanner(System.in);
173         int n = input.nextInt();
174         int p;
175         System.out.print ("Introduceti numarul p ale multimii A (p) =  ");
176         p = input.nextInt();
177
178         System.out.println(" Permutarile unei multimi cu "+n+" elemente");
179         Backtracking o1 = new Permutari(n);
180         o1.back();
181         System.out.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - -");
182
183         System.out.println(" Asezarea a "+n+" dame pe o tabla de sah");
184         Dame o2 = new Dame(n);
185         o2.back();
186         System.out.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - -");
187
188         System.out.println(" Aranjamentele unei multimi cu "+n+" elemente"+" luate cate "+p);
189         Backtracking o3 = new Aranjamente(n,p);
190         o3.back();
191         System.out.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - -");
192
193         System.out.println(" Combinarile unei multimi cu "+n+" elemente"+" luate cate "+p);
194         Backtracking o4 = new Combinari(n,p);
195         o4.back();
196         System.out.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - -");
197
198         System.out.println(" Descompunerile unui numar "+n+" in suma de numere");
199         Backtracking o5 = new DescompunereNumar(n);
200         o5.back();
201         System.out.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - -");
202     }
203 }
204
```

Întrebări?