LABORATOR 2

Clase şi obiecte în Java

Cuvinte cheie : Tipuri primitive, Tipuri de date referinţă , Structura programelor Java, Spaţii de nume,Pachete de clase, Tablouri,Definirea de clase în Java, Derivarea claselor, Metode, Mostenire, Polimorfism, Clasă abstractă, Metodă abstractă, Interfaţă.

Tipuri de date primitive – reprezintă acele date care sunt incluse implicit în limbajul Java. Sunt tipuri numerice intregi (byte, short, int, long, char), tipuri numerice reale (float, double) şi tipul boolean (boolean).

Tipuri de date referinţă – sunt clasele, tablourile şi interfeţele. Ca şi în limbajul C++, clasele reprezintă tipuri generice de date caracterizate printr-o stare definita de atributele clasei şi printr-o functionalitate definita de metodele clasei; tablourile sunt colecţii de date de acelaşi tip, iar interfeţele sunt un fel de clase abstacte şi vor fi studiate în lucrarea următoare.

Variabilele de tip primitiv se pot crea numai prin declaraţie (care este în acelaşi timp şi o definiţie). Variabilele de tip referinţă se pot crea cu operatorul new (spre deosebire de C++),care returnează o referinţă. O referinţă Java este un identificator al unui obiect şi este, de fapt un pointer ascuns, asemănător cu o referinţă C++, de care se deosebeşte în primul rând din punct de vedere sintactic (nu se mai foloseşte operatorul & la definiţia unei referinţe).

Deoarece nu se pot defini referinţe pentru tipurile primitive, în pachetul java.lang sunt definite mai multe clase care "împacheteaza" tipurile primitive, adică au ca dată membru o variabilă de tipul primitiv respectiv. Aceste clase sunt *Byte, Short, Integer, Character, Float, Double* etc. Majoritatea acestor clase (cu excepţia claselor *Integer* şi *Character*) au aceeaşi denumire ca şi tipul de date primitive corespunzător, cu primul caracter modificat în majusculă.
Structura programelor Java-O aplicaţie Java conţine cel puţin o clasă, care conţine cel putin o metodă cu numele *main* de tip ***void*** şi cu atributele ***static*** şi ***public***. Metoda *main* trebuie să aibă ca unic argument un vector de obiecte *String*. Ca şi în C, execuţia unui program începe cu funcţia *main*, doar că *main* trebuie să fie inclusă, ca metodă statică, într-o clasă şi trebuie să aibă un argument vector de şiruri. Exemplul următor este un program minimal, care afişează un text constant:

```
public class Main {
public static void main ( String arg[ ] ) {
System.out.println (" Main started ");
}
}
```

Un fişier sursă Java poate conţine mai multe clase, dar numai una din ele poate avea atributul ***public***. Numele fişierului sursă (de tip java‖) trebuie să coincidă cu numele clasei publice pe care o conţine. O clasă publică este accesibilă şi unor clase din alte pachete de clase. Compilatorul Java creează pentru fiecare clasă din fişierul sursă câte un fişier cu extensia *class* şi cu numele clasei. Dacă este necesar, se compilează şi alte fisiere sursă cu clase folosite de fişierul transmis

spre compilare. Faza de execuţie a unui program Java constă din încărcarea şi interpretarea tuturor claselor necesare execuţiei metodei ***main*** din clasa specificată în comanda *java*.

Spaţii de nume în Java -Un spaţiu de nume (*namespace*) este un domeniu de valabilitate pentru un nume simbolic ales de programator. In cadrul unui spaţiu nu pot exista două sau mai multe nume identice (excepţie fac metodele supradefinite dintr-o aceeaşi clasă). Pot exista nume identice în spaţii diferite. Fiecare clasă creează un spaţiu de nume pentru variabilele şi metodele clasei; ca urmare numele metodelor sau datelor publice dintr-o clasă trebuie precedate de numele clasei, atunci când se folosesc în alte clase.

Exemple:

Main.writeln ("abc"); // clasa Main, metoda writeln

Math.sqrt(x); // clasa Math, metoda sqrt

System.out // clasa System, variabila out

Clasele înrudite ca rol sau care se apelează între ele sunt grupate în "pachete" de clase (*package*). Instrucţiunea ***package*** se foloseşte pentru a specifica numele pachetului din care vor face parte clasele definite în fisierul respectiv; ea trebuie să fie prima instrucţiune din fişierul sursă Java. In lipsa unei instrucţiuni ***package*** se consideră că este vorba de un pachet anonim implicit, situaţia unor mici programe de test pentru depanarea unor clase. Un nume de pachet corespunde unui nume de director, cu fişierele de tip ***class*** corespunzătoare claselor din pachet.

Numele unui pachet poate avea mai multe componente, separate prin puncte. Numele de pachete cu clase predefinite, parte din JDK, încep prin *java* sau *javax*. Exemple : *java.io , java.util.regex , java.awt, javax.swing.tree* Un pachet este un spaţiu pentru numele claselor din acel pachet. In general numele unei clase publice trebuie precedat de numele pachetului din care face parte, atunci când este folosit în alt pachet. De observat că un fişier sursă nu creează un spaţiu de nume; este posibil şi chiar uzual ca în componenţa unui pachet să intre clase aflate în fişiere sursă diferite, dar care au la început aceeaşi instrucţiune ***package***.
Pachetul cu numele "java.lang" (*language*) este folosit de orice program Java şi de aceea numele lui nu mai trebuie menţionat înaintea numelui unei clase din *java.lang*.

Clasele *String,Integer,Object,System* ş.a. fac parte din pachetul java.lang. Exemplu de utilizare a unei clase dintr-un alt pachet decât *java.lang* :

```
public static void main (String arg[]) {
java.util.Random rand =new java.util.Random();
for (int i=1;i<=10;i++) // scrie 10 numere aleatoare
System.out.println ( rand.nextFloat());
}
```

Variabila cu numele "rand" este de tipul *Random*, iar clasa *Random* este definită în pachetul "java.util". Notaţia *rand.nextFloat()* exprimă apelul metodei *nextFloat()* din clasa *Random* pentru obiectul adresat de variabila ***rand***. Instrucţiunea ***import*** permite simplificarea referirilor la clase din alte pachete şi poate avea mai multe forme. Cea mai folosită formă este: *import pachet.* ;* Instrucţiunea anterioară permite folosirea numelor tuturor claselor dintr-un pachet cu numele "pachet", fără a mai fi precedate de numele pachetului. Exemplul următor ilustrează folosirea instrucţiunii ***import***:

```
import java.util.*; // sau: import java.util.Random;
class R {
public static void main (String arg[]) {
Random rand =new Random();
for (int i=1;i<=10;i++) // scrie 10 numere aleatoare
System.out.println ( rand.nextFloat());
}
}
```

Tablouri-Un tablou (*array*) este o listă de elemente de acelaşi tip plasate într-o zonă continuă de memorie, care pot fi adresate utilizând un nume, un identificator comun. Tipul elementelor din tablou trebuie să fie acelaşi şi poate fi oricare tip primitiv sau tip referinţă.

Un tablou Java este un tip referintă, deci identificatorul (numele) unui tablou reprezintă o referinţă (o adresa) la locatia unde se găseste întregul tablou de elemente. Un tablou Java este un obiect care conţine, în afară de elementele tabloului o variabilă membră publică *length* din care se poate citi (nu şi scrie) dimensiunea tabloului.

Tablourile pot fi definite pentru tipuri primitive sau pot conţine referinţe la obiecte (clase). Tablourile de date primitive conţin chiar acele date memorate în elementele tabloului, iar tablourile de obiecte (tip referinţă) conţin referinte memorate în elementele tabloului. Dimensiunile unui tablou sunt fixe (nu se mai pot modifica după creare).

Prin declararea unui tablou se crează o variabilă referinţă având valoarea null (care nu referă nimic). Nu se alocă spaţiu pentru elementele tabloului. Forma generală de declarare a unui tablou unidimensional este:

tip nume [ ]; // la fel ca in C si C++

tip [ ] nume; // specific Java

Declararea matricelor (vectori de vectori) poate avea şi ea două forme. Exemplu:

int a[ ][ ] ; // o matrice de întregi

int [][] b; // altă matrice de întregi

După declararea unui tablou urmează instantierea lui, adica alocarea zonei de memorie necesară stocarii elementelor lui. Instanţierea unui tablou se poate face în doua moduri. Cel mai utilizat mod este prin utilizarea lui *new* astfel:

Nume_tablou=new tip[nr_elemente];

Exemple:

float x[ ] = new float [10]; // aloca memorie ptr 10 reali

int n=10;

byte[ ][ ] graf = new byte [n][n];

Definirea de clase în Java-O clasă Java corespunde unui tip structură din limbajul C, dar o clasă mai conţine ca membri şi funcţii (metode) care realizează operaţii cu variabilele clasei. Clasele Java pot avea diferite niveluri de accesibilitate faţă de alte clase: ***public*** (accesibilă pentru orice altă clasă) ***private*** (inaccesibilă pentru orice altă clasă) ***protected*** (accesibilă pentru subclase) ***package*** (implicit) (accesibilă pentru clase din acelaşi pachet de clase) Clasele de bibliotecă Java sunt publice şi fiecare este definită într-un fişier separat de celelalte clase; numele clasei este acelaşi cu numele fişierului. O clasă (neabstractă) poate conţine: numai metode statice, numai date, date şi metode nestatice (*Object methods*), date si metode statice şi nestatice. Notiunea de funcţie se foloseşte în Java pentru ambele categorii de metode dintr-o clasă Java:

- Metode statice sau nestatice (cu nume diferit de numele clasei)

- Constructori de obiecte (cu acelasi nume ca şi clasa)

Specific programării orientate obiect este definirea şi utilizarea de clase cu date şi metode nestatice, numite *Object Methods* (metode ale obiectelor). Metodele obiectelor sunt in general publice pentru a putea fi apelate din afara clasei. Metodele publice nestatice ale unei clase pot fi clasificate astfel:
- Metode de acces la datele încapsulate în clasă (*getter, setter*)

- Metode moştenite de la clasa *Object* şi redefinite (suprascrise)

- Metode specifice clasei respective (determinate de rolul ei in aplicaţie)

O clasă **Complex** pentru numere complexe, va conţine partea reală şi partea imaginară a unui număr complex (ca date private), metode publice de acces la aceste date, metode mostenite (***toString(), equals()***) dar şi operaţii necesare lucrului cu numere complexe: adunare, scădere, ş.a.

```
public class Complex {
// datele clasei private
int re, im;
// constructor public
Complex (int re, int im) { this.re=re; this.im=im; }
```

```
// metode publice proprii clasei
public void add ( Complex cpx) { // adunarea a doua numere
re = re + cpx.re; im = im + cpx.im;
}
public void sub ( Complex cpx) { // scaderea a doua numere
re -= cpx.re; im -= cpx.im;
}
// metode mostenite si redefinite
public String toString (){
 return "(" + re + "," + im + ")“;}
}
```

Toate clasele Java extind clasa *Object* şi ,eventual, redefinesc unele metode moştenite de la clasa *Object*: *toString(), equals()*, ş.a.

Derivarea (extinderea) claselor-Derivarea este o tehnică de programare specifică programării orientate obiect şi este folosită pentru:

- Reutilizarea unor metode dintr-o clasă existentă în clasele derivate, fără a mai fi declarate sau definite.

- Crearea unor ierarhii de tipuri compatibile.

Derivarea înseamnă definirea unei clase **D** ca o subclasă a unei clase **A**, de la care ―moşteneşte‖ toate variabilele şi metodele dar nu şi constructorii. Clasa **A** se mai numeşte şi clasă de bază sau clasă părinte sau superclasa a lui **D**, iar **D** se numeşte subclasa lui **A**. In Java se spune că o subclasă extinde funcţionalitatea superclasei, în sensul că ea poate conţine metode şi date suplimentare. In general o subclasă este o specializare, o particularizare a superclasei şi nu extinde domeniul de utilizare al superclasei. De exemplu, o mulţime realizată ca vector este un caz particular de vector în care elementele sunt diferite între ele, iar clasa mulţime poate fi derivată din clasa vector cu redefinirea metodelor de adăugare la vector.

Moştenire(*Inheritance*) -înseamnă că toate metodele publice şi ***protected*** din clasa **A** pot fi folosite pentru obiecte din clasa **D**, fără ca acestea să fie declarate sau definite în **D**. Variabilele clasei **A** se regăsesc şi în obiectele de tip **D**. Variabilele ***private*** din superclasa **A** se moştenesc dar nu sunt direct accesibile pentru a fi folosite în noile metode definite în subclasa **D**. Subclasa **D** poate redefini metode moştenite de la clasa părinte **A** şi poate adăuga noi metode si variabile clasei **A**. Tipul **D** este un subtip al tipului **A**. La definirea unei clase derivate se foloseste cuvântul cheie ***extends*** urmat de numele clasei de bază.
Exemplul următor arată cum se poate defini o clasă *VSet* pentru multimi din vectori prin extinderea clasei *java.util.Vector* şi redefinirea a două metode moştenite : *addElement()* si *setElementAt()*. Pentru obiecte de tip *VSet* se pot folosi toate metodele publice din clasa *Vector* (peste 50 ca număr): *toString(), size()*,ş.a., aşa cum se vede din exemplul următor:

```
public static void main (String [] args) {
```

```
VSetv =new VSet();
for (int i=1;i<21;i++)
v.addElement (new Integer(i%10));
System.out.println (v.toString()); System.out.println (v.size());
}
```

Definirea clasei derivate, cu metodele mai vechi din clasa Vector, poate arăta astfel:

```
class VSet extends Vector {
public void addElement (Object obj) { // adaugare la multimea vector
if ( ! contains(obj)) // daca nu exista deja obj
super.addElement(obj); // atunci se adauga la multimea vector
}
}
```

Se observă cum metoda ***addElement()*** din subclasă apelează o metodă moştenită de la superclasa *Vector* şi anume ***contains()***. Crearea de elemente cu aceeaşi valoare se poate face şi cu metoda care modifică valoarea unui element dintr-o poziţie dată *setElementAt (Object obj, int i),* care ar trebui fie suprascrisă, fie interzisă pentru mulţimi vector (în Java nu este permis accesul prin indici la mulţimi). În acest exemplu metoda suprascrisă din subclasă apelează metoda cu acelaşi nume şi argumente din superclasă, iar pentru a deosebi cele două versiuni se foloseste cuvântul cheie ***super***. Dacă nu s-ar folosi ***super*** atunci funcţia ar fi infinit recursivă. O subclasă poate deveni superclasă pentru alte (sub)clase, iar derivarea (extinderea) poate continua pe oricâte niveluri. In Java nu este permisă extinederea simultana a mai multor clase pentru moştenire multiplă de date şi operaţii de la câteva clase. Cuvântul ***extends*** poate fi urmat de un singur nume de clasă. Nu se poate extinde o clasă finală (cu atributul ***final***) şi nu pot fi redefinite metodele din superclasă care au unul din modificatorii ***final, static, private***.

In Java, clasa *Object (java.lang.Object)* este superclasa tuturor claselor JDK şi a claselor definite de utilizatori. Orice clasă Java care nu are clauza ***extends*** în definiţie este implicit o subclasă derivată direct din clasa *Object*. Clasele abstracte şi interfeţele Java sunt şi ele subtipuri implicite ale tipului *Object*. Variabile sau argumente de tip *Object* (referinţe la tipul *Object*) pot fi înlocuite cu variabile de orice alt tip clasă, deoarece orice tip clasă este în Java derivat din şi deci compatibil cu tipul *Object.* Clasa *Object* transmite foarte puţine operaţii utile subclaselor sale; de aceea în alte ierarhii de clase (din alte limbaje) rădăcina ierarhiei de clase este o clasă abstractă. Deoarece clasa *Object* nu conţine nici o metodă abstractă, nu este obligatorie redefinirea metodelor mostenite, dar în practică se redefinesc câteva metode: *toString(), equals(), hashCode()***.**

Polimorfism-O funcţie polimorfică este o funcţie care are acelasi prototip, dar implementări (definiţii) diferite în clase diferite dintr-o ierarhie de clase (obţinute prin suprascrierea metodei). Metodele *equals()* şi *toString()* sunt exemple tipice de funcţii polimorfice, al căror efect depinde de tipul obiectului pentru care sunt apelate (altul decât tipul variabilei referinţă).

Exemple:

Object d = new Date();

Object f = new Float (3.14); String ds = d.toString(); // apel Date.toString()

String fs = f.toString(); // apel Float.toString()

In definirea clasei *Entry* polimorfismul explică de ce în metoda ***equals()*** nu este un apel recursiv; ceea ce se apelează este metoda ***equals()*** din clasa de care aparţine cheia (diferită de clasa *Entry*):

public boolean equals (Object obj) { // equals din clasa Entry

Entry e = (Entry)obj;

return key.equals (e.key); // equals din clasa variabilei key

}

Asocierea unui apel de metodă cu funcţia ce trebuie apelată se numeste "legare" (*Binding*). Legarea se poate face la compilare (legare timpurie) sau la execuţie ("legare târzie" sau "legare dinamică"). Pentru metodele finale şi statice legarea se face la compilare, adică un apel de metodă este tradus într-o instrucţiune de salt care conţine adresa funcţiei apelate. Legarea dinamică are loc în Java pentru orice metodă care nu are atributul *final* sau ***static***, metodă numită polimorfică. In Java majoritatea metodelor sunt polimorfice. Metodele polimorfice Java corespund funcţiilor virtuale din C++.

Soluţia funcţiilor polimorfice este specifică programării orientate pe obiecte şi se bazează pe existenţa unei ierarhii de clase, care conţin metode (nestatice) cu aceeaşi semnatură, dar cu efecte diferite în clase diferite. Numai metodele obiectelor pot fi polimorfice, în sensul că selectarea metodei apelate se face în funcţie de tipul obiectului pentru care se apelează metoda. O metodă statică, chiar dacă este redefinită în subclase, nu poate fi selectată astfel, datorită modului de apelare.

Clase abstracte în Java-Generalizarea şi abstractizarea în programarea cu obiecte sunt realizate şi prin clase abstracte. O superclasă defineşte un tip mai general decât tipurile definite prin subclasele sale. Uneori superclasa este atât de generală încât nu poate preciza nici variabile şi nici implementări de metode, dar poate specifica ce operaţii (metode) ar fi necesare pentru toate subclasele sale. In astfel de cazuri superclasa Java este fie o clasă abstractă, fie o interfaţă.

Exemplu Clasa *GraphicObject* poate fi o generalizare a tipurilor clasă ce reprezinta obiecte grafice(*Circle, Rectangle etc).*

```
abstract class GraphicObject {
                    int x, y;
                    . . .
```

```
        void moveTo(int newX, int newY) { //metoda normala
            . . .
        }
        abstract void draw();                    //metoda abstracta
}
```

O clasă care conţine cel puţin o metodă abstractă trebuie declarată ca abstractă, dar nu este obligatoriu ca o clasă abstractă să conţină şi metode abstracte. O clasă abstractă poate conţine date şi metode neabstracte utilizabile în subclase dar nu este instanţiabilă (nu se pot crea obiecte de acest tip clasă). Constructorul clasei abstracte este de obicei *protected* şi nu *public*.

O metodă abstractă este doar declarată (ca nume, tip şi argumente) dar nu este definită; ea este precedată de cuvântul cheie *abstract.* Ea urmează a fi definită într-o subclasă a clasei (interfeţei) care o conţine. Metodele abstracte pot să apară numai în interfeţe şi în clasele declarate abstracte.

Pentru a obtine clase instanţiabile dintr-o clasă abstractă trebuie implementate toate metodele abstracte moştenite, respectând declaraţiiile acestora (ca tip şi ca argumente).

Implementarea claselor pentru obiecte grafice ar putea fi următoarea:

```
 class Circle extends GraphicObject {

        void draw() {
            . . .       //obligatoriu implementarea
        }
    }

class Rectangle extends GraphicObject {
        void draw() {
            . . .       //obligatoriu implementarea
        }
}
```

O clasă abstractă este de multe ori o implementare parţială a unei interfeţe, pentru a simplifica definirea de clase instanţiabile care să respecte interfaţa. In clasa abstractă mai multe metode sunt definite în funcţie de câteva metode abstracte (cum ar fi cea care produce un iterator). Clasele concrete care vor extinde clasa abstracta vor avea de definit numai câteva metode Exemplu:

```
public interface Collection { // declara metode prezente in orice colectie
int size(); // dimensiune colectie
boolean isEmpty(); // verifica daca colectia este goala
… // alte metode
}
public abstract class AbstractCollection implements Collection {
public abstract int size(); // metoda abstracta
public boolean isEmpty ( return size()==0;} // metoda implementata
```

…
}

In bibliotecile de clase Java există câteva clase adaptor, care implementează o interfaţă prin metode cu definiţie nulă, unele redefinite în subclase. Ele sunt clase abstracte pentru a nu fi instanţiate direct, dar metodele lor nu sunt abstracte, pentru că nu se ştie care din ele sunt efectiv necesare în subclase. Exemplu de clasă adaptor utilă în definirea de clase ascultător la evenimente generate de tastatură:

```
public abstract class KeyAdapter implements KeyListener {
public void keyTyped(KeyEvent e) { } // la apasare+ridicare tasta
public void keyPressed(KeyEvent e) { } // numai la apasare tasta
public void keyReleased(KeyEvent e) { } // numai la ridicare tasta
}
```

O subclasă (care reacţionează la evenimente generate de taste) poate redefini numai una din aceste metode, fără să se preocupe de celalte metode nefolosite de subclasă:

```
class KListener extends KeyAdapter {
public void keyPressed(KeyEvent e) {
char ch = e.getKeyChar(); // caracter generat de tasta apasata
. . . // folosire sau afisare caracter ch
}
}
```

Interfeţe în Java-O interfaţă este o colecţie de metode abstracte şi (eventual) definiţii de constante simbolice. Ea poate fi considerată ca un caz particular de clasă abstractă. Metodele declarate într-o interfaţă sunt implicit publice şi abstracte. Din punct de vedere sintactic, diferenţele dintre clase abstracte şi interfeţe sunt:

- cuvinte cheie diferite la definire: *abstract class, interface*

- cuvinte diferite la definirea de subclase: *extends, implements*

Scopul interfeţelor este de a defini un tip compatibil simultan cu mai multe tipuri existente, dar fără riscul de moştenire multiplă. Se mai spune că o interfaţă stabileşte un contract‖ care trebuie respectat de clasele care implementează interfaţa, în sensul că aceste clase se obligă să definească toate metodele din interfaţă (la care se mai pot adăuga şi alte metode publice). Utilizarea unei interfeţe comune mai multor clase permite unificarea metodelor şi modului de utilizare a unor clase cu acelaşi rol, dar şi scrierea unor metode general aplicabile oricărei clase care respectă interfaţa comună.

Exemplu de interfaţă din JDK:

```
public interface CharSequence { // orice secventa de caractere
char charAt (int i); // caracterul din pozitia i a secventei
```

```
int length(); // lungimea secventei
String toString(); // sir echivalent secventei
CharSequence subSequence (int i, int j); // extrage o subsecventa
}
```

Această interfaţă este respectată (implementată) de clase ca *String, StringBuffer, StringBuilder, Segment, CharBuffer***.**

Exemplu:

```
public final class String implements CharSequence, Serializable {
// date
private final char value[];
private final int count;
// constructori . . .
// metode
public int length() { return count; }
public String toString() { return this; }
. . .
}
```

Prin definirea unei interfeţe sau clase abstracte se creează un tip comun mai multor clase deoarece o variabilă (sau un argument) de un tip interfaţă poate fi înlocuită fără conversie explicită cu o variabilă de orice subtip, deci cu referinţe la obiecte care implementează interfaţa sau care extind clasa abstractă. Nu pot fi create obiecte de un tip interfaţă sau clasă abstractă, dar se pot declara variabile, argumente formale şi funcţii de un tip interfaţă sau clasă abstractă. Astfel de variabile vor fi înlocuite (prin atribuire sau prin argumente efective) cu variabile de un tip clasă care implementează interfaţa respectivă.

O interfaţă poate extinde o altă interfaţă (*extends*) cu noi metode abstracte. Ca exemplu, interfaţa **List** extinde interfaţa *Collection* cu metode de acces direct, prin indici, la elementele colecţiei:

```
public interface List extends Collection {
void add ( int i, Object x); // introduce pe x in pozitia i din colectie
Object get (int i); // valoare element din pozitia i
Object set (int i, Object x); // înlocuire element din pozitia i cu x
Object remove (int i); // elimina si returneaza elementul din pozitia i
. . .
}
```

Diferenţa majoră este aceea că o interfaţă nu poate conţine declaratii de date decat statice si constante şi că o clasă poate implementa simultan mai multe interfeţe dar nu poate extinde decât o singură clasă abstractă (sau neabstractă).

Clasele care implementează o aceeaşi interfaţă pot fi foarte diverse şi nu formează o ierarhie de tipuri compatibile. Un exemplu sunt clasele cu obiecte comparabile, care toate implementează interfaţa *Comparable: String, Date, Integer, BigDecimal,* ş.a.

O clasă poate simultan să extindă o clasă (alta decât clasa *Object*, implicit extinsă) şi să implementeze una sau mai multe interfeţe. Altfel spus, o clasă poate moşteni date şi/sau metode de la o singură clasă, dar poate moşteni mai multe tipuri (poate respecta simultan mai multe interfeţe). De exemplu, mai multe clase predefinite SDK, cu date, implementează simultan interfeţele *Comparable* (obiectele lor pot fi comparate la mai mic/ mai mare), *Clonable* (obiectele lor pot fi copiate), *Serializable* (obiectele lor pot fi salvate sau serializate în fişiere disc sau pe alt mediu extern). Exemple:

public class String implements Comparable, Serializable { ...}

public class Date implements Serializable, Clonable, Comparable { ...}

Clasa *Object* conţine metoda ***clone()***, folosită numai de clasele care declară că implementează interfaţa *Clonable*. Metoda neabstractă ***clone()*** este moştenită automat de toate clasele Java, dar este aplicabilă numai pentru o parte din clase. Pentru a semnala utilizarea greşită a metodei ***clone()*** pentru obiecte ne-clonabile, se produce o excepţie de tip *CloneNotSupportedException* atunci când ea este apelată pentru obiecte din clase care nu aderă la interfaţa *Clonable*. O utilizare asemănătoare o are interfaţa *Serializable*, pentru a distinge clasele ale căror obiecte sunt serializabile (care conţin metode de salvare şi de restaurare în / din fişiere) de clasele ale căror obiecte nu pot fi serializate (fără obiecte "persistente"). Practic, toate clasele cu date sunt serializabile.
Interfeţe ca *Serializable* si *Clonable* se numesc interfeţe de "marcare" a unui grup de clase (*tagging interfaces*), pentru a permite anumite verificări.

O interfaţă care stabileste un tip comun poate fi atât de generală încât să nu conţină nici o metodă. Un exemplu este interfaţa *EventListener* (pachetul "java.util"), care stabileşte tipul ascultător la evenimente‖, dar metodele de tratare a evenimentului nu pot fi precizate nici ca prototip, deoarece depind de tipul evenimentului. Interfaţa este extinsă de alte interfeţe, specifice anumitor ascultători (pentru anumite evenimente):

```
public interface ActionListener extends EventListener {
public void actionPerformed(ActionEvent e);
}
public interface ItemListener extends EventListener {
public void itemStateChanged(ItemEvent e);
}
```

Exercitii.

1. **Automate Finite (FSM)**. Simularea software a unui automat finit.

Descrierea noţiunii de automat se poate face în mai multe moduri, care se numesc modele de automate . Noi vom defini Mealy.

Definiţie. Un automat Mealy este un 5-uplu $A = [I, S, O, f, g]$, unde

I, S, O sunt mulţimi nevide, numite respectiv alfabet de intrare, mulţime de stări şi alfabet de ieşire, iar

$f: S \times I \to S$ şi $g: S \times I \to O$ sunt funcţiile de trecere sau de tranziţie şi de ieşire sau de răspuns ale automatului.

Definiţie. Un semiautomat este un triplet $A = [I, S, f]$, în care

I, S, f sunt alfabetul de intrare, mulţimea de stări şi funcţia de tranziţie $f: S \times I \to S$.

Definiţie. Semiatomatul ataşat automatului $A = [I, S, O, f, g]$ este semiatomatul $S(A) = [I, S, f]$

Observăm că prin considerarea semiautomatului $S(A)$ se face abstracţie de comportarea la ieşire a automatului A, rămânând esenţială funcţionarea internă a automatului.

Remarcăm că unui automat i se poate ataşa, în mod unic, un semiautomat, iar un semiautomat poate fi extins la un automat, prin alegerea unui alfabet de ieşire O şi a unei funcţii g. Automatul ataşat astfel nu este unic, depinzând de această alegere.

Clasa abstracta AutomatMealyAbstract simuleaza un automat Mealy

```
interface functie {
public String calcul(String stare,String intrare);
}

Public abstract class AutomatMealyAbstract {
functie ff;
functie gg;
public AutomatMealyAbstract (functie fi, functie gi){
ff=fi;
gg=gi;
}

public abstract String f(String stare,String intrare);
```

```
Public abstract String g(String stare,String intrare);

public abstract String evolutie(String stareInitiala,String stringIntrare);

};
```

1.1. Sa se scrie o clasa AutomatMealy care extinde clasa AutomatMealyAbstract si sa se implementeze toate metodele clasei de baza.

Pentru a implementa un automat finit va trebui sa scriem clasele fc si gc care implementeaza interfata functie:

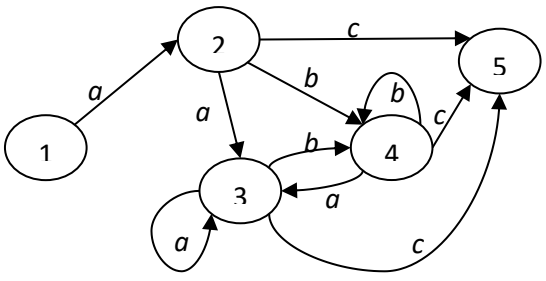
```
//Definim doua functii
class fc implements functie {
public String calcul(String stare,String intrare) {
// Implementare prin swich
}
}
class gc implements functie {
public String calcul(String stare,String intrare) {
// Implementare prin swich

}
}
```

Apoi vor fi instantiate clasele fc,gc si AutomatMealy

```
functie f=new fc();
functie g=new gc();
AutomatMealy myAutomat  = new AutomatMealy(f,g);
```

1.2. Sa se implementeze clasele fc si gc pentru automatul care recunoaste limbajul descries de expresia regulate: E=a(a|b)*c.

Functia de tranzitie f este data de graful:



Functia de iesire este g(s,x)=upercase(x)=X;

1.3. Sa se scrie un program care sa testeze codul.

2. **Masina Turing.** Simularea software a unei masini Turing.

Intuintiv vorbind o maşină Turing este specificabilă pe de o parte prin hardware-ul asociat ei şi pe de altă parte prin limbajul natural asociat ei.

Hardware-ul unei maşini Turing constă din trei elemente principale după cum urmează :

-o memorie infinită, dată sub forma unei benzi infinite într-un sens, care este presupusă a fi împărţită în celule, fiecare celulă fiind capabilă să memoreze un simbol;

-un cap de citire care se poate mişca de-a lungul benzii;

-un control finit care se poate găsi într-o stare dintr-un număr finit de stări posibile.

Distingem următoarele tipuri de operaţii pe care le poate efectua o maşină Turing :

-Operaţia de examinare a celulei benzii din poziţia capului de citire.

-Operaţia de deplasare a benzii cu o celulă la dreapta sau la stânga.

-Operaţia de schimbare a stării controlului finit.

Prin operaţia de examinare a unei celule capul de citire poate afecta celula respectivă în unul din următoarele moduri posibile :

-Poate lăsa conţinutul celulei neschimbat, dacă celula nu este vizitată prima dată.

-Schimbă conţinutul celulei ştergând simbolul pe care-l conţine şi înlocuindu-l cu alt simbol.

În cazul în care capul de citire vizitează pentru prima dată o celulă atunci conţinutul acesteia se presupune a fi blanc, acesta fiind modificat prin scrierea unui simbol neblanc în celula respectivă.

Definiţie. O maşină Turing este un 5-uplu $MT=(I,S,f;b,s_0)$ unde

-I este un alfabet finit,

-S o mulţime finită, nevidă, numită mulţimea stărilor,

-$f:(S\times I)' \to S \times \{I\setminus\{b\}\} \times \{-1,0,1\}$ este funcţia de tranziţie, $(S \times I)' \subset S \times I$ este mulţimea perechilor de forma (s,i) pentru care funcţia **f** este definită,

-$b \in I$ este simbolul “blanc”, iar

-s_0 este starea iniţială a maşinii.

Dacă $(s,i) \notin (S \times I)$' perechea (s,i) este o situaţie de oprire pentru maşina Turing.

Consideram in continuare ca masina Turing funcţionează în mod determinist. Dacă f(s,i)=(s',i',1), atunci MT găsindu-se în starea **s** şi citind simbolul **i**,trece în starea **s'**, înlocuieşte pe **i** cu simbolul **i'** şi deplasează bauda la stânga cu o celulă; dacă f(s,i)=(s',i',0) bauda rămâne pe loc, iar dacă f(s,i)=(s',i',-1), bauda se deplasează la dreapta cu o celulă, automatul acţionând în rest la fel ca mai sus.

Definiţie. O configuraţie instantanee a unui MT este un triplet γ=(s,p,n) unde :

-$s \in S$ este starea curentă a maşinii

-$p \in (I \setminus \{U\})^*$ este conţinutul porţiunii iniţiale din bandă, până la primul simbol blanc, iar

-n este numărul de ordine al celulei în dreptul căreia se află capul de citire.

Definiţie. Limbajul acceptat de o MT=(I,S,f;b;s_0) prin mulţimea de stări finale $S_1 \subseteq S$ este $L(MT)=\{p | p \in \{I \setminus \{b\}\}^*, (s_0, p, 1) \vdash^* (s,q,n_1), s \in S_1, q \in I^*\}$.

Aşadar limbajul acceptat de o MT este alcătuit din acele cuvinte care plasate pa baudă, începând cu prima celulă (automatul găsindu-se iniţial în starea s_0) conduc la o stare din S_1, activitatea maşinii începând cu prima celulă.

În continuare vom presupune că dacă maşina intră într-o stare finală se opreşte.

Clasa abstracta MasinaTuringAbstract simuleaza comportamentul unei masini Turing.

```
class triplet{
String stare;
String symbol;
Integer deplasare;
}

interface functie {
public triplet calcul(String stare,String intrare);
}

Public abstract class MasinaTuringAbstract {
functie ff;
StringBuilder bb;
public MasinaTuringAbstract (functie fi, String banda){
ff=fi;
bb = new StringBuilder(banda);
}
```

```
public abstract triplet f(String stare,String intrare);


public abstract String evolutie(String stareInitiala) ;
public  String stareBanda(){
return bb.toString().trim();
}
}
```

3.1. Sa se scrie o clasa MasinaTuring care extinde clasa MasinaTuring Abstract si sa se implementeze toate metodele clasei de baza.

Pentru a implementa o Masina Turing va trebui sa scriem clasa fc care implementeaza interfata functie:

```
class fc implements functie {
public triplet calcul(String stare,String intrare); {
// Implementare prin swich
};
};
```

3.2.Sa se implementeze clasa fc pentru Masina Turing :

4. MT=(I,S,f ; b,s_0) unde I={0,1,x,y,b}, S={0,1,2,3,4,5}, **f** este dată de tabelul :

Nr. de ordine	S	i	f(s,i)
1	0	0	(1,x,1)
2	1	0	(1,0,1)
3	1	y	(1,y,1)
4	1	1	(2,y,-1)
5	2	y	(2,y,-1)
6	2	x	(3,x,1)
7	2	0	(4,0,-1)
8	4	0	(4,0,-1)
9	4	x	(0,x,1)
10	3	y	(3,y,1)
11	3	b	(5,y,1)

adica (SXI)'={(0,0), (1,0), (1,y), (1,1), (2,y), (2,x), (2,0), (4,0), (4,x), (3,y), (3,b),}. Să presupunem că pe bandă este înscris cuvântul 000111.

Constatăm că numărul 000111 face parte din limbajul acceptat de maşina Turing dată prin mulţimea de stări finale S_1={5}.

3.3. Sa se scrie un program care sa testeze codul.