

Universitatea “Lucian Blaga” din Sibiu,
Catedra de Calculatoare și Automatizări



Analiza și proiectarea algoritmilor

**Curs, anul II – Calculatoare, Tehnologia informației,
Ingineria sistemelor multimedia**

Puncte de credit: 5

Autor: Dr. Árpád GELLÉRT

Web: <http://webpace.ulbsibiu.ro/arpad.gellert>

E-mail: arpad.gellert@ulbsibiu.ro



Cuprins (1)

Partea I – Limbajul Java

- Structura lexicală și instrucțiuni
- Variabile și constante
- Tablouri și matrici
- Tratarea excepțiilor
- Operații I/O
- Crearea claselor
- Interfețe grafice
- Fire de execuție
- Colecții



Cuprins (2)

Partea II – Analiza algoritmilor

- Complexitate, notații asimptotice
- Recurențe
- Algoritmi de căutare și sortare
- Tabele de dispersie
- Arbori binari
- Heap-uri
- Grafuri



Cuprins (3)

Partea III – Proiectarea algoritmilor

- Divide et impera
- Greedy
- Programare dinamică
- Backtracking
- Algoritmi genetici
- Rețele neuronale



Nota finală

$$\text{Nota laborator (NL)} = 0,5 * T + 0,5 * C$$

$$\text{Nota finală} = 0,4 * NL + 0,6 * E$$

- T = Test susținut la laborator din aplicațiile propuse la laborator în cadrul capitolului *Limbajul Java*;
- C = Colocviu de laborator din aplicațiile propuse la laborator în cadrul capitolelor *Analiza algoritmilor* respectiv *Proiectarea algoritmilor*;
- E = Examen susținut din capitolele *Analiza algoritmilor* și *Proiectarea algoritmilor*.



Bibliografie de bază

- [Knu00] Knuth D., *Arta programării calculatoarelor*, Teora, 2000.
- [Cor00] Cormen T., Leiserson C., Rivest R., *Introducere în algoritmi*, Agora, 2000.
- [Giu04] Giumale C., *Introducere în analiza algoritmilor*, Polirom, 2004.
- [Wai01] Waite M., Lafore R., *Structuri de date și algoritmi în Java*, Teora, 2001.
- [Log07] Logofătu D., *Algoritmi fundamentali în Java*, Polirom, 2007.
- [Tan07] Tanasă Ș., Andrei Ș., Olaru C., *Java de la 0 la expert*, Polirom, 2007.

Introducere (1)

- **Geneza cuvântului algoritm:** provine din latinizarea numelui savantului Al-Khwarizmi (algoritmi) – matematician, geograf, astronom și astrolog persan (780-850), considerat și părintele algebrei moderne.
- **Algoritmul** este o secvență de operații care transformă mulțimea datelor de intrare în datele de ieșire.
- **Proiectarea algoritmului** constă în două etape:
 - Descrierea algoritmului printr-un pseudolimbaj (schemă logică, pseudocod);
 - Demonstrarea corectitudinii rezultatelor în raport cu datele de intrare.
- **Analiza algoritmului** se referă la evaluarea performanțelor acestuia (timp de execuție, spațiu de memorie).

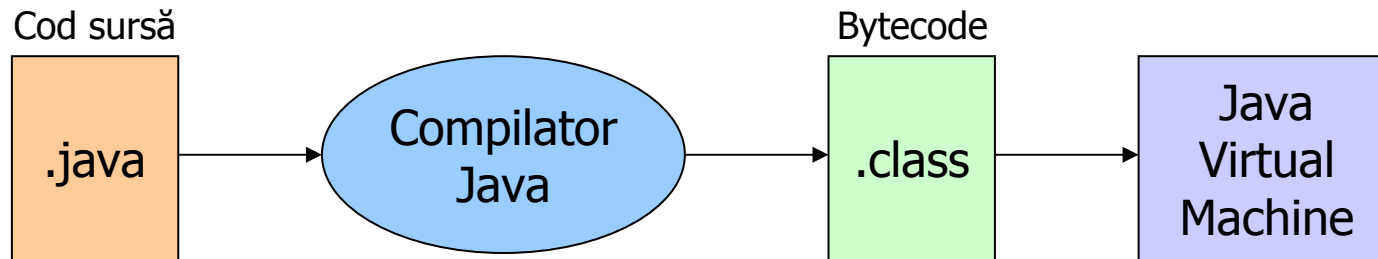




Introducere (2)

- **Implementarea algoritmilor** se va face în limbajul Java. Medii de dezvoltare:
 - Java Development Kit (JDK) – <http://www.oracle.com>
 - NetBeans – <https://netbeans.org>
 - IntelliJ – <https://www.jetbrains.com>
 - Eclipse – <http://www.eclipse.org>
 - Android Studio – <https://developer.android.com/studio/index.html>
- **Principalele caracteristici** ale limbajului Java:
 - Simplitate – elimină supraîncărcarea operatorilor, moștenirea multiplă, etc.;
 - Complet orientat pe obiecte;
 - Portabilitate – Java este un limbaj compilat și interpretat, fiind astfel independent de platforma de lucru (Write Once, Run Enywhere);
 - Oferă posibilitatea, prin JNI (Java Native Interface), de a implementa aplicații mixte (ex. Java/C++), prețul folosirii codului nativ fiind însă dependența de platformă [Gor98];
 - Permite programarea cu fire de execuție.

Introducere (3)



- În urma compilării codului sursă, se obține codul de octeți (bytecode) care este apoi interpretat de mașina virtuală Java (JVM). Astfel, aplicația poate fi rulată pe orice platformă care folosește mediul de execuție Java.
- Pe lângă aplicații obișnuite, pot fi implementate aplicații web (applet), aplicații server (servlet, JavaServer Pages, etc.), aplicații în rețea, telefonie mobilă (midlet), aplicații distribuite RMI (Remote Method Invocation), etc.
- Compilarea programelor (javac.exe):
*javac *.java*
- Rularea programelor (java.exe):
*java **



Partea I

Limbajul Java



Limbajul Java

O aplicație simplă:

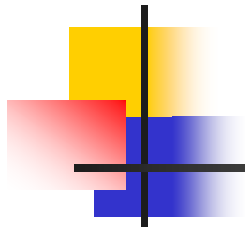
```
class HelloWorld{  
    public static void main(String args[]){  
        System.out.println("Hello world");  
    }  
}
```

- Aplicația HelloWorld afișează la consolă mesajul "Hello world".
- În orice aplicație trebuie să existe o clasă primară, care conține metoda *main* de la care începe execuția în momentul rulării. Parametrul *args* al metodei *main* este un tablou de șiruri de caractere și reprezintă argumentele din linie de comandă.
- Fișierul care conține clasa primară trebuie să aibă numele clasei (case sensitive!).
- Java permite o singură clasă publică într-un fișier.
- Compilarea programului: *javac HelloWorld.java*
- Rularea programului: *java HelloWorld*



Limbajul Java – convenții de scriere a codului [Web01]

- Principalele segmentele ale codului Java apar în următoarea ordine:
 - Declarația de pachet (*package*);
 - Directivele de *import*;
 - Declarația de clasă sau interfață;
 - Declarațiile variabilelor membru ale clasei (statice) sau ale instanței;
 - Constructori și metode;
- Convenții de nume
 - Numele de clase, implicit de constructori, încep cu majusculă (ex.: NumeClasa);
 - Numele de metode încep cu literă mică (ex.: numeMetoda);
 - Numele de variabile și obiecte încep cu literă mică (ex.: numeObiect);
 - Numele constantelor se scriu cu majuscule, cuvintele fiind despărțite prin “_” (ex.: NUME_CONSTANTA);
 - Numele trebuie să fie cât mai sugestive.



Limbajul Java – Structura lexicală

Structura lexicală a limbajului Java este foarte asemănătoare cu cea a limbajului C++.

- Comentariile se fac ca în C++.
- Operatorii sunt în mare parte la fel ca în C++.
- Instrucțiunile sunt și ele foarte asemănătoare cu cele din C++. O diferență importantă constă în faptul că expresiile care apar în instrucțiunile condiționale (*if*, *for*, *while*, *do*) sunt strict de tip *boolean*, în timp ce în C++ pot fi de tip *int*.
- Tipurile de date se clasifică în două categorii:
 - Tipuri de date primitive: *byte* (1 octet), *short* (2 octeți), *int* (4 octeți), *long* (8 octeți), *char* (2 octeți – unicode), *float* (4 octeți), *double* (8 octeți), *boolean* (1 bit);
 - Tipuri de date referință: clase, interfețe, tablouri. Există și variantele referință (clase wrapper) ale tipurilor de date primitive: *Byte*, *Short*, *Integer*, *Long*, *Character*, *Float*, *Double*, *Boolean*. Variantele referință ale tipurilor de date primitive sunt necesare pentru că în Java nu există operatorul & pentru definiția unei referințe. Pentru șiruri de caractere, pe lângă *char[]*, există și tipul referință *String*.



Limbajul Java – variabile

- Variabilele primitive se pot crea prin declarație. Exemple:
 - `int i = 10;`
 - `float f = 3.14f;`
 - `double d = 3.14;`
- Variabilele referință se pot crea doar cu operatorul *new* (care returnează o referință). Variabilele de tip *String* și începând cu JDK 1.5, cele de tip wrapper care se pot crea și prin declarație (fără *new*). Exemple:
 - `Integer i = new Integer(10);`
 - `Integer j = 20;`
 - `String h = new String("Hello");`
 - `String w = "world";`
- În Java nu este admisă supraîncărcarea operatorilor, excepție fiind doar operatorul de adunare pentru clasa *String*, care permite concatenarea șirurilor. Exemplu:

```
String h = "Hello";
String w = "world";
String s = h + " " + w;
```
- Concatenarea unui *String* cu o variabilă primitivă determină conversia implicită a acesteia în *String*. Exemplu:

```
String t = "Two";
int i = 2;
boolean b = true;
String s = t + " = " + i + " is " + b;    //Rezultat: "Two = 2 is true"
```



Limbajul Java – constante

- Pentru crearea unei constante, declarația de variabilă trebuie precedată de cuvântul cheie *final*.
- Nu este permisă modificarea valorii unei constante.
- O clasă declarată *final* nu poate fi derivată (v. moștenirea).
- Exemple

```
public class Main {  
    public static void main(String[] args) {  
        final String APA = "Analiza si proiectarea algoritmilor";  
        final int TEN = 10;  
        TEN++; //Eroare!  
        APA = "Analiza, proiectarea si implementarea algoritmilor"; //Eroare!  
        int eleven = TEN + 1;  
    }  
}
```



Limbajul Java – clasa String (1)

Principalele operații:

- **substring** cu parametrii *index0* și *indexf* – returnează subșirul care începe la *index0* și se termină la *indexf-1*, lungimea fiind *indexf-index0*. Exemplu:

```
String str = "Hello world";  
String substr = str.substring(0, 5);           //Rezultat: "Hello"
```

- **charAt** – returnează caracterul de pe o anumită poziție din șir. Exemplu:

```
String str = "Hello world";  
char c = str.charAt(6);                       //Rezultat: 'w'
```

- **length** – returnează lungimea șirului în număr de caractere. Exemplu:

```
String str = "Hello world";  
int len = str.length();                      //Rezultat: 11
```

- **equals / equalsIgnoreCase** – permite comparația unui șir cu alt șir. Returnează true dacă cele două șiruri sunt egale, respectiv false dacă ele nu sunt egale. Exemplu:

```
String hello = "Hello";  
if(hello.equals("Hello"))                    //Rezultat: true  
    System.out.println("equal");             //Rezultat: "equal"
```




Limbajul Java – clasa String (2)

- **compareTo / compareToIgnoreCase** – compară două șiruri. Returnează valoarea 0 dacă cele două șiruri sunt lexicografic egale, o valoare negativă dacă parametrul este un șir lexicografic mai mare decât șirul curent și o valoare pozitivă dacă parametrul este un șir lexicografic mai mic decât șirul curent. Exemplu:

```
String hello = "Hello";  
if(hello.compareTo("Hello") == 0)           //Rezultat: 0  
    System.out.println("equal");             //Rezultat: "equal"
```

- **trim** – elimină eventualele spații de la începutul și sfârșitul șirului. Exemplu:

```
String hello = " Hello ";  
hello = hello.trim();                       //hello="Hello"
```

- **toLowerCase / toUpperCase** – transformă toate literele din șir în litere mici / mari. Exemplu:

```
String str = "Hello world";  
str = str.toUpperCase();                     //Rezultat: "HELLO WORLD"
```

- **split** – separarea unui șir pe baza unui delimitator. Exemplu:

```
String str = "Hello world";  
String t[] = str.split(" ");                 //Rezultat: {"Hello", "world"}; 17
```



Limbajul Java – clasa StringTokenizer

- Permite separarea unui șir de caractere în simboluri;
- Se instanțiază un obiect *StringTokenizer*, specificând șirul de caractere și setul de delimitatori;
- Următoarea secvență de program afișează pe ecran simbolurile (cuvintele) șirului delimitate prin caracterele spațiu și virgulă:

```
String str = "Analiza, proiectarea si implementarea algoritmilor";  
StringTokenizer st = new StringTokenizer(str, " ,");  
while(st.hasMoreTokens())  
    System.out.println(st.nextToken());
```

- Setul de delimitatori poate fi specificat și prin funcția *nextToken*:

```
while(st.hasMoreTokens())  
    System.out.println(st.nextToken(" ,"));
```



Limbajul Java – tablouri

- Posibilități de declarare a tablourilor:
 - `int[] fibo = new int[10];`
 - `int fibo[] = new int[10];`
 - `int n = 10;`
`int fibo[] = new int[n];`
 - `int fibo[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};`
- Tablou cu elemente primitive vs. referință:
 - `int pTab[] = new int[5];` `//tablou (referinta!) cu 5 elem. primitive`
`for(int i=0; i<pTab.length; i++)`
`pTab[i] = i+1;`
 - `Integer rTab[] = new Integer[5];` `//tablou (referinta!) cu 5 elem. referinta`
`for(int i=0; i<rTab.length; i++)`
`rTab[i] = new Integer(i+1);`
- Afișarea elementelor unui tablou:
 - `for(int i=0; i<fibonacci.length; i++)`
`System.out.println(fibonacci[i]);`
 - `for(int i : fibonacci)` `//incepand cu JDK 1.5`
`System.out.println(i);`



Limbajul Java – clasa Arrays

Principalele operații:

- **fill** – permite umplerea tabloului sau a unei zone din tablou cu o anumită valoare:

```
int tab[] = new int[5];  
Arrays.fill(tab, 7); //Rezultat: tab={7,7,7,7,7}  
Arrays.fill(tab, 1, 3, 8); //Rezultat: tab={7,8,8,7,7}
```

- **equals** – compară două tablouri returnând true dacă au toate elementele egale:

```
int a[] = {1,2,3,4};  
int b[] = {1,2,3,4};  
int c[] = {1,2,4,8};  
System.out.println(Arrays.equals(a, b) ? "a==b" : "a!=b"); //Rezultat: "a==b"  
System.out.println(Arrays.equals(a, c) ? "a==c" : "a!=c"); //Rezultat: "a!=c"
```

- **sort** – sortează elementele tabloului în ordine crescătoare folosind algoritmul Quicksort pentru tipuri primitive respectiv Mergesort pentru tipuri referință:

```
int pTab[] = {9,6,2,1,5,3};  
Arrays.sort(pTab,1,4); //Rezultat: tab={9,1,2,6,5,3}  
Arrays.sort(pTab); //Rezultat: tab={1,2,3,5,6,9}  
Integer rTab[] = new Integer[5];  
for(int i=0, k=rTab.length; i<rTab.length; i++, k--) //Rezultat : tab={5,4,3,2,1}  
    rTab[i] = new Integer(k);  
Arrays.sort(rTab); //Rezultat : tab={1,2,3,4,5}
```

- **binarySearch** – returnează poziția elementului căutat în tablou sau o valoare negativă dacă valoarea căutată nu este găsită. Algoritmul folosit este căutarea binară, de aceea tabloul trebuie sortat înainte de apelul acestei metode. Exemplu:

```
int tab[] = {5,4,1,7,3}; //tablou nesortat  
int v = Arrays.binarySearch(tab, 4); //Rezultat greșit: -4  
Arrays.sort(tab); //Rezultat: tab={1,3,4,5,7}  
v = Arrays.binarySearch(tab, 4); //Rezultat corect: 2
```



Limbajul Java – matrici

- Posibilități de declarare a matricilor:
 - `int m[][] = new int[3][4];`
 - `int nRow=3, nCol=4;`
`int m[][] = new int[nRow][nCol];`
 - `int m[][] = new int[3][];`
`for(int i=0; i<3; i++)`
`m[i] = new int[4];`
 - `int m[][] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};`
- Tablou multidimensional cu număr variabil de coloane. Exemplu de alocare, inițializare și afișare:

```
int m[][] = new int[3][];
for(int i=0, k=1; i<3; i++){
    m[i] = new int[4+i];
    for(int j=0; j<4+i; j++, k++)
        m[i][j] = k;
}
for(int i=0; i<3; i++){
    for(int j=0; j<4+i; j++)
        System.out.print(m[i][j]+" ");
    System.out.println();
}
```

//Rezultat afișat:
//1 2 3 4
//5 6 7 8 9
//10 11 12 13 14 15
- O matrice *m* are *m.length* linii și *m[0].length* coloane.



Limbajul Java – conversii (1)

- Conversia variabilelor primitive

- Conversie implicită. Exemple:

- ```
int a = 3, b = 2;
```

- ```
double c = a/b;           //Conversie din int in double. Rezultat: 1.0
```

- ```
int d = 5;
```

- ```
float f = d;              //Conversie din int in float.
```

- ```
Rezultat: 5.0
```

- Conversie explicită (casting). Exemplu:

- ```
double pi = 3.14;
```

- ```
int i = (int)pi; //Conversie din double in int. Rezultat: 3
```

- Conversia variabilelor primitive în variabile referință.

Exemple:

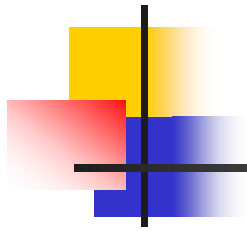
- ```
int x = 10;
```

- ```
Integer y = new Integer(x); //Conversie din int in Integer
```

- ```
float m = 3.14f;
```

- ```
Float n = new Float(m); //Conversie din float in Float
```

- ```
String str10 = String.valueOf(x); //Conversie din int in String
```



Limbajul Java – conversii (2)

- Conversia variabilelor referință în variabile primitive. Exemple:
Integer zece = new Integer(10);
int z = zece.intValue(); //Conversie din Integer in int
Double p = new Double(3.14);
double q = p.doubleValue(); //Conversie din *Double* in double
String str20 = "20";
int d = Integer.parseInt(str20); //Conversie din String in int
String strPi = "3.14";
double pi = Double.parseDouble(strPi); //Conversie String - double
- Autoboxing și inboxing (începând cu JDK 1.5). Exemple:
int z = 10;
Integer zece = z; //autoboxing
Double pi = new Double(3.14);
double p = pi; //unboxing



Limbajul Java – comparații (1)

- Variabile primitive (comparație)

```
int a = 5;
```

```
int b = 5;
```

```
System.out.println(a==b ? "a==b" : "a!=b");
```

//Rezultat : "a==b"

- Variabile referință. Dacă cel puțin una din variabilele referință s-a creat cu new, operatorii "=", "!=", "<", "<=", ">" și ">=" nu se pot folosi pentru comparații, fiind necesar apelul metodelor corespunzătoare existente în clasele wrapper.

Exemple:

- Comparație greșită (se compară adrese!):

```
Integer a = new Integer(5);
```

```
Integer b = new Integer(5);
```

```
System.out.println(a==b ? "a==b" : "a!=b");
```

//Rezultat: "a!=b"

- Comparație greșită (se compară o adresă cu o valoare!):

```
Integer a = new Integer(5);
```

```
Integer b = 5;
```

```
System.out.println(a==b ? "a==b" : "a!=b");
```

// "a!=b"



Limbajul Java – comparații (2)

- Comparație corectă (se compară valori!):
Integer a = 5;
Integer b = 5;
System.out.println(a==b ? "a==b" : "a!=b"); // "a==b"
- Comparație corectă (se compară valori!):
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a.equals(b) ? "a==b" : "a!=b"); // Rezultat : "a==b"
- Comparație corectă (se compară valori!):
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a.compareTo(b)==0 ? "a==b" : "a!=b"); // Rezultat: "a==b"
- Comparație corectă (se compară valori!):
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a.intValue()==b.intValue() ? "a==b" : "a!=b"); // Rezultat: "a==b"



Limbajul Java – clasa File (1)

Clasa *File* oferă informații despre fișiere și directoare și permite redenumirea respectiv ștergerea acestora. Principalele operații:

- **length** – returnează dimensiunea în octeți a fișierului;
- **getName** – returnează numele fișierului;
- **getPath / getAbsolutePath / getCanonicalPath** – returnează calea fișierului;
- **getAbsoluteFile / getCanonicalFile** – returnează un obiect File aferent fișierului;
- **isFile / isDirectory** – returnează *true* dacă e fișier / director;
- **canRead / canWrite** – returnează *true* dacă aplicația poate citi / modifica fișierul;
- **exists** – verifică dacă fișierul există;
- **delete** – șterge fișierul sau directorul;
- **deleteOnExit** – șterge fișierul sau directorul la închiderea mașinii virtuale;
- **list** – apelată pentru un director returnează un tablou de tip String cu toate fișierele și subdirectoarele din acel director. Exemplu:

```
File file = new File("e:\\algoritmi_curs");
String str[] = file.list();
for(int i=0; i<str.length; i++)
    System.out.println(str[i]);
```
- **listFile** – apelată pentru un director returnează un tablou de tip File cu toate fișierele și subdirectoarele din acel director;
- **listRoots** – returnează un tablou de tip File reprezentând rădăcinile sistemelor de fișiere disponibile în sistem (ex.: "C:\\", "D:\\", "E:\\");

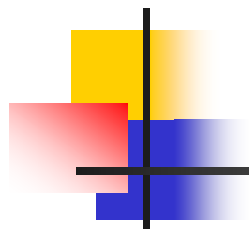


Limbajul Java – clasa File (2)

- **mkdir** – creează un director nou, returnând *true* în caz de succes. Exemplu:

```
File file = new File("algoritmi");  
System.out.println(file.mkdir());
```
- **mkdirs** – creează un director nou, inclusiv directoarele inexistente care apar în cale
- **renameTo** – permite redenumirea fișierelor, returnând *true* în caz de succes. Exemplu:

```
File file = new File("Algoritmi.pdf");  
System.out.println(file.renameTo(new File("algoritmi_curs.pdf")));
```
- **separator / separatorChar** – returnează un String / char reprezentând separatorul dependent de sistem: '/' în UNIX respectiv '\\' în Win32.



Limbajul Java – clasa System

Principalele operații

- **currentTimeMillis** – returnează un *long* reprezentând timpul în milisecunde. Poate fi folosit ca parametru în constructorul clasei *Date* pentru obținerea datei și orei în formatul dorit.
- **exit(0)** – provoacă ieșirea din aplicație.
- **gc** – rulează mecanismul de *garbage collector* pentru eliberarea zonelor de memorie ocupate de obiecte neutilizate.
- **getProperties** – returnează un obiect *Properties* din care se pot extrage diferite informații referitoare la sistemul de operare.



Limbajul Java – clasa Math

Câteva exemple

```
public class Main {  
    public static void main(String[] args) {  
        double pi = Math.PI;  
        System.out.println(pi);                //3.141592653589793  
        System.out.println(Math.round(pi));     //3 - rotunjire la cel mai apropiat intreg  
        System.out.println(Math.floor(pi));     //3.0 - rotunjire in jos  
        System.out.println(Math.ceil(pi));      //4.0 - rotunjire in sus  
        System.out.println(Math.pow(2, 3));     //8.0 - ridicare la putere  
        System.out.println(Math.sqrt(9));       //3.0 - radical  
        System.out.println(Math.exp(1));        //2.7182818284590455 - valoarea exponentiala  
        double e = Math.E;  
        System.out.println(e);                  //2.718281828459045  
        System.out.println(Math.min(2, 8));     //2 - minimul  
        System.out.println(Math.max(2, 8));     //8 - maximul  
        System.out.println(Math.abs(-5));       //5 - valoarea absoluta  
        System.out.println(Math.log(e));        //1.0 - logaritmul natural  
        System.out.println(Math.random());     //valoare aleatoare subunitara  
        System.out.println(Math.sin(pi/2));     //1.0 - sinusul, exista si celelalte functii  
        System.out.println(Math.toDegrees(pi/2)); //90.0 - conversie in grade  
        System.out.println(Math.toRadians(180)); //3.141592653589793 - conv. in rad.  
    }  
}
```

Aplicații propuse

1. Căutarea minimului și a maximului într-un tablou de n elemente;
2. Generarea și afișarea unui tablou cu numerele lui Fibonacci.
3. Căutarea minimului și a maximului într-o matrice;
4. Adunarea a două matrici;
5. Înmulțirea a două matrici.
6. Să se scrie un program care, prin intermediul clasei *File*, să permită navigarea în structura de directoare din sistem.



Limbajul Java – tratarea excepțiilor (1)

- Excepțiile sunt erorile care apar în timpul execuției programelor.
- O instrucțiune sau o secvență de instrucțiuni trebuie inclusă într-un bloc *try* în vederea tratării eventualelor excepții pe care le poate genera. Blocul *try* este urmat de unul sau mai multe blocuri *catch* prin care sunt detectate diferitele excepții. În blocul opțional *finally* se introduc zonele de cod care trebuie executate indiferent că apare sau nu o excepție. Forma generală:

```
try{  
    //instrucțiuni care pot genera Exceptia1 si Exceptia2  
}  
catch(Exceptia1 e1){  
    //instrucțiunile care se executa daca apare Exceptia1  
}  
catch(Exceptia2 e2){  
    //instrucțiunile care se executa daca apare Exceptia2  
}  
finally{  
    //instrucțiunile care se executa indiferent ca apare sau nu o exceptie  
}
```

- Dacă nu apar excepții, blocul *try* execută toate instrucțiunile. Dacă o instrucțiune din blocul *try* generează o excepție, se caută blocul *catch* corespunzător, trecând peste restul instrucțiunilor din *try*. Dacă există un *catch* corespunzător, se execută instrucțiunile din acel bloc *catch*. Dacă nu este găsit un bloc *catch* corespunzător, excepția este transmisă mai departe în ierarhia apelantă. Dacă excepția ajunge în vârful ierarhiei fără să fie tratată, programul afișează un mesaj de eroare și se întrerupe.



Limbajul Java – tratarea excepțiilor (2)

- Este posibilă ignorarea excepțiilor și transmiterea lor mai sus prin *throws* în ierarhia apelantă.
- În exemplul următor, dacă în metoda apare *Exceptia1* sau *Exceptia2*, ea este ignorată, transmisă mai departe și tratată în metoda apelantă *main* (ca în exemplul precedent):

```
public void metoda() throws Exceptia1, Exceptia2 {  
    //instructuni care pot genera Exceptia1 si Exceptia2  
}  
  
public static void main(String[] args) {  
    try{  
        metoda();  
    }  
    catch(Exceptia1 e1){  
        //instructiunile care se executa daca apare Exceptia1  
    }  
    catch(Exceptia2 e2){  
        //instructiunile care se executa daca apare Exceptia2  
    }  
    finally{  
        //instructiunile care se executa indiferent ca apare sau nu o exceptie  
    }  
}
```



Limbajul Java – tratarea excepțiilor (3)

Exemplul 1

- Implementați și apoi rulați următoarea secvență de cod:

```
String str = "I'm a String!";  
int i = Integer.parseInt(str);  
System.out.println("Program execution finished...");
```

Conversia din *String* în *int* determină eșuarea execuției programului datorită conținutului nenumeric al variabilei *str*. Astfel nu se mai ajunge la afișarea mesajului "Program execution finished...". Excepția semnalată este *NumberFormatException*.

- Tratarea excepției:

```
String str = "I'm a String!";  
try {  
    int i = Integer.parseInt(str);  
}  
catch (NumberFormatException nfe) {  
    System.out.println("Conversia nu a fost posibila!");  
}
```

Excepția fiind tratată, execuția programului nu va mai eșua.



Limbajul Java – tratarea excepțiilor (4)

Exemplul 2

- Implementați și apoi rulați următoarea secvență de cod:

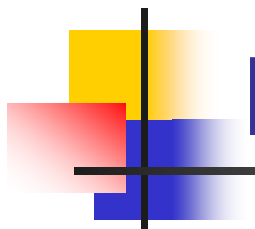
```
int tab[] = new int[10];  
tab[10] = 10;
```

Accesarea unui element inexistent, al 11-lea element, nealocat, determină o excepție `ArrayIndexOutOfBoundsException`.

- Tratarea excepției:

```
int tab[] = new int[10];  
try {  
    tab[10] = 10;  
}  
catch (ArrayIndexOutOfBoundsException aioobe) {  
    System.out.println("Ati accesat un element inexistent!");  
}
```

- Evident, excepția nu apare dacă indexul folosit pentru accesarea tabloului este între 0 și 9.



Limbajul Java – operații I/O (1)

Citirea șirurilor de caractere de la tastatură

- Fluxuri de caractere:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str = null;
try {
    str = br.readLine();
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
```

- Fluxuri de octeți:

```
DataInputStream dis = new DataInputStream(System.in);
String str = null;
try {
    str = dis.readLine();
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
```



Limbajul Java – operații I/O (2)

Citirea valorilor de la tastatură

- Clasa *DataInputStream* pe lângă metoda *readLine* (folosită pe pagina anterioară pentru introducerea șirurilor de caractere) dispune și de funcții pentru citirea valorilor (*readInt*, *readDouble*, etc.). Dar aceste metode sunt funcționale doar pentru valori scrise prin interfața *DataOutput* (*writeInt*, *writeDouble*, etc.).
- Citirea valorilor poate fi efectuată prin citire de șiruri de caractere urmată de conversia acestora:

```
DataInputStream dis = new DataInputStream(System.in);
String str = null;
try {
    str = dis.readLine();
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
int i = Integer.parseInt(str);
```



Limbajul Java – operații I/O (3)

Citirea din fișiere

Următoarea secvență de cod afișează pe ecran toate liniile citite din fișierul *input.txt*. Citirea se face prin aceeași clasă `DataInputStream` care în loc de un `InputStream` va primi ca parametru un `FileInputStream`:

```
FileInputStream fis = null;
try{
    fis = new FileInputStream("input.txt");
}
catch(FileNotFoundException fnfe){
    fnfe.printStackTrace();
}
DataInputStream dis = new DataInputStream(fis);
String str = null;
try{
    while((str = dis.readLine()) != null)
        System.out.println(str);
    dis.close();
    System.in.read();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```



Limbajul Java – operații I/O (4)

Scrierea în fișiere

- Următoarea secvență de program scrie în fișierul *data.txt* întregul 10 și valoarea float 3.14

```
try{
    FileOutputStream fos = new FileOutputStream("data.txt");
    DataOutputStream dos = new DataOutputStream(fos);
    dos.writeInt(10);
    dos.writeFloat(3.14f);
    dos.close();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```

- Fiind scrise prin metodele *writeInt* și *writeFloat*, valorile pot fi citite din fișier folosind metodele *readInt* respectiv *readFloat* ale clasei *DataInputStream*

```
try{
    FileInputStream fis = new FileInputStream("data.txt");
    DataInputStream dis = new DataInputStream(fis);
    System.out.println(dis.readInt());
    System.out.println(dis.readFloat());
    dis.close();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```



Limbajul Java – operații I/O (5)

Clasa Scanner

- Citire de la tastatură (începând cu JDK 1.5):

```
Scanner sc = new Scanner(System.in);
String str = sc.nextLine();
int n = sc.nextInt();
double d = sc.nextDouble();
System.out.println(str + " " + n + " " + d);
sc.close();
```
- Citire din FileInputStream (începând cu JDK 1.5):

```
FileInputStream f = null;
try {
    f = new FileInputStream("data.txt");
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
}
Scanner sc = new Scanner(f);
String str = sc.nextLine();
int n = sc.nextInt();
double d = sc.nextDouble();
System.out.println(str + " " + n + " " + d);
sc.close();
```



Limbajul Java – operații I/O (6)

- Citire din File (începând cu JDK 1.5):

```
File f = new File("data.txt");  
Scanner sc = null;  
try {  
    sc = new Scanner(f);  
} catch (FileNotFoundException ex) {  
    ex.printStackTrace();  
}  
String str = sc.nextLine();  
int n = sc.nextInt();  
double d = sc.nextDouble();  
System.out.println(str + " " + n + " " + d);  
sc.close();
```
- Citirea unui fișier linie cu linie (începând cu JDK 1.5): :

```
FileInputStream f = null;  
try {  
    f = new FileInputStream("input.txt");  
} catch (FileNotFoundException ex) {  
    ex.printStackTrace();  
}  
Scanner sc = new Scanner(f);  
while(sc.hasNext())  
    System.out.println(sc.nextLine());  
sc.close();
```



Limbajul Java – operații I/O (7)

Arhivare ZIP

Arhivarea unui fișier constă în citirea datelor din acel fișier prin *FileInputStream* și scrierea lor în arhivă prin *ZipOutputStream*. Pentru viteză mai mare, transferul datelor se face printr-un tablou de tip *byte*:

```
String fisier = "Algoritmi.pdf";           //fișierul care se arhiveaza
String zip = "Algoritmi.zip";             //numele arhivei
byte buffer[] = new byte[1024];
try{
    FileInputStream fileIn = new FileInputStream(fisier);
    FileOutputStream f = new FileOutputStream(zip);
    ZipOutputStream zipOut = new ZipOutputStream(f);
    zipOut.putNextEntry(new ZipEntry(fisier));
    int nBytes;
    while((nBytes = fileIn.read(buffer, 0, 1024)) != -1)
        zipOut.write(buffer, 0, nBytes);
    zipOut.close();
    fileIn.close();
    f.close();
}
catch(ZipException ze){
    System.out.println(ze.toString());
}
catch(FileNotFoundException fnfe){
    fnfe.printStackTrace();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```

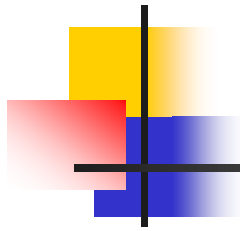



Limbajul Java – operații I/O (8)

Dezarhivare ZIP

Dezarhivarea unui fișier constă în citirea datelor din arhivă prin *ZipInputStream* și scrierea lor prin *FileOutputStream* în fișierul destinație. Pentru eficiență, transferul datelor se face printr-un tablou de tip *byte*:

```
String fisier = "Algoritmi.pdf";           //numele fișierului destinație
String zip = "Algoritmi.zip";             //numele arhivei
byte buffer[] = new byte[1024];
try{
    FileInputStream fileIn = new FileInputStream(zip);
    FileOutputStream fileO = new FileOutputStream(fisier);
    ZipInputStream zipIn = new ZipInputStream(fileIn);
    zipIn.getNextEntry();
    int nBytes;
    while((nBytes = zipIn.read(buffer, 0, 1024)) != -1)
        fileO.write(buffer, 0, nBytes);
    fileIn.close();
    fileO.close();
}
catch(ZipException ze){
    System.out.println(ze.toString());
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```



Limbajul Java – operații I/O (9)

Aplicații propuse

1. Să se scrie un program care cere introducerea numelui utilizatorului și afișează pe ecran mesajul *Hello* urmat de numele introdus (afișare-citire-afișare).
2. Citirea unui întreg până la introducerea unei valori valide (tratând excepția *NumberFormatException*).
3. Să se implementeze un program care citește de la tastatură lungimea unui tablou de tip *String* și elementele acestuia (numele studenților din semigrupă). Sortați în ordine alfabetică tabloul (v. metoda *sort* a clasei *Arrays*). Atenție, șirurile trebuie transformate după citire astfel încât toate literele să fie mici sau toate mari (v. *toLowerCase* sau *toUpperCase* din *String*).
4. Modificați prima aplicație propusă astfel încât să citească și vârsta utilizatorului. Dacă vârsta introdusă depășește 100 să afișeze mesajul "Ești bătrân!", altfel să afișeze "Ești tânăr!";
5. Să se implementeze un program care citește de la tastatură lungimea unui tablou de valori întregi și elementele acestuia. Să se sorteze tabloul folosind metoda *sort* a clasei *Arrays*. Să se afișeze pe ecran tabloul sortat.
6. Să se implementeze un program care citește dintr-un fișier într-un tablou de tip *String* numele studenților din semigrupă. Sortați în ordine alfabetică tabloul (v. metoda *sort* a clasei *Arrays*). Atenție, șirurile trebuie transformate după citire astfel încât toate literele să fie mici sau toate mari (v. *toLowerCase* sau *toUpperCase* din *String*).
7. Să se citească dintr-un fișier un tablou de valori întregi și să se afișeze pe ecran media lor aritmetică. Separarea valorilor de pe linii se va face prin clasa *StringTokenizer* prezentată anterior.
8. Studiați clasa *RandomAccessFile* care, spre deosebire de *FileInputStream* și *FileOutputStream* (acces secvențial), permite citirea respectiv scrierea unei anumite locații din fișier.



Internaționalizare și naționalizare

Java oferă suportul prin numeroase clase pentru internaționalizarea și naționalizarea aplicațiilor

- **Clasa Locale**

```
Locale locale = Locale.getDefault();
System.out.println(locale.getCountry());           //RO
System.out.println(locale.getDisplayCountry());    //România
System.out.println(locale.getDisplayCountry(Locale.FRANCE)); //Roumanie
System.out.println(locale.getDisplayLanguage());   //română
System.out.println(locale.getDisplayLanguage(Locale.FRANCE)); //roumain
System.out.println(locale.getDisplayName());        //română (România)
System.out.println(locale.getLanguage());          //ro
Locale g = Locale.GERMANY;
System.out.println(g.getLanguage());              //de
```

- **Clasa NumberFormat**

```
System.out.println(NumberFormat.getInstance(locale).format(12.3)); //12,3
System.out.println(NumberFormat.getCurrencyInstance(locale).format(6.8)); //6,80 LEI
```

- **Clasa DateFormat**

```
Date date = new Date(2009-1900, 8-1, 31);           //Y-1900, M-1, D
System.out.println(DateFormat.getDateInstance(0, locale).format(date)); //31 august 2009
System.out.println(DateFormat.getDateInstance(2, locale).format(date)); //31.08.2009
```



Limbajul Java – crearea claselor (1)

- Definiția unei clase trebuie să conțină cuvântul cheie *class*, numele clasei și corpul clasei. Opțional, înainte de cuvântul *class* pot fi folosiți modificatorii *public* (clasa este vizibilă în toate pachetele), *abstract* (v. clase abstracte) sau *final* (clasa nu poate fi derivată). Exemplu:

```
class Person {  
    private String name;  
    public void setName(String name){  
        this.name = name;  
    }  
    public String getName(){  
        return name;  
    }  
}
```
- Clasele pot conține:
 - Variabile membru;
 - Constructori;
 - Metode.
- Tipuri de acces: *private*, *protected*, *public*
 - Variabilele și metodele private pot fi accesate doar în cadrul clasei în care s-au declarat;
 - Variabilele și metodele protejate pot fi accesate în interiorul clasei, al subclasselor (v. moștenirea) sau în pachetul (*package*) în care au fost declarate;
 - Variabilele și metodele publice pot fi accesate oriunde;
 - Tipul de acces este implicit protejat (dacă nu se precizează).
- Recomandare (încapsulare, v. cursul POO):
 - Variabile private;
 - Set de metode publice care să permită accesarea variabilelor private.



Limbajul Java – crearea claselor (2)

Exemplul 1 (un singur pachet)

```
package person;

public class Person {
    private String name;
    private String address;
    private int age;
    private void setName(String name){
        this.name = name;
    }
    private String getName(){
        return name;
    }
    protected void setAddress(String address){
        this.address = address;
    }
    protected String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    int getAge(){
        return age;
    }
}
```

```
package person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "Popescu"; //Eroare!
        p.setName("Popescu"); //Eroare!
        p.setAddress("Str. N. Balcescu, Nr. 5");
        p.setAge(103);
        System.out.println("Numele: " + p.getName()); //Eroare!
        System.out.println("Adresa: " + p.getAddress());
        System.out.println("Varsta: " + p.getAge());
    }
}
```



Limbajul Java – crearea claselor (3)

Exemplul 2 (pachete diferite, clasa de bază nepublică)

```
package person;

class Person {
    private String name;
    private String address;
    private int age;
    private void setName(String name){
        this.name = name;
    }
    private String getName(){
        return name;
    }
    protected void setAddress(String address){
        this.address = address;
    }
    protected String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    int getAge(){
        return age;
    }
}
```

```
package main;
import person.Person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
    }
}
```

//Eroare!



Limbajul Java – crearea claselor (4)

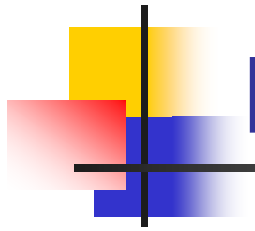
Exemplul 3 (pachete diferite, clasa de bază publică)

```
package person;

public class Person {
    private String name;
    private String address;
    private int age;
    private void setName(String name){
        this.name = name;
    }
    private String getName(){
        return name;
    }
    protected void setAddress(String address){
        this.address = address;
    }
    protected String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    int getAge(){
        return age;
    }
}
```

```
package main;
import person.Person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "Popescu"; //Eroare!
        p.setName("Popescu"); //Eroare!
        p.setAddress("Str. N. Balcescu, Nr. 5"); //Eroare!
        p.setAge(103);
        System.out.println("Numele: " + p.getName()); //Eroare!
        System.out.println("Adresa: " + p.getAddress()); //Eroare!
        System.out.println("Varsta: " + p.getAge()); //Eroare!
    }
}
```



Limbajul Java – crearea claselor (5)

Exemplul 4 (o variantă corectă)

```
package person;

public class Person {
    private String name;
    private String address;
    private int age;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

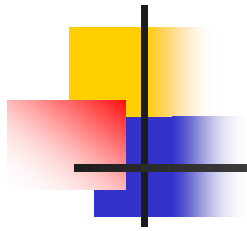
```
package person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Popescu");
        p.setAddress("Str. N. Balcescu, Nr. 5");
        p.setAge(103);
        System.out.println("Numele: " + p.getName());
        System.out.println("Adresa: " + p.getAddress());
        System.out.println("Varsta: " + p.getAge());
    }
}
```




Limbajul Java – constructori (1)

- Este o metodă publică fără tip, având același nume cu cel al clasei;
- Principalul rol al constructorului este acela de a inițializa obiectul pentru care s-a apelat;
- Dacă într-o clasă nu se declară un constructor, atunci compilatorul generează unul implicit, fără parametri;
- Într-o clasă pot coexista mai mulți constructori cu parametri diferiți (tip, număr);
- La instanțierea unui obiect se apelează constructorul corespunzător parametrilor de apel;
- În Java nu există destructori. Sistemul apelează periodic mecanismul *garbage collector*.



Limbajul Java – constructori (2)

Exemplu

```
public class Person {
    private String name;
    private String address;
    private int age;
    public Person(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person("Pop", "Sibiu", 103);
        System.out.println("Numele: " + p.getName());    //Pop
        System.out.println("Adresa: " + p.getAddress()); //Sibiu
        System.out.println("Varsta: " + p.getAge());      //103
        p.setName("Rus");
        p.setAddress("Avrig");
        p.setAge(98);
        System.out.println("Numele: " + p.getName());    //Rus
        System.out.println("Adresa: " + p.getAddress()); //Avrig
        System.out.println("Varsta: " + p.getAge());      //98
    }
}
```



Limbajul Java – moștenirea (1)

- Unul dintre marile avantaje ale programării orientate-obiect (POO) constă în reutilizarea codului;
- Toate clasele Java, cu excepția interfețelor, sunt subclase ale clasei rădăcină *Object*;
- Procesul prin care o clasă nouă refolosește o clasă veche se numește moștenire;
- O clasă, numită clasă derivată, poate moșteni variabilele și metodele unei alte clase, numită clasă de bază;
- Dacă apare aceeași metodă cu aceiași parametri atât în clasa de bază cât și în cea derivată (prin redefinire) și se apelează metoda instanței clasei derivate, se execută metoda clasei derivate;
- Derivarea unei clase în Java se face prin cuvântul cheie *extends* urmat de numele clasei de bază;
- O clasă declarată *final* nu poate fi derivată;
- Java nu permite moștenire multiplă: o clasă poate deriva o singură clasă de bază. Moștenirea multiplă este permisă doar folosind interfețele (vor fi prezentate);
- Constructorii clasei derivate pot folosi printr-un apel *super* constructorii clasei de bază pentru inițializarea variabilelor moștenite;
- În exemplul prezentat în continuare, clasele Student și Teacher moștenesc clasa Person;



Limbajul Java – moștenirea (2)

Exemplu

```
public class Person {
    private String name;
    private String address;
    private int age;
    public Person(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
public class Student extends Person {
    private int grade;           //medie
    public Student(String name, String address, int age, int grade) {
        super(name, address, age); //trebuie sa fie prima instructiune!
        this.grade = grade;
    }
    public void setGrade(int grade){
        this.grade = grade;
    }
    public int getGrade(){
        return grade;
    }
}

public class Teacher extends Person {
    private int courses;         //nr. cursuri
    public Teacher(String name, String address, int age, int courses) {
        super(name, address, age); //trebuie sa fie prima instructiune!
        this.courses = courses;
    }
    public void setCourses(int courses){
        this.courses = courses;
    }
    public int getCourses(){
        return courses;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Rus", "Avrig", 98, 4);
        System.out.println(s.getAge());           //98
    }
}
```



Limbajul Java – moștenirea (3)

- Dacă un constructor al clasei derivate nu apelează prin *super* un constructor al clasei de bază, compilatorul va apela implicit constructorul fără parametri al clasei de bază. Dacă însă clasa de bază are doar constructori cu parametri, compilatorul nu va genera constructorul implicit. În acest caz, în clasa de bază trebuie declarat și un constructor fără parametri.
- Exemplu:

```
public class Base {  
    public Base(){  
    }  
    public Base(int p) {  
    }  
}
```

```
public class Sub extends Base{  
    public Sub() {  
    }  
}
```



Limbajul Java – moștenirea (4)

Supraîncărcarea metodelor

```
public class Base {
    public void method(int i) {
        System.out.println("Base integer is " + i);
    }
    public void method(String s){
        System.out.println("Base string is " + s);
    }
}

public class Sub extends Base {
    public void method(int j){
        System.out.println("Sub integer is " + j);
    }
    public static void main(String[] args) {
        Base b1 = new Base();
        Base b2 = new Sub();
        Sub s = new Sub();
        b1.method(1);           //"Base integer is 1"
        b2.method(2);           //"Sub integer is 2"
        s.method(3);            //"Sub integer is 3"
        s.method("4");          //"Base string is 4"
    }
}
```



Limbajul Java – moștenirea (5)

Aplicații

1. Să se implementeze o aplicație care să permită introducerea de la tastatură a numărului de studenți din semigrupă urmat de datele lor (într-un tablou) și a numărului de profesori din acest semestru urmat de datele lor (într-un alt tablou). Căutați și afișați studentul cu media cea mai mare respectiv profesorul cu cele mai multe cursuri.
2. Definiți clasa **Employee** (angajat) derivată din clasa **Person** prezentată, având câmpul suplimentar **salary**. Introduceți de la tastatură 5 obiecte de tip **Employee** într-un tablou. Căutați și afișați angajatul cu salariul cel mai mare.
3. Definiți clasa **Product** cu câmpul **price** (preț). Definiți clasa **Book** (carte) derivată din clasa **Product** având câmpurile suplimentare **title**, **author** și **publisher** (editor). Introduceți de la tastatură 5 obiecte de tip **Book** într-un tablou. Căutați și afișați cartea cu prețul cel mai mic.
4. Definiți clasa **Book** (carte) cu câmpurile **title**, **author** și **publisher** (editor). Definiți clasa **LibraryCard** (fișă de bibliotecă) derivată din clasa **Book** având câmpul suplimentar **copies** (nr. exemplare). Introduceți de la tastatură 5 obiecte de tip **LibraryCard** într-un tablou. Căutați și afișați cartea cu numărul cel mai mare de exemplare.
5. Definiți clasa **Vehicle** cu câmpul **manufacturer** (fabricant). Definiți clasa **Car** derivată din clasa **Vehicle** având câmpurile suplimentare **model**, **length** (lungime), **maxSpeed** (viteză maximă), **price** (preț). Introduceți de la tastatură 5 obiecte de tip **Car** într-un tablou. Căutați și afișați mașina cu prețul cel mai mic.



Limbajul Java – clase abstracte

- O clasă trebuie declarată abstractă dacă conține cel puțin o metodă abstractă.
- Metodele abstracte sunt declarate fără implementare urmând ca ele să fie implementate în clasele derivate. O clasă abstractă nu poate fi instanțiată.
- Exemplu:

```
public abstract class Product {
    private double price;
    public Product(double price) {
        this.price = price;
    }
    public abstract double computeFinalPrice();
    public void setPrice(double price){
        this.price = price;
    }
    public double getPrice(){
        return price;
    }
}

public class Book extends Product {
    public Book(double price) {
        super(price);
    }
    public double computeFinalPrice(){ //TVA=9%
        return getPrice() + (9*getPrice())/100;
    }
}
```

```
public class Car extends Product{
    public Car(double price) {
        super(price);
    }
    public double computeFinalPrice(){ //TVA=19%
        return getPrice() + (19*getPrice())/100;
    }
}

public class Main {
    public static void main(String[] args) {
        Product p = new Product(10); //Eroare!
        Book b = new Book(100);
        System.out.println(b.computeFinalPrice());
        Car c = new Car(100000);
        System.out.println(c.computeFinalPrice());
    }
}
```




Limbajul Java – interfețe (1)

- Interfețele oferă avantajele moștenirii multiple, evitând complicațiile acesteia.
- Toate metodele dintr-o interfață sunt implicit publice și abstracte. O interfață furnizează numele metodelor fără implementarea lor. O interfață care conține o singură metodă se numește funcțională.
- Interfețele nu pot fi instanțiate.
- Clasa care folosește o interfață trebuie să implementeze toate metodele acesteia.
- Se poate deriva o interfață din alte interfețe prin același mecanism care a fost prezentat la clase (v. moștenirea).
- În continuare vor fi prezentate câteva exemple:
 - O interfață *Price* folosită de clasele *Product*, *Book* și *Car*.
 - Folosirea interfețelor *Comparator* sau *Comparable* pentru sortarea unui tablou de obiecte prin metoda *sort* din clasa *Arrays*. În cazul interfeței *Comparator* trebuie implementată metoda *compare*.
 - Interfața *Serializable* pentru serializarea obiectelor.



Limbajul Java – interfețe (2)

Exemplul 1 (interfața *Price*, implementare *computeFinalPrice*)

```
public interface Price {
    public double computeFinalPrice();
}

public abstract class Product implements Price{
    private double price;
    public Product(double price) {
        this.price = price;
    }
    public void setPrice(double price){
        this.price = price;
    }
    public double getPrice(){
        return price;
    }
}

public class Book extends Product {
    public Book(double price) {
        super(price);
    }
    public double computeFinalPrice(){ //TVA=9%
        return getPrice() + (9*getPrice())/100;
    }
}

public class Car extends Product{
    public Car(double price) {
        super(price);
    }
    public double computeFinalPrice(){ //TVA=19%
        return getPrice() + (19*getPrice())/100;
    }
}

public class Main {
    public static void main(String[] args) {
        Product p = new Product(10); //Eroare!
        Book b = new Book(100);
        System.out.println(b.computeFinalPrice());
        Car c = new Car(100000);
        System.out.println(c.computeFinalPrice());
    }
}
```



Limbajul Java – interfețe (3)

Exemplul 2 (interfața *Comparator*, implementare *compare*)

```
public class Item {  
    String str;  
    int num;  
    Item(String str, int num) {  
        this.str = str;  
        this.num = num;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Item t[] = new Item[5];  
        t[0] = new Item("c", 2);  
        t[1] = new Item("a", 1);  
        t[2] = new Item("e", 4);  
        t[3] = new Item("b", 5);  
        t[4] = new Item("d", 3);  
        //Sortare dupa campul num  
        java.util.Arrays.sort(t, new Comparator() {  
            public int compare(Object o1, Object o2) {  
                //return ((Item)o1).num-((Item)o2).num;  
                if(((Item)o1).num<((Item)o2).num) return -1;  
                if(((Item)o1).num>((Item)o2).num) return 1;  
                return 0;  
            }  
        });  
        //Sortare dupa campul str  
        java.util.Arrays.sort(t, new Comparator() {  
            public int compare(Object o1, Object o2) {  
                return ((Item)o1).str.compareTo(((Item)o2).str);  
            }  
        });  
    }  
}
```



Limbajul Java – interfețe (4)

Exemplul 3 (interfața *Comparator*, implementare *compare*)

```
public class Item implements Comparator {
    String str;
    int num;
    Item() {}
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
    public int compare(Object o1, Object o2) {
        return ((Item)o1).num-((Item)o2).num;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2);
        t[1] = new Item("a", 1);
        t[2] = new Item("e", 4);
        t[3] = new Item("b", 5);
        t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t, new Item());
        //Afisare
        for (int i=0; i<t.length; i++)
            System.out.println(t[i].str + " - " + t[i].num);
        //Sortare dupa campul str
        java.util.Arrays.sort(t, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Item)o1).str.compareTo(((Item)o2).str);
            }
        });
        //Afisare
        for (int i=0; i<t.length; i++)
            System.out.println(t[i].str + " - " + t[i].num);
    }
}
```



Limbajul Java – interfețe (5)

Exemplul 4 (interfața *Comparable*, implementare *compareTo*)

```
public class Item implements Comparable {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
    public int compareTo(Object o) {
        return num-((Item)o).num;
        //return str.compareTo(((Item)o).str);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2);
        t[1] = new Item("a", 1);
        t[2] = new Item("e", 4);
        t[3] = new Item("b", 5);
        t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t);
        //Afisare
        for (int i=0; i<t.length; i++)
            System.out.println(t[i].str + " - " + t[i].num);
        //Sortare dupa campul str
        java.util.Arrays.sort(t, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Item)o1).str.compareTo(((Item)o2).str);
            }
        });
        //Afisare
        for (int i=0; i<t.length; i++)
            System.out.println(t[i].str + " - " + t[i].num);
    }
}
```



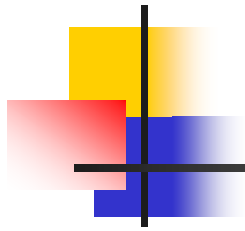
Limbajul Java – interfețe (6)

Exemplul 5 (interfața *Serializable* – permite serializarea obiectelor)

```
public class Item implements Serializable {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        FileInputStream fis = null;
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try{
            fos = new FileOutputStream("data.txt");
            fis = new FileInputStream("data.txt");
        }
        catch(FileNotFoundException e){
            e.printStackTrace();
        }
    }
}
```

```
try{
    oos = new ObjectOutputStream(fos);
    ois = new ObjectInputStream(fis);
}
catch(IOException e){
    e.printStackTrace();
}
try{
    oos.writeObject(new Item("c", 2)); //Scrierea obiectului in fisier
    oos.close(); fos.close();
}
catch(IOException e){
    e.printStackTrace();
}
try{
    Item item=(Item)ois.readObject(); //Citirea obiectului din fisier
    System.out.println(item.str + " - " + item.num);
    ois.close(); fis.close();
}
catch(IOException e){
    e.printStackTrace();
}
catch(ClassNotFoundException e){
    e.printStackTrace();
}
}
```



Limbajul Java – interfețe (7)

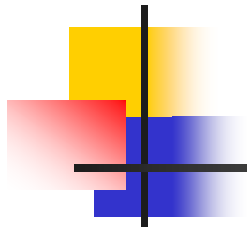
Expresii lambda

- Disponibile din JDK 1.8, expresiile lambda pot fi folosite pentru a crea instanțe ale unor interfețe funcționale [Blo17].
- Pentru expresiile lambda s-a introdus noul operator `->` în Java.
- Expresiile lambda sunt compuse din două părți. Partea din stânga operatorului `->` specifică lista parametrilor separați prin virgulă, iar partea din dreapta este corpul lambda care conține operațiile efectuate de expresia lambda. Spre exemplu, expresia lambda

`(a,b)->a+b`

- calculează suma parametrilor `a` și `b`. În cazul în care în partea dreaptă a expresiei lambda este un bloc de instrucțiuni, acesta trebuie furnizat între acolade:

`(a,b)->{return a+b;}`



Limbajul Java – interfețe (8)

Exemplul 1 (interfața *Arithmetic*, aplicare operații aritmetice)

```
public interface Arithmetic {  
    int operation(int a, int b);  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // TODO code application logic here  
        Arithmetic addition = (a,b)->a+b;  
        System.out.println(addition.operation(6,2)); //8  
        Arithmetic subtraction = (a,b)->a-b;  
        System.out.println(subtraction.operation(6,2)); //4  
        Arithmetic multiplication = (a,b)->a*b;  
        System.out.println(multiplication.operation(6,2)); //12  
        Arithmetic division = (a,b)->a/b;  
        System.out.println(division.operation(6,2)); //3  
    }  
}
```




Limbajul Java – interfețe (9)

Exemplul 2 (interfața *Comparator*, sortare folosind expresii lambda)

```
public class Item {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}

public class Comparisons {
    public int compareByStr(Item a, Item b){
        return a.str.compareTo(b.str);
    }
    public int compareByNum(Item a, Item b){
        return a.num-b.num;
    }
}

public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5]; Comparisons c = new Comparisons();
        t[0] = new Item("c", 2); t[1] = new Item("a", 1);
        t[2] = new Item("e", 4); t[3] = new Item("b", 5); t[4] = new Item("d", 3);
        java.util.Arrays.sort(t,(o1,o2) -> c.compareByNum(o1, o2)); //Sortare dupa campul num
        java.util.Arrays.sort(t,(o1,o2) -> c.compareByStr(o1, o2)); //Sortare dupa campul str
    }
}
```



Limbajul Java – interfețe (10)

Exemplul 3 (Interfața *Comparator*, fără clasa *Comparisons* din exemplul 2)

```
public class Item {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}

public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2); t[1] = new Item("a", 1);
        t[2] = new Item("e", 4); t[3] = new Item("b", 5); t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t,(o1,o2) -> {return o1.num-o2.num;});
        //Sortare dupa campul str
        java.util.Arrays.sort(t,(o1,o2) -> {return o1.str.compareTo(o2.str);});
    }
}
```

Afișarea se poate face cu ajutorul buclei:

```
for(Item i:t)
    System.out.println(i.str + " " + i.num);
```



Limbajul Java – interfețe (11)

Aplicații

1. Definiți clasa **Person** cu câmpurile **name**, **address** și **age**. Definiți clasa **Student** derivată din clasa **Person** având câmpul suplimentar **grade** (medie). Introduceți dintr-un fișier *student.txt* numărul de studenți din semigrupă urmat de datele lor, într-un tablou. Sortați și afișați studenții în ordinea crescătoare a mediei.
2. Definiți clasa **Person** cu câmpurile **name**, **address** și **age**. Definiți clasa **Teacher** derivată din clasa **Person** având câmpul suplimentar **courses** (nr. cursuri). Introduceți dintr-un fișier *teacher.txt* numărul de profesori din acest semestru urmat de datele lor, într-un tablou. Sortați și afișați profesorii în ordinea crescătoare a numărului de cursuri predate.
3. Definiți clasa **Person** cu câmpurile **name**, **address** și **age**. Definiți clasa **Employee** (angajat) derivată din clasa **Person** având câmpul suplimentar **salary** (salariu). Introduceți dintr-un fișier *employee.txt* numărul de angajați urmat de datele lor, într-un tablou. Sortați și afișați angajații în ordinea crescătoare a salariilor.
4. Definiți clasa **Product** cu câmpul **price** (preț). Definiți clasa **Book** (carte) derivată din clasa **Product** având câmpurile suplimentare **title**, **author** și **publisher** (editor). Introduceți dintr-un fișier *book.txt* numărul de cărți urmat de datele lor, într-un tablou. Sortați și afișați cărțile în ordinea crescătoare a prețului.
5. Definiți clasa **Book** (carte) cu câmpurile **title**, **author** și **publisher** (editor). Definiți clasa **LibraryCard** (fișă de bibliotecă) derivată din clasa **Book** având câmpul suplimentar **copies** (nr. exemplare). Introduceți dintr-un fișier *librarycard.txt* numărul de fișe urmat de datele acestora, într-un tablou. Sortați și afișați cărțile în ordinea crescătoare a numărului de exemplare disponibile.
6. Definiți clasa **Vehicle** cu câmpul **manufacturer** (fabricant). Definiți clasa **Car** derivată din clasa **Vehicle** având câmpurile suplimentare **model**, **length** (lungime), **maxSpeed** (viteză maximă), **price** (preț). Introduceți dintr-un fișier *car.txt* numărul de mașini, urmat de datele lor, într-un tablou. Sortați și afișați mașinile în ordinea crescătoare a prețului.



Limbajul Java – partajarea datelor (1)

- Partajarea datelor de către obiectele unei clase poate fi realizată prin intermediul membrilor statici ai clasei respective.
- În timp ce variabilele obișnuite (non-stactice) aparțin instanțelor clasei (obiectelor), variabilele declarate statice aparțin clasei și sunt partajate de către toate obiectele acesteia.
- Unei variabile statice i se alocă memorie o singură dată, la prima instanțiere a clasei. La următoarele instanțieri ale clasei variabilei statice nu i se mai alocă memorie, dar toate obiectele clasei pot accesa aceeași variabilă statică, alocată la prima instanțiere.
- Metodele statice pot fi apelate fără instanțierea clasei.
- Metodele statice nu pot utiliza variabile și metode non-stactice.
- Un exemplu de utilizare a unei variabile statice este contorizarea obiectelor instanțiate dintr-o clasă:

```
public class Person {  
    private String name;  
    static int counter;  
    public Person(String name) {  
        this.name = name;  
        counter++;  
        System.out.println(counter + " persoane.");  
    }  
    public void setName(String name){  
        this.name = name;  
    }  
    public String getName(){  
        return name;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Person p1 = new Person("Popescu");    //"1 persoane."  
        Person p2 = new Person("Ionescu");    //"2 persoane."  
    }  
}
```



Limbajul Java – partajarea datelor (2)

Exemple de folosire a metodelor statice:

```
public class Person {
    private String name;
    private int age;
    static int counter;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        increaseCounter();
    }
    public static void increaseCounter(){
        counter++;
        System.out.println(counter + " persoane.");
    }
    public static void increaseAge(){
        age++; //Eroare!
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Popescu", 61); // "1 persoane."
        Person p2 = new Person("Ionescu", 72); // "2 persoane."
        Person.increaseCounter(); // "3 persoane."
        p1 = new Person("Petrescu", 53); // "4 persoane."
    }
}
```



Limbajul Java – partajarea datelor (3)

Alte exemple de folosire a variabilelor și metodelor statice:

```
public class Person {
    private String name;
    private int age;
    static int counter;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        increaseCounter();
    }
    public static void increaseCounter(){
        counter++;
        System.out.println(counter + " persoane.");
    }
    public void printCounter(){
        System.out.println(counter);
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
public static void main(String[] args) {
    Person p1 = new Person("Popescu", 61);    //"1 persoane."
    Person p2 = new Person("Ionescu", 72);    //"2 persoane."
    Person.increaseCounter() ;                //"3 persoane."
    p1 = new Person("Petrescu", 53);          //"4 persoane."
    printCounter();                            //Eroare!
    p2.printCounter();                        //"4"
}
}
```



Limbajul Java – atribuire vs. clonare (1)

Folosirea operatorului de atribuire

În cazul obiectelor operatorul de atribuire determină atribuirea referințelor (adreselor)

```
public class Person {  
    private String name;  
    private int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public int getAge() {  
        return age;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Person p1 = new Person("Pop", 61);  
        Person p2 = p1;  
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Pop, 61"  
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Pop, 61"  
        p1.setAge(p1.getAge() + 1);  
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Pop, 62"  
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Pop, 62"  
        p2.setName("Rus");  
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Rus, 62"  
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Rus, 62"  
        p1 = new Person("Lup", 53);  
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Lup, 53"  
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Rus, 62"  
        p1.setAge(p1.getAge() + 1);  
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Lup, 54"  
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Rus, 62"  
    }  
}
```

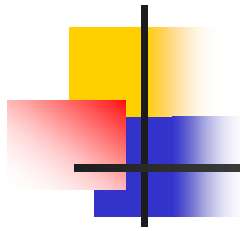


Limbajul Java – atribuire vs. clonare (2)

Clonarea obiectelor

```
public class Person implements Cloneable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Object clone() { //protected in Object
        Object obj = null;
        try {
            obj = super.clone();
        }
        catch (CloneNotSupportedException ex) {
        }
        return obj;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Pop", 61);
        Person p2 = (Person)p1.clone();
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Pop, 61"
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Pop, 61"
        p1.setAge(p1.getAge() + 1);
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Pop, 62"
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Pop, 61"
        p2.setName("Rus");
        System.out.println(p1.getName() + ", " + p1.getAge());    //"Pop, 62"
        System.out.println(p2.getName() + ", " + p2.getAge());    //"Rus, 61"
    }
}
```

Limbajul Java – interfață grafică (1)

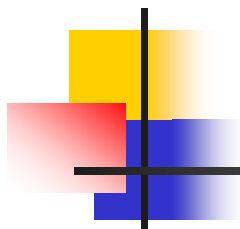
- O componentă foarte importantă a aplicațiilor moderne este interfața grafică prin care acestea comunică cu utilizatorul: se citesc datele și se afișează rezultatele.
- Mediile de dezvoltare Java oferă numeroase pachete cu diverse componente vizuale.
- În următoarea aplicație vom folosi componente din pachetul AWT (Advanced Windowing Toolkit) pentru citirea studenților într-o listă
- Fereastra neredimensionabilă a aplicației este prezentată pe pagina următoare.

Limbajul Java – interfață grafică (2)

Fereastra aplicației



- Numele studentului se introduce printr-un *TextField*;
- Acționarea butonului *Add* determină adăugarea textului din *TextFieId* în componenta *List* respectiv ștergerea din *TextFieId*;
- Adăugarea în componenta *List* determină apariția unui scroll atunci când numărul de linii introduse trece de limita de vizualizare.



Limbajul Java – interfață grafică (3)

Codul sursă al aplicației

```
public class StudentFrame extends Frame {
    private List studentList = new List();
    private TextField studentTextField = new TextField();
    private Button addButton = new Button();

    public StudentFrame() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        show();
    }

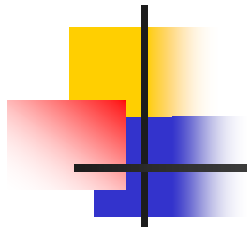
    public static void main(String[] args) {
        StudentFrame studentFrame = new StudentFrame();
    }

    private void jbInit() throws Exception {
        this.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                this._windowClosing(e);
            }
        });
        studentList.setBounds(new Rectangle(210, 60, 160, 140));
        studentTextField.setBounds(new Rectangle(25, 60, 160, 30));
        addButton.setLabel("Add");
        addButton.setBounds(new Rectangle(70, 120, 80, 25));
```

```
        addButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                addButton_actionPerformed(e);
            }
        });
        this.setSize(400, 220);
        this.setResizable(false);
        this.setLayout(null); //componentele pot fi asezate cu mouse-ul
        this.setTitle("Students");
        this.setBackground(new Color(240, 240, 240));
        this.add(addButton, null);
        this.add(studentTextField, null);
        this.add(studentList, null);
    }

    void this_windowClosing(WindowEvent e) {
        System.exit(0);    //inchiderea aplicatiei
    }

    void addButton_actionPerformed(ActionEvent e) {
        studentList.add(studentTextField.getText());
        studentTextField.setText("");    //stergere
    }
}
```



Limbajul Java – interfață grafică (4)

Tratarea evenimentelor

- Evenimentele sunt generate de acțiunile utilizatorului asupra componentelor interfeței grafice.
- Pentru tratarea unui anumit eveniment într-o clasă, trebuie implementată interfața *Listener* corespunzătoare care să prelucreze evenimentul respectiv. De exemplu, pentru tratarea evenimentelor *ActionEvent*, se implementează interfața *ActionListener*.
- Astfel, evenimentele sunt recepționate de obiecte de tip *Listener* care apelează metode de tratare a evenimentelor.
- Aplicația anterioară tratează evenimentele de închidere a ferestrei și de click pe butonul *Add*.
- În continuare vom extinde aplicația astfel încât să răspundă la apăsarea tastei ENTER, tratând evenimentul *KeyEvent*. Pentru asta se poate folosi interfața *KeyListener* implementând toate metodele (*keyPressed*, *keyReleased*, *keyTyped*) sau clasa abstractă *KeyAdapter* care ne permite să implementăm doar metodele necesare (în cazul nostru *keyPressed*).



Limbajul Java – interfață grafică (5)

```
public class StudentFrame extends Frame {
    private List studentList = new List();
    private TextField studentTextField = new TextField();
    private Button addButton = new Button();

    public StudentFrame() {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        show();
        studentTextField.requestFocus();
    }

    public static void main(String[] args) {
        StudentFrame studentFrame = new StudentFrame();
    }

    private void jbInit() throws Exception {
        this.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                this_windowClosing(e);
            }
        });
        studentList.setBounds(new Rectangle(210, 60, 160, 140));
        studentTextField.setBounds(new Rectangle(25, 60, 160, 30));
        studentTextField.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                studentTextField_keyPressed(e);
            }
        });
        addButton.setLabel("Add");
        addButton.setBounds(new Rectangle(70, 120, 80, 25));
```

```
        addButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                addButton_actionPerformed(e);
            }
        });
        this.setSize(400, 220);
        this.setResizable(false);
        this.setLayout(null); //componentele pot fi asezate cu mouse-ul
        this.setTitle("Students");
        this.setBackground(new Color(240, 240, 240));
        this.add(addButton, null);
        this.add(studentTextField, null);
        this.add(studentList, null);
    }

    void this_windowClosing(WindowEvent e) {
        System.exit(0); //inchiderea aplicatiei
    }

    void addButton_actionPerformed(ActionEvent e) {
        studentList.add(studentTextField.getText());
        studentTextField.setText(""); //stergere
        studentTextField.requestFocus();
    }

    void studentTextField_keyPressed(KeyEvent e) {
        if(e.getKeyCode()==e.VK_ENTER)
            addButton_actionPerformed(new ActionEvent(this, 0, null));
    }
}
```



Limbajul Java – interfață grafică (6)

Layout manager (aranjarea componentelor)

- Toate obiectele grafice de tip container (*Frame*, *Panel*, etc.) au o proprietate layout prin care se specifică cum să fie așezate și dimensionate componentele: *null*, *XYLayout*, *BorderLayout*, *FlowLayout*, *GridLayout*, *GridBagLayout*, etc.
- Un layout *null* (folosit și în aplicația prezentată) sau *XYLayout* păstrează pozițiile și dimensiunile originale ale componentelor. Se pot folosi în cazul containerelor neredimensionabile.
- *BorderLayout*, *FlowLayout*, *GridLayout* și *GridBagLayout* repoziționează și/sau rescalează obiectele în cazul redimensionării containerului.

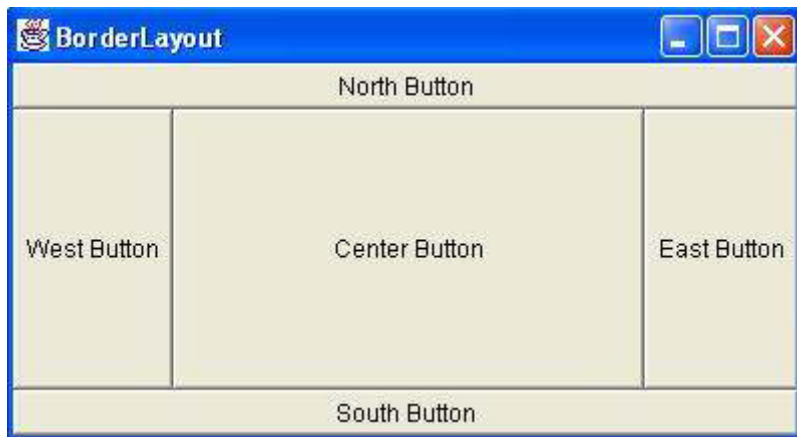
Limbajul Java – interfață grafică (7)

BorderLayout permite aranjarea componentelor din container în 5 zone: nord, sud, est, vest și centru. Componentele din nord și sud își păstrează înălțimea originală și sunt scalate la lățimea containerului. Componentele din est și vest își păstrează lățimea originală și sunt scalate vertical astfel încât să acopere spațiul dintre nord și sud. Componenta din centru se expandează pentru acoperirea întreg spațiului rămas.

Constructori:

`BorderLayout()` – fără distanță între componente;

`BorderLayout(int hgap, int vgap)` – primește ca parametri distanțele orizontale și verticale dintre componente.



```
public class BorderFrame extends Frame {
    private BorderLayout bLayout = new BorderLayout();
    private Button northButton = new Button("North Button");
    private Button southButton = new Button("South Button");
    private Button westButton = new Button("West Button");
    private Button eastButton = new Button("East Button");
    private Button centerButton = new Button("Center Button");

    public BorderFrame() {
        this.setSize(400, 220);
        this.setTitle("BorderLayout");
        this.setLayout(bLayout);
        this.setBackground(new Color(240, 240, 240));
        this.add(northButton, BorderLayout.NORTH);
        this.add(southButton, BorderLayout.SOUTH);
        this.add(westButton, BorderLayout.WEST);
        this.add(eastButton, BorderLayout.EAST);
        this.add(centerButton, BorderLayout.CENTER);
        this.show();
    }

    public static void main(String[] args) {
        new BorderFrame();
    }
}
```

Limbajul Java – interfață grafică (8)

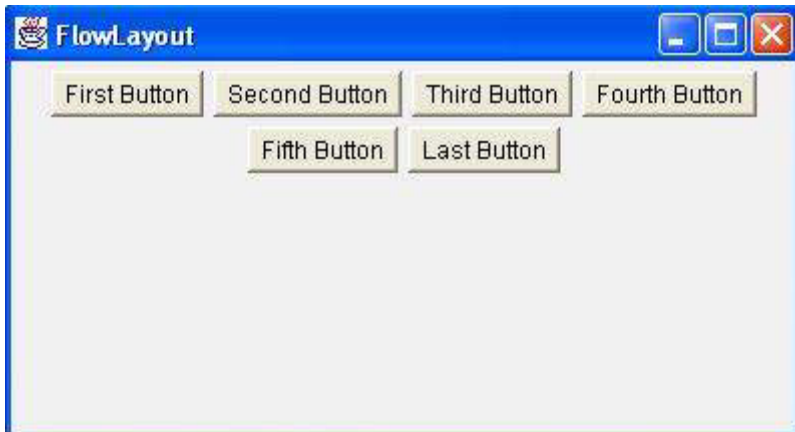
FlowLayout aranjează componentele pe linii păstrând dimensiunile. Aliniază componentele care încap într-o linie și dacă e nevoie continuă pe linia următoare. Se folosește de obicei pentru aranjarea butoanelor într-un *Panel*.

Constructori:

`FlowLayout()` – aliniere centrată și distanță implicită de 5 între componente atât pe orizontală cât și pe verticală;

`FlowLayout(int align)` – primește ca parametru tipul de aliniere (LEFT, RIGHT, CENTER din `FlowLayout`), distanța dintre componente fiind implicit 5;

`FlowLayout(int align, int hgap, int vgap)` – primește ca parametri tipul de aliniere (LEFT, RIGHT, CENTER din `FlowLayout`) și distanțele orizontale și verticale dintre componente.



```
public class FlowFrame extends Frame {
    private FlowLayout fLayout = new FlowLayout();
    private Button firstButton = new Button("First Button");
    private Button secondButton = new Button("Second Button");
    private Button thirdButton = new Button("Third Button");
    private Button fourthButton = new Button("Fourth Button");
    private Button fifthButton = new Button("Fifth Button");
    private Button lastButton = new Button("Last Button");

    public FlowFrame() {
        this.setSize(400, 220);
        this.setTitle("FlowLayout");
        this.setLayout(fLayout);
        this.setBackground(new Color(240, 240, 240));
        this.add(firstButton);
        this.add(secondButton);
        this.add(thirdButton);
        this.add(fourthButton);
        this.add(fifthButton);
        this.add(lastButton);
        this.show();
    }

    public static void main(String[] args) {
        new FlowFrame();
    }
}
```


Limbajul Java – interfață grafică (9)

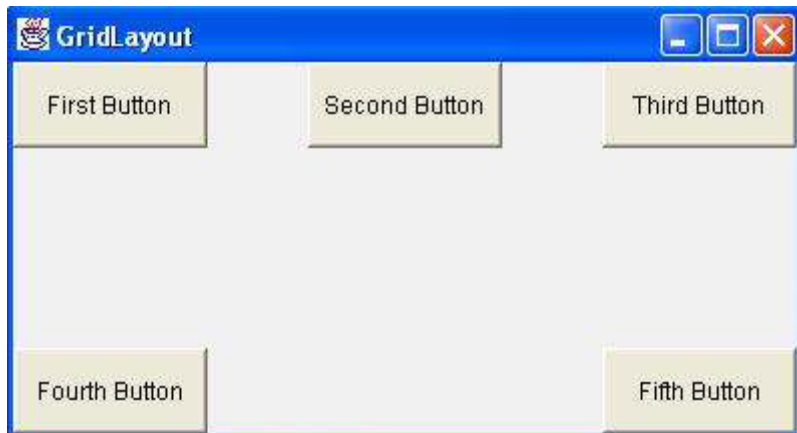
GridLayout împarte containerul în linii și coloane și păstrează componentele în celulele astfel obținute. Toate celulele au aceeași dimensiune. Fiecare componentă este expandată la dimensiunea celulei. În cazul unor interfețe complexe, se pot introduce în celule container *Panel* cu subcomponente aranjate tot prin *GridLayout*. Pentru obținerea unor spații libere se pot introduce componente *Panel* goale în celulele corespunzătoare.

Constructori:

`GridLayout()` – creează o singură linie cu câte o coloană pentru fiecare componentă adăugată, fără distanță între componente;

`GridLayout(int rows, int cols)` – primește ca parametri numărul de linii și coloane;

`GridLayout(int rows, int cols, int hgap, int vgap)` – primește ca parametri numărul de linii și coloane respectiv distanțele orizontale și verticale dintre componente.



```
public class GridFrame extends Frame {  
    private GridLayout gLayout = new GridLayout(2, 3, 50, 100);  
    private Button firstButton = new Button("First Button");  
    private Button secondButton = new Button("Second Button");  
    private Button thirdButton = new Button("Third Button");  
    private Button fourthButton = new Button("Fourth Button");  
    private Button fifthButton = new Button("Fifth Button");  
    private Panel emptyPanel = new Panel();  
}
```

```
public GridFrame() {  
    this.setSize(400, 220);  
    this.setTitle("GridLayout");  
    this.setLayout(gLayout);  
    this.setBackground(new Color(240, 240, 240));  
    this.add(firstButton);  
    this.add(secondButton);  
    this.add(thirdButton);  
    this.add(fourthButton);  
    this.add(emptyPanel);  
    this.add(fifthButton);  
    show();  
}
```

```
public static void main(String[] args) {  
    new GridFrame();  
}
```

Limbajul Java – interfață grafică (10)

GridBagLayout împarte containerul în celule care pot avea dimensiuni diferite. O componentă poate ocupa mai multe celule. Exemplu:



First Button [0,0]	Second Button [0,2]
Third Button [1,0]	
Fourth Button [2,0]	Fifth Button [2,1]
Last Button [4,1]	

Alinierea componentelor:

First Button:

gridy = 0, gridx = 0 (celula [0,0])
gridwidth = 2, gridheight = 1 (două coloane, o linie)
weightx = 0.6 (0.6 din extraspațiul orizontal)
weighty = 0.0 (0.0 din extraspațiul vertical)

Second Button:

gridy = 0, gridx = 2 (celula [0,2])
gridwidth = 2, gridheight = 1 (două coloane, o linie)
weightx = 0.3 (0.3 din extraspațiul orizontal)
weighty = 0.0 (0.0 din extraspațiul vertical)

Third Button:

gridy = 1, gridx = 0 (celula [1,0])
gridwidth = 4, gridheight = 1 (patru coloane, o linie)
weightx = 0.0 (0.0 din extraspațiul orizontal)
weighty = 0.3 (0.3 din extraspațiul vertical)

Fourth Button:

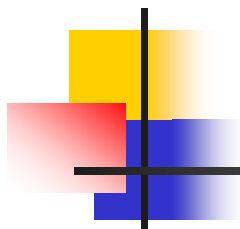
gridy = 2, gridx = 0 (celula [2,0])
gridwidth = 1, gridheight = 3 (o coloană, trei linii)
weightx = 0.0 (0.0 din extraspațiul orizontal)
weighty = 0.3 (0.3 din extraspațiul vertical)

Fifth Button:

gridy = 2, gridx = 1 (celula [2,1])
gridwidth = 3, gridheight = 2 (trei coloane, două linii)
weightx = 0.6 (0.6 din extraspațiul orizontal)
weighty = 0.6 (0.6 din extraspațiul vertical)

Last Button:

gridy = 4, gridx = 1 (celula [4,1])
gridwidth = 3, gridheight = 1 (trei coloane, o linie)
weightx = 0.6 (0.6 din extraspațiul orizontal)
weighty = 0.3 (0.3 din extraspațiul vertical)



Limbajul Java – interfață grafică (11)

GridBagLayout – codul sursă

```
public class GridBagFrame extends Frame {
    private GridBagLayout gbLayout = new GridBagLayout();
    private GridBagConstraints gbc = new GridBagConstraints();
    private Button firstButton = new Button("First Button");
    private Button secondButton = new Button("Second Button");
    private Button thirdButton = new Button("Third Button");
    private Button fourthButton = new Button("Fourth Button");
    private Button fifthButton = new Button("Fifth Button");
    private Button lastButton = new Button("Last Button");

    public GridBagFrame() {
        this.setSize(400, 220);
        this.setTitle("GridBagLayout");
        this.setLayout(gbLayout);
        this.setBackground(new Color(240, 240, 240));
        gbc.fill = GridBagConstraints.BOTH; //directii de extindere
        gbc.insets = new Insets(5, 5, 5, 5); //dist. fata de marginile celulei
        gbc.gridy = 0; //linia 0
        gbc.gridx = 0; //coloana 0
        gbc.gridwidth = 2; //ocupa doua coloane
        gbc.weightx = 0.6; //0.6 din extraspatiul orizontal
        gbLayout.setConstraints(firstButton, gbc);
        this.add(firstButton);
        gbc.gridy = 0; //linia 0
        gbc.gridx = 2; //coloana 2
        gbc.weightx = 0.3; //0.3 din extraspatiul orizontal
        gbLayout.setConstraints(secondButton, gbc);
        this.add(secondButton);
```

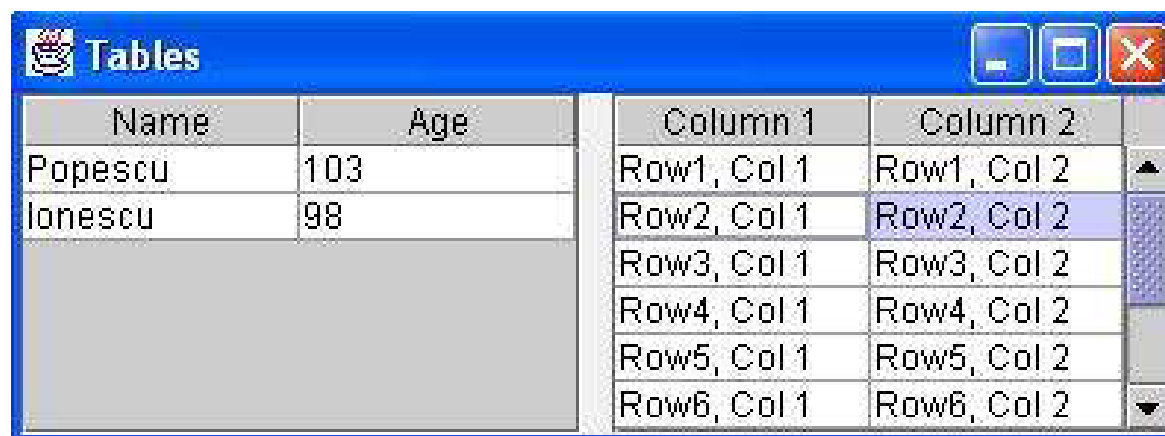
```
        gbc.gridy = 1; //linia 1
        gbc.gridx = 0; //coloana 0
        gbc.gridwidth = 4; //ocupa 4 coloane
        gbc.weightx = 0.0; //resetare
        gbc.weighty = 0.3; //0.3 din extraspatiul vertical
        gbLayout.setConstraints(thirdButton, gbc);
        this.add(thirdButton);
        gbc.gridy = 2; //linia 2
        gbc.gridx = 0; //coloana 0
        gbc.gridwidth = 1; //resetare
        gbc.gridheight = 3; //ocupa trei linii
        gbLayout.setConstraints(fourthButton, gbc);
        this.add(fourthButton);
        gbc.gridy = 2; //linia 2
        gbc.gridx = 1; //coloana 1
        gbc.gridwidth = 3; //ocupa trei coloane
        gbc.gridheight = 2; //ocupa doua linii
        gbc.weighty = 0.6; //0.6 din extraspatiul vertical
        gbc.weightx = 0.6; //0.6 din extraspatiul orizontal
        gbLayout.setConstraints(fifthButton, gbc);
        this.add(fifthButton);
        gbc.gridy = 4; //linia 4
        gbc.gridx = 1; //coloana 1
        gbc.gridheight = 1; //resetare
        gbc.weighty = 0.3; //0.3 din extraspatiul vertical
        gbLayout.setConstraints(lastButton, gbc);
        this.add(lastButton);
        show();
    }

    public static void main(String[] args) {
        new GridBagFrame();
    }
}
```

Limbajul Java – interfață grafică (12)

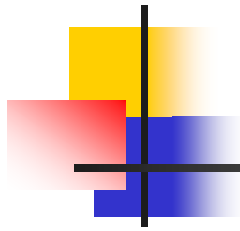
Crearea tabelelor

- O clasă care permite crearea tabelelor este *JTable*;
- Gestionarea componentelor *JTable* se poate face prin clasa *DefaultTableModel*;
- Pentru o funcționare corectă cu scroll, obiectul *JTable* trebuie adăugat pe un *JScrollPane* și nu pe *ScrollPane*!
- Aplicația următoare folosește două tabele, unul preîncărcat, iar celălalt cu încărcare dinamică a datelor:



Name	Age
Popescu	103
Ionescu	98

Column 1	Column 2
Row1, Col 1	Row1, Col 2
Row2, Col 1	Row2, Col 2
Row3, Col 1	Row3, Col 2
Row4, Col 1	Row4, Col 2
Row5, Col 1	Row5, Col 2
Row6, Col 1	Row6, Col 2



Limbajul Java – interfață grafică (13)

Codul sursă al aplicației cu tabele

```
public class TableFrame extends Frame {
    private GridLayout gLayout = new GridLayout(1, 2, 10, 10);
    private JScrollPane leftScrollPane = new JScrollPane();
    private JScrollPane rightScrollPane = new JScrollPane();
    private String leftTableColumns[] = {"Name", "Age"}; //header
    private Object leftTableData[][] = {{ "Popescu", "103"}, {"Ionescu", "98"}}; //continut
    private JTable leftTable = new JTable(leftTableData, leftTableColumns); //tabel cu date preincarcate
    private JTable rightTable = new JTable(); //tabel gol
    private String rightTableColumns[] = {"Column 1", "Column 2"}; //header
    private DefaultTableModel rightTableModel = new DefaultTableModel(rightTableColumns, 0);

    public TableFrame() {
        this.setSize(400, 150);
        this.setLayout(gLayout);
        this.setTitle("Tables");
        this.setBackground(new Color(240, 240, 240));
        this.add(leftScrollPane);
        this.add(rightScrollPane);
        leftScrollPane.getViewport().add(leftTable, null);
        rightScrollPane.getViewport().add(rightTable, null);
        rightTable.setModel(rightTableModel);
        for(int i=1; i<=10; i++){
            String row[] = {"Row" + i + ", Col 1", "Row" + i + ", Col 2"};
            rightTableModel.addRow(row);
        }
        show();
    }

    public static void main(String[] args) {
        new TableFrame();
    }
}
```



Limbajul Java – interfață grafică (14)

Aplicații

1. Introduceți în fereastra aplicației care gestionează lista de studenți un buton *Clear* care să permită ștergerea conținutului listei. Ștergerea conținutului componentei *List* (din pachetul AWT) se poate face prin metodele *clear* sau *removeAll*.
2. Modificați funcția de adăugare în listă astfel încât aceasta să păstreze lista sortată la introducerea unui nou student. Preluarea conținutului componentei *List* (din AWT) într-un tablou de tip *String* se poate efectua prin metoda *getItems*, iar preluarea liniei de pe o anumită poziție se poate face prin metoda *getItem* cu un parametru de tip *int*, reprezentând poziția în listă. Inserarea pe o poziție în listă se poate efectua prin metoda *add* cu un parametru *String* și unul de tip *int*, reprezentând poziția. De asemenea, *getItemCount* returnează numărul de linii din listă.
3. Înlocuiți lista din aplicația prezentată cu un tabel.
4. Dezvoltați aplicația astfel încât, pe lângă nume, să se introducă în tabel adresa, vârsta și media studentului.
5. Rearanjați componentele pe fereastră prin *GridLayout* sau *GridBagLayout* și setați fereastra redimensionabilă.



Limbajul Java – fire de execuție (1)

- Un fir de execuție (*thread*) este o secvență de instrucțiuni executată de un program.
- Firul de execuție primar este metoda *main* și atunci când acesta se termină, se încheie și programul.
- Un program poate utiliza fire multiple care sunt executate concurențial.
- Utilizarea firelor multiple este utilă în cazul programelor complexe care efectuează seturi de activități multiple.
- Fiecare fir de execuție are o prioritate care poate lua valori de la 1 (prioritate mică) la 10 (prioritate maximă). Firele cu prioritate mare sunt avantajate la comutare, ele primesc mai des controlul procesorului.
- Pentru implementarea unui fir de execuție în Java, se poate extinde clasa *Thread*. Deoarece Java nu acceptă moștenirea multiplă, în cazul în care a fost deja extinsă o clasă, pentru crearea unui fir de execuție trebuie implementată interfața *Runnable*. Indiferent de metoda utilizată, se suprascrive metoda *run* care trebuie să conțină instrucțiunile firului.
- Aplicația următoare pornește două fire de execuție: unul pentru afișarea numerelor și celălalt pentru afișarea literelor. Pentru a observa diferențele dintre cele două metode de implementare, firul de execuție *Numbers* extinde clasa *Thread*, în timp ce *Letters* implementează interfața *Runnable*.



Limbajul Java – fire de execuție (2)

Afișare concurențială de numere și litere:

```
public class Numbers extends Thread {           //extinde clasa Thread
    public void run(){
        for(int i=0; i<1000; i++){
            System.out.println(i);
        }
    }
}

public class Letters implements Runnable {       //implementeaza interfata Runnable
    char a = 'a';
    public void run(){
        for(int i=0; i<1000; i++){
            int c = a + i%26;
            System.out.println((char)c);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Numbers numbers = new Numbers();
        Thread letters = new Thread(new Letters());
        letters.start();
        numbers.start();
    }
}
```




Limbajul Java – fire de execuție (3)

- Principalele operații care pot fi efectuate după crearea unui fir de execuție:
 - **start** – pornește firul de execuție
 - **setPriority** – setează prioritatea firului de execuție (valoare între 1 și 10)
 - **getPriority** – returnează prioritatea firului de execuție (valoare între 1 și 10)
 - **sleep** – suspendă execuția unui fir pentru o perioadă precizată în milisecunde.
 - **yield** – suspendă temporar execuția firului curent în favoarea altor fire de execuție.
- Se recomandă oprirea firului prin încheierea execuției metodei **run**.
- Următoarea aplicație folosește un fir de execuție pentru deplasarea unei mingi pe orizontală.



Limbajul Java – fire de execuție (4)

```
public class Ball extends Thread {  
    private int px = 0;           //pozitia x  
    private int py = 0;           //pozitia y  
    private int size = 0;  
    private Color color = null;  
    private MyFrame parent = null;  
  
    public Ball(MyFrame parent, int px, int py, int size, Color color) {  
        this.parent = parent;     //referinta la fereastra  
        this.px = px;  
        this.py = py;  
        this.size = size;  
        this.color = color;  
    }  
  
    public int getPX(){  
        return px;  
    }  
  
    public int getPY(){  
        return py;  
    }  
  
    public int getSize(){  
        return size;  
    }  
  
    public Color getColor(){  
        return color;  
    }  
  
    public void run(){  
        while(px < parent.getSize().width){           //iesire din fereastra  
            px++;  
            parent.paint();  
        }  
    }  
}
```



Limbajul Java – fire de execuție (5)

```
public class MyFrame extends Frame {
    Ball ball = null;
    Image buffer = null;

    public MyFrame() {
        this.setSize(new Dimension(400, 300));
        this.setTitle("Balls");
        this.addWindowListener(new java.awt.event.WindowAdapter() {           //detectia evenimentului de inchidere a ferestrei
            public void windowClosing(WindowEvent e) {
                this._windowClosing(e);
            }
        });
        setVisible(true);
        buffer = createImage(getSize().width, getSize().height);
        ball = new Ball(this, 20, 50, 20, Color.red);
        ball.start();
    }

    void paint(){
        Graphics gbuffer = buffer.getGraphics();
        //se deseneaza mai intai in buffer (tehnica Double Buffering)
        gbuffer.setColor(Color.white);
        gbuffer.fillRect(0, 0, getSize().width, getSize().height);
        gbuffer.setColor(ball.getColor());
        gbuffer.fillOval(ball.getPX(), ball.getPY(), ball.getSize(), ball.getSize());
        paint(gbuffer);
        //se copiaza imaginea din buffer pe fereastra (tehnica Double Buffering)
        Graphics g = getGraphics();
        g.drawImage(buffer, 0, 0, getSize().width, getSize().height, 0, 0, getSize().width, getSize().height, this);
    }

    void this_windowClosing(WindowEvent e) {                                   //evenimentul de inchidere a ferestrei
        System.exit(0);
    }

    public static void main(String[] args){
        new MyFrame();
    }
}
```



Limbajul Java – fire de execuție (6)

Sincronizarea firelor de execuție

- Dacă în cadrul unui program Java există un fir de execuție care creează (produce) date și un al doilea fir de execuție care le prelucrează (consumă), de regulă se declară un bloc *synchronized*, ceea ce permite ca un singur fir să aibă acces la resurse (metode, date) la un moment dat.
- Atunci când un fir de execuție apelează *wait* în cadrul unui bloc de cod *synchronized*, alt fir poate accesa codul.
- Iar atunci când un fir de execuție încheie procesarea codului *synchronized*, el apelează metoda *notify* pentru a anunța alte fire de execuție să înceteze așteptarea.
- Reluăm în continuare aplicația care afișează numere și litere sincronizând cele două fire de execuție.



Limbajul Java – fire de execuție (7)

```
public class Numbers extends Thread {
    Main m;
    public Numbers(Main m){
        this.m = m;
    }
    public void run(){
        m.numbers();
    }
}

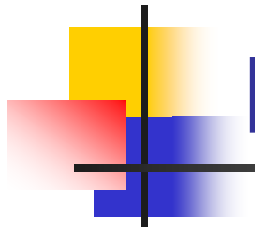
public class Letters implements Runnable {
    Main m;
    public Letters(Main m){
        this.m = m;
    }
    public void run(){
        m.letters();
    }
}

public class Main {
    public Main(){
        Numbers numbers = new Numbers(this);
        Thread letters = new Thread(new Letters(this));
        letters.start();
        numbers.start();
    }
}
```

```
public synchronized void numbers(){
    for(int i=0; i<1000; i++){
        if(i%26==0){
            try {
                this.notify();
                this.wait();
            }
            catch (InterruptedException ex) { }
        }
        System.out.println(i);
    }
    this.notify();
}

public synchronized void letters(){
    char a = 'a';
    for(int i=0; i<1000; i++){
        if(i%26==0){
            try {
                this.notify();
                this.wait();
            }
            catch (InterruptedException ex) { }
        }
        int c = a + i%26;
        System.out.println((char)c);
    }
    this.notify();
}

public static void main(String[] args) {
    new Main();
}
}
```



Limbajul Java – fire de execuție (8)

Aplicații propuse:

1. Introducerea unei întârzieri de 10 ms în algoritmul de mișcare a bilei din aplicația prezentată.
2. Modificarea aplicației astfel încât să permită pornirea mai multor mingi simultan.
3. Să se înlocuiască algoritmul de deplasare a mingii pe orizontală cu următorul algoritm de mișcare:

```
dx = 1;
dy = 1;
while (true){
    px += dx;
    py += dy;
    if((px <= 0) || (px >= frame_width))
        dx = -dx;
    if((py <= 0) || (py >= frame_height))
        dy = -dy;
    paint();
}
```

Un alt algoritm de mișcare:

```
gravity = 1;
speed = -30;
speedy = -30;
speedx = 0;
while(true){
    speedy += gravity;
    py += speedy;
    px += speedx;
    if(py > frameheight){
        speedy = speed;
        speed += 3;
    }
    if(speed == 0) break;
    paint();
}
```



Limbajul Java – colecții (1)

O colecție grupează date într-un singur obiect și permite manipularea acestora.

Clasele **Vector** și **ArrayList**

- Permit implementarea listelor redimensionabile cu elemente referință de orice tip, inclusiv *null*.
- O diferență importantă: *Vector* este sincronizat, iar *ArrayList* este nesincronizat, ceea ce-l face mai rapid. Dacă un *ArrayList* este accesat și modificat de mai multe fire de execuție concurențial, necesită sincronizare externă (a obiectului care încapsulează lista). O altă soluție constă în crearea unei instanțe sincronizate, prevenind astfel eventualele accese nesincronizate:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

- Clasa *Vector* s-a păstrat pentru compatibilitate cu versiuni JDK mai vechi de 1.2.
- Exemple

```
Vector vector = new Vector();           //vector gol
vector.addElement(new Integer(1));       //vector = {1}
vector.add(new Integer(3));              //vector = {1, 3}
vector.insertElementAt(new Integer(5), 1); //vector = {1, 5, 3}
vector.setElementAt(new Integer(2), 1);  //vector = {1, 2, 3}
```

```
ArrayList list = new ArrayList();       //lista goala
list.add(new Integer(1));                //list = {1}
list.add(new Integer(3));                //list = {1, 3}
list.add(1, new Integer(5));             //list = {1, 5, 3}
list.set(1, new Integer(2));             //list = {1, 2, 3}
```



Limbajul Java – colecții (2)

- Începând cu versiunea JDK 1.8, listele au metodă *sort*. În exemplul următor criteriul de sortare este furnizat printr-o expresie lambda. Afișarea poate fi și ea realizată cu ajutorul unei expresii lambda, apelând metoda *forEach* a listei, disponibilă tot de la JDK 1.8.

```
public class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add(new Person("Popescu", 103));
        list.add(new Person("Ionescu", 98));
        list.add(new Person("Petrescu", 49));
        list.sort((o1,o2)->(((Person)o1).name.compareTo(((Person)o2).name)));
        //{Ionescu-98, Petrescu-49, Popescu-103}
        list.forEach((o)->System.out.println(((Person)o).name+" "+((Person)o).age));
    }
}
```




Limbajul Java – colecții (3)

- Pentru evitarea conversiei repetate din *Object* în *Person*, la crearea colecției se poate preciza tipul componentelor. Astfel, exemplul anterior s-ar rescrie în felul următor:

```
public class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static void main(String[] args) {
        ArrayList<Person> list = new ArrayList();
        //ArrayList<Person> list = new ArrayList<Person>();
        list.add(new Person("Popescu", 103));
        list.add(new Person("Ionescu", 98));
        list.add(new Person("Petrescu", 49));
        list.sort((o1,o2)->o1.name.compareTo(o2.name));
        //{Ionescu-98, Petrescu-49, Popescu-103}
        list.forEach((o)->System.out.println(o.name+" "+o.age));
    }
}
```



Limbajul Java – colecții (4)

Clasa Stack

- Permite implementarea sincronizată a stivelor
- Extinde clasa Vector cu operațiile specifice stivei:
 - push – introduce un element în vârful stivei;
 - pop – scoate elementul din vârful stivei;
 - peek – preia (citește) elementul din vârful stivei, fără să-l scoată;
 - empty – verifică dacă stiva e goală;
 - search – returnează poziția unui obiect din stivă față de vârf.

- Exemple:

```
Stack stack = new Stack();           //stiva goala
System.out.println(stack.empty());   //true
stack.push(new Integer(1));          //stack = {1}
stack.push(new Integer(2));          //stack = {1, 2}
stack.push(new Integer(3));          //stack = {1, 2, 3}
System.out.println(stack.search(new Integer(3))); //1
System.out.println(stack.search(new Integer(2))); //2
System.out.println(stack.empty());   //false
System.out.println(stack.pop());     //3, stack = {1, 2}
System.out.println(stack.peek());    //2, stack = {1, 2}
System.out.println(stack.pop());     //2, stack = {1}
System.out.println(stack.pop());     //1, stiva goala
System.out.println(stack.empty());   //true
```



Limbajul Java – colecții (5)

Clasa LinkedList

- Permite implementarea stivelor și a cozilor.
- Diferența față de Stack: LinkedList este nesincronizat (deci mai rapid). Nefiind sincronizat, LinkedList necesită sincronizare externă în caz de accesare concurențială multifir.
- Exemple:

```
LinkedList stack = new LinkedList();  
stack.addLast(new Integer(1));  
stack.addLast(new Integer(2));  
stack.addLast(new Integer(3));  
System.out.println(stack.removeLast());  
System.out.println(stack.getLast());  
System.out.println(stack.removeLast());  
System.out.println(stack.removeLast());
```

```
//stiva goala  
//stack = {1}  
//stack = {1, 2}  
//stack = {1, 2, 3}  
//3, stack = {1, 2}  
//2, stack = {1, 2}  
//2, stack = {1}  
//1, stiva goala
```

```
LinkedList queue = new LinkedList();  
queue.addFirst(new Integer (1));  
queue.addFirst(new Integer(2));  
queue.addFirst(new Integer(3));  
System.out.println(queue.removeLast());  
System.out.println(queue.removeLast());  
System.out.println(queue.removeLast());
```

```
//coada goala  
//queue = {1}  
//queue = {2, 1}  
//queue = {3, 2, 1}  
//1, queue = {3, 2}  
//2, queue = {3}  
//3, coada goala
```



Limbajul Java – colecții (6)

Clasele **HashSet**, **TreeSet** și **LinkedHashSet**

- Interfața **Set** permite folosirea mulțimilor – colecții de date în care elementele sunt unice.
- Clasa **HashSet** păstrează elementele adăugate în ordinea stabilită de o funcție de dispersie, asigurând astfel operații de introducere și preluare a unui element rapide, de complexitate $O(1)$. Funcția de dispersie poate fi schimbată prin redefinirea metodei *hashCode()*.
- Clasa **TreeSet** păstrează elementele sortate în ordine crescătoare, ceea ce determină operații de introducere și preluare a unui element mai lente, de complexitate $O(\log n)$.
- Clasa **LinkedHashSet** (disponibilă începând cu versiunea JDK 1.4) păstrează elementele în ordinea în care au fost introduse asigurând operații de accesare a unui element de complexitate $O(1)$.

- Exemple

■ <code>HashSet hs = new HashSet();</code>	<code>//hs = {}</code>
■ <code>hs.add(new Integer(1));</code>	<code>//hs = {1}</code>
■ <code>hs.add(new Integer(2));</code>	<code>//hs = {2, 1}</code>
■ <code>hs.add(new Integer(4));</code>	<code>//hs = {2, 4, 1}</code>
■ <code>hs.add(new Integer(2));</code>	<code>//hs = {2, 4, 1}</code>
■ <code>TreeSet ts = new TreeSet(hs);</code>	<code>//ts = {1, 2, 4}</code>
■ <code>ts.add(new Integer(3));</code>	<code>//ts = {1, 2, 3, 4}</code>
■ <code>LinkedHashSet lhs = new LinkedHashSet();</code>	<code>//lhs = {}</code>
■ <code>lhs.add(new Integer(1));</code>	<code>//lhs = {1}</code>
■ <code>lhs.add(new Integer(2));</code>	<code>//lhs = {1, 2}</code>
■ <code>lhs.add(new Integer(4));</code>	<code>//lhs = {1, 2, 4}</code>
■ <code>lhs.add(new Integer(3));</code>	<code>//lhs = {1, 2, 4, 3}</code>



Limbajul Java – colecții (7)

Implementarea interfeței Comparable pentru obiectele încărcate în TreeSet

- Obiectele încărcate în TreeSet sunt sortate prin intermediul funcției *compareTo*. De aceea, clasa obiectelor încărcate trebuie să implementeze interfața Comparable.
- Proprietatea de set (mulțime) se aplică câmpului după care se face comparația în *compareTo*. Dacă valoarea aceluși câmp este aceeași în două obiecte, chiar dacă celelalte câmpuri diferă, în set rămâne doar unul din obiecte. De aceea, dacă în *compareTo* valorile câmpului comparat sunt egale se poate ține cont de celelalte câmpuri, aplicând astfel proprietatea de set pe întregul obiect.
- Exemplu:

```
public class Person implements Comparable{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int compareTo(Object p){
        return this.name.compareTo(((Person)p).name);
    }
}
```

```
public static void main(String[] args) {
    TreeSet ts = new TreeSet();
    ts.add(new Person("Popescu", 103));
    //{Popescu-103}
    ts.add(new Person("Ionescu", 98));
    //{Ionescu-98, Popescu-103}
    ts.add(new Person("Petrescu", 49));
    //{Ionescu-98, Petrescu-49, Popescu-103}
}
}
```



Limbajul Java – colecții (8)

- Criteriul de comparație poate fi furnizat și printr-o expresie lambda, caz în care clasa `Person` nu mai trebuie să implementeze interfața *Comparable*:

```
public class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static Comparator alphabeticNames = (p1,p2) -> {
        return ((Person)p1).name.compareTo(((Person)p2).name);
    };
    public static void main(String[] args) {
        TreeSet ts = new TreeSet(Person.alphabeticNames);
        ts.add(new Person("Popescu", 103));
        ts.add(new Person("Ionescu", 98));
        ts.add(new Person("Petrescu", 49));
    }
}
```



Limbajul Java – colecții (9)

- Conținutul colecției *ts* de tip *TreeSet* din aplicația prezentată mai sus, se poate afișa printr-o buclă *for* îmbunătățită, disponibilă începând cu JDK 1.5:

```
TreeSet<Person> ts = new TreeSet(Person.alphabeticNames);  
ts.add(new Person("Popescu", 103));  
ts.add(new Person("Ionescu", 98));  
ts.add(new Person("Petrescu", 49));  
//{Ionescu-98, Petrescu-49, Popescu-103}  
for(Person o: ts)  
    System.out.println(o.name + " " + o.age);
```

- O altă variantă de afișare a conținutului colecției constă în apelul metodei *forEach* (disponibilă începând cu JDK 1.8), cu o expresie lambda:

```
TreeSet<Person> ts = new TreeSet(Person.alphabeticNames);  
ts.add(new Person("Popescu", 103));  
ts.add(new Person("Ionescu", 98));  
ts.add(new Person("Petrescu", 49));  
//{Ionescu-98, Petrescu-49, Popescu-103}  
ts.forEach((o)->System.out.println(o.name + " " + o.age));
```



Limbajul Java – colecții (10)

Iteratori

- Iteratorii permit traversarea în ordine a colecțiilor de date.
- Exemplul 1 (aplicația anterioară):

```
TreeSet<Person> ts = new TreeSet(Person.alphabeticNames);
ts.add(new Person("Popescu", 103));
ts.add(new Person("Ionescu", 98));
ts.add(new Person("Petrescu", 49));
//{Ionescu-98, Petrescu-49, Popescu-103}
Iterator<Person> i = ts.iterator();
while(i.hasNext()){
    Person o = i.next();
    System.out.println(o.name + " " + o.age);
}
```

- Exemplul 2 (colecție de tip String):

```
TreeSet ts = new TreeSet();
ts.add("Romania");
ts.add("Anglia");
ts.add("Germania");
ts.add("Franta");
Iterator i = ts.iterator();
while(i.hasNext()) System.out.println(i.next()); //{Anglia, Franta, Germania, Romania}
```




Limbajul Java – colecții (11)

Clasele **HashMap**, **TreeMap** și **LinkedHashMap**

- Interfața **Map** permite păstrarea de perechi cheie-valoare, ambele de tip referință, cu chei unice. Principalele operații:
 - **put(k, v)** – asociază cheii **k** valoarea **v**. Dacă există deja cheia **k** atunci vechea valoare se înlocuiește cu **v**.
 - **get(k)** – returnează valoarea corespunzătoare cheii **k**.
- Clasa **HashMap** păstrează cheile în ordinea stabilită de o funcție de dispersie asigurând astfel accesări rapide, de complexitate $O(1)$.
- Clasa **TreeMap** păstrează cheile ordonate crescător, accesarea fiind astfel mai lentă, de complexitate $O(\log n)$.
- Clasa **LinkedHashMap** (disponibilă începând cu JDK 1.4) păstrează cheile în ordinea în care au fost introduse asigurând accesări de complexitate $O(1)$.
- Exemple
 - ```
HashMap hm = new HashMap();
hm.put("Romania", "Bucuresti");
hm.put("Franta", "Paris");
hm.put("Germania", "Bonn");
hm.put("Germania", "Berlin");
System.out.println(hm); //{Romania=Bucuresti, Germania=Berlin, Franta=Paris}
System.out.println(hm.get("Germania")); //Berlin
```
  - ```
TreeMap tm = new TreeMap(hm);  
System.out.println(tm);           //{Franta=Paris, Germania=Berlin, Romania=Bucuresti}
```
 - ```
LinkedHashMap lhm = new LinkedHashMap();
lhm.put("Romania", "Bucuresti");
lhm.put("Franta", "Paris");
lhm.put("Germania", "Berlin");
System.out.println(lhm); //{Romania=Bucuresti, Franta=Paris, Germania=Berlin}
```



# Limbajul Java – colecții (12)

## Implementarea interfeței Comparable pentru cheile încărcate în TreeMap

- Perechile cheie-valoare încărcate în TreeMap sunt sortate în ordinea crescătoare a cheilor, prin intermediul funcției *compareTo*. De aceea, clasa obiectelor cheie trebuie să implementeze interfața Comparable. Exemplu:

```
public class Person implements Comparable{
 private String name;
 private int age;

 public Person(String name, int age) {
 this.name = name;
 this.age = age;
 }

 public int compareTo(Object p){
 return this.name.compareTo(((Person)p).name);
 }

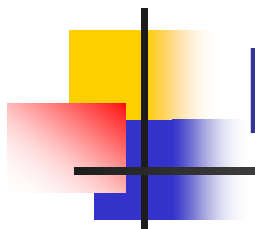
 public static void main(String[] args) {
 TreeMap tm = new TreeMap();
 tm.put(new Person("Pop", 54), "Romania"); //tm={Pop-54-Romania}
 tm.put(new Person("Brown", 98), "Anglia"); //tm={Brown-98-Anglia, Pop-54-Romania}
 tm.put(new Person("Schmitt", 49), "Germania"); //tm={Brown-98-Anglia, Pop-54-Romania, Schmitt-49-Germania}
 }
}
```



# Limbajul Java – colecții (13)

- Pentru evitarea conversiei repetate a cheilor din *Object* în *Person* respectiv a valorilor din *Object* în *String*, la crearea colecției se poate preciza tipul cheilor și a valorilor. Astfel, exemplul anterior, cu afișarea inclusă, s-ar rescrie în felul următor:

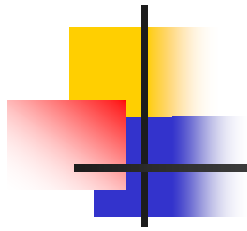
```
TreeMap<Person, String> tm = new TreeMap();
//TreeMap<Person, String> tm = new TreeMap<Person, String>();
Person p = new Person("Pop", 54);
tm.put(p, "Romania");
//tm={Pop-54-Romania}
Person q = new Person("Brown", 98);
tm.put(q, "Anglia");
//tm={Brown-98-Anglia, Pop-54-Romania}
Person r = new Person("Schmitt", 49);
tm.put(r, "Germania");
//tm={Brown-98-Anglia, Pop-54-Romania, Schmitt-49-Germania}
for(Map.Entry<Person,String> entry : tm.entrySet()) {
 Person k = entry.getKey();
 String v = entry.getValue();
 System.out.println(k.getName()+ "-" +k.getAge()+ "-" +v);
}
```



# Limbajul Java – colecții (14)

## Aplicații propuse

1. Să se modifice programele prezentate la **TreeSet** și **TreeMap** astfel încât sortarea persoanelor să se facă după vârstă în cazul în care au același nume.
2. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceți de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** și afișați cursul cu cel mai mic număr de credite.
3. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceți de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** și afișați cursul cu cel mai mare număr de credite.
4. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceți de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** și afișați-le în ordinea descrescătoare a numărului de credite.
5. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceți de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** și afișați-le în ordinea crescătoare a anului de studiu.
6. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceți de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** și afișați-le în ordinea alfabetică a titlului.
7. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceți de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** și afișați-le în ordinea alfabetică a profesorului.



# Limbajul Java – test

---

Test (T) susținut la laborator din aplicațiile propuse la laborator în cadrul capitolului *Limbajul Java* – se va implementa și rula pe calculator o aplicație care implică o moștenire respectiv folosirea unor colecții de date.