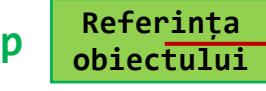


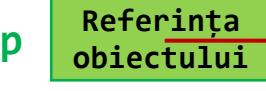
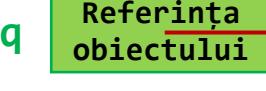
REFERINȚE

O **referință** reprezintă o modalitate de accesare a unei zone de memorie alocate dinamic (i.e., în zona de heap) prin intermediul adresei sale. Deși o referință utilizează adresa de memorie a unui obiect pentru a-l accesa, acest lucru nu se realizează într-un mod transparent pentru programator, respectiv acesta nu poate determina adresa respectivă și nici nu poate executa operații specifice pointerilor din limbajele C/C++!

Referințele, la fel ca variabilele de tip primitiv, sunt memorate în zona de stivă (stack). Practic, diferența dintre o variabilă de tip primitiv și una de tip referință constă în faptul că o variabilă de tip primitiv conține direct o valoare de un anumit tip, în timp ce o variabilă de tip referință conține doar "adresa" din zona de heap la care se află alocat un anumit obiect:

	STACK	HEAP
<code>int x = 10;</code>	<code>x</code> 	
<code>Persoana p = new Persoana();</code>	<code>p</code> 	

Efectuarea unei operații de atribuire între două referințe NU va conduce la realizarea unei copii a obiectului referit, ci obiectul respectiv va putea fi accesat prin intermediul ambelor referințe, ceea ce poate conduce la efecte colaterale nedorite:

	STACK	HEAP
<code>Persoana p = new Persoana();</code>	<code>p</code> 	
<code>Persoana q = p;</code>	<code>q</code> 	

Literalul null este utilizat pentru a indica faptul că o referință nu este asociată cu niciun obiect.

TABLOURI

În limbajul Java, tablourile sunt considerate variabile de tip referință, deci ele trebuie inițializate sau alocate dinamic înainte de a fi utilizate.

Declararea tablourilor unidimensionale (de fapt, a unor referințe spre tablouri unidimensionale!) se poate realiza în două moduri:

a) **`tip_de_date[] tablou_1, tablou_2, ..., tablou_n;`**

De exemplu, prin `int[] a, b;` se vor declara două referințe **a** și **b** spre tablouri unidimensionale cu elemente de tip `int` care trebuie ulterior să fie alocate dinamic!

b) **`tip_de_date tablou_1[], tablou_2[], ..., tablou_n[];`**

De exemplu, prin `int a[], b;` doar variabila **a** este o referință spre un tablou unidimensional cu elemente de tip `int`, iar variabila **b** este una de tipul primitiv `int`!

Tablourile unidimensionale pot fi inițializate în momentul declarării lor folosind un sir de valori, astfel:

a) **`tip_de_date[] tablou = {valoare_1, ..., valoare_n};`**

Exemplu: `int[] a = {1, 2, 3, 4, 5}, b = {10, 20, 30};`

b) **`tip_de_date tablou[] = {valoare_1, ..., valoare_n};`**

Exemplu: `int a[] = {1, 2, 3, 4, 5}, b[] = {10, 20, 30};`

Alocarea dinamică a tablourilor unidimensionale se realizează folosind operatorul `new`, astfel:

```
tablou = new tip_de_date[număr_elemente];
```

Exemplu:

```
int a[];
.....
a = new int[5];
```

```
int[] a = null;
.....
a = new int[5];
```

```
int a[] = new int[5];
int []b = new int[7];
```

Toate elementele unui tablou alocat dinamic vor fi inițializate cu valori nule de tip!

După inițializarea sau alocarea dinamică a unui tablou, data membră **length** va conține dimensiunea fizică a tabloului, adică numărul maxim de elemente pe care le poate conține tabloul.

Parcurserea unui tablou se poate realiza fie pozitional (prin intermediul indicilor elementelor), fie folosind o instrucțiune repetitivă de tipul **enhanced-for**:

```
int[] a = {10, 20, 30, 40, 50};
System.out.println(a.length);           //se va afișa valoarea 5
for(int i = 0; i < a.length; i++)      //se vor afișa valorile 10 20 30
40 50
    System.out.print(a[i] + " ");

System.out.println();

int b[] = new int[7];
System.out.println(b.length);           //se va afișa valoarea 7
for(int elem : b)                    //se vor afișa valorile 0 0 0 0 0 0 0
    System.out.print(elem + " ");
```

Accesarea unui element dintr-un tablou al cărui indice este invalid (i.e., nu este cuprins între 0 și `tablou.length-1`) va duce la lansarea excepției `ArrayIndexOutOfBoundsException` în momentul rulării programului respectiv:

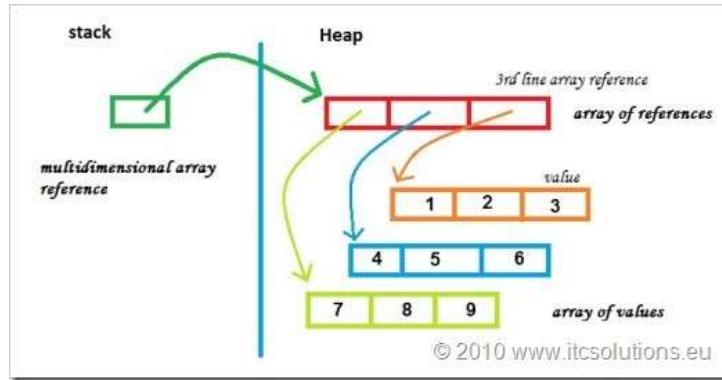
```
public class Test
{
    public static void main(String[] args)
    {
        int[] a = {10, 20, 30, 40, 50};
        System.out.println(a[0]);
        System.out.println(a[4]);
        System.out.println(a[8]);
    }
}

"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.3.1\lib\idea_rt.jar" -Dfile.encoding=UTF-8
10
50
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 8 out of bounds for length 5
    at Test.main(Test.java:6)

Process finished with exit code 1
```

Observați faptul că excepția `ArrayIndexOutOfBoundsException` a fost lansată abia în momentul în care s-a încercat accesarea elementului `a[8]`, valorile elementelor `a[0]` și `a[4]` fiind afișate corect!

În limbajul Java tablourile bidimensionale sunt, de fapt, tablouri unidimensionale ale căror elemente sunt referințe spre tablouri unidimensionale, fiecare reprezentând câte o linie (sursa imaginii: [Tutorial Java 6 - #4.3 Matrixes and Multidimensional Arrays | IT&C Solutions](#)):



Declararea tablourilor bidimensionale (de fapt, a unor referințe spre tablouri bidimensionale!) se poate realiza în mai multe moduri:

- a) `tip_de_date[][] tablou_1, tablou_2, ..., tablou_n;`
- b) `tip_de_date tablou_1[][], tablou_2[][], ..., tablou_n[][];`
- c) `tip_de_date[] tablou_1[], tablou_2[], ..., tablou_n[];`

Tablourile bidimensionale pot fi inițializate în momentul declarării, linie cu linie, aşa cum se poate observa în exemplele de mai jos:

```
int[][] a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int a[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int[] a[] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Liniile unui tablou bidimensional pot să aibă lungimi diferite:

```
int[][] a = {{1, 2}, {3}, {4, 5, 6, 7}, {8, 9}};
int a[][] = {{1, 2, 3, 4, 5, 6}, {7, 8, 9}};
```

Alocare dinamică a unui tablou bidimensional având liniile de aceeași lungime se realizează astfel:

```
referință_tablou = new tip_de_date[număr_linii][număr_coloane];
```

Exemplu:

```
int[][] a = new int[5][3];
int a[][] = new int[7][7];
```

Alocare dinamică a unui tablou bidimensional având liniile de lungimi diferite se realizează astfel:

- se alocă un tablou bidimensional precizând doar numărul de linii:
`tablou = new tip_de_date[număr_linii][];`
- se alocă, pe rând, fiecare linie din tabloul bidimensional, precizând numărul de coloane:
`tablou[0] = new tip_de_date[număr_coloane];`
`tablou[1] = new tip_de_date[număr_coloane];`
`.....`
`tablou[tablou.length-1] = new tip_de_date[număr_coloane];`

Exemplu:

```
int[][] a = new int[3][];
a[0] = new int[7];
a[1] = new int[2];
a[2] = new int[5];
```

Parcurgerea unui tablou se poate realiza fie pozitional (prin intermediul indicilor elementelor), fie folosind o instrucțiune repetitivă de tipul **enhanced-for**.

Exemplu:

```
int[][] a = {{1, 2}, {3}, {4, 5, 6, 7}, {8, 9}};
for(int i = 0; i < a.length; i++) // se va afișa (de două ori):
{ // 1 2
    for(int j = 0; j < a[i].length; j++) // 3
        System.out.print(a[i][j] + " "); // 4 5 6 7
    System.out.println(); // 8 9
}
```

```

for(int[] linie : a)
{
    for(int elem : linie)
        System.out.print(elem + " ");
    System.out.println();
}

```

CLASA Arrays

Clasa Arrays este o clasă utilitară (i.e., conține doar metode statice) dedicată manipulării tablourilor. În continuare, vom prezenta mai multe metode ale acestei clase, din versiunea sa existentă începând cu Java 11:

- **static String toString(Tip[] tablou)** – furnizează o reprezentare a tabloului transmis ca parametru sub forma unui sir de caractere sau sirul "null" dacă referința sa este null. Elementele tabloului vor fi enumerate între o pereche de paranteze drepte și despărțite între ele prin câte o virgulă și un spațiu.

Exemplu:

```

int[] t = {1, 2, 3, 4, 5};
int[][] a = {{1, 2}, {3}, {4, 5, 6}};

System.out.print(t);           // [I@4dd8dc3
System.out.print(Arrays.toString(t)); // [1, 2, 3, 4, 5]
System.out.print(Arrays.toString(a)); // [[I@6d03e736, [I@568db2f2, [I@378bf509]

```

Observați faptul că pentru tablourile bidimensionale se vor furniza referințele tablourilor corespunzătoare liniilor sale!

- **static String deepToString(Tip[] tablou)** – furnizează o reprezentare în adâncime a tabloului transmis ca parametru, respectiv dacă elementele tabloului sunt alte tablouri, atunci și acestea vor fi convertite în formatul precizat mai sus.

Exemplu:

```

int[] t = {1, 2, 3, 4, 5};
int[][] a = {{1, 2}, {3}, {4, 5, 6}};

System.out.print(a);           // [[I@4dd8dc3
System.out.print(Arrays.toString(a)); // [[I@6d03e736, [I@568db2f2, [I@378bf509]
System.out.print(Arrays.deepToString(a)); // [[1, 2], [3], [4, 5, 6]]

```

- **static void fill(Tip[] tablou, Tip valoare)** – atribuie tuturor elementelor tabloului unidimensional valoarea indicată.

Exemplu:

```
int[] t = {1, 2, 3, 4, 5};
Arrays.fill(t, 7);
System.out.println(Arrays.toString(t));           // [7, 7, 7, 7, 7]
```

- **static boolean equals(Tip[] a, Tip[] b)** – verifică dacă tablourile a și b sunt egale, respectiv dacă au același număr de elemente și elementele aflate pe aceleași poziții sunt egale.

Exemplu:

```
int[] t = {1, 2, 3, 4, 5};
int[] v = {1, 2, 3, 4, 5};
int[] w = {1, 2, 4, 4, 5};
System.out.println(Arrays.equals(t, v));          // true
System.out.println(Arrays.equals(w, v));          // false
```

```
int[][] a = {{1, 2}, {3}, {4, 5, 6}};
int[][] b = {{1, 2}, {3}, {4, 5, 6}};
System.out.println(Arrays.equals(a, b));          // false
```

Observați faptul că în cazul tablourilor bidimensionale a și b se va afișa false, deoarece se vor compara referințele liniilor lor!

- **static boolean deepEquals(Tip[] a, Tip[] b)** – verifică în profunzime dacă tablourile a și b sunt egale, respectiv dacă elementele lor sunt tot tablouri le verifică egalitatea din punct de vedere al lungimilor și al elementelor aflate pe același poziții, ci nu al referințelor.

Exemplu:

```
int[][] a = {{1, 2}, {3}, {4, 5, 6}};
int[][] b = {{1, 2}, {3}, {4, 5, 6}};
System.out.println(Arrays.deepEquals(a, b));    // true
```

- **static int mismatch(Tip[] a, Tip[] b)** – returnează cel mai mic indice k pentru care $a[k] \neq b[k]$ sau -1 dacă tablourile sunt egale. Dacă tablourile a și b nu sunt egale, atunci valoarea indicelui k este cuprinsă între 0 și minimul dintre lungimile celor două tablouri a și b.

Exemplu:

```

int[] t = {1, 2, 3};
int[] u = {1, 2, 3};
int[] v = {1, 2, 3, 4, 5};
int[] w = {1, 2, 4, 5, 5};

System.out.println(Arrays.mismatch(t, u)); // -1
System.out.println(Arrays.mismatch(u, v)); // 3
System.out.println(Arrays.mismatch(w, v)); // 2

```

Observați faptul că `Arrays.mismatch(u, v)` returnează 3, deoarece $u[0]==v[0]$, $u[1]==v[1]$ și $u[2]==v[2]$, dar în tabloul v există, în plus față de tabloul u, elementele 4 și 5, deci indicele 3 se referă strict la tabloul v!

- **static int compare(Tip[] a, Tip[] b)** – compară lexicografic cele două tablouri și returnează următoarele valori:
 - o valoare negativă dacă tabloul a este mai mic în sens lexicografic decât tabloul b;
 - o valoare pozitivă dacă tabloul a este mai mare în sens lexicografic decât tabloul b;
 - valoarea 0 dacă tabloul a este egal în sens lexicografic cu tabloul b.

Exemplu:

```

int[] t = {1, 2, 3, 4, 5};
int[] u = {1, 2, 3};
int[] v = {1, 2, 3, 4, 5};
int[] w = {1, 2, 4, 4, 5};

```

```

System.out.println(Arrays.compare(t, v)); // 0
System.out.println(Arrays.compare(t, u)); // > 0
System.out.println(Arrays.compare(w, v)); // > 0, deoarece 4 = w[3] > v[3] = 3
System.out.println(Arrays.compare(v, w)); // < 0, deoarece 3 = v[3] < w[3] = 4

```

- **static Tip[] copyOf(Tip[] tablou, int nr_elem)** – returnează un tablou format din primele nr_elem elemente ale tabloului dat ca parametru. Dacă nr_elem este strict mai mare decât lungimea tabloului, atunci se vor adăuga elemente nule de tip.

Exemplu:

```

int[] t = {1, 2, 3, 4, 5};
int[] u = Arrays.copyOf(t, t.length);
int[] v = Arrays.copyOf(t, 3);
int[] w = Arrays.copyOf(t, 8);

```

```

System.out.println(Arrays.toString(u)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(v)); // [1, 2, 3]
System.out.println(Arrays.toString(w)); // [1, 2, 3, 4, 5, 0, 0, 0]

```

- **static void sort(Tip[] tablou)** – sortează crescător elementele tabloului dat ca parametru.

Exemplu:

```
int[] t = {2, 1, 5, 2, 5, 3, -10, 7, 4, 5};
```

```
Arrays.sort(t);
```

```
System.out.println(Arrays.toString(t)); // [-10, 1, 2, 2, 3, 4, 5, 5, 5, 7]
```

- **static int binarySearch(Tip[] tablou, Tip valoare)** – caută valoarea indicată în tabloul dat folosind algoritmul de căutare binară. Tabloul trebuie să fie sortat crescător, deoarece, în caz contrar, rezultatul furnizat nu va fi corect! Metoda returnează indicele pe care se găsește valoarea respectivă în tablou (dacă valoarea se găsește de mai multe ori în tablou, atunci se returnă unul dintre indicii corespunzători) sau, dacă valoarea nu există în tablou, o valoare negativă p cu proprietatea că $-p-1$ reprezintă indicele unde ar putea fi inserată valoarea căutată în tablou astfel încât acesta să rămână sortat crescător (i.e., indicele unde ar fi trebuit să se găsească valoarea căutată).

Exemplu:

```
int[] t = {1, 2, 7, 7, 10, 10, 10, 21};
```

```
System.out.println(Arrays.binarySearch(t, 10)); // 5
```

```
System.out.println(Arrays.binarySearch(t, 9)); // -5, deoarece valoarea 9 ar  
// trebui să se găsească în tablou pe poziția  $-(-5)-1 = 4$ 
```

```
System.out.println(Arrays.binarySearch(t, -9)); // -1, deoarece valoarea -19 ar  
// trebui să se găsească în tablou pe poziția  $-(-1)-1 = 0$ 
```

```
System.out.println(Arrays.binarySearch(t, 99)); // -9, deoarece valoarea 99 ar  
// trebui să se găsească în tablou pe poziția  $-(-9)-1 = 8$ 
```

În afara metodelor prezentate mai sus, în clasa Arrays mai sunt definite și alte metode, majoritatea fiind variante ale celor prezentate mai sus (de exemplu, metode care nu prelucrează un tablou întreg, ci doar o secvență a sa, precizată prin 2 indici):
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html>.

CLASE ȘI OBIECTE

În continuare, vom prezenta maniera de implementare a principiilor *programării orientate pe obiecte* (POO) în Java. Paradigma POO în sine este evident aceeași cu cea pe care deja ați studiat-o în C++. Diferențele apar din punct de vedere al modului de implementare.

➤ Principiile de bază ale POO:

1. Abstractizarea (abstraction)

- Pe lângă tipurile de date predefinite în limbajul Java (primitive sau clase), cunoscute de mașina virtuală, în diferite aplicații este nevoie de noi tipuri de date, care să modeleze un concept sau un fenomen din lumea reală. Modelarea se realizează prin abstractizare, adică se identifică acele caracteristici/attribute relevante în contextul aplicației respective. Pe lângă caracteristicile identificate, se stabilește și un set de operații (comportament) care să acționeze asupra datelor respective.

2. Încapsularea (encapsulation)

- Încapsularea reprezintă mecanismul prin care datele și operațiile specifice sunt înglobate sub forma unui tot unitar – un obiect.
- Datele obiectului sunt ascunse (principiu ascunderii) pentru a nu putea fi accesate incorrect în anumite prelucrări. Modificarea și accesarea lor se poate realiza prin intermediul unor metode publice de tip set/get.
- De asemenea, prin încapsulare se separă detaliile de implementare față de implementarea propriu-zisă. Un utilizator trebuie doar să acceseze anumite operații, ci nu să cunoască detalii complexe de implementare.

3. Moștenirea (inheritance)

- Proprietatea prin care o clasă preia date și metode dintr-o clasă definită anterior, în scopul reutilizării codului

4. Polimorfismul (polymorphism)

- Proprietatea unui obiect de a avea comportament diferit, în funcție de context.
 - **Supraîncărcare (overloading/polimorfism static)** = pot exista două sau mai multe metode cu același nume, dar având liste de parametri diferite.
 - **Suprascriere (overriding)** = o subclă rescrie o metodă a superclasei.

- **Clasa** este o implementare a unui tip de date abstract și poate fi privită ca un şablon pentru o categorie de obiecte.

- **Sintaxa unei clase:**

```
[modificatori] class denumireClasă {
    date membre/atribute
    metode membre //nu mai pot fi implementate în afara clasei!
}
```

➤ **Modificatorii de clasă:**

- **public:** clasa poate fi accesată/instanțiată și din afara pachetului său
- **abstract:** clasa conține cel puțin o metodă fără implementare (metodă abstractă) și nu poate fi instanțiată
- **final:** clasa nu mai poate fi extinsă

Observație: Dacă nu există modificadorul public, clasa are un acces implicit, adică poate fi accesată/instanțiată doar din interiorul pachetului în care a fost creată.

În general, o clasă conține date membre, constructori și metode.

➤ **Date membre**

- Datele membre pot fi de orice tip (primitiv sau referință).
- Se declară ca orice variabilă locală, însă declararea poate fi însotită și de modificatori.
- Datele membre sunt inițializate cu valori nule de tip (spre deosebire de variabilele locale)!

▪ **Modificatori pentru date membre:**

1) **Modificatorii de acces:**

- **public:** data membră poate fi accesată și din afara clasei, însă în conformitate cu principiul ascunderii (încapsulare) acestea sunt, de obicei, private
- **protected:** data membră poate fi accesată din clasele din același pachet sau de subclasele din ierarhia sa
- **private:** data membră poate fi accesată doar din clasa din care face parte
- dacă nu este precizat niciun modificador de acces, atunci data membră respectivă are acces implicit, adică poate fi accesată doar din sursele aflate în același pachet

2) Alți modificatori:

- **static**: data membră este un câmp de clasă, adică este alocat o singură dată în memorie și partajat de toate instanțele clasei respective
- **final**: data membră poate fi doar inițializată, fără a mai putea fi modificată ulterior. Dacă data membră este un obiect, atunci nu îl se poate modifica referința, dar conținutul său poate fi modificat!

▪ Pentru o dată membră se pot combina mai mulți modificatori!

▪ În concluzie, datele membre se împart două categorii:

- **Date membre de instanță = date membre non-static**: se multiplică pentru fiecare obiect, alocându-se spațiu de memorie pentru fiecare în parte și sunt inițializate prin constructori.
- **Date membre de clasă = date membre static**: sunt partajate de către toate obiectele, se alocă o singură dată și pot fi modificate de orice instanță (obiect) al clasei respective. Exemplu: definirea unei date membre care nu depinde de un anumit obiect (TVA) sau pentru a contoriza numărul de obiecte instanțiate. Datele membre statice nu trebuie inițializate prin constructori!!!

Exemplu:

```
class Persoana{
    private int IDPersoana;
    private String nume;
    private int varsta;
    private static String nationalitate = "română";
    private static int nrPersoane = 0;
    .....
}
```

➤ Metode

➤ Sintaxa unei metode:

```
[modificatori] tipReturnat numeMetoda ([parametri]) {
    //corful metodei
}
```

- Modificatorii sunt aceiași ca la date membre, dar se adaugă și modificadorul **abstract** prin care se declară o metodă fără implementare, care urmează să fie definită obligatoriu în subclasele clasei respective.
- Utilizarea modificadorului **final** pentru o metodă împiedică redefinirea sa în subclasele clasei respective. De exemplu, o metodă care calculează TVA conține o formulă de calcul unică, care nu trebuie modificată/particularizată de către subclasele sale.
- **Parametrii unei metode nu pot să aibă valori implicite.**
- **Parametrii unei metode sunt transmiși întotdeauna doar prin valoare!**

Exemplu:

Considerăm următoarea clasă:

```
public class Test {
    static void modicare(int v[]) {
        v[0] = 100;
        v = new int[10];
        v[1] = 1000;
    }
    public static void main(String[] args) {
        int v[] = {1, 2, 3, 4, 5};
        modicare(v);
        System.out.println(Arrays.toString(v));
    }
}
```

După rulare, se va afișa următorul tablou: [100, 2, 3, 4, 5].

- Metodele statice nu pot accesa date membre sau metode non-statice, dar invers este posibil.
- Într-o clasă pot exista mai multe metode cu același nume prin intermediul mecanismului de supraîncărcare (**overloading**).
- Două metode cu același nume se consideră ca fiind supraîncărcate dacă diferă prin numărul sau tipul parametrilor lor.
- Dacă două metode au același nume și aceeași listă a parametrilor, dar diferă prin tipul returnat, atunci ele nu se vor considera supraîncărcate și compilatorul va semnaliza o eroare.

➤ Referința this

- Referința **this** reprezintă referința obiectului curent, respectiv a obiectului pentru care se accesează o dată membru sau o metodă.
- Modalități de utilizare:
 - pentru a accesa o dată membră sau pentru a apela o metodă:

```
this.nume="Popa Ion"
this.afisarePersoana();
```

- pentru a diferenția într-o metodă o dată membru de un parametru cu aceleași nume:

```
public void setNume(String nume) {
    this.nume = nume;
}
```

➤ Constructori

- Constructorii au rolul de a inițializa datele membre.
- Un constructor are numele identic cu cel al clasei și nu returnează nici o valoare.
- Un constructor nu poate fi static, final sau abstract.
- O clasă poate să conțină mai mulți constructori, prin mecanismul de supraîncărcare.
- Dacă într-o clasă nu este definit niciun constructor, atunci compilatorul va genera unul implicit (default), care va inițializa toate datele membre cu valorile nule de tip, mai puțin pe cele inițializate explicit!
- **Tipuri de constructori:**
 - **cu parametri:** inițializează datele membre cu valorile parametrilor

```
public Persoana(String nume, int varsta) {
    this.nume = nume;
    this.varsta = varsta;
}
```

- **fără parametri:** inițializează datele membre cu valori constante

```
public Persoana() {
    this.nume = "Popa Ion";
    this.varsta = 20;
}
```

- Pentru a apela constructorul cu argumente se poate utiliza referința **this**:

```
public Persoana() {
    this("Popa Ion", 20);
}
```

- De obicei, un constructor este public, dar poate fi și privat într-unul din următoarele cazuri:
 - nu dorim să fie instantiată o anumită clasă (de exemplu, dacă aceasta este o clasă de tip utilitar care conține doar date membre/metode statice - clasa Math)
 - dorim să instantiem un singur obiect din clasa respectivă (clasă singleton)

➤ Exemplu pentru Singleton Design Pattern:

- Considerăm o aplicație Java care modelează activitatea dintr-o organizație utilizând câte o clasă pentru fiecare rol (angajat, director de departament, președinte) în parte. Evident, orice organizație are un singur președinte, deci clasa President care modelează acest rol trebuie să permită o singură instanțiere a sa!
- Pentru a realiza o instanțiere unică a clasei President vom proceda astfel:
 - constructorul implicit va fi privat, pentru a împiedica instanțierea clasei;
 - vom utiliza un câmp static care pentru a reține referința singurei instanțe a clasei;
 - vom utiliza o metodă statică de tip *factory* pentru a furniza referința spre singura instanță a clasei.

```
class President {
    private static String name;
    private static President president;

    private President() {
        name = "Mr. John Smith";
    }

    public static President getPresident() {
        if (president == null)
            president = new President();
        return president;
    }

    public static void showPresident() {
        System.out.println("President: " + name);
    }
}
```

```
public class Test{
    public static void main(String[] args) {
        President p = President.getPresident();
        President q = President.getPresident();
        System.out.println(p == q);
    }
}
```

- Se observă faptul că referințele p și q sunt egale, deci am realizat o clasă singleton!
- De asemenea, se observă faptul că singura instanță a clasei este creată doar în momentul în care aceasta este solicitată, adică este apelată metoda factory getPresident. În acest caz spunem că se realizează o *instantiere târzie* (lazy initialization).
- O altă variantă de implementare a unei clase singleton constă în crearea singurei instanțe să chiar din momentul în care clasa este încărcată de mașina virtuală Java, printr-o *initializare timpurie* (early initialization):

```
class President {
    private static String name;
    private static final President president = new President();

    private President() {
        name = "Mr. John Smith";
    }

    public static President getPresident() {
        return president;
    }

    public void showPresident(){
        System.out.println("President: " + name);
    }
}
```

➤ Constructorul de copiere

În limbajul Java nu există constructor de copiere având funcționalitățile din limbajul C++!

Evident, o clasă poate să conțină un constructor având ca parametru un obiect al clasei respective, în scopul de a copia în obiectul curent datele membre ale obiectului transmis ca parametru. Totuși, acest constructor **nu va fi apelat automat** în cazurile în care se apelează un constructor de copiere în alte limbi orientate obiect (de exemplu, în limbajul C++, pentru a realiza o copie a unui parametru al unei metode transmis prin valoare).

În limbajul Java datele membre pot fi copiate în două moduri:

- **bitwise/shallow copy**: se copiază datele membre bit cu bit, inclusiv referințele!

Exemplu:

Adăugăm în clasa Persoana data membră double[] venit care va conține veniturile persoanei respective în fiecare dintre cele douăsprezece luni ale unui an:

```
class Persoana {
    private int IDPersoana;
    private String nume;
    private int varsta;
    private static String nationalitate = "română";
    private static int nrPersoane = 0;
    private double []venit;

    public Persoana() {
        this.nume = "";
        this.venit = new double[12];
    }

    public Persoana(String nume, int varsta, int[] venit) {
        this.nume = nume;
        this.varsta = varsta;

        this.venit = new double[12];
        for(int i = 0; i < venit.length; i++)
            this.venit[i] = venit[i];
    }

    public Persoana(Persoana ob) {
        this.nume = ob.nume;
        this.varsta = ob.varsta;
        this.venit = ob.venit;
    }

    public void setVenit(int luna, double suma) {
        venit[luna] = suma;
    }

    public double getVenit(int luna) {
        return venit[luna];
    }
    .....
}
```

Considerăm acum următoarea secvență de cod:

```

Persoana p1 = new Persoana("Pop Ana", 24,
                           new double[]{2000.25, 3000.50, 4000.75});
Persoana p2 = new Persoana(p1);

p2.setVenit(2, 5000);

System.out.println("Veniturile persoanei p1: " + p1.getVenit(2));
System.out.println("Veniturile persoanei p2: " + p2.getVenit(2));

```

După rularea secvenței de cod de mai sus, se va afișa:

```

Veniturile persoanei p1: 5000.0
Veniturile persoanei p2: 5000.0

```

Din cauza faptului că prin utilizarea constructorului Persoana (Persoana ob) în obiectul p2 a fost copiată referința tabloului venit din obiectul p1, modificarea venitului persoanei p2 într-o anumită lună a condus și la modificarea venitului persoanei p1 în luna respectivă!

- **deep copy:** pentru datele membre de tip referință se alocă mai întâi spațiu de memorie, după care se copiază explicit conținutul acestora.

Exemplu:

Modificăm în clasa Persoana constructorul Persoana (Persoana ob) astfel:

```

public Persoana(Persoana ob) {
    this.nume = ob.nume;
    this.varsta = ob.varsta;

    this.venit = new double[12];
    for (int i = 0; i < venit.length; i++)
        this.venit[i] = venit[i];
}

```

➤ **Blocuri de inițializare**

- Constructorii nu trebuie să initializeze date membre statice, dar le pot manipula.

Exemplu: pentru a inițializa data membră IDPersoana cu numărul curent de obiecte de tip Persoana instanțiate, trebuie să adăugăm în fiecare constructor instrucțiunea `this.IDPersoana = ++nrPersoane.`

- Totuși, există posibilitatea de a folosi un **bloc static de inițializare** a datelor membre statice. Acest bloc se apelează o singură dată, când JVM încarcă clasa respectivă. De obicei, un astfel de bloc se folosește când sunt necesare inițializări mai complexe ale datelor membre statice (conectarea la o bază de date sau un server, citirea unor informații dintr-un fișier etc.).

Exemplu: eliminăm din clasa Persoana inițializările directe ale datelor membre statice și adăugăm un bloc static de inițializare:

```
class Persoana {
    .....
    private static String nationalitate;
    private static int nrPersoane;

    static{
        nationalitate = "română";
        nrPersoane = 0;
    }
    .....
}
```

- De asemenea, se poate declara un **bloc nestatic de inițializare**, care va fi executat înaintea fiecărui apel de constructor. Acest bloc conține, de regulă, o secțiune de cod comună tuturor constructorilor. De exemplu, pentru a inițializa data membră IDPersoana cu numărul curent de obiecte de tip Persoana instanțiate, nu mai adăugăm instrucțiunea `this.IDPersoana = ++nrPersoane` în fiecare constructor, ci folosim un bloc nestatic de inițializare:

```
class Persoana {
    private int IDPersoana;
    private static int nrPersoane;
    .....
    static{
        nationalitate = "română";
        nrPersoane = 0;
    }

    {
        this.IDPersoana = ++nrPersoane;
    }

    public Persoana() {
        this.nume = "";
        this.venit = new double[12];
    }
    .....
}
```

Exemplu:

```
class Persoana {  
    String nume;  
    int varsta, ID;  
    static int nrPersoane;  
    static String nationalitate;  
  
    public Persoana() {  
        this.nume = "XYZ";  
        this.varsta = 20;  
        System.out.println("Constructorul cu 0 parametri");  
    }  
  
    public Persoana(String nume) {  
        this.nume = nume;  
        this.varsta = 20;  
        System.out.println("Constructorul cu 1 parametru");  
    }  
  
    public Persoana(String nume, int varsta) {  
        this.nume = nume;  
        this.varsta = varsta;  
        System.out.println("Constructorul cu 2 parametri");  
    }  
  
    //bloc nestatic de initializare  
    {  
        this.ID = ++nrPersoane;  
        System.out.println("Bloc nestatic de initializare");  
    }  
  
    //bloc static de initializare  
    static  
    {  
        System.out.println("Bloc static de initializare\n");  
        nationalitate = "română";  
        nrPersoane = 0;  
    }  
  
    @Override  
    public String toString() {  
        return "nume='" + nume + ", varsta=" + varsta + ", ID=" + ID;  
    }  
}
```

```

public class Test
{
    public static void main(String[] args)
    {
        Persoana p1 = new Persoana();
        System.out.println(p1);
        System.out.println();

        Persoana p2 = new Persoana("Popescu Ion");
        System.out.println(p2);
        System.out.println();

        Persoana p3 = new Persoana("Ionescu Dana", 25);
        System.out.println(p3);
        System.out.println();
    }
}

```

➤ Obiecte

- **Ciclul de viață al unui obiect:**
 - declararea unei referințe (și inițializarea sa cu null dacă este o variabilă locală):


```
Persoana p = null;
```
 - instanțierea obiectului folosind operatorul new (alocare dinamică în heap) și un constructor al clasei respective:


```
p = new Persoana(nume, vârstă);
```
 - un obiect poate fi instantiat și în momentul declarării sale:


```
Persoana p = new Persoana(nume, vârstă);
```
 - utilizarea obiectului prin intermediul metodelor sale publice:


```
p.setNume("Popescu Ion");
System.out.println(p.getNume());
```
 - distrugerea automată a obiectului
- O data membră/metodă statică poate fi apelată și cu o referință null:


```
Persoana p = null;
p.afisareNumarPersoane()
```
- **Distrugerea obiectelor se realizează automat în limbajul Java!**
- **Garbage Collection** este mecanismul prin care JVM eliberează spațiul alocat unor obiecte care nu mai sunt folosite, utilizând un fir de executare dedicat, numit **Garbage Collector (GC)**. Acest fir scanează memoria și verifică faptul că o zonă de memorie mai

este utilizată sau nu, marcând zonele nefolosite. Ulterior, zonele de memorie marcate sunt eliberate (sunt raportate ca fiind libere) și, eventual, se realizează o compactare a memoriei.

- Un obiect se consideră ca fiind neutilizat dacă, de exemplu:
 - nu mai există nicio referință, directă sau indirectă, spre obiectul respectiv;
 - obiectul a fost creat în interiorul unui bloc (local) și executarea blocului respectiv s-a încheiat;
 - obiectul face parte dintr-o insulă de izolare (*island of isolation*), adică un grup de obiecte între care există referințe, dar spre niciunul dintre ele nu mai există referințe din exteriorul grupului (<https://javasolutionsguide.blogspot.com/2015/08/how-to-make-object-eligible-for-garbage.html>).
- Înainte de a distruge un obiect, GC apelează metoda `finalize` pentru a-i oferi obiectului respectiv posibilitatea de a mai executa un set de acțiuni.
- Un obiect neutilizat nu va fi neapărat distrus imediat și nu se poate forța pornirea GC folosind `System.gc()` sau `Runtime.getRuntime().gc()` !

EXTINDEREA CLASELOR

Numita și **derivare, moștenirea** este un mecanism de refolosire a codului, specific limbajelor orientate obiect și reprezintă posibilitatea de a defini o clasă care extinde o alta clasă deja existentă, preluând funcționalitățile sale și adăugând altele noi. Pe lângă reutilizarea codului, moștenirea oferă posibilitatea de a dezvolta pas cu pas o aplicație, procedeu care poarta numele de *dezvoltare incrementală (incremental development)*. Astfel, se poate utiliza un cod deja funcțional, respectiv testat, și adăuga un alt cod nou la acesta, în felul acesta izolându-se eventualele erori din codul nou adăugat.

Terminologie: Clasa care se extinde se numește *superclasă*, iar cea care preia datele și funcționalitățile se numește *subclasă*.

Sintaxa:

```
class Subclasa extends Superclasa
{
    date și metode membre noi
}
```

Moștenirea definește o relație între superclasă și subclasa sa de tip IS_A și conduce definirea unei ierarhii de clase care are rădăcina în clasa **Object**. Practic, orice clasa din Java extinde clasa **Object**, denumită *superclasa claselor*. Din sintaxa de mai sus se poate observa faptul că, în limbajul Java, moștenirea este întotdeauna *publică și singulară*, respectiv membrii din superclasă își păstrează modificatorul de acces, iar orice clasă din ierarhia de clase are o singură superclasă directă.

Ce se moștenește?

- Subclasă moștenește toți membrii publici, protejați și implicați din superclasă, indiferent dacă sunt statici sau non-statici (de instanță).
- Membrii privați nu sunt moșteniți, dar pot fi accesați prin metode publice sau protejate din superclasă.
- Metodele constructor, nefiind considerate metode membre ale unei clase, nu se moștenesc, dar un constructor din subclasă poate apela constructorii din superclasă folosind expresia `super([argumente])`.
- Cuvântul cheie `super` poate fi folosit și pentru a accesa date membre și metode din superclasă, astfel: `super.metoda(lista arg)` sau `super.dată_membă`.

Observații:

- La instanțierea unui obiect de tip subclasă se apelează constructorii din ambele clase, mai întâi cel din superclasă și apoi cel din subclasă!
- Apelul constructorului superclasei, dacă există, trebuie să fie prima instrucțiune din constructorul subclasei (un obiect al subclasei este mai întâi de tipul superclasei).
- Dacă nu se introduce în subclasă apelul explicit al unui constructor al superclasei, atunci compilatorul va încerca să apeleze constructorul fără argumente al superclasei. Aceasta poate cauza o eroare în cazul în care superclasa nu are definit un constructor fără argumente, dar are definit unul cu argumente!
- Constructorul cu argumente din subclasă conține, de regulă, argumente pentru inițializarea datelor membre din superclasă.

Exemplu:

```

class Persoana
{
    String nume;
    int varsta;

    public Persoana(String nume , int varsta)
    {
        this.nume = nume;
        this.varsta = varsta;
    }

    @Override
    public String toString() { return nume + " " + varsta; }

    //metode de tip set/get pentru datele membre nume și vârsta
}

class Student extends Persoana
{
    String facultate;
    int grupa;
    double medie;

    public Student(String nume, int varsta, String facultate, int grupa, double medie)
    {
        super(nume, varsta);
        this.facultate = facultate;
        this.grupa = grupa;
        this.medie = medie;
    }

    @Override
    public String toString()
    {
        return super.toString() + " " + facultate + " " + grupa + " " + medie;
    }

    //metode de tip set/get pentru datele membre facultate, medie și grupă
}

```

Mecanismul de redefinire a unei metode (overriding)

Mecanismul de redefinire reprezintă un concept puternic în limbajele orientate pe obiecte care permite subclasei să redefinească o metoda moștenită și să-i modifice comportamentul. Astfel, la executare, în raport cu tipul obiectului se va invoca metoda corespunzătoare. Practic, metoda din superclasă este „ascunsă” de cea redefinită în subclasă sa.

Observații:

- O metodă din subclasă care redefineste o metodă din superclasă trebuie să păstreze lista inițială a parametrilor formali.

- Nu se pot redefini metodele de tip `final`.
- Se poate utiliza adnotarea `@Override` pentru a-i indica explicit compilatorului faptul că se va redefini o metodă din superclasa. Astfel, dacă metoda respectivă nu există în superclasa, compilatorul va furniza o eroare.
- O dată membră din superclasa va fi ascunsă prin redeclararea sa în subclasa, dar poate fi totuși accesată prin `super.dată_membră`.
- O metodă din superclasa poate fi redefinită în subclasa, dar poate fi totuși accesată prin `super.metodă([parametrii])`.
- Metodele statice pot fi redefinite doar prin metode statice (dar le ascund!!!), iar cele nestatice doar prin metode nestatice.
- Pentru o metodă redefinită se poate schimba modifierul de acces, dar fără ca nivelul de acces să scadă.
- Tipul returnat de o metodă din subclasa care redefinește o metodă din superclasa trebuie să fie unul covariant tipului de date inițial, respectându-se astfel *principiul de covarianță* (sau *principiul de substituție Liskov*):

- `void ↔ void`
- `tip de date primitiv ↔ același tip de date primitiv`
- `referință de tip superclasa ↔ referință de tip superclasa sau de tip subclasa`

Exemplu:

```
class Angajat extends Persoana
{
    String firma , functie;
    double salariu;

    public Angajat(String nume, int varsta, String firma, String functie,
                    double salariu)
    {
        super(nume, varsta);
        this.firma = firma;
        this.functie = functie;
        this.salariu = salariu;
    }

    double calculeazaVenit()
    {
        return salariu;
    }
}

class Economist extends Angajat
{
    //treapta profesională -> număr natural cuprins între 0 și 5
    int treapta_profesionala;
    //sporurile procentuale corespunzătoare treptelor profesionale
    static final double sporuri[] = {5, 10, 15, 20, 25, 30};
```

```

public Economist(int treapta_profesionala, String nume, int varsta,
                  String firma, String functie, double salariu) {
    super(nume, varsta, firma, functie, salariu);
    this.treapta_profesionala = treapta_profesionala;
}

@Override
double calculeazaVenit() {
    return salariu + salariu*sporuri[treapta_profesionala]/100;
}
}

```

Mecanismul de redefinire, alături de cel de supraîncărcare, permite definirea polimorfismului. Totuși, între cele două concepte există diferențe clare:

SUPRAÎNCĂRCARE (OVERLOADING)	REDEFINIRE (OVERRIDING)
• lista parametrilor formali trebuie să fie diferită	• lista parametrilor formali trebuie să fie identică
• tipul de date returnat nu contează	• tipul de date returnat trebuie să respecte principiul de covarianță
• se poate realiza și doar în cadrul unei singure clase (dar nu este obligatoriu!!!)	• se poate realiza doar într-o subclasă a unei superclase
• metodele de tip final sau private pot fi supraîncărcate	• metodele de tip final nu pot fi rescrise
• nivelul de acces nu contează	• nivelul de acces nu trebuie să fie mai restrictiv decât cel al metodei din superclasa
• la compilare (legare statică)	• la rulare (legare dinamică)
• mai rapidă	• mai lentă

Mai multe detalii referitoare la supraîncărcarea și redefinirea metodelor găsiți în pagina <https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.9> și în pagina <https://www.bestprog.net/en/2019/06/27/java-inheritance-overriding-and-overloading-inherited-methods-examples-dynamic-methods-scheduling/>.

Polimorfismul în limbajul Java

Conceptul în sine este o consecință a faptului ca orice obiect din Java poate fi referit prin tipul său sau prin tipul unei superclase, respectiv se poate realiza, într-un mod implicit, conversia unei subclase la o superclă a sa mecanism care poartă denumirea de *upcasting*.

Considerăm clasa A extinsă de clasa B. Pot avea loc următoarele instanțieri:

```
B b = new B();      //referirea obiectului printr-o referință de tipul clasei
A a = new B(...); //referirea obiectului printr-o referință de tipul superclasei
```

Observație: În cazul celei de-a doua instanțieri, nu este convertită valoarea, ci referința obiectului!!! Spunem că obiectul a are *tipul declarat A și tipul real B!*

Considerăm o clasă A care este extinsă de o clasă B, astfel:

```
class A {

    int dată_membră_non_statică = 1;
    static int dată_membră_statică = 1000;

    void metoda1() {
        System.out.println("Metoda non-statică 1 din clasa A!");
    }

    static void metoda2() {
        System.out.println("Metoda statică 2 din clasa A!");
    }
}

class B extends A {

    int dată_membră_non_statică = 2;
    static int dată_membră_statică = 2000;

    void metoda1() {
        System.out.println("Metoda non-statică 1 din clasa B!");
    }

    static void metoda2() {
        System.out.println("Metoda statică 2 din clasa B!");
    }
}
```

Se poate observa cum subclasa B redefinește atât datele membre, cât și cele două metode.

Rulând secvența de instrucțiuni

```
A ob = new B(); //polimorfism
```

```
System.out.println("Data membră non-statică = " + ob.dată_membră_non_statică);
System.out.println("Data membră statică = " + ob.dată_membră_non_statică);
ob.metoda1();
ob.metoda2();
```

se va afișa:

```
Data membră non-statică = 1
```

```
Data membră statică = 1
Metoda non-statică 1 din clasa B!
Metoda statică 2 din clasa A!
```

Analizând exemplul de mai sus, observăm următoarele aspecte:

- datele membre sunt ambele egale cu 1, deoarece ele se accesează după tipul declarat, ci nu după tipul real, indiferent dacă sunt date membre statice sau non-statice (de instanță);
- ob.metoda1() utilizează implementarea din subclasă, respectiv metoda redefinită, deoarece selecția sa se realizează după tipul real;
- ob.metoda2() utilizează implementarea din superclasă, deoarece metodele statice nu se redefinesc, deci selecția sa se realizează după tipul declarat.

În concluzie, la *upcasting*, o metodă de instanță este invocată în raport de tipul real, tip care se identifică la executare (*runtime*). Conceptul se mai numește și *legare dinamică* sau *legare târzie* (late binding).

În sens invers, *downcasting*-ul reprezintă accesarea unui obiect de tipul superclasei folosind o referință de tipul unei subclase și nu este implicit, necesitând o conversie explicită!

Exemplu: Considerăm clasa Angajat și două subclase ale sale, Economist și Inginer.

```
//corect, deoarece upcasting-ul se realizează implicit
Angajat a = new Economist();
Angajat b = new Inginer();

//eroare la compilare, deoarece downcasting-ul nu se realizează implicit
Inginer p = b;

//fără eroare la compilare, deoarece downcasting-ul a fost realizat explicit
Inginer p = (Inginer)b;
```

Totuși, în momentul executării ultimei instanțieri de mai sus, se va declanșa excepția ClassCastException și rularea programului se va termina. Pentru a preveni acest lucru, se verifică în prealabil dacă se poate efectua *downcasting*-ul respectiv, folosind operatorul instanceof, astfel:

```
Inginer p = null;
if(b instanceof Inginer)
    p = (Inginer)b;
```

În continuare, vom prezenta o modalitate prin care un tablou unidimensional poate să conțină informații eterogene, respectiv elementele sale să fie referințe spre instanțe ale unor clase diferite (dar care trebuie să fie toate subclase ale unei singure superclase!):

```
class Angajat {

    String nume;
    double salariu;
    int varsta;
```

```

public Angajat(String nume, double salariu, int varsta) {
    this.nume = nume;
    this.salariu = salariu;
    this.varsta = varsta;
}

double calculeazaVenit() {
    return salariu;
}
}

class Economist extends Angajat {

//treapta profesională -> număr natural cuprins între 0 și 5
int treapta_profesionala;
//sporurile procentuale corespunzătoare treptelor profesionale
static final double sporuri[] = {5, 10, 15, 20, 25, 30};

public Economist(String nume, double salariu, int varsta, int
treapta_profesionala) {
    super(nume, salariu, varsta);
    this.treapta_profesionala = treapta_profesionala;
}

@Override
double calculeazaVenit() {
    return salariu + salariu * sporuri[treapta_profesionala] / 100;
}

void afiseazaSporEconomist() {
    double spor = salariu * sporuri[treapta_profesionala] / 100;
    System.out.println("\tSpor economist: " + spor);
}
}

class Paznic extends Angajat {
    double spor_noapte;

    public Paznic(String nume, double salariu, int varsta, double spor_noapte) {
        super(nume, salariu, varsta);
        this.spor_noapte = spor_noapte;
    }

    @Override
    double calculeazaVenit() {
        return salariu + salariu * spor_noapte;
    }

    void afiseazaSporPaznic() {
        double spor = salariu * spor_noapte;
        System.out.println("\tSpor paznic: " + spor);
    }
}

```

```

public class Test {
    public static void main(String[] args) {
        //declarăm un tablou cu elemente de tipul superclasei Angajat,
        //comună subclaszelor Paznic și Economist
        Angajat t[] = new Angajat[4];

        //folosind upcasting-ul, atribuim elementelor tabloului referințe
        //spre instanțe ale celor două subclase (Paznic și Economist)
        t[0] = new Paznic("Popa Ion", 2500, 40, 0.25);
        t[1] = new Economist("Mihai Ana", 3700, 55, 4);
        t[2] = new Economist("Popescu Maria", 3000, 47, 2);
        t[3] = new Paznic("Ionescu Mihai", 2100, 35, 0.20);

        //afișăm informațiile despre fiecare angajat
        for(int i = 0; i < t.length; i++)
        {
            System.out.println("Angajat " + (i+1) + ":");
            System.out.println("\tNume: " + t[i].nume);
            //datorită polimorfismului, pentru fiecare angajat
            //se va apela metoda calculeazaVenit() din subclasa corespunzătoare
            System.out.println("\tVenit: " + t[i].calculeazaVenit());

            //pentru a putea apela metodele proprii fiecărei subclase,
            //utilizăm downcasting-ul
            if(t[i] instanceof Paznic)
            {
                Paznic aux = (Paznic)t[i];
                aux.afiseazaSporPaznic();
            }
            else
                if(t[i] instanceof Economist)
                {
                    Economist aux = (Economist)t[i];
                    aux.afiseazaSporEconomist();
                }
            System.out.println();
        }
    }
}

```

Clasa Object

Clasa `Object`, definită în pachetul `java.lang`, definește și implementează un comportament comun pentru orice obiect Java. Fiecare clasă Java este un descendant al clasei `Object`, astfel încât orice clasă moștenește metodele clasei `Object`. Aceste metode nu se utilizează în orice aplicație, dar, dacă sunt folosite, atunci trebuie cunoscute câteva principii de redefinire a lor.

- **`public final Class getClass()`**

- Este o metodă de tip `final` (nu mai poate fi redefinită) care returnează un obiect de tip `Class` care conține detalii despre clasa instanțiată în momentul executării programului.

- Clasa `Class` este definită în `java.lang`, nu are constructor public, astfel încât obiectul este construit implicit de către mașina virtuală Java cu ajutorul unor metode de tip *factory*.
- **`public String toString()`**
 - Metoda returnează o reprezentare a obiectului sub forma unui obiect de tip `String`.
 - Trebuie redefinită în fiecare clasă, deoarece acesta reprezentare este dependentă de fiecare obiect. De regulă, se construiește un sir de caractere care conține valorile câmpurilor.
 - Implicit metoda `toString` afișează un sir format astfel:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

- **`boolean equals(Object obj)`**
 - În limbajul Java, două obiecte se pot compara în două moduri, folosind:
 - **operatorul ==** care verifică dacă două obiecte sunt egale din punct de vedere al referințelor, adică dacă au aceeași referință, însă nu se testează echivalența lor din punct de vedere al conținutului!

```
Persoana p1 = new Persoana("Matei", 23);
Persoana p2 = p1;
System.out.println(p1==p2);           //se va afișa true
```

```
Persoana p3 = new Persona("Matei", 23);
System.out.println (p1==p3);          //se va afișa false
```

- **metoda boolean equals()** care, implicit, verifică dacă două obiecte au aceeași referință, folosind operatorul `==`, după cum se poate observa din exemplul următor:

```
System.out.println (p1.equals(p3));    //se va afișa false
```

- De cele mai multe ori, în aplicații este necesară o comparare a două obiecte și din punct de vedere al conținutului, caz în care trebuie redefinită metoda `equals()`.

Exemplu: Redefinirea metodei `equals` pentru clasa `Persoana` cu datele membre nume și varsta:

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;

    if (obj == null)
        return false;

    if (getClass() != obj.getClass())
        return false;

    Persoana pers = (Persoana) obj;
    if (this.varsta != pers.varsta)
        return false;
```

```

        if (!Objects.equals(this.nume, pers.nume))
            return false;

        return true;
    }
}

```

- Se poate observa faptul că pentru a compara numele celor două persoane se preferă utilizarea metodei `equals()` din clasa utilitară `Objects` (introdusă din versiunea 1.7), deoarece metodele `equals()` și `compareTo()` din clasa `String` pot furniza excepții în cazul în care argumentele sunt de tip `null`. În schimb, metoda `equals()` din clasa `Objects` va compara cele două șiruri astfel:
 - dacă ambele obiecte `String` sunt de tip `null`, atunci metoda va returna valoarea `true`;
 - dacă unul dintre obiecte `String` este `null`, iar celălalt nu este, va returna `false`;
 - dacă ambele obiecte de tip `String` nu sunt `null`, va returna rezultatul obținut prin apelarea metodei `equals()` din clasa `String`.
- **`public int hashCode()`**
 - În Java, codul hash al unui obiect este un număr întreg care este dependent de conținutul său. Implicit, codul hash este calculat de către mașina virtuală Java, utilizând un algoritm specific care nu utilizează valorile câmpurilor obiectului respectiv. Astfel, pentru a se evidenția faptul că un obiect își modifică starea pe parcursul executării programului sau pentru a ne asigura de faptul că două obiecte egale din punct de vedere al conținutului au același cod hash, se redefineste metoda `hashCode()`, astfel încât algoritmul de hash să utilizeze concret starea obiectului (valorile datelor membre). În acest sens, clasa `Objects` conține o metodă cu număr variabil de argumente care calculează codul hash pentru un anumit obiect folosind valorile câmpurilor sale: `Objects.hash(câmp_1, câmp_2, ...)`.

Exemplu: Calculul codului hash pentru un obiect de tip `Persoana`

```

@Override
public int hashCode() {
    return Objects.hash(nume, varsta);
}

```

Mai multe detalii referitoare la implementarea metodei `hashCode()` găsiți în pagina <https://www.baeldung.com/java-hashcode>.

Observație: Implementările metodelor `equals()` și `hashCode()` trebuie să țină cont de următoarele reguli:

- metoda `hashCode()` trebuie să returneze aceeași valoare în timpul rulării unei aplicații, indiferent de câte ori este apelată, dacă starea obiectului nu s-a modificat, dar nu trebuie neapărat să furnizeze aceeași valoare în cazul unor rulări diferite;
- două obiecte egale din punct de vedere al metodei `equals()` trebuie să fie egale și din punct de vedere al metodei `hashCode()`, deci trebuie să aibă și hash code-uri egale;
- două obiecte diferite din punct de vedere al conținutului nu trebuie neapărat să aibă hash-code-uri diferite, dar, dacă acest lucru este posibil, se vor obține performanțe mai bune pentru operațiile asociate unei tabele de dispersie.

Clase abstracte

În momentul în care dorim să abstractizăm un anumit concept/fenomen din lumea reală, este posibil să nu-i putem defini comportamentul său în orice situație. De exemplu, despre orice angajat, indiferent de domeniul său de activitate, trebuie să cunoaștem anumite informații precum nume, adresă, firma la care este angajat, numărul de ore lucrate zilnic etc. De asemenea, pentru orice angajat trebuie să putem calcula numărul total de ore lucrate de acesta într-o anumită perioadă de timp, să-i determinăm vechimea totală etc. Totuși, modalitatea de calcul a salariului unui angajat depinde de profesia sa (inginer, profesor, medic etc.), de anumite sporuri specifice (are anumite deduceri de impozit, lucrează noaptea sau nu, lucrează în week-end-uri, lucrează în mediu toxic sau nu etc.). În acest caz, o metodă care să calculeze salariul unui angajat fie va lua în considerare toate cazurile posibile, ceea ce conduce la o eficiență scăzută, fie va fi declarată abstractă, urmând a fi implementată **obiectiv** în clase specializede (Profesor, Inginer, Medic etc.), care vor extinde clasa Angajat.

Exemplu:

```
public abstract class Angajat {
    double salariu_baza;
    .....
    abstract public double calculSalariu();
}

class Paznic extends Angajat{
    .....
    static double spor_de_noapte = 0.25;
    .....
    Public double calculSalariu() {
        return salariu_baza + salariu_baza * spor_de_noapte;
    }
}
```

Observații:

- O clasă abstractă nu se poate instanția, deoarece nu se cunoaște integral funcționalitatea sa.
- Dacă o subclasă a unei clase abstracte nu oferă implementări pentru toate metodele abstracte moștenite, atunci subclasa este, de asemenea, abstractă, deci nu poate fi instantiată.
- O clasă abstractă poate să conțină date membre de instanță, constructori și metode publice, astfel încât subclasele sale pot apela constructorul din superclasa, respectiv pot redefini membrii săi.

Polimorfismul se poate implementa cu ușurință folosind clase abstracte. În exemplul de mai sus, clasa Angajat este o clasă abstractă, deci nu poate fi instantiată. Astfel, în momentul compilării este posibil să nu se cunoască tipul concret al unui angajat, dar un obiect de tip subclasă (Inginer, Profesor etc.) poate fi accesat printr-o referință de tipul superclasei, respectiv prin referința clasei abstracte (polimorfism!).

Exemplu: Considerăm clasa abstractă Angajat, care conține metoda abstractă double calculSalariu() și 3 subclase ale sale Inginer, Profesor și Economist, toate trei implementând în mod specific metoda calculSalariu() și având definiții constructori și alte metode necesare. Fișierul text angajați.csv conține pe fiecare linie informații despre fiecare angajat al unei firme, despărțite între ele prin virgule, iar fiecare linie începe cu un sir de caractere care indică profesia angajatului respectiv. Pentru a încărca

într-un singur tablou unidimensional informațiile despre toți angajații firmei, indiferent de profesie, vom declara un tablou a cu elemente de tipul clasei abstracte `Angajat` (de fapt, referințe!) și, folosind polimorfismul, vom instanția pentru fiecare element al tabloului un obiect de tipul uneia dintre cele 3 subclase:

```
.....
Scanner in = new Scanner(new File("exemplu.txt"));
int n = in.nextInt();
in.nextLine();
Angajat []a = new Angajat[n];

String linie;
for(int i = 0; i < n; i++)
{
    linie = in.nextLine();
    String []aux = linie.split(",");
    switch(aux[0].toUpperCase())
    {
        //se apelează constructorul corespunzător fiecărei subclase
        case "PROFESOR":
            a[i] = new Profesor(...);
            break;
        case "INGINER":
            a[i] = new Inginer(...);
            break;
        case "ECONOMIST":
            a[i] = new Economist(...);
            break;
    }
}

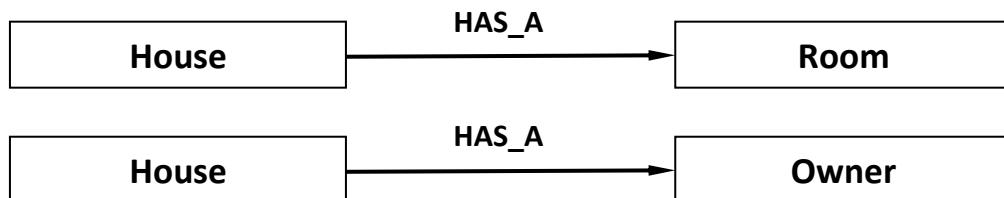
for(int i = 0; i < n; i++)
    System.out.println(a[i].getNumar() + " -> " + a[i].calculSalariu() + " RON");
.....
```

Datorită polimorfismului, pentru fiecare element al tabloului a se va apela varianta metodei `calculSalariu()` din subclasa sa!

AGREGARE ȘI COMPOZIȚIE

- Agregarea și compozitia reprezintă alte două modalități de interconectare (asociere) a două clase, alături de mecanismul de extindere a claselor (moștenire).
- Practic, agregarea și compozitia reprezintă alte modalități de reutilizare a codului.
- Asocierea a două clase se realizează prin încapsularea în clasa container a unei referințe, de un tip diferit, către un obiect al clasei asociate (încapsulate).
- Conceptual, compozitia este diferită de agregare în raport de ciclul de viață al obiectului încapsulat, astfel:
 - dacă ciclul de viață al obiectului încapsulat este dependent de ciclul de viață al obiectului container, atunci relația de asociere este de tip *compoziție* (strong association);
 - dacă obiectul încapsulat poate să existe și după distrugerea containerului său, atunci relația de asociere este de tip *agregare* (weak association).

Exemple:



- Se poate observa cu ușurință faptul că relația de asociere dintre clasele House și Room este una de tip compozitie (dacă este distrusă întreaga casă, atunci, în mod automat, va fi distrusă și camera respectivă), iar relația de asociere dintre clasele House și Owner este una de tip agregare (chiar dacă este distrusă întreaga casă, proprietarul său poate să trăiască în continuare).
- Din punct de vedere al implementării, diferențierea dintre cele două tipuri de asocieri se realizează prin modul în care obiectul container încapsulează referința spre obiectul asociat. Astfel, în cazul unei relații de agregare este suficientă încapsularea în clasa container a unei referințe spre obiectul asociat, deoarece acesta poate exista și independent:

<pre> class Person{ private String name; private String SSN; } </pre>	<pre> class House{ private String address; private Person owner; public House(Person owner,...) { this.owner = owner; } } </pre>
---	--

În cazul unei relații de compozиție se va încapsula în clasa container o referință a unei copii locale a obiectului asociat:

```
class Room{
    private float width;
    private float length;
    .....

    public Room(Room r) {
        this.width = r.width;
        .....
    }
}

class House{
    private String address;
    private Room dining;
    .....

    public House(Room dining,...) {
        this.dining = new Room(dining);
        .....
    }
}
```

- În exemplul de mai sus, am presupus faptul că în clasa Room este definit un constructor care să initializeze obiectul curent de tip Room cu valorile altui obiect de acest tip ("constructor de copiere"). Evident, dacă acest constructor nu există, se va utiliza unul dintre constructorii existenți, eventual împreună cu metode de tip set/get.
- În concluzie, compozиția și agregarea sunt relații de tip HAS_A, care se folosesc în momentul în care dorim să reutilizăm o clasă existentă, dar nu există o relație de tipul IS_A între ea și noua clasă, deci nu putem să utilizăm moștenirea.
- Cu alte cuvinte, dacă noua clasă este asemănătoare, din punct de vedere al modelarii, cu o clasă definită anterior, atunci se va utiliza extinderea, realizându-se o specializare a sa prin redefinirea unor metode. Dacă noua clasă nu este asemănătoare cu o clasă deja definită, dar are nevoie de metodele sale (fără a le modifica!), atunci se va utiliza compozиția sau agregarea.
- Agregarea se va utiliza în cazul în care obiectul container nu poate controla complet obiectul asociat (extern), acesta fiind creat/modificat de alte obiecte, iar compozиția se va utiliza când obiectul container trebuie să aibă control complet asupra obiectului asociat (dacă nu furnizăm metode de acces pentru obiectul asociat, atunci nimeni din exterior nu-l poate modifica).

ȘIRURI DE CARACTERE

- În limbajul Java sunt predefinite 3 clase pentru manipularea la nivel înalt a șirurilor de caractere:
 1. clasa String
 2. clasa StringBuilder
 3. clasa StringBuffer
- De asemenea, șirurile de caractere poate fi implementate și manipulate direct (fără a utiliza metode predefinite din cele 3 clase menționate mai sus), prin intermediul tablourilor cu elemente de tip char. Deși această abordare are dezavantajul unor implementări mai complicate, acest lucru este compensat de o viteză de execuție mai mare și o utilizare mai eficientă a memoriei!

CLASA String

- Folosind clasa String, un șir de caractere poate fi instantiat în două moduri:
 1. `String s = "exemplu";`
 2. `String s = new String("exemplu");`
- Diferența dintre cele două metode constă în zona de memorie în care va fi alocat șirul respectiv:
 1. Se va utiliza o zona de memorie specială, numită *tabelă de șiruri* (string literal/constant pool). Practic, în această zonă se păstrează toate șirurile deja create, iar în momentul în care se va inițializa un nou șir de caractere se va verifica dacă acesta există deja în tabelă. În caz afirmativ, nu se va mai aloca un nou șir în tabelă, ci se va utiliza referința șirului deja existent, ceea ce va conduce la o optimizare a utilizării memoriei (vor exista mai multe referințe spre un singur șir). În momentul în care spre un șir din tabelă nu va mai exista nicio referință activă, șirul va fi eliminat din tabelă.
 2. Se va utiliza zona de memorie heap.

- **Exemplu:**

```

String sir_1 = "exemplu";
String sir_2 = "exemplu";
String sir_3 = new String("exemplu");
String sir_4 = new String("exemplu");
System.out.println(sir_1 == sir_2);      // se va afișa true
System.out.println(sir_3 == sir_4);      // se va afișa false
System.out.println(sir_1 == sir_3);      // se va afișa false
  
```

- Un avantaj foarte important al utilizării tabelei de şiruri îl constituie faptul că operaţia de comparare a două şiruri din punct de vedere al conţinuturilor lor se poate realiza direct, prin compararea referinţelor celor două şiruri, utilizând operatorul ==. Evident, această variantă este mai rapidă decât utilizarea metodei boolean equals(String sir), care verifică egalitatea celor două şiruri caracter cu caracter.
- Un şir de caractere alocat dinamic, folosind operatorul new, poate fi plasat în tabela de şiruri folosind metoda String intern():

```
String sir_1 = "exemplu";
String sir_2 = new String("exemplu");
System.out.println(sir_1 == sir_2);           // se va afişa false
sir_2 = sir_2.intern();
System.out.println(sir_1 == sir_2);           // se va afişa true
```

- Odată creat un şir de caractere, conţinutul său nu mai poate fi modificat. Orice operaţie de modificare a conţinutului său va conduce la construcţia unui alt şir! Astfel, după executarea secvenţei de cod:

```
String sir_1 = "programare";
sir_1.toUpperCase();
System.out.println(sir_1);
```

se va afişa programare! Practic, prin instrucţiunea sir_1.toUpperCase() se va crea un nou şir având conţinutul PROGRAMARE, deci fără a modifica şirul sir_1! Astfel, vor exista două şiruri, unul având conţinutul "programare" şi referinţa păstrată în sir_1, respectiv unul având conţinutul "PROGRAMARE" a cărui referinţă nu este stocată în nicio variabilă! Evident, chiar dacă şirul nu poate fi modificat din punct de vedere al conţinutului, se poate modifica conţinutul unei variabile care conţine referinţa sa: sir_1 = sir_1.toUpperCase(). Astfel, şirul de caractere sir_1 va conţine acum referinţa şirului "PROGRAMARE"!

- În general, dacă instanțele unei clase nu mai pot fi modificate din punct de vedere al conţinutului după ce au fost create, spunem că respectiva clasă este o *clasă imutabilă*.
- Clasa String pune la dispoziţia programatorilor metode pentru:
 1. determinarea numărului de caractere:
 - int length()
 2. extragerea unui subşir:
 - String substring(int beginIndex)
 - String substring(int beginIndex, int endIndex)
 3. extragerea unui caracter:
 - char charAt(int index)

4. compararea lexicografică a două siruri:

- int compareTo(String anotherString)
- int compareToIgnoreCase(String anotherString)
- boolean equals(Object anotherObject)
- boolean equalsIgnoreCase(String anotherString)

5. transformarea tuturor literelor în litere mici sau în litere mari:

- String toLowerCase()
- String toUpperCase()

6. eliminarea spațiilor de la începutul și sfârșitul sirului:

- String trim()

7. căutarea unui caracter sau a unui subșir:

- int indexOf(int ch)
- int indexOf(int ch, int fromIndex)
- int indexOf(String str)
- int indexOf(String str, int fromIndex)
- int lastIndexOf(int ch)
- int lastIndexOf(int ch, int fromIndex)
- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)
- boolean startsWith(String prefix)
- boolean startsWith(String prefix, int toffset)
- boolean endsWith(String suffix)

8. reprezentarea unei valori de tip primitiv sau a unui obiect sub forma unui sir de caractere:

- static String valueOf(boolean b)
- static String valueOf(char c)
- static String valueOf(double d)
- static String valueOf(float f)
- static String valueOf(int i)
- static String valueOf(long l)
- static String valueOf(Object obj)

- Informații detaliate despre toate metodele din clasa String pot fi găsite în pagina: <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>.
- În afara metodelor menționate anterior, în clasa String există mai multe metode care necesită utilizarea unor *expresii regulate* (regex).
- O *expresie regulată* (regex) este o secvență de caractere prin care se definește un şablon de căutare. De obicei, expresiile regulate se utilizează pentru a testa validitatea datelor de intrare (de exemplu, pentru a verifica dacă un sir conține un CNP formal corect) sau pentru realizarea unor operații de căutare/înlocuire/parsare într-un sir de caractere.

- Câteva reguli uzuale pentru definirea unei expresii regulate sunt următoarele:
 - [abc] – şirul este format doar dintr-una dintre literele a sau b sau c
 - [^abc] – şirul este format din orice caracter, mai puțin literele a, b și c
 - [a-z] – şirul este format dintr-o singură literă mică
 - [a-zA-Z] – şirul este format dintr-o singură literă mică sau mare
 - [a-z] [A-Z] – şirul este format dintr-o literă mică urmată de o literă mare
 - [abc] + – şirul este format din orice combinație a literelor a, b și c, iar lungimea sa este cel puțin 1
 - [abc] * – şirul este format din orice combinație a literelor a, b și c, iar lungimea sa poate fi chiar 0
 - [abc] {5} – şirul este format din orice combinație a literelor a, b și c de lungime exact 5
 - [abc] {5, } – şirul este format din orice combinație a literelor a, b și c de lungime cel puțin 5
 - [abc] {5,10} – şirul este format din orice combinație a literelor a, b și c cu lungimea cuprinsă între 5 și 10
- Câteva exemple de utilizare a metodelor care necesită expresii regulate:
 1. pentru a verifica dacă un şir de caractere are o anumită formă particulară se foloseşte metoda boolean matches (String regex):
 - a) şirul s începe cu o literă mare, apoi conține doar litere mici (cel puțin una!):


```
boolean ok = s.matches("[A-Z][a-z]+");
```
 - b) şirul s conține doar cifre:


```
boolean ok = s.matches("[0-9]+");
```
 - c) şirul s conține un număr de telefon Vodafone:


```
boolean ok = s.matches("(072|073)[0-9]{7}");
```
 2. pentru a înlocui în şirul s un subşir de o anumită formă cu un alt şir, folosind metodele String replaceAll(String regex, String replacement), respectiv String replaceFirst(String regex, String replacement):
 - a) înlocuim spațiile consecutive cu un singur spațiu:


```
s = s.replaceAll("[ ]{2,}", " ");
```
 - b) înlocuim cuvântul "are" cu "avea":


```
s = s.replaceAll("\bare\b", "avea");
```
 - c) înlocuim fiecare vocală cu *:


```
s = s.replaceAll("[aeiouAEIOU]", "*");
```
 - d) înlocuim prima vocală cu *:


```
s = s.replaceFirst("[aeiouAEIOU]", "*");
```
 - e) înlocuim grupurile formate din cel puțin două vocale cu *:


```
s = s.replaceAll("[aeiouAEIOU]{2,}", "*");
```

3. pentru a împărți un sir s în subșiruri (stocate într-un tablou de siruri), în raport de anumiți delimitatori, folosind metoda `String[] split(String regex)`:

a) împărțirea textului în caractere:

```
String[] w = s.split("");
```

b) împărțirea textului în cuvinte de lungime nenulă:

```
String[] w = s.split("[ .,:;!?]");
```

c) extragerea numerelor naturale :

```
String[] w = s.split("[^0-9]+");
```

CLASA `StringBuilder`

- Un dezavantaj major al obiectelor imutabile de tip `String` este dat de faptul că orice modificare a unui sir de caractere necesită construcția unui nou sir sau chiar a mai multora. De exemplu, pentru a înlocui al patrulea caracter dintr-un sir s cu *, se vor construi în tabela de siruri alte 4 siruri de caractere:

```
String s = "exemplu";
String t = s.substring(0, 3) + "*" + s.substring(4);
```

Practic, în exemplu de mai sus se vor crea în tabela de siruri, dacă nu există deja, sirurile "exe", "exe*", "plu" și "exe*plu"!

- Așadar, sunt situații în care se preferă utilizarea unui sir de caractere care să poată fi modificat direct, de exemplu, când se construiește dinamic un sir prin concatenarea mai multor siruri.
- Obiectele de tip `StringBuilder` sunt asemănătoare cu cele de tip `String`, însă nu mai sunt imutabile, deci pot fi direct modificate.
- Intern, obiectele de tip `StringBuilder` sunt alocate în zona de memorie heap și sunt tratate ca niște tablouri de caractere. Dimensiunea tabloului se modifică dinamic, pe măsură ce sirul este construit (initial, sirul are o lungime de 16 caractere):

```
StringBuilder sb = new StringBuilder();
sb.append("exemplu");
```

- Deoarece nu sunt imutabile, sirurile de tip `StringBuilder` nu sunt *thread-safe*, respectiv două sau mai multe fire de executare pot modifica simultan același sir, efectele fiind imprevizibile!

- Clasa `StringBuilder` conține, în afara unor metode asemănătoare celor din clasa `String` (de exemplu, metodele `indexOf`, `lastIndexOf` și `substring`), mai multe metode specifice:
 - modificarea lungimii șirului prin trunchiere sau extindere cu caracterul '\u0000':
 - `void setLength(int newLength)`
 - adăugarea la sfârșitul șirului a unor caractere obținute prin conversia unor valori de tip primitiv sau obiecte:
 - `StringBuilder append(boolean b)`
 - `StringBuilder append(char c)`
 - `StringBuilder append(double d)`
 - `StringBuilder append(float f)`
 - `StringBuilder append(int i)`
 - `StringBuilder append(long lng)`
 - `StringBuilder append(Object obj)`
 - `StringBuilder append(String str)`
 - `StringBuilder append(StringBuffer sb)`
 - inserarea în șir, începând cu poziția offset, a unor caractere obținute prin conversia unor valori de tip primitiv sau obiecte:
 - `StringBuilder insert(int offset, boolean b)`
 - `StringBuilder insert(int offset, char c)`
 - `StringBuilder insert(int offset, double d)`
 - `StringBuilder insert(int offset, float f)`
 - `StringBuilder insert(int offset, int i)`
 - `StringBuilder insert(int offset, long l)`
 - `StringBuilder insert(int offset, Object obj)`
 - `StringBuilder insert(int offset, String str)`
 - ștergerea unor caractere din șir:
 - `StringBuilder delete(int start, int end)`
 - `StringBuilder deleteCharAt(int index)`
 - înlocuirea unor caractere din șir:
 - `StringBuilder replace(int start, int end, String str)`
 - `void setCharAt(int index, char ch)`

CLASA `StringBuffer`

- Singura diferență dintre clasa `StringBuilder` și clasa `StringBuffer` constă în faptul că aceasta este thread-safe, adică metodele sale sunt sincronizate, fiind executate pe rând, sub excludere reciprocă! Din acest motiv, metodele sale sunt mai lente decât cele echivalente din clasa `StringBuilder`.

CLASE IMUTABILE

- Așa cum deja am menționat anterior, o clasă este imutabilă dacă nu mai putem modifica conținutul unei instanțe a sa (un obiect) după creare. Astfel, orice modificare a obiectului respectiv presupune crearea unui nou obiect și înlocuirea referinței sale cu referința noului obiect creat.
- În limbajul Java există mai multe clase imutabile predefinite: `String`, clasele înfășurătoare (`Integer`, `Float`, `Boolean` etc.), `BigInteger` etc.
- Principalele avantaje ale utilizării claselor imutabile sunt următoarele:
 - sunt implicit thread-safe (nu necesită sincronizare într-un mediu concurrent);
 - sunt ușor de proiectat, implementat, utilizat și testat;
 - sunt mai rapide decât clasele mutabile;
 - obiectele pot fi reutilizate folosind o tabelă de referințe și o metodă de tip factory pentru instantierea lor;
 - pot fi utilizate pe post de chei în structuri de date asociative (de exemplu, tabele de dispersie - `HashMap`);
 - programele care utilizează doar clase mutabile pot fi ușor adaptate pentru utilizarea într-un mediu distribuit.
- Singurul dezavantaj important al claselor imutabile îl constituie faptul că sunt create mai multe obiecte intermediare, respectiv câte unul pentru fiecare operație efectuată.
- De obicei, crearea unei clase imutabile trebuie să respecte următoarele reguli:
 1. clasa nu va permite rescrierea metodelor sale, fie declarând clasa de tip `final`, fie declarând constructorii ca fiind `private` și folosind metode de tip `factory` pentru a crea obiecte;
 2. toate câmpurile vor fi declarate ca fiind `final` (li se vor atribui valori o singură dată, printr-un constructor cu parametri) și `private` (nu li se pot modifica valorile direct);
 3. clasa nu va conține metode de tip `set` sau alte metode care pot modifica valorile câmpurilor;
 4. dacă există câmpuri care sunt referințe spre obiecte mutabile, se va împiedica modificarea acestora, astfel:
 - a. nu se vor folosi referințe spre obiecte externe, ci spre copii ale lor (se va folosi compoziția, ci nu agregarea!)

Exemplu: Fie o clasă Student care conține data membră facultate de tipul unei clase mutabile Facultate:

Greșit:

```
public Student(Facultate f, ...){
    this.facultate = f;           //agregare, deci obiectul
    .....                         //extern poate fi modificat!
}
```

Corect:

```
public Persoana(Date dn, ...){
    this.facultate = new Facultate(f);   //compoziție, deci obiectul
    .....                         //extern nu poate fi modificat
}
```

- b. nu se vor returna referințe spre câmpurile mutabile, ci se vor returna referințe spre copii ale lor:

Greșit:

```
public Facultate getFacultate(){
    return this.facultate;
}
```

Corect:

```
public Facultate getFacultate(){
    return new Facultate(facultate);
}
```

- În continuare, vom prezenta complet clasele Facultate și Student menționate anterior:

Clasa mutabilă Facultate:

```
public class Facultate {
    private String denumire;
    private String adresa;
    private String email;
    private String telefon;

    public Facultate(String denumire, String adresa, String email,
                     String telefon) {
        this.denumire = denumire;
        this.adresa = adresa;
        this.email = email;
        this.telefon = telefon;
    }
}
```

```

public Facultate(Facultate facultate) {
    this.denumire = facultate.denumire;
    this.adresa = facultate.adresa;
    this.email = facultate.email;
    this.telefon = facultate.telefon;
}

public String getDenumire() { return denumire; }

public void setDenumire(String denumire) { this.denumire = denumire; }

public String getAdresa() { return adresa; }

public void setAdresa(String adresa) { this.adresa = adresa; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

public String getTelefon() { return telefon; }

public void setTelefon(String telefon) { this.telefon = telefon; }
}

```

Clasa imutabilă Student:

```

//regula 1
public final class Student
{
    //regula 2
    private final String nume;
    private final Facultate facultate;
    private final int grupa;
    private final double medie;

    public Student(String nume, Facultate facultate, int grupa, double medie) {
        this.nume = nume;
        //regula 4a
        this.facultate = new Facultate(facultate);
        this.grupa = grupa;
        this.medie = medie;
    }

    public String getNume() {
        return nume;
    }
}

```

```

public Facultate getFacultate() {
    //regula 4b
    return new Facultate(facultate);
}

public int getGrupa() {
    return grupa;
}

public double getMedie() {
    return medie;
}
}

```

Observați faptul că în clasa Student nu sunt definite metode de tip set, deci este respectată și regula 3!

CLASE DE TIP ÎNREGISTRARE (RECORDS)

- În Java 15 au fost introduse *clasele de tip înregistrare* sau, pe scurt, *înregistrări (records)*.
- O *înregistrare* este o clasă imutabilă utilizată pentru a manipula o mulțime fixă de valori, denumite *componentele înregistrării*. De obicei, înregistrările sunt utilizate pentru încărcarea unor date dintr-o anumită sursă (de exemplu, un fișier sau o bază de date) și, eventual, transportarea acestora către o anumită destinație, folosind facilitățile limbajului Java pentru programarea în rețea.
- O *înregistrare* se declară într-un mod foarte concis, precizând doar tipul și numele componentelor sale în descriptorul *înregistrării*:

```
[modificatori de acces] record denumire (descriptor) {}
```

Exemplu:

```
public record Student(String nume, int grupa, double medie) {}
```

- Orice *înregistrare* este în mod implicit o clasă de tip **final** care extinde clasa **java.lang.Record**, deci o *înregistrare* nu poate fi abstractă, nu poate fi extinsă și nici nu poate extinde alte clase sau alte *înregistrări*. Totuși, o *înregistrare* poate implementa una sau mai multe interfețe.

- Pentru o înregistrare, compilatorul va genera automat o clasă de tip **final** având următoarele componente:
 - câte o dată membră privată și finală pentru fiecare componentă;
 - un constructor canonic public care va avea câte un parametru pentru fiecare componentă a înregistrării și va utiliza valoarea parametrului respectiv pentru a inițializa componenta corespunzătoare;
 - câte o metodă de tip **get** pentru fiecare componentă, denumirea unei metode fiind identică cu denumirea componentei asociate;
 - implementarea metodei **equals(Object)** din clasa **Object**;
 - implementarea metodei **hashCode()** din clasa **Object**;
 - implementarea metodei **toString()** din clasa **Object**.
- Metodele vor utiliza toate componentele înregistrării!**

Exemplu:

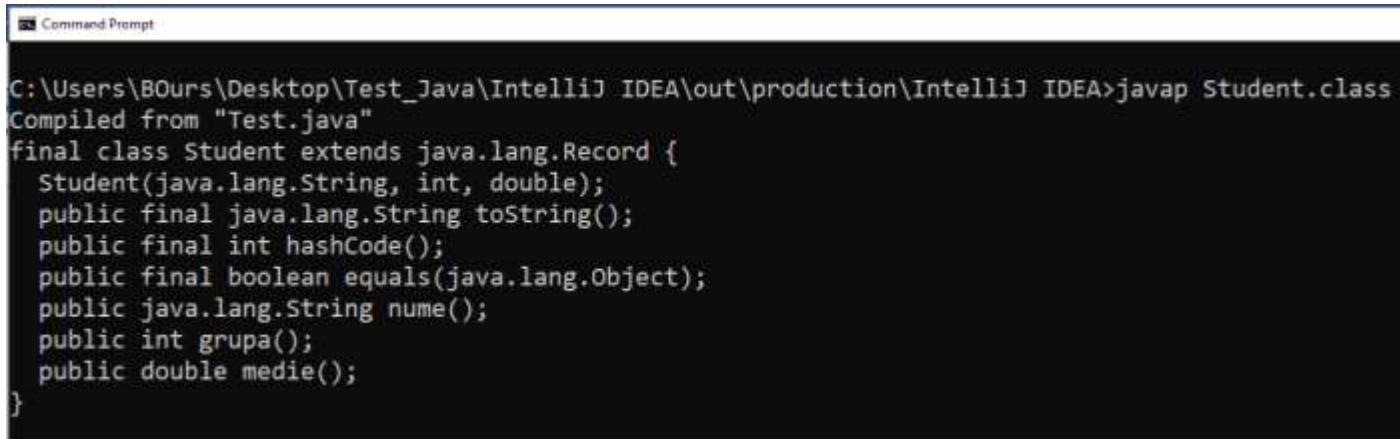
```
record Student(String nume, int grupa, double medie) {}

public class Test
{
    public static void main(String[] args)
    {
        Student stud_1 = new Student("Popescu Ion", 231, 9.50);
        Student stud_2 = new Student("Ionescu Ana", 232, 9.00);

        System.out.println(stud_1.nume() + ", " + stud_1.grupa() + ", "
                           + stud_1.medie());
        System.out.println(stud_2); //se va utiliza metoda toString()!
        System.out.println(stud_1.equals(stud_2));
    }
}
```



Folosind programul utilitar javap putem afișa metodele generate automat de compilatorul Java pentru înregistrarea Student definită mai sus:



```
C:\Users\BOurs\Desktop\Test_Java\IntelliJ IDEA\out\production\IntelliJ IDEA>javap Student.class
Compiled from "Test.java"
final class Student extends java.lang.Record {
    Student(java.lang.String, int, double);
    public final java.lang.String toString();
    public final int hashCode();
    public final boolean equals(java.lang.Object);
    public java.lang.String nume();
    public int grupa();
    public double medie();
}
}
```

- În cadrul unei înregistrări se pot adăuga constructori supraîncărcați, dar aceștia trebuie să apeleze explicit constructorul canonic:

```
record Student(String nume, int grupa, double medie) {
    public Student(String nume, int grupa) {
        this(nume, grupa, 0);
    }
}
```

- Atenție, nu se poate declara un constructor având parametrii pentru toate componentele înregistrării specificați în ordinea din descriptorul înregistrării, deoarece va intra în conflict cu constructorul canonic!
- Constructorul canonic poate fi redefinit, de obicei pentru a realiza prelucrări suplimentare sau validări ale componentelor înregistrării:

```
record Student(String nume, int grupa, double medie) {
    Student(String nume, int grupa, double medie) {
        if(medie < 0)
            medie = -medie;

        this.nume = nume.toUpperCase();
        this.grupa = grupa;
        this.medie = medie;
    }
}
```

- În cadrul unei înregistrări constructorul canonic poate fi accesat și prin intermediul unui *constructor compact*, adică un constructor public care nu are o listă a parametrilor, această fiind dedusă automat din descriptorul înregistrării:

```
record Student(String nume, int grupa, double medie) {
    Student {
        if(medie < 0)
            medie = -medie;
    }
}
```

În cazul modificării constructorului compact, se vor preciza doar prelucrări suplimentare pe care dorim să le efectuăm înaintea inițializării componentelor înregistrării, deoarece instrucțiunile necesare inițializării componentelor vor fi adăugate automat de compilator!

- O înregistrare este o clasă imutabilă de tip *shallowly immutable*, respectiv datele membre de tip referință vor fi copiate superficial (*shallow copy*), ci nu în adâncime (*deep copy*), ceea ce poate afecta imutabilitatea înregistrării respective! Pentru a evita acest aspect, trebuie să redefinim constructorul canonic și metodele de tip `get` corespunzătoare componentelor mutabile conform regulii 4 prezentate în secțiunea dedicată claselor imutabile (în exemplul următor am considerat clasa mutabilă `Facultate` definită anterior):

```
record Student(String nume, Facultate facultate, int grupa, double medie) {
    Student(String nume, Facultate facultate, int grupa, double medie) {
        this.nume = nume;
        //regula 4a
        this.facultate = new Facultate(facultate);
        this.grupa = grupa;
        this.medie = medie;
    }

    @Override
    public Facultate facultate() {
        //regula 4b
        return new Facultate(facultate);
    }
}
```

- Într-o înregistrare nu se pot declara date membre de instanță, dar se pot declara metode nestatice (dar nu se recomandă acest lucru, deoarece rolul principal al înregistrărilor este acela de a stoca date, ci nu acela de a le prelucra). De asemenea, se pot adăuga date, metode și blocuri de inițializare statice:

```
record Student(String nume, int grupa, double medie) {  
    private static String facultate = "Facultatea de Drept";  
  
    public static String getFacultate() {  
        return facultate;  
    }  
  
    public static void setFacultate(String facultate) {  
        Student.facultate = facultate;  
    }  
}
```

Observați faptul că datele membre statice nu mai sunt implicit de tip final și nu se mai generează automat metode de tip get pentru ele!

- În concluzie, înregistrările reprezintă o modalitate simplă de implementare a unor clase imutabile care să permită stocarea și, eventual, transportarea unor date .

INTERFETE

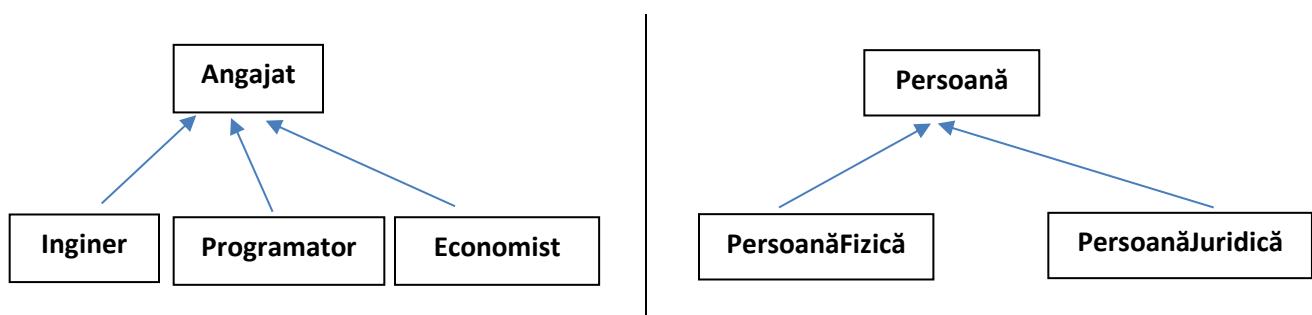
- O **interfață** este un tip de date abstract utilizat pentru a specifica un comportament pe care trebuie să-l implementeze o clasă.
- **Syntaxă:**

```
public interface numeInterfață{
    constante    //public, static și final
    metode abstracte (fără implementare) //public și abstract
    metode implicite (default) cu implementare
    metode statice cu implementare
    metode private cu implementare
}
```

- Datele membre sunt implicit **public, static și final**, deci sunt constante care trebuie să fie inițializate.
- Metodele membre sunt implicit **public**, iar cele fără implementare sunt implicit **abstract**.
- Interfețele definesc un set de operații (capabilități) comune mai multor clase care nu sunt înrudite (în sensul unei ierarhii de clase).

- **Exemple:**

Să presupunem că aveam următoarele ierarhii de clase:



După cum știți, una dintre operațiile des întâlnite în orice aplicație este cea de sortare (clasament, top etc.). În limbajul Java sunt definite metode generice (care nu țin cont de tipul elementelor) pentru a realiza sortarea unei structuri de obiecte, folosind un anumit criteriu de comparație (comparator). Astfel, într-o clasă se poate adăuga suplimentar un criteriu de comparație a obiectelor, sub forma unei metode (de exemplu, se poate realiza sortarea persoanelor juridice după cifra de afaceri, inginerii alfabetic după nume etc.). Cu alte cuvinte,

o interfață dedicată oferă o operație (capabilitate) de sortare, dar pentru a putea fi utilizată o clasă trebuie să specifică modalitatea de compararea a obiectelor.

Standardul Java oferă două interfețe pentru a compara obiectele în vederea sortării lor. Una dintre ele este interfața `java.lang.Comparable`, interfață care asigură o **sortare naturală** a obiectelor după un anumit criteriu.

```
public interface Comparable<Tip>{
    public int compareTo(Tip obiect);
}
```

Generalizând, într-o interfață se încapsulează un set de operații care nu sunt specifice unei anumite clase, ci, mai degrabă, au un caracter transversal (trans-ierarhic). Interfața în sine nu face parte dintr-o ierarhie de clase, ci este externă acesteia.

Un alt exemplu de operație pe care o poate realiza un obiect de tipul unei clase din ierarhiile de mai sus poate fi cel de plată online, folosind un cont bancar. Operația în sine poate fi realizată atât de către o categorie de angajați (de exemplu, programatori), cât și de persoane fizice sau juridice. Putem observa, din nou, cum o interfață care încapsulează operații specifice unei plăți online conține capabilități comune mai multor clase diferite conceptual (Angajat, PersoanăFizică etc.).

O interfață specifică unei plăți online poate să conțină următoarele operații:

- *autentificare* (pentru o persoană fizică se poate realiza folosind CNP-ul și o parolă, iar pentru o persoană juridică se poate folosi CUI-ul firmei și o parolă);
- *verificarea soldului curent*;
- *efectuarea unei plăți*.

```
public interface OperăriiContBancar{
    boolean autentificare();
    double soldCurent();
    void plată(double suma);
}
```

- Implementarea unei anumite interfețe de către o clasă oferă o anumită certificare clasei respective (clasa este capabilă să efectueze un anumit set de operații). Astfel, o interfață poate fi privită ca o operație de tip CAN_DO.
- În concluzie, interfața poate fi văzută ca un serviciu (API) care poate fi implementat de orice clasă. Clasa își anunță intenția de a implementa serviciul respectiv, într-o maniera specifică, realizând-se astfel un contract între clasă și interfață, cu o clauză clară: clasa trebuie să implementeze metodele abstracte din interfață.

- **Implementarea unei interfețe** se realizează utilizând următoarea sintaxă:

```
[modificatori] class numeClasa implements numeInterfață_1,
                                         numeInterfață_2,..., numeInterfață_n
```

- Se poate observa cum o clasă poate să implementeze mai multe interfețe în scopul de a dobândi mai multe capabilități. De exemplu, pentru o interfață grafică trebuie să tratăm atât evenimente generate de mouse, cât și evenimente generate de taste, deci vom implementa două interfețe: `MouseListener` și `KeyListener`.
- Revenind la exemplul anterior, clasa `Inginer` implementează interfața `Comparable`, oferind un criteriu de comparație (sortare alfabetică după nume):

```
class Inginer implements Comparable<Inginer>{
    private String nume;
    .....
    public int compareTo(Inginer ob) {
        return this.nume.compareTo(ob.nume);
    }
}
```

- Pentru un tablou cu obiecte de tip `Inginer` se poate apela metoda statică `sort` din clasa utilitară `Arrays`:

```
Inginer tab[] = new Inginer[10];
.....
Arrays.sort(tab); //sortare naturală, metoda sort nu mai are nevoie de un alt
                  //argument pentru a specifica criteriul de sortare, ci se va utiliza
                  //împlicit metoda compareTo implementată în clasa Inginer
```

- Clasa `Inginer` implementează interfața `OperatiiContBancar`, oferind implementări pentru toate cele trei metode abstracte:

```
class Inginer implements OperatiiContBancar{
    private String contBancar;
    .....
    public boolean autentificare() {
        //conectare la server-ul băncii pe baza CNP-ului și a unei parole
    }
    public double soldCurent() {
        //interrogarea contului folosind API-ul server-ului băncii
    }
    void plată(double suma) {
        //accesarea contului în scopul efectuării unei plăti folosind API-ul server-ului băncii
    }
}
```

- Clasa PersoanăJuridică implementează interfața OperațiiContBancar, oferind implementări pentru toate cele trei metode abstracte:

```
class PersoanăJuridică implements OperațiiContBancar{
    private String contBancar;
    .....
    public boolean autentificare(){
        // conectare la server-ul băncii utilizând CUI-ul firmei și o parolă
    }
    double soldCurent(){
        // interogarea contului folosind API-ul server-ului băncii
    }
    void plată(double suma){
        //accesarea contului în scopul efectuării plășii folosind API-ul server-ului băncii
    }
}
```

➤ **Observații:**

- Dacă o clasă implementează două interfețe care conțin metode abstracte cu aceeași denumire, atunci apare un conflict de nume care induce următoarele situații:
 - dacă metodele au signature diferite, clasa trebuie să implementeze ambele metode;
 - dacă metodele au aceeași signature și același tip pentru valoarea returnată, clasa implementează o singură metodă;
 - dacă metodele au aceeași signature, dar tipurile valorilor returnate diferă, atunci implementarea nu va fi posibilă și se va obține o eroare de compilare.
- În cazul câmpurilor cu același nume, conflictele se pot rezolva prefixând numele unui câmp cu numele interfeței (chiar dacă au tipuri diferite).
- **O interfață nu se poate instanția, însă un obiect de tipul clasei care o implementează poate fi accesat printr-o referință de tipul interfeței. În acest caz, comportamentul obiectului este redus la cel oferit de interfață, alături de cel oferit de clasa Object:**

```
OperațiiContBancar p = new Inginer();
System.out.println("Sold curent: " + p.soldCurent());
```

➤ **În concluzie, în limbajul Java un obiect poate fi referit astfel:**

1. printr-o referință de tipul clasei sale => se pot accesa toate metodele publice încapsulate în clasă, alături de cele moștenite din clasa Object;
2. printr-o referință de tipul superclasei (polimorfism) => se pot accesa toate metodele moștenite din superclasă, cele redefinite în subclasă, alături de cele moștenite din clasa Object;
3. printr-o referință de tipul unei interfețe pe care o implementează => se pot accesa metodele implementate din interfață, alături de cele moștenite din clasa Object.

➤ Extinderea interfețelor

Să presupunem faptul că o interfață ce conține doar metode abstracte este implementată de mai multe clase C1, C2, ..., Cn etc. Fiecare clasă oferă o implementare pentru toate metodele abstracte din interfață implementată, astfel clasele C1, C2, ... Cn pot fi instanțiate. Ulterior, dezvoltatorul dorește să mai introducă și alte funcționalități în interfață, respectiv alte metode abstracte. În acest caz, clasele C1, C2, ... Cn devin abstracte și nu mai pot fi instanțiate!!! O soluție pentru a elimina acest neajuns, specifică până în versiunea Java 7, este aceea de a extinde interfața inițială și de a adăuga noile metode abstracte în subinterfața sa.

Sintaxa pentru extinderea interfețelor:

```
interface subInterfata extends superInterfata1, superInterfata2,  
....superInterfataN
```

Exemplu: Să presupunem faptul ca dorim să modificăm interfața OperăriiContBancar prin includerea unui nou serviciu, respectiv a metodei void sendSMS (String message) pentru a trimite un mesaj informativ de tip sms unui client ce folosește serviciul OperăriiContBancar. Pentru ca o serie de clase care implementează interfața OperăriiContBancar să nu fie afectate, se poate defini o subinterfață OperăriiContBancarSMS a superinterfeței OperăriiContBancar.

```
interface OperariiContBancarSMS extends OperariiContBancar {  
    void sendSMS (String message);  
}
```

Utilitatea interfețelor

1. Definirea unor funcționalități ale unei clase

Așa cum am văzut mai sus, cu ajutorul interfețelor se pot defini funcționalități comune unor clase independente, care nu se află în aceeași ierarhie, fără a forța o legătură între ele (capabilități trans-ierarhice).

2. Definirea unor grupuri de constante

O interfață poate fi utilizată și pentru definirea unor grupuri de constante. De exemplu, mai jos este definită o interfață care încapsulează o serie de constante matematice, utilizate în diferite expresii și formule de calcul. Clasa TriunghiEchilateral implementează interfața ConstanteMatematice în scopul de a folosi constanta SQRT_3 (o aproximare a valorii $\sqrt{3}$) în formula de calcul a ariei unui triunghi echilateral:

```

public interface ConstanteMatematice{
    double PI = 3.14159265358979323846;
    double SQRT_2 = 1.41421356237;
    double SQRT_3 = 1.73205080757;
    double LN_2 = 0.69314718056;
}

class TriunghiEchilateral implements ConstanteMatematice{
    double latura;

    public TriunghiEchilateral(double x) {
        latura = x;
    }

    double Aria() {
        return latura*latura*ConstanteMatematice.SQRT_3/4;
    }
}

```

Totuși, metoda poate fi ineficientă, deoarece o clasă s-ar putea să folosească doar o constantă din interfață implementată sau un set redus de constante. Prin implementarea interfeței, clasa preia în semnătura sa toate constantele, ci nu doar pe acelea pe care le folosește. În acest sens, o metodă mai eficientă de încapsulare a unor constante este dată de utilizarea unei enumerări, concept pe care va fi prezentat într-un curs ulterior.

3. Implementarea mecanismului de callback

O altă utilitate importantă a unei interfețe o constituie posibilitatea de a transmite o metodă ca argument al unei alte metode (**callback**).

În limbajul Java nu putem transmite ca argument al unei funcții/metode un pointer către o altă metodă, aşa cum este posibil în limbajele C/C++. Totuși, această facilitate, care este foarte utilă în diverse aplicații (de exemplu, în programarea generică), se poate realiza în limbajul Java folosind interfețele.

Implementarea mecanismului de callback în limbajul Java se realizează, de obicei, astfel:

1. se definește o interfață care încapsulează metoda generică sub forma unei metode abstracte;
2. se definește o clasă care conține o metodă pentru realizarea prelucrării generice dorite (metoda primește ca parametru o referință de tipul interfeței pentru a accesa metoda generică);
3. se definesc clase care implementează interfața, respectiv clase care conțin implementările dorite pentru metoda generică din interfață;

4. se realizează prelucrările dorite apelând metoda din clasa definită la pasul 2 în care parametrul de tipul referinței la interfață se înlocuiește cu instanțe ale claselor definite la pasul 3.

Exemplul 1: Să presupunem faptul că dorim să calculăm următoarele 3 sume:

$$\begin{aligned}S_1 &= 1 + 2 + \dots + n \\S_2 &= 1^2 + 2^2 + \dots + n^2 \\S_3 &= [\text{tg}(1)] + [\text{tg}(2)] + \dots + [\text{tg}(n)],\end{aligned}$$

unde prin $[x]$ am notat partea întreagă a numărului real x .

Desigur, o soluție posibilă constă în implementarea a trei metode diferite care să returneze fiecare câte o sumă. Totuși, o soluție mai elegantă se poate implementa observând faptul că toate cele 3 sume sunt de forma următoare:

$$S_k = \sum_{i=1}^n f_k(i)$$

unde termenii generali sunt $f_1(i) = i$, $f_2(i) = i^2$ și $f_3(i) = [\text{tg}(i)]$.

Astfel, se poate implementa o metodă generică pentru calculul unei sume de această formă care să utilizeze mecanismul de callback pentru a primi ca parametru o referință spre termenul general al sumei.

Urmând pașii amintiți mai sus, se definește mai întâi o interfață care încapsulează funcția generică:

```
public interface FuncțieGenerică{
    int funcție(int x);
}
```

Într-o clasă utilitară, definim o metodă care să calculeze suma celor n termeni generici:

```
public class Suma{
    private Suma(){ //într-o clasă utilitară constructorul este privat!
    }

    public static int CalculeazăSuma(FuncțieGenerică fg , int n){
        int s = 0;

        for(int i = 1; i <= n; i++)
            s = s + fg.funcție(i);

        return s;
    }
}
```

}

Ulterior, definim clase care implementează interfața respectivă, oferind implementări concrete ale funcției generice:

```
public class TermenGeneral_1 implements FuncțieGenerică{
    @Override
    public int funcție(int x){
        return x;
    }
}

public class TermenGeneral_2 implements FuncțieGenerică{
    @Override
    public int funcție(int x){
        return x * x;
    }
}
```

La apel, metoda `CalculeazăSumă` va primi o referință de tipul interfeței, dar spre un obiect de tipul clasei care implementează interfața:

```
public class Test_callback {
    public static void main(String[] args) {
        FuncțieGenerică tgen_1 = new TermenGeneral_1();
        int S_1 = Suma.CalculeazăSumă(tgen_1, 10);
        System.out.println("Suma 1: " + S_1);

        //putem utiliza direct un obiect anonim
        int S_2 = Suma.CalculeazăSumă(new TermenGeneral_2(), 10);
        System.out.println("Suma 2: " + S_2);

        //putem utiliza o clasă anonimă
        int S_3 = Suma.CalculeazăSumă(new FuncțieGenerică() {
            public int funcție(int x) {
                return (int) Math.tan(x);
            }
        }, 10);
        System.out.println("Suma 3: " + S_3);
    }
}
```

Exemplul 2: Mai sus, am văzut cum sortarea unor obiecte se poate realiza implementând interfața `java.lang.Comparable` în cadrul clasei respective, obținând astfel un singur criteriu de comparație care asigură sortarea naturală a obiectelor. Dacă aplicația necesită mai multe sortări, bazate pe criterii de comparație diferite, atunci se poate utiliza interfața `java.lang.Comparator` și mecanismul de callback.

De exemplu, pentru a sorta descrescător după vârstă obiecte de tip `Inginer` memorate într-un tablou `t`, vom defini următorul comparator:

```
public class ComparatorVârste implements Comparator<Inginer>{  
    public int compare (Inginer ing1, Inginer ing2) {  
        return ing2.getVârsta() - ing1.getVârsta();  
    }  
}
```

La apel, metoda statică `sort` a clasei utilitare `Arrays` va primi ca parametru un obiect al clasei `ComparatorVârste` sub forma unei referințe de tipul interfeței `Comparator`:

```
Arrays.sort(t, new ComparatorVârste());
```

INTERFETE MARKER

- Interfețele marker sunt interfețe care nu conțin nicio constantă și nicio metodă, ci doar anunță mașina virtuală Java faptul că se dorește asigurarea unei anumite funcționalități la rularea programului, iar mașina virtuală va fi responsabilă de implementarea funcționalității respective. Practic, interfețele marker au rolul de a asocia metadate unei clase, pe care mașina virtuală să le folosească la rulare într-un anumit scop.
- În standardul Java sunt definite mai multe interfețe marker, precum `java.io.Serializable` care este utilizată pentru a asigura salvarea obiectelor sub forma unui sir de octeți într-un fișier binar sau `java.lang.Cloneable` care asigură clonarea unui obiect.

Interfața `java.lang.Cloneable`

O clasă care implementează interfața `Clonable` permite apelul metodei `Object.clone()` pentru instanțele sale, în scopul de a realiza o copie a lor. Interfața în sine nu conține nicio metodă, fiind interfață marker, ci doar anunță mașina virtuală Java faptul că instanțele clasei care o implementează au funcționalitatea de clonare.

Prin convenție, o clasă care implementează interfața `Cloneable`, redefineste metoda `Object.clone()` (care are acces protejat) printr-o metodă cu acces public.

Exemplu: Considerăm clasa `Angajat`

```
public class Angajat {
    private String nume;
    private int varsta;
    private double salariu;

    public Angajat(String nume, int varsta, double salariu) {
        this.nume = nume;
        this.varsta = varsta;
        this.salariu = salariu;
    }

    //metode get, set, toString()

    //redefinirea metodei clone din clasa Object
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

```

public class Clona {
    public static void main(String[] args) throws
        CloneNotSupportedException {

        Angajat a1 = new Angajat("Matei", 23, 4675.67);

        Angajat a2 = (Angajat)a1.clone();
    }
}

```

Apelul metodei `clone()` pentru obiectul `a1` conduce, în momentul executării, la apariția excepției `java.lang.CloneNotSupportedException`, deoarece clasa `Angajat` nu implementează interfața marker `Cloneable`!!!

```

public class Angajat implements Cloneable{
.....
}

```

Observație: Clonarea unui obiect presupune, în sine, copierea acestuia la o alta adresă HEAP alocată pentru obiectul destinație. Însă, în cazul în care obiectul conține o referință către alt obiect (aggregare/compoziție), redefinirea metodei `clone()` nu alocă implicit o nouă adresă HEAP pentru obiectul încapsulat. Să presupunem faptul că se specifică pentru fiecare `Angajat` și departamentul în care acesta activează. Considerăm astfel clasa `Departament`:

```

public class Departament {
    private int id;
    private String denumire;

    public Departament(int id, String denumire) {
        this.id = id;
        this.denumire = denumire;
    }
    //metode set, get și toString()
}

```

Modificăm clasa `Angajat`, adăugând câmpul `departament`:

```

public class Angajat implements Cloneable{
    private String nume;
    private int varsta;
    private double salar;
    private Departament departament;

    public Angajat(String nume, int varsta, double salar,
                    Departament departament) {
        this.nume = nume;
        this.varsta = varsta;
    }
}

```

```

        this.salariu = salariu;
        this.departament = departament;
    }
    //metode get, set

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class TestClona {
    public static void main(String[] args)
        throws CloneNotSupportedException {
        Departament departament = new Departament(1, "HR");
        Angajat a1 = new Angajat("Matei", 23, 4675.67, departament);

        Angajat a2 = (Angajat)a1.clone();

        a2.getDepartament().setDenumire("Finante");

        System.out.println(a1.getDepartament() + " " +
                           a2.getDepartament());
    }
}

```

Clonarea obiectului a1 s-a realizat cu succes, însă setarea câmpului departament pentru obiectul a2 conduce și la modificarea câmpului departament pentru obiectul a1, deoarece metoda `clone()` din clasa `Angajat` realizează doar o clonă a referinței de tip `Departament`! S-a realizat astfel ceea ce poartă denumirea de *shallow cloning*. Pentru a evita ca ambele câmpuri de tip `Departament` să aibă aceeași referință, se creează o clonă care este independentă de obiectul original, astfel încât orice modificare a clонei să nu conducă și la modificarea obiectului original. Practic, se redefineste metoda `clone()` și în clasa `Departament`:

```

public class Departament implements Cloneable{
    .....
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Angajat implements Cloneable{
    .....
    @Override
    public Object clone() throws CloneNotSupportedException {
        .....

```

```

Angajat clona = (Angajat)super.clone();

clona.setDepartament((Departament)clona.getDepartament().clone());
return clona;
}
}

```

CLASE ADAPTOR

- O interfață poate să conțină multe metode abstracte. De exemplu, interfața `MouseListener` conține 8 metode asociate unor evenimente produse de mouse (`mousePressed()`, `mouseReleased()` etc.). O clasă care implementează o astfel de interfață, evident, trebuie să ofere implementare pentru toate metodele abstracte. Totuși, de cele mai mult ori, în practică o clasă va folosi un set restrâns de metode dintre cele specificate în interfață. De exemplu, din interfața `MouseListener` se folosește, de obicei, metoda asociată evenimentului `mouseClicked()`.
- O soluție pentru această problemă o constituie definirea unei *clase adaptor*, respectiv o clasă care să implementeze minimal (cod vid) toate metodele din interfață. Astfel, dacă o clasă dorește să implementeze doar câteva metode din interfață, poate să prefere extinderea clasei adaptor, redefinind doar metodele necesare.

ÎMBUNĂTĂȚIRI ADUSE INTERFEȚELOR ÎN JAVA 8 ȘI JAVA 9

- Un dezavantaj major al interfețelor specifice versiunilor anterioare Java 8 îl constituie faptul că modificarea unei interfețe necesită modificarea tuturor claselor care o implementează. O soluție posibilă ar fi aceea de a extinde interfața respectivă și de a încapsula în sub-interfață metodele suplimentare. Totuși, această soluție nu conduce la o utilizare imediată sau implicită a interfeței nou create. Astfel, pentru a elmina acest neajuns, începând cu versiunea Java 8 o interfață poate să conțină și metode cu implementări implicite (*default*) sau metode statice cu implementare.

```

interface numeInterfață{
    .....
    default tipRezultat metodăImplicită(...) {
        //implementare implicită
    }

    static tipRezultat metodăStatică(...) {
        //implementare
    }
}

```

- În acest fel, o clasă care implementează interfața preia implicit implementările metodelor default. Dacă este necesar, o metodă default poate fi redefinită într-o clasă care implementează interfața respectivă.
- În plus, o metodă dintr-o interfață poate fi și statică, dacă nu dorim ca metoda respectivă să fie preluată de către clasă. Practic, metoda va aparține strict interfeței, putând fi invocată doar în cadrul unei clase care implementează interfața sau, direct, prin numele interfeței. De regulă, o metodă statică este una de tip utilitar.

Exemplu: Considerăm interfața `InterfațăAfișareŞir` în care definim o metodă default afișezăŞir pentru afișarea unui sir de caractere sau a unui mesaj corespunzător dacă sirul este vid. Verificarea faptului că un sir este vid se realizează folosind metoda statică (utilitară) `esteŞirVid`, deoarece nu considerăm necesar ca această metodă să fie preluată în clasele care vor implementa interfața.

```
public interface InterfațăAfișareŞir {
    default void afișezăŞir(String str) {
        if (!esteŞirVid(str))
            System.out.println("Sirul: " + str);
        else
            System.out.println("Sirul este vid!");
    }
    static boolean esteŞirVid(String str) {
        System.out.println("Metoda esteŞirVid din interfață!");
        return str == null ? true : (str.equals("") ? true : false);
    }
}

public class ClasaAfișareŞir implements InterfațăAfișareŞir {
    //Override -> nu se poate utiliza adnotarea deoarece metoda este statică și nu se preia din interfață
    public static boolean esteŞirVid(String str) {
        System.out.println("Metoda esteŞirVid din clasă!");
        return str.length() == 0;
    }
}

public class Test {
    public static void main(String args[]) {
        ClasaAfișareŞir c = new ClasaAfișareŞir();
        c.afișezăŞir("exemplu");
        c.afișezăŞir(null);

        //System.out.println(InterfațăAfișareŞir.esteŞirVid(null));
        //System.out.println(ClasaAfișareŞir.esteŞirVid(null));
    }
}
```

Dacă vom elmina comentariile din metoda `main` și vom rula programul, va apărea o eroare în momentul apelării metodei `esteSirViD` din clasă. De ce?

➤ Extinderea interfețelor care conțin metode default

În momentul extinderii unei interfețe care conține o metodă default pot să apară următoarele situații:

- sub-interfață nu are nicio metodă cu același nume => sub-interfață va moșteni metoda default din super-interfață;
- sub-interfață conține o metodă abstractă cu același nume => metoda redevine abstractă în sub-interfață (i.e., metoda nu mai este default);
- sub-interfață redefinește metoda default tot printr-o metodă default;
- sub-interfață extinde două super-interfețe care conțin două metode default cu aceeași semnatură și același tip returnat => sub-interfață trebuie să redefinească metoda (nu neapărat tot de tip default) și, eventual, poate să apeleze în implementarea sa metodele din super-interfețe folosind sintaxa `SuperInterfata.super.metoda()`;
- sub-interfață extinde două super-interfețe care conțin două metode default cu aceeași semnatură și tipuri returnate diferite => moștenirea nu este posibilă.

➤ Reguli pentru extinderea interfețelor și implementarea lor (problema rombului)

1. Clasele au prioritate mai mare decât interfețele (dacă o metodă default dintr-o interfață este rescrisă într-o clasă, atunci se va apela metoda din clasa respectivă);
2. Interfețele "specializate" (sub-interfețele) au prioritate mai mare decât interfețele "generale" (super-interfețe);
3. Nu există regula 3! Dacă în urma aplicării regulilor 1 și 2 nu există o singură interfață câștigătoare, atunci clasele trebuie să rezolve conflictul de nume explicit, respectiv vor implementa metoda default, eventual apelând una dintre metodele default printr-o construcție sintactică de forma `Interfață.super.metoda()`.

Exemplu: Considerăm două interfețe `Poet` și `Scriitor`:

```
interface Poet {
    default void scrie(){
        System.out.println("Metoda default din interfața Poet!");
    }
}

interface Scriitor {
    default void scrie(){
        System.out.println("Metoda default din interfața Scriitor!");
    }
}
```

```

class Multitalent implements Poet, Scriitor {
    public static void main(String args[]){
        Multitalent autor = new Multitalent();
        autor.scrie();
    }
}

```

Apelul metodei `scrie()` pentru obiectul `autor` conduce la apariția unei erori la compilare, respectiv:

```

class Multitalent inherits unrelated defaults for scrie() from types
Poet and Scriitor

```

Pentru a elmina ambiguitatea cauzată de implementarea celor două interfețe care conțin metode cu semnatura identică, clasa trebuie să redefinăască metoda respectivă:

```

public class Multitalent implements Poet, Writer {
    @Override
    public void scrie()
    {
        System.out.println("Metoda din clasa Multitalent!");
    }
}

```

➤ În Java 9 a fost adăugată posibilitatea ca o interfață să conțină metode private, statice sau nu. Regulile de definire sunt următoarele:

- metodele private trebuie să fie definite complet (să nu fie abstracte);
- metodele private pot fi statice, dar nu pot fi default.
- Principala utilitate a metodelor private este următoarea: dacă mai multe metode default conțin o porțiune de cod comun, atunci aceasta poate fi mutată într-o metodă privată și apoi apelată din metodele default. Astfel, o metodă privată nu este accesibilă din afara interfeței (chiar dacă este statică), nu este necesară implementarea sa în clasele care vor implementa interfața și nici nu va fi preluată implicit (deoarece nu este default).

Exemplu: Considerăm următoarea implementare specifică versiunii Java 8:

```

public interface Calculator {
    default void calculComplex_1(...) {
        Cod comun
        Cod specific 1
    }
    default void calculComplex_2(...) {
        Cod comun
        Cod specific 2
    }
}

```

Un dezavantaj evident este faptul că o secvență de cod este repetată în mai multe metode. O variantă de rezolvare ar putea fi încapsularea codului comun într-o metoda default:

```
public interface Calculator {
    default void calculComplex_1(...) {
        codComun(...);
        Cod specific 1
    }

    default void calculComplex_2(...) {
        codComun(...);
        Cod specific 2
    }

    default void codComun(...) {
        Cod comun
    }
}
```

Totuși, în acest caz metoda default care încapsulează codul comun va fi moștenită de către toate clasele care vor implementa interfața respectivă. Soluția oferită în Java 9 constă în posibilitatea de a încapsula codul comun într-o metoda privată (statică sau nu). Astfel, metoda privată nu va fi moștenită de către clasele care implementează interfața:

```
public interface Calculator{
    default void calculComplex_1(...) {
        codComun(...);
        Cod specific 1
    }

    default void calculComplex_2(...) {
        codComun(...);
        Cod specific 2
    }

    private void codComun(...) {
        Cod comun
    }
}
```

CONTROLUL MOȘTENIRII/IMPLEMENTĂRII ÎN JAVA

În versiunile anterioare versiunii Java 17, lansată în septembrie 2021, nu exista niciun mecanism prin care să poată fi controlată într-un mod detaliat extinderea unei clase. Astfel, extinderea unei clase putea fi controlată fie prin intermediul specificatorului **final** (restricționând-o astfel complet), fie declarând clasa accesibilă doar la nivel de pachet (i.e., specificatorul implicit de acces), ceea ce restricționează extinderea sa la subclase aflate în același pachet (dar restricționează și accesarea sa din exteriorul pachetului!).

În versiunea Java 17 a fost introdus conceptul de *clasă/interfață sealed* care permite un control detaliat al moștenirii prin precizarea explicită a subclaserelor care pot extinde o superclasă sau a claselor care pot implementa o interfață.

O clasă *sealed* se declară astfel:

```
[specificatori] sealed class Clasă permits Subclase {  
    .....  
}
```

Dacă superclasa extinde o clasă și/sau implementează anumite interfețe, atunci cuvântul **permits** și lista subclaserelor care pot să implementeze clasa respectivă se vor scrie la sfârșitul antetului său, aşa cum se poate observa din următorul exemplu:

```
public sealed class Angajat extends Persoana implements Comparable  
    permits Economist, Paznic, Inginer {  
    .....  
}
```

Declarările unei clase *sealed* și ale subclaserelor permise trebuie să respecte următoarele reguli:

- clasa *sealed* și subclasele permise trebuie să facă parte din același modul sau, dacă sunt declarate într-un modul anonim, din același pachet;
- fiecare subclasă permisă trebuie să extindă direct clasa *sealed*;
- fiecare subclasă permisă trebuie să specifice în mod explicit modul în care va continua controlul moștenirii inițiat de superclasa sa, folosind exact unul dintre următorii modificatori:
 - **final**: subclasa respectivă nu mai poate fi extinsă;
 - **sealed**: subclasa respectivă poate fi extinsă doar în mod controlat (i.e., doar de subclasele pe care le permite explicit);

- **non-sealed**: subclasa respectivă poate fi extinsă fără nicio restricție (i.e., de orice altă clasă), deci o clasă sealed nu poate obliga subclasele sale să restricționeze ulterior moștenirea.

Pentru clasa `Angajat` din exemplul se mai sus, o variantă de declarare a subclaserelor poate fi următoarea:

```
public final class Paznic extends Angajat {
    .....
}

public non-sealed class Economist extends Angajat {
    .....
}

public sealed class Inginer extends Angajat
    permits InginerElectronist, InginerMecanic {
    .....
}
```

O clasă sealed nu poate conține în lista subclaserelor permise clase de tip record, deoarece acestea nu pot extinde o altă clasă (i.e., orice clasă de tip record extinde în mod implicit clasa `java.lang.Record`), iar unei clase de tip record nu îi putem aplica modificatorul `sealed` deoarece clasele de acest tip nu pot fi extinse (i.e., sunt implicit de tip `final`).

În cazul unei interfețe, folosind modificatorul `sealed`, putem specifica subinterfețele care o pot extinde sau clasele care o pot implementa, astfel:

```
[public] sealed interface Interfață permits Subinterfețe, Clase {
    .....
}
```

O subinterfață care extinde o interfață sealed trebuie să respecte reguli asemănătoare celor precizate în cazul claselor sealed, cu observația că unei interfață îi putem aplica doar modificatorii `sealed` și `non-sealed`. În lista claselor care pot implementa o interfață putem preciza și clase de tip record, cu observația că acestea vor fi implicit de tip `final`, deci nu putem să ele aplicăm modificatorii `sealed` și `non-sealed`.

În concluzie, folosind mecanismul de control al moștenirii/implementării, un programator poate crea ierarhii care modeleză într-un mod complet și sigur un anumit concept.

ENUMERĂRI

O **enumerare** este un tip special de clasă care poate încapsula o serie de constante. Enumerările au fost adăugate în limbajul Java începând cu versiunea Java 5.

O enumerare se declară folosind următoarea sintaxă:

```
public enum DenumireaEnumerării
{
    instanțele enumerării (constantele propriu-zise)
    [câmpuri private care rețin valorile unei constante]
    [constructor privat care valorile asociate unei constante]
    [metode care furnizează valorile asociate unei constante]
}
```

De exemplu, putem să definim o enumerare care să conțină constante asociate celor 4 puncte cardinale astfel:

```
public enum PuncteCardinale{
    NORD, EST, SUD, VEST;
}
```

Observați faptul că numele constantelor sunt scrise cu litere mari, respectând astfel o regulă de bună practică în orice limbaj de programare!

Accesarea unei constante se realizează într-o manieră statică, prin numele său:

```
PuncteCardinale p = PuncteCardinale.NORD;
```

Observați faptul că un obiect de tip enumerare nu trebuie instantiat folosind operatorul new!

Dacă dorim să asociem anumite valori constantelor dintr-o enumerare, atunci trebuie să declarăm în enumerarea respectivă date membre corespunzătoare (de obicei, de tip final și private), un constructor privat care să le initializeze și, eventual, metode publice de tip get:

```
public enum PuncteCardinale{
    NORD(1, 'N'), EST(2, 'E'), SUD(3, 'S'), VEST(4, 'V');

    private final int valoare;
    private final char simbol;

    private PuncteCardinale(int valoare, char simbol) {
        this.valoare = valoare;
        this.simbol = simbol;
    }
}
```

```

public int getValoare() {
    return valoare;
}

public char getSimbol() {
    return simbol;
}
}

```

Orice enumerare este implicit o clasă de tip final care extinde clasa `java.lang.Enum`, deci o enumerare nu mai poate extinde nicio altă clasă și nici nu mai poate fi extinsă!

Clasa `java.lang.Enum` conține o serie de metode care vor fi moștenite implicit de orice enumerare:

- **public final String name()**: furnizează numele unei constante sub forma unui sir de caractere;
- **public final int ordinal()**: furnizează numărul de ordine al unei constante în cadrul enumerării.

Pentru orice enumerare, compilatorul va genera automat o metodă statică denumită `values()` care va furniza toate constantele din enumerarea respectivă sub forma unui tablou unidimensional:

```

for(PuncteCardinale pc: PuncteCardinale.values())
    System.out.println("Constanta "+pc.name()+" are indexul
"+pc.ordinal());

```

De asemenea, clasa `java.lang.Enum` conține și metode redefinite din clasa `Object`:

- **public final boolean equals(Object other)**
- **public final int hashCode()**
- **public String toString()**

Metoda `toString()` poate fi redefinită pentru a furniza o descriere mai amănunțită a unei constante:

```

@Override
public String toString() {
    return "Constanta " + name() + " are valoarea " + valoare +
        " si simbolul " + simbol;
}

```

O enumerare poate să încapsuleze metode de instanță (vezi metodele de tip `get` din exemplul de mai sus) și metode statice. De exemplu, următoarea metodă statică va furniza constanta având asociată o valoare `x` sau `null` dacă nu există nicio constantă cu valoarea respectivă:

```

public static PuncteCardinale getPunctCardinalValoare(int x) {
    for (PuncteCardinale pc : PuncteCardinale.values())
        if (x == pc.getValoare())
            return pc;
    return null;
}

```

O enumerare poate să încapsuleze o metodă abstractă, caz în care fiecare instanță a enumerării (i.e., fiecare constantă) trebuie să implementeze metoda respectivă. De exemplu, metoda abstractă `getSuccesor()` va furniza succesorul fiecărui punct cardinal, în sensul acelor de ceasornic:

```

public enum PuncteCardinale {
    NORD(1, 'N') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.EST;
        }
    },
    EST(2, 'E') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.SUD;
        }
    },
    SUD(3, 'S') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.VEST;
        }
    },
    VEST(4, 'V') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.NORD;
        }
    };
}

private final int valoare;
private final char simbol;

private PuncteCardinale(int valoare, char simbol) {
    this.valoare = valoare;
    this.simbol = simbol;
}

public int getValoare() {
    return valoare;
}

public char getSimbol() {
    return simbol;
}

public abstract PuncteCardinale getSuccesor();
}

```

În încheiere, precizăm faptul că enumerările sunt foarte utile pentru a modela liste finite de constante (e.g., opțiunile dintr-un meniu, stările în care se poate afla o mașină sau un robot, denumirile monedelor acceptate pentru plată de către un magazin etc.). De asemenea, o enumerare cu o singură constantă reprezintă o modalitate foarte simplă de implementare a unei clase singleton ([Java Singletons Using Enum - DZone Java](#)).

EXCEPTII

O **excepție** este un eveniment care întrerupe executarea normală a unui program. Exemple de excepții: împărțirea unui număr întreg la 0, încercarea de deschidere a unui fișier inexistent, accesarea unui element inexistent într-un tablou, procesarea unor date de intrare incorecte etc.

Tratarea excepțiilor devine stringentă în aplicații complexe, formate din mai multe module (de exemplu, o interfață grafică care implică apelurile unor metode din alte clase). De regulă, rularea unui program presupune o succesiune de apeluri de metode, spre exemplu, metoda `main()` apelează metoda `f()` a unui obiect, aceasta la rândul său apelează o metodă `g()` a altui obiect și.a.m.d. astfel încât, în orice moment, există mai multe metode care și-au început executarea, dar nu și-au încheiat-o deoarece punctul de executare se află într-o altă metodă. Succesiunea de apeluri de metode a căror executare a început, dar nu s-a și încheiat este numită **call-stack** (stiva cu apeluri de metode) și reprezintă un concept important în logica tratării erorilor.

Să presupunem, de exemplu, că avem o aplicație cu o interfață grafică care conține un buton "Statistică persoane". În momentul apăsării butonului, se apelează o metodă "AchiziționeazaButon", pentru a trata evenimentul, care la rândul său apelează o metodă "CalculStatistică" dintr-o altă clasă, iar aceasta, la rândul său, apelează o metodă "ÎncarcăDateDinFișier". Se obține astfel un call-stack. În această situație, pot să apară mai multe excepții care pot proveni din diferite metode aflate pe call-stack: calea fișierului cu datele persoanelor este greșită sau fișierul nu există, unele persoane au datele eronate în fișier etc. Indiferent de metoda în care va apărea o excepție, aceasta trebuie semnalată utilizatorului în interfața grafică, adică trebuie să aibă loc o propagare a excepției, fără a bloca funcționalitatea aplicației.

O variantă de rezolvare ar fi utilizarea unor coduri pentru excepții, dar acest lucru ar complica foarte mult codul (multe `if-uri`), iar coduri precum -1, -20 etc. nu sunt descriptive pentru excepția apărută. În limbajul Java, există un mecanism eficient de tratare a excepțiilor. Practic, o excepție este un obiect care încapsulează detalii despre excepția respectivă, precum metoda în care a apărut, metodele din call-stack afectate, o descriere a sa etc.

Tipuri de excepții:

- **erori:** sunt generate de hardware sau de JVM, ci nu de program, ceea ce înseamnă că nu pot fi anticipate, deci *nu este obligatorie tratarea lor* (exemplu: `OutOfMemoryError`)
- **excepții la compilare:** sunt generate de program, ceea ce înseamnă că pot fi anticipate, deci *este obligatorie tratarea lor* (exemplu: `IOException`, `SQLException` etc.)
- **excepții la rulare:** sunt generate de o situație particulară care poate să apară la rulare, ceea ce înseamnă că pot fi foarte numeroase (nu există o listă completă a lor), *deci nu este obligatorie tratarea lor* (exemplu: `IndexOutOfBoundsException`, `NullPointerException`, `ArithmeticException` etc.)

Deoarece există mai multe situații în care pot apărea excepții, Java pune la dispoziție o ierarhie complexă de clase dedicate (Fig. 1).

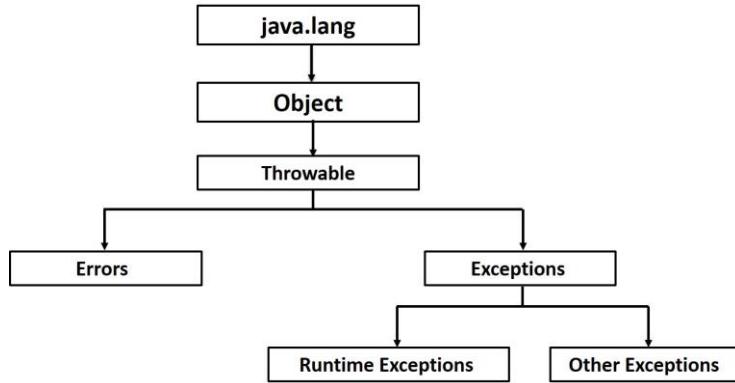


Fig. 1 - Ierarhia de clase pentru tratarea excepțiilor

Se poate observa cum există o multitudine de tipuri derivate din `Exception` sau `RuntimeException`, distribuite în diverse pachete Java. De regulă, excepțiile nu sunt grupate într-un singur pachet (nu există un pachet `java.exception`), ci sunt

definite în aceleasi pachete cu clasele care le generează. De exemplu, `IOException` este definită în `java.io`, `AWTException` în `java.awt` etc. Lista de excepții definite în fiecare pachet poate fi găsită în documentația Java API.

Exemple de excepții uzuale:

- `IOException` - apare în operațiile de intrare/ieșire (de exemplu, citirea dintr-un fișier sau din rețea). O subclăsă a clasei `IOException` este `FileNotFoundException`, generată în cazul încercării de deschidere a unui fișier inexistent;
- `NullPointerException` - folosirea unei referințe cu valoarea `null` pentru accesarea unui membru public sau default dintr-o clasă;
- `ArrayIndexOutOfBoundsException` - folosirea unui index incorrect, respectiv negativ sau strict mai mare decât dimensiunea fizică a unui tablou - 1;
- `ArithmaticException` - operații aritmetice nepermise, precum împărțirea unui număr întreg la 0;
- `IllegalArgumentException` – utilizarea incorrectă a unui argument pentru o metodă. O subclăsă a clasei `IllegalArgumentException` este `NumberFormatException` care corespunde erorilor de conversie a unui `String` într-un tip de date primitiv din cadrul metodelor `parseTipPrimitiv` ale claselor wrapper;
- `ClassCastException` - apare la conversia unei referințe către un alt tip de date incompatibil;
- `SQLException` - excepții care apar la interogarea serverelor de baze de date.

Mecanismul folosit pentru manipularea excepțiilor predefinite este următorul:

- *generarea excepției*: când apare o excepție, JVM instanțiază un obiect al clasei `Exception` care încapsulează informații despre excepția apărută;
- *lansarea/aruncarea excepției*: obiectul generat este transmis mașinii virtuale;
- *propagarea excepției*: JVM parcurge în sens invers call-stack-ul, căutând un handler (un cod) care tratează acel tip de eroare;
- *prinderea și tratarea excepției*: primul handler găsit în call-stack este executat ca reacție la apariția erorii, iar dacă nu se găsește niciun handler, atunci JVM oprește executarea programului și afișează un mesaj descriptiv de eroare.

Sintaxa utilizată pentru tratarea exceptiilor:

```

try {
    bloc de instrucțiuni
}
catch(Excepție_A e) {
    Tratare excepție A
}
catch(Excepție_B e) {
    Tratare excepție B (mai generală)
}
finally {
    Bloc care se execută întotdeauna
}

```

Observații:

- Un bloc try-catch poate să conțină mai multe blocuri catch, însă acestea trebuie să fie specificate de la particular către general (și în această ordine vor fi și tratate). De exemplu Excepție_A este o subclasa a clasei Excepție_B

Exemplu: Următoarea aplicație, care citește două numere întregi dintr-un fișier text, conține un bloc catch pentru a trata excepția care poate să apară dacă se încercă deschiderea unui fișier inexistent, dar poate să conțină și un bloc catch care tratează excepții de tipul ArithmeticException și/sau excepții de tipul InputMismatchException.

```

public class Test {
    public static void main(String[] args) {
        int a, b;
        try {
            Scanner f = new Scanner(new File("numere.txt"));
            a = f.nextInt();
            b = f.nextInt();
            double r;
            r = a / b;
            System.out.println(r);
        }
        catch(FileNotFoundException e) {
            System.out.println("Fisier inexistent");
        }
        catch(InputMismatchException e) {
            System.out.println("Format incorect al unui numar");
        }
        catch(ArithmeticException e) {
            System.out.println("Impartire la 0");
        }
        finally {
            System.out.println("Bloc finally");
        }
    }
}

```

```

    }
}

```

- Blocurile catch se exclud reciproc, respectiv o excepție nu poate fi tratată de mai multe blocuri catch.

Exemplu:

- dacă nu există fișierul numere.txt, atunci se lansează și se tratează doar excepția FileNotFoundException, afișând-se în fereastra System mesajul "Fisier inexistent", fără a se executa și blocurile ArithmeticException și InputMismatchException;
- dacă în fișierul numere.txt sunt valorile abc 0, atunci se lansează și se tratează doar InputMismatchException, fără a se executa și blocul ArithmeticException;
- dacă în fișierul numere.txt sunt valorile 13 0, atunci se lansează și se tratează ArithmeticException, fără a se executa InputMismatchException.

- Blocul finally nu are parametri și poate să lipsească, dar, dacă există, atunci se execută întotdeauna, indiferent dacă a apărut o excepție sau nu. Scopul său principal este acela de a eliberarea anumite resurse deschise, de exemplu, fișiere sau conexiuni de rețea.
- Blocul finally va fi executat întotdeauna după blocurile try și catch, astfel:
 - dacă în blocul try nu apare nicio excepție, atunci blocul finally este executat imediat după try;
 - dacă în blocul try este aruncată o excepție, atunci:
 - dacă există un bloc catch corespunzător, acesta va fi executat după întreruperea executării blocului try, urmat de blocul finally;
 - dacă nu există un bloc catch, atunci se execută blocul finally imediat după blocul try, după care JVM cauță un handler în metoda anterioară din call-stack;
 - blocul finally se execută chiar și atunci când folosim instrucțiunea return în cadrul blocurilor try sau catch!

Exemplu: După rularea programului de mai jos, se vor afișa mesajele Înainte de return și Bloc finally!

```

public class Test {
    static void test() {
        try {
            System.out.println("Înainte de return");
            return;
        }
        finally {
            System.out.println("Bloc finally");
        }
    }

    public static void main(String[] args) {
        test();
    }
}

```

Observație: instrucțiunea try-catch este un dispecer de excepții, similar instrucțiunii switch(TipExcepție), direcționând-se astfel fiecare excepție către blocul de cod care o tratează.

Excepții definite de către programator

Așa cum am precizat mai sus, standardul Java oferă o ierarhie complexă de clase pentru manipularea diferitelor tipuri de excepții, care pot să acopere multe dintre erorile întâlnite în programare. Totuși, pot exista situații în care trebuie să fie tratate anumite excepții specifice pentru logica aplicației (de exemplu, excepția dată de adăugarea unui element într-o stivă plină, introducerea unui CNP invalid, utilizarea unei date calendaristice anterioare unui proces etc.). În plus, excepțiile standard deja existente nu descriu întotdeauna detaliat o situație de eroare (de exemplu, IllegalArgumentException poate fi o informație prea vagă, în timp ce CNPInvalidException descrie mai bine o eroare și poate să permită o tratare separată a sa).

În acest sens, programatorul își poate defini propriile excepții, prin clase care extind fie clasa Exception (o excepție care trebuie să fie tratată), fie clasa RuntimeException (o excepție care nu trebuie să fie tratată neapărat).

Lansarea unei excepții se realizează prin clauza throw new ExcepțieNouă(<listă argumente>).

Exemplu: Vom implementa o stivă de numere întregi folosind un tablou unidimensional, precum și excepții specifice, astfel:

- definim o clasă StackException pentru manipularea excepțiilor specifice unei stive:

```
public class StackException extends Exception {
    public StackException(String mesaj) {
        super(mesaj);
    }
}
```

- definim o interfață Stack în care precizăm operațiile specifice unei stive, inclusiv excepțiile:

```
public interface Stack {
    void push(Object item) throws StackException;
    Object pop() throws StackException;
    Object peek() throws StackException;
    boolean isEmpty();
    boolean isFull();
    void print() throws StackException;
}
```

- definim o clasă StackArray în care implementăm operațiile definite în interfața Stack utilizând un tablou unidimensional, iar posibilele excepții le lansăm utilizând excepții descriptive de tipul StackException:

```
public class StackArray implements Stack {
    private Object[] stiva;
    private int varf;

    public StackArray(int nrMaximElemente) {
        stiva = new Object[nrMaximElemente];
```

```

        varf = -1;
    }

@Override
public void push(Object x) throws StackException {
    if (isFull())
        throw new StackException("Nu pot să adaug un element într-o
                                stivă plină!");

    stiva[++varf] = x;
}

@Override
public Object pop() throws StackException {
    if (isEmpty())
        throw new StackException("Nu pot să extrag un element dintr-o
                                stivă vidă!");

    Object aux = stiva[varf];
    stiva[varf--] = null;
    return aux;
}

@Override
public Object peek() throws StackException {
    if (isEmpty())
        throw new StackException("Nu pot să accesez elementul din
                                vârful unei stive vide!");

    return stiva[varf];
}

@Override
public boolean isEmpty() {
    return varf == -1;
}

@Override
public boolean isFull() {
    return varf == stiva.length - 1 ;
}

@Override
public void print() throws StackException {
    if (isEmpty())
        throw new StackException("Nu pot să afișez o stivă vidă!");

    System.out.println("Stiva: ");
    for(int i = varf; i >= 0; i--)
        System.out.print(stiva[i] + " ");
    System.out.println();
}
}

```

- Testăm clasa StackArray efectuând operații de tip push și pop în mod aleatoriu asupra unei stive

care poate să conțină maxim 3 numere întregi:

```
public class Test_StackArray {
    public static void main(String[] args) {
        StackArray st = new StackArray(3);

        Random rnd = new Random();
        for(int i = 0; i < 20; i++)
            try {
                int aux = rnd.nextInt();

                if(aux % 2 == 0)
                    st.push(1 + rnd.nextInt(100));
                else
                    st.pop();

                st.print();
            }
            catch(StackException ex) {
                System.out.println(ex.getMessage());
            }
    }
}
```

„Aruncarea” unei excepții

Dacă în corpul unei metode nu se tratează o anumită excepție sau un set de excepții, în antetul metodei se poate folosi clauza **throws** pentru ca acesta/acestea să fie tratate de către o metodă apelantă.

Sintaxa:

```
tip_returnat numeMetoda(<listă argumente>) throws listaExcepții
```

Exemplu:

```
void citire() throws IOException {
    System.in.read();
}

void citeșteLinie() {
    citire();
}
```

Metoda `citeșteLinie`, la rândul său, poate să “arunce” excepția `IOException` sau să o trateze printr-un bloc `try-catch`.

În concluzie, aruncarea unei excepții de către o metodă presupune, de fapt, pasarea explicită a responsabilității către codul apelant al acesteia. Vom proceda astfel numai când dorim să forțăm codul client să trateze excepția în cauză.

Observație: La redefinirea unei metode care “aruncă” excepții, nu se pot preciza prin clauza `throws` excepții suplimentare.

Observație: Începând cu Java 7, a fost introdusă instrucțiunea *try-with-resources* care permite închiderea automată a unei resurse, adică a unui surse de date de tip flux (de exemplu, un flux asociat unui fișier, o conexiune cu o bază de date etc.).

Sintaxă:

```
try (deschidere Resursă_1; Resursă_2) {
    .....
}
catch (...) {
    .....
}
```

Pentru a putea fi utilizată folosind o instrucțiune de tipul *try-with-resources*, clasa corespunzătoare unei resurse trebuie să implementeze interfața `AutoCloseable`. Astfel, în momentul terminării executării instrucțiunii se va închide automat resursa respectivă. Practic, după executarea instrucțiunii *try-with-resources* se vor apela automat metodele `close` ale resurselor deschise.

Exemplu: Indiferent de tipul lor, fluxurile asociate fișierelor se închid folosind metoda `void close()`, de obicei în blocul `finally` asociat instrucțiunii `try-catch` în cadrul căreia a fost deschis fluxul respectiv:

```
try {
    FileOutputStream fout = new FileOutputStream("numere.bin");
    DataOutputStream dout = new DataOutputStream(fout);
    .....
}
catch (...) {
    .....
}
finally {
    if(dout != null)
        dout.close();
}
```

Toate tipurile de fluxuri bazate pe fișiere implementează interfața `AutoCloseable`, deci pot fi deschise utilizând o instrucțiune de tipul *try-with-resources*.

```
try(FileOutputStream fout = new FileOutputStream("numere.bin");
     DataOutputStream dout = new DataOutputStream(fout);) {
    .....
}
catch (...) {
    .....
}
```

Observație: În momentul închiderii unui flux stratificat, aşa cum este fluxul `dout` din exemplul de mai sus, JVM va închide automat și fluxul primitiv pe bază căruia acesta a fost deschis!

FLUXURI DE INTRARE/IEȘIRE

Operațiile de intrare/ieșire sunt realizate, în general, cu ajutorul claselor din pachetul `java.io`, folosind conceptul de *flux* (stream).

Un *flux* reprezintă o modalitate de transfer al unor informații în format binar de la o sursă către o destinație.

În funcție de modalitatea de prelucrare a informației, precum și a direcției canalului de comunicație, fluxurile se pot clasifica astfel:

- după direcția canalului de comunicație:
 - de intrare
 - de ieșire
- după modul de operare asupra datelor:
 - la nivel de octet (flux pe 8 biți)
 - la nivel de caracter (flux pe 16 biți)
- după modul în care acționează asupra datelor:
 - primitive (doar operațiile de citire/scriere)
 - procesare (adaugă la cele primitive operații suplimentare: procesare la nivel de buffer, serializare etc.)

În concluzie, pentru a deschide orice flux se instanțiază o clasă dedicată, care poate conține mai mulți constructori:

- un constructor cu un argument prin care se specifică calea fișierului sub forma unui sir de caractere;
- un constructor care primește ca argument un obiect de tip File;
- un constructor care primește ca argument un alt flux.

Clasa `File` permite operații specifice fișierelor și directoarelor, precum creare, ștergere, mutare etc., mai puțin operații de citire/scriere.

Metode uzuale ale clasei `File`:

- `String getAbsolutePath()` – returnează calea absolută a unui fișier;
- `String getName()` – returnează numele unui fișier;
- `boolean createNewFile()` – creează un nou fișier, iar dacă fișierul există deja metoda returnează false;
- `File[] listFiles()` – returnează un tablou de obiecte `File` asociate fișierelor dintr-un director.

Fluxurile primitive permit doar operații de intrare/ieșire. După modul de operarea asupra datelor, fluxurile primitive se împart în două categorii:

1. **prelucrare la nivel de caracter (fișiere text)**: informația este reprezentată prin caractere Unicode, aranjate pe linii (separatorul poate fi '\r\n' (Windows), '\n' (Unix/Linux) sau '\r' (Mac)).

Informația fiind reprezentată prin caracter Unicode, se obține un flux pe 16 biți.

Pentru deschiderea unui flux primitiv la nivel de caracter de intrare se instanțiază clasa `FileReader`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui sir de caractere, fie printr-un constructor care primește ca argument un obiect de tip `File`.

```
FileReader fin = new FileReader("exemplu.txt");
```

sau

```
File f = new File("exemplu.txt");
FileReader fin = new FileReader(f);
```

Operația de citire a unui caracter se realizează prin metoda `int read()`.

Observație: Deschiderea unui fișier impune tratarea excepției `FileNotFoundException`.

Pentru deschiderea unui flux primitiv de ieșire la nivel de caracter se instanțiază clasa `FileWriter`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui `String`, fie printr-un constructor care primește ca argument un obiect de tip `File`.

```
FileWriter fout = new FileWriter("exemplu.txt");
```

sau

```
File f = new File("exemplu.txt");
FileWriter fout = new FileWriter (f);
```

Pentru deschiderea unui flux primitiv de ieșire la nivel de caracter în modul `append` (adăugare la sfârșitul fișierului), se utilizează constructorul `FileWriter(String fileName, boolean append)`.

Dacă parametrul `append` are valoarea `true`, atunci operațiile de scriere se vor efectua la sfârșitul fișierului (dacă fișierul nu există, mai întâi se va crea un fișier vid). Dacă parametrul `append` are valoarea `false`, atunci operațiile de scriere se vor efectua la începutul fișierului (indiferent de faptul că fișierul există sau nu, mai întâi se va crea un fișier vid, posibil prin suprascrierea unuia existent).

Operația de scriere a unui caracter se realizează prin metoda `void write(int ch)`.

Clasa `FileWriter` pune la dispoziție și alte metode pentru a scrie informația într-un fișier text:

- `public void write(String string)` - scrie în fișier sirul de caractere transmis ca parametru
- `public void write(char[] chars)` - scrie în fișier tabloul de caractere transmis ca parametru

Observație: Scrierea informației într-un fișier impune tratarea excepției `IOException`.

Exemplu: Copierea caracter cu caracter a fișierului text `test.txt` în fișierul text `copie_caractere.txt`

```
FileReader fin = new FileReader("test.txt");
FileWriter fout = new FileWriter("copie_caractere.txt", true);
```

```
int c;
while((c = fin.read()) != -1)
    fout.write(c);
```

2. prelucrare la nivel de octet(fișiere binare): informația este reprezentată sub forma unui sir octeți neformatați (2 octeți nu mai reprezintă un caracter) și nu mai există o semnificație specială pentru caracterele '\r' și '\n'.

Fișierele binare sunt des utilizate, deoarece acestea permit memorarea unor informații complexe, folosind un şablon, precum imagini, fișiere video etc. De exemplu, un fișier Word are un şablon specific, diferit de cel al unui fișier PDF.

Pentru deschiderea unui flux primitiv de intrare la nivel de octet se instanțiază clasa `FileInputStream`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui `String`, fie printr-un constructor care primește ca argument un obiect de tip `File`:

```
FileInputStream fin = new FileInputStream("test.txt");
```

sau

```
File f = new File("exemplu.txt");
FileInputStream fin = new FileInputStream(f);
```

Operația de citire a unui octet se realizează prin metoda `int read()`.

Clasa `FileInputStream` pune la dispoziție și alte metode pentru a realiza citirea informației dintr-un fișier binar, precum:

- `int read(byte[] bytes)` - citește un tablou de octeți și returnează numărul octețiilor citiți

Pentru deschiderea unui flux primitiv de ieșire la nivel de octet se instanțiază clasa `FileOutputStream`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui sir de caractere, fie printr-un constructor care primește ca argument un obiect de tip `File`:

```
FileOutputStream fout = new FileOutputStream("test.txt");
```

sau

```
File f = new File("exemplu.txt");
FileOutputStream fout = new FileOutputStream(f);
```

Operația de scriere a unui octet se realizează prin metoda `void write(int b)`.

Clasa `FileOutputStream` pune la dispoziție și alte metode pentru a realiza scrierea informației într-un fișier binar:

- `void write(byte[] bytes)` - scrie un tablou de octeți

Exemple:

1. Copierea directă a întregului conținut al fișierului text `test.txt` în fișierul text `copie_octeti.txt`.

```
FileInputStream fin = new FileInputStream("test.txt");
FileOutputStream fout = new FileOutputStream("copie_octeti.txt")
int dimFisier = fin.available(); //metoda returnează numărul de octeți din fișier
```

```

byte []buffer = new byte[dimFisier];
fin.read(buffer);           //se citesc toți octeții din fișierul de intrare
fout.write(buffer);         // se scriu toți octeții în fișierul de ieșire

```

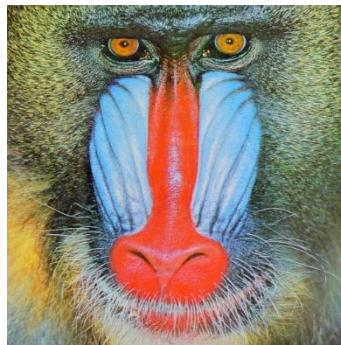
2. Formatul BMP (bitmap) pe 24 de biți este un format de fișier binar folosit pentru a stoca imagini color digitale bidimensionale având lățime, înălțime și rezoluție arbitrară. Practic, imaginea este considerată ca fiind un tablou bidimensional de pixeli, iar fiecare pixel este codificat prin 3 octeți corespunzători intensităților celor 3 canale de culoare R (red), G(green) și B(blue). Intensitatea fiecărui canal de culoare R, G sau B este dată de un număr natural cuprins între 0 și 255. De exemplu, un pixel cu valorile (0, 0, 0) reprezintă un pixel de culoare neagră, iar un pixel cu valorile (255, 255, 255) unul de culoare albă.

Formatul BMP cuprinde o zonă cu dimensiune fixă, numita *header*, și o zonă de date cu dimensiune variabilă care conține pixelii imaginii propriu-zise. Header-ul, care ocupă primii 54 de octeți ai fișierului, conține informații despre formatul BMP, precum și informații despre dimensiunea imaginii, numărul de octeți utilizati pentru reprezentarea unui pixel etc. Dimensiunea imaginii în octeți este specificată în header printr-o valoare întreagă, deci memorată pe 4 octeți, începând cu octetul cu numărul de ordine 2. Dimensiunea imaginii în pixeli este exprimată sub forma $W \times H$, unde W reprezintă numărul de pixeli pe lățime, iar H reprezintă numărul de pixeli pe înălțime. Lățimea imaginii exprimată în pixeli este memorată pe patru octeți începând cu octetul al 18-lea din header, iar înălțimea este memorată pe următorii 4 octeți fără semn, respectiv începând cu octetul al 22-lea din header.

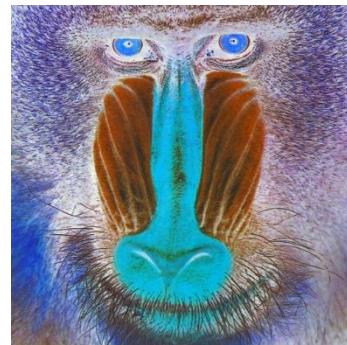
După cei 54 de octeți ai header-ului, într-un fișier BMP urmează zona de date, unde sunt memorate ÎN ORDINE INVERSĂ liniile de pixeli ai imaginii, deci ultima linie de pixeli din imagine va fi memorată prima, penultima linie va fi memorată a doua, ..., prima linie din imagine va fi memorată ultima. Deoarece codarea unei imagini BMP pe 24 de biți într-un fișier binar respectă standardul *little-endian*, octeții corespunzători celor 3 canale de culoare RGB sunt memorati de la dreapta la stânga, în ordinea BGR!

Pentru rapiditatea procesării imaginilor la citire și scriere, imaginile în format BMP au proprietatea că fiecare linie este memorată folosind un număr de octeți multiplu de 4. Dacă este necesar, acest lucru de realizează prin adăugarea unor octeți de completare (*padding*) la sfârșitul fiecărei linii, astfel încât numărul total de octeți de pe fiecare linie să devină multiplu de 4. Numărul de octeți corespunzători unui linii este $3 \times W$ (câte 3 octeți pentru fiecare pixel de pe o linie). Astfel, dacă o imagine are $W = 11$ pixeli în lățime, atunci numărul de octeți de padding este 3 ($3 \times 11 = 33$ octeți pe o linie, deci se vor adăuga la sfârșitul fiecărei linii câte 3 octeți de completare, astfel încât să avem $33 + 3 = 36$ multiplu de 4 octeți). De obicei, octeții de completare au valoarea 0.

În continuare, considerăm imaginea *baboon.bmp* ca fiind imaginea de intrare, iar imaginea de ieșire ca fiind complementara sa, care se obține prin scăderea valorii fiecărui canal de culoare al unui pixel din valoarea maximă posibilă pe un canal de culoare).



baboon.bmp



complementara_baboon.bmp

Pentru a construi imaginea de ieșire, copiem mai întâi header-ul imaginii de intrare în imaginea de ieșire și apoi parcurgem fișierul de intrare la nivel de octet (variabila octet) pentru a accesa valorile de pe fiecare canal de culoare R, G și B din fiecare pixel și scriem în fișierul de ieșire valoarea complementară a octetului, respectiv $255 - \text{octet}$:

```
public class Prelucrare_BMP {
    public static void main(String[] args) throws FileNotFoundException,
        IOException {
        FileInputStream fin = new FileInputStream("baboon.bmp");
        FileOutputStream fout = new FileOutputStream("complement_baboon.bmp");

        byte[] header = new byte[54];
        fin.read(header);
        fout.write(header);

        int octet;
        while((octet = fin.read()) != -1)
            fout.write(255 - octet);

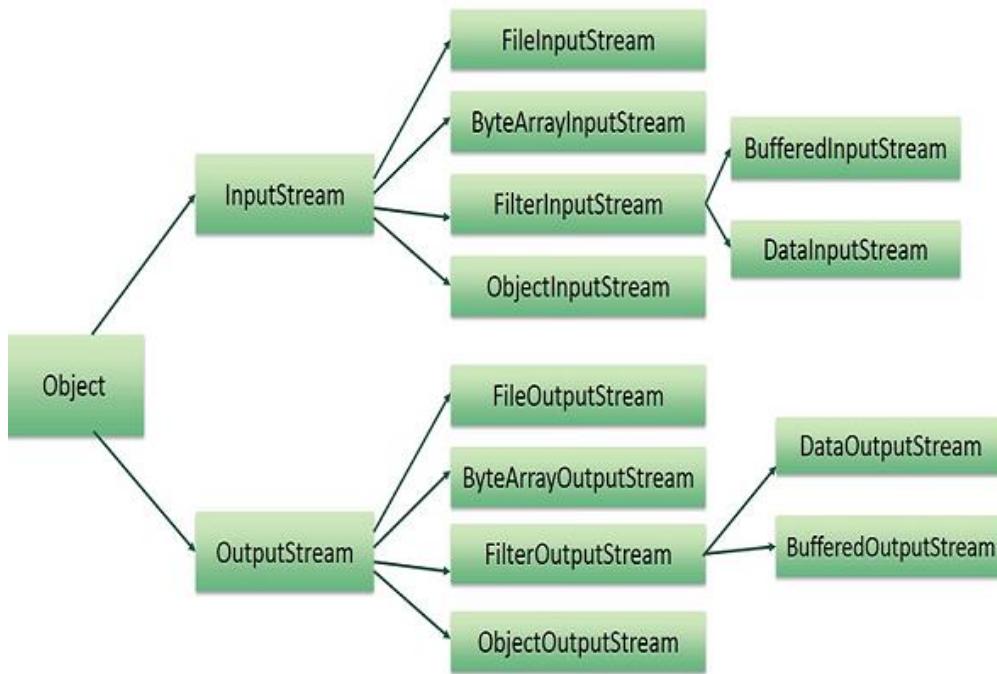
        fin.close();
        fout.close();
    }
}
```

Fluxuri de procesare

Limbajul Java pune la dispoziție o serie de fluxuri de intrare/ieșire care au o structură stratificată pentru a adăuga funcționalități suplimentare pentru fluxurile primitive, într-un mod dinamic și transparent. De exemplu, se poate adăuga la un flux primitiv binar de intrare operații care permit citirea tipurilor primitive (de exemplu, pentru a citi un număr întreg se grupează câte 4 octeți) sau a unui sir de caractere.

Această modalitate de a oferi implementări stratificate este cunoscută sub numele de *Decorator Pattern*. Conceptul în sine impune ca obiectele care adaugă funcționalități (*wrappers*) unui obiect să aibă o interfață comună cu acesta. În felul acesta, se obține transparență, adică un obiect poate fi folosit fie în forma primitivă, fie în forma superioară stratificată (decorat).

Limbajul Java pune la dispoziție o ierarhie complexă de clase pentru a prelucra fluxurile de procesare, aşa cum se poate observa în figura de mai jos.



Observație: O ierarhie asemănătoare există și pentru fluxurile care procesează alte fluxuri la nivel de caracter.

Constructorii claselor pentru fluxurile de procesare nu primesc ca argument un dispozitiv extern de memorare a datelor, ci o referință a unui flux primitiv.

```

FluxPrimitiv flux = new FluxPrimitiv(<lista arg>);
FluxDeProcesare fluxProcesare = new FluxDeProcesare(flux);
  
```

Exemple de fluxuri de procesare:

1. Fluxurile de procesare DataInputStream/DataOutputStream

Fluxul procesat nu mai este interpretat la nivel de octet, ci octeții sunt grupați astfel încât aceștia să reprezinte date primitive sau șiruri de caractere (String). Cele două clase furnizează metode pentru citirea și scrierea datelor la nivel de tip primitiv, prezentate în tabelul de mai jos:

DataInputStream	DataOutputStream
boolean readBoolean()	void writeBoolean(boolean v)
byte readByte()	void writeByte(byte v)
char readChar()	void writeChar(int v)
double readDouble()	void writeDouble(double v)
float readFloat()	void writeFloat(float v)
int readInt()	void writeInt(int v)
long readLong()	void writeLong(long v)
short readShort()	void writeShort(int v)
String readUTF()	void writeUTF(String str)

În exemplul de mai jos se realizează scrierea formatată a unui tablou de numere reale în fișierul binar *numere.bin*. Ulterior, folosind un flux binar se realizează citirea formatată a tabloului.

```

public class Fluxuri_date_primitive {
    public static void main(String[] args) {
        try(DataOutputStream fout = new DataOutputStream(
                new FileOutputStream("numere.bin")));
            double v[] = {1.5 , 2.6 , 3.7 , 4.8 , 5.9};
            fout.writeInt(v.length);
            for(int i = 0; i < v.length; i++)
                fout.writeDouble(v[i]);
        }
        catch (IOException ex) {
            System.out.println("Eroare la scrierea in fisier!");
        }

        try(DataInputStream fin = new DataInputStream(
                new FileInputStream("numere.bin")));
            int n = fin.readInt();
            double []v = new double[n];

            for(int i = 0; i < v.length; i++)
                v[i] = fin.readDouble();
            for(int i = 0; i < v.length; i++)
                System.out.print(v[i] + " ");
        }
        catch (IOException ex) {
            System.out.println("Eroare la citirea din fisier!");
        }
    }
}

```

2. Fluxuri de procesare pentru citirea/scrierea datelor folosind un buffer

Operațiile de citire/scriere la nivel de caracter/octet, specifice fluxurilor primitive, conduce la un număr mare de accesări ale fluxului respectiv (și, implicit, ale dispozitivului de memorie externă pe care este stocat fișierul asociat), ceea ce poate afecta eficiența din punct de vedere al timpului de execuție. În scopul de a elimina acest neajuns, fluxurile de procesare la nivel de buffer introduc în procesele de scriere/citire o zonă auxiliară de memorie, astfel încât informația să fie accesată în blocuri de caractere/octeți având o dimensiune predefinită, ceea ce conduce la scăderea numărului de accesări ale fluxului respectiv.

Clase pentru citirea/scrierea cu buffer:

- `BufferedReader`, `BufferedWriter` – fluxuri de procesare la nivel de buffer de caractere
- `BufferedInputStream`, `BufferedOutputStream` – fluxuri de procesare la nivel de buffer de octeți

Constructori:

- `FluxProcesareBuffer flux = new FluxProcesareBuffer(
 new FluxPrimitiv("cale fisier"));`
- `FluxProcesareBuffer flux = new FluxProcesareBuffer(
 new FluxPrimitiv("cale fisier"), int dimBuffer);`

Dimensiunea implicită a buffer-ului utilizat este de 512 octeți.

Metodele uzuale ale acestor clase sunt: `read/readline`, `write`, `flush` (goiese explicit buffer-ul, chiar dacă acesta nu este plin).

Exemplu: Fișierul `date.in` conține un text dispus pe mai multe linii. În fișierul `date.out` sunt afișate, pe fiecare linie, cuvintele sortate crescător lexicografic.

```
public class CitireBuffer {
    public static void main(String[] args) {
        try(BufferedReader fin = new BufferedReader(new FileReader("date.in"));
            BufferedWriter fout = new BufferedWriter(new FileWriter("date.out")));
        {
            String linie;
            while((linie=fin.readLine())!=null)
            {
                String cuv[] = linie.split(" ");
                Arrays.sort(cuv);
                System.out.println(Arrays.toString(cuv));
                for(int i=0; i<cuv.length; i++)
                    fout.write(cuv[i]+" ");
                fout.write("\n");
            }
        }
        catch (FileNotFoundException ex) {
            System.out.println("Fisierul nu exista!");
        }
        catch(IOException ex) {
            System.out.println("Operatie de citire/scriere esuata!");
        }
    }
}
```

Fluxuri de procesare cu acces aleatoriu

Toate fluxurile de procesare prezentate anterior sunt limitate la o accesare secvențială a sursei/destinației de date. Astfel, nu putem accesa (citi scrie) direct un anumit octet/caracter/valoare din flux, ci trebuie să accesăm, pe rând, toate valorile aflate înaintea sa, de la începutul fluxului respectiv. Dacă pentru unele categorii de fluxuri accesarea secvențială este indispensabilă (de exemplu, în cazul unor fluxuri cu ajutorul cărora se transmit date într-o rețea), în cazul anumitor tipuri de fișiere se poate opta pentru o accesare directă, mai eficientă în cazul în care nu este necesară procesarea tuturor datelor din fișier, ci doar a unei poziții cunoscute (de exemplu, lățimea unei imagini în format *bitmap* (BMP) este memorată pe 4 octeți, începând cu octetul 18, iar pe următorii 4 octeți, începând cu octetul 22, este memorată înălțimea sa).

Pentru accesarea aleatorie a octetilor unui fișier, în limbajul Java este utilizată clasa `RandomAccessFile`, care nu aparține niciunei ierarhii de clase menționate până acum. Accesarea aleatorie a octetilor unui fișier se realizează prin intermediul unui *cursor* asociat fișierului respectiv (file pointer) care memorează numărul de ordine al octetului curent (în momentul deschiderii unui fișier, cursorul asociat este pozitionat pe primul octet din fișier – octetul cu numărul de ordine 0). Practic, fișierul este privit ca un tablou unidimensional de octeți memorat

pe un suport extern, iar cursorul reprezintă indexul octetului curent. Orice operație de citire/scriere se va efectua asupra octetului curent, după care se va actualiza valoarea cursorului. De exemplu, dacă octetul curent este octetul 10 și vom scrie în fișier valoarea unei variabile de tip `int`, care se memorează pe 4 octeți, valoarea cursorului va deveni 14.

Deschiderea unui fișier cu acces aleatoriu se poate realiza utilizând unul dintre cei 2 constructori ai clasei `RandomAccessFile`, unul având ca parametru un obiect de tip `File`, iar celălalt având ca parametru calea fișierului sub forma unui sir de caractere:

- `RandomAccessFile(File file, String mode)`
- `RandomAccessFile(String name, String mode)`

Parametrul `mode` este utilizat pentru a indica modalitatea de deschidere a fișierului, astfel:

- `"r"` – fișierul este deschis doar pentru citire (dacă fișierul nu există, se va lansa excepția `FileNotFoundException`);
- `"rw"` – fișierul este deschis pentru citire și scriere (dacă fișierul nu există, se va crea unul vid).

Clasa `RandomAccessFile` implementează interfețele `DataInput` și `DataOutput` (care sunt implementate, de exemplu, și de clasele `DataInputStream/DataOutputStream`), deci conține metode pentru citirea/scrierii:

- *octeților sau tablourilor de octeți* – utilizând metode `read/write` asemănătoare celor din clasele `FileInputStream/FileOutputStream`;
- *valori de tip primitiv sau siruri de caractere* – utilizând metodele `readTip/writeTip` asemănătoare celor din clasele `DataInputStream` și `DataOutputStream`

În cazul apariției unor erori la scrierea/citirea datelor se va lansa o excepție de tipul `IOException`.

În afara metodelor pentru citirea/scrierea datelor, clasa `RandomAccessFile` conține și metode specifice pentru poziționarea cursorului fișierului:

- `long getFilePointer()` – furnizează valoarea curentă a cursorului asociat fișierului, raportată la începutul fișierului (octetul cu numărul de ordine 0);
- `void seek(long pos)` – mută cursorul asociat fișierului pe octetul cu numărul de ordine `pos` față de începutul fișierului (octetul cu numărul de ordine 0);
- `int skipBytes(int n)` – mută cursorul asociat fișierului peste `n` octeți față de poziția curentă.

Observație: În limbajul Java, toate fișierele binare sunt considerate în mod implicit ca fiind de tip *big-endian* în mod implicit, respectiv octetul cel mai semnificativ dintr-un grup de octeți va fi memorat primul în fișierul binar. În cazul în care o aplicație Java manipulează fișiere binare de tip *little-endian* (octetul cel mai semnificativ dintr-un grup de octeți va fi memorat ultimul), create, de exemplu, utilizând limbajele C/C++ în sistemul de operare Microsoft Windows, acest fapt poate genera probleme foarte mari, deoarece datele vor fi interpretate eronat!

De exemplu, să considerăm o valoare `int x = 720`, care se memorează pe 4 octeți în limbajele C/C++/Java. În baza 2, valoarea `x` este egală cu 1011010000, deci, folosind standardul *little-endian*, reprezentarea sa internă va fi egală cu 11010000 | 00000010 | 00000000 | 00000000 (prin | am delimitat octeții). Dacă această reprezentare binară va fi interpretată folosind standardul *big-endian*, atunci ea va fi considerată ca fiind egală, în baza 10, cu -805175296!

Pentru a rezolva această problemă, se poate proceda în două moduri:

- dacă valoarea este de tip `char`, `short`, `int` sau `long`, se poate utiliza metoda `reverseBytes` din clasa `Înfășurătoare corespunzătoare`. De exemplu, citim dintr-un fișier `fin` cu acces aleatoriu o valoare `int x=fin.readInt()` și schimbăm ordinea octetilor `x=Integer.reverseBytes(x)` sau, direct, prin `x=Integer.reverseBytes(fin.readInt())`.
- o altă variantă, care poate fi utilizată pentru orice tip de date, constă în utilizarea unui obiect de tip `ByteBuffer` pentru manipularea sirurilor de octeți:

```

RandomAccessFile fin = new RandomAccessFile("fisier.bin", "r");

//citim din fișier o valoare de tip double direct,
//sub forma unui sir de 8 octeți
byte []valoareDouble = new byte[8];
fin.read(valoareDouble);

fin.close();

//alocăm un obiect de tip ByteBuffer care să permită manipularea
//a 8 octeți și stabilim ordinea lor ca fiind little-endian
ByteBuffer aux = ByteBuffer.allocate(8);
aux.order(ByteOrder.LITTLE_ENDIAN);

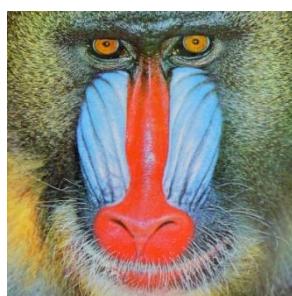
//încarcăm sirul de octeți în obiectul ByteBuffer și apoi
//preluăm valoarea de tip double astfel obținută
aux.put(valoareDouble);
double x = aux.getDouble(0);
System.out.println("x = " + x);

```

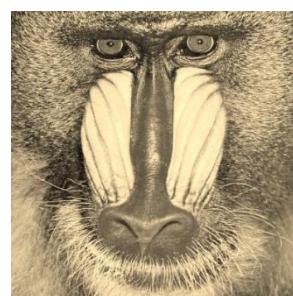
Exemplu: O imagine color poate fi transformată într-o imagine sepia înlocuind valorile (R, G, B) ale fiecărui pixel cu valorile (R', G', B') definite astfel:

$$\begin{aligned} R' &= \min \{[0.393 * R + 0.769 * G + 0.189 * B], 255\} \\ G' &= \min \{[0.349 * R + 0.686 * G + 0.168 * B], 255\} \\ B' &= \min \{[0.272 * R + 0.534 * G + 0.131 * B], 255\} \end{aligned}$$

unde prin $[x]$ am notat partea întreagă a numărului real x .



baboon.bmp (color)



baboon.bmp (sepia)

În următoarea aplicație Java, vom utiliza un fișier cu acces aleatoriu pentru a afișa dimensiunea imaginii în octeți și în pixeli, după care vom transforma imaginea color inițială într-una sepia, ținând cont de faptul că fișierele BMP sunt implicit de tip *little-endian*:

```
public class Prelucrare_BMP_sepia {
    public static void main(String[] args) throws FileNotFoundException,
        IOException {

        //deschidem fișierul în mod mixt, deoarece trebuie să efectuăm și
        //operații de citire și operații de scriere
        RandomAccessFile img = new RandomAccessFile("baboon.bmp", "rw");

        //citim din fișier dimensiunea imaginii în octeți și o afișam
        byte []b = new byte[4];
        ByteBuffer aux = ByteBuffer.allocate(4);

        img.seek(2);
        img.read(b);

        aux.put(b);
        aux.order(ByteOrder.LITTLE_ENDIAN);
        int imgBytes = aux.getInt(0);

        System.out.println("Dimensiunea imaginii: " + imgBytes + " bytes");

        //citim din fișier dimensiunea imaginii în pixeli și o afișam
        img.seek(18);

        int imgWidth = img.readInt();
        imgWidth = Integer.reverseBytes(imgWidth);

        int imgHeight = img.readInt();
        imgHeight = Integer.reverseBytes(imgHeight);

        System.out.println("Dimensiunea imaginii: " + imgWidth + " x " +
                           imgHeight + " pixeli");

        //calculăm padding-ul imaginii și îl afișăm
        int imgPadding;

        if(imgWidth % 4 != 0)
            imgPadding = 4 - (3 * imgWidth) % 4;
        else
            imgPadding = 0;

        System.out.println("Padding-ul imaginii: " + imgPadding + " bytes");

        //modificăm imaginea color într-una sepia

        //în tabloul de octeți pixelRGB citim valorile pixelului curent color
        byte []pixelRGB = new byte[3];
```

```

//în tabloul de octeți auxRGB vom calcula noile valori ale pixelului
//curent transformat în sepia, folosind formulele de mai sus și înănd
//cont de faptul că în fișier canalele de culoare sunt în ordinea BGR
byte []auxRGB = new byte[3];
double tmp = 0;

//mutăm cursorul la începutul zonei de date, imediat după header-ul
//de 54 de octeți
img.seek(54);

for(int h = 0; h < imgHeight; h++) {
    for(int w = 0; w < imgWidth; w++) {
        //citim valorile RGB ale pixelului curent în ordinea BGR
        img.read(pixelRGB);

        //calculăm valorile sepia ale pixelului curent
        tmp = 0.272*Byte.toUnsignedInt(pixelRGB[2]) +
              0.534*Byte.toUnsignedInt(pixelRGB[1]) +
              0.131*Byte.toUnsignedInt(pixelRGB[0]);
        auxRGB[0] = (byte) (tmp <= 255 ? tmp : 255);

        tmp = 0.349*Byte.toUnsignedInt(pixelRGB[2]) +
              0.686*Byte.toUnsignedInt(pixelRGB[1]) +
              0.168*Byte.toUnsignedInt(pixelRGB[0]);
        auxRGB[1] = (byte) (tmp <= 255 ? tmp : 255);

        tmp = 0.393*Byte.toUnsignedInt(pixelRGB[2]) +
              0.769*Byte.toUnsignedInt(pixelRGB[1]) +
              0.189*Byte.toUnsignedInt(pixelRGB[0]);
        auxRGB[2] = (byte) (tmp <= 255 ? tmp : 255);

        //ne întoarcem 3 octeți în fișier pentru a suprascrize valorile
        //color ale pixelului curent cu cele sepia calculate mai sus
        img.seek(img.getFilePointer() - 3);
        img.write(auxRGB);
    }
}

//după ce am prelucrat toți pixelii de pe o linie, sărim peste
//pixelii de padding
img.skipBytes(imgPadding);
}

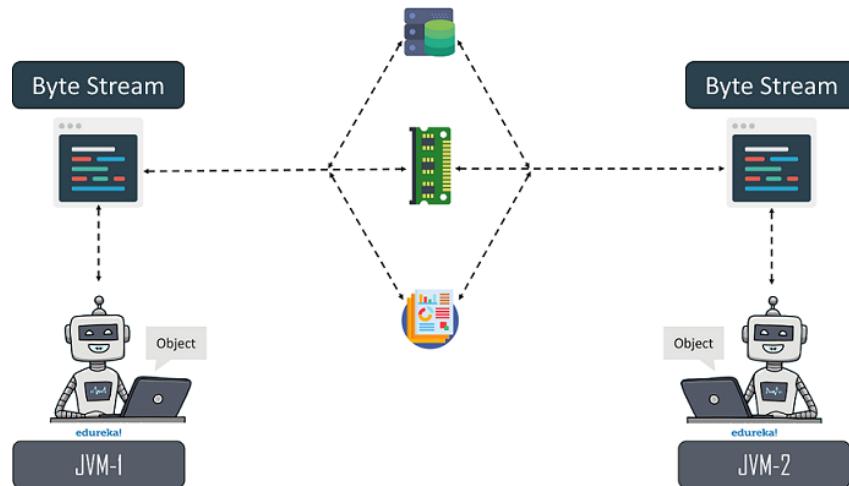
img.close();
}
}

```

SERIALIZAREA OBIECTELOR

Ciclul de viață al unui obiect este determinat de executarea programului, respectiv obiectele instantiatе în cadrul său sunt stocate în memoria internă, astfel încât, după ce acesta își termină executarea, zona de memorie alocată programului este eliberată. În cadrul aplicațiilor, de cele mai multe ori, se dorește salvarea obiectelor între diferite rulări ale programului sau se dorește ca acestea să fie transmise printr-un un canal de comunicație. O soluție aparent simplă pentru rezolvarea acestei probleme ar constitui-o salvarea stării unui obiect (valorile datelor membre) într-un fișier (text sau binar) și restaurarea ulterioră a acestuia, pe baza valorilor salvate, folosind un constructor al clasei. Totuși, această soluție devine foarte complicată dacă unele date membre sunt referințe către alte obiecte, deoarece ar trebui salvate și restaurate și stările acelor obiecte externe! Mai mult, în acest caz nu s-ar salva funcționalitățile obiectului (metodele sale) și constructorii.

Limbajul Java permite o rezolvare simplă și eficientă a acestei probleme prin intermediul mecanismelor de *serializare* și *deserializare* (sursa imaginii: <https://www.edureka.co/blog/serialization-in-java/>):



Serializarea este mecanismul prin care un obiect este transformat într-o secvență de octeți din care acesta să poată fi refăcut ulterior, iar *deserializarea* reprezintă mecanismul invers serializării, respectiv dintr-o secvență de octeți serializați se restaurează obiectul original.

Utilizarea mecanismului de serializare prezintă mai multe avantaje:

- obiectele pot fi salvate/restaurate într-un mod unitar pe/de pe diverse medii de stocare (fișiere binare, baze de date etc.);
- obiectele pot fi transmise foarte simplu între mașini virtuale Java diferite, care pot rula pe calculatoare având arhitecturi sau sisteme de operare diferite;
- timpul necesar serializării sau deserializării unui obiect este mult mai mic decât timpul necesar salvării sau restaurării unui obiect pe baza valorilor datelor sale membre (de exemplu, în momentul deserializării unui obiect nu se apelează constructorul clasei respective);
- cea mai simplă și mai rapidă metodă de clonare a unui obiect (*deep copy*) o reprezintă serializarea/deserializarea sa într-un/dintr-un tablou de octeți.

Obiectele unei clase sunt serializabile dacă respectiva clasă implementează interfața `Serializable`. Această interfață este una de marcat, care nu conține nicio metodă abstractă, deci, prin implementarea sa clasa respectivă doar anunță mașina virtuală Java faptul că dorește să-i fie asigurat mecanismul de serializare. O clasă nu este implicit serializabilă, deoarece clasa `java.lang.Object` nu implementează interfața `Serializable`. Totuși, anumite clase standard, cum ar fi clasa `String`, clasele înfășurătoare (wrapper), clasa `Arrays` etc., implementează interfața `Serializable`.

Pentru prezentarea mecanismului de serializare/deserializare, vom considera definită clasa `Student`, care implementează interfața `Serializable`, având datele membre `String nume`, `int grupa`, un tablou `note` cu elemente de tip `int` pentru a reține notele unui student, `double medie` și o dată membră statică `facultate` de tip `String`, respectiv metodele de tip `set/get` corespunzătoare, metoda `toString()` și constructori:

```
public class Student implements Serializable
{
    private static String facultate;
    private String nume;
    private int grupa, note[];
    private double medie;
    ...
}
```

Serializarea unui obiect se realizează astfel:

- se deschide un flux binar de ieșire utilizând clasa `java.io.ObjectOutputStream`:
`FileOutputStream file = new FileOutputStream("studenti.bin");`
`ObjectOutputStream fout = new ObjectOutputStream(file);`
- se salvează scrie obiectul în fișier apelând metoda `void writeObject(Object ob)`:
`Student s = new Student("Ion Popescu", 241, new int[]{10, 9, 10, 7, 8});`

Deserializarea unui obiect se realizează astfel:

- se deschide un flux binar de intrare utilizând clasa `java.io.ObjectInputStream`:
`FileInputStream file = new FileInputStream("studenti.bin");`
`ObjectInputStream fin = new ObjectInputStream(file);`
- se citește/restaurează obiectul din fișier apelând metoda `Object readObject()`:
`Student s = (Student)fin.readObject();`

Mecanismul de serializare a unui obiect presupune salvarea, în format binar, a următoarelor informații despre acesta:

- denumirea clasei de apartenență;
- versiunea clasei de apartenență, implicit aceasta fiind hash-code-ul acesteia, calculat de către mașina virtuală Java;
- valorile datelor membre de instanță.
- antetele metodelor membre.

Observații:

- Implicit NU se serializează datele membre statice și nici corporile metodelor, ci doar antetele lor.
- Explicit NU se serializează datele membre marcate prin modificadorul transient (de exemplu, s-ar putea să nu dorim salvarea anumitor informații confidențiale: salariaj unei persoane, parola unui utilizator etc.).
- Serializarea nu tine cont de specificatorii de acces, deci se vor serializa și datele/metodele private!
- În momentul sterilizării unui obiect se va serializa întregul graf de dependențe asociat obiectului respectiv, adică obiectul respectiv și toate obiectele referite direct sau indirect de el.

Exemplu:

Considerăm clasa Nod care modelează un nod al unei liste simplu înlățuite:

```
class Nod implements Serializable
{
    Object data;
    Nod next;

    public Nod(Object data)
    {
        this.data = data;
        this.next = null;
    }
}
```

Folosind clasa Nod, vom construi o listă circulară, formată din numerele naturale cuprinse între 1 și 10, pe care apoi o vom salva/serializa în fișierul binar lista.ser, scriind în fișier doar primul său nod (obiectul prim) – restul nodurilor listei vor fi salvate/serializate automat, deoarece ele formează graful de dependențe asociat obiectului prim:

```
public class Serializare_listă_circulară
{
    public static void main(String[] args)
    {
        Nod prim = null, ultim = null;

        for (int i = 1; i <= 10; i++)
        {
            Nod aux = new Nod(i);

            if (prim == null) prim = ultim = aux;
            else
            {
                ultim.next = aux;
                ultim = aux;
            }
        }
        ultim.next = prim;
    }
}
```

```

System.out.println("Lista care va fi serializată:");
Nod aux = prim;
do
{
    System.out.print(aux.data + " ");
    aux = aux.next;
}
while(aux != prim);

try (ObjectOutputStream fout = new ObjectOutputStream(
        new FileOutputStream("lista.ser")))
{
    fout.writeObject(prim);
}
catch (IOException ex) { System.out.println("Excepție: " + ex); }
}
}

```

Pentru a restaura lista circulară salvată/serializată în fișierul binar `lista.ser`, vom citi/deserializa din fișier doar primul său nod (obiectul `prim`), iar restul nodurilor listei vor fi restaurate/deserializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim` (evident, clasa `Nod` trebuie să fie vizibilă):

```

public class Deserializare_listă_circulară
{
    public static void main(String[] args)
    {
        try (ObjectInputStream fin = new ObjectInputStream(
                new FileInputStream("lista.ser")))
        {

            Nod prim = (Nod) fin.readObject();

            System.out.println("Lista deserializată:");
            Nod aux = prim;
            do
            {
                System.out.print(aux.data + " ");
                aux = aux.next;
            }
            while(aux != prim);

            System.out.println();
        }
        catch (Exception ex)
        {
            System.out.println("Excepție: " + ex);
        }
    }
}

```

În exemplul prezentat, graful de dependențe asociat obiectului prim este unul ciclic, deoarece lista este circulară (deci există o referință indirectă a obiectului prim către el însuși), dar mecanismul de serializare gestionează fără probleme o astfel de situație complicată!

- Dacă un obiect care trebuie serializat conține referințe către obiecte neserializabile, atunci va fi generată o excepție de tipul `NotSerializableException`.
- Dacă o clasă serializabilă extinde o clasă neserializabilă, atunci datele membre accesibile ale superclasei nu vor fi serializate. În acest caz, superclasa trebuie să conțină un constructor fără argumente pentru a inițializa în procesul de restaurare a obiectului datele membre moștenite.
- Dacă se modifică structura clasei aflată pe mașina virtuală care realizează serializarea obiectelor (de exemplu, prin adăugarea unui câmp nou privat, care, oricum, nu va fi accesibil), fără a se realiza aceeași modificare și pe mașina virtuală destinație, atunci procesul de deserializare va lansa la executare excepția `InvalidClassException`. Excepția apare deoarece cele două clase nu mai au aceeași versiune. Practic, versiunea unei clase se calculează în mod implicit de către mașina virtuală Java printr-un algoritm de hash care este foarte sensibil la orice modificare a clasei. În practică, sunt diferite situații în care se dorește modificarea clasei pe mașina virtuală care realizează procesul de serializare (de exemplu, adăugând date sau metode private care vor fi utilizate doar intern), fără a afecta, însă, procesul de deserializare. O soluție în acest sens o constituie introducerea unei noi date membre în clasă, `private static final long serialVersionUID`, prin care se definește explicit versiunea clasei - caz în care mașina virtuală Java nu va mai calcula, implicit, versiunea clasei respective pe baza structurii sale. Un exemplu bun în acest sens se găsește în pagina <https://www.baeldung.com/java-serial-version-uid>.

EXTERNALIZAREA OBIECTELOR

Deși mecanismul de serializare este unul foarte puternic și util, totuși, el are și câteva dezavantaje:

- serializarea unui obiect nu ține cont de modificatorii de acces ai datelor membre, deci vor fi salvate în format binar și datele de tip `protected/private`, ceea ce permite reconstituirea valorilor lor prin tehnici de *reverse engineering* (https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3_5-Serialization.pdf);
- serializarea nu salvează datele membre statice;
- serializarea unui obiect necesită destul de mult spațiu de stocare, deoarece se salvează multe informații auxiliare (de exemplu, cele referitoare la superclasele clasei corespunzătoare obiectului);
- serializarea unui obiect se realizează destul de lent, fiind un proces recursiv vizavi de superclasă și/sau referințe către alte obiecte.

Dezavantajele menționate anterior pot fi ameliorate sau eliminate folosind mecanismul de *externalizare*, care este, de fapt, o serializare complet controlată de către programator (implicit se salvează doar numele clasei). Astfel, programatorul poate decide datele care vor fi salvate în format binar, precum și modalitatea utilizată pentru salvarea lor. De exemplu, pentru a asigura confidențialitatea unei date membre a unui obiect la serializare/deserializare (de exemplu, salariul unui angajat), este necesară criptarea/decriptarea întregului obiect, ceea ce va crește considerabil durata celor două procese. Folosind mecanismul de externalizare, se poate crita/decripta doar data membră respectivă!

Externalizarea obiectelor unei clase este condiționată de implementarea interfeței Externalizable:

```
public interface Externalizable extends Serializable
{
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
                                              ClassNotFoundException;
}
```

Practic, în cadrul celor două metode writeExternal și readExternal, programatorul își poate implementa proprii algoritmi de salvare și restaurare a unui obiect.

Exemplu:

Vom prezenta modul în care mecanismul de externalizare poate fi aplicat în cazul clasei Student, menționată anterior:

```
public class Student implements Externalizable
{
    public static String facultate;

    String nume;
    int grupa, note[];
    double medie;
    .....

    //se va salva denumirea facultății (prin serializare nu ar fi fost salvată,
    //deoarece este o dată membră statică)
    //nu se vor salva notele unui student
    //media va fi "criptată" folosind formula 2*medie+3
    @Override
    public void writeExternal(ObjectOutput out) throws IOException
    {
        out.writeUTF(facultate);
        out.writeUTF(nume);
        out.writeInt(grupa);
        out.writeDouble(2*medie+3);
        out.writeObject(note);
    }

    //media va fi "decriptată" folosind formula inversă (medie-3)/2
    @Override
    public void readExternal(ObjectInput in) throws IOException,
                                              ClassNotFoundException
    {
        facultate = in.readUTF();
        nume = in.readUTF();
        grupa = in.readInt();
        medie = in.readDouble();
        medie = (medie - 3)/2;
    }
}
```

După implementarea celor două metode `writeExternal` și `readExternal`, salvarea/restaurarea datelor se va realiza la fel ca și în cazul serializării, apelând metodele `writeObject` și `readObject`.

Exemplu:

Considerăm un tablou `s` cu elemente de tip `Student`:

```
Student s[] = new Student[5];
Student.facultate = "Facultatea de Matematica si Informatica";

s[0] = new Student(...);
.......
```

Salvarea tabloului `s` într-un fișier binar `studenti_extern.ser` se poate realiza astfel:

```
try(ObjectOutputStream fout = new ObjectOutputStream(
        new FileOutputStream("studenti_extern.ser")))
{
    fout.writeObject(s);
}
catch (IOException ex)
{
    System.out.println("Excepție: " + ex);
}
```

Restaurarea tabloului `s` din fișierul binar `studenti_extern.ser` se poate realiza astfel:

```
Student s[];

try(ObjectInputStream fin = new ObjectInputStream(
        new FileInputStream("studenti_extern.ser")))
{
    s = (Student [])fin.readObject();
}
catch (Exception ex)
{
    System.out.println(ex);
}
```

Un aspect pe care nu trebuie să-l pierdem din vedere în momentul utilizării externalizării este faptul că se vor pierde toate facilitățile puse la dispoziție de mecanismul de serializare (salvarea automată a informațiilor despre superclasele clasei respective, salvarea automată a grafului de dependențe etc.), deci un programator va trebui să le implementeze explicit!

COLECȚII

O *colecție* este un obiect container care grupează mai multe elemente într-o structură unitară. Intern, elementele dintr-o colecție se află într-o relație specifică unei structuri de date (lineară, asociativă, arborescentă etc.), astfel încât asupra lor se pot efectua operații de căutare, adăugare, modificare, ștergere, parcursere etc.

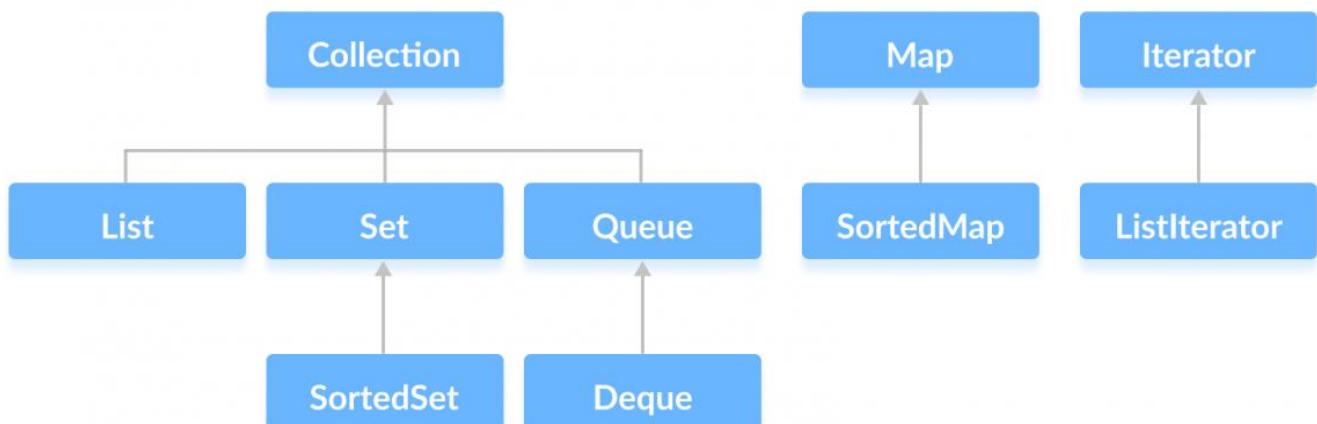
Limbajul Java oferă un framework/API performant pentru crearea și managementul structurilor dinamice de date (tablou, liste, mulțime, tabelă de asocieri etc) – *Java Collections Framework*, astfel încât programatorul să fie degrevat de implementarea optimă a lor.

Framework-ul Collections are o arhitectură bazată pe interfețe și clase care permite reprezentarea și manipularea unitară a colecțiilor, într-un mod independent de detaliile de implementare, astfel:

- *interfețe* – definesc într-un mod abstract operațiile specifice diverselor colecții;
- *clase* – conțin implementări concrete ale colecțiilor definite în interfețe, iar începând cu Java 1.5 ele sunt generice (tipul de date concret al elementelor din colecție se precizează prin operatorul `< >` (operatorul diamond): `List<Persoana> lp = new ArrayList<>();`);
- *algoritmi polimorfici* – sunt metode statice, grupate în care clasa utilitară `Collections`, care implementează optim operații generice caracteristice colecțiilor de date, cum ar fi: căutare, sortare, copiere, determinarea minimului/maximului etc.

Principalele ierarhii de interfețe puse la dispoziție de framework-ul Java Collections sunt reprezentate în figura de mai jos (sursa: <https://www.programiz.com/java-programming/collections>):

Java Collections Framework



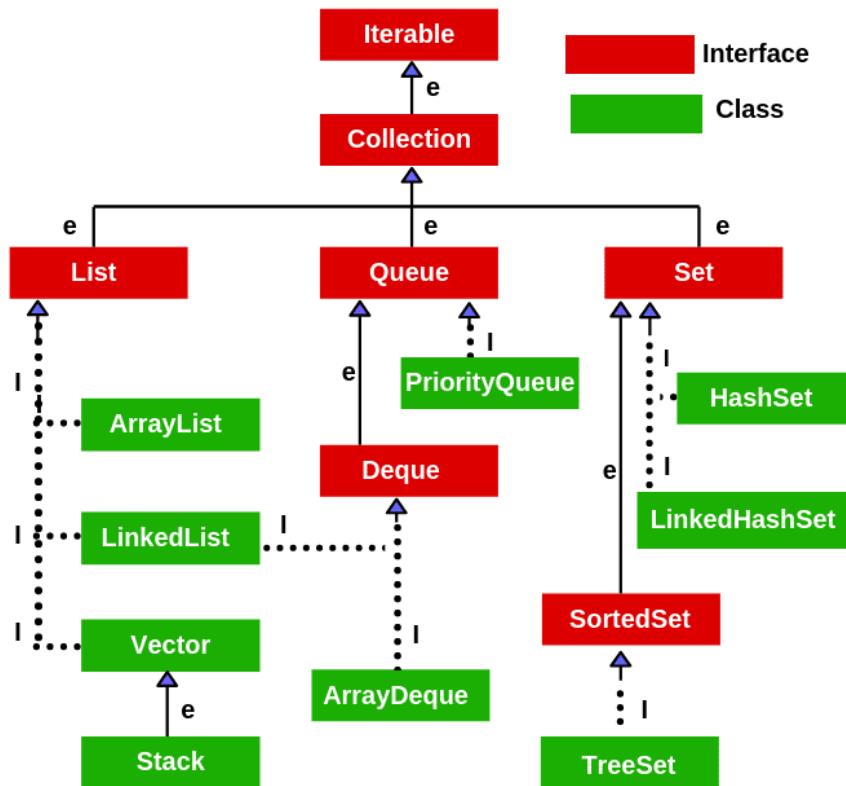
Interfață Collection

Nucleul principal al framework-ului Java Collections este reprezentat de interfața `Collection` care conține o serie de metode fundamentale de prelucrare specifice tuturor colecțiilor. O parte dintre metodele uzuale ale interfeței `Collection` sunt:

- `public int size()` – returnează numărul total de elemente din colecție;
- `public boolean add(E e)` – inserează în colecția curentă elementul `e`;
- `public boolean addAll(Collection<E> c)` – inserează în colecția curentă toate elementele din colecția `c`;
- `public boolean remove(Object e)` – șterge elementul `e` din colecția curentă;
- `public boolean contains(Object e)` – caută în colecția curentă elementul `e`;
- `public Iterator iterator()` – returnează un iterator pentru colecția curentă;
- `public Object[] toArray()` – realizează conversia colecției într-un tablou cu obiecte de tip `Object`.

Alte metode ale interfeței `Collection` sunt prezentate aici: <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>.

Ierarhia formată din interfețele care extind interfața `Collection`, precum și clasele care le implementează, este reprezentată în figura de mai jos (sursa: [https://www.scientecheeasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0](https://www.scientecheasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0)):



Parcursarea unei colecții presupune obținerea, pe rând, a unei referințe către fiecare obiect din colecție. O modalitate generală de a parcurge o colecție, independent de tipul ei, este reprezentată de *iteratori*. Astfel, vârful celor ierarhiei Collection este interfața Iterable.

Interfața List

Interfața List extinde interfața Collection și modelează o colecție de elemente ordonate care permite inclusiv memorarea elementelor duplicate.

Interfața List adaugă metode suplimentare fata de interfața Collection, corespunzătoare operațiilor care utilizează index-ului fiecărui element, considerat ca fiind de un tip generic E:

- accesarea unui element: E get(int index), E set(int index);
- adăugarea/ștergerea element: void add(int index, E element), E remove(int index);
- determinarea poziției unui element în cadrul colecției: int indexOf(Object e), int lastIndexOf(Object e).

Cele mai utilizate implementări ale interfeței List sunt clasele ArrayList și LinkedList.

Clasa ArrayList

Clasa ArrayList oferă o implementare a unei liste utilizând un tablou unidimensional care poate fi redimensionat dinamic:

```
List<Tip> listaTablou = new ArrayList<>();
ArrayList<Tip> listaTablou = new ArrayList<>();
```

Se poate observa cum o colecție ArrayList poate fi referită atât printr-o referință de tipul interfeței implementate (List), cât și printr-o referință de tipul colecției.

Capacitatea implicită a unei astfel de liste este egală cu 10, iar pentru a specifica explicit o altă capacitate se poate utiliza un constructor care primește ca argument un număr întreg:

```
List<Tip> listaTablou = new ArrayList<>(50);
```

Exemple:

```
List<Integer> lista1 = new ArrayList<>();
lista1.add(0, 1); // adaugă 1 pe poziția 0
lista1.add(1, 2); // adaugă 2 pe pozitia 1
System.out.println(lista1); // [1, 2]

List<Integer> lista2 = new ArrayList<Integer>();
lista2.add(1); // adaugă 1 la sfârșitul listei
lista2.add(2); // adaugă 2 la sfârșitul listei
lista2.add(3); // adaugă 3 la sfârșitul listei
System.out.println(lista2); // [1, 2, 3]
```

```
// adaugă elementele din lista2 începând cu poziția 1
list1.addAll(1, lista2);
System.out.println(list1); // [1, 1, 2, 3, 2]

// șterge elementul de pe poziția 1
list1.remove(1);
System.out.println(list1); // [1, 2, 3, 2]

// afișează elementul aflat pe poziția 3
System.out.println(list1.get(3)); // 2

// înlocuiește valoarea aflată pe poziția 1 cu valoarea 5
list1.set(1, 5);
System.out.println(list1); // [1, 5, 3, 2]
```

Observații:

- Accesarea unui element se realizează cu complexitatea $\mathcal{O}(1)$.
- Adăugarea unui element la sfârșitul listei prin metoda `add(T elem)` se realizează cu complexitatea $\mathcal{O}(1)$ dacă nu este depășită capacitatea listei sau cu complexitatea $\mathcal{O}(n)$ în caz contrar.
- Adăugarea unui element pe o anumită poziție prin metoda `add(E element, int index)` se realizează cu complexitatea $\mathcal{O}(n)$.
- Căutarea sau ștergerea unui element se realizează cu complexitatea $\mathcal{O}(n)$.

Clasa LinkedList

Clasa `LinkedList` oferă o implementare a unei liste utilizând o listă dublu înlanțuită, astfel fiecare nod al listei conține o informație de tip generic `E`, precum și două referințe: una către nodul anterior și una către nodul următor.

Constructorii clasei `LinkedList` sunt:

- `LinkedList()` – creează o listă vidă;
- `LinkedList(Collection C)` – creează o listă din elementele colecției `C`.

Pe lângă metodele implementate din interfața `List`, clasa `LinkedList` conține și câteva metode specifice:

- accesarea primului/ultimoelement: `E getFirst()`, `E getLast()`;
- adăugarea la începutul/sfârșitul listei: `void addFirst(E elem)`, `void addLast(E elem)`;
- ștergerea primului/ultimoelement: `E removeFirst()`, `E removeLast()`.

Exemple:

```
LinkedList<String> lista = new LinkedList<>();

// adăugarea unor elemente în listă
lista.add("A");
lista.add("B");
lista.addLast("C");
lista.addFirst("D");
lista.add(2, "E");
lista.add("F");
lista.add("G");
System.out.println(lista); // [D, A, E, B, C, F, G]
```

```

// ștergerea unor elemente din listă
lista.remove("B");
lista.remove(3);
lista.removeFirst();
lista.removeLast();
System.out.println(lista); // [A, E, F]

// căutarea unui element în listă
boolean rezultat = lista.contains("E");
System.out.println(rezultat); // true

// operații de accesare a unui element
Object element = lista.get(2);
System.out.println(element); // F
lista.set(2, "Y");
System.out.println(lista); // [A, E, Y]

```

Observații:

- Accesarea unui element se realizează cu complexitatea $\mathcal{O}(n)$.
- Adăugarea unui element la sfârșitul listei, folosind metoda `add(E elem)`, se realizează cu complexitatea $\mathcal{O}(1)$.
- Adăugarea unui element pe poziția `index`, folosind metoda `add(E elem, int index)`, se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Căutarea unui element se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Ștergerea unui element se realizează cu o complexitate egală cu $\mathcal{O}(n)$.

Pentru ca o aplicație să obțină performanțe cât mai bune din punct de vedere al timpului de executare, trebuie selectată colecția corespunzătoare funcționalității aplicației, astfel:

- dacă operațiile de accesare sunt predominante în cadrul aplicației, atunci se preferă utilizarea colecției `ArrayList`;
- dacă operațiile de actualizare (inserare/ștergere) sunt predominante, atunci se preferă utilizarea colecției `LinkedList`.

Pe lângă cele două implementări ale interfeței `List`, mai există și alte clase care o implementează:

- clasa `Vector` – are o funcționalitate similară clasei `ArrayList` și oferă metode sincronizate specifice aplicațiilor cu mai multe fire de executare;
- clasa `Stack` – extinde clasa `Vector` și oferă o implementare a unui vector cu funcționalitățile specifice structurii de date *stiva* (LIFO).

Interfața Set

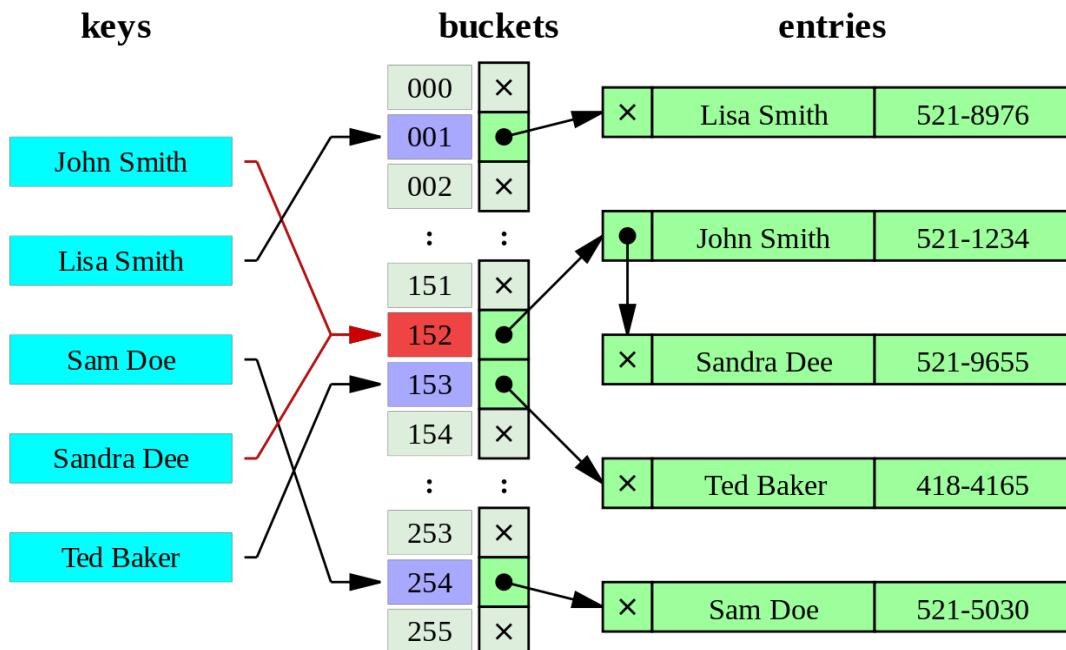
Interfața `Set` extinde interfața `Collection` și modelează o colecție de elemente care nu conțin duplicate, respectiv o colecție de tip mulțime.

Interfața `Set` nu adaugă metode suplimentare celor existente în interfața `List` și este implementată în clasele `HashSet`, `TreeSet` și `LinkedHashSet`.

Clasa HashSet

Într-un curs anterior, am văzut cum fiecare obiect are asociat un număr întreg, numit *hash-code*, puternic dependent față de orice modificare a datelor membre ale obiectului. Hash-code-ul unui obiect se calculează în metoda `int hashCode()`, moștenită din clasa `Object`, folosind algoritmi specifici care implică valorile datelor membre relevante pentru logica aplicației.

Clasa `HashSet` implementează o mulțime folosind o *tabelă de dispersie* (hash table). O tabelă de dispersie este un tablou unidimensional, numit *bucket array*, în care indexul unui element se calculează pe baza hash-code-ului său. Fiecare componentă a bucket array-ului va conține o listă cu obiectele care au același hash-code (*coliziuni*), aşa cum se poate observa în figura de mai jos (sursa: https://en.wikipedia.org/wiki/Hash_table):



Practic, o operație de inserare a unui obiect în tabela de dispersie presupune parcurgerea următorilor pași:

- se apelează metoda `hashCode` a obiectului respectiv, iar valoarea obținută se utilizează pentru a calcula indexul bucket-ului asociat obiectului respectiv;
- dacă bucket-ul respectiv este vid, atunci se adaugă direct obiectul respectiv și operația de inserare se încheie;
- dacă bucket-ul respectiv nu este vid, se parcurge lista asociată și, folosind metoda `equals`, se verifică dacă obiectul este deja inserat în tabelă, iar în caz negativ obiectul se adaugă la sfârșitul listei.

Evident, într-un mod asemănător, se vor efectua și operațiile de căutare, actualizare sau ștergere.

Se observă foarte ușor faptul că performanțele unei tabele de dispersie sunt puternic influențate de performanțele algoritmului de calcul al hash-code-ului unui obiect, respectiv acesta trebuie să fie sensibil la orice modificare a datelor membre pentru a minimiza numărul de coliziuni (obiecte diferite din punct de vedere al conținutului, dar care au același hash-code), ideal fiind ca hash-code-ul unui obiect să fie unic. În acest caz, lista asociată oricărui bucket va fi foarte scurtă, deci operațiile de căutare/inserare/ștergere/modificare vor avea complexitatea $\mathcal{O}(1)$, altfel, în cazul existenței multor coliziuni, complexitatea poate ajunge $\mathcal{O}(n)$.

Un alt aspect foarte important îl constituie implementarea/rescrierea corectă și a metodei `equals`, moștenită tot din clasa `Object`, în concordanță cu implementarea metodei `hashCode`, respectând următoarele reguli:

- metoda `hashCode` trebuie să returneze aceeași valoare în timpul rulării unei aplicații, indiferent de câte ori este apelată, dacă starea obiectului nu s-a modificat, dar nu trebuie să furnizeze aceeași valoare în cazul unor rulări diferite;
- două obiecte egale din punct de vedere al metodei `equals` trebuie să fie egale și din punct de vedere al metodei `hashCode`, deci trebuie să aibă și hash code-uri egale;
- nu trebuie neapărat ca două obiecte diferite din punct de vedere al conținutului să aibă hash-code-uri diferite, dar, dacă acest lucru este posibil, se vor obține performanțe mai bune pentru operațiile asociate unei tabele de dispersie.

Dacă a doua regulă nu este respectată, adică două obiecte egale din punct de vedere al conținutului (metoda `equals`) au hash-code-uri diferite (metoda `hashCode`), atunci operațiile de căutare/inserare într-o tabelă de dispersie vor fi incorecte. Astfel, în cazul în care se încearcă inserarea celui de-al doilea obiect după inserarea primului, operația de căutare a celui de-al doilea obiect se va efectua după valoarea hash-code-ului său, diferită de cea a primului obiect, deci îl va căuta în alt bucket și nu îl va găsi, ceea ce va conduce la inserarea și a celui de-al doilea obiect în tabela, deși el are același conținut cu primul obiect!

De obicei, acest aspect negativ apare în momentul în care programatorul nu rescrie metodele `hashCode` și `equals` într-o clasă ale cărei instanțe vor fi utilizate în cadrul unor colecții bazate pe tabele de dispersie, deoarece, implicit, metoda `hashCode` furnizează o valoare calculată pe baza referinței obiectului respectiv, iar metoda `equals` testează egalitatea a două obiecte comparând referințele lor. Astfel, două obiecte diferite cu același conținut vor fi considerate diferite de metoda `equals` și vor avea hash-code-uri diferite!

Exemplu: Considerăm definită clasa `Persoana` în care nu am rescris metodele `hashCode` și `equals`:

```
HashSet<Persoana> lp = new HashSet<>();
Persoana p1 = new Persoana("Popescu Ion", 23);
Persoana p2 = new Persoana("Popescu Ion", 23);

lp.add(p1);
lp.add(p2);

System.out.println(lp.size());
```

În urma rulării secvenței de cod de mai sus, se va afișa valoarea 2, deoarece ambele obiecte `p1` și `p2` vor fi inserate în `HashSet`-ul `lp`! Evident, problema se rezolvă implementând corect metodele `equals` și `hashCode` în clasa `Persoana`.

O problemă asemănătoare apare dacă se modifică valoarea unei date membre a unui obiect care este folosit pe post de cheie într-un `HashMap` (de exemplu, se modifică numele unei persoane), în cazul în care dacă valoarea datei membre respective este utilizată în implementările metodelor `hashCode` și `equals`. Din acest motiv, pentru chei se recomandă utilizarea unor obiecte care sunt instanțe ale unor clase imutabile!

Observații:

- Într-un HashSet se poate insera și valoare null, evident, o singură dată.
- O colecție de tip HashSet nu păstrează elementele în ordine inserării lor și nici nu pot efectua operații de sortare asupra sa.
- Implicit, *capacitatea inițială* (numărul de bucket-uri) a unei colecții de tip HashSet este 16, iar apoi aceasta este incrementată pe măsură ce se inserează elemente. Capacitatea inițială se poate stabili în momentul instantierii sale, folosind constructorul HashSet (int capacitate). În plus, pentru o astfel de colecție este definit un *factor de umplere* (load factor) care reprezintă pragul maxim permis de populare a colecției, depășirea sa conducând la dublarea capacității acesteia. Implicit, factorul de umplere este egal cu valoarea 0.75, ceea ce înseamnă că după ce se vor utiliza 75% din numărul de bucket-uri curente, numărul acestora va fi dublat. Astfel, considerând valorile implicate, prima dublare a numărului de bucket-uri va avea loc după ce se vor ocupa $0.75 \times 16 = 12$ bucket-uri, a doua dublare după ce se vor ocupa $0.75 \times 32 = 24$ de bucket-uri și.m.d.

Clasa LinkedHashSet

Implementarea clasei LinkedHashSet este similară cu implementarea clasei HashSet, diferența constând în faptul că elementele vor fi stocate în ordinea inserării lor.

Exemplu: Pentru a găsi numerele distincte dintr-un fișier text, vom utiliza un obiect nrđist de tip HashSet în care vom insera, pe rând fiecare număr din fișier:

```
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner in = new Scanner(new File("numere.txt"));
        HashSet<Integer> nrđist = new HashSet();

        while (in.hasNextLine()) {
            String linie = in.nextLine();
            String[] numere = linie.split("[ ,.;:?!]+");
            for (String nr : numere)
                nrđist.add(Integer.parseInt(nr));
        }

        System.out.println("Valorile distincte din fisier: ");
        for (int x : nrđist)
            System.out.print(x + " ");

        in.close();
    }
}
```

După executarea programului de mai sus, se vor afișa valorile distincte din fișierul text, într-o ordine oarecare. Dacă în locul clasei HashSet vom utiliza clasa LinkedHashSet, atunci valorile distincte vor fi afișate în ordinea inserării, adică în ordinea în care ele apar în fișierul text.

Clasa TreeSet

Intern, clasa TreeSet implementează o mulțime utilizând un arbore binar de tip Red-Black pentru a stoca elemente într-o anumită ordine, respectiv în ordinea lor naturală când se utilizează constructorul fără parametri ai clasei și clasa corespunzătoare obiectelor implementează interfața Comparable sau într-o ordine specificată în constructorul clasei printr-un argument de tip Comparator:

```
TreeSet t = new TreeSet();
TreeSet t = new TreeSet(Comparator comp);
```

Observații:

- Metodele add, remove și contains au o complexitate specifică structurii arborescente binare de tip Red-Black, respectiv $\mathcal{O}(\log_2 n)$.
- Colectia TreeSet este utilă în aplicații care necesită stocarea unui număr mare de obiecte sortate după un anumit criteriu, regăsirea informației fiind rapidă.

Exemplu: Revenind la exemplu anterior, dacă dorim să valorile distincte în ordine descrescătoare, mai întâi definim comparatorul

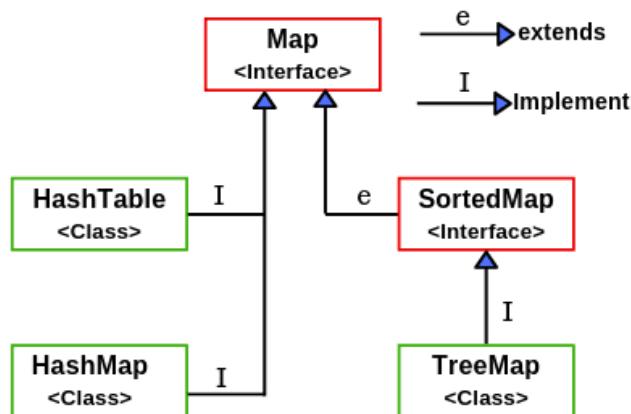
```
class cmpNumere implements Comparator<Integer> {
    @Override
    public int compare(Integer x, Integer y) {
        return y - x;
    }
}
```

și utilizăm constructorul corespunzător al clasei TreeSet:

```
TreeSet<Integer> nrlist = new TreeSet(new cmpNumere());
```

Interfața Map

Interfața Map, deși face parte din framework-ul Java Collections, nu extinde interfața Collection, ci este rădâcina unei ierarhii separate, aşa cum se poate observa în figura de mai jos (sursa: [https://www.scientecheeasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0](https://www.scientecheasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0))



Interfața Map modelează comportamentul colecțiilor ale căror elemente sunt de perechi de tipul *cheie – valoare* (definite în interfața `Map.Entry<T, R>`), prin care se asociază unei chei care trebuie să fie unice o singură valoare. Exemple: număr de telefon – persoană, CNP – persoană, cuvânt – frecvență sau într-un text, număr matricol – student etc.

Câteva metode uzuale din interfața Map sunt următoarele:

- `R put (T cheie, R valoare)` – inserează perechea cheie-valoare în colecție în cazul în care cheia nu există deja și returnează `null`, iar altfel înlocuiește vechea valoare asociată cheii cu noua valoare și returnează vechea valoare;
- `R get (T cheie)` – returnează valoarea asociată cheii indicate sau `null` dacă în colecție nu există cheia respectivă;
- `boolean containsKey(T cheie)` – returnează `true` dacă în colecție există cheia respectivă sau `false` în caz contrar;
- `boolean containsValue(R valoare)` – returnează `true` dacă în colecție există valoarea respectivă sau `false` în caz contrar;
- `Set<Map.Entry<K, V>> entrySet()` – returnează o mulțime care conține toate perechile cheie-valoare din colecție;
- `Set<K> keySet()` – returnează o mulțime care conține toate cheile din colecție;
- `Collection<V> values()` – returnează o colecție care conține toate valorile din colecția de tip Map;
- `R remove(Object cheie)` – dacă în colecție există cheia indicată, atunci elimină din colecție perechea având cheia respectivă și returnează valoarea cu care era asociată, altfel returnează `null`;
- `boolean remove(Object cheie, Object valoare)` – dacă în colecție există perechea cheie-valoare dată, atunci o elimină și returnează `true`, altfel returnează `false`;
- `void clear()` – elimină toate perechile existente în colecție.

Interfața Map este implementată în clasele `HashMap` și `TreeMap`, pe care le vom prezenta în continuare.

Clasa `HashMap`

Intern, implementarea clasei `HashMap` utilizează o tabelă de dispersie în care indexul bucket-ului în care va fi plasată o anumită valoare este dat de hash-code-ul corespunzător cheii (`cheie.hashCode()`), deci toate operațiile de căutare/inserare/ștergere se vor efectua în funcție de hash-code-ul cheii!

Complexitățile minime și medii ale metodelor `get`, `put`, `containsKey` și `remove` sunt $\mathcal{O}(1)$ în cazul implementării în metoda `hashCode()` a unei funcții de dispersie bune, care generează valori uniform distribuite, dar se poate ajunge la o complexitate maximă egală cu $\mathcal{O}(n)$, unde n reprezintă numărul de elemente din `HashMap`-ul respectiv, în cazul utilizării unei funcții de dispersie slabe, care produce multe coliziuni.

Observații:

- Într-un `HashMap` este permisă utilizarea valorii `null` atât pentru cheie, cât și pentru valoare.
- Într-un `HashMap` se poate asocia aceeași valoare mai multor chei.
- Într-un `HashMap` nu se menține ordinea de inserare și nici nu se poate stabili o anumită ordine a perechilor!
- Într-un `HashMap` se pot realiza și mapări complexe:

```

//h1 conține studenții anului I folosind perechi număr_matricol - student
HashMap<String, Student> h1 = new HashMap<>();
//h2 conține studenții anului II folosind perechi număr_matricol - student
HashMap<String, Student> h2 = new HashMap<>();
//m conține studenții din fiecare an folosind perechi an_studiu - studenți
HashMap<Integer, HashMap<String, Student>> m = new HashMap();

h1.put("11111", new Student("Ion Popescu", 141, new int[]{10, 9, 10, 7, 8}));
h1.put("22222", new Student("Anca Pop", 142, new int[]{9, 10, 10, 8}));
h2.put("12121", new Student("Ana Ionescu", 241, new int[]{8, 9, 10}));
h2.put("12345", new Student("Radu Mihai", 242, new int[]{9, 10, 8}));

m.put(1, h1);
m.put(2, h2);

for(Map.Entry<Integer, HashMap<String, Student>> hms : m.entrySet()) {
    System.out.println("An " + hms.getKey() + ": ");
    for(Map.Entry<String, Student> s : hms.getValue().entrySet())
        System.out.println(s);
}

```

Exemplu: Pentru a calcula frecvența cuvintelor dintr-un fișier, vom folosi un HashMap cu perechi de forma *cuvânt – frecvență_cuvânt*. Fiecare cuvânt din fișier va fi căutat în HashMap și dacă nu există deja, va fi inserat cu frecvența 1, altfel i se va actualiza frecvența mărită cu 1 (prin reinserare):

```

public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner in = new Scanner(new File("exemplu.txt"));
        HashMap<String, Integer> fcuv = new HashMap();

        while(in.hasNextLine()) {
            String linie = in.nextLine();
            String []cuvinte = linie.split("[ ,.;:?!]+");
            for(String cuvant : cuvinte)
                if(fcvu.containsKey(cuvant))
                    fcvu.put(cuvant, fcvu.get(cuvant) + 1);
                else
                    fcvu.put(cuvant, 1);
        }

        System.out.println("Frecvențele cuvintelor din fisier: ");
        for(Map.Entry<String, Integer> aux : fcvu.entrySet())
            System.out.println(aux.getKey() + " -> " + aux.getValue());

        in.close();
    }
}

```

Clasa TreeMap

Intern, implementarea clasei TreeMap utilizează un arbore binar de tip Red-Black pentru a menține perechile *cheie-valoare* sortate fie în ordine naturală a cheilor, dacă se utilizează constructorul fără parametri, fie în ordinea indusă de un comparator transmis ca parametru al constructorului. Astfel, dacă în exemplul anterior înlocuim obiectul de tip HashMap cu un obiect de tip TreeMap și utilizăm tot constructorul fără argumente, cuvintele din fișier vor fi afișate în ordine alfabetică. Dacă dorim să afișăm cuvintele în ordinea crescătoare a lungimilor lor, iar în cazul unor cuvinte de lungimi egale în ordine alfabetică, definim comparatorul

```
class cmpCuvinte implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        if(s1.length() != s2.length())
            return s1.length() - s2.length();
        else
            return s1.compareTo(s2);
    }
}
```

și utilizăm constructorul corespunzător din clasa TreeMap:

```
TreeMap<String, Integer> fcuv = new TreeMap(new cmpCuvinte());
```

Observații:

- Perechile dintr-un TreeMap pot fi sortate doar folosind criterii care implică doar cheile, ci nu și valorile, deci un comparator care va fi transmis ca parametru constructorului clasei TreeMap trebuie să țină cont de această restricție! Pentru a sorta perechile folosind criterii care implică valorile, de obicei, se preferă extragerea tuturor perechilor într-o colecție care permite realizarea operației de sortare după diverse criterii într-un mod simplu (de exemplu, o listă sau un tablou unidimensional).
- Operațiile de inserare/căutare/ștergere într-un TreeMap se realizează tot pe baza hash-code-ului corespunzător cheii, dar utilizarea unui arbore Red-Black garantează o complexitate egală cu $\mathcal{O}(\log_2 n)$ pentru metodele get, put, containsKey și remove.

Interfața Iterator

Rolul general al unui iterator este acela de a parurge elementele unei colecții de orice tip, mai puțin a celor care fac parte din ierarhia interfeței Map. Orice colecție c din ierarhia interfeței Collection conține o implementare a metodei iterator () care returnează un obiect de tip Iterator<Tip>:

```
Iterator itr = c.iterator();
```

În interfața Iterator sunt definite următoarele metode pentru accesarea elementelor unei colecții:

- public Object next () – returnează succesorul elementului curent;
- public boolean hasNext () – returnează true dacă în colecție mai există elemente nevizitate sau false în caz contrar.

Un iterator nu permite modificarea valorii elementului curent și nici adăugarea unor elemente noi în colecție!

Exemplu:

```
LinkedList<String> lista = new LinkedList<>();

lista.add("Ion");
lista.add("Vasile");
lista.addLast("Ana");
lista.addFirst("Radu");
lista.add(2, "Ioana");

Iterator itr = lista.iterator();
while(itr.hasNext())
    System.out.println(itr.next());
```

Orice colecție conține metode `remove` pentru ștergerea unui element având o anumită poziție și/sau o anumită valoare. Totuși, în cazul în care o colecție este parcursă fie "clasic", utilizând o instrucție de tip *enhanced-for*, fie cu un iterator, aceste metode nu pot fi utilizate, aşa cum vom vedea în următoarele două exemple:

Exemplu 1: Ștergerea valorilor egale cu 1 dintr-o listă folosind o instrucție de tip *enhanced-for*

```
List<Integer> lista = new ArrayList<>();

lista.add(1);
.....
for(Integer item:lista)
    if(item == 1)
        lista.remove(item);
```

Exemplu 2: Ștergerea numerelor pare dintr-o listă folosind un iterator

```
List<Integer> numere = new ArrayList<Integer>();

numere.add(101);
.....
Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
    Integer nr = itr.next();
    if (nr % 2 == 0)
        numere.remove(nr);
}
```

Deși apelul metodei `remove` este formal corect, în momentul executării sevențelor de cod de mai sus apare excepția `ConcurrentModificationException`, deoarece operația de ștergere se realizează în timpul iterării colecției!

De obicei, această excepție apare în aplicații multi-thread (aplicații cu mai multe fire de executare), unde nu este permis ca un fir de executare să modifice o colecție în timp ce un alt fir de executare parcurge colecția respectivă. Totuși, excepția apare și în aplicații cu un singur fir de executare, dacă se realizează parcurgea unei colecții cu un iterator de tip *fail fast iterator*, aşa cum este cel utilizat în implementarea internă a instrucțiunii *enhanced for!*

O soluție sigură pentru a șterge un element dintr-o colecție presupune utilizarea metodei `void remove()` a unui iterator atașat unei colecții. Această metodă default este definită în interfața `Iterator` și permite ștergerea elementului curent (elementul referit de iterator):

```
Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
    Integer number = itr.next();
    if (number % 2 == 0)
        itr.remove();
}
```

În concluzie, ștergerea unui element dintr-o colecție se poate realiza folosind metodele `remove` definite în colecția respectivă, dacă aceasta nu este parcursă într-o manieră *fail fast iterator*, sau utilizând metoda `remove` din interfața `Iterator`, în caz contrar.

JAVA DATABASE CONNECTIVITY (JDBC)

O modalitate de a se asigura persistența datelor în cadrul unei aplicații o reprezintă utilizarea unei *baze de date*. O bază de date este gestionată de un sistem de gestiune a bazelor de date (SGBD) dedicat, de obicei aflat pe un server, astfel încât baza de date poate fi utilizată, în mod independent și transparent, de mai multe aplicații, posibil implementate în limbiage de programare diferite.

Java DataBase Connectivity (JDBC) este un API dedicat accesării bazelor de date din cadrul unei aplicații Java, care permite conectarea la un server de baze de date, precum și executarea unor instrucțiuni SQL. Accesarea unei baze de date din cadrul unei aplicații Java se realizează într-o manieră transparentă, independentă de sistemul de gestiune al bazelor de date utilizat. Practic, pentru fiecare SGBD există un driver dedicat (un program instalat local) care transformă cererile efectuate din cadrul programului Java în instrucțiuni care pot fi înțelese de către SGBD-ul respectiv (Fig. 1). Există mai multe tipuri de drivere disponibile, însă, în prezent, cele mai utilizate sunt *driverele native Java*. Acestea sunt scrise complet în limbajul Java și folosesc socket-uri pentru a comunica direct cu o bază de date, obținându-se astfel o performanță ridicată din punct de vedere al timpului de executare.

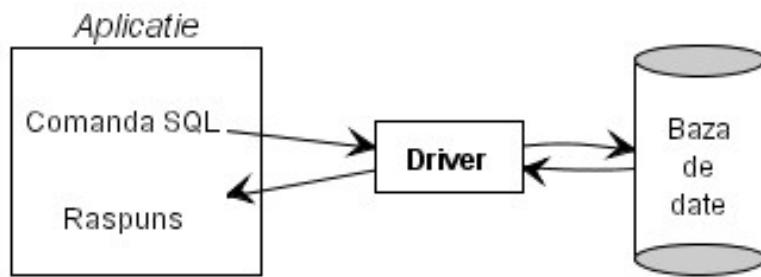


Figura 1. Comunicarea dintre o aplicație Java și un SGBD (https://profs.info.uaic.ro/~acf/java/slides/ro/jdbc_slide.pdf)

Procedura de instalare a unui drivere poate fi diferită de la un driver la altul. De exemplu, în cazul driverului MySQL, driverul este o arhivă de tip jar. Într-un mediu de dezvoltare, driverul poate fi specificat sub forma unei biblioteci atașată proiectului sau poate fi deja disponibil (de exemplu, versiunile noi de NetBeans conțin suport implicit pentru MySQL). Astfel, JDBC dispune de clasa `DriverManager` care administrează încărcarea driverelor, precum și obținerea conexiunilor către baza de date (Fig. 2). Odată conexiunea deschisa, JDBC oferă clientului un API care nu depinde de softul de baze de date folosit, ceea ce facilitează eventuale migrări între diferite SGBD-uri. Cu alte cuvinte, nu este necesar să scriem un program pentru a accesa o bază de date Oracle, alt program pentru a accesa o bază de date Sybase etc., ci este suficient să scriem un singur program folosind API-ul JDBC, iar acesta va fi capabil să comunice cu drivere diferențiate, trimițând secvențe SQL către baza de date dorită.

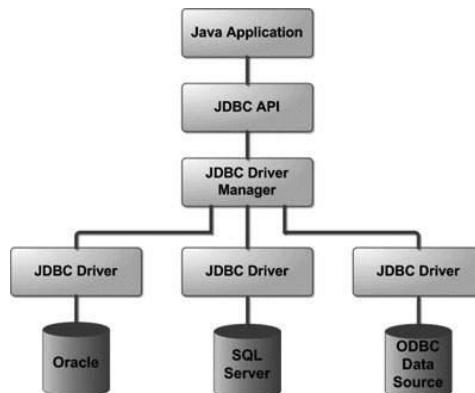


Figura 2. JDBC Driver Manager (https://www.tutorialspoint.com/jdbc/jdbc_introduction.htm)

Arhitectura JDBC

Nucleul JDBC conține o serie de clase și interfețe aflate în pachetul `java.sql`, precum:

- Clasa `DriverManager`: gestionează driverele JDBC instalate și alegeră driverele potrivite pentru realizarea unei conexiuni la o bază de date;
- Interfața `Connection`: gestionează o conexiune cu o bază de date (orice comandă SQL este executată în contextul unei conexiuni);
- Interfețele `Statement` / `PreparedStatement` / `CallableStatement`: sunt utilizate pentru a executa comenzi SQL în SGBD sau pentru a apela proceduri stocate;
- Interfața `ResultSet`: stochează sub formă tabelară datele obținute în urma executării unei comenzi SQL;
- Clasa `SQLException`: utilizată pentru tratarea erorilor specifice JDBC.

Etapele realizării unei aplicații Java folosind JDBC

1. Încărcarea driver-ului specific

Prima etapă constă din înregistrarea, pe mașina virtuală unde rulează aplicația, a driver-ului JDBC necesar pentru comunicarea cu baza de date respectivă. Acest lucru presupune încărcarea în memorie a clasei care implementează driver-ul și poate fi realizată prin apelul metodei statice `void forName(String driver)` din clasa `Class`. De exemplu, încărcarea unui driver pentru o conexiune cu MySQL se poate realiza prin `Class.forName("com.mysql.jdbc.Driver")`, iar pentru o conexiune cu Oracle prin `Class.forName("oracle.jdbc.OracleDriver")`.

Începând cu versiunea JDBC 4.0, inclusă în Java SE 6, acest pas nu mai este obligatoriu, deoarece, la prima încercare de conectare la o bază de date, mașina virtuală Java va încărca automat toate driver-ele disponibile (pe care le găsește în *class path*).

2. Stabilirea unei conexiuni cu o bază de date

După înregistrarea unui driver JDBC, acesta poate fi utilizat pentru a stabili o conexiune cu o bază de date de tipul respectiv. O *conexiune* (sesiune) la o bază de date reprezintă un context prin care sunt trimise sevențe SQL din cadrul aplicației către SGBD și sunt primite înapoi rezultatele obținute.

Având în vedere faptul ca pot exista mai multe drivere încărcate în memorie, se va specifica, pe lângă un identificator al bazei de date, și driverul care trebuie utilizat. Acest lucru se realizează prin intermediul unei adrese specifice, numită JDBC URL, având formatul `jdbc:sub-protocol:identifier`, unde:

- câmpul `sub-protocol` specifică tipul de driver care va fi utilizat (de exemplu `sqlserver`, `mysql`, `postgresql` etc.);
- câmpul `identifier` specifică adresa unei mașini gazdă (inclusiv un număr de port), numele bazei de date și, eventual, numele utilizatorului și parola sa.

De exemplu, pentru o conexiune cu o bază de date denumită BD, care este stocată local folosind SGBD-ul MySQL, poate fi utilizat URL-ul de tip JDBC `jdbc:mysql://localhost:3306/BD`, iar dacă s-ar utiliza SGBD-ul Oracle, atunci URL-ul ar putea fi `jdbc:oracle:thin:@localhost:1521:BD`.

Deschiderea unei conexiuni se realizează prin intermediul metodelor statice `Connection getConnection(String url)` sau `Connection getConnection(String url, String user, String password)` din clasa `DriverManager`. Ambele metode returnează un obiect de tip `Connection`, clasă care conține o serie de metode pentru a gestiona conexiunea cu baza de date.

Exemplu:

- Deschiderea unei conexiuni la baza de date Firma, găzduită local utilizând MySQL:

```
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/Firma");
sau
Connection con=DriverManager.getConnection ("jdbc:mysql://localhost:3306/Firma",
                                         "popescuion","12345");
```

- Deschiderea unei conexiuni la baza de date Firma, găzduită local utilizând Apache Derby:

```
Connection con=DriverManager.getConnection("jdbc:derby://localhost:1527/Firma");
sau
Connection con=DriverManager.getConnection ("jdbc:derby://localhost:1527/Firma",
                                         "popescuion","12345");
```

La primirea unui URL de tip JDBC, `DriverManager`-ul va parcurge lista driverelor încărcate în memorie (de exemplu, începând cu JDK 6, driver-ul pentru Apache Derby este încărcat automat), până când unul dintre ele va recunoaște URL-ul respectiv. Dacă nu există nici un driver potrivit, atunci va fi lansată o excepție de tipul `SQLException`, cu mesajul "No suitable driver found for ...".

3. Crearea unui obiect de tip Statement

După realizarea unei conexiuni cu o bază de date, acesta poate fi folosită pentru executarea unor comenzi SQL (interrogarea sau actualizarea bazei de date), precum și pentru extragerea unor informații referitoare baza de date (meta-date).

Obiectele de tip Statement sunt utilizate pentru a executa instrucțiuni SQL (interrogări, actualizări ale datelor sau modificări ale structurii) în cadrul unei conexiuni.

JDBC pune la dispoziția programatorului 3 tipuri de statement-uri, sub forma a 3 interfețe:

- Statement – pentru comenzi SQL simple, fără parametri;
- PreparedStatement – pentru comenzi SQL parametrizate;
- CallableStatement – pentru apelarea funcțiilor sau procedurilor stocate.

Interfața Statement

Crearea unui obiect Statement se realizează apelând metoda Statement createStatement() pentru un obiect de tip Connection: Statement stmt = con.createStatement().

Executarea unei secvențe SQL poate fi realizată prin intermediul următoarelor metode:

- a) metoda ResultSet executeQuery(String sql) – este folosită pentru executarea interrogărilor de tip SELECT și returnează un obiect de tip ResultSet care va conține rezultatul interrogării sub o formă tabelară, precum și meta-datele interrogării (de exemplu, denumirile coloanelor selectate, numărul lor etc.).

Exemplu

Extragerea datelor despre o persoană stocate în tabela Angajati din baza de date Firma:

```
String sql = "SELECT * FROM Angajati";
ResultSet rs = stmt.executeQuery(sql);
```

Pentru a parcurge înregistrările rezultate în urma unei interrogări de tip SELECT, un obiect de tip ResultSet utilizează un cursor, poziționat inițial înaintea primei linii. În clasa ResultSet sunt definite mai multe metode pentru a muta cursorul în cadrul structurii tabelare, în scopul parcurgerii sale: boolean first(), boolean last(), boolean next(), boolean previous(). Toate cele 4 metode întorc valoarea true dacă mutarea cursorului a fost efectuată cu succes sau false în caz contrar.

Pentru a extrage informațiile de pe fiecare linie se utilizează metode de forma TipData getTipData(int coloană) sau TipData getTipData(String coloană), unde TipData reprezintă tipul de dată al unei coloane, iar argumentul coloană indică fie numărul de ordine din cadrul tabelului (începând cu 1), fie numele acesteia.

Exemplu:

Afișarea datelor angajaților stocate în tabela Angajati din baza de date Firma:

```
while(rs.next())
    System.out.println(rs.getString("Nume") + " " + rs.getInt("Varsta") + " " +
        rs.getDouble("Salariu"));
```

- b) metoda `int executeUpdate(String sql)` – este folosită pentru executarea unor interogări SQL de tipul Data Manipulation Language (DML), care permit actualizări ale datelor de tipul UPDATE/INSERT/DELETE, sau de tipul Data Definition Language (DDL) care permit manipularea structurii bazei de date (de exemplu, CREATE/ALTER/DROP TABLE). Metoda returnează numărul de linii modificate în urma efectuării unor interogări de tip DML sau 0 în cazul interogărilor de tip DDL.

Exemplu:

```
String qrySQL = "INSERT INTO Angajati VALUES('1234567890999',
                                              'Albu Ioan', 3210.10)";

sau
String qrySQL = "UPDATE Angajati SET Salariu = 1.10*Salariu
                  WHERE Salariu <= 2500";

sau
String qrySQL = "DELETE FROM Angajati WHERE Nume LIKE 'Geo%';

int n = stmt.executeUpdate(qrySQL);
System.out.println("Au fost modificate " + n + " înregistrări!");
```

Interfața PreparedStatement

Crearea unui obiect de tip PreparedStatement se realizează apelând metoda `PreparedStatement prepareStatement(String sql)` pentru un obiect de tip `Connection` și primește ca argument o instrucțiune SQL cu unul sau mai mulți parametri. Fiecare parametru este specificat prin intermediul unui semn de întrebare (?). Obiectele de tip PreparedStatement sunt utilizate în cazul în care este necesară executarea repetată a unei interogări SQL, eventual cu valori diferite ale parametrilor, deoarece aceasta va fi precompilată, deci se va executa mai rapid.

Exemplu:

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
```

Obiectul `pstmt` conține o instrucțiune SQL precompilată care este trimisă către baza de date, însă pentru a putea fi executată este necesară stabilirea valorilor pentru fiecare parametru. Setarea valorilor parametrilor se realizează prin metode de tip `void setTipData(int index, TipData valoare)`, unde `TipData` este tipul de date corespunzător parametrului respectiv, iar prin argumentele metodei se specifică indexul parametrului (începând de la 1) și valoarea pe care dorim să i-o atribuim.

Exemplu:

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 45);
pstmt.executeUpdate();
```

Executarea unei instrucțiuni SQL folosind un obiect de tip PreparedStatement se realizează apelând una dintre metodele `ResultSet executeQuery()` sau `int executeUpdate()`, asemănătoare cu cele definite pentru un obiect de tip Statement.

Interfața CallableStatement

Această interfață este utilizată pentru executarea subprogramelor atașate unei baze de date, respectiv funcții și proceduri stocate. Diferențele dintre funcții și proceduri stocate sunt următoarele:

- procedurile sunt folosite pentru a efectua prelucrări în baza de date (de exemplu, operații de actualizare), în timp ce funcțiile sunt folosite pentru a efectua calcule (de exemplu, pentru a determina numărul de angajați care au salariul maxim);
- procedurile nu returnează nimic prin numele lor (dar pot returna mai multe valori prin parametrii de intrare- ieșire, într-un mod asemănător funcțiilor de tip `void` din C/C++), în timp ce funcțiile returnează o singură valoare (într-un mod asemănător funcțiilor din C/C++ care returnează un tip de date primitiv, diferit de `void`);
- procedurile pot avea parametrii de intrare, de ieșire și de intrare- ieșire, în timp ce funcțiile pot avea doar parametrii de intrare;
- funcțiile pot fi apelate în proceduri, dar invers nu.

Funcțiile și procedurile stocate sunt utilizate într-o bază de date pentru a efectua calcule sau prelucrări complexe. De exemplu, se poate implementa o funcție stocată care să calculeze profitul mediu adus de contractele pe care o firmă le are cu o altă firmă într-un anumit interval calendaristic sau se poate implementa o procedură stocată care să efectueze prelucrări complexe ale mai multor tabele.

Exemple:

- a) o funcție stocată care calculează suma salariilor tuturor bărbaților sau tuturor femeilor dintr-o firmă, în raport de valoarea parametrului Tip:

```
CREATE FUNCTION totalSalarii(Tip VARCHAR(1)) RETURNS double
BEGIN
    DECLARE total DOUBLE;
    DECLARE aux CHAR;

    IF(Tip = 'B') THEN
        SET aux = '1';
    ELSE
        SET aux = '2';
    END IF;

    SELECT SUM(Salariu) INTO total FROM Angajati WHERE LEFT(CNP, 1) = aux;
    RETURN total;
END
```

- b) o procedură stocată care verifică dacă un angajat există în tabela Angajați (pe baza CNP-ului) și în raport de rezultatul obținut inserează sau actualizează datele sale:

```
CREATE PROCEDURE inserareAngajat(IN CNP VARCHAR(13), IN Nume VARCHAR(45),
                                    IN Salariu DOUBLE, OUT rezultat INT)
BEGIN
    DECLARE cnt INT;
    SELECT COUNT(*) INTO cnt FROM angajati WHERE angajati.CNP = CNP;
```

```

IF(cnt = 0) THEN
    INSERT INTO Angajati VALUES(CNP,Nume,Salariu);
    SET rezultat = 1;
ELSE
    UPDATE Angajati SET Angajati.Nume = Nume, Angajati.Salariu = Salariu
                           WHERE Angajati.CNP =CNP;
    SET rezultat = 2;
END IF;
END

```

Apelarea funcții stocate `totalSalarii` definită mai sus necesită efectuarea următorilor pași:

- se creează un obiect `sfunc` de tip `CallableStatement` folosind un obiect `conn` de tip `Connection`:

```
sfunc = conn.prepareCall("{?=call totalSalarii(?) }");
```

Primul ? reprezintă valoarea returnată de funcție (“parametrul de ieșire”), iar cel dintre paranteze reprezintă parametrul de intrare. Dacă funcția ar fi avut mai mulți parametri de intrare, atunci se punea câte un ? pentru fiecare, de exemplu `funcție(?, ?, ?)`.

- se specifică tipul rezultatului întors de funcție - se spune că “se înregistrează parametrul de ieșire (valoarea returnată de funcție)”:

```
sfunc.registerOutParameter(1, Types.DOUBLE);
```

Valoarea 1 identifică primul ? din apelul metodei `prepareCall` de mai sus!

- se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`, asemănătoare cu cele definite pentru `PreparedStatement`:

```
sfunc.setString(2, "B");
```

Deoarece valoarea 1 identifică valoarea returnată de funcție, parametrii de intrare sunt numerotați de la 2!

- se execută funcția stocată:

```
sfunc.execute();
```

- se preia rezultatul întors de funcția stocată, folosind metode de tipul `getTipData(int index_parametru_de_intrare)`:

```
double total = sfunc.getDouble(1);
```

Valoarea parametrului este 1 deoarece rezultatul întors de funcție se identifică prin numărul de ordine 1!

Apelarea procedurii stocate `inserareAngajat` definită mai sus necesită efectuarea următorilor pași:

- se creează un obiect `sproc` de tip `CallableStatement` folosind un obiect `conn` de tip `Connection`:

```
sproc = conn.prepareCall("{call inserareAngajat(?, ?, ?, ?)}");
```

Semnele de întrebare dintre paranteze reprezintă parametrii de intrare/ieșire/intrare-ieșire ai procedurii.

- se specifică tipurile parametrilor de ieșire ai procedurii:

```
sproc.registerOutParameter(4, Types.DOUBLE);
```

- se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`:

```
sproc.setString(1, "1234567890999");
sproc.setString(2, "Vasilescu Ion");
sproc.setDouble(3, 3333.33);
```

- se execută procedura stocată:

```
sproc.execute();
```

- se preiau eventualele rezultatele întoarse de procedura stocată, folosind metode de tipul `getTipData(int index_parametru_de_ieșire)`:

```
double rezultat = sproc.getInt(4);
```

4. Închiderea unei conexiuni cu o bază de date

Conexiunea cu o bază de date se închide utilizând metoda `void close()` din clasa `Connection`, dacă nu este utilizat un bloc de tip `try-with-resources`.

PROGRAMARE FUNCȚIONALĂ ÎN LIMBAJUL JAVA

Programarea funcțională este o paradigmă de programare creată de John Backus în 1977 ca o alternativă declarativă pentru programarea imperativă utilizată în momentul respectiv. Practic, în *programarea imperativă* un algoritm se implementează utilizând instrucțiuni pentru a descrie detaliat fiecare pas care trebuie efectuat, în timp ce în *programarea declarativă* este specificată doar logica algoritmului, fără a intra în detalii de implementare. De exemplu, folosind cele două paradigmă de programare, suma numerelor naturale cuprinse între 1 și un n dat se poate calcula astfel:

Programare imperativă

```
int s = 0;
for(int i = 1; i <= n; i++)
    s = s + i;
System.out.println(s);
```

Programare declarativă/funcțională

```
int s = IntStream.rangeClosed(1, n).sum();
System.out.println(s);
```

Chiar dacă nu aveți încă suficiente cunoștințe pentru a înțelege în detaliu exemplul dat pentru programarea declarativă, se poate intui destul de ușor modul în care va fi executată secvența de cod respectivă: metoda `rangeClosed` va furniza numerele de la 1 la n (sub forma unui flux asemănător celor de intrare de la fișiere), după care metoda `sum` le va aduna. Foarte simplu, nu? Totuși, trebuie să menționăm faptul că metodele "magice" `rangeClosed` și `sum` sunt implementate folosind programarea imperativă...

Programarea funcțională este o paradigmă de programare de tip declarativ, bazată pe *lambda calculus* creat de Alonzo Church în anul 1936, în care funcțiile și proprietățile acestora sunt utilizate pentru a construi un program, fără a utiliza instrucțiuni de control. Totuși, dacă nu se pot evita anumite operații cu caracter iterativ, în programarea funcțională se preferă implementarea lor într-o manieră recursivă. Astfel, executarea unui program constă în evaluarea unor funcții într-o anumită ordine, într-un mod asemănător operației de compunere. De exemplu, expresia `rangeClosed(1, n).sum()` este chiar compunerea funcției `sum` cu funcția `rangeClosed(1, n)`, ceea ce matematic s-ar scrie `sum(rangeClosed(1, n))`. Mai mult, putem afișa direct rezultatul compunând 3 funcții: `System.out.println(sum(rangeClosed(1, n)))`.

Principiile programării funcționale pot fi implementate într-un limbaj de programare dacă acesta îndeplinește următoarele condiții:

- se pot defini și manipula ușor funcții complexe, care primesc funcții ca parametri sau returnează funcții ca rezultate;
- apelarea de mai multe ori a unei funcții cu aceleași valori ale parametrilor va furniza același rezultat (de exemplu, o metodă `int suma(int x) {return x + this.salariu;}` va returna valori diferite la două apeluri `suma(1000)` dacă între ele valoarea datei membre `salariu` a obiectului curent este modificată);

- apelarea unei funcții nu produce efecte colaterale, adică nu sunt modificate variabile externe și nu se modifică valorile parametrilor funcției (de exemplu, prin apelarea metodei `void suma(int x) { this.salariu = this.salariu + x; }` se vor produce efecte colaterale, deoarece se va modifica valoarea datei membre `salariu` a obiectului curent), deci se recomandă utilizarea obiectelor imutabile și transmiterea parametrilor unei metode prin valoare.

Lambda expresii

Evident, principalul impediment pentru implementarea programării funcționale în limbajul Java l-a constituit faptul că nu se pot defini și manipula ușor funcții complexe, deoarece în limbajul Java nu se pot defini funcții independente (ci doar metode în cadrul unor clase sau metode default în cadrul unor interfețe) și, în plus, utilizarea funcțiilor ca parametri ai unor metode sau ca rezultate furnizate de metode se realizează prin intermediul unui mecanism complicat (vezi exemplul de calcul al unei sume generice din cursul dedicat interfețelor). Astfel, pentru a permite implementarea unor concepte din programarea funcțională în limbajul Java, în versiunea 8 apărută în anul 2014, au fost introduse *lambda expresiile*.

O *lambda expresie* este o funcție anonimă care nu aparține niciunei clase. O lambda expresie are următoarea sintaxă:

```
(lista parametrilor) -> {expresie sau instrucțiuni}
```

Se observă faptul că pentru o lambda expresie nu se precizează tipul rezultatului returnat, acesta fiind dedus automat de compilator!

Exemple:

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

Definirea unei lambda expresii se realizează ținând cont de următoarele reguli de sintaxă:

- lista parametrilor poate fi vidă:

```
() -> System.out.println("Hello lambdas!")
```

- tipul unui parametru poate fi indicat explicit sau poate fi ignorant, fiind dedus din context:

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

- dacă lambda expresia are un singur parametru fără tip, atunci se pot omite parantezele:

```
a -> {return a*a;}
```

- dacă lambda expresia nu conține instrucțiuni, ci doar o expresie, atunci accoladele și instrucțiunea `return` pot fi omise:

```
a -> a*a
(a, b) -> a+b
(x, y) -> {if(x>y) return x; else return y;}
```

Utilitatea lambda expresiilor

În cursul dedicat interfețelor, am văzut cum putem să transmitem o metodă ca parametru al altrei metode, folosind o interfață în cadrul mecanismului de callback. De regulă, interfața respectivă conține o singură metodă, cea pe care dorim să o transmitem ca parametru, și poate să fie implementată diferit, în funcție de context.

O *interfață funcțională* este o interfață care conține o singură metodă abstractă.

Astfel, putem să definim mai multe clase care vor implementa câte o variantă a metodei respective, anonime sau nu, iar instanțele lor vor fi manipulate printr-o referință de tipul interfeței

Exemplul 1:

Reluăm, pe scurt, exemplul de calcul al unei sume generice din cursul dedicat interfețelor:

- Definim interfața funcțională FuncțieGenerică:

```
public interface FuncțieGenerică{
    int funcție(int x);
}
```

- În clasa utilitară Suma definim o metodă care să calculeze suma celor n termeni generici:

```
public class Suma{
    private Suma() { }

    public static int CalculeazăSuma(FuncțieGenerică fg , int n) {
        .....
    }
}
```

- Definim clase care implementează interfața respectivă, oferind implementări concrete ale funcției generice:

```
public class TermenGeneral_1 implements FuncțieGenerică{
    @Override
    public int funcție(int x){ return x; }
}
```

```
public class TermenGeneral_2 implements FuncțieGenerică{
    @Override
    public int funcție(int x){ return x * x; }
}
```

- La apel, metoda CalculeazăSuma va primi o referință de tipul interfeței, dar spre un obiect de tipul clasei care implementează interfața:

```
FuncțieGenerică tgen_1 = new TermenGeneral_1();
int S_1 = Suma.CalculeazăSuma(tgen_1, 10);
```

```
//putem utiliza direct un obiect anonim
int S_2 = Suma.CalculeazăSuma(new TermenGeneral_2(), 10);

//putem utiliza o clasă anonimă
int S_3 = Suma.CalculeazăSuma(new FuncțieGenerică() {
    public int funcție(int x) {
        return (int) Math.tan(x);
    }
}, 10);
```

Observați faptul că este necesară definirea unei clase pentru fiecare implementare concretă a funcției generice, iar utilizarea clasei anonime conduce la un cod destul de greu de urmărit, mai ales dacă metoda ar fi fost una mai complexă. O soluție mult mai elegantă o constituie utilizarea unor lambda expresii:

```
FuncțieGenerica f = x -> x;
int S_1 = Suma.CalculeazaSuma(f, 10);
int S_2 = Suma.CalculeazaSuma(x -> 1/x, 10);
int S_3 = Suma.CalculeazaSuma(x -> Math.tan(x), 10);
```

Astfel, utilizând lambda expresii, codul devine mai scurt (nu mai este necesară definirea claselor TermenGeneral_1 și TermenGeneral_2), mai ușor de urmărit și, foarte important, mult mai ușor scalabil (dacă dorim să calculăm o altă sumă, nu vom fi obligați să mai definim o altă clasă, ci doar vom utiliza altă lambda expresie).

Exemplul 2:

Interfața Comparator este o interfață funcțională, utilizată pentru sortarea colecțiilor de obiecte, care conține o singură metodă abstractă:

```
int compare(Object ob1, Object ob2)
```

De exemplu, pentru a sorta o listă tp cu elemente de tip Persoana (nume, vârstă, salariu) putem să procedăm în mai multe moduri:

- **Soluția "clasică", folosind o clasă anonimă:**

```
Arrays.sort(tp, new Comparator() {
    public int compare(Object p1, Object p2) {
        return ((Persoana)p1).getNumă().
            compareTo(((Persoana)p2).getNumă());
    }
});
```

- **Soluții cu lambda expresii:**

```
Arrays.sort(tp, (Object p1, Object p2) -> ((Persoana)p1).getNumă().
    compareTo(((Persoana)p2).getNumă()));
```

```
Arrays.sort(tp, (p1, p2) -> ((Persoana)p1).getNumă().
    compareTo(((Persoana)p2).getNumă()));
```

```
Arrays.sort(tp, (p1, p2) -> p1.getNumă().compareTo(p2.getNumă()));
```

Descriptori

O lambda expresie nu este de sine stătătoare, ci ea trebuie apelată într-un context care implică o interfață funcțională. Practic, signatura metodei din interfață precizează forma lambda expresiei.

Exemplu:

Considerăm următoarea interfață funcțională:

```
public interface calculSuma{
    long suma(int a, int b);
}
```

Se observă faptul că interfața poate fi asociată cu o lambda expresie de forma `(int, int) -> long`.

În API-ul din Java 8, în pachetul `java.util.function`, au fost introduse mai multe interfețe funcționale numite *descriptori funcționali* pentru a descrie signatura metodei abstracte dintr-o interfață funcțională, deci, implicit, și forma unei lambda expresii care poate fi utilizată pentru a implementa respectiva metodă abstractă.

Principalele interfețe funcționale definite în acest pachet sunt:

- **Predicate<T>** – descrie o metodă cu un argument generic de tip T care returnează true sau false (un predicat).

Interfața conține metoda abstractă `boolean test(T ob)` care evaluează predicatul definit prin lambda expresie.

Exemplu:

Pentru a afișa persoanele din tabloul tp care au cel puțin 30 de ani, definim un predicat criteriu corespunzător și apoi îl aplicăm asupra fiecărui element din tablou pentru a verifica dacă îndeplinește condiția cerută:

```
Predicate<Persoana> criteriu = pers -> pers.getVarsta() >= 30;
for (Persoana p : tp)
    if (criteriu.test(p))
        System.out.println(p);
```

Folosind un predicat, se poate parametriza foarte ușor o metodă care să afișeze persoanele dintr-un tablou care îndeplinesc un anumit criteriu:

```
static void afisare(Persoana[] tp, Predicate<Persoana> criteriu) {
    for(Persoana p : tp)
        if(criteriu.test(p))
            System.out.println(p);
}

afisare(tp , criteriu);
```

În plus, interfața `Predicate` conține și câteva metode default corespunzătoare operatorilor logici `and`, `or` și `negate`.

Exemplu:

Definim o metodă parametrizată pentru afișarea persoanelor dintr-un tablou care îndeplinește simultan două criterii:

```
static void afisare(Persoana[] tp, Predicate<Persoana> criteriu_1,
                     Predicate<Persoana> criteriu_2) {
    for(Persoana p : tp)
        if(criteriu_1.and(criteriu_2).test(p))
            System.out.println(p);
}
```

Definim două predicate corespunzătoare celor două criterii și apelăm metoda `afisare`:

```
Predicate<Persoana> pred_1 = pers -> pers.getVarsta() >= 30;
Predicate<Persoana> pred_2 = pers -> pers.getNume().startsWith("P");

afisare(tp, pred_1, pred_2);
```

De asemenea, putem să apelăm direct metoda `afisare`, fără a mai defini separat cele două predicate:

```
afisare(tp, p -> p.getVarsta() >= 20, p -> p.getNume().startsWith("P"));
```

- **Consumer<T>** – descrie o metodă cu un argument de tip `T` care nu returnează nimic (un consumator, deoarece doar consumă parametrul).

Interfața conține metoda abstractă `void accept(T ob)` care efectuează acțiunea indicată prin lambda expresie.

Exemplu:

Definim o metodă parametrizată pentru a efectua o anumită acțiune asupra persoanelor dintr-un tablou care îndeplinește un anumit criteriu:

```
static void afisare(Persoana[] persoane, Predicate<Persoana> criteriu,
                     Consumer<Persoana> prelucrare) {
    for(Persoana p:persoane)
        if(criteriu.test(p))
            prelucrare.accept(p);
}
```

Definim un criteriu sub forma unui predicator și acțiunea de afișare a numelui persoanei folosind un obiect de tip `Consumer`:

```
Predicate<Persoana> criteriu = pers -> pers.getVarsta() >= 30;
Consumer<Persoana> actiune = pers -> System.out.println(pers.getNume());
```

```
afisare(tp, criteriu, actiune);
```

În plus, interfața Consumer conține și metoda default andThen care permite efectuarea secvențială a mai multor prelucrări.

Exemplu:

Sortăm persoanele din tablou în ordinea crescătoare a vîrstelor și apoi le afișăm:

```
Consumer<Persoana[]> sortare = tablou -> Arrays.sort(tablou,
    (p1, p2) -> p1.getVarsta() - p2.getVarsta());
```

```
Consumer<Persoana[]> afisare = tablou -> {
    for (Persoana aux : tablou)
        System.out.println(aux);
};
```

```
sortare.andThen(afisare).accept(tp);
```

- **Function<T, R>** – descrie o metodă cu un argument de tip T care returnează o valoare de tip R (o funcție de tipul $f: T \rightarrow R$).

Interfața conține metoda abstractă **R apply(T ob)** care returnează rezultatul obținut prin aplicarea operației indicate prin lambda expresie asupra obiectului curent.

Exemplu:

Definim o funcție care calculează cât ar deveni salariul unei persoane după o majorare cu 20%:

```
Function<Persoana, Double> marire = pers -> pers.getSalariu() * 1.2;
```

```
for(Persoana crt:tp)
    System.out.println(crt.getNumar() + " " + marire.apply(crt));
```

În plus, interfața Function conține și metodele default andThen și compose care permit efectuarea secvențială a mai multor prelucrări.

Exemplu:

Definim funcțiile $f(x) = x^2$ și $g(x) = 2x$, după care calculăm $(f \circ g)(x)$ și $(g \circ f)(x)$ în mai multe moduri:

```
Function<Integer, Integer> f = x -> x*x;
Function<Integer, Integer> g = x -> 2*x;
```

```
System.out.println("f ∘ g = " + f.compose(g).apply(2)); //va afișa 16
System.out.println("f ∘ g = " + g.andThen(f).apply(2)); //va afișa 16
System.out.println("g ∘ f = " + g.compose(f).apply(2)); //va afișa 8
System.out.println("g ∘ f = " + f.andThen(g).apply(2)); //va afișa 8
```

- **Supplier<R>** – descrie o metodă fără argumente care returnează o valoare de tip R (un furnizor).

Interfața conține metoda abstractă **R get()** care returnează rezultatul obținut prin aplicarea operației indicate prin lambda expresie.

```
Supplier<Persoana> furnizor = () -> new Persoana("", 0, 0.0);
Persoana p = furnizor.get();
```

Acet tip de metodă este utilizat, de obicei, în cadrul claselor de tip factory.

În afară celor 4 descriptori funcționali fundamentali de mai sus, în pachetul `java.util.function` mai sunt definiți și alți descriptori funcționali suplimentari, obținuți fie prin particularizarea celor fundamentali, fie prin extinderea lor:

- *funcții cu două argumente* (unul de tipul generic T și unul de tipul generic U): `BiPredicate<T, U>, BiFunction<T, U, R> și BiConsumer<T, U>`
- *funcții specialize:*
 - `IntPredicate, IntConsumer`: descriu un predicat și un consumator și un furnizor cu un argument de tip `int` (sunt definite în mod asemănător și pentru alte tipuri de date primitive);
 - `IntSupplier`: descrie un furnizor de valori de tip `int`;
 - `IntFunction<R>, LongFunction<R>, DoubleFunction<R>`: descriu funcții având un parametru de tipul indicat în numele descriptorului, iar rezultatul este de tipul generic R;
 - `ToIntFunction<T>, ToLongFunction<T>, ToDoubleFunction<T>`: descriu funcții având un parametru de tipul generic T, iar rezultatul este de tipul indicat în numele descriptorului;
 - `DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction`: descriu funcții care au tipul argumentului și tipul rezultatului indicate în numele descriptorului
- *operatori:*
 - `interface UnaryOperator<T> extends Function<T, T>`: descrie un operator unar, adică o funcție cu un parametru de tipul generic T care întoarce un rezultat tot de tip T;

Exemplu:

```
UnaryOperator<Integer> sqr = x -> x*x;
System.out.println(sqr.apply(4)); //va afișa 16
```

- `public interface BinaryOperator<T> extends BiFunction<T, T, T>`: descrie un operator binar

Exemplu:

```
BinaryOperator<Integer> suma = (x, y) -> x + y;
System.out.println(suma.apply(4, 5)); //va afișa 9
```

Referințe către metode

Referințele către metode pot fi utilizate în locul lambda expresiilor care conțin doar apelul standard al unei anumite metode.

Exemplu:

Următoarea lambda expresie afișează sirul de caractere primit ca parametru

```
Consumer<String> c = s -> System.out.println(s);
```

și poate fi rescrisă folosind o referință spre metoda `println` astfel:

```
Consumer<String> c = System.out::println;
```

Practic, metoda `println` este referită direct prin numele său, argumentul său fiind dedus în mod automat din apelul de forma `c.accept(un sir de caractere)` !

În funcție de context, există următoarele 4 tipuri de referințe către metode:

- *referință către o metodă statică*: lambda expresia `(args) -> Class.staticMethod(args)` este echivalentă cu `Class::staticMethod`.

Exemplu: lambda expresia `Function<Double, Double> sinus = x -> Math.sin(x)` este echivalentă cu `Function<Double, Double> sinus = Math::sin`.

- *referință către o metodă de instanță a unui obiect arbitrar*: lambda expresia `(obj, args) -> obj.instanceMethod(args)` este echivalentă cu `ObjectClass::instanceMethod`.

Exemplu: lambda expresia `BiFunction<String, Integer, String> subsir = (a, b) -> a.substring(b)` este echivalentă cu `BiFunction<String, Integer, String> subsir = String::substring`.

- *referință către o metodă de instanță a unui obiect particular*: lambda expresia `(args) -> obj.instanceMethod(args)` este echivalentă cu `obj::instanceMethod`. Atenție, în acest caz obiectul particular `obj` trebuie să existe și să fie accesibil din lambda expresie! O lambda expresie poate accesa și variabile locale, dar acestea trebuie să fie efectiv finale, adică fie sunt declarate cu final, fie nu sunt declarate cu final, dar sunt inițializate și apoi nu mai sunt modificate!

Exemplu: considerând obiectul `Persoana p = new Persoana("Ionescu Ion", 35, 1500.5)`, lambda expresia `Supplier<String> numep = () -> p.getNum()` este echivalentă cu `Supplier<String> numep = p::getNum`.

- *referință către un constructor*: lambda expresia `(args) -> new Class(args)` este echivalentă cu `Class::new`.

Exemplu: lambda expresia `Supplier<Persoana> pnoua = () -> new Persoana()` este echivalentă cu `Supplier<Persoana> pnoua = Persoana::new`.

Metoda `forEach`

În interfața `Iterable`, existentă în limbajul Java încă din versiunea 1.5, a fost adăugată în versiunea 8 o nouă metodă denumită `forEach` care permite parcurgerea unei structuri de date. Implementarea implicită a acestei metode este următoarea:

```
default void forEach(Consumer<? super T> action) {
    for (T t : this)
        action.accept(t);
}
```

Practic, metoda `forEach` reprezintă o nouă modalitate de a parurge o colecție, folosind lambda expresiile sau referințele spre metode.

Exemplu:

Considerăm o listă care conține numele unor orașe și prezentăm mai multe modalități de parcurgere a sa:

```
ArrayList<String> listaOrase = new ArrayList<>(Arrays.asList("București",
                                                               "Paris", "Londra", "Berlin", "Roma"));

//accesând direct fiecare element
for (int i = 0; i < listaOrase.size(); i++)
    System.out.println(listaOrase.get(i));

//folosind un iterator
Iterator it = listaOrase.iterator();
while (it.hasNext())
    System.out.println(it.next());

//folosind instrucțiunea enhanced for
for (String oras : listaOrase)
    System.out.println(oras);

//folosind metoda forEach și lambda expresii
listaOrase.forEach((oras) -> System.out.println(oras + " "));

//folosind metoda forEach și referințe spre metode
listaOrase.forEach(System.out::println);
```

STREAM-URI

Un *stream*, aşa cum îi spune numele, este un flux de date, respectiv o secvență de elemente preluate dintr-o sursă care suportă operații de procesare (parcuregere, modificare, ștergere etc.).

Noțiunea de *stream* a fost introdusă în versiunea Java 8 în scopul de a asigura prelucrarea datelor dintr-o sursă de date care suportă operații de procesare, într-o manieră intuitivă și transparentă. În versiunile anterioare, prelucrarea unei surse de date presupune utilizarea unor metode specifice sursei respective: selecția elementelor cu o anumită proprietate se poate realiza prin parcurearea colecției cu ajutorul unei instrucții iterative, operația de sortare se poate efectua folosind metoda `sort` din clasa utilitară `Collections` etc.

De exemplu, să presupunem faptul că se dorește extragerea dintr-o listă a informațiilor despre persoanele care au vârstă cel puțin egală cu 30 de ani și afișarea lor în ordine alfabetică. O soluție pentru o versiune anterioară versiunii 8 este prezentată mai jos:

```
ArrayList<Persoana> lp = new ArrayList<>(); //lista de Persoane
lp.add(new Persoana("Matei", 23));
...
ArrayList<Persoana> ln = new ArrayList<>(); //lista nouă

for(Persoana item: lp)
    if(item.getVarsta()>=30)
        ln.add(item);

Collections.sort(ln, new Comparator<Persoana>()
{
    public int compare(Persoana p1, Persoana p2) {
        return p1.getNume().compareTo(p2.getNume());
    }
});

System.out.println(ln);
```

O soluție pentru o versiune mai mare sau egală cu 8 este prezentată mai jos:

```
lp.stream().filter(p ->p.getVarsta()>=30).
sorted(Comparator.comparing(Persoana::getNume)).forEach(System.out::println);
```

Comparând cele două soluții, se poate observa faptul că utilizarea unui stream asociat unei colecții, alături de metode specifice, lambda expresii sau referințe către metode, conduce la o prelucrare mult mai facilă a datelor dintr-o colecție.

Caracteristicile unui stream

- Stream-urile nu sunt colecții de date (obiecte container), ci ele pot fi asociate cu diferite colecții. În consecință, un stream nu stochează elementele unei colecției, ci doar le prelucrează!
- Prelucrările efectuate asupra unui stream sunt asemănătoare interogărilor SQL și pot fi exprimate folosind lambda expresii și/sau referințe către metode.
- Un stream nu este reutilizabil, respectiv poate fi prelucrat o singură dată. Pentru o altă prelucrare a elementelor aceleiași colecții este necesară operația de asociere a unui nou stream pentru aceeași sursă de date.
- Majoritatea operațiilor efectuate de un stream furnizează un alt stream, care la rândul său poate fi prelucrat. În concluzie, se poate crea un lanț de prelucrări succesive.
- Stream-urile permit programatorului specificarea prelucrărilor necesare pentru o sursă de date, într-o manieră declarativă, fără a le implementa. Metodele utilizate pentru a prelucra un stream sunt implementate în clasa `java.util.stream.Stream`

Etapele necesare pentru utilizarea unui stream

- Crearea stream-ului
- Aplicarea unor operații de prelucrare succesive asupra stream-ului (operații intermediare)
- Aplicarea unei operații de închidere a stream-ului respectiv

În continuare, vom detalia fiecare dintre cele 3 etape necesare utilizării unui stream.

➤ Crearea unui stream

În sine, crearea unui stream presupune asocierea acestuia la o sursă de date. Astfel, în raport cu sursa de date cu care se asociază, un stream se poate crea prin mai multe modalități:

1. *deschiderea unui stream asociat unei colecții*: se utilizează metoda `Stream<T> stream()` existentă în orice colecție:

```
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo"});
Stream<String> stream = words.stream();
```

2. *deschiderea unui stream asociat unei sir de constante*: se utilizează metoda statică cu număr variabil de parametri `Stream.of(T... values)` din clasa `Stream`:

```
Stream<String> stream = Stream.of("hello", "hola", "hallo", "ciao");
```

3. *deschiderea unui stream asociat unei tablou de obiecte*: se poate utiliza tot metoda `Stream.of(T... values)` menționată anterior:

```
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> stream = Stream.of(words);
```

4. *deschiderea unui stream asociat unei tablou de valori de tip primitiv:* se poate utiliza tot metoda Stream.of(T... values), însă vom obține un stream format dintr-un singur obiect de tip tablou (array):

```
int[] nums = {1, 2, 3, 4, 5};
Stream<int[]> stream = Stream.of(nums)
System.out.println(Stream.of(nums).count()); // se va afișa valoarea 1
```

Se poate observa cum metoda Stream.of returnează un stream format dintr-un obiect de tip tablou cu valori de tip int, ci nu returnează un stream format din valorile de tip int memorate în tablou. Astfel, deschiderea unui stream asociat unui tablou cu elemente de tip primitiv se realizează prin apelul metodei stream din clasa java.util.Arrays:

```
int[] nums = {1, 2, 3, 4, 5};
System.out.println(Arrays.stream(nums).count()); // se va afișa valoarea 5
```

5. *deschiderea unui stream asociat cu un sir de valori generate folosind un algoritm specificat:* se poate utiliza metoda Stream.generate(Supplier<T> s) care returnează un stream asociat unui sir de elemente generate după regula specificată printr-un argument de tip Supplier<T>. Metoda este utilă pentru a genera un stream asociat unui sir aleatoriu sau unui sir de constante, cu o dimensiune teoretic infinită:

```
Stream.generate(() -> Math.random()).forEach(System.out::println);
Stream.generate(new Random()::nextDouble).forEach(System.out::println);
```

Dimensiunea maximă a sirului generat poate fi stabilită folosind metoda Stream<T>.limit(long maxSize):

```
Stream.generate(() -> Math.random()).limit(5).forEach(System.out::println);
```

O altă posibilitate constă în utilizarea metodei Stream<T>.iterate(T seed, UnaryOperator<T> f) care returnează un stream, teoretic infinit, asociat sirului de valori obținute prin apeluri successive ale funcției f, iar primul element al sirului este indicat prin argumentul seed:

```
Stream.iterate(1, i -> i * 2).limit(5).forEach(System.out::println);
```

Mai multe modalități de creare a unui stream sunt descrise în paginile <https://www.baeldung.com/java-8-streams> și <https://www.geeksforgeeks.org/10-ways-to-create-a-stream-in-java/>.

➤ Operații intermediare

După crearea unui stream, asupra acestuia se pot aplica operații succesive de prelucrare, cum ar fi: operația de sortare a elementelor după un anumit criteriu, filtrarea lor după o anumită condiție, asocierea lor cu o anumită valoare etc. O operație intermediară aplicată asupra unui stream furnizează un alt stream asupra căruia se poate aplica o altă operație intermediară, obținându-se astfel un sir succesiv de prelucrări de tip *pipeline* (vezi Figura 1 – sursa: <https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-cheat-sheet.html>):

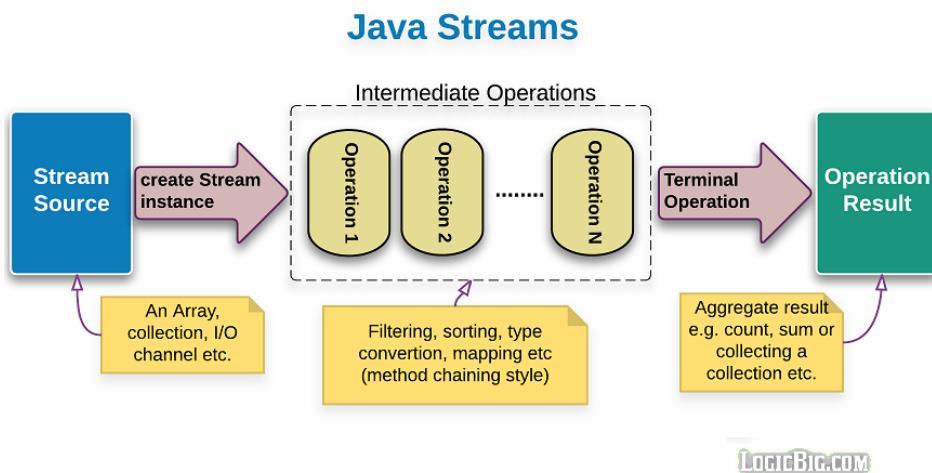


Figura 1. Etapele prelucrării unui stream

Observații:

- Operațiile intermediare nu sunt efectuate decât în momentul în care este invocată o operație de închidere!
- Operațiile intermediare pot fi de tip *stateful*, în care, intern, se rețin informații despre elementele prelucrate anterior (de exemplu: `sort`, `distinct`, `limit` etc.) sau pot fi de tip *stateless*, respectiv nu se rețin informații suplimentare despre elementele prelucrate anterior (de exemplu: `filter`).
- Operațiile intermediare de tip *stateless* pot fi efectuate simultan, prin paralelizare, în timp ce operațiile de tip *stateful* se pot executa doar secvențial.

Pentru prezentarea operațiilor intermediare, vom considera clasa `Persoana`, în care sunt definite câmpurile `String nume`, `int varsta`, `double salariu`, `String profesie` și `String[] competente`, metodele de tip `set/get` corespunzătoare, metoda `toString()` și constructori.

În continuare, sunt prezentate operații intermediare uzuale pentru o colecție cu obiecte de tip `Persoana`. Astfel, vom considera creată o listă de persoane `lp`:

```
List<Persoana> lp = new ArrayList<Persoana>();
lp.add(new Persoana(...));
lp.add(new Persoana(...));
....
```

- Stream<T> **filter**(Predicate<? super T> predicate) – returnează un stream nou, format din elementele stream-ului inițial care îndeplinesc condiția specificată prin argumentul de tip Predicate.

Exemplu:

```
lp.stream().filter(p->p.getVarsta()>=40).forEach(System.out::println);
Într-o operație de filtrare se pot aplica mai multe criterii de selecție, prin utilizarea unor condiții specificate prin mai multe predicate:
```

```
Predicate<Persoana> c1 = p -> p.getVarsta()>=20;
Predicate<Persoana> c2 = p -> p.getSalariu()>=3000;
lp.stream().filter(c1.and(c2.negate())).forEach(System.out::println);
```

- Stream<T> **sorted**(Comparator<? super T> comparator) – returnează un stream nou, obținut prin sortarea stream-ului inițial conform ordinii indicate prin comparatorul specificat prin argumentul de tip Comparator.

Exemplu:

```
lp.stream().sorted((p1,p2) -> p1.getVarsta() - p2.getVarsta()).  
forEach(System.out::println);
```

Începând cu versiunea Java 8, în interfața Comparator a fost inclusă metoda statică comparing care returnează un obiect de tip Comparator creat pe baza unei funcții specificată printr-un argument de tip Function<T>:

```
lp.stream().sorted(Comparator.comparing(Persoana::getVarsta)).  
forEach(System.out::println);
```

În plus, în interfața Comparator a fost introdusă și metoda reversed(), care permite inversarea ordinii de sortare din crescătoare (implicite!) în descrescătoare:

```
lp.stream().sorted(Comparator.comparing(Persoana::getVarsta).reversed()).  
forEach(System.out::println);
```

- Stream<T> **sorted()** – returnează un stream nou, obținut prin sortarea stream-ului inițial conform ordinii naturale a elementelor sale (clasa corespunzătoare elementelor stream-ului, în acest caz clasa Persoana, trebuie să implementeze interfața Comparable).

Exemplu:

```
lp.stream().sorted().forEach(System.out::println);
```

- Stream<T> **limit**(long maxSize) – returnează un stream nou, format din cel mult primele maxSize elemente din stream-ul inițial.

Exemplu:

```
lp.stream().sorted(Comparator.comparing(Persoana::getVarsta).limit(3)).  
forEach(System.out::println);
```

- Stream<T> **distinct()** – returnează un stream nou, format din elementele distincte ale stream-ului inițial. Implicit, elementele sunt comparate folosind hash-code-urile lor, ceea ce poate conduce la valori duplicate dacă în clasa respectivă nu sunt implementate corespunzător metodele hashCode () și equals () !

Exemplu:

```
lp.stream().distinct().forEach(System.out::println);
```

- Stream<R> **map**(Function<T, R> mapper) – returnează un stream nou, cu elemente de un tip R, obținut prin aplicarea asupra fiecărui obiect de tipul T din stream-ul inițial a regulii de asociere specificate prin funcția mapper.

Exemplu: afișarea profesiilor distincte ale persoanelor din lista lp

```
lp.stream().map(Persoana::getProfesie).distinct().  
        forEach(System.out::println);
```

- Stream<R> **flatMap**(Function<T, Stream<R>> mapper) – returnează un stream nou, obținut prin concatenarea stream-urilor rezultate prin aplicarea funcției indicate prin argumentul de tip Function asupra fiecărui obiect de tip T din stream-ul inițial. Astfel, unui obiect din stream-ul inițial i se asociază un stream care poate să fie format dintr-unul sau mai multe obiecte de tip R!

Exemplu: afișarea competențelor distincte ale persoanelor din lista lp

```
lp.stream().flatMap(p->Stream.of(p.getCompetente())).distinct().  
        forEach(System.out::println);
```

➤ **Operații de închidere**

Operațiile de închidere se aplică asupra unui obiect de tip Stream și pot returna fie un obiect de un anumit tip (primitiv sau nu), fie nu returnează nicio valoare (`void`).

- void **forEach**(Consumer< T> action) – operația nu returnează nicio valoare, ci execută o anumită prelucrare, specificată prin argumentul de tip Consumer, asupra fiecărui element dintr-un stream.

Exemplu:

```
lp.stream().forEach(System.out::println);
```

- T **max**(Comparator<T> comparator) – operația returnează valoarea maximă dintre elementele unui stream, în raport cu criteriul de comparație precizat prin argumentul de tip Comparator.

Exemplu: afișarea unei persoane cu vârsta cea mai mare

```
System.out.println(lp.stream().max((p1,p2)->p1.getVarsta()-  
                                         p2.getVarsta()));
```

- T **min**(Comparator<T> comparator) – operația returnează valoarea minimă dintre elementele unui stream, în raport cu criteriul de comparație precizat prin argumentul de tip Comparator.

Exemplu: afișarea unei persoane cu salariul minim

```
System.out.println(lp.stream().max(Comparator.  
comparing(Persoana::getSalariu)));
```

- T **reduce**(T identity, BinaryOperator<T> accumulator) – efectuează o operație de reducere a stream-ului curent folosind o funcție de acumulare asociativă (care indică modul în care se reduc două obiecte într-unul singur) și returnează valoarea obținută prin aplicarea succesivă a funcției de acumulare.

Exemplu: afișarea sumei salariilor tuturor persoanelor din lista lp

```
Double ts = lp.stream().map(Persoana::getSalariu).  
reduce(0.0, (x, y) -> x + y);  
System.out.println("Total salarii: " + ts);
```

- R **collect**(Collector<T,A,R> collector) – efectuează o operație de colectare, specificată prin argumentul de tip Collector, a elementelor asociate stream-ului inițial și poate returna fie o colecție, fie o valoare de tip primitiv sau un obiect.

Clasa Collector cuprinde o serie de metode statice care implementează operații specifice de colectare a datelor, precum definirea unei noi colecții din elementele asociate unui stream, efectuarea unor calcule statistice asupra elementelor asociate unui stream, gruparea elementelor unui stream după o anumită valoare etc., astfel:

- colectorii **toList()**, **toSet()**, **toMap()** returnează o colecție de tipul specificat, formată din elementele asociate unui stream.

Exemplu: construcția unei liste cu obiecte de tip Persoana care au salariul mai mare sau egal decât 3000 RON

```
List<Persoana> lnoua=lp.stream().filter(p->p.getSalariu()>=3000).  
collect(Collectors.toList());
```

- colectorul **joining**(String delimitator) returnează un sir de caractere obținut prin concatenarea elementelor unui stream format din siruri de caractere, folosind sirul delimitator indicat prin parametrul său.

Exemplu:

```
String s = lpers.stream().filter(p -> p.getSalariu()>=3000).  
map(Persoana::getNumar).collect(Collectors.joining(","));
```

- colectorul **counting()** returnează numărul de elemente dintr-un stream.

Exemplu:

```
Long result = givenList.stream().collect(counting());
```

- colectorii **averagingDouble()**, **averagingLong()** și **averagingInt()** returnează media aritmetică a elementelor de tip double, long sau int dintr-un stream.

Exemplu: calcularea salariului mediu al persoanelor din lista lp

```
Double sm = lp.stream().collect(averagingDouble(Persoana::getSalariu));
```

- colectorii **summingDouble()**, **summingLong()** și **summingInt()** returnează suma elementelor de tip double, long sau int dintr-un stream.

Exemplu: calcularea sumei tuturor salariilor persoanelor din lista lp

```
Double st = lp.stream().collect(summingDouble(Persoana::getSalariu));
```

- colectorul **groupingBy()** realizează o grupare a elementelor după o anumită valoare, returnând astfel o colecție de tip Map, ale cărei elemente vor fi perechi de forma <valoare de grupare, lista obiectelor corespunzătoare>.

Exemple:

1. *gruparea persoanelor pe categorii de vârstă*

```
Map<Integer, List<Persoana>> lgv = lp.stream()
                                         .collect(groupingBy(Persoana::getVarsta));
```

2. *suma salariilor pe categorii de vârstă*

```
Map<Integer, Double> lgs = lp.stream().collect(groupingBy(
                                         Persoana::getVarsta, summingDouble(Persoana::getSalariu)));
```

În continuare, vom prezenta o modalitate de afișare a cuvintelor distincte din fișierul text *exemplu.txt* în ordinea descrescătoare a frecvențelor lor:

```
//creăm un stream format din toate liniile
//din fișierul text "exemplu.txt"
Files.lines(Paths.get("exemplu.txt"))
        //creăm un stream în care înlocuim fiecare linie
        //cu un tablou format din cuvintele de pe linia respectivă
        .map(linie -> linie.split("[^\\w]+"))

        //creăm un stream în care înlocuim fiecare tablou de cuvinte
        //cu un stream format din cuvintele din tabloul respectiv
        .flatMap(tablou -> Arrays.stream(tablou))

        //păstrăm doar cuvintele cu lungime nenulă, adică
        //eliminăm liniile vide din fișierul text dat
        .filter(cuvant -> cuvant.length() > 0)

        //grupăm cuvintele și calculăm frecvența fiecărui cuvânt,
        //rezultatul fiind obținut într-un Map cu perechi cheie-valoare de
        //forma (cuvânt, frecvență_cuvânt)
        .collect(Collectors.groupingBy(cuvant -> cuvant,
                                       Collectors.counting()))
```

```
//creăm un nou stream din perechile din Map-ul obținut anterior  
.entrySet().stream()  
  
//sortăm perechile în ordinea descrescătoare a frecvențelor  
.sorted(Map.Entry.<String, Long>comparingByValue().reversed())  
  
//afișam rezultatul  
.forEach(System.out::println);
```

În afara operațiilor de prelucrare (intermediare) și a celor de închidere prezentate anterior, mai există și alte operații de acest tip, pe care le puteți studia în paginile următoare:
<https://javaconceptoftheday.com/java-8-stream-intermediate-and-terminal-operations/> și
<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>.

FIRE DE EXECUTARE

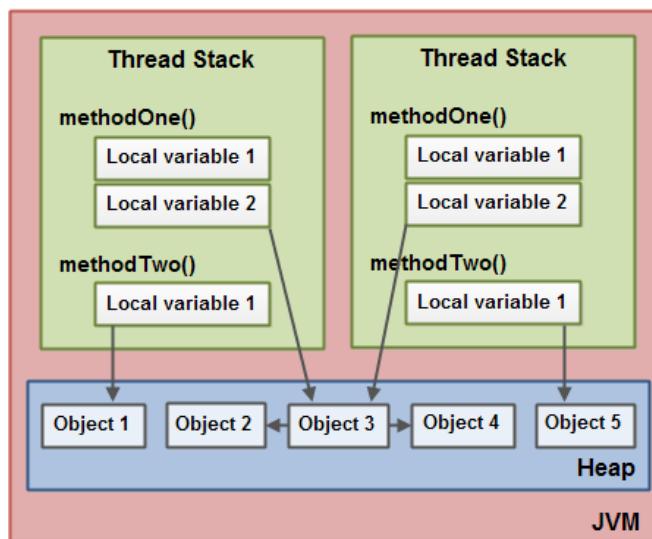
Primele sisteme de operare erau monotasking, respectiv erau capabile să execute un singur program la un moment dat (de exemplu, sistemul de operare MS-DOS). Ulterior, SO au devenit multitasking, respectiv pot rula simultan mai multe programe. Dacă sistemul de calcul este multiprocesor, atunci rularea programelor se realizează efectiv în paralel, iar în cazul sistemelor de calcul monoprocesor rularea lor în paralel este, de fapt, simulată (de exemplu, sistemul de operare Windows).

Un program aflat în executare se numește *proces*. Fiecărui proces îi se asociază un segment de cod, un segment de date și resurse (fișiere, memorie alocată dinamic etc.). Procesele NU execută instrucțiuni, ci doar creează un context în care acestea să fie executate. Unitatea de executare a unui proces este *firul de executare*.

Un *fir de executare* este o succesiune secvențială de instrucțiuni care se execută în cadrul unui proces. Orice proces conține cel puțin un fir de executare principal, din care pot fi create alte fire, care, la rândul lor, pot crea alte fire. Un fir de executare nu poate fi rulat independent, ci doar în cadrul unui proces.

Unui fir de executare îi sunt alocate o secvență de instrucțiuni, un set de registri și o stivă, proprii acestuia. Firele de executare din cadrul aceluiași proces pot accesa, simultan, resursele procesului părinte (memoria heap și sistemul de fișiere).

De exemplu, să presupunem faptul că avem două fire de executare în cadrul aceluiași program (sursa imaginii: <http://tutorials.jenkov.com/java-concurrency/java-memory-model.html>):



Fiecare fir de executare are asociat propriul segment de memorie de tip stivă. Variabilele locale declarate în cadrul metodelor sunt salvate în zone diferite de tip stivă, fiecare zonă fiind asociată unui anumit fir, deci aceste zone nu sunt partajate de către cele două fire.

Obiectele sunt alocate în zona de memorie heap, comună tuturor firelor, dar referințele către obiecte sunt stocate în zona de stivă. Atenție, în cazul în care ambele fire vor acționa simultan asupra unui obiect partajat (Object 3), rezultatele obținute pot fi diferite de la o rulare la alta, deoarece ele nu se execută secvențial!

Procesele rulează în contexte diferite, deci comutarea între ele este mai lentă. În schimb, comutarea între firele de executare din cadrul unui proces este mult mai rapidă.

Utilizarea mai multor fire de executare în cadrul unui program va conduce la creșterea performanțelor, mai ales în sistemele multiprocesor sau multicore. Astfel, o operație de lungă durată din cadrul unui program poate fi executată pe un fir separat, în timp ce alte operații se pot executa pe alte fire, în mod independent (nu sunt blocați). De exemplu, într-un browser, o operație de download se execuță pe un fir separat, astfel încât să nu blocheze.

Utilizarea firelor de executare implică probleme referitoare la sincronizarea lor, accesarea unor resurse comune (excludere reciprocă), comunicarea între ele etc.

Mașina virtuală Java (JVM) își adaptează strategia de multithreading după tipul sistemului de calcul pe care rulează.

Crearea și pornirea firelor de executare

Clasele și interfețele necesare utilizării firelor de executare în limbajul Java sunt incluse în pachetul `java.lang.Thread`.

Un fir de executare poate fi creat prin două metode:

- extinderea clasei `Thread`
- implementarea interfeței `Runnable`

În ambele variante, trebuie redefinită/implementată metoda `void run()`, scriind în cadrul său secvența de cod pe care dorim să o executăm pe un fir separat.

Exemple:

- prin extinderea clasei `Thread`

```
class FirDeExecutare extends Thread {
    .....
    @Override
    public void run() {
        secvența de cod asociată firului de executare
    }
}
.....
FirDeExecutare f = new FirDeExecutare(); // firul nu pornește automat!
f.start(); //trebuie apelată metoda start() care va invoca metoda run() !
```

- prin implementarea interfeței funcționale Runnable

```
class FirDeExecutare implements Runnable {
    .....
    @Override
    public void run() {
        secvența de cod asociată firului de executare
    }
}
.....
FirDeExecutare f = new FirDeExecutare(); // firul nu pornește automat!
Thread t = new Thread(f);
t.start(); //trebuie apelată metoda start() care va invoca metoda run() !
```

Atenție, pentru lansarea unui fir de executare, indiferent de modalitatea în care acesta a fost creat, trebuie apelată metoda `start()`, care mai întâi va crea contextul necesar unui nou fir de executare (stiva proprie, setul de regiștrii etc.) și apoi va executa metoda `run()` în cadrul noului fir. Dacă am apela direct metoda `run()`, atunci aceasta ar fi executată ca o metodă obișnuită, în cadrul firului curent!

Firile se execută concurrent (luptă între ele pentru accesul la resursele comune), motiv pentru care există un arbitru (o componentă a mașinii virtuale Java) numită *planificator (thread scheduler)*. Acesta gestionează memoria, oferind fiecărui fir spațiul de memorie propriu necesar executării și, în plus, selectează firul care se va executa la un moment dat (devine activ), celelalte fiind trecute în aşteptare. Algoritmul de alegere a firului care va deveni activ este dependent de implementarea planificatorului!

Un program se termină când se încheie executarea tuturor firelor lansate din cadrul său.

Exemplu

Clasa `FirDeExecutare` de mai jos utilizează un fir de executare pentru a afișa pe ecran de 100 de ori un caracter `c` primit prin intermediul constructorului clasei:

```
class FirDeExecutare extends Thread
{
    char c;

    public FirDeExecutare(char c)
    {
        this.c = c;
    }

    @Override
    public void run()
    {
        for(int i = 0; i < 100; i++)
            System.out.print(c + " ");
    }
}
```

În clasa `Test_Thread` vom lansa în executare două fire, unul care va afișa cifra 1 și unul care va afișa cifra 2, iar în metoda `main()`, după lansarea celor două fire, vom afișa de 100 de ori cifra 0:

```
public class Test_Thread
{
    public static void main(String[] args)
    {
        FirDeExecutare fir_1 = new FirDeExecutare('1');
        FirDeExecutare fir_2 = new FirDeExecutare('2');

        fir_1.start();
        fir_2.start();

        for(int i = 0; i < 100; i++)
            System.out.print("0 ");
        System.out.println();
    }
}
```

Rulând programul de mai sus de mai multe ori, pe ecran se vor afișa diverse combinații aleatorii formate din cifrele 0, 1 și 2, în care fiecare cifră apare de exact 100 de ori:

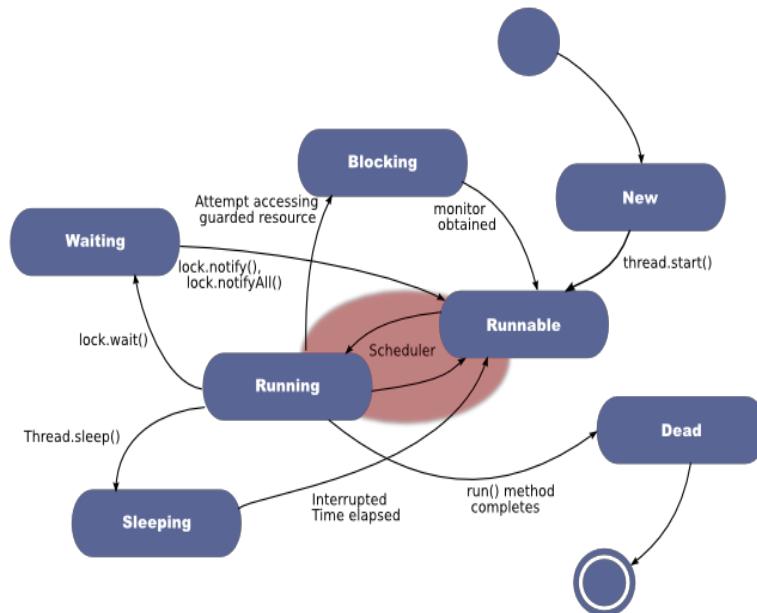
Practic, în acest program sunt executate concurent 3 fire (`fir_1`, `fir_2` și firul principal - cel asociat metodei `main()`), pe care planificatorul le execută într-un mod aleatoriu, astfel: fiecare fir este executat o perioadă de timp, după care el este suspendat și se trece la executarea altui fir, până când toate cele 3 fire își vor încheia executarea. În exemplul de mai sus, se observă faptul că deși firul principal se termină primul (ultimele 4 cifre de 0 sunt afișate pe penultimul rând), executarea programului se încheie doar după ce se termină de executat și celelalte două fire (care afișează cifrele 1 și 2).

Dacă am înlocui instrucțiunile `fir_1.start()` și `fir_2.start()` cu `fir_1.run()` și `fir_2.run()`, pe ecran se va afișa întotdeauna combinația 1 1 ... 1 2 2 ... 2 0 0 ... 0, fiecare cifră de câte 100 de ori, deoarece metoda `run()` nu va fi executată pe un fir separat, ci va fi executată ca o metodă obisnuită, în firul principal al aplicatiei!

Ciclul de viață al fișelor de executare

Din exemplele anterior prezentate, se observă faptul că modul în care sunt selectate/executate firele de către planificator este aleatoriu, deci este necesară utilizarea unor mecanisme specifice programării concurente pentru managementul acestora.

În figura de mai jos, sunt prezentate stările în care se poate afla un fir de executare (sursa imaginii: <http://booxs.biz/EN/java/Threads%20in%20Java.html>):



Din figura de mai sus observăm faptul că un fir de executare se poate afla într-una dintre următoarele stări:

- *fir nou* (new) – obiectul de tip Thread a fost creat;
- *fir rulabil* (runnable) – după apelarea metodei `start()` a firului de executare, acesta este adăugat în grupul de fire aflate în aşteptare (rulabile), deci nu este neapărat executat imediat;
- *fir activ* (running) – firul intră în executare, ca urmare a alegerii sale de către planificator;
- *fir blocat* (waiting/sleeping/blocking) – executarea firului este întreruptă momentan;
- *fir terminat* (dead) – executarea firului s-a încheiat

Există mai multe situații în care dorim ca firul activ să devină inactiv:

- *pentru a da ocazia altor fire să se execute* – se poate utiliza metoda statică `void sleep(long ms)` din clasa `Thread` care suspendă executarea firului curent pentru `ms` milisecunde;
- *pentru a aștepta eliberarea unei resurse partajate* – se poate utiliza metoda `void wait()`, iar firul redevine rulabil după ce un alt fir apelează metoda `void notify()` sau metoda `void notifyAll()` (toate cele 3 metode sunt definite în clasa `Object`!);
- *pentru a aștepta încheierea executării unui alt fir* – se poate utiliza metoda `void join()` care suspendă executarea firului până la terminarea firului curent.

Exemplu:

Vom relua primul exemplu prezentat (în care se afișau pe ecran diverse combinații aleatorii formate din cifrele 0, 1 și 2) și vom modifica doar clasa `Test_Thread`, astfel (liniile de cod adăugate sunt scrise cu font îngroșat):

```
public class Test_Thread
{
    public static void main(String[] args)
    {
        FirDeExecutare fir_1 = new FirDeExecutare('1');
        FirDeExecutare fir_2 = new FirDeExecutare('2');

        fir_1.start();
        fir_2.start();

        try
        {
            fir_1.join();
            fir_2.join();
        }
        catch (InterruptedException ex)
        {
            System.out.println("Eroare fire de executare!");
        }

        for(int i = 0; i < 100; i++)
            System.out.print("0 ");
        System.out.println();
    }
}
```

Cele două apeluri ale metodei `join()` obligă firul părinte (în acest caz, firul principal al aplicației) să aștepte terminarea celor două fire lansate de el în executare (firele `fir_1` și `fir_2`) înainte să-și continue executarea, deci pe ecran se vor afișa diverse combinații aleatorii formate din cifrele 1 și 2 (de câte 100 de ori fiecare), terminate întotdeauna cu exact 100 de cifre de 0:

Un fir de executare poate fi oprit și în mod forțat. Până în versiunea Java 1.5 se putea utiliza metoda `stop()`. Ulterior, din motive de securitate, aceasta a fost considerată ca fiind “depășită”, împreună cu alte două metode, respectiv `suspend()` și `resume()`: <https://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>.

În prezent, pentru oprirea forțată a unui fir de executare, se utilizează una dintre următoarele metode:

- se folosește o variabilă locală, de obicei de tip boolean, pentru a controla executarea codului din interiorul firului, deci codul din metoda `run()`;

Exemplu

Clasa FirDeExecutare de mai jos utilizează un fir de executare pentru a afișa pe ecran numere naturale consecutive în cadrul unei instrucțiuni iterative while controlată de variabila booleană stop:

```
class FirDeExecutare implements Runnable
{
    int cnt;
    boolean stop;

    public FirDeExecutare()
    {
        cnt = 0;
        stop = false;
    }

    @Override
    public void run()
    {
        while(!stop)
            System.out.println(++cnt + " ");
    }

    public void OprireFir() { stop = true; }
}
```

În clasa Test_Stop se va lansa în executare un fir de tipul celui mai sus menționat, iar în firul principal se vor citi siruri de caractere (atenție, pe ecran se vor afișa numere naturale!) până când utilizatorul va introduce cuvântul "stop", după care se va apela metoda OprireFir() pentru a întrerupe forțat executarea firului care afișează numerele naturale:

```
public class Test_Stop
{
    public static void main(String[] args)
    {
        String s , aux;

        FirDeExecutare ob = new FirDeExecutare();
        Thread fir = new Thread(ob);
        fir.start();

        Scanner in = new Scanner(System.in);

        aux = "";
        while ((s = in.next()).compareTo("stop") != 0)
            aux = aux + s + " ";

        ob.OpriseFir();

        System.out.println("Cuvintele citite: " + aux);
    }
}
```

De exemplu, dacă de la tastatură vom introduce, pe rând, cuvintele "un", "exemplu" și "stop", atunci pe ecran se va afișa un text de tipul următor:

```
1 2 3 4 5 ... 2292929 2292930 2292931 2292932 2292933 2292934 2292935 2292936
2292937 2292938 2292939 Cuvintele citite: un exemplu
```

Observație foarte importantă: Dacă variabila care controlează executarea codului din interiorul metodei `run()` nu este locală, atunci ea trebuie declarată (în exteriorul firului) ca fiind volatilă (de exemplu, volatile boolean stop) pentru ca valoarea să fie actualizată din memoria principală la fiecare accesare. Altfel, deoarece fiecare fir de executare are propria sa stivă, este posibil să se utilizeze o copie locală a variabilei externe respective, deși, între timp, valoarea variabilei respective a fost modificată de alt fir! Mai multe informații despre acest subiect găsiți în paginile <http://tutorials.jenkov.com/java-concurrency/volatile.html> și <https://dzone.com/articles/java-volatile-keyword-0>.

- se folosește metoda `void interrupt()` pentru a îintrerupe forțat executarea firului, iar în interiorul metodei `run()` se utilizează metoda statică `boolean interrupted()` din clasa `Thread` pentru a testa dacă firul curent a fost îintrerupt sau nu.

Exemplu

Vom relua exemplul anterior folosind metodele menționate mai sus:

```
class FirDeExecutare implements Runnable
{
    int cnt;

    public FirDeExecutare()
    {
        cnt = 0;
    }

    @Override
    public void run()
    {
        while(!Thread.interrupted())
            System.out.println(++cnt + " ");
    }
}

public class Test_Interrupt
{
    public static void main(String[] args)
    {
        String s , aux;

        FirDeExecutare ob = new FirDeExecutare();
        Thread fir = new Thread(ob);
        fir.start();
    }
}
```

```

Scanner in = new Scanner(System.in);

aux = "";
while ((s = in.next()).compareTo("stop") != 0)
    aux = aux + s + " ";

fir.interrupt();

System.out.println("Cuvintele citite: " + aux);
}
}
}

```

Accesarea concurentă a unor resurse comune

Mai devreme, am văzut faptul că mai multe fire pot să partajeze o resursă comună. Un exemplu concret îl reprezintă vânzarea on-line a unor bilete de tren, folosind o aplicație client-server care utilizează o bază de date comună pentru a reține locurile vândute. Evident, există posibilitatea ca, într-un anumit moment, mai mulți operatori să vândă același loc, ceea ce reprezintă o eroare gravă! Evident, în acest caz resursa comună este baza de date, iar operația care poate să conducă la rezervarea de mai multe ori a aceluiași loc este cea de vânzare, deci aceasta este o secțiune critică.

O *secțiune critică* este o secvență de cod care gestionează o resursă comună mai multor de fire de executare care acționează simultan.

Pentru a rezolva o problema de sincronizare de tipul celei precizate anterior, trebuie ca secțiunea critică să se execute prin excludere reciprocă, adică în momentul în care un fir acționează asupra resursei comune, restul firelor vor fi blocați. Astfel, în exemplul de mai sus, în momentul vânzării unui anumit loc de către un operator, toți ceilalți operatori care ar dori să vândă același loc vor fi blocați.

Controlul accesului într-o secțiune critică (la o resursă comună) se face folosind cuvântul cheie `synchronized`.

Exemplu:

Mai întâi, vom considera o clasă `Counter` care implementează un simplu contor:

```

class Counter
{
    private long count;

    Counter() { count = 0; }

    public long getCount() { return count; }

    public void add() { count++; }
}

```

De asemenea, vom considera o clasă `CounterThread` care incrementează de 10000 de ori un contor de tipul `Counter`, utilizând un fir de executare dedicat:

```

class CounterThread extends Thread
{
    private Counter counter = null;

    public CounterThread(Counter counter)
    {
        this.counter = counter;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 10000; i++)
            counter.add();
    }
}

```

În continuare, vom considera o clasă CounterThread care incrementează de 10000 de ori un contor de tipul Counter, utilizând un fir de executare dedicat:

```

class CounterThread extends Thread
{
    private Counter counter = null;

    public CounterThread(Counter counter)
    {
        this.counter = counter;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 10000; i++)
            counter.add();
    }
}

```

În metoda main() a clasei Test_Sincronizare, mai întâi vom crea un contor counter și apoi două fir de executare, fir_1 și fir_2, care vor accesa contorul comun counter:

```

public class Sincronizare
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter counter = new Counter();

        Thread thread_1 = new CounterThread(counter);

        Thread thread_2 = new CounterThread(counter);

        thread_1.start();
        thread_2.start();
    }
}

```

```

        thread_1.join();
        thread_2.join();

        System.out.println("Counter: " + counter.getCount());
    }
}

```

Observați faptul că am apelat metoda `join()` pentru ambele fire de executare, astfel încât în firul principal al aplicației valoarea contorului să fie afișată doar după ce sunt încheiate executările ambelor fire! Rulând programul de mai multe ori, se va afișa, de obicei, o valoare strict mai mică decât valoarea 20000 pe care o anticipam. Acest lucru se întâmplă deoarece, în mai multe momente de timp, ambele fire vor încerca să incrementeze contorul comun (evident, fără a reuși), ceea ce va conduce la o valoare finală eronată a contorului.

Pentru a rezolva această problemă, trebuie să realizăm incrementarea contorului (în metoda `add()` din clasa `Counter`) sub excludere reciprocă, respectiv în momentul în care un fir de executare incrementează contorul, celălalt fir să fie suspendat și abia după ce primul fir termină operația de incrementare a contorului să se reia executarea celui de-al doilea fir.

Pentru asigurarea excluderii reciproce, putem să utilizăm cuvântul cheie `synchronized` în două moduri:

- *la nivel de metodă*, adăugând cuvântul cheie `synchronized` în antetul metodei `add()`:

```

synchronized public void add()
{
    count++;
}

```

- *la nivel de bloc de instrucțiuni*, adăugând cuvântul cheie `synchronized` doar pentru secțiunea critică:

```

public void add()
{
    synchronized(this)
    {
        count++;
    }
}

```

Dacă se apelează o metodă nestatică sincronizată pentru un obiect, atunci alte fire nu mai pot apela, pentru același obiect, nicio altă metodă nestatică sincronizată.

Dacă se apelează o metodă statică sincronizată pentru un obiect, atunci alte fire nu mai pot apela, pentru nici un alt obiect al clasei respective, nicio altă metodă statică sincronizată. Practic, în acest caz, sincronizarea se realizează la nivel de clasă, ci nu de obiect!

În ambele cazuri prezentate mai sus, alte fire pot apela metode nesincronizate. De asemenea, metodele nestaticice și cele statice se sincronizează în mod diferit, deci nu sunt sincronizate unele cu celelalte!

Utilizarea mai multor metode sincronizate va conduce la un timp de executare mare. Pentru a evita acest lucru, dacă o metodă conține doar un bloc de instrucțiuni care necesită sincronizare (o secțiune critică), atunci se va sincroniza doar blocul respectiv. În acest caz, alte fire pot invoca și alte metode, sincronizate sau nu!

În practică, sunt multe situații în care firele de executare nu trebuie doar să se excludă reciproc, ci să și coopereze. În acest scop, există metode specifice definite în clasa `Object`, respectiv metodele `wait()` și `notify() / notifyAll()`. Pentru a înțelege modalitatea de utilizare a acestor metode, vom considera faptul că asupra unui obiect acționează mai multe fire de executare și unul dintre fire preia controlul exclusiv asupra obiectului, însă nu-și poate finaliza acțiunea fără ca un alt fir să execute o acțiune suplimentară. În acest caz, firul respectiv se va auto-suspenda, folosind metoda `wait()`, trecând astfel într-o stare de așteptare și eliberând controlul asupra obiectului partajat. În acest moment, un alt fir poate prelua controlul exclusiv asupra obiectului pentru a efectua acțiunea suplimentară, iar după efectuarea acesteia, firul respectiv va întări firul sau firele aflate în așteptare, folosind metodele `notify()` sau `notifyAll()`, despre faptul că a efectuat o acțiune. Aceste două metode, spre deosebire de metoda `wait()`, nu vor ceda imediat controlul asupra obiectului, respectiv, dacă în codul firului respectiv mai există alte instrucțiuni după apelul unei dintre cele două metode, acestea vor fi executate și abia apoi va fi cedat controlul asupra obiectului partajat.

Problema producător-consumator este un exemplu clasic în care este evidențiată necesitatea de colaborare între două fire de executare: "*Un producător și un consumator își desfășoară simultan activitățile, folosind în comun o bandă de lungime fixă. Producătorul produce câte un obiect și îl plasează la un capăt al benzii, iar consumatorul preia câte un obiect de la celălalt capăt al benzii și îl consumă.*"

În simularea activităților producătorului și consumatorului, trebuie să ținem cont de faptul că producătorul și consumatorul plasează/preiau obiecte pe/de pe banda comună în ritmuri aleatorii, ceea ce poate conduce la următoarele situații limită:

- *producătorul încearcă să plaseze un obiect pe banda plină* – în acest caz producătorul trebuie să se auto-suspende, folosind metoda `wait()`, până în momentul în care consumatorul va prelua cel puțin un obiect de pe banda plină;
- *consumatorul încearcă să preia un obiect de pe banda vidă* – în acest caz consumatorul trebuie să se auto-suspende, folosind metoda `wait()`, până în momentul în care producătorul va plasa cel puțin un obiect pe banda vidă.

După fiecare acțiune reușită de plasare/preluare a unui obiect pe/de pe bandă, producătorul/consumatorul va apela metoda `notify()` pentru a permite deblocarea unui eventual consumator/producător suspendat.

Pentru a implementa această problemă în limbajul Java, vom defini mai întâi clasa `BandaComună`, considerând obiectele ca fiind numere naturale nenule:

```
class BandaComună
{
    private LinkedList<Integer> banda;
    private int dimMaximăBandă;
```

```

public BandaComună(int dimMaximaBanda)
{
    banda = new LinkedList();
    this.dimMaximăBandă = dimMaximaBanda;
}

public synchronized void PlaseazăObiect(int x) throws InterruptedException
{
    while (banda.size() == dimMaximăBandă)
        wait();

    banda.add(x);
    System.out.println("Producător: obiect " + x);
    notify();
}

public synchronized void PreiaObiect() throws InterruptedException
{
    while (banda.size() == 0)
        wait();

    int x = banda.remove(0);
    System.out.println("Consumator: obiect " + x);
    notify();
}
}

```

Observați faptul că operațiile de plasare/preluare a unui obiect pe/de pe banda comună se execută sub excludere reciprocă! De ce a fost nevoie de aceste restricții?

În continuare, definim clasa Producător, în cadrul căreia vor fi produse 10 obiecte, identificate prin numerele naturale de la 1 la 10:

```

class Producător extends Thread
{
    private BandaComună banda;

    public Producător(BandaComună banda) { this.banda = banda; }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
            try
            {
                Thread.sleep((int) (Math.random() * 100));
                banda.PlaseazăObiect(i);
            }
            catch (InterruptedException e) {}
    }
}

```

Pentru a simula mai bine ritmul aleatoriu de plasare a obiectelor pe bandă, am întârziat operația respectivă cu un număr aleatoriu cuprins între 0 și 99 de milisecunde, utilizând metoda `sleep`.

Într-un mod asemănător definim și clasa `Consumator`:

```
class Consumator extends Thread
{
    private BandaComună banda;

    public Consumator(BandaComună banda)
    {
        this.banda = banda;
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            try
            {
                Thread.sleep((int) (Math.random() * 100));
                banda.PreiaObiect();
            }
            catch (InterruptedException ex) {}
        }
    }
}
```

În clasa `Producător_Consumator` vom simula efectiv activitatea producătorului și consumatorului, utilizând o bandă comună de lungime 5:

```
public class Producător_Consumator
{
    public static void main(String[] args)
    {
        BandaComună b = new BandaComună(5);

        Producător p = new Producător(b);
        Consumator c = new Consumator(b);

        p.start();
        c.start();
    }
}
```

Rulând programul, vom obține diverse variante de simulare a activităților producătorului și consumatorului, una dintre ele fiind următoarea:

```
Producător: obiect 1
Consumator: obiect 1
Producător: obiect 2
Producător: obiect 3
```

```

Producător: obiect 4
Consumator: obiect 2
Consumator: obiect 3
Producător: obiect 5
Consumator: obiect 4
Producător: obiect 6
Consumator: obiect 5
Producător: obiect 7
Consumator: obiect 6
Producător: obiect 8
Consumator: obiect 7
Producător: obiect 9
Consumator: obiect 8
Producător: obiect 10
Consumator: obiect 9
Consumator: obiect 10

```

Încheiem precizând faptul că programarea folosind fire de executare este un domeniu important și actual al informaticii, tratat pe larg în cadrul unor discipline precum calcul paralel și concurrent sau calcul distribuit.

SOCKET-URI

Programarea cu socket-uri se referă la posibilitatea de a transmite date între două sau mai multe calculatoare interconectate prin intermediul unei rețele.

Modelul utilizat pe scară largă în sistemele distribuite este sistemul Client-Server, care constă din:

- o mulțime de procese de tip server, fiecare jucând rolul de gestionar de resurse pentru o colecție de resurse de un anumit tip (baze de date, fișiere, servicii Web, imprimantă etc.);
- o mulțime de procese de tip client, fiecare executând activități care necesită acces la resurse hardware/software disponibile, prin partajare pe servere.

Servele sunt cele care își încep primele activități, oferind clienților posibilitatea de a se conecta la ele (spunem că acceptă conexiuni de la clienți).

Un client își manifestă dorința de a se conecta și, dacă serverul este gata să accepte conexiunea, aceasta se realizează efectiv. În continuare, informațiile (datele) sunt transmise bidirectional. Teoretic, activitatea unui server se desfășoară la infinit.

Pentru conectarea la un server, clientul trebuie să cunoască adresa serverului și numărul portului dedicat. Un port nu este o locație fizică, ci o extensie software corespunzătoare unui serviciu. Un server poate oferi mai multe servicii, pentru fiecare fiind alocat câte un port.

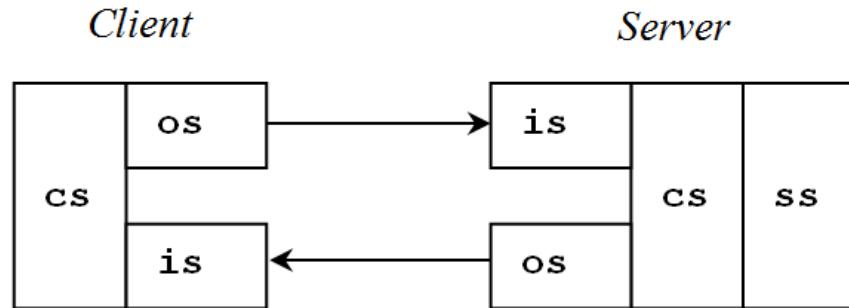
Porturile din intervalul 0...1023 sunt în general rezervate pentru servicii speciale, cum ar fi: 20/21 (FTP), 25 (email), 80 (HTTP), 443(HTTPS) etc.

Cea mai simplă modalitate de comunicare între două calculatoare dintr-o rețea o constituie socket-urile, care folosesc protocolul TCP/IP, în care un calculator se identifică prin IP-ul său.

În pachetul `java.net` sunt definite două clase care pot fi utilizate pentru comunicarea bazată pe socket-uri:

- `ServerSocket` – pentru partea de server;
- `Socket` – pentru partea de client.

Oricărui socket îi sunt atașate două fluxuri: unul de intrare și unul de ieșire. Astfel, comunicarea folosind socket-uri se reduce la operații de scriere/citire în/din fluxurile atașate.



În cazul unui server, mai întâi trebuie creat un socket de tip server, folosind constructorul `ServerSocket(int port)`. Se observă faptul că se poate preciza doar portul care va fi asociat server-ului, IP-ul implicit fiind cel al calculatorului respectiv (din motive de securitate, limbajul Java nu permite crearea unui server "la distanță" (pe un alt calculator) deoarece ar fi posibilă clonarea unui server care, de exemplu, ar putea furniza servicii neautorizate.

După crearea server-ului, se va apela metoda `Socket accept()`, astfel server-ul intrând într-o stare în care așteaptă conectarea unui client. După ce un client s-a conectat, metoda va întoarce un socket de tip client (`Socket`), ale cărui fluxuri vor fi folosite pentru comunicarea bidirectională.

Fluxurile asociate unui socket se pot prelua folosind următoarele metode:

- `InputStream getInputStream();`
- `OutputStream getOutputStream();`

Închiderea unui socket se realizează folosind metoda `void close()`.

Exemplu:

Vom prezenta un program foarte simplu de tip chat, care permite transmiterea unor mesaje între 2 utilizatori, până când clientul va transmite mesajul "STOP". Implementarea server-ului este următoarea:

```

public class ChatServer
{
    public static void main(String[] sir) throws IOException
    {
        ServerSocket ss = null;
        Socket cs = null;

        Scanner sc = new Scanner(System.in);

        System.out.print("Portul: ");

```

```

//instantiem server-ul
int port = sc.nextInt();
ss = new ServerSocket(port);
sc.nextLine();

System.out.println("Serverul a pornit!");

//server-ul aşteaptă un client să se conecteze
cs = ss.accept();

System.out.println("Un client s-a conectat la server!");

//server-ul preia fluxurile de la/către client
DataInputStream dis = new DataInputStream(cs.getInputStream());
DataOutputStream dos = new DataOutputStream(cs.getOutputStream());

//citim linia de text transmisa de către client și o afișăm,
//după care citim o linie și o transmitem clientului
//chat-ul se închide când clientul transmite cuvântul STOP
while(true)
{
    String linie = dis.readUTF();
    System.out.println("Mesaj receptionat: " + linie);
    if (linie.equals("STOP"))
        break;
    System.out.print("Mesaj de trimis: ");
    linie = sc.nextLine();
    dos.writeUTF(linie);
}

dis.close();
dos.close();
cs.close();
ss.close();
}
}

```

În cazul unui client, se va încerca realizarea unei conexiuni cu un server chiar în momentul creării unui socket de tip client, folosind constructorul `Socket(String adresa_server, int port)`. În cazul în care conexiunea este realizată, se vor prelua fluxurile asociate socket-ului și se vor utiliza pentru comunicarea bidirectională.

Exemplu:

Implementarea clientului de chat este următoarea:

```

public class ChatClient
{
    public static void main(String[] sir) throws IOException
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Adresa serverului: ");
        String adresa = sc.next();
        System.out.print("Portul serverului: ");

```

```

int port = sc.nextInt();
sc.nextLine();

//conectarea la server
Socket cs = new Socket(adresa, port);
System.out.println("Conectare reusita la server!");

//preluăm fluxurile de intrare/ieșire de la/către server
DataInputStream dis = new DataInputStream(cs.getInputStream());
DataOutputStream dos = new DataOutputStream(cs.getOutputStream());

//citim o linie de text de la tastatură și o transmitem server-ului,
//după care așteptam răspunsul server-ului
//chat-ul se închide tastând cuvântul STOP

while(true)
{
    System.out.print("Mesaj de trimis: ");
    String linie = sc.nextLine();
    dos.writeUTF(linie);
    if (linie.equals("STOP"))
        break;
    linie = dis.readUTF();
    System.out.println("Mesaj receptionat: " + linie);
}

cs.close();
dis.close();
dos.close();
}
}

```

Încheiem prezentarea acestui exemplu precizând faptul că prima dată clientul trebuie să transmită un mesaj către server, iar apoi server-ul și clientul trebuie să își vor transmită unul altuia, pe rând, mesaje.