**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 subclaselor ce pot extinde o superclasă sau a claselor ce 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 subclaselor care pot să implementeze clasa respectivă se vor scrie la sfârșitul antetului său, așa 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 subclaselor 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 subclaselor 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 subclaselor 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 modelează într-un mod complet și sigur un anumit concept.

**ENUMERĂRI**

O **enumerare** este un tip de data de referința care poate încapsula o un set de constante.

* Sintaxa

public enum denumire

{

instante ale enumerarii

[camp privat care retine valoarea unei constante]

[constructor privat care instantiaza o referinta enum]

[o metoda care returneaza valoarea unei referinte]

}

* Exemple
* Enumerare constante fără valori asociate

public enum Level {

HIGH,

MEDIUM,

LOW;

}

Accesarea unei constante se sealizează prin numele referinței:

Level level = Level.HIGH

* Enumerare constante cu valori asociate

public enum Saptamana

{

LUNI(1), MARTI(2), MIERCURI(3), JOI(4), VINERI(5), SAMBATA(6), DUMINICA(7);

private final int zi;

private Saptamana ()

{

this.zi = 0;

}

private Saptamana (int zi)

{

this.zi = zi;

}

public int getValue()

{

return zi;

}

}

public class Test\_enumerari

{

public static void main(String[] args)

{

Enumerare f = Enumerare.DUMINICA;// 7

System.out.println(f.getValue());

}

}

* Observații
* Orice enumerare este extinsă din clasa java.lang.Enum care conține o serie de metode specifice unui tip de data de referintă enum, precum:
* String name(): returnează numele unei instante a enumerării, stabilit la declararea sa
* int ordinal(): returnează numărul de ordine al unei insatnțe a enumerării (prima instanță este indexată cu 0)
* String toString(): returnează o reprezentare sub forma unui șir de caracter pentru o instanță a enumerării
* static T valueOf(Class<T> enumType, String name)

: returnează valoarea unei instanțe a enumerării

* Se poate obține o structură de date care să conțină toate valorile constantelor prin apelul metidei values()

for (Saptamana level : Saptamana.values()) {  
 System.out.println(level);  
}

* O enumerare poate să încapsuleze metode statice:

public static Saptamana getZiByValue(int x) {  
 for (Saptamana item **: Saptamana.values()**) {  
 if (x == item.getValue())  
 return item;  
 }  
 return null;  
}

* O enumerare poate să încapsuleze o metodă abstractă, în acest caz fiecare instanță enumerării trebuie să implementeze metoda abstractă

public enum States {  
 Comanda(1)  
 {  
 public States getNextState()  
 {  
 return PlasareComanda;  
 }  
 },  
 PlasareComanda(2)  
 {  
 public States getNextState()  
 {  
 return LivrareComanda;  
 }  
 },  
 LivrareComanda(3)  
 {  
 public States getNextState()  
 {  
 return Comanda;  
 }  
 };  
  
 private int state;  
 private States(int x)  
 {  
 this.state =x;  
 }  
  
 **public abstract States getNextState();**  
}

* Pentru o instanță a unei enumerări se pot asocia mai multe valori

**EXCEPȚII**

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 ș.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 "AcțiuneButon", pentru a trata evenimentul, care la rândul său apelează o metodă "CalculStatistică" dintr-o altă clasă, iar aceasta, la rândul său, apelează o metodă "ÎncărcareDateDinFișier". 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*(exemple: 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* (exemple: 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).

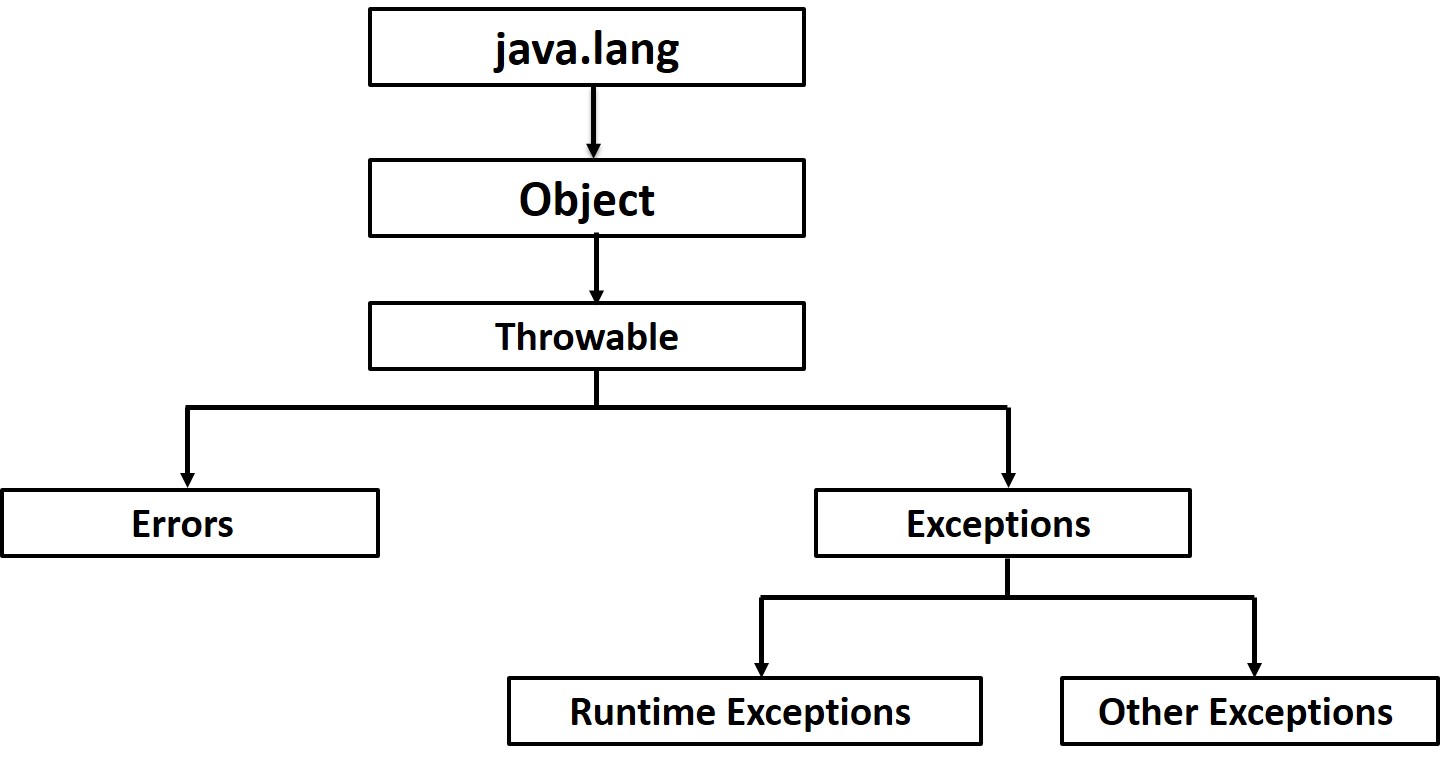


Fig. 1 - Ierarhia de clase pentru tratarea excepțiilor

Se poate observa cum există o multitudine de tipuri derivate din Exception sau RuntimeException*,* distribuite în diverse pachete Java. De regulă, excepțiile nu grupate într-un singur pachet (nu există un pachet java.exception), ci sunt definite în aceleași pachete cu clasele care le generează. De exemplu, IOException este definită în java.io*,* AWTExceptionînjava.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 subclasă 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 incorect, respectiv negativ sau strict mai mare decât dimensiunea fizică a unui tablou - 1;
* ArithmeticException - operații aritmetice nepermise, precum împărțirea unui număr întreg la 0;
* IllegalArgumentException – utilizarea incorectă a unui argument pentru o metodă. O subclasă a clasei IllegalArgumentException este NumberFormatException care corespunde erorilor de conversie a unui String într-un tip de date primitiv din cadrul metodelor parseTipPrimitivale 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.

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 subclasă 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 blocuri 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 mai 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;
* dacăîn fișierul numere.txt sunt valorile 13 0, atunci se lansează și se tratează ArithmeticException, fără a se mai 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 exista un bloc catch corespunzător, acesta va fi executat după întreruperea executării blocului try*,* urmat de bloculfinally;
  + dacă nu exisă un bloc catch*,* atunci se execută bloculfinally imediat dupa blocul try, după care JVM caută 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(TipExceptie), 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 sa 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 forțam 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, așa cum este fluxul dout din exemplul de mai sus, JVM va închide automat și fluxul primitiv pe bază căruia acesta a fost deschis!