



## Programare orientată pe obiecte

1. Testarea programelor
2. Depanarea programelor
3. Introducere în I/E Java



## Testarea

- *Testarea software* : procesul folosit la identificarea corectitudinii, completitudinii, securității și calității software
- *Testarea funcțională* : determină dacă sistemul satisface specificațiile clientului.
- *Testarea tip cutie neagră*:
  - Proiectantul testelor ignoră structura internă a implementării.
  - Testul este condus de comportamentul extern așteptat al sistemului
  - Sistemul este tratat ca o "cutie neagră": comportamentul este observabil, dar structura internă nu este cunoscută.



## Proiectarea, planificarea și testarea cazurilor

- Proiectarea testelor începe de obicei cu analiza
  - Specificațiilor funcționale ale sistemului și a
  - Cazurilor de utilizare: a modurilor în care va fi folosit sistemul.
- Un caz de testare este definit de
  - Declararea obiectivelor cazului;
  - Setul de date pentru caz;
  - Rezultatele așteptate.
- Un plan de teste este un set de cazuri de testare. Pentru a-l dezvolta:
  - Analizăm caracteristicile pentru a identifica cazurile de test.
  - Considerăm seturile de stări posibile pe care le poate asuma un obiect.
  - Testele trebuie să fie reprezentative.



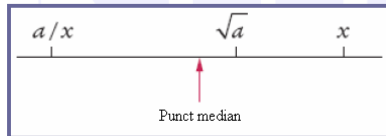
## Testarea unităților

- Cel mai important instrument de testare
- Verifică o singură metodă sau un set de metode care cooperează
- Nu testează întregul program în curs de dezvoltare; testează doar clasele luate izolat
- Pentru fiecare test furnizăm o clasă simplă numită *test harness* (engl. harness = ham, harnașament)
- Test harness alimentează cu parametri metodele care se testează



## Exemplu: Realizarea unui test harnesses

- Pentru a calcula rădăcina pătrată a lui  $a$  folosim un algoritm comun:
  1. Ghicim o valoare a lui  $x$  care poate fi apropiată de rădăcină pătrată dorită ( $x = a$  este ok)
  2. Rădăcina pătrată reală este undeva între  $x$  și  $a/x$
  3. Luăm punctul median  $(x + a/x) / 2$  ca valoare mai bună



- 4. Repetăm procedura. Stop când două valori succesive sunt foarte apropiate una de alta
- Metoda converge repede. Demo: root1
  - Sunt 8 tentative pentru rădăcina pătrată a lui 100:

OOP12 - M. Joldoș - T.U. Cluj

5



## Testarea programului

- Clasa `RootApproximator` funcționează corect pentru toate intrările?  
Trebuie testată cu mai multe valori
- Re-testarea cu alte valori, în mod repetat, nu este o idee bună; testele nu sunt repetabile
- Dacă se rezolvă o problemă și e nevoie de re-testare, e nevoie să ne reamintim toate intrările
- Soluție: scriem test harnesses care să ușureze repetarea testelor de unități

OOP12 - M. Joldoș - T.U. Cluj

6



## Furnizarea intrărilor pentru teste

- Există diverse mecanisme pentru furnizarea cazurilor de test
- Unul dintre acestea este scrierea intrărilor de test în codul test harness ("hardwire").
  - Exemplu: root2/RootAproximationHarness1
- Pur și simplu se execută test harness ori de câte ori se rezolvă o eroare (bug) în clasa care se testează
- Alternativă: să punem intrările într-un fișier

OOP12 - M. Joldoș - T.U. Cluj

7



## Furnizarea intrărilor pentru teste

- Putem genera automat cazurile de testat
- Pentru puține intrări posibile este fezabil să rulăm un număr (reprezentativ) de teste într-un ciclu
  - Exemplu: root2/RootAproximationHarness2
- Testul anterior este restricționat la un subset mic de valori
- Alternativa: generarea aleatoare a cazurilor de test
  - Exemplu: root2/RootAproximationHarness3

OOP12 - M. Joldoș - T.U. Cluj

8



## Furnizarea intrărilor pentru teste

- Alegerea corespunzătoare a cazurilor de test este importantă în depanarea programelor
- Testăm toate caracteristicile metodelor de testat
- Testăm *cazurile tipice*  
100, 1/4, 0.01, 2, 10E12, pentru `SquareRootApproximator`
- Testăm *cazurile limită*: testăm cazurile care sunt la limita intrărilor acceptabile  
0, pentru `SquareRootApproximator`



## Furnizarea intrărilor pentru teste

- Programatorii greșesc adesea la tratarea condițiilor limită
  - Împărțirea cu zero, extragerea de caractere din șiruri vide, accesarea referințelor nule
- Adunăm cazuri de test negative: intrări pe care ne așteptăm ca programul să le respingă
  - Exemplu: radical din -2. Testul trece dacă harness se termină cu eșecul aserțiunii (dacă este activată verificarea aserțiunilor)



## Citirea intrărilor dintr-un fișier

- E mai elegant să punem intrările pentru teste într-un fișier
- Redirectarea intrării: `java Program < data.txt`
- Unele IDE-uri nu suportă redirectarea intrării. În acel caz folosim fereastra de comandă (shell).
- Redirectarea ieșirii: `java Program > output.txt`
- Exemplu: `root2/RootAproximationHarness4`
- Fișierul test.in:
 

```
100
4
2
1
0.25
0.01
```
- Rularea programului:
 

```
java RootApproximatorHarness4 < test.in > test.out
```



## Evaluarea cazurilor de test

- De unde știm dacă ieșirea este corectă?
- Calculăm valorile corecte cu mâna
  - D.e., pentru un program de salarizare, calculăm manual taxele
- Furnizăm intrări de test pentru care știm răspunsurile
  - D.e., rădăcina pătrată a lui 4 este 2, iar pentru 100 este 10
- Verificăm că valorile de ieșire satisfac anumite proprietăți
  - D.e., pătratul rădăcinii pătrate = valoarea inițială
- Folosim un oracol: o metodă lentă, dar sigură pentru a calcula rezultatul în scop de testare
  - D.e., folosim `Math.pow` pentru a calcula mai lent  $x^{1/2}$  (echivalentul rădăcinii pătrate a lui  $x$ )
- Exemplu: `root3/RootAproximationHarness5, 6`



## Testarea regresivă

- Salvăm cazurile de test
- Folosim cazurile de test salvate în versiunile următoare
- *Suită de teste*: un set de teste pentru testarea repetată
- Ciclarea = eroare care a fost reparată, dar reapare în versiuni ulterioare
- **Testarea regresivă**: repetarea testelor anterioare pentru a ne asigura că eșecurile cunoscute ale versiunilor precedente nu apar în versiuni mai noi



## Acoperirea testelor

- **Testarea tip cutie neagră**: testează funcționalitatea fără a ține seama de structura internă a implementării
- **Testarea tip cutie albă**: ia în considerare structura internă la proiectarea testelor
- **Acoperirea testelor**: măsoară câte părți dintr-un program au fost testate
- Trebuie să ne asigurăm că fiecare parte a programului a fost testată măcar o dată de un caz de test
  - D.e., ne asigurăm că am executat fiecare ramură în cel puțin un caz de test



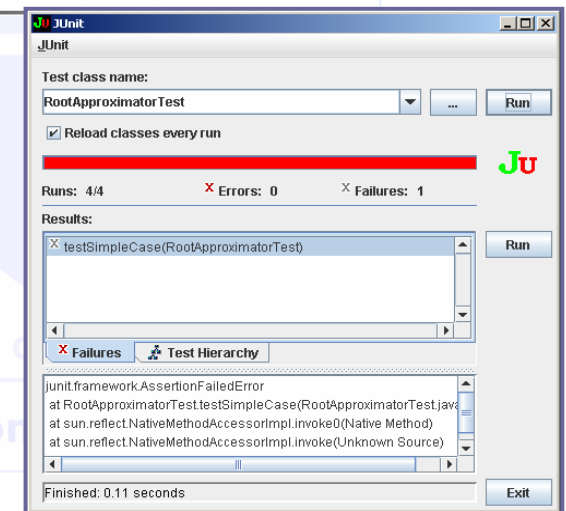
## Acoperirea testelor

- Sugestie: scrieți primele cazuri de test înainte de a termina scrierea completă a programului → vă permite să intuiți mai bine ce ar trebui să facă programul
- Programele de azi pot fi dificil de testat
  - GUI (folosirea mouse)
  - Conexiunile în rețea (întârzierea și căderile)
  - Există unelte pentru a automatiza testarea în aceste scenarii
  - Principiile de bază ale testării regresive și ale acoperirii complete se mențin



## Testarea unităților cu JUnit

- <http://junit.org>
- Preconstruit în unele IDE cum sunt BlueJ și Eclipse
- Filozofia: ori de câte ori implementăm o clasă, implementăm și o clasă însoțitoare, de test
- Demo: proiectul din subdirectorul junit
- În dreapta se află un exemplu cu UI Swing UI de lucru cu junit 3.8.1
  - junit 4.x este diferit, nu are încă UI Swing





## Trasarea execuției programului

- Mesaje care arată calea urmată de execuție

```
if (status == SINGLE)
{
    System.out.println("status is SINGLE");
    . . .
}
```

- Neajuns: trebuie eliminate atunci când s-a terminat testarea și repuse înapoi când apare o altă eroare
- Soluția: folosim clasa **Logger** (pentru jurnalizare) pentru a stopa scrierea mesajelor din trasare fără a le elimina din program (`java.util.logging`)



## Jurnalizarea

- Mesajele de jurnalizare pot fi dezactivate la terminarea testării
- Folosim obiectul global `Logger.global`
- Jurnalizăm un mesaj
- Implicit, mesajele jurnalizate se tipăresc. Le inhibăm cu

```
Logger.global.setLevel(Level.OFF);
```

- Jurnalizarea poate fi o problemă de gândit (nu trebuie să jurnalizăm nici prea multă informație, nici prea puțină)
- Unii programatori preferă depanarea (descrisă în secțiunea următoare) jurnalizării

```
Logger.global.info("status is SINGLE");
```



## Jurnalizarea

- La trasarea cursului execuției, cele mai importante evenimente sunt intrarea în și ieșirea dintr-o metodă
- La începutul metodei, tipărim parametrii:

```
public TaxReturn(double anIncome, int aStatus) {
    Logger.global.info("Parameters: anIncome = " + anIncome
        + " aStatus = " + aStatus);
    . . .
}
```

- La sfârșitul metodei, tipărim valoarea de retur:

```
public double getTax() {
    . . .
    Logger.global.info("Return value = " + tax);
    return tax;
}
```



## Jurnalizarea

- Biblioteca de jurnalizare are un set de nivele predefinite:

SEVERE	<i>Cea mai mare valoare;</i> menită pentru mesaje extrem de importante (d.e. erori de program fatale).
WARNING	Destinată mesajelor de avertizare.
INFO	Pentru mesaje de execuție informative.
CONFIG	Mesaje informative despre setările de configurare/setup.
FINE	Folosit pentru detalii mai fine la depanarea/diagnosticarea problemelor.
FINER	Mai în detaliu.
FINEST	<i>Cea mai mică valoare;</i> cel mai mare grad de detaliu.

- Pe lângă aceste nivele:
  - nivelul ALL, care activează jurnalizarea tuturor înregistrărilor și
  - nivelul OFF care poate fi folosit la dezactivarea jurnalizării.
  - Se pot defini nivele individualizate (vezi documentația Java)
- Demo: loggingEx



## Avantajele jurnalizării

- Jurnalizarea poate genera informații detaliate despre funcționarea unei aplicații.
- O dată adăugată la aplicație, nu mai are nevoie de intervenția umană.
- Jurnalele de aplicație pot fi salvate și studiate ulterior.
- Dacă sunt suficient de detaliate și formate corespunzător, jurnalele de aplicație pot furniza o sursă pentru audit.
- Prin surprinderea erorilor care nu pot fi raportate utilizatorilor, jurnalizarea poate ajuta personalul de sprijin în determinarea cauzelor problemelor apărute.
- Prin surprinderea mesajelor foarte detaliate și a celor specificate de programatori, jurnalizarea poate ajuta la depanare.
- Poate fi o unealtă de depanare acolo unde nu sunt disponibile depanatoarele – adesea aceasta este situația la aplicații distribuite sau multi-fir.
- Jurnalizarea rămâne împreună cu aplicația și poate fi folosită oricând se rulează aplicația.



## Costurile jurnalizării

- Jurnalizarea adaugă o încărcare suplimentară la execuție datorată generării mesajelor și I/E pe dispozitivele de jurnalizare.
- Jurnalizarea adaugă o încărcare suplimentară la programare, pentru că trebuie scris cod suplimentar pentru a genera mesajele.
- Jurnalizarea crește dimensiunea codului.
- Dacă jurnalele sunt prea "vorbărețe" sau prost formate, extragerea informației din acestea poate fi dificilă.
- Instrucțiunile de jurnalizare pot scădea lizibilitatea codului.
- Dacă mesajele de jurnalizare nu sunt întreținute odată cu codul din jur, atunci pot cauza confuzii și deveni o problemă de întreținere.
- Dacă nu sunt adăugate în timpul dezvoltării inițiale, adăugarea ulterioară poate necesita un volum mare de muncă pentru modificarea codului.



## Depanarea



## Folosirea unui depanator

- Depanator = program folosit la rularea altui program care permite analizarea comportamentului la execuție al programului rulat
- Depanatorul permite oprirea și repornirea programului, precum și execuția sa pas-cu-pas.
- Cu cât sunt mai mari programele, cu atât sunt mai greu de depanat prin simpla jurnalizare
- Depanatoarele pot fi parte a IDE (Eclipse, BlueJ, Netbeans) sau programe separate (JSwat)
- Trei concepte cheie:
  - Puncte de întrerupere (breakpoints)
  - Execuție pas-cu-pas (Single-stepping)
  - Inspectarea variabilelor



## Despre depanatoare

- Programele se întâmplă să aibă erori de logică
- Uneori problema poate fi descoperită imediat
- Alteori trebuie determinată
- Un depanator poate fi de mare ajutor
  - Câteodată este exact unealta necesară
  - Alteori, nu
- Depanatoarele sunt în esență asemănătoare
  - "Dacă știi unul, le știi pe toate"
- BlueJ vine cu un depanator simplu care oferă funcționalitatea cea mai importantă
  - Din nefericire, depanatorul BlueJ se mai blochează...



## Ce face un depanator

- Depanatorul permite execuția linie cu linie, instrucțiune cu instrucțiune
- La fiecare pas se pot examina valorile variabilelor
- Se pot seta **puncte de întrerupere (breakpoints)** și se poate spune depanatorului să "continue" (să ruleze mai departe la viteza maximă) până când întâlnește următorul punct de întrerupere
  - La următorul punct de întrerupere se poate relua execuția pas cu pas
- Punctele de întrerupere rămân active până când sunt înlăturate
- Execuția este suspendată ori de câte ori se întâlnește un punct de întrerupere
- În depanator, programul rulează la viteza maximă până ajunge la un punct de întrerupere
- La oprirea execuției putem:
  - Inspecta variabile
  - Executa programul linie cu linie, sau
  - Continua rularea la viteză maximă până la următorul punct de întrerupere



## Setarea unui punct de întrerupere

Clic pe marginea din stânga pentru a seta/elimina puncte de întrerupere

 A red square icon on the left margin of line 19 indicates a breakpoint. A 'saved' button is visible at the bottom right of the editor window."/>


## Atingerea unui punct de întrerupere

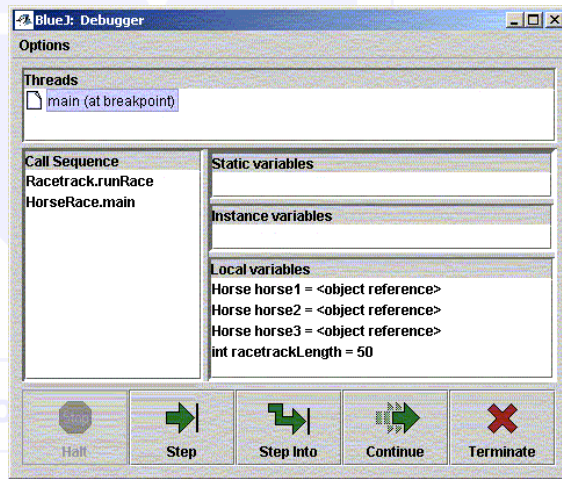
Rulăm programul normal; el se va opri automat la fiecare punct de întrerupere

 A red square icon on the left margin of line 19 indicates a breakpoint. A 'saved' button is visible at the bottom right of the editor window."/>



## Fereastra depanatorului

- Când se ajunge la un punct de întrerupere, se va deschide fereastra depanatorului
- Sau, o putem deschide din meniu

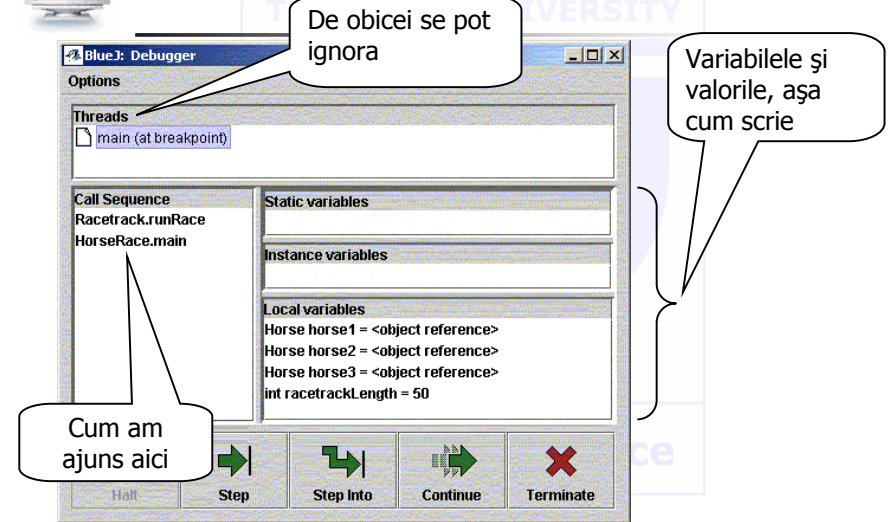


OOP12 - M. Joldoș - T.U. Cluj

29



## Părțile ferestrei depanatorului

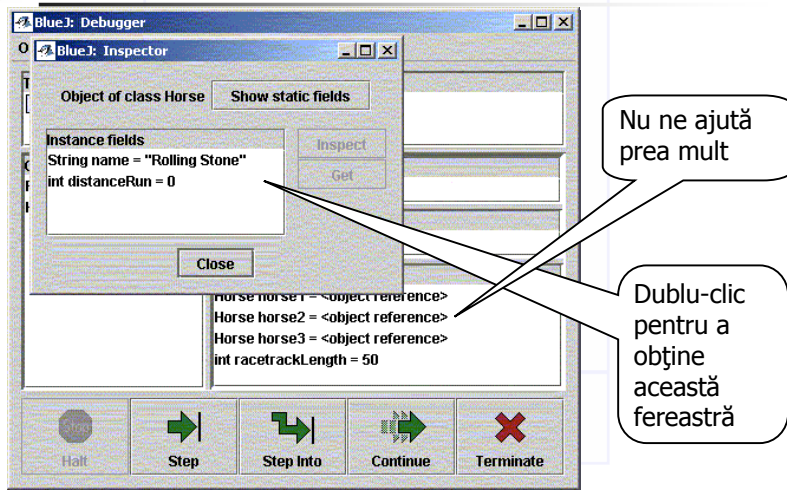


OOP12 - M. Joldoș - T.U. Cluj

30



## Vizualizarea obiectelor

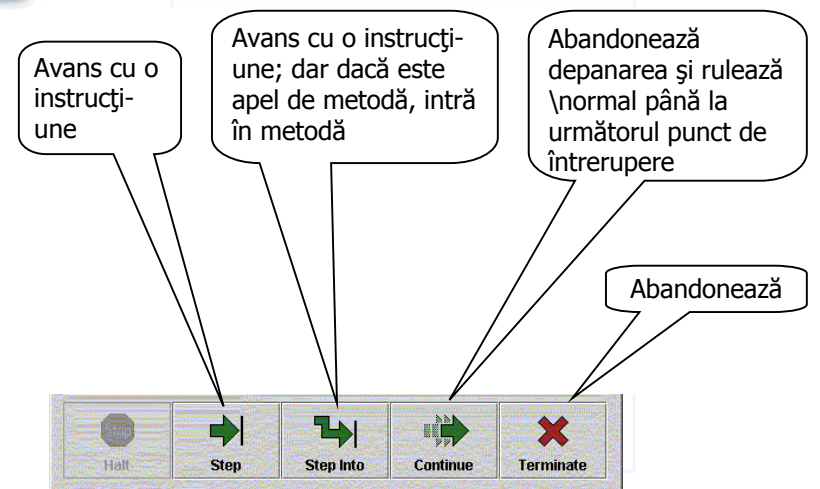


OOP12 - M. Joldoș - T.U. Cluj

31



## Comenzile depanatorului



OOP12 - M. Joldoș - T.U. Cluj

32

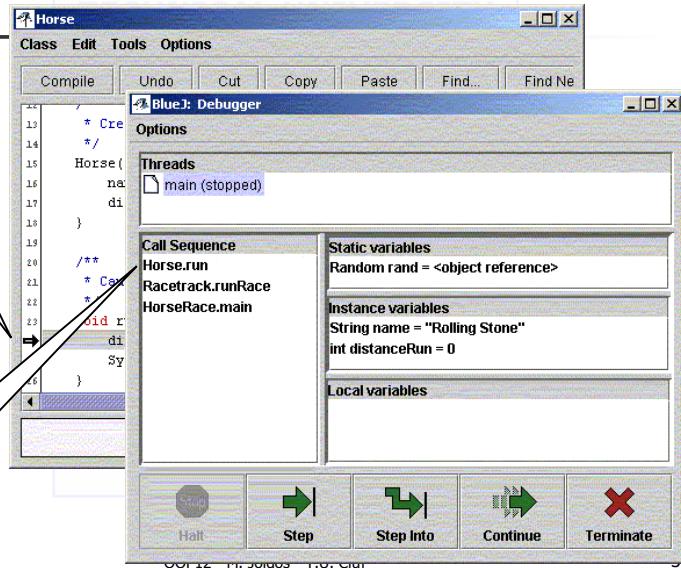




## Execuția unei metode pas cu pas

Aici suntem în noua metodă

Ne arată cum am ajuns în metodă



## Atenționări

- Depanatorul din BlueJ este *foarte* subțire—folosiți-l cu grijă
- Depanatoarele pot fi foarte utile, dar nu înlocuiesc reflectarea asupra problemei
- Adesea, *instrucțiunile print* sunt încă o soluție mai bună



## Introducere în I/E Java



## Introducere în I/E Java

- Sistemul de I/E este foarte complex
  - Încearcă să facă multe lucruri folosind componente reutilizabile
  - Există de fapt trei sisteme de I/E: cel original din JDK 1.0 și unul mai nou începând cu JDK 1.2 care se suprapune și îl înlocuiește parțial
  - Pachetul `java.nio` din JDK 1.4 este și mai nou, dar nu-l vom trata aici
- Efectuarea de operații de I/E cere programatorului să folosească o serie de clase complexe
  - De obicei se creează clase auxiliare cum sunt `StdIn`, `FileIn` și `FileOut` pentru a ascunde această complexitate



## Introducere în I/E Java

- Motivele complexității Java I/E:
  - Sunt multe tipuri diferite de surse și absorbante (sinks)
  - Două tipuri diferite de acces la fișiere
    - Acces secvențial
    - Acces aleator
  - Două tipuri diferite de formate de stocare
    - Formatat
    - Neformatat
  - Trei sisteme de I/E diferite (vechi și noi)
  - O mulțime de clase "filtru" sau "modificator"



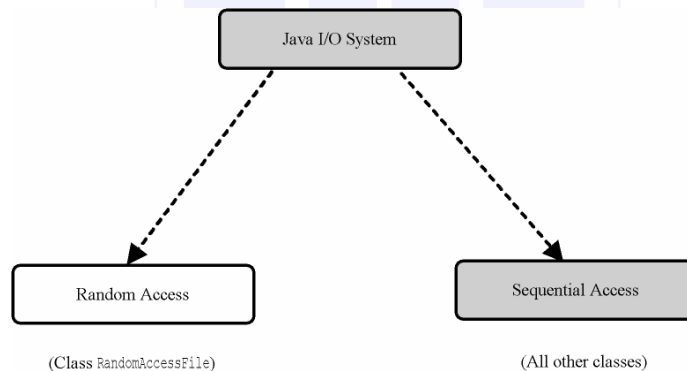
## Accesul aleatoriu vs Secvențial

- Accesul secvențial
  - Fișierul este prelucrat octet după octet
  - Poate fi ineficient
- Accesul aleator
  - Permite accesul la locații arbitrare în fișier
  - Doar fișierele disc suporta accesul aleator
    - `System.in` și `System.out` nu-l suportă
  - Fiecare fișier disc are o poziție specială pentru indicatorul de de fișier
    - Se poate citi sau scrie la poziția curentă a indicatorului



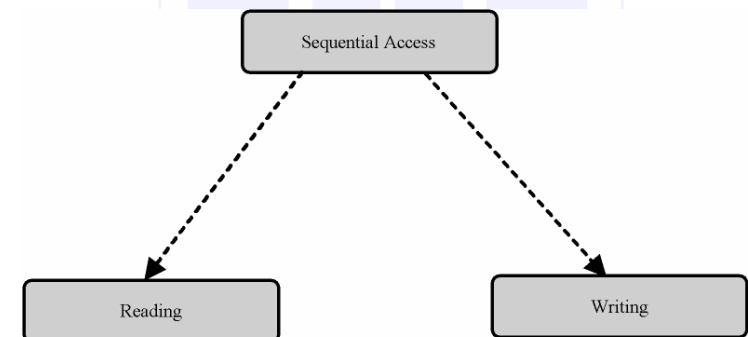
## Structura sistemului de I/E Java (`java.io`)

- Sistemul de I/E Java este divizat în clase pentru accesul secvențial și clase pentru accesul aleatoriu (engl. random, numit și acces direct):



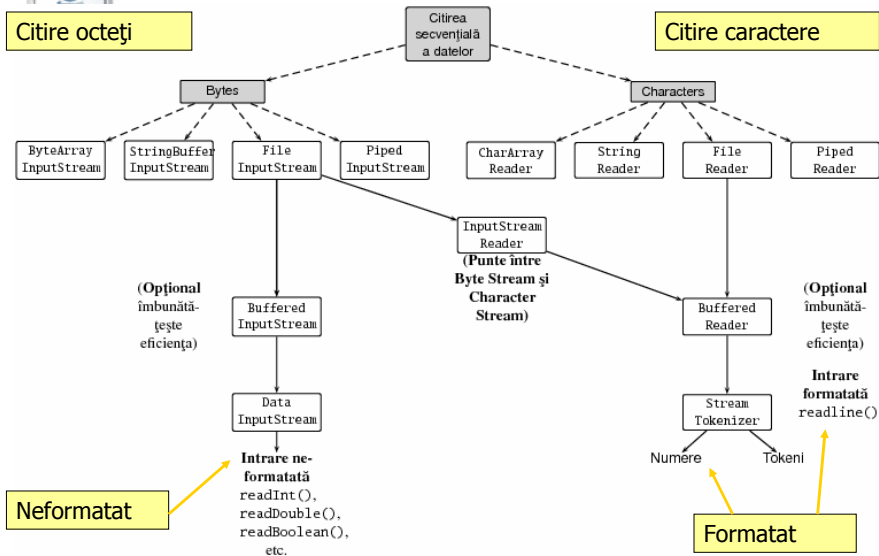
## Structura sistemului de I/E Java (`java.io`)

- Accesul secvențial este subîmpărțit în clase pentru citire și clase pentru scriere:





## Clase pentru citirea secvențială a datelor (din java.io)

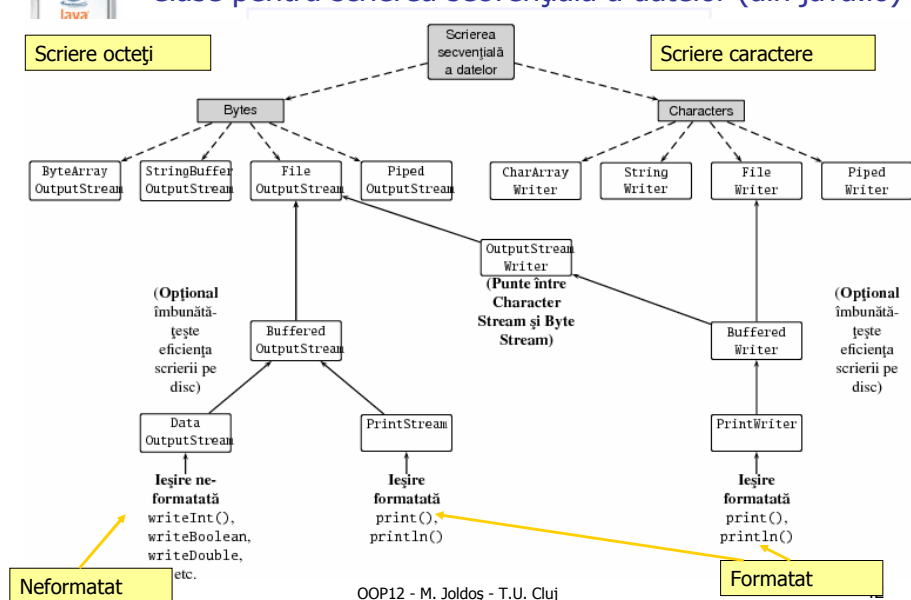


OOP12 - M. Joldoș - T.U. Cluj

41



## Clase pentru scrierea secvențială a datelor (din java.io)



OOP12 - M. Joldoș - T.U. Cluj



## Excepții

- Toate clasele de I/E Java aruncă excepții, cum este **FileNotFoundException** și excepția mai generală **IOException**
- Programele Java trebuie să intercepteze explicit excepțiile de I/E în structuri **try / catch** pentru a gestiona problemele de I/E.
  - Această structură *trebuie* să trateze **IOException**, care este clasa generală de excepții de I/E
  - Poate trata excepțiile de nivel mai jos separat – cum este cazul cu **FileNotFoundException**. Aceasta permite programului să ofere utilizatorului informații inteligente și opțiuni în cazul în care nu se găsește un fișier.

OOP12 - M. Joldoș - T.U. Cluj

43



## Folosirea I/E Java

- Procedura generală pentru folosirea I/E Java este:
  - Creăm o structură **try/catch** pentru excepțiile de I/E
  - Alegem o clasă de intrare sau ieșire pe baza tipului de I/E (formatat sau neformat, secvențial sau direct) și tipul de flux (stream) de intrare sau ieșire (fișier, conductă [pipe], etc.)
  - Împachetăm clasa de intrare sau ieșire într-o clasă tampon (buffer) pentru creșterea eficienței
  - Folosim clase filtru sau modificatoare pentru a traduce datele în forma corespunzătoare pentru intrare sau ieșire (d.e., **DataInputStream** sau **DataOutputStream**)

OOP12 - M. Joldoș - T.U. Cluj

44



## Exemplu: Citirea de `string`-uri dintr-un fișier secvențial formatat

- Alegem clasa `FileReader` pentru a citi date secvențiale formate.
  - Deschidem fișierul prin crearea unui obiect `FileReader`
  - Împachetăm `FileReader` într-un `BufferedReader` pentru eficiență
  - Citim fișierul cu metoda `BufferedReader` numită `readLine()`.
  - Închidem fișierul folosind metoda `close()` a lui `FileReader`.
  - Tratăm excepțiile de I/E folosind o structură `try/catch`.



## Exemplu

```

// Interceptam exceptiile daca apar
try
{
// Creeaza BufferedReader
BufferedReader in =
new BufferedReader( new FileReader(args[0]) );
// Read file and display data
while( (s = in.readLine()) != null)
{
System.out.println(s);
}
// Incheie fisierul file
in.close();
}
// Intercepteaza FileNotFoundException
catch (FileNotFoundException e)
{
System.out.println("File not found: " + args[0]);
}
// Intercepteaza alte IOExceptions

```

Includem I/E într-o structură `try/catch`  
 Deschidem fișierul prin crearea unui `FileReader` împachetat într-un `BufferedReader`  
 Citim linii cu `readLine()`  
 Închidem fișierul cu `close()`  
 Tratăm excepțiile



## ScannerS

- În loc să citim direct din `System.in` sau dintr-un fișier text folosim un `Scanner`
  - Întotdeauna trebuie să spunem lui `Scanner` ce să citească
  - D.e. îl instanțiem cu o referință pentru a citi din `System.in`

```
java.util.Scanner scanner =
new java.util.Scanner(System.in);
```
- Ce anume face `Scanner`?
  - Divizează intrarea în unități gestionabile numite *token-i*

```
Scanner scanner = new Scanner(System.in);
String userInput = scanner.nextLine();
```

`nextLine()` ia tot ce s-a tastat la consolă până când utilizatorul introduce un retur de car (apasă tasta "Enter")
  - Token-ii au mărimea linii de intrare și sunt de tipul `String`



## Alte metode din clasa `Scanner`

Pentru a citi un:	Folosim metoda <code>scanner</code>
boolean	<code>boolean nextBoolean()</code>
double	<code>double nextDouble()</code>
float	<code>float nextFloat()</code>
int	<code>int nextInt()</code>
long	<code>long nextLong()</code>
short	<code>short nextShort()</code>
String (care apare pe linia următoare, până la '\n')	<code>String nextLine()</code>
String (care apare pe linia următoare, până la următorul ' ', '\t', '\n')	<code>String next()</code>



## Excepții pentru Scanner

- **InputMismatchException**
  - Aruncată de toate metodele `nextType()`
  - Semnificație: token-ul nu poate fi convertit într-o valoare de tipul specificat
  - **Scanner** nu avansează la token-ul următor, astfel că acest token poate fi încă regăsit
- Tratarea acestei excepții
  - Preveniți-o
    - Testați token-ul următor folosind o metodă `hasNextType()`
    - Metoda nu avansează, doar verifică tipul token-ului următor
  - `boolean hasNextBoolean()`    `boolean hasNextLong()`
  - `boolean hasNextDouble()`    `boolean hasNextShort()`
  - `boolean hasNextFloat()`    `boolean hasNextLine()`
  - `boolean hasNextInt()`    `Vezi documentația pentru detalii despre metodele clasei Scanner!`
  - Interceptați-o
    - Tratați excepția o dată interceptată



## Fluxuri (streams) de obiecte

- Clasa **ObjectOutputStream** poate salva obiecte întregi pe disc
- Clasa **ObjectInputStream** poate citi obiectele de pe disc înapoi în memorie
- Obiectele sunt salvate în format binar; de aceea folosim fluxuri (streams)
- Fluxul pentru ieșire de obiecte salvează toate variabilele instanță
  - Exemplu: Scrierea unui obiect **BankAccount** într-un fișier
 

```
BankAccount b = . . . ;
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("bank.dat"));
out.writeObject(b);
```



## Exemplu: citirea unui obiect BankAccount dintr-un fișier

- **readObject** returnează o referință la un **Object**
  - Este nevoie să ne reamintim tipurile obiectelor care au fost salvate și să folosim o forțare (cast) de tip
 

```
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```
- Metoda **readObject** poate arunca o excepție de tipul **ClassNotFoundException**
  - este o excepție verificată
  - trebuie fie interceptată, fie declarată



## Scrierea și citirea unui ArrayList într-un/dintr-un fișier

### ■ Scrierea

```
ArrayList<BankAccount> a = new ArrayList<BankAccount>();
// Now add many BankAccount objects into a
out.writeObject(a);
```

### ■ Citirea

```
ArrayList<BankAccount> a = (ArrayList<BankAccount>)
in.readObject();
```



## Serializabil

- Obiectele care sunt scrise într-un flux de obiecte trebuie să aparțină unei clase care implementează interfața **Serializable**. D.e.
 

```
class BankAccount implements Serializable {...}
```
- Interfața **Serializable** nu are metode.
- Obiectele care se salvează trebuie să marcheze toate câmpurile neserializabile (d.e. referințe la obiecte **Thread**, **OutputStream** și subclasele sale și **Socket**) ca **transient**
- **Serializare**: procesul de salvare a obiectelor într-un flux
  - Fiecărui obiect îi este atribuit un număr de serie pe flux
  - Dacă același obiect este salvat de două ori, a doua oară se salvează numai numărul de serie
  - La citire, numerele de serie duplicate sunt restaurate ca referințe la același obiect
- **Demo**: serial



## API de I/E noi pentru Java

- **Buffer (zonă tampon)**: o secvență liniară de elemente de un tip *primitiv* precizat
  - Consolidează operațiile de I/E
  - Patru proprietăți (toate cu valori întotdeauna pozitive)
    - Capacitate: numărul de elemente pe care le conține (nu se schimbă)
    - Limita: indexul primului element care nu trebuie scris sau citit
    - Poziție: indexul următorului element de citit sau scris
    - Marcaj: index la care se va reseta poziția la invocarea metodei `reset()`
  - Invariant:  $0 \leq \text{marcaj} \leq \text{poziție} \leq \text{limita} \leq \text{capacitate}$



## API de I/E noi pentru Java

- **Buffer (continuare)**
  - Operații **put** și **get**
    - Relative (la poziția curentă) sau absolute
  - **clear**
    - pregătește tamponul pentru o secvență de operații *channel-read* sau *put* relative: pune limita = capacitate, poziție = 0.
  - **flip**
    - pregătește tamponul pentru o secvență de operații *channel-write* sau *get* relative: pune limita = poziție, apoi poziție = 0.
  - **rewind**
    - pregătește tamponul pentru re-citirea datelor pe care le conține deja: lasă limita neschimbată, pune poziție = 0.
  - **reset**
    - Resetează poziția tamponului la cea marcată anterior



## API de I/E noi pentru Java

- **Channels (canale)**
  - Conexiunea cu un dispozitiv de I/E
    - Au obiecte pereche în **java.io**, unul dintre: **FileInputStream**, **FileOutputStream**, **RandomAccessFile**, **Socket**, **ServerSocket** or **DatagramSocket**
    - Interacționează eficient cu tamponurile
  - Interfața **ReadableByteChannel**
    - Metoda **read** citește o secvență de octeți din canal în tamponul specificat
  - Interfața **WritableByteChannel**
    - Metoda **write** scrie o secvență de octeți în canal din tamponul specificat
  - Citiri distribuitoare (scattering) și scrieri colectoare (gathering)
    - operează cu secvențe de tamponuri
  - Clasa **FileChannel**



## API de I/E noi pentru Java

- Zăvoare pentru fișiere (file locks)
  - Restricționează accesul la o porțiune de fișier
    - `FileChannel`, poziție, mărime
    - Exclusive sau partajate
- Seturi de caractere (charsets)
  - Pachetul `java.nio.charset`
    - Clasa `Charset`
      - Metode `decode`, `encode`
    - Clasa `CharsetDecoder`, `CharsetEncoder`
- Demo: `fileChannel`



## Rezumat

- Testarea software
  - testarea funcțională; proiectarea, planificarea și testarea cazurilor
  - test harnesses
  - furnizarea intrării pentru teste
  - evaluarea rezultatelor testelor
  - acoperirea testelor
  - testarea unităților de program – JUnit
- Depanarea
  - folosirea unui depanator
- Introducere în I/E Java
  - structura I/E Java
  - clase pentru citire/scriere
  - acces secvențial/aleator
  - excepții
  - procedură generală de efectuare a I/E
- Clasa `Scanner`
- Noua I/E Java
  - Zone tampon, canale
- Trasarea execuției unui program
  - Jurnalizarea