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, securităţii şi calităţii software
- *Testarea funcţională* : determină dacă sistemul satisface specificaţiile clientului.
- *Testarea tip cutie neagră*:
  - Proiectantul testelor ignoră structura internă a implementării.
  - Testul este condus de comportamentul extern aşteptat al sistemului
  - Sistemul este tratat ca o "cutie neagră": comportamentul este observabil, dar structura internă nu este cunoscută.

Proiectarea, planificarea şi testarea cazurilor

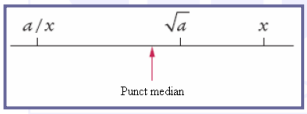
- Proiectarea testelor începe de obicei cu analiza
  - Specificaţiilor funcţionale ale sistemului şi a
  - Cazurilor de utilizare: a modurilor în care va fi folosit sistemul.
- Un caz de testare este definit de
  - Declararea obiectivelor cazului;
  - Setul de date pentru caz;
  - Rezultatele aşteptate.
- 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 unităţilor

- 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, harnaşament)
- 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* şi *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:

Testarea programului

- Clasa `RootApproximator` funcţionează 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 şi e nevoie de re-testare, e nevoie să ne reamintim toate intrările
- Soluţie: scriem test harnesses care să uşureze repetarea testelor de unităţi

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 şi 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 fişier

Furnizarea intrărilor pentru teste

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

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 greşesc adesea la tratarea condiţiilor limită
  - Împărţirea cu zero, extragerea de caractere din şiruri vide, accesarea referinţelor nule
- Adunăm cazuri de test negative: intrări pe care ne aşteptăm ca programul să le respingă
  - Exemplu: radical din -2. Testul trece dacă harness se termină cu eşecul aserţiunii (dacă este activată verificarea aserţiunilor)

Citirea intrărilor dintr-un fişier

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

```
100
4
2
1
0.25
0.01
```

- Rularea programului:

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


Evaluarea cazurilor de test

- De unde ştim dacă ieşirea este corectă?
- Calculam valorile corecte cu mâna
  - D.e., pentru un program de salarizare, calculăm manual taxele
- Furnizăm intrări de test pentru care ştim răspunsurile
  - D.e., rădăcina pătrată a lui 4 este 2, iar pentru 100 este 10
- Verificăm că valorile de ieşire satisfac anumite proprietăţi
  - D.e., pătratul rădăcinii pătrate = valoarea iniţială
- 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 eşecurile cunoscute ale versiunilor precedente nu apar în versiuni mai noi

Acoperirea testelor

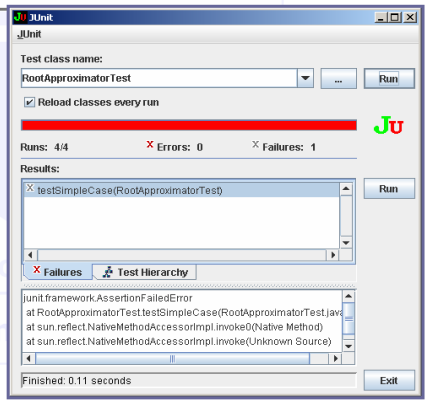
- Testarea tip cutie neagră: testează funcţionalitatea fără a ţine seama de structura internă a implementării
- Testarea tip cutie albă: ia în considerare structura internă la proiectarea testelor
- Acoperirea testelor: măsoară câte părţi dintr-un program au fost testate
- Trebuie să ne asigurăm că fiecare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

Acoperirea testelor

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

Testarea unităţilor cu JUnit

- http://junit.org
- Preconstruit în unele IDE cum sunt BlueJ şi Eclipse
- Filozofia: ori de câte ori implementăm o clasă, implementăm si o clasă însoţitoare, 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 execuţiei programului

- Mesaje care arată calea urmată de execuţie

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

- Neajuns: trebuie eliminate atunci când s-a terminat testarea şi repuse înapoi când apare o altă eroare
- Soluţia: 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 informaţie, nici prea puţină)
- Unii programatori preferă depanarea (descrisă în secţiunea următoare) jurnalizării

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

Jurnalizarea

- La trasarea cursului execuţiei, cele mai importante evenimente sunt intrarea în şi ieşirea dintr-o metodă
- La începutul metodei, tipărim parametrii:

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

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

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

Avantajele jurnalizării

- Jurnalizarea poate genera informaţii detaliate despre funcţionarea unei aplicaţii.
- O dată adăugata la aplicaţie, nu mai are nevoie de intervenţia umană.
- Jurnalele de aplicaţie pot fi salvate şi studiate ulterior.
- Dacă sunt suficient de detaliate şi formatate corespunzător, jurnalele de aplicaţie 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 şi 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 situaţia la aplicaţii distribuite sau multi-fir.
- Jurnalizarea rămâne împreună cu aplicaţia şi poate fi folosită oricând se rulează aplicaţia.

Costurile jurnalizării

- Jurnalizarea adaugă o încărcare suplimentară la execuţie datorata generării mesajelor şi I/E pe dispozitivele de jurnalizare.
- Jurnalizarea adaugă o încărcare suplimentară la programare, pentru că trebuie scris cod suplimentar pentru a genera mesajele.
- Jurnalizarea creşte dimensiunea codului.
- Dacă jurnalele sunt prea "vorbăreţe" sau prost formatate, extragerea informaţiei din acestea poate fi dificilă.
- Instrucţiunile de jurnalizare pot scădea lizibilitatea codului.
- Dacă mesajele de jurnalizare nu sunt întreţinute odată cu codul din jur, atunci pot cauza confuzii şi deveni o problemă de întreţinere.
- Dacă nu sunt adăugate în timpul dezvoltării iniţiale, 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 execuţie al programului rulat
- Depanatorul permite oprirea şi repornirea programului, precum şi execuţia 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)
  - Execuţie pas-cu-pas (Single-stepping)
  - Inspectarea variabilelor

Despre depanatoare

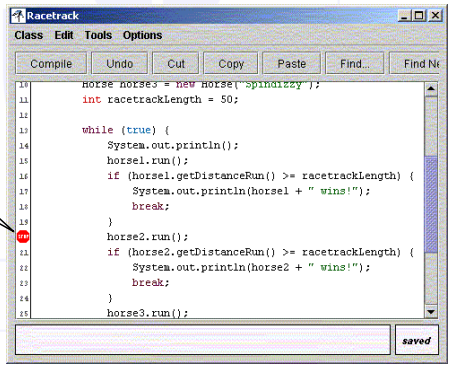
- Programele se întâmplă să aibă erori de logică
- Uneori problema poate fi descoperită imediat
- Alteori trebuie determinată
- Un depanator poate fi de mare ajutor
  - Câteodată este exact unealta necesară
  - Alteori, nu
- Depanatoarele sunt în esenţă asemănătoare
  - "Dacă ştii unul, le ştii pe toate"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

Ce face un depanator

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

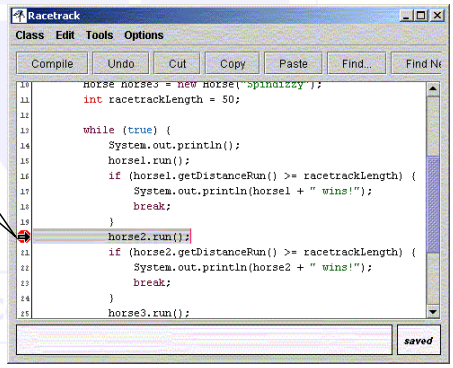
Setarea unui punct de întrerupere

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



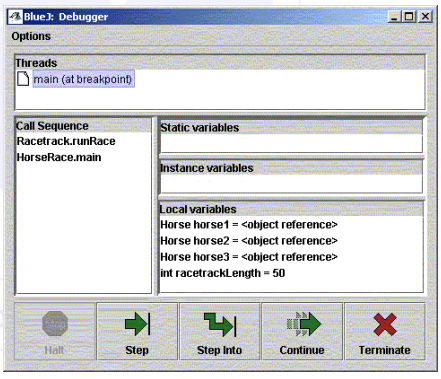
Atingerea unui punct de întrerupere

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



Fereastra depanatorului

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

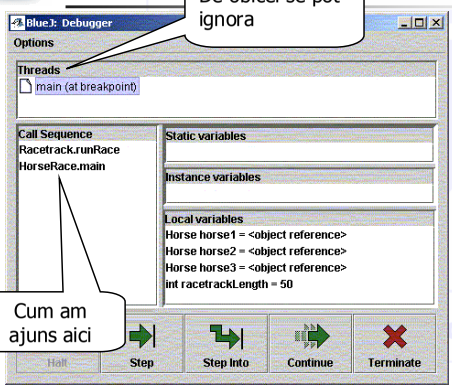


Părţile ferestrei depanatorului

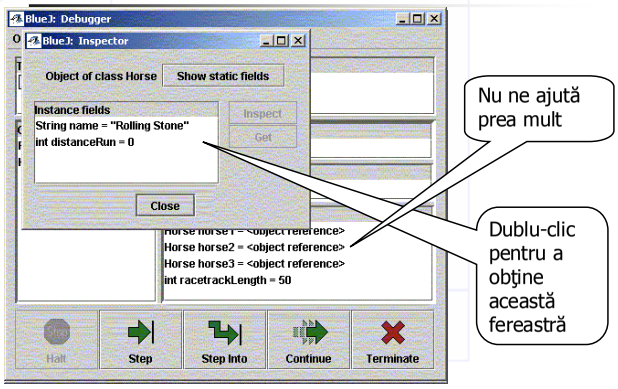
De obicei se pot ignora

Variabilele şi valorile, aşa cum scrie

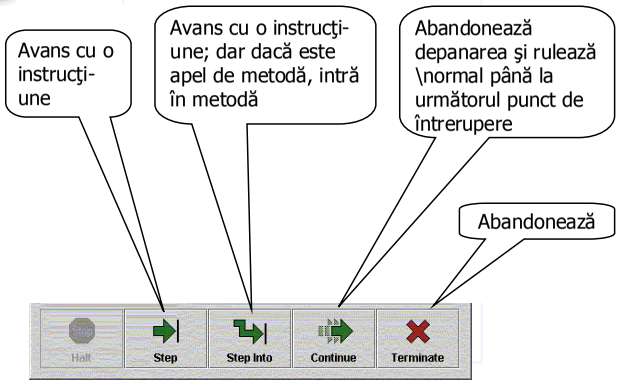
Cum am ajuns aici



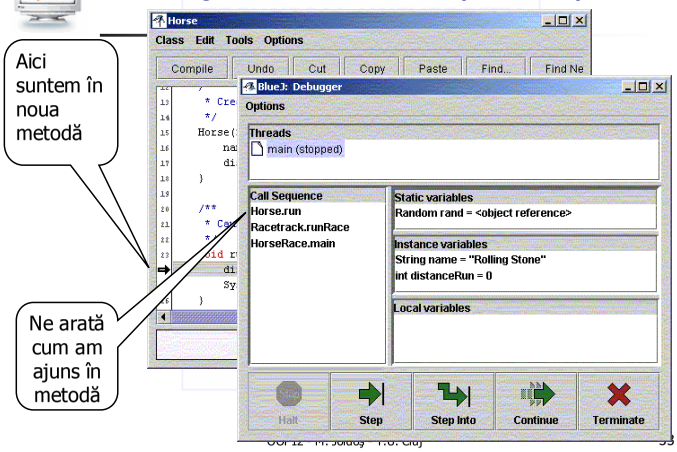
Vizualizarea obiectelor



Comenzile depanatorului



Execuţia unei metode pas cu pas



Atenţionări

- Depanatorul din BlueJ este *foarte* subţire—folosiţi-l cu grijă
- Depanatoarele pot fi foarte utile, dar nu înlocuiesc reflectarea asupra problemei
- Adesea, *instrucţiunile print* sunt încă o soluţie 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 şi unul mai nou începând cu JDK 1.2 care se suprapune şi îl înlocuieşte parţial
  - Pachetul `java.nio` din JDK 1.4 este şi mai nou, dar nu-l vom trata aici
- Efectuarea de operaţii de I/E cere programatorului să folosească o serie de clase complexe
  - De obicei se creează clase auxiliare cum sunt `StdIn`, `FileIn` şi `FileOut` pentru a ascunde această complexitate

Introducere în I/E Java

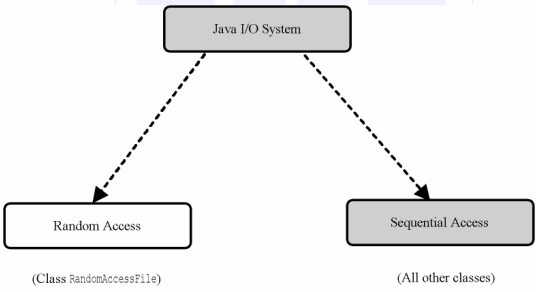
- Motivele complexităţii Java I/E:
  - Sunt multe tipuri diferite de surse şi absorbante (sinks)
  - Două tipuri diferite de acces la fişiere
    - Acces secvenţial
    - Acces aleator
  - Două tipuri diferite de formate de stocare
    - Formatat
    - Neformatat
  - Trei sisteme de I/E diferite (vechi şi noi)
  - O mulţime de clase "filtru" sau "modificator"

Accesul aleatoriu vs Secvenţial

- Accesul secvenţial
  - Fişierul este prelucrat octet după octet
  - Poate fi ineficient
- Accesul aleator
  - Permite accesul la locaţii arbitrare în fişier
  - Doar fişierele disc suporta accesul aleator
    - `System.in` şi `System.out` nu-l suportă
  - Fiecare fişier disc are o poziţie specială pentru indicatorul de de fişier
    - Se poate citi sau scrie la poziţia curenta a indicatorului

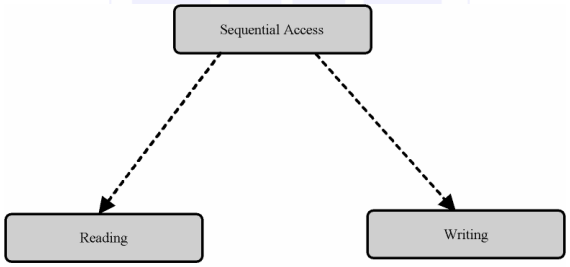
Structura sistemului de I/E Java (`java.io`)

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

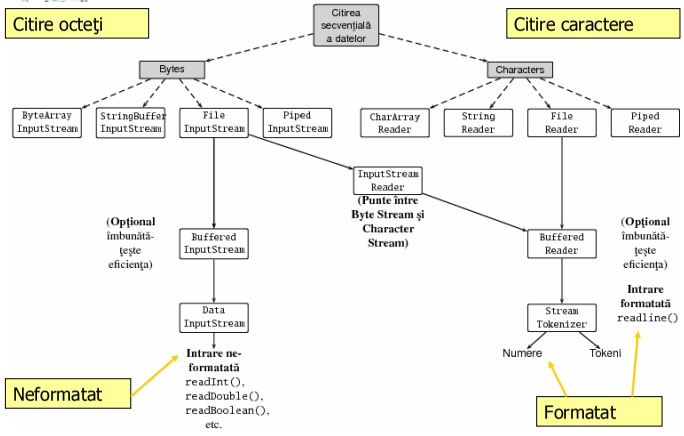


Structura sistemului de I/E Java (`java.io`)

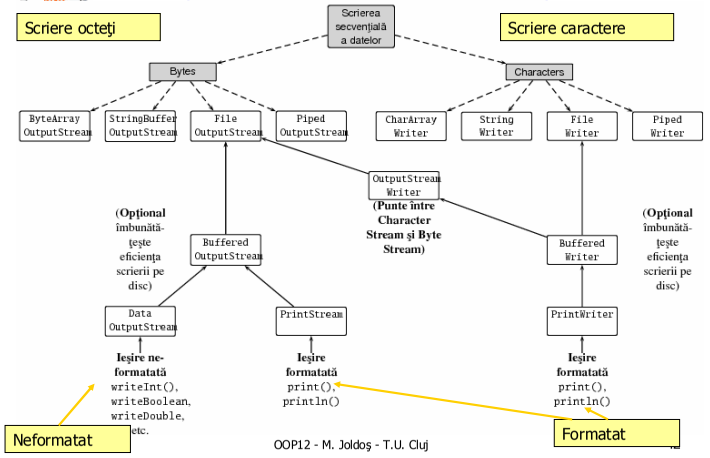
- Accesul secvenţial este subîmpărţit în clase pentru citire şi clase pentru scriere:



Clase pentru citirea secvenţială a datelor (din java.io)



Clase pentru scrierea secvenţială a datelor (din java.io)



Excepţii

- Toate clasele de I/E Java aruncă excepţii, cum este **`FileNotFoundException`** şi excepţia mai generală **`IOException`**
- Programele Java trebuie să intercepteze explicit excepţiile 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 excepţii de I/E
  - Poate trata excepţiile de nivel mai jos separat – cum este cazul cu **`FileNotFoundException`**. Aceasta permite programului să ofere utilizatorului informaţii inteligente şi opţiuni în cazul în care nu se găseşte un fişier.

Folosirea I/E Java

- Procedura generală pentru folosirea I/E Java este:
  - Creăm o structură **`try/catch`** pentru excepţiile de I/E
  - Alegem o clasă de intrare sau ieşire pe baza tipului de I/E (formatat sau neformatat, secvenţial sau direct) şi tipul de flux (stream) de intrare sau ieşire (fişier, conductă [pipe], etc.)
  - Împachetăm clasa de intrare sau ieşire într-o clasă tampon (buffer) pentru creşterea eficienţei
  - Folosim clase filtru sau modificatoare pentru a translata datele în forma corespunzătoare pentru intrare sau ieşire (d.e., **`DataInputStream`** sau **`DataOutputStream`**)

Exemplu: Citirea de **`String`**-uri dintr-un fişier secvenţial formatat

- Alegem clasa **`FileReader`** pentru a citi date secvenţiale formatate.
  - Deschidem fişierul prin crearea unui obiect **`FileReader`**
  - Împachetăm **`FileReader`** într-un **`BufferedReader`** pentru eficienţă
  - Citim fişierul cu metoda **`BufferedReader`** numită **`readLine()`.**
  - Închidem fişierul folosind metoda **`close()`** a lui **`FileReader`.**
  - Tratăm excepţiile de I/E folosind o structură **`try/catch`**.

Exemplu

Includem I/E într-o structură **`try/catch`**

Deschidem fişierul prin crearea unui **`FileReader`** împachetat într-un **`BufferedReader`**

Citim linii cu **`readLine()`**

Închidem fişierul cu **`close()`**

Tratăm excepţiile

```
// 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);
  }
  // Inchide fisierul file
  in.close();
}
// Intercepteaza FileNotFoundException
catch (FileNotFoundException e)
{
  System.out.println("File not found: " + args[0]);
}
// Intercepteaza alte IOExceptions
```


`Scanner`s

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

```
java.util.Scanner scanner =
    new java.util.Scanner(System.in);
```

- Ce anume face **`Scanner`**?
  - Divizează intrarea în unităţi 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 şi sunt de tipul **`String`**

Alte metode din clasa **`Scanner`**

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

Excepţii pentru `Scanner`

- `InputMismatchException`
  - Aruncată de toate metodele `next`*`Type`*`()`
  - Semnificaţie: 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 excepţii
  - Preveniţi-o
    - Testaţi token-ul următor folosind o metodă `hasNext`*`Type`*`()`
    - Metoda nu avansează, doar verifică tipul token-ului următor
  - `boolean hasNextBoolean()`
  - `boolean hasNextByte()`
  - `boolean hasNextDouble()`
  - `boolean hasNextFloat()`
  - `boolean hasNextInt()`
  - `boolean hasNextLong()`
  - `boolean hasNextShort()`
  - `boolean hasNextLine()`
  - Vezi documentaţia pentru detalii despre metodele clasei `Scanner`!
  - Interceptaţi-o
    - Trataţi excepţia 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 ieşire de obiecte salvează toate variabilele instanţă
  - Exemplu: Scrierea unui obiect `BankAccount` într-un fişier

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


Exemplu: citirea unui obiect `BankAccount` dintr-un fişier

- `readObject` returnează o referinţă la un `Object`
  - Este nevoie să ne reamintim tipurile obiectelor care au fost salvate şi să folosim o forţare (cast) de tip

```
ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

- Metoda `readObject` poate arunca o excepţie de tipul `ClassNotFoundException`
  - este o excepţie verificată
  - trebuie fie interceptată, fie declarată

Scrierea şi citirea unui `ArrayList` într-un/dintr-un fişier

- 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 aparţină unei clase care implementează interfaţa `Serializable`. D.e.

```
class BankAccount implements Serializable {...}
```

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

API de I/E noi pentru Java

- Buffer (zonă tampon): o secvenţă liniară de elemente de un tip *primitiv* precizat
  - Consolidează operaţiile de I/E
  - Patru proprietăţi (toate cu valori întotdeauna pozitive)
    - Capacitate: numărul de elemente pe care le conţine (nu se schimbă)
    - Limita: indexul primului element care nu trebuie scris sau citit
    - Poziţie: indexul următorului element de citit sau scris
    - Marcaj: index la care se va reseta poziţia la invocarea metodei `reset()`
  - Invariant: 0 <= *marcaj* <= *pozitie* <= *limita* <= *capacitate*

API de I/E noi pentru Java

- Buffer (continuare)
  - Operaţii **`put`** şi **`get`**
    - Relative (la poziţia curentă) sau absolute
  - **`clear`**
    - pregăteşte tamponul pentru o secvenţă de operaţii *channel-read* sau *put* relative: pune limita = capacitate, pozitie = 0.
  - **`flip`**
    - pregăteşte tamponul pentru o secvenţă de operaţii *channel-write* sau *get* relative: pune limita = pozitie, apoi pozitie = 0.
  - **`rewind`**
    - pregăteşte tamponul pentru re-citirea datelor pe care le conţine deja: lasă limita neschimbată, pune poziţie = 0.
  - **`reset`**
    - Resetează poziţia 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`**
    - Interacţionează eficient cu tampoanele
  - Interfaţa **`ReadableByteChannel`**
    - Metoda **`read`** citeşte o secvenţă de octeţi din canal în tamponul specificat
  - Interfaţa **`WriteableByteChannel`**
    - Metoda **`write`** scrie o secvenţă de octeţi în canal din tamponul specificat
  - Citiri distribuitoare (scattering) şi scrieri colectoare (gathering)
    - operează cu secvenţe de tampoane
  - Clasa **`FileChannel`**

API de I/E noi pentru Java

- Zăvoare pentru fişiere (file locks)
  - Restricţionează accesul la o porţiune de fişier
    - **`FileChannel`**, poziţie, 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 funcţională; proiectarea, planificarea şi testarea cazurilor
  - test harnesses
  - furnizarea intrării pentru teste
  - evaluarea rezultatelor testelor
  - acoperirea testelor
  - testarea unităţilor de program – JUnit
- Trasarea execuţiei unui program
  - Jurnalizarea
- Depanarea
  - folosirea unui depanator
- Introducere în I/E Java
  - structura I/E Java
  - clase pentru citire/scriere
  - acces secvenţial/aleator
  - excepţii
  - procedură generală de efectuare a I/E
- Clasa Scanner
- Noua I/E Java
  - Zone tampon, canale