Programare orientată pe obiecte

1. Testarea și depanarea programelor
2. Introducere în I/E Java

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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
  - 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 dezvolta un plan de teste
  - 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

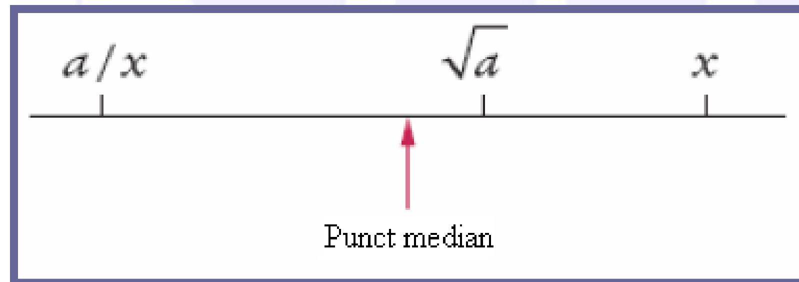
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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

- 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 pentru *x*



  4. Repetăm procedura (pasul 3) și ne oprim atunci când două valori succesive sunt foarte apropiate una de alta
- Metoda converge repede

Testarea programului

- Clasa scrisă pentru rezolvarea problemei 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 *teste harness* 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")
  - 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
- Putem genera automat cazurile de testat
  - Pentru puţine intrări posibile este fezabil să rulăm un număr (reprezentativ) de teste într-un ciclu
  - Testul anterior este restricţionat la un subset mic de valori
  - Alternativa: generarea aleatoare a cazurilor de test

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Furnizarea intrărilor pentru teste

- Alegerea corespunzătoare a cazurilor de test este importantă în depanarea programelor
  - Testăm toate caracteristicile metodelor de testat
  - Testăm *cazurile tipice* – exemplu: 100, 1/4, 0.01, 2, 10E12, pentru problema descrisă anterior
  - Testăm *cazurile limită*: testăm cazurile care sunt la limita intrărilor acceptabile – exemplu: 0, pentru problema descrisă anterior
- 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, când 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 acest caz folosim fereastra de comandă (shell)
- Redirectarea ieşirii: `java Program > output.txt`
- Exemplu:
  - Fişierul test.in:

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

  - Rularea programului:

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

Evaluarea cazurilor de test

- De unde ştim dacă ieşirea este corectă?
- Calculăm 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 o altă metodă 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)

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 testată 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
- În dreapta se află un exemplu cu UI Swing UI de lucru cu junit 3.8.1

JUnit

JUnit

Test class name:

RootApproximatorTest ... Run

Reload classes every run

Runs: 4/4 Errors: 0 Failures: 1

Results:

testSimpleCase(RootApproximatorTest)

Run

Failures Test Hierarchy

junit.framework.AssertionFailedError
at RootApproximatorTest.testSimpleCase(RootApproximatorTest.java
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)

Finished: 0.11 seconds

Exit

Exemplu simplu cu JUnit

```
public class Calculate {
  public int sum(int var1, int var2) {
   System.out.println("Adding values: " + var1 + " + " + var2);
   return var1 + var2;
  }
}
```

```
import static org.junit.Assert.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculateTest {
  Calculate calculation = new Calculate();
  int sum = calculation.sum(2, 5);
  int testSum = 7;
  @Test
  public void testSum() {
  System.out.println("@Test sum(): " + sum + " = " + testSum);
  assertEquals(sum, testSum);
 }
}
```

Exemplu simplu cu JUnit

eclipse-workspace - JUnitTestingExample/src/CalculateTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer | JUnit

Finished after 0.175 seconds

Runs: 1/1 Errors: 0 Failures: 0

CalculateTest [Runner: JUnit 5] (0.009 s)

1 CalculateTest
2 Tst
3 MainException
4 TestMultipleCatchBlock
5 AssertDemo
6 Demo
7 MainClass (3)
Run As → 1 JUnit Test Alt+Shift+X, T
Run Configurations...
Organize Favorites...

Catch... | ExceptionThrown.j... | MainException.java | Tst.java | Calculate.java | CalculateTest.java

```
Assert.assertEquals;
.api.Test;
st {
n = new Calculate();
) {
11        System.out.println("@Test sum(): " + sum + " = " + testSum);
12        assertEquals(sum, testSum);
13    }
14
15 }
16
```

Failure Trace

Problems | Javadoc | Declaration | Console

<terminated> CalculateTest [JUnit] C:\Program Files\Java\jre1.8.0_144\bin\javaw.exe (Dec 10, 2018, 12:32:18 PM)

```
Adding values: 2 + 5
@Test sum(): 7 = 7
```

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)

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

- Unii programatori preferă depanarea (*debugging*) în locul jurnalizării (*logging*)

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 returnată:

```
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
  - Nivelul OFF care poate fi folosit la dezactivarea jurnalizării
  - Se pot defini nivele individualizate (vezi documentaţia Java!)

Exemplu pentru Logger

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
public class LoggerExample {
  private static final Logger LOGGER =
                           Logger.getLogger(LoggerExample.class.getName());
  public static void main(String[] args) throws SecurityException,
                                                                 IOException {
        LOGGER.info("Logger Name: "+LOGGER.getName());
        LOGGER.warning("Can cause ArrayIndexOutOfBoundsException");
        //An array of size 3
        int []a = {1,2,3};
        int index = 4;
        LOGGER.config("index is set to "+index);
        try{
            System.out.println(a[index]);
        }catch(ArrayIndexOutOfBoundsException ex){
            LOGGER.log(Level.SEVERE, "Exception occur", ex);
        }
    }
}
```

Avantajele jurnalizării

- Jurnalizarea poate genera informaţii detaliate despre funcţionarea unei aplicaţii
- După ce a fost adăugată la aplicaţie, nu mai are nevoie de intervenţia umană
- Jurnalele de aplicaţie pot fi salvate şi studiate ulterior
- Prin surprinderea erorilor care nu pot fi raportate utilizatorilor, jurnalizarea poate ajuta 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 (*multithreaded*)
- Jurnalizarea rămâne împreună cu aplicaţia şi poate fi folosită oricând se rulează aplicaţia

Costurile jurnalizării

- Jurnalizarea adaugă o încărcare suplimentară la execuţie datorată 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

- Depanator (*debugger*)= 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"
- Depanatorul permite execuţia linie cu linie, instrucţiune cu instrucţiune
- La fiecare pas se pot examina valorile variabilelor

Despre depanatoare

- 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

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
    - Unul mai nou începând cu JDK 1.2 care se suprapune şi îl înlocuieşte partial pe primul
    - Pachetul **`java.nio`** din JDK 1.4 este şi mai nou
- 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"

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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 suportă accesul aleator
    - `System.in` şi `System.out` nu-l suportă
  - Fiecare fişier disc are o poziţie specială pentru indicatorul de fişier
    - Se poate citi sau scrie la poziţia curentă a indicatorului

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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 (numit şi acces direct):

Java I/O System

Random Access

Sequential Access

(Class `RandomAccessFile`)

(All other classes)

TECHNICAL UNIVERSITY

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)

Citire octeţi

Citire caractere

Citirea secvenţială a datelor

Bytes

Characters

ByteArray InputStream

StringBuffer InputStream

File InputStream

Piped InputStream

CharArray Reader

String Reader

File Reader

Piped Reader

InputStream Reader
(Punte între Byte Stream şi Character Stream)

(Opţional îmbunătă-ţeşte eficienţa)

Buffered InputStream

Buffer Reader

(Opţional îmbunătă-ţeşte eficienţa)

Data InputStream

Stream Tokenizer

Intrare formatată
readline()

Numere

Tokeni

Intrare ne-formatată
readInt(),
readDouble(),
readBoolean(),
etc.

Neformatat

Formatat

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

Scriere octeţi

Scriere caractere

Scrierea secvenţială a datelor

Bytes

Characters

ByteArray OutputStream

StringBuffer OutputStream

File OutputStream

Piped OutputStream

CharArray Writer

String Writer

File Writer

Piped Writer

OutputStream Writer

(Punte între Character Stream şi Byte Stream)

(**Opţional** îmbunătă-ţeşte eficienţa scrierii pe disc)

Buffered OutputStream

Buffer Writer

(**Opţional** îmbunătă-ţeşte eficienţa scrierii pe disc)

Data OutputStream

PrintStream

PrintWriter

Ieşire ne-formatată
writeInt(), writeBoolean, writeDouble, etc.

Ieşire formatată
print(), println()

Ieşire formatată
print(), println()

Neformatat

Formatat

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` – 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]);
}
// Interceptează alte IOExceptions
```

Scanner

- 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 apasă tasta "Enter"
  - *Token*-ii au mărimea liniilor 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 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 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>();
// Se adauga mai multe obiecte BankAccount in 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`

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

- Interfaţa `Serializable` nu are metode
- Serializare: procesul de salvare a obiectelor într-un flux
  - Fiecărui obiect îi este atribuit un număr de serie pe flux
  - Dacă 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

Exemplu Serializare/Deserializare

```
import java.io.*;
public class Angajat implements Serializable  {
  transient int a; //a nu va fi serializat datorita lui transient
  static int b; //b nu va fi serializat deoarece este static
  String name;
  int age;

  public Angajat(String name, int age, int a, int b)
  {
    this.name = name;
    this.age = age;
    this.a = a;
    this.b = b;
  }
}
```

Exemplu Serializare/Deserializare

```
import java.io.*;
public class ExempluSerializare {
  public static void afisareDate(Angajat object1) {
     System.out.println("name = " + object1.name);
     System.out.println("age = " + object1.age);
     System.out.println("a = " + object1.a);
     System.out.println("b = " + object1.b);
  }
  public static void main(String[] args) {
     Angajat object = new Angajat("Pop Dorel", 20, 2, 1000);
     String filename = "angajat.dat";
     // Serializare
     try {
          // Salveaza obiectul in fisier
          FileOutputStream file = new FileOutputStream (filename);
          ObjectOutputStream out = new ObjectOutputStream (file);
          out.writeObject(object);
          out.close(); file.close();
          System.out.println("Obiect serializat\n" + "Date inainte de deserializare:");
          afisareDate(object);
          object.b = 2000; // se schimba valoarea variabilei statice
       }catch (IOException ex) {
           System.out.println("IOException is caught");
       }
```

Exemplu Serializare/Deserializare

```
    // Deserializare
    object = null;
    try {
        // Citeste obiect din fisier
        FileInputStream file = new FileInputStream (filename);
        ObjectInputStream in = new ObjectInputStream (file);
        // Deserializeaza obiect
        object = (Angajat)in.readObject();
        in.close();
        file.close();
        System.out.println("Obiect deserializat\n Date dupa deserializare.");
        afisareDate(object);
      }
      catch (IOException ex) {
        System.out.println("IOException is caught");
      }
      catch (ClassNotFoundException ex) {
        System.out.println("ClassNotFoundException is caught");
      }
  }
}
```

Zone tampon și canale

- Zonele tampon au fost create în primul rând pe post de containere pentru datele (de tipuri primitive) trimise/recepţionate pe/de pe canale

Buffer

CharBuffer IntBuffer DoubleBuffer ShortBuffer LongBuffer FloatBuffer ByteBuffer

MappedByteBuffer

- Canalele sunt conducte spre servicii de I/E de nivel jos; ele sunt întotdeauna orientate pe octeţi; ele ştiu doar cum să folosească obiecte **`ByteBuffer`**

Vederi pentru `Buffer`

- Presupunem că avem un fişier care conţine caractere Unicode stocate ca valori pe 16 biţi (codificare UTF-16 nu UTF-8)
  - UTF = Unicode Transformation Format
- Pentru a citi o bucată din acest fişier în zona tampon putem crea o vedere `CharBuffer` a octeţilor respectivi:

```
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

  - Creează o vedere a `ByteBuffer` care se comportă ca un `CharBuffer` (combină fiecare pereche de octeţi din tampon într-o valoare caracter pe 16 biţi)
- Clasa `ByteBuffer` are metode de acces ad-hoc la valorile primitive
  - D.e., pentru a accesa ca întreg patru octeţi dintr-o zonă tampon

```
int fileSize = byteBuffer.getInt();
```

Exemplu de vederi pentru `Buffer`

```
import java.nio.*;
public class Buffers {
  public static void main(String[] args)   {
     try {
       float[] floats = {
           6.612297E-39F, 9.918385E-39F,
           1.1093785E-38F, 1.092858E-38F,
           1.0469398E-38F, 9.183596E-39F
       }
       ByteBuffer bb =
          ByteBuffer.allocate(floats.length * 4 );
       FloatBuffer fb = bb.asFloatBuffer();
       fb.put(floats);
       CharBuffer cb = bb.asCharBuffer();
       System.out.println(cb.toString());
     } catch (Exception e) {
       System.out.println(e.getMessage();
       e.printStackTrace();
     }
   }
}
```

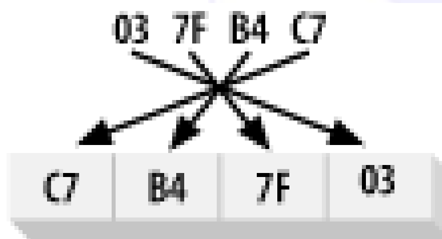
FloatBuffer: 6.612297E-39 | 9.918385E-39 | 1.0193785E-38 | 1.092858E-38 | 1.0469398E-38 | 9.183596E-39

CharBuffer: H | e | l | l | o | (space) | w | o | r | l | d | !

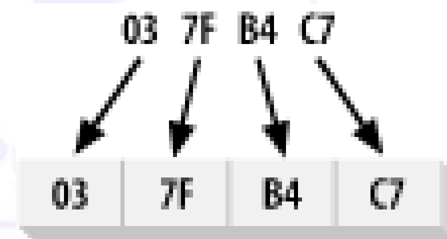
ByteBuffer: 00480065006C006C006F00200077006F0072006C00640021

Interschimbarea octeţilor

- *Endian-ness* : ordinea de combinare a octeţilor pentru a forma valori numerice mai mari
  - Când octetul cel mai semnificativ ca ordine numerică este primul stocat în memorie (la adresa mai mică) avem ordinea *big-endian*
  - Cazul opus, în care cel mai puţin semnificativ octet apare primul, este *little-endian*



little endian



big endian

Vederi ale **`Buffer`** şi Endian-ness

- Fiecare obiect tampon are o setare a *ordinii octeţilor*
  - Cu excepţia lui **`ByteBuffer`**, proprietatea poate fi numai citită şi nu se poate schimba
- Setarea ordinii octeţilor la obiectele **`ByteBuffer`** poate fi modificată oricând
  - Aceasta afectează ordinea rezultată pentru orice vederi create pentru acel obiect **`ByteBuffer`**
  - Dacă datele Unicode din fişier au fost codificate ca UTF-16LE (little-endian) trebuie să setăm ordinea pentru **`ByteBuffer`** înainte de a crea vederea **`CharBuffer`**:

```
byteBuffer.order(ByteOrder.LITTLE_ENDIAN);
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

- Noua vedere moşteneşte ordinea lui **`ByteBuffer`**
- Setarea ordinii octeţilor la momentul apelului *afectează* modul de combinare pentru formarea valorii returnate sau divizate pentru ceea ce este stocat în zona tampon

Citiri distribuitoare (*scatering*) şi scrieri colectoare (*gathering*)

- D.e., cu o singură cerere de citire de pe canal putem pune primii 32 octeţi în tamponul **`header`**, următorii 768 octeţi în tamponul **`colorMap`** şi restul în **`imageBody`**
- Canalul umple fiecare tampon pe rând până când toate sunt pline sau nu mai sunt date de citit

```
. . .
ByteBuffer header = ByteBuffer.allocate (32);
ByteBuffer colorMap = ByteBuffer (256 * 3);
ByteBuffer imageBody = ByteBuffer (640 * 480);
ByteBuffer [] scatterBuffers = { header, colorMap,
   imageBody };
fileChannel.read (scatterBuffers);
```

Transferuri directe pe canale

- Transferul pe canal permite interconectarea a două canale astfel încât datele să fie transferate direct dintr-un canal în celălalt fără intervenţii suplimentare

- Deoarece metodele **`transferTo()`** şi **`transferFrom()`** apartin clasei **`FileChannel`**, trebuie ca sursa şi destinaţia unui transfer pe canal să fie obiecte **`FileChannel`** (d.e., nu se poate transfera de la un *socket* la altul)

- Celălalt capăt poate fi orice **`ReadableByteChannel`** sau **`WritableByteChannel`**, după nevoi

Exemplu de transfer direct între canale

```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class DirectChannelTransfer {
  public static void main(String args[])  throws IOException {
    // verifica argumentele de pe linia de comanda
    if (args.length != 2) {
      System.err.println("Lipsesc numele de fisiere");
      System.exit(1);
    }
    // get channels
    FileInputStream fis =  new FileInputStream(args[0]);
    FileOutputStream fos =  new FileOutputStream(args[1]);
    FileChannel fcin = fis.getChannel();
    FileChannel fcout = fos.getChannel();
    // executa copierea fisierului
     fcin.transferTo(0, fcin.size(), fcout);
    // incheie
    fcin.close();
    fcout.close();
    fis.close();
    fos.close();
  }
}
```

Metoda **transferTo** transferă octeţi din canalul sursă (**fcin**) în canalul destinaţie specificat (**fcout**). Transferul este executat tipic *fără citiri şi scrieri explicite la nivel utilizator* pe canal.

Expresii regulate

- Expresiile regulate (`java.util.regex`) sunt parte a NIO
- Clasa `String` ştie de expresii regulate prin adăugarea următoarelor metode:

```
package java.lang;
public final class String implements java.io.Serializable,
  Comparable, CharSequence
 {
  // Lista partiala din API
  public boolean matches (String regex)
  public String [] split (String regex)
  public String [] split (String regex, int limit)
  public String replaceFirst (String regex, String
                                          replacement)
  public String replaceAll (String regex, String
                                          replacement)
 }
```

Exemple de expresii regulate

```
public static final String VALID_EMAIL_PATTERN =
    "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@" +
    "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$";
...
if (emailAddress.matches (VALID_EMAIL_PATTERN))
{
  addEmailAddress (emailAddress);
}
else
{
  throw new IllegalArgumentException (emailAddress);
}

// imparte sirul lineBuffer (care contine o serie de valori separate prin
    virgule) in subsiruri si returneaza sirurile respective intr-un tablou
String [] tokens = lineBuffer.split ("\\s*,\\s*");
```

Exemple de expresii regulate

```
public static final String VALID_EMAIL_PATTERN =
    "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@" +
    "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$";
```

^	**începutul liniei**
[_A-Za-z0-9-\\+]+	**trebuie sa înceapă cu String-ul din [ ], trebuie să conțină unul sau mai multe (+)**
(	**început grup #1**
\\.[_A-Za-z0-9-]+	**urmat de un punct "." și String-ul din paranteze [ ], trebuie să conțină unul sau mai multe (+)**
)*	**sfârșit grup #1, acest grup este optional (*)**
@	**trebuie să conțină simbolul "@"**
[A-Za-z0-9-]+	**urmat de String-ul din paranteze [ ], trebuie să conțină unul sau mai multe (+)**
(	**început grup #2**
\\.[A-Za-z0-9]+	**urmat de un punct "." și String-ul din paranteze [ ], trebuie să conțină unul sau mai multe (+)**
)*	**sfârșit grup #2, acest grup este opțional (*)**
(	**început grup #3**
\\.[A-Za-z]{2,}	**urmat de punct "." și String-ul din paranteze [ ], cu lungimea minimă 2**
)	**sfârșit grup #3**
$	**sfârșitul liniei**

Exemplu cu expresii regulate

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class EmailValidator {
  private Pattern pattern;
  private Matcher matcher;
  private static final String EMAIL_PATTERN =
        "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@" +
        "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$";

  public EmailValidator() {
     pattern = Pattern.compile(EMAIL_PATTERN);
  }

  public boolean validate(final String hex) {
     matcher = pattern.matcher(hex);
     return matcher.matches();
  }
}
```