Lab. 6: Tratarea Excepţiilor.
Colecţii

Cuprins

Recapitulare laborator 5
Excepţii în Java
    Best Practices
Colecţii
    Parcurgerea colecţiilor
    Liste. Clasele LinkedList şi ArrayList
    Algoritmi
Exerciţii

Recapitulare laborator 5

În laboratorul 5 am studiat interfeţele în Java şi necesitatea lor în crearea de prototipuri pe care clasele le pot implementa. Intuitiv, interfeţele determină un "contract", un protocol între clase, respectiv o clasă care implementează o interfaţă trebuie să implementeze metodele definite în acea interfaţă. Spre deosebire de cazul claselor, unde se poate moşteni cel mult o clasă, interfeţele pot extinde oricâte alte interfeţe. Modul în care se realizează aceasta este explicat în laboratorul 5.

Una dintre problemele rezolvate cu ajutorul interfeţelor este cea a moştenirii multiple în Java. Practic, dacă plecând de la clasele `C1, C2, ..., Cn` dorim să construim o nouă clasă care să moştenească unele dintre metodele lor, putem crea interfeţele `I1, I2, ..., In`, implementate respectiv de clasele `C1, C2, ..., Cn`, iar clasa noastră să implementeze toate acele interfeţe. În fine, putem folosi interfeţe pentru a trimite metode ca parametrii.

 Încercaţi să răspundeţi la următoarele întrebări:

- Prin ce se diferenţiază o interfaţă de o clasă în Java ?

- Cum se realizează extinderea interfeţelor ?
- Care sunt beneficiile utilizării interfeţelor ?
- Cum se rezolvă problema moştenirii multiple în Java prin intermediul interfeţelor?
- La ce trebuie să avem grijă atunci când implementăm interfeţe în clasele noastre ?
- Prin ce se diferenţiază o interfaţă de o clasă abstractă ?

Excepţii în Java

Referinţă: https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Best Practices

1. **Folosiți excepții specifice** - Clasele de bază de excepții nu dau informații utile, de aceea în Java există multe clase de excepții. De exemplu, în loc de IOException mai bine folosiți FileNotFoundException etc. Folosind excepții specifice, este mai uşor de făcut debug şi ajută aplicația să trateze excepțiile corespunzător.
2. **Throw Early or Fail Fast** - Este de preferat să aruncăm excepții cât mai repede. Când se face debug, un stack trace lung este mai greu de urmarit decât unul scurt.
3. **Prinderea târzie** - Java obligă ca o excepție sa fie ori tratată ori declarată în semnătura metodei. Uneori programatorii o prind şi o reţin într-un log, dar este periculos deoarece programul nu află de excepție. Excepțiile ar trebui prinse când pot fi tratate corespunzător. Aceeaşi metodă poate fi folosită de mai multe aplicații şi excepțiile pot fi tratate diferit de fiecare, de aceea e de preferat ca excepțiile să fie tratate de apelant.
4. **Închiderea resurselor** - Deoarece excepțiile opresc programul, este de preferat să închidem toate resursele folosite in blocul finally sau începând cu Java 7 se poate folosi try-with-resources şi lăsăm maşina virtuală să le închidă.
5. **Logarea excepțiilor** - Este bine ca întotdeauna sa logăm mesajele de excepție chiar dacă apelantul va şti de ce a apărut acea excepție. Ar trebui evitate blocurile catch care doar consumă excepțiile şi nu oferă infomații care să ajute la debug.
6. **Prinderea mai multor excepții cu acelaşi catch** - De obicei, logăm detaliile excepției şi anunțăm şi userul. În cazul Java 7 putem folosi prinderea excepțiilor cu un singur catch, astfel se reduce dimensiunea codului şi arată mai curat. Totuşi atenție să nu o folosiți şi atunci când este de preferat tratarea excepțiilor individual.
7. **Folosirea excepțiilor proprii** - Este bine să definim strategie de tratare a excepțiilor în etapa de design şi, în loc să aruncăm şi să tratăm excepții multiple, putem crea excepții proprii. Se poate crea o şi metodă care să se ocupe cu tratarea excepțiilor proprii.
8. **Convenții de numire şi organizare în pachete** - Când creați propriile excepții aveți grijă să se termine în Exception pentru a fi clar din nume că este o excepție. De asemenea,

aveți grijă să fie grupate asemănător ca în JDK, de exemplu IOException este clasa de bază pentru toate excepțiile generate de operațiile IO.

9. **Utilizați excepțiile cu grijă** - Excepțiile sunt costisitoare şi uneori nu este nevoie de ele, putem intoarce o variabilă booleană care să anunțe succesul sau eşecul operației. De exemplu, când se actualizează valori de la third party-uri nu este de dorit să aruncăm o excepție când conexiunea eşuează.
10. **Documentați excepțiile aruncate** - Folosiți cuvântul cheie din javadoc *@throws* pentru a specifica clar excepțiile aruncate de o metodă. Este foarte util când puneți la dispoziție o interfață ce va fi folosită de alte aplicații.

Colecţii

O colecție este un obiect care grupează mai multe obiecte într-o singură unitate, aceste obiecte fiind organizate în anumite forme. Prin intermediul lor, avem acces la diferite tipuri de date cum ar fi vectori, liste înlănțuite, stive, mulțimi matematice, tabele de dispersie, etc.

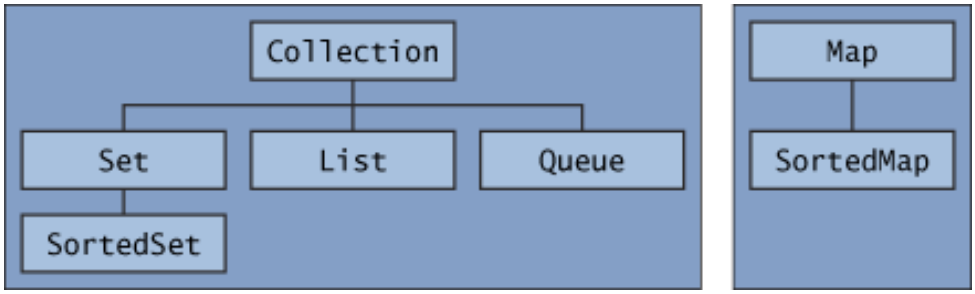
Colecțiile sunt folosite atât pentru memorarea şi manipularea datelor, cât şi pentru transmiterea unor informații de la o metodă la alta. Interfețele reprezintă nucleul mecanismului de lucru cu colecții, scopul lor fiind de a permite utilizarea structurilor de date *independent de modul lor de implementare.*

Majoritatea tipurilor care fac parte din categoria colecții puse la dispoziție de Java se află în pachetul java.util.

Există mai multe tipuri de colecții:

- **mulțime** (`HashSet`, `TreeSet`) - este o colecție care nu conține duplicate şi este menită să abstractizeze noțiunea matematică de mulțime
- **listă** (`LinkedList`, `ArrayList`, `Vector`, `Stack`) - colecție ordonată, eventual cu duplicate
- **coadă** (`PriorityQueue`, `ArrayDeque`)
- **dicționare de perechi cheie-valoare** (tabele de dispersie - `HashMap`, `HashTable`) - asociază unei chei o anumită valoare. Funcția nu este neapărat injectivă, deci mai multor chei li se poate asocia aceeaşi valoare. Atât cheile, cât şi valorile sunt obiecte.

Alegerea unei anumite clase depinde de natura problemei ce trebuie rezolvată. Mai multe detalii despre colecții puteți găsi la: https://docs.oracle.com/javase/tutorial/collections/



Observaţii:

- Există o interfaţă, numită Collection, pe care o implementează majoritatea claselor ce desemnează colecţii din `java.util`. Aceasta conţine metode utile precum:
  - `add(T)` - adaugă un obiect
  - `addAll(Collection)` - adaugă o întreagă altă colecţie
  - `clear()` - şterge toate elementele
  - `contains(T)` - verifică dacă obiectul respectiv există. Se foloseşte metoda `equals`!
  - `iterator()` - intoarce un `Iterator` cu care poate fi parcursă colecţia
  - `remove(T)` - şterge elementul din colecţie
  - `size()` - întoarce numărul de elemente din colecţie
  - `toArray()` - întoarce un vector de obiecte: `Object[]`
  - `isEmpty()` etc.
- Din versiunea Java 5 au fost introduse **tipurile generice** sau tipuri parametrizate. Astfel, programatorii pot specifica tipul obiectelor cu care o anumită clasă lucrează prin intermediul parametrilor de tip. De exemplu, pentru a defini o listă de numere întregi se poate folosi:

```
List<Integer> myList = new ArrayList<Integer>();
```

Parcurgerea colecţiilor

Colecţiile pot fi parcurse (element cu element) folosind:

- **iteratori**
- o construcţie **for** specială (cunoscută sub numele de **for-each**).

Un **iterator** este un obiect care permite traversarea unei colecţii şi modificarea acesteia (ex: ştergere de elemente) în mod selectiv. Puteţi obţine un iterator pentru o colecţie, apelând metoda sa `iterator()`.

Interfața **Iterator** este următoarea:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // optional
}
```

Metodele au următorul comportament:

- `hasNext` întoarce `true` dacă mai **există** elemente neparcurse încă de iteratorul respectiv
- `next` întoarce **următorul** element
- `remove` elimină din colecție ultimul element întors de `next`. În mod evident, `remove` nu poate fi apelat decât o singură dată după un apel `next`. Dacă această regulă nu este respectată, vom primi o eroare.

Este util să folosim iteratori când dorim:

- **ştergerea** elementului curent, în timpul iterării
- să iterăm mai multe colecții în paralel.

Exemplu:

```
Collection c  = new ArrayList();
Iterator   it = c.iterator();

while (it.hasNext()) {
     //verificari asupra elementului curent: it.next();
     it.remove();
}
```

Acest cod este *polimorfic*, ceea ce înseamnă că funcționează pentru **orice** colecție, indiferent de implementare.

Construcția **for-each** permite de asemenea traversarea unei colecții. **for-each** este foarte similar cu **for**. Se bazează, în spate, pe un iterator, pe care îl ascunde. Prin urmare **nu** putem şterge elemente în timpul iterării.

Următorul exemplu parcurge elementele unei colecții şi le afisează.

```
Collection collection = new ArrayList();
for (Object o : collection)
```

```
    System.out.println(o);
```

Liste. Clasele LinkedList şi ArrayList

`ArrayList` şi `LinkedList` sunt două implementări ale unei liste în Java. Alegerea colecţiei folosite într-un program depinde de natura problemei ce trebuie rezolvată. Complexităţile pentru principalele metode sunt:

	get	**add**	**contains**	**next**	**remove**
ArrayList	1	1	n	1	n
LinkedList	n	1	n	1	1

Observaţii:

- Accesarea unui element este mai rapidă (timp constant) pentru `ArrayList`, în timp ce pentru `LinkedList` este lentă, accesarea unui element într-o listă înlanţuită necesitând parcurgerea secvenţială a listei pâna la elementul respectiv.
- La eliminarea unui elemente din listă, pentru `ArrayList` este necesar un proces de reindexare (shift la stânga) a elementelor, în timp ce pentru `LinkedList` este rapidă, presupunând doar simpla schimbare a unor legături.
- Astfel, vom folosi `ArrayList` dacă avem nevoie de regăsirea unor elemente la poziţii diferite în listă, şi `LinkedList` dacă facem multe operatii de editare (ştergeri, inserări) în listă.
- De asemenea, `ArrayList` foloseste mai eficient spaţiul de memorie decât `LinkedList`, deoarece aceasta din urmă are nevoie de o structură de date auxiliară pentru memorarea unui nod.

Algoritmi

Colecţiile pun la dispoziţie metode care efectuează diverse operaţii utile cum ar fi: căutarea, sortarea definite pentru colecţii (pentru obiecte ce implementează interfeţe ce descriu colecţii). Aceşti algoritmi se numesc şi polimorfici deoarece aceeaşi metodă poate fi folosită pe implementari diferite ale unei colecţii.

Cei mai importanţi algoritmi sunt **metode statice definite în clasa `Collections`** şi permit efectuarea unor operatii utile cum ar fi cautarea, sortarea, etc. Algoritmi similari pentru tablouri se găsesc în clasa `java.util.Arrays`.

Mai multe detalii aici.

Exerciţii

1. Dându-se un fişier, să se afişeze frecvenţa cuvintelor care apar în el. De exemplu, dacă fişierul conţine “Ana are mere”, se va afişa:

```
Ana - 1
are - 1
mere - 1
```

   Observaţie: Nu uitaţi de `FileNotFoundException`
2. Să se creeze o listă care să nu permită introducerea de elemente duplicate. La introducerea unui element existent, semnalaţi eroare. Parcurgeţi lista folosind un iterator şi afişaţi elementele sale.
3. Creaţi o listă înlănţuită de elemente de tip `Persoană` (presupunem că aveţi o clasă `Persoană` implementată, dacă nu, o puteţi implementa). Adăugaţi 5 elemente în listă. Ordonaţi persoanele după nume şi, pentru nume egale, după vârstă. Hint: Folosiți interfaţa `Comparator`.
4. Se dau fişierele:

https://www.dropbox.com/sh/ngbni36c1x4l520/AAAp4Qo7kXQ1ozgAR_jEj7lHa?dl=0

Fişierul date.in conține pe prima linie un număr N şi pe următoarele N linii câte un număr întreg. Fişierul query.in conține pe prima linie un număr M şi pe următoarele M linii căte un număr întreg. Să se salveze numerele din fişierul date.in în colecţii bazate pe cele listate mai jos şi să se caute numerele din fişierul query.in în fiecare din acele structuri. Pentru fiecare tip de colecție să se afişeze atât timpul pentru adăugarea în colecție cât şi timpul pentru găsirea tuturor celor M elemente în colecția respectivă. Colecțiile folosite vor fi bazate pe:

- List
- Set
- Map

5. (*) Se dă o listă cu N elemente. Fiecare element din listă apare de 2 ori cu excepția unuia. Afişați acel element. Încercați să rezolvați problema fără memorie suplimentară (Lista în care se reţin elementele nu se consideră la memoria folosită de soluția voastră).
6. (*) Se dau două liste de numere întregi ordonate crescător. Să se găsească mediana celor două liste. Mediana este elementul aflat pe poziția din mijloc. Dacă lungimea este pară se poate afişa oricare dintre cele două elemente aflate pe poziția din mijloc. Care este cea mai bună complexitate ce se poate obține?
7. (***) Se dă o matrice binară de maxim 10^5 linii şi 500 de coloane. Să se găsească cardinalul maxim al unui subset de linii din matrice ştiind că 2 linii pot fi în acelaşi subset

doar dacă ele sunt formate doar din 0 sau doar din 1. De asemenea, aveți următoarea operație la dispoziție: se alege o coloană a matricii şi se inversează toate valorile de pe acea coloană (0 devine 1 şi 1 devine 0).