Metode Avansate de Programare
Pachete, Clase imbricate, Genericitate, Colecții în Java

Arthur Molnar
arthur.molnar@ubbcluj.ro

Universitatea Babeș-Bolyai

2023

Overview

Pachete I

- Utilizate pentru a grupa clasele și interfețele
- Ajută la evitarea conflictelor de nume și scrierea de cod sursă mai ușor de întreținut
- Platforma Java e livrată cu un număr de pachete deja existente: `java.lang`, `java.util`, `java.sql`, `java.time`, `java.time.format` și multe altele[1]
- Pachete definite de utilizator:
  - Se definesc folosind cuvântul cheie `package`, care trebuie să fie prima instrucțiune din fișierul `.java`
  - Pachetele se regăsesc într-o structură de directoare care reflectă numele lor; interfețele și clasele se salvează în directorul cu numele pachetului din care acestea fac parte

Pachete II

- Dacă un fișier `.java` nu include instrucțiunea `package`, interfețele și clasele declarate sunt parte din pachetul implicit (eng. *default package*)
- Pentru a utiliza clasa `ClassName` declarată în pachetul `PackageName`:

1.
```
        // ...
public static void main(String args[]){
    PackageName.ClassName obj =
                            new PackageName.ClassName();
}
```

2. **sau folosind un import generic**

```
        import PackageName.*
        // ...
public static void main(String args[]){
    ClassName obj = new ClassName();
}
```

Pachete III

- Dintr-un pachet putem importa unul, mai multe, sau toate interfețele și clasele
- Un fișier sursă Java poate conține oricâte instrucțiuni `import`, cât timp ele sunt primele instrucțiuni din fișier
- Pachetul `java.lang` (eng. *Java language*) este importat în mod implicit de compilatorul Java

[1] https://docs.oracle.com/en/java/javase/17/docs/api/allpackages-index.html

Clase imbricate I

- O clasă declarată în cadrul altei clase se numește clasă imbricată (eng. *nested class*)
- Utilizarea lor permite gruparea logică a claselor înrudite, sau a căror existență are logică doar în contextul oferit de o altă clasă

Clase imbricate II

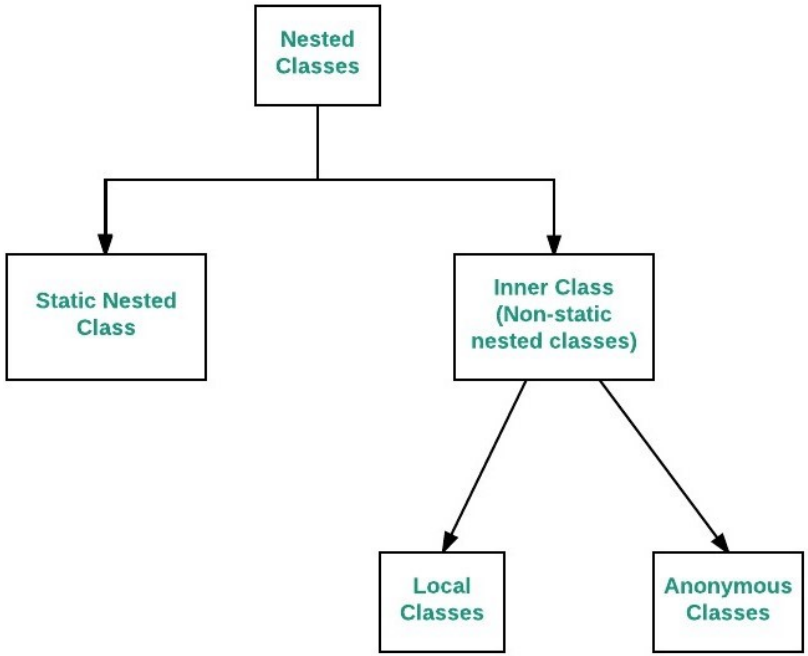

Figure: Sursa figurii: GeeksForGeeks

Clase imbricate III

- O clasă imbricată nu poate exista în mod independent de clasa în care a fost definită (eng. *outer class*)
- O clasă imbricată poate accesa toți membrii clasei în care a fost definită
- O clasă imbricată este un membru al clasei în care a fost definită. Astfel, ea poate fi declarată `private`, `protected`, `public` sau implicit (dacă nu e specificat un alt mecanism de protecție).
- O clasă imbricată poate fi instanțiată doar atunci când există o instanță a clasei în care a fost definită, cu excepția claselor imbricate marcate `static`[2]
- Pentru a accesa obiectul corespunzător clasei în care s-a definit clasa imbricată se utilizează expresia `Outer.this`, unde `Outer` este numele clasei externe

Clase imbricate IV

Exemplu

`lecture.examples.lecture3.NestedClasses.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

[2] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Genericitate în Java

- Care sunt clasele obiectelor conținute în instanța de `LinkedList` din exemplul anterior?
- Problema este că avem nevoie de un downcast pentru fiecare element referit din listă, atâta timp cât avem nevoie de ceva mai specific decât `java.lang.Object`

```
String elem = (String)iterator.element();
```

- Procesul poate induce erori la rulare, codul devine mai greu de citit

Tipuri de date generice I

- Un *tip generic* este o clasă sau interfață parametrizată
- Ca scop și funcționalitate, sunt asemănătoare șabloanelor în C++ (eng. *C++ templates*)
- Convenții de denumire a parametrilor:
  - **E** - Element (utilizat de *Java Collections Framework*)
  - **K** - Cheie (eng. *Key*)
  - **V** - Valoare (eng. *Value*)
  - **N** - Număr (eng. *Number*)
  - **T** - Tip (eng. *Type*)
- O clasă generică nu poate avea date membre generice. **De ce nu?**

Tipuri de date generice II

- Definirea unei clase generice:

```
class ClassName<T1, T2, ..., Tn> { /* ... */ }
```

- Instanțierea unui tip generic:

```
ClassName<Integer> = new ClassName<Integer>();
```

- Tipurile primitive (`int`, `byte`, `char`, `float`, `double`, ...) nu pot fi utilizate la instanțierea tipurilor generice
- Genericitatea în Java există la momentul compilării – compilatorul înlocuiește tipurile și metodele generice cu cast la `Object`, din acest motiv se pot utiliza doar tipuri convertibile la `java.lang.Object`[3]

Exemple

`lecture.examples.lecture3.generics.GenericLinkedList.java` și `lecture.examples.lecture3.generics.GenericStack.java`

[3] https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Metode generice

- Metode care utilizează propriile tipuri de parametri
- Pot fi statice sau nu
- Lista tipurilor de parametri generici e specificată în cadrul unor paranteze ascuțite, înaintea tipului de dată returnat de metodă

Exemplu

`lecture.examples.lecture3.GenericMethods.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Ștergerea tipurilor *(eng. type erasure)* I

- Spre deosebire de mecanismul șabloanelor în C++ (eng. *C++ templates*), în Java nu se creează câte o clasă nouă pentru fiecare instanță a unei clase generice
- Compilatorul Java aplică mecanismul ștergerii tipurilor:
  - toți parametrii generici sunt înlocuiți de `Object` sau de tipurile limitatoare (eng. *bounded type parameters*, dacă acestea sunt specificate)
  - sunt introduse cast-uri adiționale acolo unde este nevoie pentru a păstra verificările legate de tipuri
- Bytecode-ul produs nu include tipuri noi de date (în C++, fiecare instanțiere a unei clase șablon generează un nou tip de dată)

Ștergerea tipurilor *(eng. type erasure)* II

```
class Stack<E>
{
    // ...

    E top()
    {
        // ...
    }
}
```

- La compilare, referința la tipul **E** va fi înlocuită de referință la `Object`, cu cast-uri către tipul **E** introduse acolo unde este nevoie

Ștergerea tipurilor *(eng. type erasure)* III

```
class Stack
{
    // ...

    Object top()
    {
        // ...
    }
}
```

Parametri de tip cu limitări (eng. *Bounded Type Parameters*)

- Permit restrângerea tipurilor de dată ce pot fi utilizate într-un tip generic
- Pentru a specifica o limită superioară (dpdv. al relației de moștenire), numele parametrului trebuie să fie urmat de cuvântul cheie `extends`:

```
class Calculator<T extends Number>
class Calculator<T extends Number & Comparable>
```

- O variabilă de tipul **T** poate apela metode din clasele sau interfețele specificate ca și limite superioare

Exemplu

`lecture.examples.lecture3.BoundedTypeParameters.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Clasele generice și tipurile derivate

```
class Base { /* ... */ }
class Derived extends Base { /* ... */ }
```

- **Dar**: List<Derived> **nu este** o clasă derivată a List<Base> (vezi explicația)
- Nu există nici o relație între cele două tipuri

```
public static void printCollection
                        (List<Person> persons)
```

- Metoda de mai sus nu poate fi utilizată pentru o listă de studenți (`List<Student>`, unde `Student` este o clasă derivată din `Person`).

Wildcards

- Semnul de întrebare (?) este cunoscut ca un wildcard în programarea cu tipuri generice
- Reprezintă un tip necunoscut
- Wildcard mărginit superior (dpdv. al relației de moștenire):

```
public static void printCollection
                (LinkedList<? extends Person>)
```

- Wildcard mărginit inferior (dpdv. al relației de moștenire):

```
public static void printCollection
                (LinkedList<? super Student>)
```

Exemplu

`lecture.examples.lecture3.generics.Wildcards.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Colecții Java (eng. Java Collections Framework)

- O colecție este un obiect care reprezintă un grup de obiecte (ex. clasa `std::vector` din C++ STL, sau clasele `list`, `dict` din Python 3)
- Un cadru de aplicații pentru colecții (eng. *collection framework*) este o arhitectură unificată pentru reprezentarea și manipularea colecțiilor, ceea ce permite lucrul cu acestea în mod independent de detaliile reprezentării
- Avantaje:
  - Reutilizarea codului
  - Reducerea efortului necesar programării
  - Îmbunătățirea performanțelor
  - Algoritmi implementați polimorfic
  - Utilizarea tipurilor generice

Ierarhia de clase

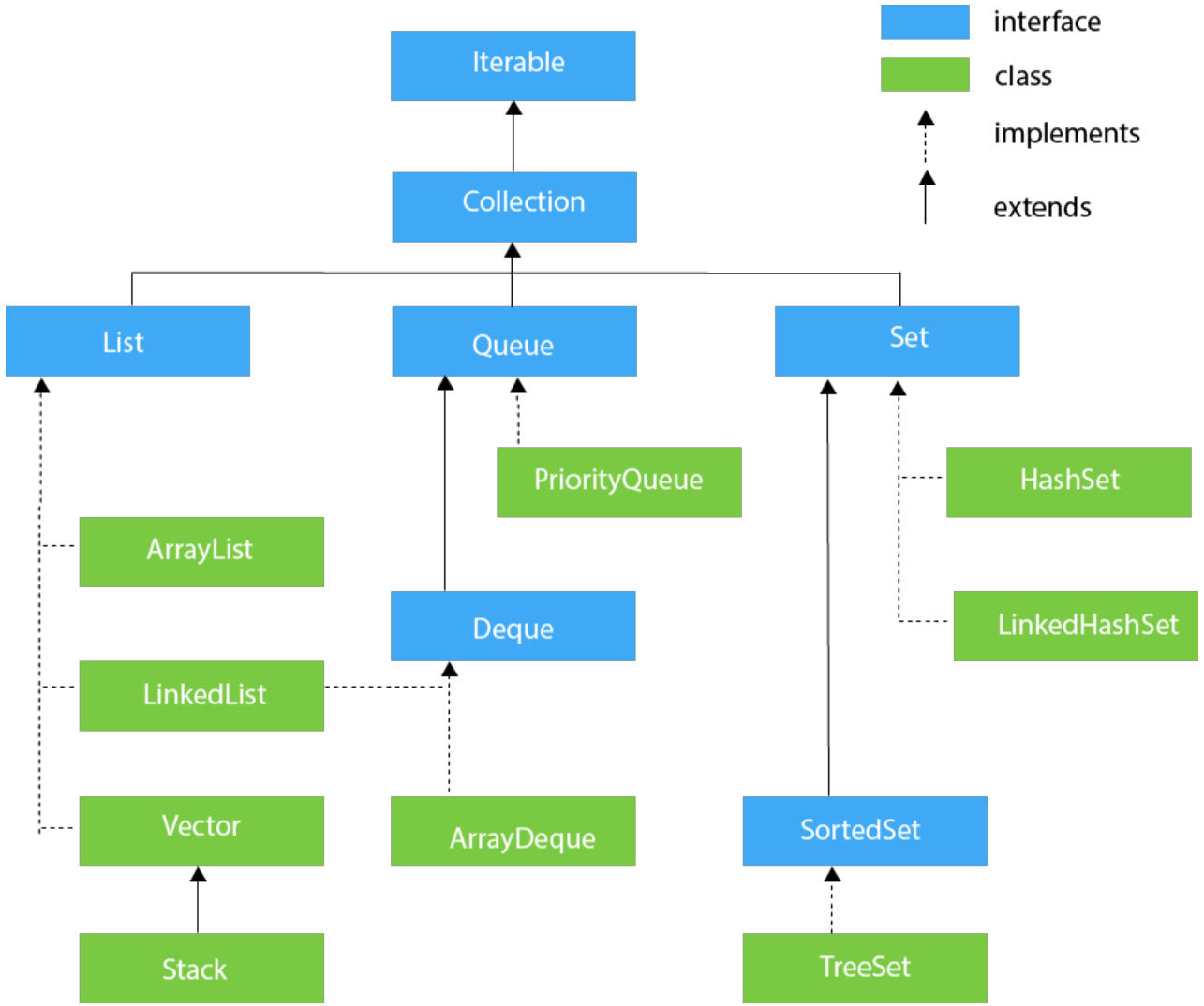

Figure: Sursa figurii: JavaTpoint

Implementări

Interfețe	**Implementare sub formă de**				
	Tabelă de dispersie	Tablou redimensionabil	Arbore	Listă înlănțuită	Tabelă de dispersie + listă înlănțuită
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Table: Detalii implementări colecții

Exemple

```
import java.util.*;
// ...
ArrayList<Integer> list = new ArrayList<>();
List<Integer> list = new ArrayList<>();
List<Integer> list = new LinkedList<>();
List<Integer> w = new Vector<>();
Map<Integer, String> map = new HashMap<>();
```

Interfețe pentru colecții

- `java.util.Collection<E>` este interfața rădăcină pentru ierarhia colecțiilor
- Include metode pentru operațiile de bază
  - `boolean add(Object)`
  - `boolean addAll(Collection)`
  - `void clear()`
  - `boolean contains(Object)`
  - `boolean containsAll(Collection)`
  - `boolean equals(Object)`
  - `boolean isEmpty()`
  - `Iterator iterator()`
  - `boolean remove(Object)`
  - `boolean removeAll(Collection)`
  - `boolean retainAll(Collection)`
  - `int size()`
  - `Object[] toArray()`
  - `Object[] toArray(Object[])`

Iteratori

- Furnizează o implementare generică pentru traversarea unei colecții, pentru accesarea și ștergerea elementelor unei colecții, indiferent de implementarea acesteia
- `java.util.Iterator<E>`
- Orice clasă ce implementează interfața `java.util.Collection<E>` trebuie să returneze un `java.util.Iterator<E>` prin metoda `iterator`
- Interfața `java.lang.Iterable<T>` este interfața rădăcină a ierarhiei
- Orice clasă ce implementează interfața *Iterable* poate fi iterată folosind bucla *for-each*

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

Exemplu

`lecture.examples.lecture3.Iterators.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbclui-ro/MAP

Compararea obiectelor I

- Există 2 moduri de a compara obiecte în Java:
  - Implementarea interfeței `java.lang.Comparable`:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

  - Prin utilizarea unui `Comparator`:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Compararea obiectelor II

- Obiectele comparabile pot fi comparate folosind un singur criteriu
- Pentru sortare, dacă obiectele implementează `java.lang.Comparable`, se spune că sunt sortate în ordine naturală (eng. *sorted by their natural order*) - obiectul în sine știe cum trebuie să fie sortat
- Pentru a permite mai multe criterii de sortare folosim `java.util.Comparator`
- `Comparator` este extern obiectului, fiind necesar câte o instanță pentru fiecare criteriu de sortare

Exemplu

`lecture.examples.lecture3.Comparators.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

java.util.Collections

- Clasa conține exclusiv metode statice pentru a manipula sau returna colecții
- Exemple:
  - Sortarea unei liste (`sort`)
  - Căutarea unui obiect (`binarySearch`)
  - Minimul sau maximul într-o colecție (`min`, `max`)
  - Inversarea ordinii, rearanjarea aleatorie (`reverse`, `shuffle`)
  - Copierea unei liste (`copy`).

Liste

- Clase ce implementează interfața `java.util.List`: `ArrayList`, `LinkedList`, `Vector`.
- `Vector` și `ArrayList` utilizează în mod intern un tablou și sunt redimensionabile
- Metodele clasei `Vector` sunt sincronizate, iar ale `ArrayList` nu
- `LinkedList` implementează o listă dublu înlănțuită

Mulțimi

- Clase ce implementează interfața `java.util.Set`: `HashSet`, `LinkedHashSet`, `TreeSet`.
- `HashSet` utilizează o tabelă de dispersie, ordinea elementelor în parcurgere nu e garantată, valorile duplicate nu sunt permise
- `LinkedHashSet` este o versiune ce implementează o listă dublu înlănțuită. Aceasta păstrează ordinea
- `TreeSet` extinde interfața *SortedSet*.

Dicționare

- Clase care implementează interfața java.util.Map: `HashMap`, `WeakHashMap`, `TreeMap`
- `HashMap` stochează perechi (cheie, valoare). Se folosește o tabelă de dispersie, în care elementele se accesează prin cheie iar cheile sunt iterabile
- `WeakHashMap` se aseamănă cu `HashMap`, dar când o cheie nu este utilizată în cadrul aplicației, acea intrare este ștearsă din memorie
- `TreeMap` utilizează un arbore roșu-negru ca structură de date

Alegerea corectă a colecției Java

- `HashMap` și `HashSet` au performanță ceva mai bună ca `LinkedHashMap`, dar ordinea elementelor la iterare este nedefinită. Accesarea elementelor e foarte rapidă
- `TreeSet` și `TreeMap` sunt ordonate și sortate, dar mai lente
- `LinkedList` poate adăuga elemente rapid la începutul listei, și elementele se pot șterge rapid prin iteratori. Accesarea elementelor aleatoare nu este rapidă
- Accesarea elementelor este rapidă în `Vector` și `ArrayList`. Inserarea și ștergerea elementelor nu sunt rapide

Exemplu

`lecture.examples.lecture3.Collections.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Rezumat

- Pachetele se utilizează pentru organizarea codului sursă și eliminarea conflictelor de nume
- Clasele imbricate permit gruparea logică a claselor între care există relație de compunere
- Utilizarea de metode și clase generice
- Java Collections Framework