Colecții și tipuri generice

1. Scopul lucrării

Obiectivele de învățare ala acestei sesiuni de laborator sunt:

- Înțelegerea modului de folosire a colecțiilor și a genericelor
- Acumularea cunoștințelor privind utilizarea claselor și interfețelor importante care sunt incluse în pachetul `java.util`
- Acumularea de experiență de programare în utilizarea colecțiilor Java.

O *colecție* (numită și container) este un obiect care grupează mai multe elemente într-o singură unitate. Colecțiile sunt folosite la stocarea, regăsirea, manipularea și comunicarea datelor agregate. De obicei ele reprezintă elemente de date care formează un grup natural, cum ar fi o mână la un joc de cărți (o colecție de cărți), un director de poștă electronică (o colecție de mesaje) sau o carte de telefon (o mapare de la nume la numere de telefon).

2. Cadrul de lucru Java Collections

Un *cadru de lucru* este o arhitectură unificată destinată reprezentării și manipulării colecțiilor. Toate cadrele de lucru din JCF (Java Collection Framework) conțin:

- **Interfețe:** tipuri de date abstracte care reprezintă colecțiile și care permit ca respectivele colecții să fie manipulate independent de detaliile de implementare.
- **Implementări:** implementări concrete ale interfețelor colecțiilor. Sunt, în esență, structuri de date reutilizabile.
- **Algoritmi:** metode care efectuează operații utile, cum sunt sortarea și căutarea, pe/în obiecte care implementează interfețele. Algoritmii sunt *polimorfi*: adică, aceeași metodă se poate folosi pe implementări diferite ale interfeței colecție corespunzătoare. Algoritmii sunt funcționalitate reutilizabilă.

Beneficiile Java Collections Framework (JCF)

Principalele beneficii rezultate din folosirea Java Collections Framework sunt:

- **Reducerea efortului de programare:** prin furnizarea de structuri de date și algoritmi, utile în dezvoltarea aplicațiilor; prin facilitarea interoperabilității între API-uri neînrudite.
- **Creșterea vitezei și calității programelor:** implementările sunt interschimbabile, astfel că programele pot fi ajustate prin schimbarea implementărilor colecțiilor.
- **Favorizarea reutilizării software:** noile structuri de date care se conformează interfețelor colecție standard sunt natural reutilizabile, lucru valabil și pentru algoritmii care operează pe obiectele care implementează respectivele interfețe.

În scrierea aplicațiilor putem să ne concentram eforturile asupra problemei în sine și nu asupra modului de reprezentare și manipulare a datelor.

Ierarhia JCF

Interfețele din JCF formează o ierarhie, cum se poate observa în figura 1.

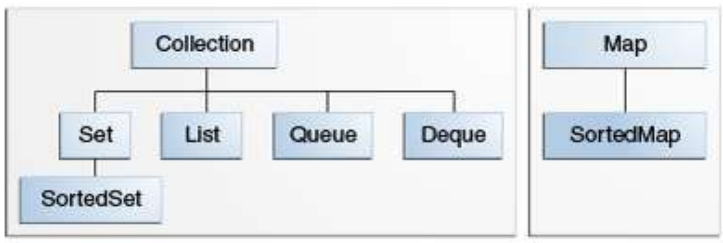

Figura 1. Principalele interfețe colecție

Rezumatul JCF

Lista următoare prezintă principalele componente ale JCF.

- `List` – Extinde *`Collection`*. Elementele sunt accesibile secvențial sau pe baza indexului lor.
- `Map` – Stochează perechi cheie-valoare, accesibile rapid pe baza cheii. D.e., figura 2 prezintă o mapare de la persoane la culorile favorite.
- `SortedMap` - Extinde *`Map`*, prin adăugarea accesului în ordine.
- `Set` - Extinde *`Collection`*. Conține doar valori unice.
- `SortedSet` - Extinde *`Set`*; elementele pot fi accesate în ordine.
- `Deque` – Coada cu două capete, folosită atât pentru operații cu *stivă* (LIFO) cât și cu *coadă* (FIFO).
- `Collections` – o clasă care conține un set de metode statice pentru lucrul cu structuri de date.

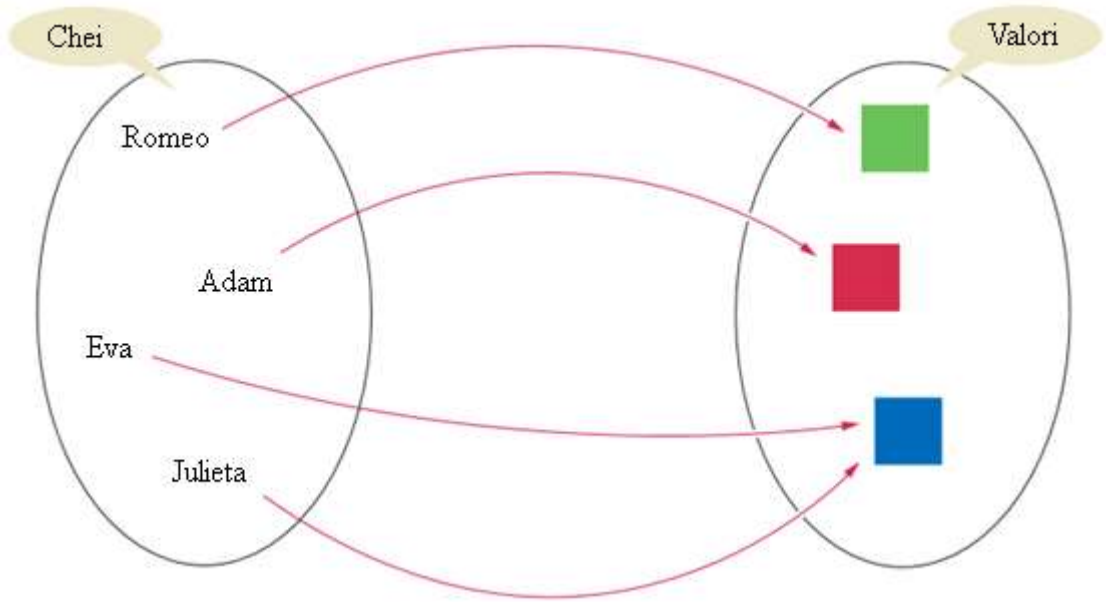

Figura 2. Un exemplu de mapare

Interfața `Collection` modelează un grup de obiecte numite elemente. Scopul acestei interfețe este să faciliteze folosirea colecțiilor la nivel maxim de generalitate. În definiția interfeței se observă trei categorii de metode.

```
public interface Collection<e> {
	// Operatii de baza la nivel de element
	boolean isEmpty();
	boolean contains(E e);
	boolean add(E e);                       // Optional
	boolean remove(E e);                    // Optional
	Iterator<E> iterator();
	// Operatii la nivel de colectie
	int size();
	boolean containsAll(Collection<E> c);
	boolean addAll(Collection<E> c);        // Optional
	boolean removeAll(Collection<E> c);     // Optional
	boolean retainAll(Collection<E> c);     // Optional
	void clear(); // Optional
	// Operatii de conversie in tablou unidimensional
	Object[] toArray();
	E[] toArray(E[] a);
}
```

Interfața `Set` Modelează noțiunea de mulțime în sens matematic. O mulțime nu poate avea elemente duplicate. Interfața `Set` definește aceleași metode ca interfața `Collection` .

3. Clase și interfețe importante

În cele ce urmează, cele mai utile clase sunt cu litere aldine (îngroșate).

```
AbstractCollection<E> implementeaza Collection<E>, Iterable<E>
    AbstractList<E> implementeaza List<E>
        ArrayList<E> implementeaza RandomAccess
        AbstractSequentialList<E>
            LinkedList<E> implementeaza Deque<E>
        Vector<E> implementeaza RandomAccess<E>  // Echivalent syncronizat al ArrayList
            Stack<E>  // Adds push(), pop(), and peek()
    AbstractSet<E> implementeaza Set<E>
        HashSet<E>
            LinkedHashSet<E>
        TreeSet<E> implementeaza SortedSet<E>
        EnumSet<E>   // Implementare Bitset pentru clasa Enum.
    AbstractQueue<E> implementeaza Queue<E>
        PriorityQueue<E>
    ArrayDeque<E> implementeaza Queue<E> Deque<E>
```

`Map` leagă o Cheie de o Valoare

```
AbstractMap<K, V> implementeaza Map<K, V>
    HashMap<K, V>
        LinkedHashMap<K, V> // Cheile pot fi iterate in ordinea inserarii.
    TreeMap<K, V> implementeaza SortedMap<K, V>
    EnumMap<K, V>          // Cheile trebuie sa fie din aceeasi clasa Enum.
    WeakHashMap<K, V>      // Utilizare speciala - Cheile sunt referinte slabe[1].
    IdentityHashMap<K, V>  // Utilizare speciala - Cheile trebuie sa fie identice.

Map.Entry<K, V>       // Map key/value pair.
```

Interfețe

```
Iterator<E>           // Interfata necesita hasNext(), next(), ?remove()
    ListIterator<E>   // Interfata
Comparator <T>         // Interfata necesita compare() si equals()
// urmatoarele interfete din java.lang sunt utilizate de obicei in colectii.
Iterable<T>          // Interfata necesita iterator()
Comparable<T>        // Interfata necesita compareTo()
```

Clase și interfețe concrete

Câteva dintre cele mai utile clase. Am omis interfețele utilitare, cum sunt Cloneable și Serializable.

Clasa	*Implementarea*
Cele mai utilizate clase	
`ArrayList`	Secvență de valori stocate într-un tablou re-dimensionabil
`LinkedList`	Secvență de valori stocate într-o listă înlănțuită
`HashMap`	Perechi cheie/valoare în tabelă de dispersie

[1] O *referință slabă* (*weak reference*), este o referință care nu e destul de tare ca să forțeze un obiect sa rămână în memorie. Referințele slabe permit colectorului de memorie reziduală să determine accesibilitatea unui obiect

TreeMap	Perechi cheie/valoare în arbore binar echilibrat
HashSet	Valoare unică, stocată în tabela de dispersie. Implementează Set.
TreeSet	Valoare unică, stocată în arbore binar echilibrat. Implementează Set.
Interfețe	
Collection	Metode comune tuturor structurilor de date
List	Metode fundamentale pentru List (listă). Implementate de ArrayList și LinkedList.
Map	Metode fundamentale pentru Map (mapare). Implementate de HashMap și TreeMap.
Map.Entry	Perechi cheie/valoare în Set returnate de Map.entrySet().
Set	Metode fundamentale pentru Set. Implementate de HashSet și TreeSet.
Iterator	Metode pentru iterare „înainte” (de la primul spre ultimul element)
ListIterator	Metode suplimentare pentru mers înapoi.
Clase specializate	
BitSet	Tablou de biți extensibil.
LinkedBlockingDeque	Poate avea o limită superioară fixă. Poate bloca obținerea unui element până când este adăugat un element.
LinkedHashMap	Tabelă de dispersie în care elementele pot fi accesate și în ordinea adăugării
LinkedHashSet	Tabelă de dispersie în care elementele pot fi accesate și în ordinea adăugării
WeakHashMap	Tabelă de dispersie care folosește referințe slabe
Preferences	Pentru opțiuni de program care să persiste
Properties	Pre-Java 2, comparat cu Preferences
Clase mai vechi pentru care există înlocuitor	
HashTable	Versiune mai veche, sincronizată de HashMap.
Vector	Versiune mai veche, sincronizată de ArrayList, încă în uz.
Clase învechite	
Dictionary	Clasă abstractă învechită. Nu o folosiți.

Implementări de interfețe

	Implementări			
Interfața	**Tablou**	**Arbore echilibrat**	**Listă înlănțuită**	**Tabelă de dispersie**
List	ArrayList		LinkedList	
Map		TreeMap		HashMap
Set		TreeSet		HashSet
Deque	ArrayDeque		LinkedList	

Perechi cheie-valoare

Perechile cheie-valoare sunt stocate în *mapări*.

Interfețe Map

- *Map* implementată de HashMap și TreeMap
- *SortedMap* implementată de TreeMap.
- *Map.Entry* care descrie metodele de acces la perechile cheie-valoare.

Clase care implementează `Map`

Câteva clase implementează interfața `Map`, inclusiv `HashMap`, `TreeMap`, `LinkedHashMap`, `WeakHashMap`, `ConcurrentHashMap`, și `Properties`. Cea mai utilă clasă este `HashMap`.

- `java.util.HashMap` este implementată cu o tabelă de dispersie. Timp de acces O(1). Intrările nu sunt sortate.
- `java.util.LinkedHashMap` este implementată cu o tabelă de dispersie. Timp de acces O(1). Intrările sunt sortate fie în ordinea introducerii, fie în ordinea ultimului acces, lucru util la implementarea politicii de caching LRU (least recently used – cel mai recent folosit).
- `java.util.TreeMap` este implementată ca arbore binar echilibrat. Timp de acces O(log N). Intrările sunt sortate.

Metode ale interfeței `Map`

Câteva dintre cele mai utile metode ale interfeței `Map` sunt rezumate mai jos. *m* este o `Map`, *b* este un `boolean`, *i* este un `int`, *set* este un `Set`, *col* este o `Collection`, *key* este un `Object` folosit pe post de cheie la stocarea unei valori, *val* este un `Object` stocat ca valoare asociată cheii.

Rezultat	Metodă	Descriere
	Adăugarea de perechi cheie valoare la o mapare	
obj =	*m*.**put**(*key*, *val*)	Creează o mapare de la *key* la *val*. Returnează valoarea anterioară asociată cu cheia respectivă (sau `null`).
	m.**putAll**(*map2*)	Adaugă toate intrările cheie-valoare dintr-o alta mapare, *map2*.
	Eliminarea perechilor cheie-valoare dintr-o mapare	
	m.**clear**()	Elimină toate elementele dintr-o mapare
obj =	*m*.**remove**(*key*)	Șterge maparea de la *key* la orice. Returnează valoarea anterioară asociată cu cheia respectivă (sau `null`).
	Regăsirea de informatie dintr-o mapare	
b =	*m*.**containsKey**(*key*)	Returnează `true` dacă *m* conține cheia *key*
b =	*m*.**containsValue**(*val*)	Returnează `true` dacă *m* conține *val* ca valoare
obj =	*m*.**get**(*key*)	Returnează valoarea corespunzătoare lui *key*, sau `null` dacă nu există mapare. Dacă a fost stocat `null` pe post de valoare, folosiți `containsKey` pentru a verifica dacă există o mapare.
b =	*m*.**isEmpty**()	Returnează `true` dacă *m* nu conține mapări (perechi).
i =	*m*.**size**()	Returnează numărul de mapări din *m*.
	Regăsirea tuturor cheilor, valorilor sau perechilor cheie-valoare (necesară pentru iterare)	
set =	*m*.**entrySet**()	Returnează mulțimea valorilor `Map`.Entry pentru toate mapările.
set =	*m*.**keySet**()	Returnează `Set` de chei.
col =	*m*.**values**()	Returnează o vedere Collection a valorilor din *m*.

Metodele din interfața `Map.Entry`

Fiecare element este o mapare și are o *cheie* si o *valoare*. Fiecare pereche cheie-valoare este salvată într-un obiect de tipul `java.util.Map.Entry`. Mulțimea acestor intrări se poate obține apelând metoda `entrySet()` a mapării. Iterarea peste o mapare se realizează prin iterarea peste acest set.

În următorul tabel, *me* reprezintă un obiect `Map.Entry`.

Rezultat	Metodă	Descriere
obj =	*me*.**getKey**()	Returnează cheia din pereche.
obj =	*me*.**getValue**(*key*)	Returnează valoarea din pereche.

obj =	*me*.**setValue**(*val*)	Operație *optional* și s-ar putea să nu fie suportată de toate obiectele `Map.Entry`. Setează valoare din pereche, ceea ce modifică `Map` căreia îi aparține respectiva pereche. Returnează valoarea originală

Constructorii clasei `HashMap`

`HashMap` are următorii constructori:

Rezultat	Constructor	Descriere
hmap =	`new HashMap()`	Creează o nouă `HashMap` având capacitatea inițială implicită 16 și un factor de încărcare 0.75.
hmap =	`new HashMap(`*initialCapacity*`)`	Creează o nouă `HashMap` având capacitatea inițială (`int`) specificată
hmap =	`new HashMap(`*initialCapacity*`,` *loadFactor*`)`	Creează o nouă `HashMap` având capacitatea inițială specificată și care nu va excede factorul de încarcare specificat (`float`).
hmap =	`new HashMap(`*mp*`)`	Creează o nouă `HashMap` cu elemente din *Map mp*

Metodele interfeței `SortedMap`

Interfața *SortedMap* este folosită de `TreeMap` și adaugă metode care să reflecte că o `TreeMap` este sortată.

Rezultat	Metodă	Descriere
comp =	`comparator()`	Returnează comparatorul folosit la compararea cheilor. `null` dacă se folosește ordinea naturală (d.e. în cazul lui `String`).
obj =	`firstKey()`	Cheia primului element (în ordinea sortată).
obj =	`lastKey()`	Cheia ultimului element (în ordinea sortată).
smp =	`headMap(`*obj*`)`	Returnează *SortedMap* cu toate elementele mai mici decât *obj*.
smp =	`tailMap(`*obj*`)`	Returnează *SortedMap* cu toate elementele mai mari sau egale cu *obj*.
smp =	`subMap(`*fromKey*`,` *toKey*`)`	Returnează *SortedMap* cu toate elementele mai mari sau egale cu *fromKey* și mai mic decât *toKey*.

Constructorii clasei `TreeMap`

`TreeMap` implementează metodele din interfețele *Map* și *SortedMap*. Spre deosebire de `HashMap`, `TreeMap` ține arborele binar echilibrat în ordine sortată după cheie. Dacă cheia are o ordine naturală (cum e cazul lui `String`) e bine, dar adesea va trebui să furnizați un obiect `Comparator` care să spună cum se compară cheile. Are următorii constructori:

Rezultat	Constructor	Descriere
tmap =	`new TreeMap()`	Creează o nouă `TreeMap`. Cheile sunt sortate în ordine naturală.
tmap =	`new TreeMap(`*comp*`)`	Creează o nouă `TreeMap` folosind comparatorul *comp* pentru a sorta cheile.
tmap =	`new TreeMap(`*mp*`)`	Creează o nouă `TreeMap` din *Map mp* folosind ordinea naturală.
tmap =	`new TreeMap(`*smp*`)`	Creează o nouă `TreeMap` din *SortedMap smp* folosind ordinea cheilor din *smp*.

Clasa `Collections`

Câteva dintre metodele **statice** utilitare, din clasa `java.util.Collections` sunt prezentate pe scurt în cele ce urmează.

Să presupunem că am scris următoarele declarații, în care T reprezintă un tip clasa sau interfață.

```
int i;
List<T> listC;   // Lista de obiecte Comparable.
List<T> list;    // Orice fel de lista. Nu trebuie sa fie Comparable.
Comparator <T> comp;
T key;
T t;
Collection<T> coll;  // Orice fel de Collection (List, Set, Deque).
Collection<T> collC; // Collection care implementeaza Comparable.
```

Rearanjare – Sortare, amestecare, . . .		
	`Collections.sort(listC)`	Sortează `listC`. Elementele trebuie să fie *Comparable<T>*. Sortare stabilă, O(N log N).
	`Collections.sort(list, comp)`	Sortează `list` folosind un comparator.
	`Collections.shuffle(list)`	Pune elementele din `list` în ordine aleatoare.
	`Collections.reverse(list)`	Inversează elementele din `list`.
Căutare		
`i =`	`Collections.binarySearch(listC, key)`	Caută `key` în `list`. Returnează indexul elementului sau o valoare negativă dacă nu-l găsește. Folosește căutarea binară.
`i =`	`Collections.binarySearch(list, key, comp)`	Caută `key` în `list` folosind `Comparator` comp.
`t =`	`Collections.max(collC)`	Returnează obiectul *Comparable* din `collC`, care are valoare maximă.
`t =`	`Collections.max(coll, comp)`	Obiectul din `coll` cu valoare maximă, folosind `Comparator comp`.
`t =`	`Collections.min(collC)`	Returnează obiectul *Comparable* din `collC`, care are valoare minimă.
`t =`	`Collections.min(coll, comp)`	Obiectul din `coll` cu valoare minimă, folosind `Comparator comp`.

Mai sunt multe altele, acestea sunt doar câteva.

Căutarea poate fi implementată și în clasele care implementează interfețele

Toate clasele *`List`* și *`Set`* implementează metoda `contains()`, care execută o căutare liniară pe liste (O(N)), una binară pe TreeSets (O(log N)) și una bazată pe funcții de dispersie pe HashSets (O(1)).

Mapările definesc `get()` pentru a căuta o cheie. Pentru `HashMap` căutarea este O(1), iar pentru `TreeMap` cautarea este O(log N).

Sortarea poate fi implicită într-o clasă care implementeaza interfețele

Datele din `TreeSet` și cheile din `TreeMap` sunt tot timpul sortate. Acestea sunt implementate ca arbori binari, astfel că inserarea și căutarea sunt ambele O(log N). Se poate folosi un iterator pentru a regăsi datele (`TreeSet`s) sau cheile (`TreeMap`s) în ordinea sortată. Fie clasa elementului trebuie să fie `Comparable`, sau trebuie furnizat un `Comparator`.

4. Comparatori

Definirea propriei ordini de sortare

Principala utilizare a comparatorilor este trecerea lor ca argumente la ceva ce sortează, fie una dintre metodele de sortare implicite, fie la o structură de date care sortează implicit (d.e. `TreeSet` sau `TreeMap`).

Interfața `java.util.Comparator`

Interfața `java.util.Comparator` poate fi folosită pentru a crea obiecte care să fie transmise metodelor de sortare sau structurilor de date care sortează. Un `Comparator` trebuie să definească o funcție `compare` care primește ca argumente două `Object` și returnează -1, 0, sau 1 (mai mic, egal, mai mare).

Comparatorii nu sunt necesari dacă există o ordine de sortare naturală

Comparatorii nu sunt necesari pentru tablourile de valori primitive sau tablourile sau colecțiile de obiecte care au o ordine naturală (cum sunt, d.e., `String, BigInteger` etc.)

`String` are și un comparator predefinit pentru sortarea fără a ține seama dacă literele sunt mari sau mici, `String.CASE_INSENSITIVE_ORDER` .

Exemplu de folosire a obiectelor `Comparator`

În exemplul următor se citesc fișierele dintr-un director și se sortează în două moduri.

```
// File: arrays/filelist/Filelistsort.java
// Scop: Listarea conținutului directorului implicit(home) al unui utilizator
//       Demonstreaza folosirea Comparators pentru a sorta acelasi tablou
//       dupa doua criterii diferite.
// Author: Fred Swartz 2006-Aug-23  Public domain.

import java.util.Arrays;
import java.util.Comparator;
import java.io.*;

public class Filelistsort {

    //=================================================== main
    public static void main(String[] args) {
        //... Creaza comparatorii pentru sortare.
        Comparator<File> byDirThenAlpha = new DirAlphaComparator();
        Comparator<File> byNameLength   = new NameLengthComparator();

        //... Creaza un obiect a File pentru directorul utilizatorului.
        File dir = new File(System.getProperty("user.home"));
        File[] children = dir.listFiles();

        System.out.println("Fisierele dupa director, apoi alfabetic ");
        Arrays.sort(children, byDirThenAlpha);
        printFileNames(children);

        System.out.println("Fisierele dupa lungimea numelui lor (cel mai lung primul)");
        Arrays.sort(children, byNameLength);
        printFileNames(children);
    }

    //========================================== printFileNames
    private static void printFileNames(File[] fa){
        for (File oneEntry : fa) {
            System.out.println("   " + oneEntry.getName());
        }
    }
}

/////////////////////////////////////////////// DirAlphaComparator
// Pentru a sorta directoarele inaintea fisierelor, apoi alfabetic.
class DirAlphaComparator implements Comparator<File> {

    // Interfata Comparator necesita definirea metodei compare.
    public int compare(File filea, File fileb) {
        //... Sorteaza directoarele inaintea fisierelor,
        //    altfel alfabetic fara a tine seama de majuscule/minuscule.
```

```
        if (filea.isDirectory() && !fileb.isDirectory()) {
            return -1;

        } else if (!filea.isDirectory() && fileb.isDirectory()) {
            return 1;

        } else {
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}

////////////////////////////////////////////// NameLengthComparator
// Pentru a sorta dupa lungimea numelui de fisier/director (cel mai lung primul).
class NameLengthComparator implements Comparator<File> {

    // Interfata Comparator necesita definirea metodei compare.
    public int compare(File filea, File fileb) {
        int comp = fileb.getName().length() - filea.getName().length();
        if (comp != 0) {
            //... daca lungimile sunt diferite, am terminat.
            return comp;
        } else {
            //... daca lungimile sunt egale, sorteaza alfabetic.
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}
```

5. Generice

Folosirea claselor generice este uzuală, scrierea lor, mai puțin. Genericele sunt în principal o cale de a permite autorilor de biblioteci să scrie ceva care să permită utilizatorilor să adapteze la tipurile proprii.

Problema de bază – tipuri restricționate sau complet deschise

Atunci când e nevoie să scrieți ceva care funcționează bine cu obiecte din multe clase sau interfețe date ca parametri, aveți o problemă. Când alegeți un tip T, atunci puteți folosi doar obiecte de acel tip sau ale subclaselor sale. Dacă doriți ceva mai general, atunci adesea trebuie să mergeți pe ierarhia de moștenire până la `Object`, ceea ce funcționează pentru toate tipurile, așa că e generalizat complet. Acesta este felul în care au fost scrise colecțiile Java până la Java 5 – adeseori aveau `Object` ca tip al parametrilor. Era foarte convenabil pentru implementatori, dar mai puțin folositor pentru utilizatori.

Restrângerea tipului. Pentru majoritatea structurilor de date există un singur tip care ar trebui de fapt folosit; spre exemplu, aveți un `ArrayList` de `String`, sau un `ArrayList` de `Date`, dar rareori le amestecați. Într-adevăr, adeseori este considerat stil foarte prost să le amestecați. Totuși, metodele de bibliotecă care lucrează cu `Object` permit folosirea și adăugarea din greșeală de tipuri diferite.

Tip-area statică vs. dinamică

Tip-area statică/tare. Unul dintre elementele de atracție ale Java este că are ceea ce se numește *„tip-are tare”* (*strong typing*) – tipul variabilelor (și alte elemente) este declarat, iar valorile asignate variabilei trebuie să fie de acel tip. Aceasta dă mai mult de lucru la scrierea programului pentru a le pune în ordine, dar mesajele de eroare date de compilator sunt o soluție mult mai bună decât să se permită codului să facă atribuiri de tip incorecte la execuție.

Tip-area slabă/dinamică este folosită în unele limbaje (d.e. în Ruby), care permit să se atribuie variabilelor valori de tipuri diferite, care apoi au tipul ultimei valori atribuite. Cei care au propus această soluție spun că: să nu trebuiască să ai grijă de specificarea corectă a tipurilor în codul sursă face codificarea mai rapidă, iar cu TDD (Test-Driven Development, dezvoltarea dirijată de testare) orice atribuiri greșite vor fi descoperite și corectate rapid.

Java folosește tip-area statică/tare, iar introducerea genericelor permite tip-are și mai tare.

Exemple care compară stilul vechi cu genericele

Exemplu ne-generic

```
// Utilizare tipica inainte de Java 5
List greetings = new ArrayList();
greetings.add("We come in peace.");
greetings.add("Take me to your leader.");
greetings.add("Resistance is futile.");

Iterator it = greetings.iterator();
while (it.hasNext()) {
    String aGreeting = (String)it.next();
    attemptCommunication(aGreeting);
}
```

Același exemplu folosind generice

```
// Acelasi exemplu folosind generice.
List<String> greetings = new
ArrayList<String>();
greetings.add("We come in peace.");
greetings.add("Take me to your leader.");
greetings.add("Resistance is futile.");

Iterator<String> it = greetings.iterator();
while (it.hasNext()) {
    String aGreeting = it.next();  // No
downcast.
    attemptCommunication(aGreeting);
}
```

Specificarea tipului elementelor din colecție are consecințe bune:

- Încercarea de adăugare a ceva de tip greșit dă eroare de compilare.
- Obținerea elementelor nu are nevoie de forțare de tip explicită (downcast), deși acest exemplu nu arată cât de des ne ajută acest lucru.
- Specificarea tipului oferă multă documentație.

Numirea parametrilor tip - T, U, ...

Deși sunt posibile multe nume pentru tipul parametrilor, tradițional aceștia sunt scriși cu litere mari luate din secvența T, U, V, ... Alte litere mari se folosesc atunci când au semnificație specifică d.e. K pentru tipul unei chei (*key*), E pentru tipul unui *element*.

Citirea parametrilor tip

Notație	Semnificație
`List<T>`	`List` de elemente de tip T (T este *un tip concret*)
`List<?>`	`List` de orice tip (? este tip metacaracter nelimitat - *unbounded wildcard*)
`List<? super T>`	`List` de orice tip (? este tip metacaracter limitat - *bounded wildcard* – supertip al lui T)
`List<? extends T>`	`List` de orice tip (? este tip metacaracter limitat - *bounded wildcard* – subtip al lui T)
`List<U extends T>`	`List` de orice tip (U trebuie sa fie supertip al lui T)

Unde NU puteți folosi tipuri generice în definirea unei clase/metode generice

Limitări, nu toate evidente:

- Nu puteți crea obiecte de tip T.
- Nu puteți crea tablouri de tip T.
- Nu le puteți folosi pentru statice.

Implementarea cu ștergerea tipului a tipurilor generice în Java

Pentru a păstra compatibilitatea cu implementările JVM (Java Virtual Machine), genericele Java sunt văzute doar de compilator. La momentul execuției, toată informația de tip generic a fost „ștearsă” și înlocuită cu `Object`. Acesta e un mod isteț de obținere a compatibilității, dar, de asemenea e cauza câtorva complicații, d.e. interzicerea tablourilor de tipuri generice.

Exemplu de clasă generică: o matrice generică

Codul următor implementează o clasă `GenericMatrix` și o subclasă `IntegerMatrix`.

```
public class IntegerMatrix extends GenericMatrix<Integer> {
  @Override /** Aduna doi intregi */
  protected Integer add(Integer o1, Integer o2) {
    return o1 + o2;
  }

  @Override /** Inmulteste doi intregi */
  protected Integer multiply(Integer o1, Integer o2) {
    return o1 * o2;
  }

  @Override /** Specifica zero pentru un intreg */
  protected Integer zero() {
    return 0;
  }
}

public abstract class GenericMatrix<E extends Number> {
  /** Metoda abstracta pentru adunarea a doua elemente ale matricelor */
  protected abstract E add(E o1, E o2);

  /** Metoda abstracta pentru inmultirea a doua elemente ale matricelor */
  protected abstract E multiply(E o1, E o2);

  /** Metoda abstracta pentru definirea elementului zero al matricelor */
  protected abstract E zero();

  /** Aduna doua matrice */
  public E[][] addMatrix(E[][] matrix1, E[][] matrix2) {
    // Check bounds of the two matrices
    if ((matrix1.length != matrix2.length) ||
        (matrix1[0].length != matrix2[0].length)) {
      throw new RuntimeException(
        "Matricele au dimensiuni diferite ");
    }

    E[][] result =
      (E[][])new Number[matrix1.length][matrix1[0].length];

    // Efectueaza adunarea
    for (int i = 0; i < result.length; i++)
      for (int j = 0; j < result[i].length; j++) {
        result[i][j] = add(matrix1[i][j], matrix2[i][j]);
      }

    return result;
  }

  /** Inmulteste doua matrice */
  public E[][] multiplyMatrix(E[][] matrix1, E[][] matrix2) {
    // Verifica limitele
    if (matrix1[0].length != matrix2.length) {
      throw new RuntimeException(
        "Matricele nu au dimensiuni compatibile ");
    }

    // Creaza matricea rezultat
    E[][] result =
      (E[][])new Number[matrix1.length][matrix2[0].length];

    // Efectueaza inmultirea a doua matrici
```

```
    for (int i = 0; i < result.length; i++) {
      for (int j = 0; j < result[0].length; j++) {
        result[i][j] = zero();

        for (int k = 0; k < matrix1[0].length; k++) {
          result[i][j] = add(result[i][j],
            multiply(matrix1[i][k], matrix2[k][j]));
        }
      }
    }

    return result;
  }

  /** Tipareste matricele, operatorul si rezultatul operatiei */
  public static void printResult(
      Number[][] m1, Number[][] m2, Number[][] m3, char op) {
    for (int i = 0; i < m1.length; i++) {
      for (int j = 0; j < m1[0].length; j++)
        System.out.print(" " + m1[i][j]);

      if (i == m1.length / 2)
        System.out.print("  " + op + "  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m2.length; j++)
        System.out.print(" " + m2[i][j]);

      if (i == m1.length / 2)
        System.out.print("  =  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m3.length; j++)
        System.out.print(m3[i][j] + " ");

      System.out.println();
    }
  }
}

public class TestIntegerMatrix {
  public static void main(String[] args) {
    // Creaza tablourile de intregi m1, m2
    Integer[][] m1 = new Integer[][]{{1, 2, 3}, {4, 5, 6}, {1, 1, 1}};
    Integer[][] m2 = new Integer[][]{{1, 1, 1}, {2, 2, 2}, {0, 0, 0}};

    // Creaza o instanta de IntegerMatrix
    IntegerMatrix integerMatrix = new IntegerMatrix();

    System.out.println("\nm1 + m2 is ");
    GenericMatrix.printResult(
      m1, m2, integerMatrix.addMatrix(m1, m2), '+');

    System.out.println("\nm1 * m2 is ");
    GenericMatrix.printResult(
      m1, m2, integerMatrix.multiplyMatrix(m1, m2), '*');
  }
}
```

6. Iteratori

Colecțiile `List` și `Set` furnizează *iteratori*, care sunt obiecte care permit traversarea secvențială a întregii colecții. Interfața `java.util.Iterator<E>` prevede traversarea într-un singur sens, iar `java.util.ListIterator<E>` prevede traversarea în două sensuri (de la început la sfârșit și invers). `Iterator<E>` este un înlocuitor pentru clasa mai veche `Enumeration` care se folosea înainte ca să fie adaugate colecțiile în Java.

Crearea unui `Iterator`

Iteratorii sunt creați invocând metoda *`iterator()`* sau *`listIterator()`* a unei `List`, `Set`, sau a altei colecții de date cu iteratori.

Metodele unui `Iterator`

`Iterator` definește trei metode, dintre care una este opțională.

Rezultat	Metodă	Descriere
`b =`	*`it`*`.`**`hasNext`**`()`	`true` dacă mai sunt elemente de parcurs.
`obj =`	*`it`*`.`**`next`**`()`	Returnează obiectul următor. Dacă se accesează o lista generică, iteratorul va întoarce ceva de tipul listei. Iteratorii pre-generici returnau întotdeauna tipul `Object`, astfel încât era nevoie de obicei de o forțare de tip (downcast).
	`it``.remove()`	Elimină cel mai recent element care a fost returnat de `next`. Nu toate colectiile suportă `delete`. Va fi aruncata o excepție *UnsupportedOperationException* dacă colecția nu suportă *remove()*.

Exemplu cu generice

Un iterator ar putea fi folosit astfel:

```
ArrayList<String> alist = new ArrayList<>();
// . . . Adauga Strings la alist

for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    String s = it.next();  // Nu e nevoie de downcasting
    System.out.println(s);
}
```

Exemplul anterior cu for-each

```
for (String s: alist) {
    System.out.println(s);
}
```

Exemplu pre Java 5, cu iterator explicit și downcasting

În versiuni de Java < 5, iteratorul s-ar putea folosi astfel:

```
ArrayList alist = new ArrayList(); // are tipul Object.
// . . . Adauga Strings la alist

for (Iterator it = alist.iterator(); it.hasNext(); ) {
    String s = (String)it.next();  // Downcasting e necesar pre Java 5.
    System.out.println(s);
}
```

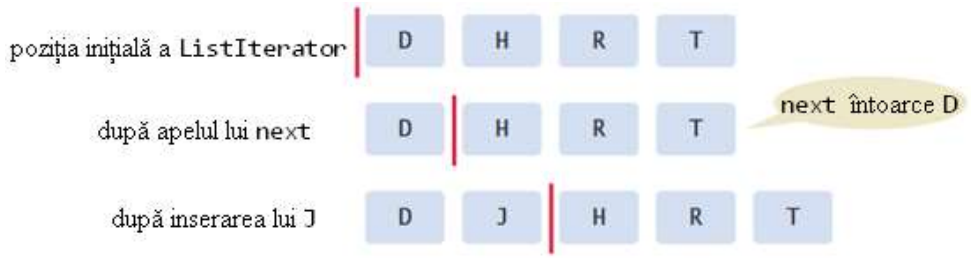

Figura 3. O vedere conceptuală a ListIterator. Observați pozițiile marcate cu roșu.

Metodele `ListIterator`

`ListIterator` este implementat doar de clasele care implementează interfața `List` (`ArrayList`, `LinkedList` și `Vector`). `ListIterator` furnizează următoarele:

Rezultat	Metodă	Descriere
Iterare înainte		
`b =`	*`it`*`.`**`hasNext()`**	`true` dacă există un element următor în colecție
`obj =`	*`it`*`.`**`next()`**	Returnează elementul următor.
Iterare înapoi		
`b =`	*`it`*`.`**`hasPrevious()`**	`true` dacă există un element precedent în colecție
`obj =`	*`it`*`.`**`previous()`**	Returnează elementul precedent.
Obținerea indexului unui element		
`i =`	*`it`*`.`**`nextIndex()`**	Returnează indexul elementului care ar fi returnat de un apel la *next()*.
`i =`	*`it`*`.`**`previousIndex()`**	Returnează indexul elementului care ar fi returnat de un apel la *previous()*.
Metode opționale de modificare. Aruncă UnsupportedOperationException dacă nu sunt suportate.		
	`it``.add(`*`obj`*`)`	Inserează *obj* în colecție la poziția „curentă” (înainte de următorul element care ar fi returnat de `next()` și după un element care ar fi returnat de `previous()`).
	`it``.set()`	Înlocuiește elementul „curent” (cel mai recent element returnat de un apel la `next` sau `previous()`).
	`it``.remove()`	Șterge elementul „curent” (cel mai recent element returnat de un apel la `next()` sau `previous()`).

Cum să NU faceți

Întrebare: Ce face bucla următoare? Observați amestecul de iterator și index.

```
ArrayList<String> alist = new ArrayList<String>();
// . . . Adauga Strings la alist

int i = 0;
for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    System.out.println(alist.get(i++));
}
```

Răspuns: Aruncă o excepție când depășește sfârșitul.

După ce `hasNext()` returnează `true`, singura modalitate de a avansa iteratorul este prin apelul lui `next()`. Dar elementul este obținut cu `get()`, așa că iteratorul nu avansează. `hasNext()` va continua să fie întotdeauna `true` (fiindcă există un prim element), și, în sfârșit, `get()` va cere ceva dincolo de sfârsitul lui `ArrayList`. Folosiți **fie** schema cu iterator:

```
for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    System.out.println(it.next());
}
```

Fie cea cu indexare, dar **nu le amestecați**:

```
for (int i=0; i < alist.size(); i++) {
    System.out.println(alist.get(i));
}
```

7. Exemple de aplicații folosind JCF

Un exemplu din Java Tutorial

Exemplul este din Oracle Java Tutorial.
Scrieți un program care să citească un fișier text, al cărui nume este primul argument de pe linia de comandă, într-o `List`. Programul va tipări apoi linii aleatoare din fișier, numărul de linii fiind specificat de cel de-al doilea argument din linia de comandă. Scrieți programul în așa fel, încât o colecție dimensionată corect se alocă toată odată, în loc să fie expandată gradat pe măsură ce se citește fișierul. Sugestie: Pentru a determina numărul de linii din fișier folosiți `java.io.File.length` pentru a obține mărimea fișierului, apoi împărțiți această dimensiune cu o mărime medie presupusă a unei linii.
Soluție: Deoarece accesăm aleatoriu lista vom folosi `ArrayList`. Estimăm numărul de linii din fișier împărțind cu 50 mărimea fișierului. Apoi dublăm valoarea obținută, deoarece este mai eficient să supraestimăm decât să subestimăm.

```
import java.util.*;
import java.io.*;

public class FileList {
    public static void main(String[] args) {
        final int assumedLineLength = 50;
        File file = new File(args[0]);
        List<String> fileList =
            new ArrayList<String>((int)(file.length() / assumedLineLength) * 2);
        BufferedReader reader = null;
        int lineCount = 0;
        try {
            reader = new BufferedReader(new FileReader(file));
            for (String line = reader.readLine(); line != null;
                    line = reader.readLine()) {
                fileList.add(line);
                lineCount++;
            }
        } catch (IOException e) {
            System.err.format("Nu pot citi %s: %s%n", file, e);
            System.exit(1);
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {}
            }
        }
        int repeats = Integer.parseInt(args[1]);
        Random random = new Random();
        for (int i = 0; i < repeats; i++) {
            System.out.format("%d: %s%n", i,
                    fileList.get(random.nextInt(lineCount - 1)));
        }
    }
}
```

Acest program consumă cel mai mult timp cu cititul fișierului, așa că pre-alocarea `ArrayList` are efecte minore asupra performanțelor sale. Precizarea capacității inițiale în avans este mai probabil să fie utilă atunci când programul Dvs. creează obiecte `ArrayList` fără operații de intrare/ieșire intercalate.

Exemplu de folosire a HashSet

Exemplul următor folosește HashSet pentru a număra cuvintele cheie dintr-un program sursă Java.

```
import java.util.*;
import java.io.*;

public class CountKeywords {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.print("Introduceti numele unui fisier sursa Java: ");
    String filename = input.nextLine();

    File file = new File(filename);
    if (file.exists()) {
      System.out.println("Numarul de cuvinte cheie din " + filename
        + " este " + countKeywords(file));
    }
    else {
      System.out.println("Fisierul " + filename + " nu exista");
    }
  }

  public static int countKeywords(File file) throws Exception {
    // Array of all Java keywords + true, false and null
    String[] keywordString = {"abstract", "assert", "boolean",
        "break", "byte", "case", "catch", "char", "class", "const",
        "continue", "default", "do", "double", "else", "enum",
        "extends", "for", "final", "finally", "float", "goto",
        "if", "implements", "import", "instanceof", "int",
        "interface", "long", "native", "new", "package", "private",
        "protected", "public", "return", "short", "static",
        "strictfp", "super", "switch", "synchronized", "this",
        "throw", "throws", "transient", "try", "void", "volatile",
        "while", "true", "false", "null"};

    Set<String> keywordSet =
      new HashSet<String>(Arrays.asList(keywordString));
    int count = 0;

    Scanner input = new Scanner(file);

    while (input.hasNext()) {
      String word = input.next();
      if (keywordSet.contains(word))
        count++;
    }

    return count;
  }
}
```

Exemplu de folosire a TreeMap

Exemplul următor folosește TreeMap pentru a contoriza aparițiile cuvintelor.

```
import java.util.*;

public class CountOccurrenceOfWords {
  public static void main(String[] args) {
    // Set text in a string
    String text = "Buan dimineata. Sa aveti ore cu folos. " +
      "Vizita placuta!. Distrati-va!";
```

```
    // Create a TreeMap to hold words as key and count as value
    Map<String, Integer> map = new TreeMap<String, Integer>();

    String[] words = text.split("[ \n\t\r.,;:!?(){}]");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

      if (key.length() > 0) {
        if (!map.containsKey(key)) {
          map.put(key, 1);
        }
        else {
          int value = map.get(key);
          value++;
          map.put(key, value);
        }
      }
    }

    // Get all entries into a set
    Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

    // Get key and value from each entry
    for (Map.Entry<String, Integer> entry: entrySet)
      System.out.println(entry.getKey() + "\t" + entry.getValue());
  }
}
```

8. Cum se alege o colecție?

Se poate face folosind, spre ghidare următorii pasi:

1. Determinați modul de acces la valori. Cum puteți accesa valorile individuale? Aveți mai multe opțiuni:
   - Valorile sunt accesate folosind o poziție întreagă. Utilizați un `ArrayList`
   - Valorile sunt accesate printr-o cheie care nu este o parte a obiectului. Utilizați o `Map`.
   - Valorile sunt accesate doar la unul dintre capete. Utilizați o coadă (pentru FIFO) sau o stivă (pentru LIFO).
   - Nu aveți nevoie de acces la valori individuale de poziție. Rafinați alegerea în pașii 3 și 4
2. Determinați tipul elementelor sau tipurile cheie/valoare. Pentru o listă sau set, determinați tipul de elementele pe care doriți să le stocați. De exemplu, dacă colecția reprezintă un set de cărți, atunci tipul de element este de `Book`. Asemănător pentru o mapare se determină tipurile cheilor și valorile asociate. Dacă doriți ca să căutați cărți după ID puteți folosi `Map <Integer, Book>` sau `Map <String, Book>`, în funcție de tipul de ID.
3. Determinați dacă ordinea elementelor sau a cheilor contează. Aveți câteva posibilități:
   - Elementele sau cheile trebuie sortate. Folosiți un `TreeSet` sau un `TreeMap`. Treceți la pasul 6.
   - Elementele trebuie regăsite în ordinea inserării. Folosiți `LinkedList` sau `ArrayList`.
   - Nu contează decât să puteți vizita toate elementele. Dacă ați ales `Map` la pasul 1 folosiți `HashMap` și treceți la pasul 5.
4. Determinați ce operații trebuie să fie eficiente pentru colecție. Aveți următoarele posibilități:
   - Regăsirea elementelor trebuie să fie eficientă. Folosiți un `HashSet`.
   - Trebuie să fie eficient să adăugați sau să ștergeți elemente la început, la sfârșit, sau în poziția curentă. Alegeți `LinkedList`
   - Inserați sau ștergeți doar la sfârșit sau colectați puține elemente astfel încât viteza nu contează. Folosiți `ArrayList`.

5. Pentru `HashSet` și `Map` decideți dacă e nevoie să implementați `hashcode` și `equals`. Aveți următoarele posibilități:
   - Dacă elementele sau cheile aparțin unei clase implementate deja, verificați dacă clasa are `hashcode` și `equals` proprii. Aceasta este valabil pentru clasele de bibliotecă Java cum sunt `String, Integer, Rectangle` etc.
   - Dacă nu, decideți dacă trebuie să comparați elementele pe baza identității. Acesta este cazul în care nu construiți două elemente distincte cu același conținut. Dacă acesta este cazul nu e nevoie de implementare pentru `hashcode` și `equals` . Implementarea din `Object` e suficientă.
   - Altfel trebuie să vă implementați propriile metode.
6. Dacă folosiți un arbore binar decideți dacă trebuie să furnizați un comparator. Verificati clasa setului de elemente sau de chei. Implementează `Comparable`? Dacă da, metoda `compareTo` sortează în ordinea corectă? Dacă da, nu trebuie făcut nimic. Iarăși, acesta este cazul claselor de bibliotecă Java. Dacă nu, trebuie să implementați interfața `Comparable` sau să declarați o clasă care implementează interfața `Comparator`.

9. Mersul lucrării

1. Studiați și executați codul prezentat pentru a înțelege modul de funcționare și utilizare a tipurilor din exemple.

2. Creați, pe modelul clasei `IntegerMatrix` matrice de alte tipuri (`DoubleMatrix, LongMatrix` etc.) și verificați că funcționează corect.

3. Faceți modificări în codul exemplelor date (d.e. adăugați metode, tipuri) și observați efectele modificărilor.

4. Implementați o clasa FacebookAccount care să simuleze o versiune simplificată a Facebook-ului. Un cont de Facebook ar trebui să conțină cel puțin: nume, vârsta, locația curentă și o listă de prieteni. Un obiect de tipul FacebookAccount ar trebui să poată adauga/șterge noi prieteni în lista lui de prieteni, de a afișa lista de prieteni, de a căuta toti prietenii care sunt dintr-o locație curentă specificată de utilizator. Indicații implementare: folosiți colecții pentru manipularea listei de prieteni.

5. Implementați o clasa PetHotel care să simuleze un registru cu toți câinii ce sunt cazați în hotel. Indicații implementare: Implementați problema utilizând colecții pentru manipularea câinilor din hotel.

6. Implementați o clasă TablaSah care să păstreze poziții pe tabla de șah. Figurile şi pionii sunt și ei clase. Verificați corectitudinea mutărilor. Indicații implementare: Implementați problema utilizând colecții pentru manipularea pieselor de pe tabla de șah.