Curs7

TreeSet ... 3
Crearea unui TreeSet ... 4
Adăugarea elementelor ... 5
Comparatorul ... 5
Returnarea elementelor ... 6
Folosirea submulțimilor ... 7
Sortarea colecțiilor ... 8
Interfața Comparable ... 8
Comparator ... 11
Dictionary, HashTable şi Properties ... 13
Clasa Dictionary ... 13
Clasa HashTable ... 14
Creearea HashTable ... 16
Adăugarea perechilor de chei-valoare ... 16
Ştergerea unei perechi ... 16
Mărimea unui HashTable ... 17
Operații cu HashTable ... 17
Returnarea obiectelor din HashTable ... 17
Căutarea elementelor ... 18
Verificarea egalității între HashTable ... 19
Clasa Properties ... 20
Setarea şi preluarea elmentelor ... 21
Încărcarea şi salvarea datelor ... 21
Map ... 22
Interfața Map.Entry ... 23
Clasa HashMap ... 23
Creearea unui HashMap ... 24
Adăugarea în HashMap ... 24
Ştergerea unui element ... 24
Operații cu HashMap ... 25
Clasa WeakHashMap ... 25
Clasa TreeMap ... 27
Crearea unui TreeMap ... 27
Operațiuni cu Map ... 28

Clasa Collections 30
Sortarea 30
Căutarea 31

TreeSet

Implementările interfeței Set, sunt HashSet şi TreeSet. Mai jos este o ierarhie completă a claselor abstracte şi interfețelor.

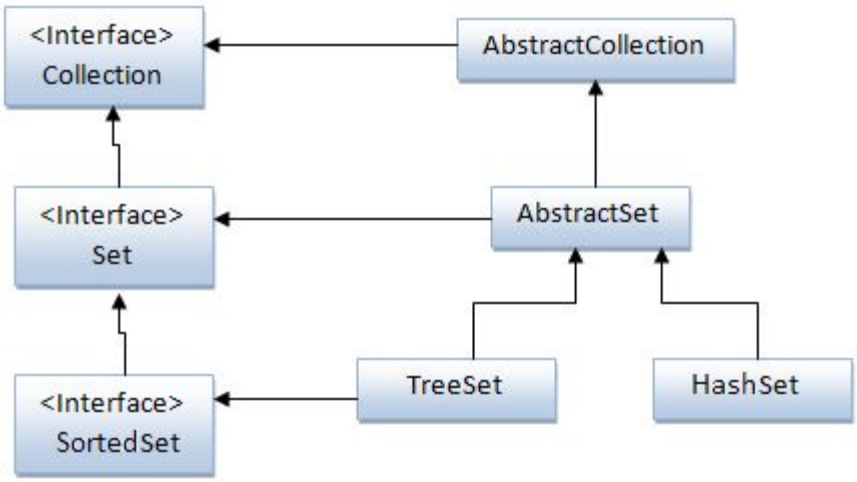

Figura 7.1 Ierarhia claselor Set

Clasa *TreeSet* funcționează ca şi un *HashSet* cu mențiunea că păstrează elementele într-o ordine. Aceste elemente sunt ordonate într-un arbore balansat şi anume un arbore roşu-negru. Păstrând elementele într-un arbore roşu-negru, costul unei căutări devine logaritmic si ordinul de complexitate este O(log n) .

Un arbore roşu-negru respectă următoarele reguli:

1. Fiecare nod este roşu sau negru
2. Rădăcina este întotdeauna un nod negru
3. Dacă nodul este roşu, copii lui trebuie sa fie de culoare neagră
4. Fiecare cale de la rădăcină la frunze trebuie sa conțină acelaşi număr de noduri negre.

Mai jos avem o schiță a unui arbore de acest tip:

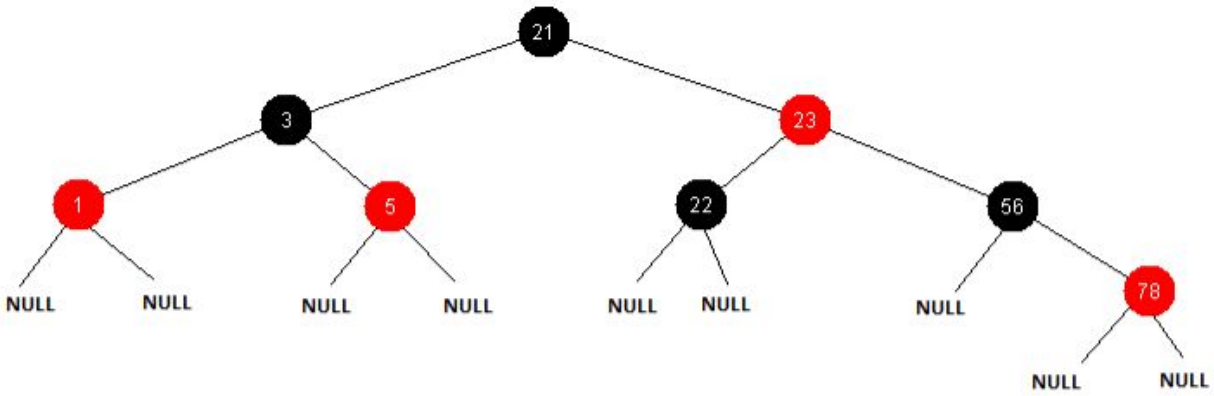

Figura 7.2 Arbore roşu-negru

Iată metodele implementate de această clasă:

Nume metoda	**Descriere**
TreeSet()	Constructorul unui *TreeSet*.
add()	Adaugă un element în mulțime
addAll()	Adaugă o colecție de elemente în mulțime
clear()	Şterge elementele din mulțime
clone()	Creează o clonă cu elementele din mulțime
comparator()	Returnează un obiect de tip Comparator
contains ()	Verifică existența unui obiect în mulțime
first()	Returnează primul element din mulțime
headSet()	Returnează un subset de elemente de la începutul mulțimii
isEmpty()	Verifică dacă mulțimea este goală.
iterator()	Returnează un obiect din set ce permite vizitarea mulțimii
last()	Returnează ultimul element din şir
remove()	Şterge un element din mulțime
size()	Returnează numărul de elemente din subset
subSet()	Returnează o submulțime din mulțimea inițială
tailSet()	Returnează o submulțime de elemente de la sfârşitul mulțimii inițiale

Crearea unui TreeSet

Clasa oferă patru constructori. Primii doi creează TreeSet-uri goale:

```
public TreeSet()
public TreeSet(Comparator comp)
```

Pentru a menține o anumită ordine în această structură, elementele adăugate în arbore trebuie să fie sortate într-un anume fel. Al doilea constructor permite specificarea obiectului de tip Comparator, ce va ajuta la sortarea acestei structuri.

Al doilea set de constructori, constituie constructori de copiere:

```
public TreeSet(Collection col)
public TreeSet(SortedSet set)
```

Dacă o colecție transmisă ca parametru, este de tip *SortedSet,* atunci clasa *TreeSet* va efectua unele optimizări, la adăugarea elementelor.

Adăugarea elementelor

Pentru a adăuga un singur element se poate apela metoda *add*:

Metoda *add* este aceeaşi ca şi pentru *HashSet*. Diferența majoră este că, elementele ce sunt adăugate trebuie să implementeze interfața *Comparable* sau constructorul *TreeSet* trebuie să aibă un parametru de tip *Comparator*. Dacă nici una din aceste condiții nu este adevărată, atunci va fi aruncată excepția *ClassCastException*. Fiecare element din această colecție trebuie să fie comparabile.

Comparatorul

Metoda *comparator()* returnează obiectul *Comparator* al acestui arbore:

```
public Comparator comparator()
```

Această metodă nu este apelată frecvent, dar dacă ne interesează ce tip de comparator este folosit, poate fi utilă. Metoda va returna null, în cazul în care se alege ordinea naturală de sortare a elementelor.

Mai jos este un exemplu de sortare:

```
import java.util.*;
public class Comparare
{
    public static void main (String args[]) throws Exception
    {
        String elements[] = {"Radu", "Andrei", "Ion","Vasile",
"Mircea"};
        //creearea unui set cu un comparator
        Set set = new TreeSet(Collections.reverseOrder());
        for (int i=0, n=elements.length; i<n; i++) {
        set.add(elements[i]);
        }
        //afisez elementele setului
        System.out.println(set);
```

```
        //afisez comparatorul actualului set
        System.out.println(((TreeSet)set).comparator());
    }
}
```

Acest program va avea ca rezultat, în urma rulării:

```
[Vasile, Radu, Mircea, Ion, Andrei]
java.util.Collections$ReverseComparator@3e25a5
```

Vom reveni ulterior asupra metodelor de comparare.

Returnarea elementelor

Se pot folosi metodele *first()* şi *last()* pentru a returna primul sau ultimul element din *set*:

```
public Object first()
public Object last()
```

Aceste elemente de la capetele colecției se găsesc pe baza comparării elementelor şi anume primul element este cel mai mic iar ultimul este cel mai mare.

Dacă nu există nici un element în colecție, va fi aruncată excepția *NoSuchException*.

Un mod de a parcurge elementele unei colecții *TreeSet*,este metoda *iterator()*:

```
public Iterator iterator()
```

Din moment ce elemente sunt sortate, metoda va returna elementele în ordinea din arbore.

Mai jos este un exemplu pentru a parcurge elementele unui set:

```
import java.util.*;
public class Iterare
{
    public static void main(String args[])
    {
        String elements[] = {"Radu", "Andrei", "Ion","Vasile",
"Mircea"};
        Set set = new TreeSet(Arrays.asList(elements));
        Iterator iter = set.iterator();
        while (iter.hasNext())
        {
            System.out.println(iter.next());
        }
    }
}
```

Acest exemplu are ca rezultat:

```
Andrei
Ion
Mircea
Radu
Vasile
```

După cum se poate vedea, folosirea iteratorului este aceeaşi cu utilizările prezentate în cazul celorlalte colecții.

Folosirea submulțimilor

Din moment ce un *TreeSet*, este ordonat, un subarbore va fi tot ordonat. Iată două metode ce returnează un subarbore format din elementele celui original:

```
public SortedSet headSet(Object toElement)
public SortedSet tailSet(Object fromElement)
public SortedSet subSet(Object fromElement, Object toElement)
```

Toate metodele vor returna un obiect, o interfață a arborelui original care, reflectă elementele din colecția originală. Altfel spus, dacă ștergem un obiect din submulțime, el va dispare şi din arborele original. Dacă adăugam un element subarborelui, el va fi adăugat în arborele original. Mai jos avem un exemplu de utilizare a acestei funcții:

```
public static void main(String args[])
    {
        String elements[] = {"Radu", "Andrei", "Ion","Vasile",
"Mircea"};
        TreeSet set = new TreeSet(Arrays.asList(elements));
        SortedSet subset = set.subSet("Ion", "Vasile");
        System.out.println(subset);
        subset.remove("Mircea");
        subset.remove("Mihai");
        subset.add("Mihai");
        System.out.println(set);
    }
```

Rezultatul va fi:

```
[Ion, Mircea, Radu]
[Andrei, Ion, Mihai, Radu, Vasile]
```

Intervalul din care sunt preluate elementele este:

```
fromElement <= set view < toElement
```

Dacă dorim ca `toElement` să fie în subarbore, va trebui, fie să transmitem ca parametru, următorul element din colecție, fie să folosim un truc, şi anume să adăugăm ceva la sfârşit:

```
SortedSet headSet = set.headSet(toElement+"\0");
```

Dacă nu vrem ca primul element să fie în subarbore, va trebui să apelăm la un truc asemănător:

```
SortedSet tailSet = set.tailSet(fromElement+"\0");
```

Sortarea colecțiilor

În acest capitol vom vedea ce mecanism implementează colecțiile, pentru a sorta elementele, într-un mod cât mai eficient. Există în principiu două modalități de a sorta colecțiile: implementând interfața *Comparable* sau printr-un *Comparator* personalizat.

Interfața Comparable

Clasele definite de Java au deja implementate interfața *Comparable*. Interfața *Comparable* are o metodă şi anume *compareTo()*, ce defineşte modul în care se compară şi implicit, se sortează obiectele.

Mai jos avem o serie de clase ce implementează natural aceasta interfață:

Nume clasă	**Aranjare**
BigDecimal	Numeric (cu semn)
BigInteger	Numeric (cu semn)
Byte	Numeric (cu semn)
Character	Numeric (fără semn)
CollationKey	Alfabetic, pe setări locale
Date	Cronologic
Double	Numeric (cu semn)
File	Alfabetic legat de cale
Float	Numeric (cu semn)
Integer	Numeric (cu semn)
Long	Numeric (cu semn)
ObjectStreamField	Alfabetic de tip *String*
Short	Numeric (cu semn)
String	Alfabetic

Există câteva condiții pe care trebuie să le respectăm atunci când scriem o metodă *compareTo*:

```
public int compareTo(Object Obj)
```

1. **Elementele trebuie să fie comparabile mutual**. Dacă două elemente sunt comparabile mutual, înseamnă că au variabile ce pot fi distinctive. Pe de altă parte există obiecte ce nu pot fi comparabile mutual ca de exemplu un *File*, un *TreeSet* etc. În cazul acesta vom primi excepția *ClassCastException* atunci când implementăm interfața *Comparable*.
2. **Valoarea de returnat exprimă poziția relativă într-o ordonare naturală**. Metoda *compareTo* poate returna trei valori. Va returna un număr negativ, dacă obiectul curent vine înaintea obiectului cu care se face comparația. Va returna un număr negativ, dacă obiectul curent ajunge în colecție după elementul cu care se face comparația. Va returna zero dacă obiectele sunt egale.
3. **Ordonarea naturală se va face pe baza metodei *equals()***. Faptul că două elemente sunt egale sau nu, ar trebui definit prin suprascrierea metodei *equals*, aşa cum am văzut şi în cadrul colecțiilor. Această condiție este mai mult o recomandare.
4. **Niciodată nu se apelează metoda direct**. Acest mecanism de sortare, este implementat în cadrul FrameWork-ului şi nu avem acces direct. Singurul lucru pe care trebuie să îl facem este sa implementăm metoda, şi ea va fi automat apelată la operațiile aplicate pe colecții.

Pentru a demonstra funcționarea acestui mecanism avem exemplul de mai jos:

```
import java.util.*;
public class Angajat implements Comparable
{
    String departament;
    String nume;
    public Angajat(String departament, String nume)
    {
        this.departament = departament;
        this.nume = nume;
    }
    public String getdepartament()
    {
        return departament;
    }
    public String getnume()
    {
        return nume;
    }
    public String toString()
    {
    return "[dep=" + departament +  ",nume=" + nume + "]";
    }
    public int compareTo(Object obj)
    {
        Angajat decomparat = (Angajat)obj;
        //daca sunt din departamente diferite
        //vom compara departamentele
            int deptComp =
            departament.compareTo(decomparat.getdepartament());
        //daca sunt din acelasi departament
```

```
            //vom compara numele angajatilor
            return ((deptComp == 0) ? nume.compareTo(decomparat.getnume())
                      : deptComp);
        }
        public boolean equals(Object obj)
        {
            if (!(obj instanceof Angajat))
            {
                return false;
            }
            Angajat a = (Angajat)obj;
            return departament.equals(a.getdepartament()) &&
          nume.equals(a.getnume());
        }
        public int hashCode()
        {
            return 43*departament.hashCode() + nume.hashCode();
        }
    }
```

În acest exemplu avem o clasă cu două variabile membru şi anume *departament* si *nume*. Aceste două elemente sunt comparabile, deci le vom putea utiliza în sortare.

Metoda *compareTo()* va returna implementa logica dorită şi anume de a sorta mai întâi după departament şi apoi după nume. Pentru aceasta se va calcula un număr `deptComp` pe baza comparării departamentului obiectului transmis ca parametru şi al obiectului actual.

Vom folosi aceasta valoare `deptComp` în rezultatul final astfel:

```
return ((deptComp == 0) ? nume.compareTo(decomparat.getnume()):deptComp);
```

adică, în cazul in care `deptComp` este diferit de zero, deci avem două departamente diferite, returnează rezultatul comparării acestora. Altfel returnează rezultatul comparării obiectelor *nume* de tip *String*. Se observă că logica poate fi extinsă în cazul în care avem mai multe proprietăți de comparat şi se poate stabili o prioritate în această operație.

Pentru a observa cum sunt folosite aceste mecanisme de comparație avem exemplul de mai jos:

```
class Companie
{
    public static void main (String args[])
    {
        Angajat angajati[] = {
        new Angajat("HR", "Irina"),
        new Angajat("HR", "Cristina"),
        new Angajat("Ingineri", "Daniel"),
        new Angajat("Ingineri", "Vlad"),
        new Angajat("Ingineri", "Octavian"),
        new Angajat("Vanzari", "Emil"),
```

```
            new Angajat("Vanzari", "Eugen"),
            new Angajat("RC", "Bogdan "),
            new Angajat("RC", "Avram"),
            };
            Set set = new TreeSet(Arrays.asList(angajati));
            //parcurgem multimea sortata deja
            //insa ea contine obiecte
            for(Object o : set)
            {
                //asa ca va trebui sa apelam cast
                Angajat a = (Angajat)o;
                System.out.println(a);
            }
        }
}
```

Automat, la constituirea noii colecții formată din obiecte de tip *Angajat*, se apelează metoda de sortare şi anume *compareTo*. După cum se observă această metodă nu este apelată explicit. Rezultatul este:

```
[dep=HR,nume=Cristina]
[dep=HR,nume=Irina]
[dep=Ingineri,nume=Daniel]
[dep=Ingineri,nume=Octavian]
[dep=Ingineri,nume=Vlad]
[dep=RC,nume=Avram]
[dep=RC,nume=Bogdan ]
[dep=Vanzari,nume=Emil]
[dep=Vanzari,nume=Eugen]
```

Această metodă de sortare este una complexă şi acoperă multe cazuri, însă obligă ca obiectele ce vor fi sortate să implementeze interfața *Comparable*. De asemenea pot fi comparate, doar obiecte de acelaşi tip, ceea ce pentru majoritatea cazurilor este suficient.

Comparator

Atunci când nu dorim să impunem ordonarea naturală a claselor, sau clasele nu implementează interfața *Comparable*, avem la dispoziție metoda *compare* din interfața *Comparator*:

```
public int compare(Object obj1, Object obj2)
```

De asemenea trebuie suprascrisă metoda *equals()*, în cazul în care vorbim despre obiecte complexe (cu mai multe variabile membru).

Anterior am folosit deja un Comparator pentru a demonstra lucrul cu TreeSet:

```
Set set = new TreeSet(Collections.reverseOrder());
```

Pentru a exemplifica lucrul cu această interfață avem următorul scenariu: un manager nou, inspectează compania descrisă anterior, şi doreşte să afle mai întâi numele angajatului şi apoi departamentul. Cum această clasă poate implementa doar o singură interfață *Comparable* cu o singură metodă *compareTo* va trebui să implementăm o clasă separat de clasa de bază şi anume un *Comparator*.

Mai jos avem un exemplu pentru o astfel de clasă:

```
class AngComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Angajat a1 = (Angajat)obj1;
        Angajat a2 = (Angajat)obj2;
        int nameComp = a1.getnume().compareTo(a2.getnume());
        return ((nameComp == 0) ?
        a1.getdepartament().compareTo(a2.getdepartament()) :
        nameComp);
    }
}
```

Aceasta este clasa care poate constitui un comparator pentru o nouă listă după cum vom vedea mai jos. De această dată, metoda *compare* are două obiecte *obj1* şi *obj2* pe care le va utiliza pentru comparație. Logica este inversă față de exemplul anterior, şi anume se verifică mai întâi numele angajatului şi apoi se verifică numele departamentului din care acesta face parte. Utilizarea acestui *Comparator* este în funcția *main* rescrisă:

```
      public static void main (String args[])
    {
        Angajat angajati[] = {
        new Angajat("HR", "Irina"),
        new Angajat("HR", "Cristina"),
        new Angajat("Ingineri", "Daniel"),
        new Angajat("Ingineri", "Vlad"),
        new Angajat("Ingineri", "Octavian"),
        new Angajat("Vanzari", "Emil"),
        new Angajat("Vanzari", "Eugen"),
        new Angajat("RC", "Bogdan "),
        new Angajat("RC", "Avram"),
        };
      //noua coletie are un comparator de tip
      //AngComparator
        Set set = new TreeSet(new AngComparator());
        set.addAll(Arrays.asList(angajati));
        //parcurgem multimea sortata deja
        //insa ea contine obiecte
        for(Object o : set)
        {
```

```
            //asa ca va trebui sa apelam cast
            Angajat a = (Angajat)o;
            System.out.println(a);
        }
    }
```

In continuare vom discuta despre colecții ce conțin, nu doar un element, ci pe fiecare poziție sunt câte două elemente, adică o pereche formată din cheie şi valoare.

Dictionary, HashTable şi Properties

Aceste clase ajută în lucrul cu perechi de chei şi valori, în foarte multe aplicații acesta fiind o necesitate. Mai jos avem diagrama ce exprimă ierarhia acestor clase:

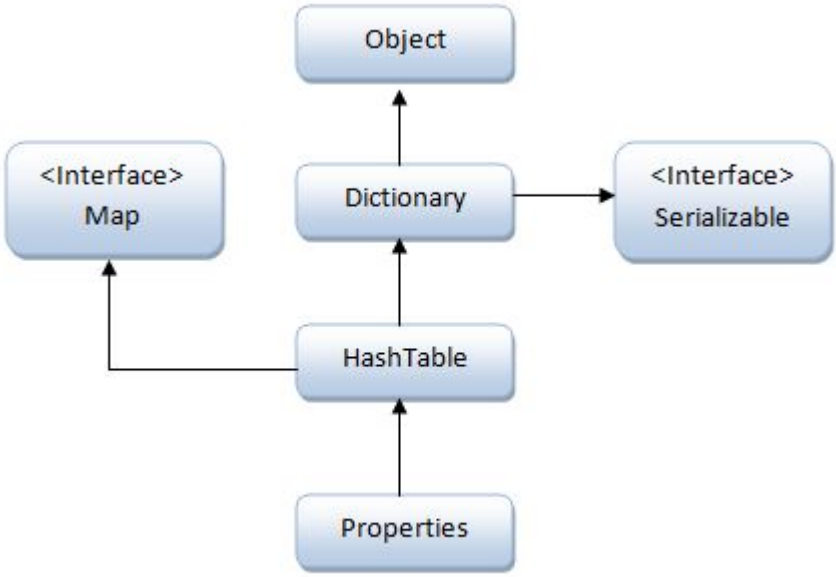

Figura 7.3 Ierarhia claselor Dictionary, HashTable, Properties

Un obiect *Dictionary* funcționează ca o carte de telefoane. Caut un anumit număr după numele persoanei. Numele este cheia din dicționar şi telefonul este valoarea. Presupunem în cadrul acestei comparații, că numele persoanei este unic, deoarece, după cum vom vedea cheia trebuie să fie unică.

Clasa Dictionary

Dictionary este o clasă abstractă ce conține doar metode abstracte. Este alcătuit din perechi:
<cheie> - valoare

Mai jos sunt descrise metodele din această clasă:

Numele metodei	**Descriere**
elements()	Returnează un obiect din dicționar ce permite vizitarea tuturor cheilor
get() 1.0	Returnează o valoare din dicționar
isEmpty()	Verifică dacă un dicționar este gol
keys()	Returnează colecția de chei din dicționar
put()	Introduce un element format din cheie şi valoare
remove()	Şterge un element din dicționar
size()	Returnează numărul de element din dicționar

Deoarece Dictionary este o clasă abstractă, ea nu va fi folosită direct, ci vor exista implementări ale acesteia, ca de exemplu *HashTable*.

Clasa HashTable

Un *HashTable* este un *Dictionary* ce se bazează pe un algoritm de *hashing* ce converteşte cheile in coduri *hash* pentru a căuta mai rapid în colecție. O funcție *hash* este o funcție matematică ce converteşte o valoare numerică în date de mărime relativ mică, de obicei un întreg, ce poate folosi ca index într-un şir. Valorile pe care funcția *hash* le returnează se numesc coduri *hash* (a se vedea funcția *hashCode()*). Mai jos avem funcțiile pe care un *HashTable* le conține:

Nume metodă	**Descriere**
Hashtable()	Constructorul colecției
clear()	Şterge elementele din colecție
clone()	Creează un *HashTable* cu aceleaşi elemente
contains()	Verifică existența unui obiect în colecție
containsKey()	Verifică existența unei chei în *HashTable*
containsValue()	Verifică existența unei valori în *HashTable*
elements()	Returnează un element ce permite vizitarea colecției
entrySet()	Returnează un set de perechi cheie – valoare
equals()	Verifică egalitatea între două obiecte
get()	Returnează valoarea de la o anumită cheie.
hashCode()	Calculează codul *hash* al unei colecții
isEmpty()	Verifică dacă colecția este goală sau nu.
keys()	Returnează o colecție de chei din *HashTable*.
keySet()	Returnează o colecție de chei din *HashTable*.
put()	Pune o pereche cheie-valoare în *HashTable*
putAll()	Pune o colecție de cheie-valoare în *HashTable*
rehash()	Măreşte capacitatea colecției
remove()	Şterge un element din colecție
size()	Returnează numărul de elemente din colecție
toString()	Returnează întregul *HashTable* ca un *String*
values()	Returnează o colecție de valori din *HashTable*

Avantajul principal al acestei colecții este că inserarea într-o astfel de structură este făcută în timp *O(1)* . Nu contează cât de mare este structura, va lua cam acelaşi timp, pentru inserarea unui element. Cum se poate acest lucru?

Atunci când folosim perechi chei-valoare, cheile sunt convertite în cod *hash*, folosind un algoritm de *hash*. Apoi acest cod este redus la o structură internă, pentru a fi folosit ca un index. Pentru două elemente egale, codul va fi acelaşi iar pentru două elemente diferite codul este diferit.

Dacă *hashcode*-ul este acelaşi pentru elemente diferite, discutăm de coliziune. Dacă sunt multe coliziuni, timpul de inserție şi căutare, creşte şi se pierde astfel avantajul principal. Dacă există elemente multiple cu aceeaşi cheie, va trebui traversată o listă înlănțuită pentru a ajunge la valori. Mai jos avem un *HashTable* cu mai multe coliziuni:

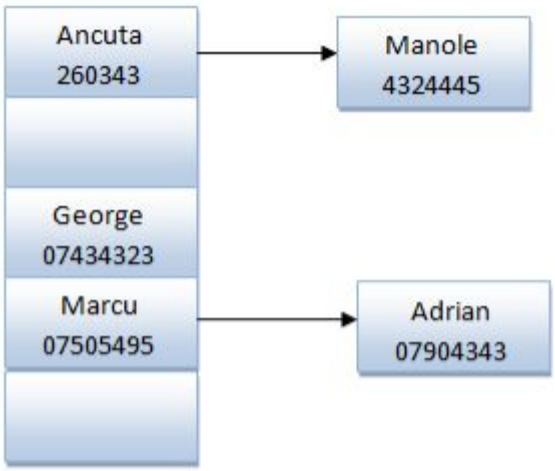

Figura 7.4 un HashTable cu mai multe coliziuni

Mai jos avem şi funcția care „permite” crearea acestor coliziuni

```
public class HashFunction {
    public static int hashFunction(String str)
    {
        int sum=0;
        for(int i=0;i<str.length();i++)
            sum=sum+Character.getNumericValue(str.charAt(i));
        return sum;
    }
     public static void main (String args[])
     {
         System.out.println(hashFunction("Marcu"));
         System.out.println(hashFunction("Adrian"));
         System.out.println(hashFunction("Ancuta"));
         System.out.println(hashFunction("Manole"));
     }
}
```

În cadrul acestei funcții se calculează suma codurilor ASCII ale fiecărui caracter din nume. La rularea acestui program avem următoarele coduri:

```
101
101
114
114
```

Acesta este motivul apariției coliziunilor în *HashTable*.

Creearea HashTable

Există patru constructori pentru un *HashTable*, dintre care primii trei sunt:

```
public Hashtable()
public Hashtable(int initialCapacity)
public Hashtable(int initialCapacity, float loadFactor)
```

Ultimii doi constructori permit specificarea unei capacități inițiale. Rata de mărire a capacității implicite, care poate fi specificată în cel de-al doilea constructor este de dublul capacității actuale +1. Se poate modifica prin ajustarea acelui procent.

Al patrulea constructor inițializează *HashTable* prin copierea unei colecții în tabelul actual.

```
public Hashtable(Map t)
```

Adăugarea perechilor de chei-valoare

Spre deosebire de clasele studiate anterior, trebuie să oferim o cheie dar şi o valoare:

```
public Object put(Object key, Object value)
```

într-un *HashTable*, cheile sau valorile nu pot fi *null*. Încercarea de introducere o cheie *null* în *HashTable* va produce eroarea *NullPointerException*. Introducerea unei perechi în care cheia se afla deja în colecție, va avea ca efect înlocuirea valorii de la acea cheie cu valoarea nouă. Obiectul returnat va fi vechea valoare, sau dacă obiectul nu era în colecție, obiectul returnat este *null*.

Afişarea unui *HashTable* se face natural pentru ca implementează metoda *toString*().

Ştergerea unei perechi

Se face utilizând funcția *remove()*:

```
public Object remove(Object key)
```

Dacă cheia `key` este prezentă în colecție, atunci va fi eliminata perechea cheie-valoare specificată prin `key` şi valoarea este returnată. O altă metodă de a şterge elementele dintr-o colecție de acest tip, este *clear()*:

```
public void clear()
```

Această metodă va șterge toate perechile din *HashTable*.

Mărimea unui HashTable

Putem controla mărimea unui *HashTable* doar după crearea acestuia. Mai jos sunt câteva funcții pentru controlul mărimii unui *HashTable*:

```
public int size()
public boolean isEmpty()
```

Aceste funcții returnează mărimea colecției şi verifică dacă *HashTable* este gol sau nu.

Există o funcție de redimensionare a colecției, dar este *protected*:

```
protected void rehash()
```

Aceasta permite crearea unui nou şir intern mai mare, inserând toate valorile din şirul deja existent.

Operaţii cu HashTable

Returnarea obiectelor din HashTable

Există câteva moduri de a returna date din *HashTable*. Cel mai simplu este apelând *get*:

```
public Object get(Object key)
```

dacă este găsită cheia dată ca parametru, atunci funcția va returna valoarea corespunzătoare cheii. Dacă nu este găsită, atunci funcția va returna *null*.

Dacă vrem să aflăm cheile din colecție putem folosi:

```
public Enumeration keys()
public Set keySet()
```

diferența constă în modul în care sunt returnate cheile. Prima metodă returnează ca un `Enumeration`. A doua metodă, returnează colecția sub forma unui *Set*. Pe lângă chei, se pot returna şi valorile din *HashTable*:

```
public Enumeration elements()
public Collection values()
```

Metoda *elements()* returnează o mulțime de valori sub forma unei *Enumeration*. Metoda *values()* returnează aceleaşi date, dar sub forma unei *Collection*.

Cea mai complexă metodă de returnare a elementelor dintr-o colecție *HashTable* este *entrySet()*

```
public Set entrySet()
```

Aceasta returnează o colecție de tip *Set* ce conține perechile de chei-valori. Acest tip de data returnat este *Map*, şi vom discuta despre el în cele ce urmează. Iată două moduri de a accesa cheile unei astfel de colecții:

```
Enumeration enum = hash.keys();
while (enum.hasMoreElements())
{
      String key = (String)enum.nextElement();
      System.out.println(key + " : " + hash.get(key));
}
```

Un alt mod, implică folosirea întregului obiect din colecție:

```
Set set = hash.entrySet();
Iterator it = set.iterator();
while (it.hasNext())
{
      Map.Entry entry = (Map.Entry)it.next();
      System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

Căutarea elementelor

Clasa *Hashtable* conține trei metode, care ne permit căutarea unei chei sau a unei valori în colecție. Cea mai simplă rămâne metoda *get()* discutată mai sus.

```
public boolean containsKey(Object key)
```

Această funcție verifică existența unei chei în colecție. Alte două metode vor verifica existența unei valori în colecție:

```
public boolean contains(Object value)
public boolean containsValue(Object value)
```

Ambele realizează acelaşi lucru, însă pentru că *HashTable* implementează şi interfața *Map*, avem această duplicitate. Este de recomandat să folosim aceste două funcții cât mai puțin posibil, deoarece se parcurge întreaga colecție, algoritmul având cel un ordin *O(n)*.

Verificarea egalității între HashTable

Aceasta se poate realiza folosind funcția *equals()*:

```
public boolean equals(Object o)
```

Egalitatea este definită de interfața *Map,* iar nu la nivelul *HashTable*. În consecință, *HashTable* poate fi egal cu orice alt *Map* şi nu doar un alt *HashTable*. Regula de bază este că, două *HashMap*-uri sunt egale atunci când au aceleaşi perechi cheie-valoare. Ordinea nu contează.

HashTable este imutabil. Dacă un *HashTable* este inițializat cu un set de elemente, atunci pentru redimensionare va trebui să inițializăm un nou obiect. Pentru a obține un *Map*, ce nu este imutabil avem următoarea funcție *unmodifiableMap*:

```
Hashtable h = new Hashtable();
// se va umple hashtable
Map m = Collections.unmodifiableMap(h);
```

Mai jos avem un exemplu de folosire a *HashTable* pentru a număra cuvintele dintr-un fișier.

```
import java.io.*;
import java.util.*;
public class Cuvinte
{
    static final Integer ONE = new Integer(1);
    public static void main (String args[]) throws IOException
    {
        Hashtable map = new Hashtable();
        FileReader fr = new FileReader(args[0]);
        BufferedReader br = new BufferedReader(fr);
        String line;
        while ((line = br.readLine()) != null)
        {
            processLine(line, map);
        }
        Enumeration en = map.keys();
        while (en.hasMoreElements())
        {
            String key = (String)en.nextElement();
            System.out.println(key + " : " + map.get(key));
        }
     }
    static void processLine(String line, Map map)
    {
        StringTokenizer st = new StringTokenizer(line);
        while (st.hasMoreTokens())
        {
            //map este cheia
```

```
                //iar  st.nextToken() este valoare
                addWord(map, st.nextToken());
            }
        }
        static void addWord(Map map, String word)
        {
            Object obj = map.get(word);
            if (obj == null)
            {
                map.put(word, ONE);
            }
            else
            {
                int i = ((Integer)obj).intValue() + 1;
                map.put(word, new Integer(i));
            }
        }
}
```

Ideea acestui program este ca in *map*, variabilă de tip *HashTable* să ținem o pereche de chei valori, unde cheia reprezintă cuvântul citit din fişier, iar valoarea reprezintă numărul de apariții al acelui cuvânt.

Metoda *processLine()* împarte linia citită în cuvinte separate prin spațiu, şi apelează metoda *addWord*. Aici se verifică dacă mai avem înregistrat cuvântul în colecție, dacă nu adăugam perechea (cuvânt nou, 1) semn că apare o dată. Altfel, dacă a mai fost înregistrat, incrementăm valoarea corespunzătoare cheii, cu unu.

Pentru un fișier simplu ca mai jos:

```
eee fff eeds
fff aaa eeds    aaa
```

rezultatul este:

```
fff : 2
eeds : 2
eee : 1
aaa : 2
```

Clasa Properties

Clasa *Properties* reprezintă un *HashTable* specializat. În această clasă atât cheile cât şi valorile sunt *String*. Atunci când lucrăm şi cu *HashTable* şi cu *Properties*, va trebui să ținem cont de ierarhie: un *Properties* poate fi un *HashTable*, însă un *HashTable* nu poate funcționa ca un *Properties*.

Mai jos avem metodele specifice acestei clase:

Nume	**Descriere**
Properties	constructorul clasei
getProperty()	returnează o valoare pentru o cheie din listă
list()	listează proprietățile li valorile aferente
load ()	încarcă proprietățile dintr-un stream
propertyNames()	returnează o colecție de chei din listă
setProperty()	setează o pereche cheie-valoare în listă
store()	salvează lista de proprietăți într-un stream

Setarea şi preluarea elmentelor

Pentru aceasta există trei metode şi anume:

```
public Object setProperty(String key, String value)
public String getProperty(String key)
public String getProperty(String key, String defaultValue)
```

prima metodă va modifica cheia `key` cu valoarea `value`. Metoda `getProperty` returnează valoarea de la cheia `key`. A doua metodă de `getProperty` va returna valoarea de la cheia `key`, iar dacă aceasta este null, va returna valoarea `defaultValue`. Pe lângă aceste metode, există o funcție ce returnează o listă cu cheile din *Properties*:

```
public Enumeration propertyNames()
```

Încărcarea şi salvarea datelor

Aceste două operații sunt foarte importante deoarece simplifică exportarea şi importarea structurilor de date de acest tip:

```
void load(InputStream inStream) throws IOException
void store(OutputStream out, String header) throws IOException
```

Iată de exemplu, cum se poate salva în fişier toate datele unei astfel de colecții:

```
import java.io.*;
import java.util.*;
public class Proprietati
{
    public static void main(String args[]) throws IOException
    {
        Properties prop = new Properties();
        FileOutputStream fo = new FileOutputStream(args[0]);
        prop.setProperty("cheia1", "valoarea1");
```

```
            prop.setProperty("cheia2", "valoarea2");
            prop.setProperty("cheia3", "valoarea3");
            prop.setProperty("cheia1", "valoarea");
            prop.save(fo, "Aici vin comentarii");
        }
    }
```

Ceea ce se salvează în fişier este:

```
#Aici vin comentarii
#Fri Nov 20 19:53:44 EET 2009
cheia3=valoarea3
cheia2=valoarea2
cheia1=valoarea
```

Metoda *load()* va încerca să găsească în fişierul sursă, o asemenea structură, de aceea este bine să respectăm formatul, deşi metoda va putea să încarce diferite formate, însă nu de fiecare dată cu rezultatul aşteptat.

Map

Interfața Map aparține framework-ului Collections şi vine să înlocuiască clasa Dictionary. În timp ce Dictionary este o clasă abstractă, era imperativ ca metodele ei să se afle într-o interfață. Aceasta este Map, şi suportă lucrul cu perechi cheie-valoare.

Deşi face parte din FrameWork, interfața Map nu extinde Collection, ci este rădăcina unei noi ierarhii. Sunt patru clase ce implementează această interfață: HashMap, WeakHashMap, TreeMap şi HashTable. Toate elementele din aceste colecții sunt de tip *Map.Entry.* Figura de mai jos reprezintă ierarhia acestor clase şi interfețe.

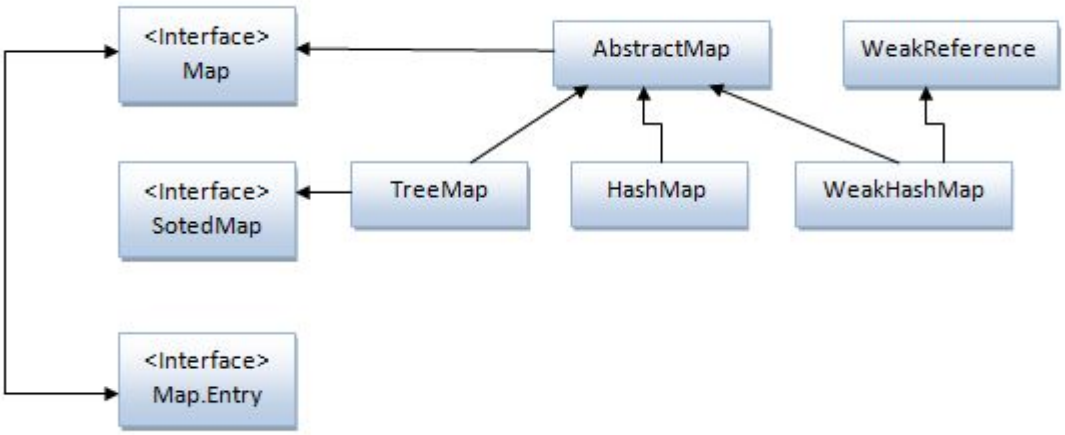

Figura 7.5 Ierarhia claselor Map

Interfața Map.Entry

Elementele unei *Map* sunt de tip *Map.Entry*, o interfață conținută în *Map*. Fiecare pereche de cheie-valoare este o instanță a acestei interfață. Nu vom crea propriu-zis instanțe, dar clasele concrete vor returna obiecte ce deja au implementat această interfață.

Mai jos avem funcțiile acestei interfețe:

Metoda	**Descriere**
equals()	verifică egalitatea cu un alt obiect
getKey()	returnează cheia din Map.Entry
getValue()	returnează valoarea din Map.Entry
hashCode()	calculează codul hash pentru obiectul actual
setValue()	schimbă valoarea din Map.Entry

Iată cum se pot apela aceste metode pentru a afişa entitățile dintr-o colecție de tip Properties:

```
Properties props = System.getProperties();
        Iterator iter = props.entrySet().iterator();
        while (iter.hasNext())
        {
            Map.Entry entry = (Map.Entry)iter.next();
            System.out.println(entry.getKey() + " -- " +
entry.getValue());
        }
```

Acelaşi cod poate fi scris cu vechile metode din *Properties* şi *Enumeration* astfel:

```
Properties props = System.getProperties();
        Enumeration en = props.propertyNames();
        while (en.hasMoreElements())
        {
            String key = (String)en.nextElement();
            System.out.println(key + " -- " + props.getProperty(key));
        }
```

Clasa HashMap

Clasa *HashMap* este folosită la implementarea interfeței Map şi anume o colecție de perechi-valori în care elementele nu sunt ordonate. Mai jos sunt metodele implementate:

Nume metodă	**Descriere**
containsKey()	Verifică existența unei chei in *hash map*
containsValue()	Verifică existența unei valori in *hash map*
entrySet()	Returnează o colecție de perechi sub forma unui *map*

get()	Returnează valoarea pentru o anumită cheie
isEmpty()	Verifică dacă o colecție este goală sau nu
keySet()	Returnează toate cheile din *hash map*
put()	Plasează o pereche în *hash map*
putAll()	Introduce in *hash map* o colecție de perechi cheie-valoare
remove()	Scoate din colecție o pereche
size()	Returnează numărul de elemente dintr-un *hash map*
values()	Returnează o colecție de valori din *hash map*

Creearea unui HashMap

Există patru constructori pentru crearea unui *HashMap*:

```
public HashMap()
public HashMap(int initialCapacity)
public HashMap(int initialCapacity, float loadFactor)
public HashMap(Map map)
```

În timp ce primii trei vor iniția liza o colecție goală, al patrulea va crea *hash map*-ul pe baza elementelor din parametru.

Adăugarea în HashMap

Pentru a adăuga în *hash map* avem două metode:

```
public Object put(Object key, Object value)
public void putAll(Map map)
```

Spre deosebire de *HashTable*, atât cheia cât şi valoarea unui element nou introdus pot fi *null*. Pentru a copia o colecție de perechi de la un *Map* la altul se foloseşte a doua metodă. Afişarea elementelor unui *HashMap* se face natural, deoarece suprascrie metoda *toString()*.

Ştergerea unui element

Pentru a şterge dintr-un *HashMap*, avem metoda:

```
public Object remove(Object key)
```

Dacă cheia se află în *HashMap*, perechea va fi ştearsă, şi valoarea obiectului va fi returnată. Dacă obiectul nu se află în colecție atunci valoarea *null*, va fi returnată. Atenție, şi *null* poate fi valoare legitimă în colecție. De aceea este de dorit să evităm plasarea *null* ca valoare în colecție.

Altă metodă de a şterge toate elementele din colecție este:

```
public void clear()
```

Operaţii cu HashMap

Se pot returna obiectele din colecție, iar pentru aceasta avem o serie de metode:

```
public Object get(Object key)
```

Dacă cheia nu este găsită în colecție, se returnează *null*. Altfel se va returna valoarea de la cheia specificată ca parametru. Aceeaşi remarcă ar fi de făcut, că elementele într-un HashMap pot fi *null*. O alta metodă este :

```
public Set keySet()
```

şi returnează colecția de chei sub forma unei mulțimi. Pentru a prelua aceste chei ca o colecție generală avem metoda:

```
public Collection values()
```

pentru a returna elementele complete, adică perechea de cheie-valoare, avem la dispoziție metoda:

```
public Set entrySet()
```

Aceasta returnează o colecție de obiecte de tip *Map.Entry*.

Căutarea elementelor se face prin două metode:

```
public boolean containsKey(Object key)
public boolean containsValue(Object value)
```

Prima metodă este asemănătoare metodei *get()*, dar în loc de a returna o valoare a obiectului, vom avea o valoare booleană, *true* dacă cheia se află în colecție sau *false* în caz contrar.

A doua metodă, verifică existența unei valori specifice în *HashMap*. Valorile sunt comparate, şi această comparare este făcută în timp liniar, de aceea este mai bine să folosim prima metodă.

Clasa WeakHashMap

Această clasă funcționează identic ca şi un *HashMap*, cu o diferență importantă: dacă managerul de memorie Java nu mai are o referință puternică la un obiect, atunci acea pereche va fi ştearsă. Pentru a înțelege această frază trebuie să înțelegem ce este o referință slabă. În general, obiectele care nu conțin date direct, ci conțin referințe la un alt obiect sunt denumite referințe. Aceste clase se găsesc În pachetul java.lang.ref, de exemplu *public abstract class Reference<T>*.

Există patru tipuri de referință:

1. Referințe puternice, nu au o clasă specială
2. Referințe soft sunt ca un *cache*. Când memoria este puțină GC va elibera arbitrar aceste referințe.

3. Referințe slabe: acestea sunt mai slabe decât cele soft. Dacă singura referință a unui obiect sunt doar de tip slab, GC va putea şterge obiectul oricând.
4. Referințe fantomă. Acestea permite o notificare înainte ca GC să apeleze finalizarea pe obiectul în cauză.

Mai jos avem o exemplificare a acestui mecanism şi cum se lucrează cu referințe slabe:

```
import java.util.*;
public class Weak
{
    private static Map map;
    public static void main (String args[])
    {
        map = new WeakHashMap();
        map.put(new String("Cheie"), "Valoare");
        //o metoda ce se executa pe un alt fir
        //decat cel curent
        Runnable runner = new Runnable()
        {
            public void run()
            {
                while (map.containsKey("Cheie"))
                {
                    try
                    {
                        //intra in standby 0.5 secunde
                        Thread.sleep(500);
                    }
                    catch (InterruptedException ignored)
                    {
                    }
                    System.out.println("Thread waiting");
                    //sistemul GC va rula fortat
                    System.gc();
                    }
             }
        };
        Thread t = new Thread(runner);
        System.out.println(map);
        //incep procedura run
        t.start();
        //firul principal sta si asteapta ca
        System.out.println("Main waiting");
        try
        {
            //cel nou sa isi incheie executia
            t.join();
            //dupa ce thread-ul t nu mai este
            System.out.println(map);
```

```
        }
        catch (InterruptedException ignored)
        {
        }
    }
}
```

Exemplul poate fi greu de înțeles acum pentru că nu am prezentat încă firele de execuție (*Thread*). Totuşi concluzia este că după apelarea Garbage Collector-ului, se pierd legăturile slabe, deci se pierd elementele din `map`. Acesta este rezultatul rulării programului:

```
{Cheie=Valoare}
Main waiting
Thread waiting
{}
```

Clasa TreeMap

Ultima implementare a interfeței *Map* este un *TreeMap*. Un *TreeMap* este un *Map* care menține cheile într-o ordine prin intermediul unui arbore balansat, de tip roşu-negru. Pe lângă metodele expuse de Map şi discutate mai sus avem următoarele:

Metoda	**Descriere**
firstKey()	Returnează prima cheie din *Map*
headMap()	Returnează sub map-ul de la începutul map-ului original
lastKey()	Returnează ultima cheie din colecție
subMap()	Returnează un subarbore oarecare din colecție
tailMap()	Returnează subarborele de la sfârşitul celui original

Acestea provin tocmai din faptul că *TreeMap* implementează interfața *SortedMap*, lucru ce impune existența acestor metode.

Crearea unui TreeMap

Există patru constructori pentru un *TreeMap*. Primul este fără argument creează un *Map* gol, al doilea este un constructor de copiere:

```
public TreeMap()
public TreeMap(Map map)
```

Pe lângă aceştia, mai există doi constructori: unul acceptă un *Comparator* pentru a defini un mod de ordonare personalizat şi al doilea acceptă un *SortedMap*, fiind un constructor de copiere optimizat.

```
public TreeMap(Comparator comp)
public TreeMap(SortedMap map)
```

Operaţiuni cu Map

Pentru a vizualiza elementele unui arbore avem la dispoziție metodele de mai jos:

```
SortedSet headMap(Object toKey)
SortedSet tailMap(Object fromKey)
SortedSet subMap(Object fromKey, Object toKey)
```

Pentru a specifica subarborele, pentru `headMap` avem parametrul `toKey` ce indică subarborele de la început până la acea cheie, iar pentru `tailMap` avem parametrul `fromKey` ce indică subarborele de la `fromKey` până la sfârşit. Pentru a include şi elementele de la capăt avem următorul truc:

```
Map headMap = map.headMap(toKey+"\0");
```

În cazul celei de-a treia metodă şi anume `subMap`, se va returna un arbore cuprins între:

```
fromKey <= map keys < toKey
```

Metodele *firstKey()* şi *lastKey()* permit accesarea rapidă a primului, respectiv ultimului element din map:

```
Object firstKey()
Object lastKey()
```

Mai jos avem un exemplu pentru ilustrarea acestor operațiuni:

```
import java.util.*;

class MyComparator implements Comparator<String>
{

    public int compare(String o1, String o2)
    {
        //sa comparam perechile
       return o2.compareTo(o1);
    }

}
public class Studenti
{
      public static void main (String[ ] args)
      {
               Map<String, Double> students = new TreeMap<String, Double>(new
MyComparator());

            students.put ("Sebastian",6.0);
            students.put ("Bogdan", 7.8);
            students.put ("Andrei", 4.67);
```

```
            students.put ("Remus", 8.96);
            students.put ("Catalin", 8.55);
            System.out.println (students);

            Map<String, Double> group =
                    ((TreeMap<String, Double>)students).subMap("Bogdan",
"Remus");
            System.out.println (group);

            System.out.println (students.remove ("Andrei"));
            System.out.println (students.remove ("Boggdan"));
            System.out.println (students.containsKey ("Bogdan"));
            System.out.println (students.containsKey ("Andrei"));
            System.out.println (students.containsValue (7.8));

            System.out.println (students);
      }
}
```

În cazul acestui exemplu, instrucțiunea :

```
Map<String, Double> group =
                    ((TreeMap<String, Double>)students).subMap("Bogdan",
"Remus");
```

Ar produce o eroare de tipul:

```
Exception in thread "main" java.lang.IllegalArgumentException: fromKey >
toKey
```

Aceasta are loc tocmai din modul în care este construit acest arbore, si anume pe baza unui comparator, `MyComparator`, care va inversa ordinea firească a cheilor. De aceea, va trebui să considerăm care este rădăcina arborelui şi să alegem subarborele corect:

```
Map<String, Double> group =
                    ((TreeMap<String, Double>)students).subMap("Remus",
"Bogdan");
```

În cazul acesta va fi format dintr-un singur element, cel marcat cu font italic:

```
{Sebastian=6.0, Remus=8.96, Catalin=8.55, Bogdan=7.8, Andrei=4.67}
{Catalin=8.55}
4.67
null
true
false
true
```

```
{Sebastian=6.0, Remus=8.96, Catalin=8.55, Bogdan=7.8}
```

Pentru a include şi capetele intervalului va trebui să folosim acel truc descris mai sus.

Clasa Collections

Clasa *Collections* este o clasa a FrameWork-ului ce constă din metode statice şi obiecte pentru a înlesni lucrul cu colecții. Clasa *Arrays* este clasa corespondentă, ce lucrează cu şiruri.

Toți membrii acestei clase sunt descrişi mai jos.

Metoda	**Descrierea**
EMPTY_LIST	Reprezintă o listă goală imutabilă
EMPTY_MAP	Reprezintă o colecție Map goală imutabilă
EMPTY_SET	Reprezintă o mulțime goală imutabilă
binary_search()	caută un element în listă folosind tehnica de căutare binară
copy()	copiază elementele dintre două liste
enumeration()	converteşte o listă la un *Enumeration*
fill()	umple lista cu un element anume
max()	caută maximul dintr-o colecție
min()	caută minimul dintr-o colecție
nCopies()	creează o listă imutabilă cu multiple copii ale unui element
reverse()	inversează elementele într-o listă
reverseOrder()	returnează un comparator ce inversează ordinea elementelor
shuffle()	reordonează elementele aleatoriu
singleton()	returnează un set imutabil de un element
singletonList()	returnează o listă imutabilă de un element
singletonMap()	returnează un Map imutabil de un element
sort()	reordonează elementele în listă

Pe lângă acestea, mai sunt o serie de metode ce creează colecții *thread-safe*, însă nu vom discuta despre aceasta acum. În continuare vom prezenta câteva din metodele de mai sus, pe cele mai importante.

Sortarea

Metoda *sort()* permite sortarea elementelor unei liste:

```
public static void sort(List list)
public static void sort(List list, Comparator comp)
```

Iată un exemplu pentru folosirea acestei metode:

```
import java.util.*;
public class Colectii
{
    public static void main(String args[]) throws Exception
    {
        List list = Arrays.asList(args);
        Collections.sort(list);
        for (int i=0, n=list.size(); i<n; i++)
        {
            if (i != 0) System.out.print(", ");
            System.out.print(list.get(i));
        }
        System.out.println();
    }
}
```

Lista obținută din șirul argumentelor este cel mai probabil nesortată. Prin apelarea acestei metode vom ordona elementele listei. O altă metodă ar fi de a folosi ordinea inversă şi anume:

```
Collections.sort(list, Collections.reverseOrder());
```

Căutarea

Căutarea unui element este cea binară, şi este implementată prin cele două metode:

```
public static int binarySearch(List list, Object key)
public static int binarySearch(List list, Object key, Comparator comp)
```

Căutarea binară constă în împărțirea unui sir *sortat* în două şi compararea elementului de căutat cu mijlocul șirului. Dacă nu găsim elementul, căutăm mai departe. Cum? Dacă elementul din mijloc este mai mic decât numărul de căutat, vom căuta în dreapta mijlocului, dacă nu vom căuta în stânga. Adică, vom căuta fie in subşirul delimitat de 0 şi mijloc adică stânga, fie în subşirul delimitat de mijloc si sir.lentgh adică dreapta. Aplicăm recursiv aceasta metodă până când elementul căutat este egal cu mijlocul sau nu mai avem unde căuta pentru ca subşirul a devenit gol. Iată mai jos un exemplu pentru această căutare:

În şirul 1 3 4 6 8 9 11 14 15 17 19 20 23 25 vom căuta elementul 4. Șirul are 14 elemente vom căuta mijlocul si anume 13/2 = 7. Elementul de pe poziția 7 este14, comparăm cu 4 este mai mare deci vom căuta în stânga: 1 3 4 6 8 9 11 14. Aplicăm acelaşi algoritm pe un subşir de 8 elemente şi mijlocul va fi la 7/2 = 3. Comparăm elementul de pe poziția 3 si anume 6 cu 4, este mai mare. Aplicam aceeaşi strategie pe subşirul 1 3 4 6. Avem 4 elemente / 2 = 2 şi găsim mijlocul egal cu elementul de căutat. Se încheie algoritmul.

În funcțiile de mai sus `key` este elementul de căutat iar `list` este *șirul* nostru.

Mai jos avem şi un exemplu de căutare si sortare folosind *Collections*.

```
import java.util.*;
public class Colectii
{
    public static void main(String args[]) throws Exception
    {
        String nume[] = {"George", "Victor", "Laura", "Raluca",
        "Costel", "Maria", "Dorel"};
        // Convertim la colectii
        List list = new ArrayList(Arrays.asList(nume));
        // sortam lista
        Collections.sort(list);
        System.out.println("Sorted list: [length: " + list.size() +
"]");
        System.out.println(list);
        // caut un element
        int index = Collections.binarySearch(list, "Victor");
        System.out.println("Found Victor @ " + index);
        // Search for element not in list
        index = Collections.binarySearch(list, "fals");
        System.out.println("Didn't find fals @ " + index);
        // Insert
        int newIndex = -index -1;
        list.add(newIndex, "false");
        System.out.println(list);
    }
}
```

Iată rezultatul:

Sorted list: [length: 7]
[Costel, Dorel, George, Laura, Maria, Raluca, Victor]
Found Victor @ 6
Didn't find fals @ -8
[Costel, Dorel, George, Laura, Maria, Raluca, Victor, false]