COLECȚII

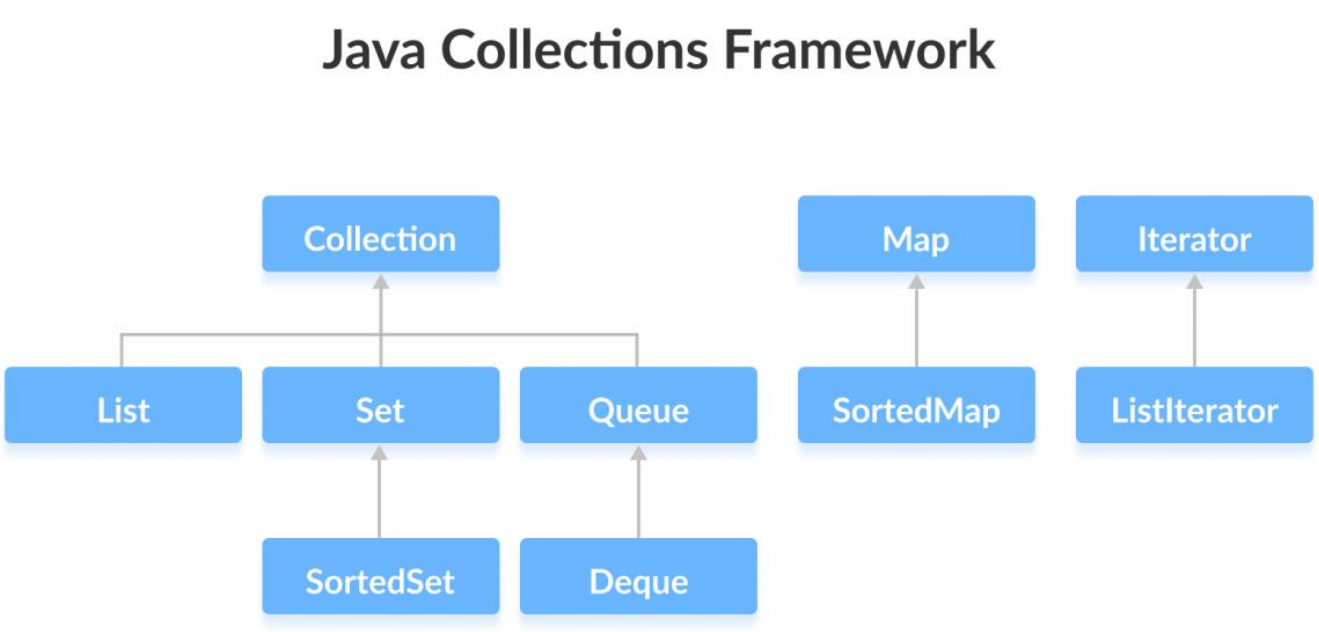
O *colecție* este un obiect container care grupează mai multe elemente într-o structura unitară. Intern, elementele dintr-o colecție se află într-o relație specifică unei structuri de date (lineară, asociativă, arborescentă etc.), astfel încât asupra lor se pot efectua operații de căutare, adăugare, modificare, ștergere, parcurgere etc.

Limbajul Java oferă un framework/API performant pentru crearea și managementul structurilor dinamice de date (tablou, liste, mulțime, tabelă de asocieri etc) – *Java Collections Framework*, astfel încât programatorul să fie degrevat de implementarea optimă a lor.

Framework-ul Collections are o arhitectură bazată pe interfețe și clase care permite reprezentarea și manipularea unitară a colecțiilor, într-un mod independent de detaliile de implementare, astfel:

- *interfețe* – definesc într-un mod abstract operațiile specifice diverselor colecții;
- *clase* – conțin implementări concrete ale colecțiilor definite în interfețe, iar începând cu Java 1.5 ele sunt generice (tipul de dată concret al elementelor din colecție se precizează prin operatorul <> (operatorul diamond): `List<Persoana> lp = new ArrayList<>();`
- *algoritmi polimorfici* – sunt metode statice, grupate în care clasa utilitară `Collections`, care implementează optim operații generice caracteristice colecțiilor de date, cum ar fi: căutare, sortare, copiere, determinarea minimului/maximului etc.

Principalele ierarhii de interfețe puse la dispoziție de framework-ul Java Collections sunt reprezentate în figura de mai jos (sursa: https://www.programiz.com/java-programming/collections):



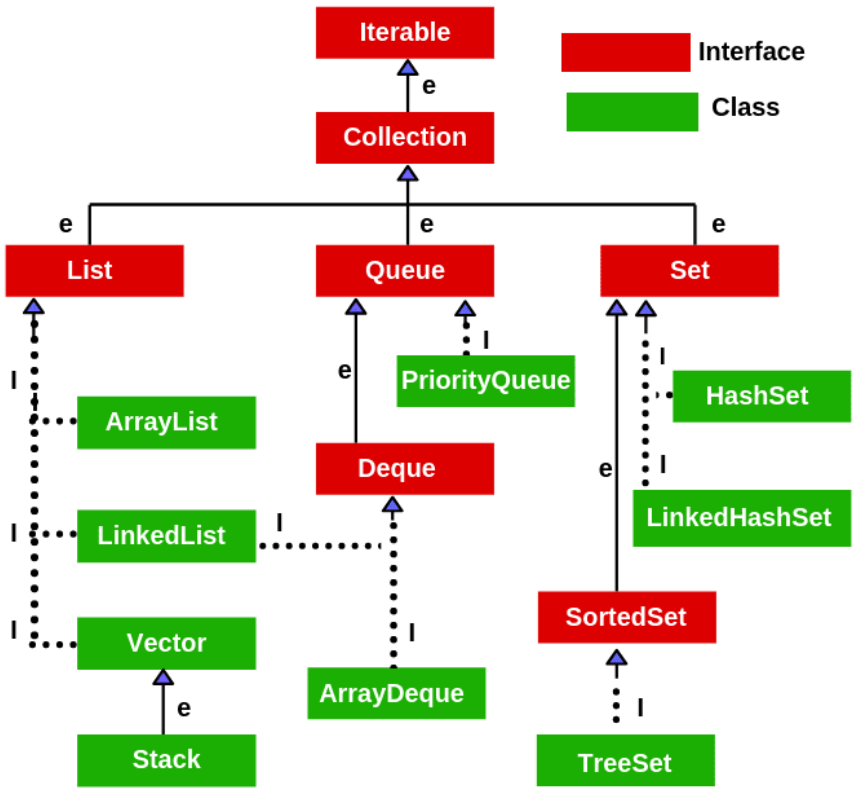
Interfața Collection

Nucleul principal al framework-ului Java Collections este reprezentat de interfața `Collection` care conține o serie de metode fundamentale de prelucrare specifice tuturor colecțiilor. O parte dintre metodele uzuale ale interfeței `Collection` sunt:

- `public int size()` – returnează numărul total de elemente din colecție;
- `public boolean add(E e)` – inserează în colecția curentă elementul `e`;
- `public boolean addAll(Collection<E> c)` – inserează în colecția curentă toate elementele din colecția `c`;
- `public boolean remove(Object e)` – șterge elementul `e` din colecția curentă;
- `public boolean contains(Object e)` – caută în colecția curentă elementul `e`;
- `public Iterator iterator()` – returnează un iterator pentru colecția curentă;
- `public Object[] toArray()` – realizează conversia colecției într-un tablou cu obiecte de tip `Object`.

Alte metode ale interfeței `Collection` sunt prezentate aici: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.

Ierarhia formată din interfețele care extind interfața `Collection`, precum și clasele care le implementează, este reprezentată în figura de mai jos (sursa: https://www.scientecheasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0):



Parcurgerea unei colecții presupune obținerea, pe rând, a unei referințe către fiecare obiect din colecție. O modalitate generală de a parcurge o colecție, independent de tipul ei, este reprezentată de *iteratori*. Astfel, vârful celor ierarhiei `Collection` este interfața `Iterable`.

Interfața List

Interfața `List` extinde interfața `Collection` și modelează o colecție de elemente ordonate care permite inclusiv memorarea elementelor duplicate.

Interfața `List` adaugă metode suplimentare fata de interfața `Collection`, corespunzătoare operațiilor care utilizează index-ului fiecărui element, considerat ca fiind de un tip generic `E`:

- accesarea unui element: `E get(int index),E set(int index);`
- adăugarea/ștergere element: `void add(int index, E element),E remove(int index);`
- determinarea poziției unui element în cadrul colecției: `int indexOf(Object e), int lastlndexOf(Object e).`

Cele mai utilizate implementări ale interfeței `List` sunt clasele `ArrayList` și `LinkedList`.

Clasa `ArrayList`

Clasa `ArrayList` oferă o implementare a unei liste utilizând un tablou unidimensional care poate fi redimensionat dinamic:

```
List<Tip> listaTablou = new ArrayList<>();
ArrayList<Tip> listaTablou = new ArrayList<>();
```

Se poate observa cum o colecție `ArrayList` poate fi referită atât printr-o referință de tipul interfeței implementate (`List`), cât și printr-o referință de tipul colecției.

Capacitatea implicită a unei astfel de liste este egală cu 10, iar pentru a specifica explicit o altă capacitate se poate utiliza un constructor care primește ca argument un număr întreg:

```
List<Tip> listaTablou = new ArrayList<>(50);
```

Exemple:

```
List<Integer> lista1 = new ArrayList<>();
lista1.add(0, 1); // adaugă 1 pe poziția 0
lista1.add(1, 2); // adaugă 2 pe pozția 1
System.out.println(lista1); // [1, 2]

List<Integer> lista2 = new ArrayList<Integer>();
lista2.add(1); // adaugă 1 la sfârșitul listei
lista2.add(2); // adaugă 2 la sfârșitul listei
lista2.add(3); // adaugă 3 la sfârșitul listei
System.out.println(lista2); // [1, 2, 3]
```

```
// adaugă elementele din lista2 începând cu poziția 1
lista1.addAll(1, lista2);
System.out.println(lista1); // [1, 1, 2, 3, 2]

// șterge elementul de pe poziția 1
lista1.remove(1);
System.out.println(lista1); // [1, 2, 3, 2]

// afișează elementul aflat pe poziția 3
System.out.println(lista1.get(3)); // 2

// înlocuiește valoarea aflată pe poziția 1  cu valoarea 5
lista1.set(1, 5);
System.out.println(lista1); // [1, 5, 3, 2]
```

Observații:

- Accesarea unui element se realizează cu complexitatea $\mathcal{O}(1)$.
- Adăugarea unui element la sfârșitul listei prin metoda `add(T elem)` se realizează cu complexitatea $\mathcal{O}(1)$ dacă nu este depășită capacitatea listei sau cu complexitatea $\mathcal{O}(n)$ în caz contrar.
- Adăugarea unui element pe o anumită poziție prin metoda `add(E element, int index)` se realizează cu complexitatea $\mathcal{O}(n)$.
- Căutarea sau ștergerea unui element se realizează cu complexitatea $\mathcal{O}(n)$.

Clasa `LinkedList`

Clasa `LinkedList` oferă o implementare a unei liste utilizând o listă dublu înlănțuită, astfel fiecare nod al listei conține o informație de tip generic `E`, precum și două referințe: una către nodul anterior și una către nodul următor.

Constructorii clasei `LinkedList` sunt:

- `LinkedList()` – creează o listă vidă;
- `LinkedList(Collection C)` – creează o listă din elementele colecției `C`.

Pe lângă metodele implementate din interfața `List`, clasa `LinkedList` conține și câteva metode specifice:

- accesarea primului/ultimului element: `E getFirst(), E getLast();`
- adăugarea la începutul/sfârșitul listei: `void addFirst(E elem), void addLast(E elem);`
- ștergerea primului/ultimului element: `E removeFirst(),E removeLast().`

Exemple:

```
LinkedList<String> lista = new LinkedList<>();

// adăugarea unor elemente în listă
lista.add("A");
lista.add("B");
lista.addLast("C");
lista.addFirst("D");
lista.add(2, "E");
lista.add("F");
lista.add("G");
System.out.println(lista); // [D, A, E, B, C, F, G]
```

```
// ștergerea unor elemente din  listă
lista.remove("B");
lista.remove(3);
lista.removeFirst();
lista.removeLast();
System.out.println(lista); // [A, E, F]

// căutarea unui element în listă
boolean rezultat = lista.contains("E");
System.out.println(rezultat); // true

// operații de accesare a unui element
Object element = lista.get(2);
System.out.println(element); // F
lista.set(2, "Y");
System.out.println(lista); // [A, E, Y]
```

Observații:
- Accesarea unui element se realizează cu complexitatea $\mathcal{O}(n)$.
- Adăugarea unui element la sfârșitul listei, folosind metoda `add(E elem)`, se realizează cu complexitatea $\mathcal{O}(1)$.
- Adăugarea unui element pe poziția `index`, folosind metoda `add(E elem, int index)`, se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Căutarea unui element se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Ștergerea unui element se realizează cu o complexitate egală cu $\mathcal{O}(n)$.

Pentru ca o aplicație să obțină performanțe cât mai bune din punct de vedere al timpului de executare, trebuie selectată colecția corespunzători funcționalității aplicației, astfel:
- dacă operațiile de accesare sunt predominante în cadrul aplicației, atunci se preferă utilizarea colecției `ArrayList`;
- dacă operațiile de actualizare (inserare/ștergere) sunt predominante, atunci se preferă utilizarea colecției `LinkedList`.

Pe lângă cele două implementări ale interfeței `List`, mai există și alte clase care o implementează:
- clasa `Vector` – are o funcționalitate similară clasei `ArrayList` și oferă metode sincronizate specifice aplicațiilor cu mai multe fire de executare;
- clasa `Stack` – extinde clasa `Vector` și oferă o implementare a unui vector cu funcționalitățile specifice structurii de date *stiva* (LIFO) .

Interfața Set

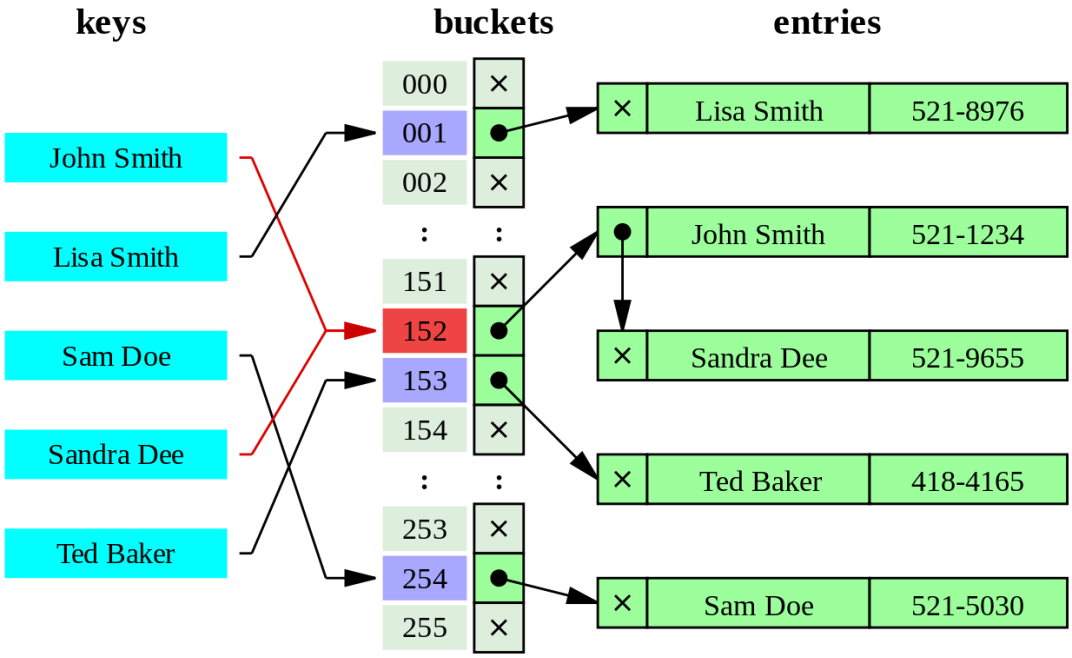
Interfața `Set` extinde interfața `Collection` și modelează o colecție de elemente care nu conțin duplicate, respectiv o colecție de tip mulțime.

Interfața `Set` nu adaugă metode suplimentare celor existente în interfața `List` și este implementată în clasele `HashSet`, `TreeSet` și `LinkedHashSet`.

Clasa `HashSet`

Într-un curs anterior, am văzut cum fiecare obiect are asociat un număr întreg, numit *hash-code*, puternic dependent față de orice modificare a datelor membre ale obiectului. Hash-code-ul unui obiect se calculează în metoda `int hashCode()`, moștenită din clasa `Object`, folosind algoritmi specifici care implică valorile datelor membre relevante pentru logica aplicației.

Clasa `HashSet` implementează o mulțime folosind o *tabelă de dispersie* (hash table). O tabelă de dispersie este un tablou unidimensional, numit *bucket array*, în care indexul unui element se calculează pe baza hash-code-ului său. Fiecare componentă a bucket array-ului va conține o listă cu obiectele care au același hash-code (*coliziuni*), așa cum se poate observa în figura de mai jos (sursa: https://en.wikipedia.org/wiki/Hash_table):



Practic, o operație de inserare a unui obiect în tabela de dispersie presupune parcurgerea următorilor pași:

- se apelează metoda `hashCode` a obiectului respectiv, iar valoarea obținută se utilizează pentru a calcula indexul bucket-ului asociat obiectului respectiv;
- dacă bucket-ul respectiv este vid, atunci se adaugă direct obiectul respectiv și operația de inserare se încheie;
- dacă bucket-ul respectiv nu este vid, se parcurge lista asociată și, folosind metoda `equals`, se verifică dacă obiectul este deja inserat în tabelă, iar în caz negativ obiectul se adaugă la sfârșitul listei.

Evident, într-un mod asemănător, se vor efectua și operațiile de căutare, actualizare sau ștergere.

Se observă foarte ușor faptul că performanțele unei tabele de dispersie sunt puternic influențate de performanțele algoritmului de calcul al hash-code-ului unui obiect, respectiv acesta trebuie să fie sensibil la orice modificare a datelor membre pentru a minimiza numărul de coliziuni (obiecte diferite din punct de vedere al conținutului, dar care au același hash-code), ideal fiind ca hash-code-ul unui obiect să fie unic. În acest caz, lista asociată oricărui bucket va fi foarte scurtă, deci operațiile de căutare/inserare/ștergere/modificare vor avea complexitatea $\mathcal{O}(1)$, altfel, în cazul existenței multor coliziuni, complexitatea poate ajunge $\mathcal{O}(n)$.

Un alt aspect foarte important îl constituie implementarea/rescrierea corectă și a metodei `equals`, moștenită tot din clasa `Object`, în concordanță cu implementarea metodei `hashCode`, respectând următoarele reguli:

- metoda `hashCode` trebuie să returneze aceeași valoare în timpul rulării unei aplicații, indiferent de câte ori este apelată, dacă starea obiectului nu s-a modificat, dar nu trebuie să furnizeze aceeași valoare în cazul unor rulări diferite;
- două obiecte egale din punct de vedere al metodei `equals` trebuie să fie egale și din punct de vedere al metodei `hashCode`, deci trebuie să aibă și hash code-uri egale;
- nu trebuie neapărat ca două obiecte diferite din punct de vedere al conținutului să aibă hash-code-uri diferite, dar, dacă acest lucru este posibil, se vor obține performanțe mai bune pentru operațiile asociate unei tabele de dispersie.

Dacă a doua regulă nu este respectată, adică două obiecte egale din punct de vedere al conținutului (metoda `equals`) au hash-code-uri diferite (metoda `hashCode`), atunci operațiile de căutare/inserare într-o tabelă de dispersie vor fi incorecte. Astfel, în cazul în care se încearcă inserarea celui de-al doilea obiect după inserarea primului, operația de căutare a celui de-al doilea obiect se va efectua după valoarea hash-code-ului său, diferită de cea a primului obiect, deci îl va căuta în alt bucket și nu îl va găsi, ceea ce va conduce la inserarea și a celui de-al doilea obiect în tabela, deși el are același conținut cu primul obiect!

De obicei, acest aspect negativ apare în momentul în care programatorul nu rescrie metodele `hashCode` și `equals` într-o clasă ale cărei instanțe vor fi utilizate în cadrul unor colecții bazate pe tabele de dispersie, deoarece, implicit, metoda `hashCode` furnizează o valoare calculată pe baza referinței obiectului respectiv, iar metoda `equals` testează egalitatea a două obiecte comparând referințele lor. Astfel, două obiecte diferite cu același conținut vor fi considerate diferite de metoda `equals` și vor avea hash-code-uri diferite!

Exemplu: Considerăm definită clasa Persoana în care nu am rescris metodele `hashCode` și `equals`:

```
HashSet<Persoana> lp = new HashSet<>();
Persoana p1 = new Persoana("Popescu Ion", 23);
Persoana p2 = new Persoana("Popescu Ion", 23);

lp.add(p1);
lp.add(p2);

System.out.println(lp.size());
```

În urma rulării secvenței de cod de mai sus, se va afișa valoarea `2`, deoarece ambele obiecte `p1` și `p2` vor fi inserate în HashSet-ul `lp`! Evident, problema se rezolvă implementând corect metodele `equals` și `hashCode` în clasa `Persoana`.

O problemă asemănătoare apare dacă se modifică valoarea unei date membre a unui obiect care este folosit pe post de cheie într-un HashMap (de exemplu, se modifică numele unei persoane), în cazul în care dacă valoarea datei membre respective este utilizată în implementările metodelor `hashCode` și `equals`. Din acest motiv, pentru chei se recomandă utilizarea unor obiecte care sunt instanțe ale unor clase imutabile!

Observații:

- Într-un HashSet se poate insera și valoare `null`, evident, o singură dată.
- O colecție de tip `HashSet` nu păstrează elementele în ordine inserării lor și nici nu pot efectua operații de sortare asupra sa.
- Implicit, *capacitatea* inițială (numărul de bucket-uri) a unei colecții de tip HashSet este 16, iar apoi aceasta este incrementată pe măsură ce se inserează elemente. Capacitatea inițială se poate stabili în momentul instanțierii sale, folosind constructorul `HashSet(int capacitate)`. În plus, pentru o astfel de colecție este definit un *factor de umplere* (load factor) care reprezintă pragul maxim permis de populare a colecției, depășirea sa conducând la dublarea capacității acesteia. Implicit, factorul de umplere este egal cu valoarea 0.75, ceea ce înseamnă că după ce se vor utiliza 75% din numărul de bucket-uri curente, numărul acestora va fi dublat. Astfel, considerând valorile implicite, prima dublare a numărului de bucket-uri va avea loc după ce se vor ocupa 0.75*16 = 12 bucket-uri, a doua dublare după ce se vor ocupa 0.75*32 = 24 de bucket-uri ș.a.m.d.

Clasa `LinkedHashSet`

Implementarea clasei `LinkedHashSet` este similară cu implementarea clasei `HashSet`, diferența constând în faptul că elementele vor fi stocate în ordinea inserării lor.

Exemplu: Pentru a găsi numerele distincte dintr-un fișier text, vom utiliza un obiect `nrdist` de tip `HashSet` în care vom insera, pe rând fiecare număr din fișier:

```
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner in = new Scanner(new File("numere.txt"));
        HashSet<Integer> nrdist = new HashSet();

        while (in.hasNextLine()) {
            String linie = in.nextLine();
            String[] numere = linie.split("[ ,.;:?!]+");
            for (String nr : numere)
                nrdist.add(Integer.parseInt(nr));
        }

        System.out.println("Valorile distincte din fisier: ");
        for (int x : nrdist)
            System.out.print(x + " ");

        in.close();
    }
}
```

După executarea programului de mai sus, se vor afișa valorile distincte din fișierul text, într-o ordine oarecare. Dacă în locul clasei HashSet vom utiliza clasa LinkedHashSet, atunci valorile distincte vor fi afișate în ordinea inserării, adică în ordinea în care ele apar în fișierul text.

Clasa `TreeSet`

Intern, clasa `TreeSet` implementează o mulțime utilizând un arbore binar de tip Red-Black pentru a stoca elemente într-o anumită ordine, respectiv în ordinea lor naturală când se utilizează constructorul fără parametri ai clasei și clasa corespunzătoare obiectelor implementează interfața `Comparable` sau într-o ordine specificată în constructorul clasei printr-un argument de tip `Comparator`:

```
TreeSet t = new TreeSet();
TreeSet t = new TreeSet(Comparator comp);
```

Observații:

- Metodele `add`, `remove` și `contains` au o complexitate specifică structurii arborescente binare de tip Red-Black, respectiv $\mathcal{O}(\log_2 n)$.
- Colecția `TreeSet` este utilă în aplicații care necesită stocarea unui număr mare de obiecte sortate după un numit criteriu, regăsirea informației fiind rapidă.

Exemplu: Revenind la exemplu anterior, dacă dorim să valorile distincte în ordine descrescătoare, mai întâi definim comparatorul

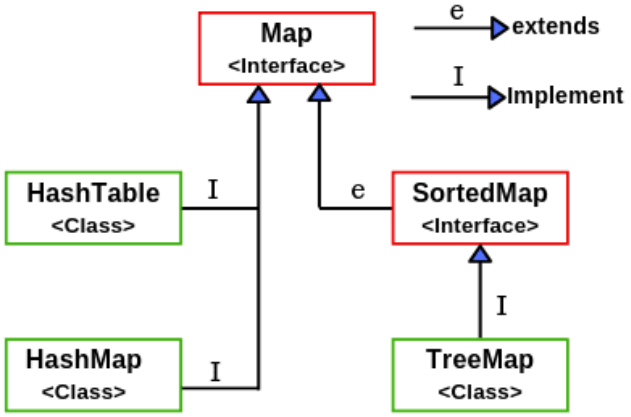
```
class cmpNumere implements Comparator<Integer> {
    @Override
    public int compare(Integer x, Integer y) {
        return y - x;
    }
}
```

și utilizăm constructorul corespunzător al clasei `TreeSet`:

```
TreeSet<Integer> nrdist = new TreeSet(new cmpNumere());
```

Interfața Map

Interfața Map, deși face parte din framework-ul Java Collections, nu extinde interfața Collection, ci este rădăcina unei ierarhii separate, așa cum se poate observa în figura de mai jos (sursa: https://www.scientecheasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0)



Interfața `Map` modelează comportamentul colecțiilor ale căror elemente sunt de perechi de tipul *cheie* – *valoare* (definite în interfața `Map.Entry<T,R>`), prin care se asociază unei chei care trebuie să fie unice o singură valoare. Exemple: număr de telefon – persoană, CNP – persoană, cuvânt – frecvența sa într-un text, număr matricol – student etc.

Câteva metode uzuale din interfața `Map` sunt următoarele:

- `R put (T cheie, R valoare)` – inserează perechea `cheie-valoare` în colecție în cazul în care cheia nu există deja și returnează `null`, iar altfel înlocuiește vechea valoare asociată cheii cu noua valoare și returnează vechea valoare;
- `R get(T cheie)` – returnează valoarea asociată cheii indicate sau `null` dacă în colecție nu există cheia respectivă;
- `boolean containsKey(T cheie)` – returnează `true` dacă în colecție există cheia respectivă sau `false` în caz contrar;
- `boolean containsValue(R valoare)` – returnează `true` dacă în colecție există valoarea respectivă sau `false` în caz contrar;
- `Set<Map.Entry<K,V>> entrySet()` – returnează o mulțime care conține toate perechile cheie-valoare din colecție;
- `Set<K> keySet()` – returnează o mulțime care conține toate cheile din colecție;
- `Collection<V> values()` – returnează o colecție care conține toate valorile din colecția de tip `Map`;
- `R remove(Object cheie)` – dacă în colecție există cheia indicată, atunci elimină din colecție perechea având cheia respectivă și returnează valoarea cu care era asociată, altfel returnează `null`;
- `boolean remove(Object cheie, Object valoare)` – dacă în colecție există perechea `cheie-valoare` dată, atunci o elimină și returnează true, altfel returnează `false`;
- `void clear()` – elimină toate perechile existente în colecție.

Interfața `Map` este implementată în clasele `HashMap` și `TreeMap`, pe care le vom prezenta în continuare.

Clasa `HashMap`

Intern, implementarea clasei HashMap utilizează o tabelă de dispersie în care indexul bucket-ului în care va fi plasată o anumită valoare este dat de hash-code-ul corespunzător cheii (`cheie.hashCode()`), deci toate operațiile de căutare/inserare/ștergere se vor efectua în funcție de hash-code-ul cheii!

Complexitățile minime și medii ale metodelor `get`, `put`, `containsKey` și `remove` sunt $\mathcal{O}(1)$ în cazul implementării în metoda hashCode() a unei funcții de dispersie bune, care generează valori uniform distribuite, dar se poate ajunge la o complexitate maximă egală cu $\mathcal{O}(n)$, unde n reprezintă numărul de elemente din HashMap-ul respectiv, în cazul utilizării unei funcții de dispersie slabe, care produce multe coliziuni.

Observații:

- Într-un HashMap este permisă utilizarea valorii `null` atât pentru cheie, cât și pentru valoare.
- Într-un HashMap se poate asocia aceeași valoare mai multor chei.
- Într-un HashMap nu se menține ordinea de inserare și nici nu se poate stabili o anumită ordine a perechilor!
- Într-un HashMap se pot realiza și mapări complexe:

```
//h1 conține studenții anului I folosind perechi număr_matricol - student
HashMap<String, Student> h1 = new HashMap<>();
//h2 conține studenții anului II folosind perechi număr_matricol - student
HashMap<String, Student> h2 = new HashMap<>();
//m conține studenții din fiecare an folosind perechi an_studiu – studenti
HashMap<Integer, HashMap<String, Student>> m = new HashMap();

h1.put("11111", new Student("Ion Popescu", 141, new int[]{10, 9, 10, 7, 8}));
h1.put("22222", new Student("Anca Pop", 142, new int[]{9, 10, 10, 8}));
h2.put("12121", new Student("Ana Ionescu", 241, new int[]{8, 9, 10}));
h2.put("12345", new Student("Radu Mihai" , 242 , new int[]{9, 10, 8}));

m.put(1, h1);
m.put(2, h2);

for(Map.Entry<Integer, HashMap<String, Student>> hms : m.entrySet()) {
      System.out.println("An " + hms.getKey() + ": ");
      for(Map.Entry<String, Student> s : hms.getValue().entrySet())
            System.out.println(s);
   }
```

Exemplu: Pentru a calcula frecvența cuvintelor dintr-un fișier, vom folosi un HashMap cu perechi de forma *cuvânt – frecvență_cuvânt*. Fiecare cuvânt din fișier va fi căutat în HashMap și dacă nu există deja, va fi inserat cu frecvența 1, altfel i se va actualiza frecvența mărită cu 1 (prin reinserare):

```
public class Test {
   public static void main(String[] args) throws FileNotFoundException {
         Scanner in = new Scanner(new File("exemplu.txt"));
         HashMap<String, Integer> fcuv = new HashMap();

         while(in.hasNextLine()) {
               String linie = in.nextLine();
               String []cuvinte = linie.split("[ ,.;:?!]+");
               for(String cuvant : cuvinte)
                     if(fcuv.containsKey(cuvant))
                           fcuv.put(cuvant, fcuv.get(cuvant) + 1);
                     else
                           fcuv.put(cuvant, 1);
         }

         System.out.println("Frecventele cuvintelor din fisier: ");
         for(Map.Entry<String, Integer> aux : fcuv.entrySet())
               System.out.println(aux.getKey() + " -> " + aux.getValue());

         in.close();

   }
}
```

Clasa TreeMap

Intern, implementarea clasei TreeMap utilizează un arbore binar de tip Red-Black pentru a menține perechile *cheie-valoare* sortate fie în ordine naturală a cheilor, dacă se utilizează constructorul fără parametri, fie în ordinea indusă de un comparator transmis ca parametru al constructorului. Astfel, dacă în exemplul anterior înlocuim obiectul de tip HashMap cu un obiect de tip TreeMap și utilizăm tot constructorul fără argumente, cuvintele din fișier vor fi afișate în ordine alfabetică. Dacă dorim să afișăm cuvintele în ordinea crescătoare a lungimilor lor, iar în cazul unor cuvinte de lungimi egale în ordine alfabetică, definim comparatorul

```
class cmpCuvinte implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        if(s1.length() != s2.length())
            return s1.length() - s2.length();
        else
            return s1.compareTo(s2);
    }
}
```

și utilizăm constructorul corespunzător din clasa TreeMap:

```
TreeMap<String, Integer> fcuv = new TreeMap(new cmpCuvinte());
```

Observații:

- Perechile dintr-un TreeMap pot fi sortate doar folosind criterii care implică doar cheile, ci nu și valorile, deci un comparator care va fi transmis ca parametru constructorului clasei TreeMap trebuie să țină cont de această restricție! Pentru a sorta perechile folosind criterii care implică valorile, de obicei, se preferă extragerea tuturor perechilor într-o colecție care permite realizarea operației de sortare după diverse criterii într-un mod simplu (de exemplu, o listă sau un tablou unidimensional).
- Operațiile de inserare/căutare/ștergere într-un TreeMap se realizează tot pe baza hash-code-ului corespunzător cheii, dar utilizarea unui arbore Red-Black garantează o complexitate egală cu $\mathcal{O}(\log_2 n)$ pentru metodele `get`, `put`, `containsKey` și `remove`.

Interfața `Iterator`

Rolul general al unui iterator este acela de a parcurge elementele unei colecții de orice tip, mai puțin a celor care fac parte din ierarhia interfeței `Map`. Orice colecție `c` din ierarhia interfeței `Collection` conține o implementare a metodei `iterator()` care returnează un obiect de tip `Iterator<Tip>`:

```
Iterator itr = c.iterator();
```

În interfața `Iterator` sunt definite următoarele metode pentru accesarea elementelor unei colecții:

- `public Object next()` – returnează succesorul elementului curent;
- `public boolean hasNext()` – returnează `true` dacă în colecție mai există elemente nevizitate sau `false` în caz contrar.

Un iterator nu permite modificarea valorii elementului curent și nici adăugarea unor elemente noi în colecție!

Exemplu:

```
LinkedList<String> lista = new LinkedList<>();

lista.add("Ion");
lista.add("Vasile");
lista.addLast("Ana");
lista.addFirst("Radu");
lista.add(2, "Ioana");

Iterator itr = lista.iterator();
while(itr.hasNext())
   System.out.println(itr.next());
```

Orice colecție conține metode `remove` pentru ștergerea unui element având o anumită poziție și/sau o anumită valoare. Totuși, în cazul în care o colecție este parcursă fie "clasic", utilizând o instrucțiune de tip *enhanced-for*, fie cu un iterator, aceste metode nu pot fi utilizate, așa cum vom vedea în următoarele două exemple:

Exemplu 1: Ștergerea valorilor egale cu 1 dintr-o listă folosind o instrucțiune de tip *enhanced-for*

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

lista.add(1);
............

for(Integer item:lista)
   if(item == 1)
         lista.remove(item);
```

Exemplu 2: Ștergerea numerelor pare dintr-o listă folosind un iterator

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

numere.add(101);
...............

Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
   Integer nr = itr.next();
   if (nr % 2 == 0)
         numere.remove(nr);
}
```

Deși apelul metodei `remove` este formal corect, în momentul executării secvențelor de cod de mai sus apare excepția `ConcurrentModificationException`, deoarece operația de ștergere se realizează în timpul iterării colecției!

De obicei, această excepție apare în aplicații multi-thread (aplicații cu mai multe fire de executare), unde nu este permis ca un fir de executare să modifice o colecție în timp ce un alt fir de executare parcurge colecția respectivă. Totuși, excepția apare și în aplicații cu un singur fir de executare, dacă se realizează parcurgea unei colecții cu un iterator de tip *fail fast iterator*, așa cum este cel utilizat în implementarea internă a instrucțiunii *enhanced for*!

O soluție sigură pentru a șterge un element dintr-o colecție presupune utilizarea metodei `void remove()` a unui iterator atașat unei colecții. Această metodă `default` este definită în interfața `Iterator` și permite ștergerea elementului curent (elementul referit de iterator):

```
Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
   Integer number = itr.next();
   if (number % 2 == 0)
         itr.remove();
}
```

În concluzie, ștergerea unui element dintr-o colecție se poate realiza folosind metodele `remove` definite în colecția respectivă, dacă aceasta nu este parcursă într-o manieră *fail fast iterator*, sau utilizând metoda `remove` din interfața `Iterator`, în caz contrar.

JAVA DATABASE CONNECTIVITY (JDBC)

O modalitate de a se asigura persistența datelor în cadrul unei aplicații o reprezintă utilizarea unei *baze de date*. O bază de date este gestionată de un sistem de gestiune a bazelor de date (SGBD) dedicat, de obicei aflat pe un server, astfel încât baza de date poate fi utilizată, în mod independent și transparent, de mai multe aplicații, posibil implementate în limbaje de programare diferite.

Java DataBase Connectivity (JDBC) este un API dedicat accesării bazelor de date din cadrul unei aplicații Java, care permite conectarea la un server de baze de date, precum și executarea unor instrucțiuni SQL. Accesarea unei baze de date din cadrul unei aplicații Java se realizează într-o manieră transparentă, independentă de sistemul de gestiune al bazelor de date utilizat. Practic, pentru fiecare SGBD există un driver dedicat (un program instalat local) care transformă cererile efectuate din cadrul programului Java în instrucțiuni care pot fi înțelese de către SGBD-ul respectiv (Fig. 1). Există mai multe tipuri de drivere disponibile, însă, în prezent, cele mai utilizate sunt *driverele native Java*. Acestea sunt scrise complet în limbajul Java și folosesc socket-uri pentru a comunica direct cu o bază de date, obținându-se astfel o performanță ridicată din punct de vedere al timpului de executare.

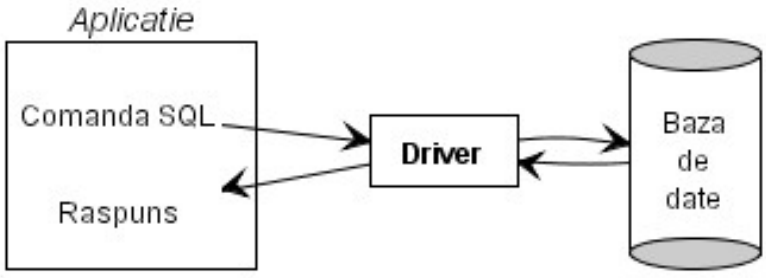


Figura 1. Comunicarea dintre o aplicație Java și un SGBD (https://profs.info.uaic.ro/~acf/java/slides/ro/jdbc_slide.pdf)

Procedura de instalare a unui drivere poate fi diferită de la un driver la altul. De exemplu, în cazul driverului MySQL, driverul este o arhivă de tip jar. Într-un mediu de dezvoltare, driverul poate fi specificat sub forma unei biblioteci atașată proiectului sau poate fi deja disponibil (de exemplu, versiunile noi de NetBeans conțin suport implicit pentru MySQL). Astfel, JDBC dispune de clasa `DriverManager` care administrează încărcarea driverelor, precum și obținerea conexiunilor către baza de date (Fig. 2). Odată conexiunea deschisa, JDBC oferă clientului un API care nu depinde de softul de baze de date folosit, ceea ce facilitează eventuale migrări între diferite SGBD-uri. Cu alte cuvinte, nu este necesar să scriem un program pentru a accesa o bază de date Oracle, alt program pentru a accesa o bază de date Sybase etc., ci este suficient să scriem un singur program folosind API-ul JDBC, iar acesta va fi capabil să comunice cu drivere diferite, trimițând secvențe SQL către baza de date dorită.

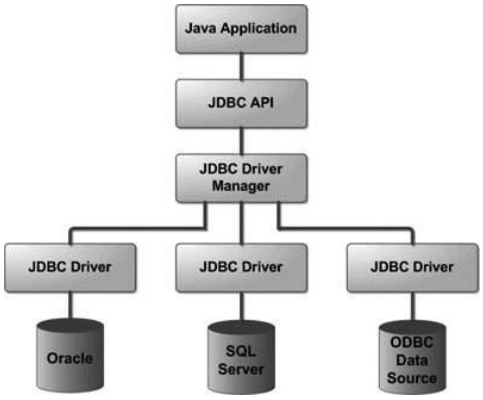


Figura 2. JDBC Driver Manager (https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm)

Arhitectura JDBC

Nucleul JDBC conține o serie de clase și interfețe aflate în pachetul `java.sql`, precum:

- Clasa `DriverManager`**:** gestionează driver-ele JDBC instalate și alege driver-ul potrivit pentru realizarea unei conexiuni la o bază de date;
- Interfața `Connection`**:** gestionează o conexiune cu o bază de date (orice comandă SQL este executată în contextul unei conexiuni);
- Interfețele `Statement / PreparedStatement / CallableStatement`**:** sunt utilizate pentru a executa comenzi SQL în SGBD sau pentru a apela proceduri stocate;
- Interfața `ResultSet`**:** stochează sub forma tabelară datele obținute în urma executării unei comenzi SQL;
- Clasa `SQLException`**:** utilizată pentru tratarea erorilor specifice JDBC.

Etapele realizării unei aplicații Java folosind JDBC

1. Încărcarea driver-ului specific

Prima etapă constă din înregistrarea, pe mașina virtuală unde rulează aplicația, a driver-ului JDBC necesar pentru comunicarea cu baza de date respectivă. Acest lucru presupune încărcarea în memorie a clasei care implementează driver-ul și poate fi realizată prin apelul metodei statice `void forName(String driver)` din clasa `Class`. De exemplu, încărcarea unui driver pentru o conexiune cu MySQL se poate realiza prin `Class.forName("com.mysql.jdbc.Driver")`, iar pentru o conexiune cu Oracle prin `Class.forName("oracle.jdbc.OracleDriver")`.

Începând cu versiunea JDBC 4.0, inclusă în Java SE 6, acest pas nu mai este obligatoriu, deoarece, la prima încercare de conectare la o bază de date, mașina virtuală Java va încărca automat toate driver-ele disponibile (pe care le găsește în *class path*).

2. Stabilirea unei conexiuni cu o bază de date

După înregistrarea unui driver JDBC, acesta poate fi utilizat pentru a stabili o conexiune cu o bază de date de tipul respectiv. O *conexiune* (sesiune) la o bază de date reprezintă un context prin care sunt trimise secvențe SQL din cadrul aplicației către SGBD și sunt primite înapoi rezultatele obținute.

Având în vedere faptul ca pot exista mai multe drivere încărcate în memorie, se va specifica, pe lângă un identificator al bazei de date, și driverul care trebuie utilizat. Acest lucru se realizează prin intermediul unei adrese specifice, numită JDBC URL, având formatul `jdbc:sub-protocol:identificator`, unde:

- câmpul `sub-protocol` specifică tipul de driver care va fi utilizat (de exemplu `sqlserver`, `mysql`, `postgresql` etc.);
- câmpul `identificator` specifică adresa unei mașini gazdă (inclusiv un număr de port), numele bazei de date și, eventual, numele utilizatorului și parola sa.

De exemplu, pentru o conexiune cu o bază de date denumită `BD`, care este stocată local folosind SGBD-ul MySQL, poate fi utilizat URL-ul de tip JDBC `jdbc:mysql://localhost:3306/BD`, iar dacă s-ar utiliza SGBD-ul Oracle, atunci URL-ul ar putea fi `jdbc:oracle:thin:@localhost:1521:BD`.

Deschiderea unei conexiuni se realizează prin intermediul metodelor statice `Connection getConnection(String url)` sau `Connection getConnection(String url, String user, String password)` din clasa `DriverManager`. Ambele metode returnează un obiect de tip `Connection`, clasă care conține o serie de metode pentru a gestiona conexiunea cu baza de date.

Exemplu:

- Deschiderea unei conexiuni la baza de date `Firma`, găzduită local utilizând `MySQL`:

```
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/Firma");
sau
Connection con=DriverManager.getConnection ("jdbc:mysql://localhost:3306/Firma",
                                            "popescuion","12345");
```

- Deschiderea unei conexiuni la baza de date `Firma`, găzduită local utilizând `Apache Derby`:

```
Connection con=DriverManager.getConnection("jdbc:derby://localhost:1527/Firma");
sau
Connection con=DriverManager.getConnection ("jdbc:derby://localhost:1527/Firma",
                                            "popescuion","12345");
```

La primirea unui URL de tip JDBC, DriverManager-ul va parcurge lista driverelor încărcate în memorie (de exemplu, începând cu JDK 6, driver-ul pentru Apache Derby este încărcat automat), până când unul dintre ele va recunoaște URL-ul respectiv. Dacă nu există nici un driver potrivit, atunci va fi lansată o excepție de tipul `SQLException`, cu mesajul "`No suitable driver found for ...`".

3. Crearea unui obiect de tip `Statement`

După realizarea unei conexiuni cu o bază de date, acesta poate fi folosită pentru executarea unor comenzi SQL (interogarea sau actualizarea bazei de date), precum și pentru extragerea unor informații referitoarea baza de date (meta-date).

Obiectele de tip `Statement` sunt utilizate pentru a executa instrucțiuni SQL (interogări, actualizări ale datelor sau modificări ale structurii) în cadrul unei conexiuni.

JDBC pune la dispoziția programatorului 3 tipuri de statement-uri, sub forma a 3 interfețe:
- `Statement` – pentru comenzi SQL simple, fără parametri;
- `PreparedStatement` – pentru comenzi SQL parametrizate;
- `CallableStatement` – pentru apelarea funcțiilor sau procedurilor stocate.

Interfața `Statement`

Crearea unui obiect `Statement` se realizează apelând metoda `Statement createStatement()` pentru un obiect de tip `Connection: Statement stmt = con.createStatement()`.

Executarea unei secvențe SQL poate fi realizată prin intermediul următoarelor metode:

a) metoda `ResultSet executeQuery(String sql)` – este folosită pentru executarea interogărilor de tip `SELECT` și returnează un obiect de tip `ResultSet` care va conține rezultatul interogării sub o formă tabelară, precum și meta-datele interogării (de exemplu, denumirile coloanelor selectate, numărul lor etc.).

Exemplu
Extragerea datelor despre o persoană stocate în tabela `Angajati` din baza de date `Firma`:
```
String sql = "SELECT * FROM Angajati";
ResultSet rs = stmt.executeQuery(sql);
```

Pentru a parcurge înregistrările rezultate în urma unei interogări de tip `SELECT`, un obiect de tip `ResultSet` utilizează un cursor, poziționat inițial înaintea primei linii. În clasa `ResultSet` sunt definite mai multe metode pentru a muta cursorul în cadrul structurii tabelare, în scopul parcurgerii sale: `boolean first()`, `boolean last()`, `boolean next()`, `boolean previous()`. Toate cele 4 metode întorc valoarea `true` dacă mutarea cursorului a fost efectuată cu succes sau `false` în caz contrar.

Pentru a extrage informațiile de pe fiecare linie se utilizează metode de forma `TipData getTipData(int coloană)` sau `TipData getTipData(String coloană)`, unde `TipData` reprezintă tipul de dată al unei coloane, iar argumentul `coloană` indică fie numărul de ordine din cadrul tabelului (începând cu 1), fie numele acesteia.

Exemplu:
Afișarea datelor angajaților stocate în tabela `Angajati` din baza de date `Firma`:
```
while(rs.next())
   System.out.println(rs.getString("Nume") + " " + rs.getInt("Varsta") + " "  +
   rs.getDouble("Salariu"));
```

b) metoda `int executeUpdate(String sql)` – este folosită pentru executarea unor interogări SQL de tipul Data Manipulation Language (DML), care permit actualizări ale datelor de tipul `UPDATE/INSERT/DELETE`, sau de tipul Data Definition Language (DDL) care permit manipularea structurii bazei de date (de exemplu, `CREATE/ALTER/DROP TABLE`). Metoda returnează numărul de linii modificate în urma efectuării unor interogări de tip DML sau 0 în cazul interogărilor de tip DDL.

Exemple:

```
String qrySQL = "INSERT INTO Angajati VALUES('1234567890999',
                                                'Albu Ioan',3210.10)";
```

sau

```
String qrySQL = "UPDATE Angajati SET Salariu = 1.10*Salariu
                                                WHERE Salariu <= 2500";
```

sau

```
String qrySQL = "DELETE FROM Angajati WHERE Nume LIKE 'Geo%'";

int n = stmt.executeUpdate(qrySQL);
System.out.println("Au fost modificate " + n + " înregistrări!");
```

Interfața `PreparedStatement`

Crearea unui obiect de tip `PreparedStatement` se realizează apelând metoda `PreparedStatement prepareStatement(String sql)` pentru un obiect de tip `Connection` și primește ca argument o instrucțiune SQL cu unul sau mai mulți parametri. Fiecare parametru este specificat prin intermediul unui semn de întrebare (?). Obiectele de tip `PreparedStatement` sunt utilizate în cazul în care este necesară executarea repetată a unei interogări SQL, eventual cu valori diferite ale parametrilor, deoarece aceasta va fi precompilată, deci se va executa mai rapid.

Exemplu:

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
```

Obiectul `pstmt` conține o instrucțiune SQL precompilată care este trimisă către baza de date, însă pentru a putea fi executată este necesară stabilirea valorilor pentru fiecare parametru. Setarea valorilor parametrilor se realizează prin metode de tip `void setTipData(int index, TipData valoare)`, unde `TipData` este tipul de date corespunzător parametrului respectiv, iar prin argumentele metodei se specifică indexul parametrului (începând de la 1) și valoarea pe care dorim să i-o atribuim.

Exemplu:

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 45);
pstmt.executeUpdate();
```

Executarea unei instrucțiuni SQL folosind un obiect de tip `PreparedStatement` se realizează apelând una dintre metodele `ResultSet executeQuery()` sau `int executeUpdate()`, asemănătoare cu cele definite pentru un obiect de tip `Statement`.

Interfața `CallableStatement`

Această interfață este utilizată pentru executarea subprogramelor atașate unei baze de date, respectiv funcții și proceduri stocate. Diferențele dintre funcții și proceduri stocate sunt următoarele:

- procedurile sunt folosite pentru a efectua prelucrări în baza de date (de exemplu, operații de actualizare), în timp ce funcțiile sunt folosite pentru a efectua calcule (de exemplu, pentru a determina numărul de angajați care au salariul maxim);
- procedurile nu returnează nimic prin numele lor (dar pot returna mai multe valori prin parametrii de intrare-ieșire, într-un mod asemănător funcțiilor de tip `void` din C/C++), în timp ce funcțiile returnează o singură valoare (într-un mod asemănător funcțiilor din C/C++ care returnează un tip de date primitiv, diferit de `void`);
- procedurile pot avea parametrii de intrare, de ieșire și de intrare-ieșire, în timp ce funcțiile pot avea doar parametrii de intrare;
- funcțiile pot fi apelate în proceduri, dar invers nu.

Funcțiile și procedurile stocate sunt utilizate într-o bază de date pentru a efectua calcule sau prelucrări complexe. De exemplu, se poate implementa o funcție stocată care să calculeze profitul mediu adus de contractele pe care o firmă le are cu o altă firmă într-un anumit interval calendaristic sau se poate implementa o procedură stocată care să efectueze prelucrări complexe ale mai multor tabele.

Exemple:

a) o funcție stocată care calculează suma salariilor tuturor bărbaților sau tuturor femeilor dintr-o firmă, în raport de valoarea parametrului `Tip`:

```
CREATE FUNCTION totalSalarii(Tip VARCHAR(1)) RETURNS double
BEGIN
   DECLARE total DOUBLE;
   DECLARE aux CHAR;

   IF(Tip = 'B') THEN
        SET aux = '1';
   ELSE
        SET aux = '2';
   END IF;

   SELECT SUM(Salariu) INTO total FROM Angajati WHERE LEFT(CNP,1) = aux;
   RETURN total;
END
```

b) o procedură stocată care verifică dacă un angajat există în tabela `Angajați` (pe baza `CNP`-ului) și în raport de rezultatul obținut inserează sau actualizează datele sale:

```
CREATE PROCEDURE inserareAngajat(IN CNP VARCHAR(13), IN Nume VARCHAR(45),
                                        IN Salariu DOUBLE, OUT rezultat INT)
BEGIN
   DECLARE cnt INT;
   SELECT COUNT(*) INTO cnt FROM angajati WHERE angajati.CNP = CNP;
```

```
   IF(cnt = 0) THEN
      INSERT INTO Angajati VALUES(CNP,Nume,Salariu);
      SET rezultat = 1;
   ELSE
      UPDATE Angajati SET Angajati.Nume = Nume, Angajati.Salariu = Salariu
                                                      WHERE Angajati.CNP =CNP;
      SET rezultat = 2;
   END IF;
END
```

Apelarea funcții stocate `totalSalarii` definită mai sus necesită efectuarea următorilor pași:

- se creează un obiect `sfunc` de tip `CallableStatement` folosind un obiect `conn` de tip `Connection`:

  ```
  sfunc = conn.prepareCall("{?=call totalSalarii(?)}");
  ```

  Primul ? reprezintă valoarea returnată de funcție (“parametrul de ieșire”), iar cel dintre paranteze reprezintă parametrul de intrare. Dacă funcția ar fi avut mai mulți parametri de intrare, atunci se punea câte un ? pentru fiecare, de exemplu `funcție(?,?,?)`.

- se specifică tipul rezultatului întors de funcție - se spune că “se înregistrează parametrul de ieșire (valoarea returnată de funcție)”:

  ```
  sfunc.registerOutParameter(1, Types.DOUBLE);
  ```

  Valoarea 1 identifică primul ? din apelul metodei `prepareCall` de mai sus!

- se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`, asemănătoare cu cele definite pentru `PreparedStatement`:

  ```
  sfunc.setString(2, "B");
  ```

  Deoarece valoarea 1 identifică valoarea returnată de funcție, parametrii de intrare sunt numerotați de la 2!

- se execută funcția stocată:

  ```
  sfunc.execute();
  ```

- se preia rezultatul întors de funcția stocată, folosind metode de tipul `getTipData(int index_parametru_de_intrare)`:

  ```
  double total = sfunc.getDouble(1);
  ```

  Valoarea parametrului este 1 deoarece rezultatul întors de funcție se identifică prin numărul de ordine 1!

Apelarea procedurii stocate `inserareAngajat` definită mai sus necesită efectuarea următorilor pași:

- se creează un obiect `sproc` de tip `CallableStatement` folosind un obiect `conn` de tip `Connection`:

  ```
  sproc = conn.prepareCall("{call inserareAngajat(?,?,?,?)}");
  ```

Semnele de întrebare dintre paranteze reprezintă parametrii de intrare/ieșire/intrare-ieșire ai procedurii.

- se specifică tipurile parametrilor de ieșire ai procedurii:

```
sproc.registerOutParameter(4, Types.DOUBLE);
```

- se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`:

```
sproc.setString(1, "1234567890999");
sproc.setString(2, "Vasilescu Ion");
sproc.setDouble(3, 3333.33);
```

- se execută procedura stocată:

```
sproc.execute();
```

- se preiau eventualele rezultatele întoarse de procedura stocată, folosind metode de tipul `getTipData(int index_parametru_de_ieșire)`:

```
double rezultat = sproc.getInt(4);
```

4. Închiderea unei conexiuni cu o bază de date

Conexiunea cu o bază de date se închide utilizând metoda `void close()` din clasa `Connection`, dacă nu este utilizat un bloc de tip `try-with-resources`.