Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Collections Framework

Sipos Róbert

siposr@hit.bme.hu

2014. 03. 24.

Általános áttekintés

- A Java 2 platformban került bevezetésre a Collections Framework.
- A collection olyan objektum, mely objektumok egy csoportját tartalmazza (mint a Vector osztály).
- A Collections Framework (CF) egységes architektúra, mely lehetővé teszi ezen objektumok tárolását, illetve a konkrét megvalósítástól független adatmanipulációt.

CF előnyei

- Használatával könnyebb a programozás (az adatstruktúrák és algoritmusok léteznek, nem kell őket újra meg újra megvalósítani)
- Hatékonyabb (különböző implementációk, adott feladathoz méretezett)
- Különböző API-k közötti interoperabilitás
- Nem kell újabb meg újabb API-kat megismerni (illetve tervezni és implementálni)
- SW újrafelhasználás

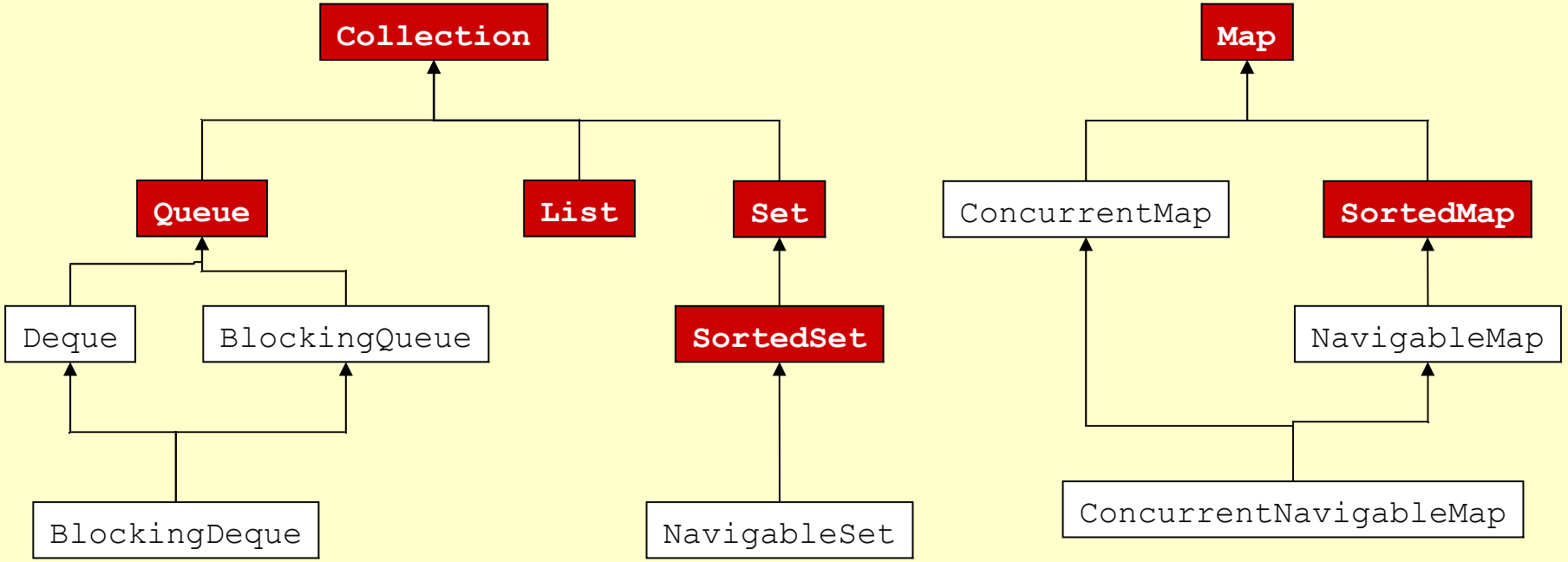
CF tartalma

- Collection Interface-ek
- Általános célú implementációk
- Korábbi implementációk (collection jellegű osztályok korábbi verziókból, pl. Vector, Hashtable)
- Speciális célú implementációk (hatékonyság)
- Konkurrens implementációk
- Wrapper implementációk (funkcionalitások pl. más implementációkkal történő szinkronizáció)
- Kényelmi implementációk (hatékony mini-implementációk)
- Abszrakt implementációk (részfunkciók)
- Algoritmusok (statikus metódusok: pl. listák rendezése)
- Infrastruktúra (a collection interfészek támogatására)
- Tömbök manipulálása

CF interfészek 1.

- `java.util` package
- 14 Collection interfész létezik.
- A legalapvetőbb interfész a `Collection`,
  - ☞ leszármazottai: `Set`, `List`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `BlockingQueue`, `BlockingDeque`
  - ☞ A másik collection interfész: `Map`, `SortedMap`, `NavigableMap`, `ConcurrentMap`, `ConcurrentNavigableMap`
    - ➢ Noha ezek az interfészek nem a `Collection` interfész leszármazottai, de tartalmaznak collection-kezelő műveleteket.
- A collection interfészekben definiált metódusok nagyrésze opcionális. Néhány implementáció nem valósítja meg ezeket a metódusokat, hívásuk `UnsupportedOperationException`-t eredményez.

CF Interfészek 2.



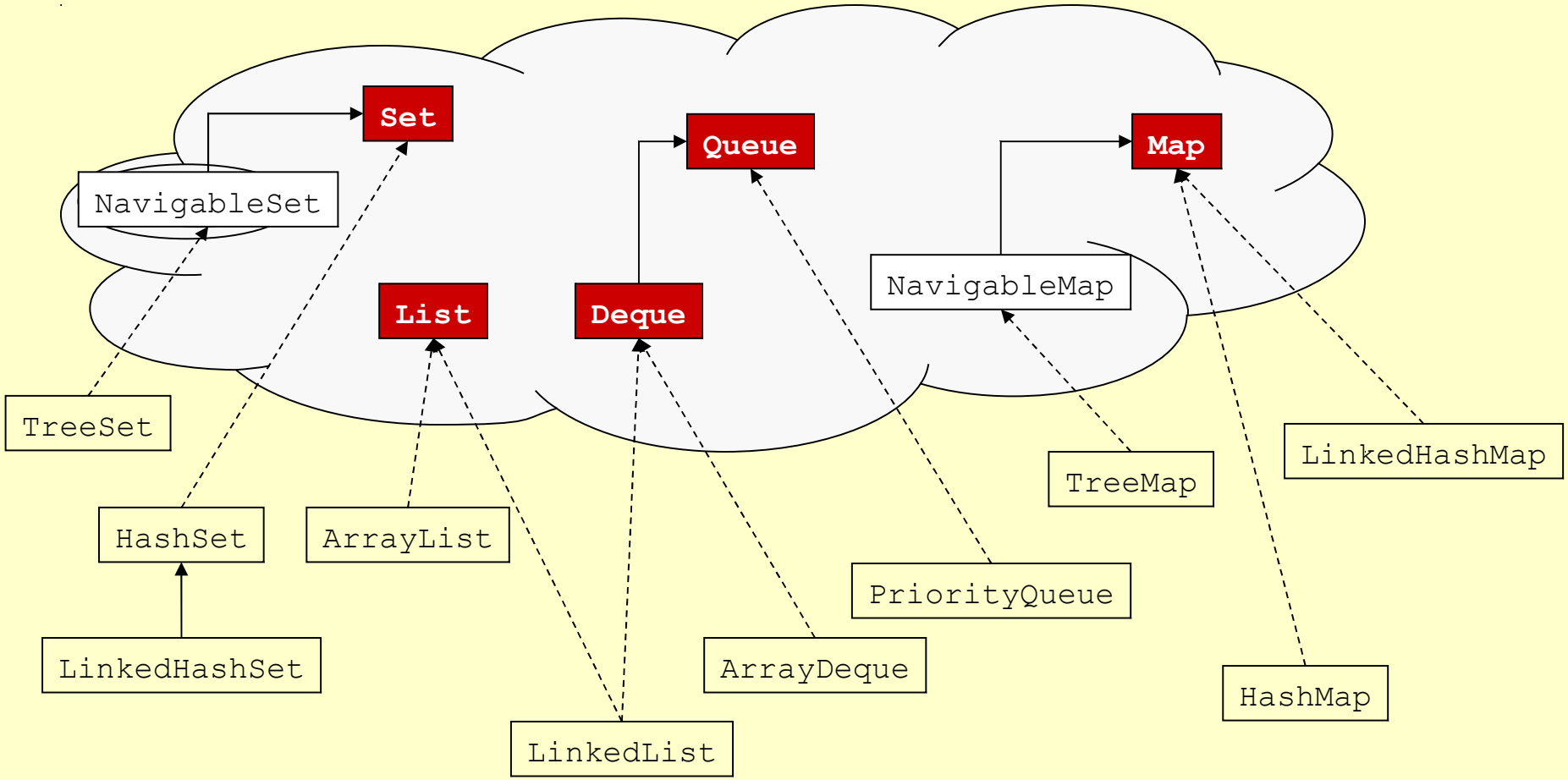
CF implementációk

- Elnevezési konvenciók
  - *<Implementációs stílus><Interface>*
- Általános célú implementációk
  - Minden opcionális műveletet implementálnak és a tartalmazott elemekre nézve nincsen korlátozás.
  - Nem szinkronizáltak, de a Collections osztály tartalmaz a szinkronizációs wrappereket
  - Minden implementáció tartalmaz ún. fail-fast iterátorokat, melyek az illegális konkurrens hozzáférést azonnal detektálják és azonnal leállnak (hibajelzéssel).

CF-ben használatos kifejezések

- ***modifiable*** / ***unmodifiable***
  - ☞ Azon collectionöket, melyek nem támogatják a módosító műveleteket (pl. add, remove, clear) nem módosíthatónak (unmodifiable) nevezzük.
- ***fixed size*** / ***variable size***
  - ☞ Azon collectionöket, melyek garantálják, hogy méretük állandó (habár a bennük tárolt objektumok változhatnak) fix méretű (fix size) collectionöknek nevezzük. A nem fix méretű collectionöket változó méretű collectionöknek (variable size) nevezzük.
- ***random access*** / ***sequental access***
  - ☞ Azon collectionöket, melyek a gyors hozzáférést (konstans időben) biztosítanak véletlen hozzáférésű (random access) collectionöknek, míg azokat, melyek ezt nem támogatják soros hozzáférésű (sequental access) collectionöknek nevezzük.

Általános célú implementációk



Wrapper implementációk

- `Collections.unmodifiable`<Interface>
  - ☞ a paraméterként kapott interfész nem módosítható nézetét adja vissza, mely `UnsupportedOperationException-t` dob, ha módosítani akarjuk
- `Collections.synchronized`<Interface>
  - ☞ a paraméterként kapott interfész szinkronizált (thread-safe) változatát adja vissza
- `Collections.checked`<Interface>
  - ☞ dinamikusan típusvédett változatot ad vissza

Kényelmi implementációk

- `Arrays.asList`
- `EMPTY_SET`, `EMPTY_LIST`, `EMPTY_MAP`
  - ☞ immutable set-et, listát, illetve map-et tartalmazó konstansok
- `singleton`, `singletonList`, `singletonMap`
  - ☞ olyan immutable (1 elemű) set-et, listát vagy map-et ad vissza, mely a paraméterként kapott elemet tartalamzza.
- `nCopies`
  - ☞ olyan listát ad vissza, mely az adott elemet n-szer tartalmazza.

Korábbi implementációk

- `Vector`
  - szinkronizált
  - resizable
  - tömb-alapú lista megvalósítás
  - újabb metódusokkal kiegészítve
- `Hashtable`
  - szinkronizált
  - hash tábla (Map) implementáció
  - null kulcsok/értékek nem megengedettek

Speciális célú és konkurrens implementációk

- Speciális
  - `WeakHashMap`
  - `IdentityHashMap`
  - `CopyOnWriteArrayList`
  - `CopyOnWriteArraySet`
  - `EnumSet`
  - `EnumMap`
- Konkurrens implementációk
  - `ConcurrentLinkedQueue`
  - `LinkedBlockingQueue`
  - `ArrayBlockingQueue`
  - `PriorityBlockingQueue`
  - `DelayQueue`
  - `SynchronousQueue`
  - `LinkedBlockingDeque`
  - `ConcurrentHashMap`
  - `ConcurrentSkipListSet`
  - `ConcurrentSkipListMap`

Absztrakt Implementációk

- `AbstractCollecion`
- `AbstractSet`
- `AbstractList`
- `AbstractSequentialList`
- `AbstractQueue`
- `AbstractMap`

Algoritmusok 1.

- A `Collections` osztály tartalmaz néhány hasznos (statikus) metódust
- `sort(List)`
  - listákat rendez merge sort algoritmussal garantált O(nlogn) hatékonyság, stabilitás (egyenlő elemek sorrendjét nem cseréli fel)
- `binarySearch (List, Object)`
  - bináris keresést végez
- `reverse(List)`
  - ellenkező irányba rendez
- `shuffle(List)`
  - az elemek véletlen permutációját állítja elő
- `fill(List, Object)`
  - a lista minden elemét az adott objektummal helyettesíti
- `copy(List dest, List src)`
  - a céllistába másolja a forráslista elemeit

Algoritmusok 2.

- `min(Collection)`
  - a legkisebb elemet adja vissza
- `max(Collection)`
  - a legnagyobb elemet adja vissza
- `rotate(List list, int distance)`
- `replaceAll(List list, Object oldVal, Object newVal)`
  - a listában minden oldVal elemet newVal elemre cserél.
- `indexOfSubList(List source, List target)`
  - a listában előforduló allista első megjelenésének indexét adja vissza
- `lastIndexOfSubList(List source, List target)`
- `swap(List, int, int)`
  - a lista adott elemeit kicseréli

Algoritmusok 3.

- `frequency (Collection Object)`:
  - hányszor szerepel az adott objektum a collectionben
- `disjoint (Collection, Collection)`
  - diszjunkt-e a két collection
- `addAll(Collection<? super T>, T...)`
  - a paraméterként kapott T típusú elemek mindegyikét a paraméterként kapott collectionhöz adja
- `newSetFromMap(Map)`
- `asLifoQueue(Deque)`
  - egy `Deque` tartalmát LIFO Queue-ként adja vissza.

Infrastruktúra

- Iterátorok
  - `Iterator`
  - `ListIterator`
- Rendezés
  - `Comparable` (natural ordering)
  - `Comparator`
- RuntimeExceptions
  - `UnsupportedOperationException`
  - `ConcurrentModificationException`
- Hatékonyság
  - `RandomAccess`
- Tömbök
  - `Arrays` osztály: statikus metódusok (rendezés, keresés, összehasonlítás, hash, másolás, átméretezés, `String` konverzió)

Interfészek áttekintése 1.

- `Collection`
  - A collection hierarchia csúcsa. Objektumok egy csoportját reprezentálja. Ezen objektumkat elemek-nek nevezzük.
- `Set`
  - A matematikai halmazfogalomnak megfelelő szerkezet. Két egyforma elemet nem tartalmazhat.
- `List`
  - Rendezett collection. Az elemeket index alapján lehet elővenni.
- `Queue`
  - Az egyszerű collection osztályhoz képest további műveleteket tartalmaz.
  - A `Queue-k` általában FIFO-k, de léteznek prioritásos sorok is, melyek az elemek természetes rendezésének megfelelően rendezik az elemeket.
- `Map`
  - Kulcs-érték leképezések. A kulcsok halmazt alkotnak.

Interfészek áttekintése 2.

- `SortedSet`
  - Olyan `Set`, mely az elemeket növekvő sorrendben tárolja
- `SortedMap`
  - A kulcsok szerint rendezetten tárolja az elemeket.

A Collection interfész 1.

- A `Collection` interfészt általában arra használjuk, hogy elemeket paraméterként átadjunk, és mindehhez a lehető legáltalánosabb adatszerkezetet használjuk.
- Pl. minden általános célú implementációnak van egy olyan konstruktora, mely egy `Collection`-t vesz át paraméterként.
- Egy `Collection<String> c` lehet lista, halmaz vagy tetszőleges osztály, mely megvalósítja a Collection interfészt.

```
List<String> list = new ArrayList<String> (c);
```

Collection interfész 2.

```
int size()
boolean isEmpty()
boolean contains(Object element)
boolean add(E element) (optional)
boolean remove(Object element) (optional)
Iterator<E> iterator()
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c) (optional)
boolean removeAll(Collection<? extends E> c) (optional)
boolean retainAll(Collection<? extends E> c) (optional)
void clear() (optional)
Object[] toArray()
<T> T[] toArray(T[] a)
```

Iterator-ok (Iterator interfész)

- Az `Iterator` olyan objektum, mely végigmegy egy `Collection` elemein, illetve adott esetben el tudja távolítani a `Collection` elemeit.
- A `Collection` interfész `iterator` metódusát hívva a `Collcetion` elemeit felsoroló `Iterator` objektumot kapjuk vissza.
  - `hasNext`: igaz, ha a felsorolásban van még elem
  - `next`: visszaadja a következő elemet a sorban
  - `remove`: az adott elemet eltávolítja a collectionből (next hívásonként egyszer szabad hívni)
    - iteráció alatt ez az egyetlen biztonságos mód elem eltávolítására

`Iterator<E>` Interfész metódusai

```
boolean hasNext ()
E next()
void remove()
```

opcionális

Elemek felsorolása (Collection)

- for-each konstrukció

```
for (Object o : collection)
        System.out.println (o);
```

- iterátorok (a for-each konstrukció elrejti az `Iterator`-t, így elem nem törölhető)

```
static void filter (Collection<?> c) {
   for (Iterator<?> it = c.iterator (); it.hasNext();)
      if (!condition(it.next())
         it.remove();
}
```

- A `Collection` elemei csak objektumok lehetnek (primitív típusok nem). Ha egy `Collection`höz primitív típust adunk, az autoboxing/autounboxing funkció működésbe lép.

Csoportos műveletek (Collection) 1.

- A csoportos műveletek a teljes `Collection`-ön értelmezettek.
- `containsAll`
  - ☞ igaz, ha az adott `Collection` a paraméterként kapott `Collection` minden elemét tartalmazza
- `addAll`
  - ☞ a paraméterként kapott `Collection` minden elemét az adott `Collection` elemeihez adja
- `removeAll`
  - ☞ minden olyan elemet eltávolít az adott `Collection`-ből, ami eleme a paraméterként kapott `Collection`-nek
- `retainAll`
  - ☞ minden olyan elemet eltávolít az adott `Collection`-ből, ami nem eleme a paraméterként kapott `Collection`-nek

Csoportos műveletek (Collection) 2.

- `clear`
  - ☞ minden elemet eltávolít a `Collection`-ből
- Az `addAll`, `removeAll` és `retainAll` műveletek igaz értéket adnak vissza, ha a collectionön történt módosítás.

```
c.removeAll (Collections.singleton (e));
c.removeAll (Collections.singleton (null));
```

Tömb műveletek (Collection)

- `toArray`
  - egy collection tartalmát tömbbé transzformálja
  - korábbi API-k tömb paramétert vártak
  - argumentum nélküli alak: új tömböt hoz létre
- `toArray(T[] array)`

```
Object[] a = c.toArray ();
String[] s = c.toArray (new String [0])
```

A Set interfész

- A `Set` egy olyan `Collection`, mely nem tartalmazhatja többször ugyanazt az elemet.
- A `Set` interfésznek a `Collection`től örökölt metódusai vannak azzal a megkötéssel, hogy duplikátumok nem szerepelhetnek benne.
- Általános célú megvalósítások:
  - `HashSet`
    - leghatékonyabb (hash-es) megvalósítás, de az elemek sorrendje (iterator) nem kötött
  - `TreeSet`
    - megvalósítása egy piros-fekete fa (önkiegyenlítő bináris B-fa)
    - lassabb, mint a HashSet, de az elemek sorrendje kötött (érték szerint rendezettek)
  - `LinkedHashSet`
    - megvalósítása egy hashtáblával kombinált láncolt lista
    - sorrend: a halmazba való bekerülés sorrendje (insertion-order)

Set interfész példa

- Duplikátumok kiszűrése

```
Collection<Type> noDups = new HashSet<Type> (c);
Collection<Type> noDupsOrderPreserved = new LinkedHashSet<Type> (c);


public static <E> Set<E> removeDups (Collection<E> c) {
    return new LinkedHashSet<E> (c);
}
```

Alapvető metódusok (Set)

- `size` : kardinalitás (elemek száma)
- `isEmpty` : az adott halmaz üres vagy sem
- `add`: az adott elemet a halmazhoz adja (ha még nincs benne), a visszatérési érték megmutatja, hogy történt-e változás a halmazon
- `remove`: eltávolítja az adott elemet a halmazból (ha benne van)
- `iterator` : felsorolja a halmaz elemeit

1. Példa (Set)

```
public class DuplicateFinder {
   public static void main (String[] args) {
      Set<String> s = new HashSet<String> ();
      for (String a : args)
         if (!s.add(a))
            System.out.println ("Duplicate detected: " + a);

      System.out.pritnln (s.size() + " distinct words: " : s);
   }
}
```

Csoportos műveletek (Set) 1.

- A `Collection` csoportos műveletei a `Set` interfészen gyakorlatilag a halmazműveleteket jelentik.
- `s1.containsAll(s2)`
  - ☞ s2 ⊆ s1(igaz, ha s2 az s1 halmaz részhalmaza)
- `s1.addAll(s2)`
  - ☞ s1 ∪ s2 (s1 és s2 halmazok uniója, s1 módosításával)
- `s1.retainAll (s2)`
  - ☞ s1 ∩ s2 (s1 és s2 halmazok metszete, s1 módosításával)
- `s1.removeAll (s2)`
  - ☞ s1 \ s2 (s1 halmaz és s2 halmaz aszimmetrikus különbsége, s1 módosításával)

Csoportos műveletek (Set) 2.

- Ha úgy akarjuk két halmaz unióját, metszetét illetve különbségét megkapni, hogy egyik halmazt sem módosítjuk:

```
Set<Type> union = new HashSet<Type> (s1);
Set<Type> intersection = new HashSet<Type> (s1);
Set<Type> difference = new HashSet<Type> (s1);

union.addAll(s2);
intersection.retainAll(s2);
difference.removeAll(s2);
```

2. Példa (Set)

```
public class DuplicateFinderv2 {
   public static void main (String[] args) {
      Set<String> uniques = new HashSet<String> ();
      Set<String> dups = new HashSet<String> ();

      for (String a : args)
         if (!uniques.add(a)) dups.add(a);
      uniques.removeAll(dups);
      System.out.println ("Unique words:    " + uniques);
      System.out.println ("Duplicate words: " + dups);
   }
}
```

3. Példa (Set)

- Szimmetrikus differencia (s1 ∪ s2) \ (s1 ∩ s2)

```
Set<Type> symmetricDiff = new HashSet<Type> (s1);
symmetricDiff.addAll(s2);
Set<Type> intersect = new HashSet<Type> (s1);
intersect.retainAll(s2);
symmetricDiff.removeAll(intersect);
```

A List interfész

- Rendezett `Collection` (sequence)
- Tartalmazhat duplikátumokat
- További metódusai (a `Collection` interfészen kívüliek)
  - elérés pozíció szerint (az adott elem numerikusan ábrázolt helye a listában)
  - keresés (eredmény az adott elem pozíciója)
  - felsorolás (a felsorolás műveletek kiegészítése a listára jellemző műveletekkel)
  - tartomány műveletek
- Az általános célú megvalósítás két `List` implementációt tartalmaz
  - `ArrayList` (általában a leghatékonyabb megvalósítás)
  - `LinkedList` (bizonyos körülmények között a leghatékonyabb)
  - Noha a `Vector` nem eleme a CF-nek, visszamenőleg kiegészült, hogy a `List` interfészt megvalósítsa

A List interfész metódusai

```
E get(int index)
E set(int index, E element) (opcionális)
boolean add(E element) (opcionális)
void add(int index, E element) (opcionális)
E remove(int index) (opcionális)
boolean addAll(int index,
   Collection<? extends E> c) (opcionális)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
List<E> subList(int from, int to)
```

A List interfész metódusai

- `get`, `indexOf`
- `set`, `remove`: a listában az adott pozíción szereplő korábbi elemet adja vissza
- `addAll`: a paraméterként kapott elemeket az adott pozíciótól kezdve másolja (szúrja be) a listába.

List kontra Vector

- A `List` használata könnyebb
  - rövidebb metódusnevek
  - felcserélt paraméterek

```
a[i] = a[j].times (a[k]);
v.setElementAt(v.elementAt(j).times(v.elementAt(k)),i);  // Vector
v.set(i, v.get(j).times(v.get(k)));
```

1. Példa (List)

- Elemcsere megvalósítása

```
public static <E> void swap (List<E> a, int i , int j) {
   E tmp = a.get (i);
   a.set (i, a.get (j));
   a.set (j, tmp);
}
```

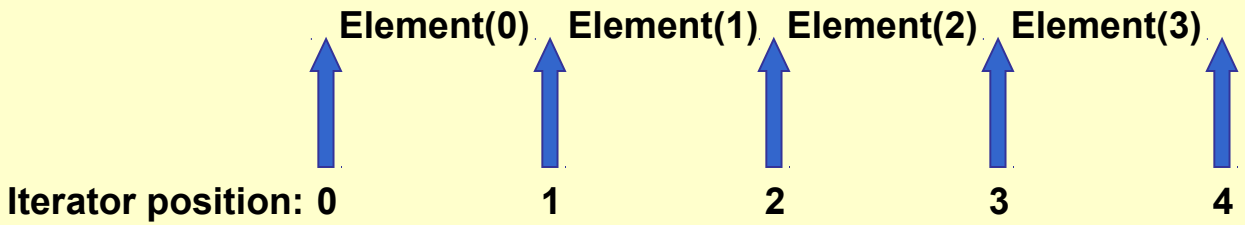
- Véletlen permutáció előállítása (*gyors* és *fair*)

```
public static void shuffle (List<?> list, Random rnd) {
   for (int i = list.size(); i > 1; i--)
      swap (list, i-1, rnd.nextInt(i);
}
```

Listaelemek felsorolása (ListIterator interfész)

- A `List` interfész által biztosított `ListIterator` interfészt megvalósító objektum mindkét irányú navigációt biztosít a listában, illetve lehetőséget ad a lista adott elemének módosítására is.
  - ☞ `hasPrevious`, `previous`: analóg a `hasNext`, `next` metódusokkal

```
ListIterator it = list.iterator ()
   it.hasNext ()
   it.next ()
```




```
it = list.listIterator (list.size())
   it.hasPrevious ()
   it.previous ()
```

ListIterator<E> interfész metódusai

```
boolean hasPrevious()
E previous()
int nextIndex()
int previousIndex()
void set(E e)
void add(E e)
```

opcionális

opcionális

+ az Iteratortól örököltek

ListIterator példák

- Elemek ellenkező irányban történő felsorolása:

```
for (ListIterator<Type> it = list.listIterator (list.size());
     it.hasPervoius ();) {
   Type t = it.previous ();
}
```

- Az add metódus egy új elemet ad az aktuális kurzorpozíció elé

```
public static <E> void replace (List<E> list, E val,
                                List<? extends E> newVals) {
   for (ListIterator<E> it = list.listIterator (); it.hasNext();)
      if (val == null ? it.next () == null :
                        val.equals (it.next ())) {
         it.remove ();
         for (E e : newVals) it.add (e);
      }
}
```

Tartományok

- `subList (int fromIndex, int toIndex)`
  - ☞ a lista adott szakaszának lista nézetét adja vissza `[from, toIndex)`.
- Nézet: a visszaadott listán történő módosítás az eredeti listát is módosítja.

```
list.subList (fromIndex, toIndex).clear ();
```

List-re értelmezett algoritmusok

- `sort`
- `shuffle`
- `reverse`
- `rotate`
- `swap`
- `replaceAll`
- `fill`
- `copy`
- `binarySearch`
- `indexOfSubList`
- `lastIndexOfSubList`

A Queue interfész

- A `Queue` interfész a feldolgozási sorokban tárolt elemeket reprezentálja.
- A `Collection` interfész által biztosított metódusokon kívül a `Queue` interfész a következő metódusokkal egészül ki:
  - `E element ();`
  - `boolean offer (E e);`
  - `E peek ();`
  - `E poll ();`
  - `E remove ();`
- A `Queue` metódusoknak alapvetően két formája létezik
  - Ha a művelet nem sikerül, `Exception` dobódik (`add`, `remove`, `element`)
  - A művelet sikertelenségére a visszatérési értékből lehet következtetni (`offer`, `poll`, `peek`)

Queue-k általában 1.

- Általában FIFO szerint rendezettek az elemek (First In First Out)
- A prioritásos sorok (priority queue) az elemek értéke szerint rendezettek.
- Az első elem (a sor feje) mindig az az elem, ami a következő poll/remove híváskor eltávolításra kerül
- Néhány implementációban lehetséges méretet megadni (bounded queues).
  - `java.util.concurrent`
  - `java.util implementations`
- Az `add` és az `offer` új elemet szúr be
  - `add` – `IllegalStateException`
  - `offer` – `false`

Queue-k általában 2.

- A `remove` és a `poll` metódusok eltávolítják a következő elemet (a sor fejét)
  - `remove` – `NoSuchElementException`
  - `poll` – `null`
- Az `element` és `peek` visszaadják, de nem törlik az első elemet.
  - `element` – `NoSuchElementException`
  - `peek` – `null`
- Általános esetben null elemet nem tehetünk egy sorba (kivéve a `LinkedList` implementációnál)
  - Ezt azonban nem ajánlatos kihasználni, hiszen a poll és a `peek` metódusok null értéket adnak vissza üres sor esetében.
- Általánosságban a sorok nem használják az `equals` és a `hashCode` metódusokat.

Queue metódusok összefoglalása

	Exception	Speciális visszatérési érték
Beszúrás	`add(e)`	`offer(e)`
Eltávolítás	`remove()`	`poll()`
Elemvizsgálat	`element()`	`peek()`

Queue példák

```
public class Countdowns {
   public static void main (String[] args) throws
                                                InterruptedException {
      int time = Integer.parseInt (args[0]);
      Queue<Integer> queue = new LinkedList<Integer> ();
      for (int i = time; i >= 0; i--) queue.add (i);
      while (!queue.isEmpty()) {
         System.out.println (queue.remove());
         Thread.sleep (1000);
      }
   }
}
```

```
static <E> List<E> heapSort (Collection<E> c) {
   Queue<E> queue = new PriorityQueue<E> (c);
   List<E> result = new ArrayList<E> ();
   while (!queue.isEmpty ()) result.add (queue.remove());
   return result;
}
```

A Map interfész

- A `Map` olyan adastruktúra, mely kulcsokhoz tárol értékeket. A kulcsok terében nem lehetnek duplikátumok.
  - matematikai függvény fogalomnak felel meg
- A Java platformban három általános célú megvalósítás létezik:
  - `HashMap`
  - `TreeMap`
  - `LinkedHashMap`
- A `Hashtable` utólagos módosítással lett alkalmas a `Map` interfész megvalósítására.

A Map interfész metódusai

```
public interface Map<K,V> {
   V put (K key, V value);
   V get (Object key);
   V remove (Object key);
   boolean containsKey (Object key);
   boolean containsValue (Object value);
   int size ();
   boolean isEmpty ();

   void putAll (Map<? extens K, ? extends V) m);
   void clear ();

   public Set<K> keySet ();
   public Collection<V> values ();
   public Set<Map.Entry<K,V>> entrySet ();

   public interface Entry<K,V> {
       K getKey ();
       V getValue ();
       V setValue (V value);
   }
}
```

Map műveletek 1.

```
public class Freq {
   public static void main (String[] args) {
      Map<String, Integer> m = new HashMap<String, Integer>();
      for (String a : args) {

         a = a.toLowerCase ();
         Integer freq = m.get(a);
         m.put (a, freq == null ? 1 : freq + 1);
      }
      System.out.println (m.size() + " distinct words: ");
      System.out.println (m);

   }
}
```

```
java Freq ha jöttök lesztek ha hoztok esztek
```

```
5 distinct words:

{hoztok=1, lesztek=1, esztek=1, jöttök=1, ha=2}

{esztek=1, ha=2, hoztok=1, jöttök=1, lesztek=1}  TreeMap
```

Map műveletek 2.

- Copy constructor

```
Map<K, V> copy = new HashMap<K, V> (m);
```

- A `putAll` metódus kicsit mást eredményez, mint Set esetében (az azonos kulcsokhoz a paraméterként kapott Map-ban lévő értékeket helyezi el)

```
static <K, V> Map<K, V> newAttributeMap (Map<K, V> defaults,
                                         Map<K, V> overrides) {
   Map<K, V> results = new HashMap<K, V> (defaults);
   result.putAll (overrides);
   return result;
}
```

Collection nézetek

- A Map interfésznek három `Collection` nézete van:
  - `keySet`
  - `values`
  - entrySet
- A `Collection` view-k azt biztosítják, hogy egy Map elemeit fel lehessen sorolni.

```
for (KeyType key : m.keySet ()) System.out.println (key);

for (Iterator<Type> it = m.keySet().iterator (); it.hasNext ();) {
   if (it.next().isRemovable()) it.remove ();

}

for (Map.Entry<KeyType, ValType> e : m.entrySet ()) {
   System.out.pritnln (e.getKey () + ": " + e.getValue ());
}
```

  - nem hoz mindig létre új példányt

Collection nézetek tulajdonságai

- Mindhárom `Collection` nézet esetében az `iterator.remove` hatására az egész bejegyzést törlődik `Map`-ből.
- Az `entrySet` view segítségével lehetőség van az adott érték megváltoztatására:

```
Map.Entry<K, V> map.entrySet().iterator ();
Map.Entry<K,V> entry = iterator.next ();
if (entry != null) entry.setValue (newValue);
```

- Elemek eltávolítása `Collection` view-k segítségével
  - ☞ `remove`, `removeAll`, `retainAll`, clear
  - ☞ `Iterator.remove`
- A `Collection` view-k semmilyen körülmények között sem támogatják elemek hozzáadását. (`keySet` and `values` esetében nincs értelme, `entrySet` esetében pedig felesleges)

Map algebra

- Leképezések részhalmaza (Submap)

```
if (m1.entrySet ().containsAll (m2.entrySet ()) {...}
```

- Két kulcshalmaz egyenlősége:

```
if (m1.keySet ().equals (m2.keySet ()) {...}
```

- Közös kulcsok

```
Set<K> commonKeys = new HashSet<K> (m1.keySet ());
commonKeys.retainAll (m2.keySet ());
```

  - ☞ értékekre hasonlóképp

Map példa

- Vegyünk egy céget. A főnökök (`Map`) olyan bejegyzéseket tartalmaz, amely minden dolgozóhoz hozzárendeli a saját főnökét.
- Kérdezzük le azokat a dolgozókat, akiknek nincs beosztottjuk:

```
Set<Dolgozo> nemFonokok = new HashSet<Dolgozo> (fonokok.keySet ());
nemFonokok .removeAll (fonokok.values ());
```

- Rúgjunk ki minden dolgozót, akinek a Petra nevű hölgy a főnöke:

```
Dolgozo petra = ...;
fonokok.values ().removeAll (Collections.singleton (petra));
```

- Kik azok a dolgozók, akinek most nincs főnökük?

```
Map<Dolgozo, Dolgozo> m = new HashMap<Dolgozo, Dolgozo> (fonokok);
m.values ().removeAll (fonokok.keySet ());
Set<Dolgozo> arvak = m.keySet ();
```

- Először létrehozunk egy ideiglenes Map-et, és eltávolítjuk belőle mindazokat, akik az eredeti fonokok-ben értékek (vagyis főnökök)

Multimap

- Olyan adathalmaz, amikor egy kulcshoz több érték tartozik
- A CF erre vonatkozóan nem tartalmaz interfészeket, de egyszerűen megvalósítható, ha a Map értékek egy-egy listát tartalmazank.
- Példa: Anagramma-kereső
  - ☞ parancssori paraméterként megadjuk, hogy legalább hány elemet tartalmazzon egy csoportnyi anagramma (pl. 1-nek nincs is értelme)
  - ☞ a többi paraméter tartalmazza a szavakat

Multimap példa

```
public class Anagrams {
   public static void main (String[] args) {
      int minGroupSize = Integer.parseInt (args[0]);
      Map<String, List<String>> m =
                                 new HashMap<String, List<String>> ();

      for (int i = 1; i<args.length; i++) {
         String alpha = alphabetize (args[i]);
         List<String> l = m.get (alpha);
         if (l==null) m.put (alpha, l = new ArrayList<String>());
         l.add(args[i]);
      }
      for (List<String> l : m.values ())
         if (l.size () >= minGroupSize)
            System.out.println (l.size() + ": " + l);
   }

   private static String alphabetize (String s) {
      char[] a = s.toCharArray ();
      Arrays.sort (s);
      return new String (a);
   }
}
```

Egyenlőségvizsgálat

- `Collection` leszármazottak esetében
  - ☞ `equals`
- `Map`-ek esetében
  - ☞ `hashCode`
  - ☞ `equals`

`equals` és `hashCode` metódusok

- A Collections Framework metódusai az objektumok összehasonlítására azok equals metódusát használják.
- Az egyes implementációk azonban ezt kiegészíthetik egyéb vizsgálatokkal, pl.:
  - hash kód vizsgálata
  - referencia vizsgálata
- Collections Framework használata esetén, ha az `equals` és a `hashCode` metódusok nincsenek a specifikációknak megfelelően megvalósítva, akkor a különböző referenciájú objektumok nem lesznek egyenlők (még akkor sem, ha egyforma adatokat tartalmaznak).

Rendezés 1.

- A CF-ben két rendezésfogalom létezik, a **természetes rendezés** (natural ordering), és az **összehasonlító osztály** segítségével történő rendezés.
- Természetes rendezés esetén egy objektum saját maga tudja összehasonlítani magát egy másik objektummal. Ehhez az objektumnak meg kell valósítania a `Comparable` interfészt.
- Az összehasonlító osztállyal történő rendezés esetén egy `Comparator` interfészt leszármaztató osztály segítségével történik a rendezés.

Rendezés 2.

- Mind a `Comparator`, mind a `Comparable` esetében egy metódust kell megvalósítanunk, amely két objektumot hasonlít össze.
- Ezek visszatérési értéke negatív, ha az első objektum kisebb a másodiknál, pozitív, ha az első nagyobb, illetve nulla, ha egyenlőek.
- Általában, de nem feltétlenül igaz, hogy:
  - `compare (x,y) == 0 ≡ x.equals (y)`
- Mindig igaz kell legyen, hogy:
  - `sgn (compare (x,y)) == -sgn (compare (y,x))`
  - `(compare (x,y) > 0 && compare (y,z) > 0) → compare (x,z) > 0`
  - `compare (x,y) == 0 → sgn (compare (x,z)) == sgn (compare (y,z))`

Osztály	Rendezés	Osztály	Rendezés
`Byte`	Előjeles szám	`BigInteger`	Előjeles szám
`Character`	Nem előjeles szám	`BigDecimal`	Előjeles szám
`Long`	Előjeles szám	`Boolean`	Boolean.FALSE < Boolean.TRUE
`Integer`	Előjeles szám	`File`	Rendszerfüggő lexikografikus
`Short`	Előjeles szám	`String`	Lexikografikus
`Double`	Előjeles szám	`Date`	Kronológiai sorrend
`Float`	Előjeles szám	`CollectionKey`	Locale-specifikus lexikografikus

Természetes rendezés

```
public class Element implements Comparable<? extends Element> {
   private int i;
   public int compareTo (Element e) { return i - e.i; }
}
```

Összehasonlító osztállyal történő rendezés

```
public class Element {
   private int i;
   public int getI () { return i; }
}

public class Comp extends Comparator<? extends Element> {
   public int compare (Element e1, Element e2) {
      return e1.getI () - e2.getI ();
   }

   public boolean equals (Object o) {
      return false;
   }
}
```

Rendezés 4.

- Természetes rendezés (natural ordering)
  - ☞ static <T extends Comparable<? super T>> void sort(List<T> list)
  - ☞ Az elemeknek implementálniuk kell a `Comparable` interfészt. Ha olyan elemeket tartalmazó listákat akarunk rendezni, amelyek nem implementálják a `Comparable` interfészt, `ClassCastException` dobódik.
- Összehasonlító osztállyal történő rendezés
  - ☞ static <T> void sort(List<T> list, Comparator<? super T> c)
- Osztályok közötti összehasonlítás nem támogatott.

A SortedSet interfész 1.

- A `SortedSet` olyan halmaz, mely az elemeket növekvő sorrenben tárolja (az elemek természetes rendezésének vagy `Comparator` osztály által megadott rendezésnek megfelelően)
- Az `Iterator` a rendezésnek megfelelően sorolja fel az elemeket, a `toArray` pedig rendezett tömböt eredményez
- `Set-et` kiegészítő metódusok
  - `SortedSet<E> subSet (E fromElement, E toElement)`
  - `SortedSet<E> headSet (E toElement)`
  - `SortedSet<E> tailSet (E fromElement)`
  - `E first ()`
  - `E last ()`
  - `Comparator<? super E> comparator ()`

A SortedSet interfész 2.

- A konstruktorok rendezik a paraméterként kapott `Collection`-t.
- A `Comparator`-os konstruktor a paraméterként kapott `Comparator` osztályt használja a rendezéshez.
- Megvalósító osztály: `TreeSet`

A SortedMap interfész

- A természetes rendezéshez a kulcsokat veszi alapul
- Kiegészítő metódusok
  - `Comparator<? super K> comparator ()`
  - `SortedMap<K, V> subMap (K fromKey, K toKey)`
  - `SortedMap<K, V> head (K toKey)`
  - `SortedMap<K, V> tail (K fromKey)`
  - `K firstKey ()`
  - `K lastKey ()`

Általános célú implementációk

Adat-szerkezet	Adatmodell				
	Hash tábla	Tömb	Kegyenlített fa	Láncolt lista	Hashtábla + láncolt lista
Set	Hashet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedDeque	
Map	HashMap		TreeMap		LinkedHashMap

Általános célú Set implementációk

- `HashSet`
  - konstans idejű hozzáférés
  - kapacitással arányos a felsorolása
  - kapacitás, telítettségi faktor (load factor)
- `TreeSet`
  - `SortedSet` implementáció is
  - lassú
- `LinkedHashSet`
  - a kettő között van (hashSet + egy láncolt lista az elemek beszúrási sorrendjét nyilvántartani)
  - beszúrási sorrend megőrzése
  - paraméterek, mint a `HashSet`nél. de a felsorolási idő nem a kapacitással arányos

Speciális Set implementációk

- `EnumSet`
  - Enum típusú elemeket tartalmaz
- `CopyOnWriteArraySet`
  - Ha módosítom, akkor nem a meglevő adatszerkezetbe szúr be, hanem egy teljesen új példányt hoz létre (a régit pedig eldobja)
  - Thread-safe (olcsón)
  - Akkor van értelme, ha a a módosító műveletek nagyon ritkák a lekérdező műveletekhez képest + fontos a szálbiztosság

List implementációk

- `ArrayList`
  - konstans idejű pozícionális hozzáférés (tömb implementáció)
  - `System.arraycopy` olcsó
  - kezdeti kapacitás (initial capacity) jó megválasztása fontos
- `LinkedList`
  - lista elejére való befűzés illetve a lista belsejéből való törlés konstans idejű, de a pozícionális hozzáférés lineáris.
  - A `Queue` interfészt is implementálja
- Speciális megvalósítás
  - `CopyOnWriteArrayList`

Sebességek összehasonlítása

- Mivel az `ArrayList` tömbbel, a `LinkedList` listával van megvalósítva, ezért a hozzáférés sebessége a hozzáférés módjától is függ
  - ☞ `ArrayList` esetében a `get (index)`, `LinkedList` esetében az iterator-os megoldás a gyorsabb.

```
ArrayList a = new ArrayList ();
LinkedList l = new LinkedList ();
...
  for (int i = 0; i<a.size (); i++) {
      Object o = a.get (i);
      ...
  }
  Iterator it = l.iterator ();
  while (it.hasNext ()) {
      Object o = it.next ();
      ...
  }
```

Map implementációk

- `HashMap`
- `TreeMap`
  - `SortedMap` implementáció
- `LinkedHashMap`
  - kulcshozzáférés alapján is rendezhető (a legutolsót a legvégére)
  - `removeEldest`
- Speciális implementációk
  - `EnumMap`
  - `WeakHashMap`
  - `IdentityHashMap`
- Konkurrens implementációk
  - `ConcurrentHashMap`

Queue implementációk

- `LinkedList`
  - FIFO jellegű műveletek
  - `PriorityQueue`
    - kupac adatszerkezeten alapul
- Konkurrens implementációk
  - `LinkedBlockingQueue`
    - opcionálisan méretkorlátozott FIFO blokkoló sor, láncolt lista
  - `ArrayBlockingQueue`
    - méretkorlátozott blokkoló tömb megvalósítás
  - `PriorityBlockingQueue`
    - nem korlátozott blokkoló sor kupac megvalósítás
  - `DelayQueue`
    - idő-alapú ütemezéses sor kupac megvalósítással
  - `SychronousQueue`

Konkurrens hozzáférés 1.

- A CF megvalósításai **nem szinkronizáltak**, tehát egyszerre több szálból is hozzáférhetünk az adatokhoz.
- Ez természetesen veszélyeket rejt magában, kivédésük két módon lehetséges:
  - Az `Iterator` és a `ListIterator` úgynevezett **fail-fast iterátor**ok. Ez azt jelenti, hogy ha az adatszerkezetet az iterátor saját metódusain kívül strukturálisan módosította valami, az iterátor a következő metódushívásra `ConcurrentModificationException`-t dob. Így a hiba rögtön kijön, nem fordulhat elő az, hogy valamikor később inkonzisztens adatokat kapunk.
    Ez a viselkedés azonban nem garantálható minden esetben, tehát nem támaszkodhatunk arra, hogy ennek segítségével detektáljuk a konkurrens hozzáférést.

Konkurrens hozzáférés 2.

- A másik lehetőség, hogy az adatszerkezetet becsomagoljuk egy szinkronizált burkolóba. Ezeket a `Collections` osztály `synchronized` kezdetű metódusaival hozhatjuk létre:

```
...
List syncList = Collections.synchronizedList (new ArrayList ());
...
```

- Nagyon fontos, hogy mivel az iterátorok metódusai nincsenek benne a szinkronizációs mechanizmusban, az iterátoron keresztüli hozzáféréseket manuálisan kell a listára szinkronizálni:

```
...
synchronized (list) {
   Iterator i = list.iterator ();
   while (i.hasNext ()) { ... }
}
...
```

ConcurrentModificationException

- `RuntimeException`
- Jellemző eset: egy szál módosítani akar egy `Collections`-t, miközben egy másik szál iterál rajta.
- Nem kell feltétlenül két külön szálnak lenni:

```
Collection c;
...
Iterator i = c.iterator ();
while (i.hasNext ()) {
   c.remove ( ... );                          // Eltávolítunk egy elemet
   i.next ();                          // ConcurrentModificationException
}
...
```

Konkurrens Wrapper implementációk

- mindegyik thread-safe `Collection`-t ad vissza

```
public static <T> Collection<T>
                          synchronizedCollection (Collection<T> c);

public static <T> Set<T>           synchronizedSet (Set<T> s);

public static <T> List<T>          synchronizedList (List<T> l);

public static <K,V> Map<K,V>       synchronizedMap (Map<K,V> m);

public static <T> SortedSet<T>     synchronizedSortedSet<T> ();

public static <K,V> SortedMap<K,V> synchronizedSortedMap<K,V> m);
```

```
List<T> list = Collections.synchronizedList (new ArrayList<T> ());
```

- szinkronizált `Collection`-t mindig szinkronizált blokkban kell felsorolni

```
Collection<T> c = Collections.synchronizedCollection (myCollection);
synchronized (c) {
   for (T e : c) doSomething(e);
}
```

Kényelmi megoldások

- mini-implementációk
  - ☞ az általános célúnál kényelmesebb, hatékonyabb
- nincs szükség mindenre
- ezek az implementációk statikus factory metódusok segítségével érhetők el (nem publikus konstruktorok)

Üres collectionök

- A Collections osztáy metódusai biztosítják üres halmazok, listák és leképezések előállítását

```
Collections.emptySet ();
Collections.emptyList ();
Collections.emtyMap ();


passenger.declareGoodsForTax (Collections.emptySet ());
```

Egyelemű változtathatatlan lista

- Van olyan, hogy egy elemet valamilyen Collecitonként kell kezelnünk. Erre való a Collections.singleton

```
c.removeAll (Collectioins.singleton (e));
```

- A fenti programkód az adott elem összes előfordulását eltávolítja egy colleciton-ből.

```
job.values ().removeAll (Collections.singleton (LAWYER));
```

Immutable Multiple-Copy List

- Alkalomadtán szükségünk lehet olyan változtathatatlan listára, amely egy adott elemet tartalmaz többször.
- A Collections.nCopies éppen ilyen listát ad vissza.
- Két fő alkalmazása van
  - egy újonnan létrehozott lista elemeinek inicializálása (természetesen nem kell, hogy null érték legyen)

```
List<Type> list =
        new ArrayList<Type> (Collections.nCopies (1000, (Type)null);
```

  - a másik lehet pl. hogy egy String-ekből álló lista végére 50-szer példányban szertnénk hozzáfűzni, hogy "Nem verekszem a padban."

```
buntetes.addAll (Collections.nCopies (50, "Nem verekszem a padban"));
```

  - mivel az addAll metódus indexes paraméterezését használtuk, a lista közepére is fűzhetnénk az új mondatokat.

Tömbök lista nézete

- Arrays.asList metódus
  - ☞ a listában történő változtatások a tömbben is megtörténnek (és viszont)
  - ☞ a lista mérete változatlan
  - ☞ add vagy remove hatására UnsupportedOperationException dobódik
- Célja: a tömbalapú és a lista-alapú API-k közötti átmenet biztosítása
- Másik haszna: fix méretű lista esetén gyorsabb és hatékonyabb, mint egy általános célú List objektum.

```
List<String> list = Arrays.asList (new String[size]);
```

Algoritmusok

- Mindegyik algoritmus a Collections osztályban került megvalósításra, mindegyik statikus és az első argumentum az adott collection, melyen az adott algoritmust el kell végezni.
- Nagyrészt listákon végezhetők ezek az algoritmusok, de van néhány, mely az általánosabb Collection-ön is végezhető.
- Rendezés
- Keverés
- Megszokott adatmanipulációk
- Keresés
- Kompozíció
- Szélsőértékek

Rendezés 1.

- A sort algoritmus egy lista elemeit teszi növekvő sorrendbe. Két formája létezik
  - természetes rendezés
  - összehasonlító osztállyal történő rendezés
- A rendezés gyors és stabil, optimalizált összefésüléses rendezést (merge sort) valósít meg.
  - gyors: garantáltan O(n log n) időben lefut (illetve majdnem rendezett listák esetében gyorsabban). Empirikus tesztek igazolják, hogy akár olyan gyors is lehet, mint egy agyon-optimalizált quick-sort, mely ugyan általában gyorsabb, de nem stabil és nem garantálja az O (n log n) hatékonyságot.
  - stabil: az egyenlő értékű elemek sorrendjét nem változtatja meg. Ez fontos lehet, ha egymás után ugyanazt a listát különböző attribútumok alapján rendezzük. (A szerint, azon belül B szerint)

Rendezés 2.

```
public class Sort {
   public static void main (String[] args) {
      List<String> list = Arrays.asList (args);
      Collections.sort (list);
      System.out.println (list);
   }
}
```

```
List<List<String>> winners = new ArrayList<List<String>> ();
for (List<String> l : m.values ())
   if (l.size() >= minGroupSize) winners.add (l);

Collections.sort (winners, new Comparator<List<String>> () {
   public int compare (List<String> o1, List<String> o2) {
      return o2.size () - o1.size ());
   }});

for (List<String> l : winners) {
   System.out.println (l.size () + ": " + l);
}
```

Keverés 1.

- A shuffle algoritmus gyakorlatilag a sort ellentettje:
  - ☞ az esetlegesen meglévő kis rendezettséget is eltünteti
- Az algoritmus újrarendezi a listát, minden lehetséges permutációt egyenlő eséllyel állít elő (fair).
- Az algoritmus felhasználása pl. kártyaleosztásnál.
- Két formája van
  - Paraméterként kap egy listát
  - Paraméterként kap egy listát és egy véletlenforrást (a Random osztály egy példányát várja véletlenforrásként)

Keverés 2.

```
public static void shuffle (List<?> list, Rangom rnd) {
   for (int i = list.size (); i > 1; i--)
      swap (list, i-1, rnd.nextInt (i));
}
```

```
public class Shuffle {
   public static void main (String[] args) {
      List<String> list = new ArrayList<String> ();
      for (String a : args) lists.add (a);
      Collections.shuffle (list, new Random ());
      System.out.println (list);
   }
}
```

```
public class Shuffle {
   public static void main (String[] args) {
      List<String> list = Collectioins.asList (args);
      Collections.shuffle (list);
      System.out.println (list);
   }
}
```

Adatmanipulációk

- A Collections osztály 5 adatmanipulációs algoritmust biztosít.
- Ezeket a műveleteket listákon végezhetjük.
  - addAll – minden elemet az adott collectionhöz adja. A hozzáadandó elemek lehetnek valamilyen collectionben, illetve egyenként definiáltak is (varargs).
  - copy – két argumentuma van, egy cél lista és egy forráslista. Az algoritmus veszi a forráslista elemeit és felülírja velük a céllista elemeit. A céllista legalább olyan hosszú, mint a forráslista, ha hosszabb, a fennmaradó elemek változatlanok maradnak.
  - fill – a listában minden elemet felülír a paraméterként kapott értékkel. (pl. újrainicializáláshoz)
  - swap – kicserél két adott pozíción lévő elemet
  - reverse – megfordítja egy lista elemeinek a sorrendjét (ListIterator-ral is elérhető ugyanez)

Keresés

- A binarySearch algoritmus rendezett listában keres.
- Az algoritmusnak két formája van
  - ☞ az egyik paraméter egy lista, amelyben keresni kell és egy elem, amit meg kell keresni (természetes rendezés)
  - ☞ további paraméter egy Comparator interfészt megvalósító osztály, továbbá feltételezi, hogy a lista növekvő sorba rendezett

```
int pos = Collections.binarySearch (list, key);
if (pos < 0) l.add (-pos-1);
```

Kompozíció

- A frequency és disjoint algoritmusok egy vagy több collection kompozícióját tesztelik.
- frequency – megszámolja, hogy egy edott elem hányszor szerepel a collectionben.
- disjoint – megmondja, hogy két collection diszjunkt-e vagy sem

Szélsőértékek keresése

- A min és max algoritmusok egy collection legkisebb illetve legnagyobb elemét adja vissza.
  - az egyik algoritmus az elemek természetes rendezését veszi alapul
  - a másik algoritmus összehasonlító osztályt vár

Interoperabilitás – kompatibilitás 1.

- Collection interfészek
- Korábbi Java típusok
  - Vector
  - Hashtable
  - tömb
  - Enumeration
- Ha a régi API tömböt adott vissza, az új API viszont Collection kér.

```
T[] result = oldMethod (arg);
newMethod (Arrays.asList (result));
```

Interoperabilitás – kompatibilitás 2.

- Ha a régi API Vector-ral vagy Hashtable-lel tér vissza, az minden további nélkül adható tovább:

```
Vector result = oldMethod (arg);
newMethod (result);

Hashtable tResult = oldMethod (arg);
newMethod (tResult);
```

- A legritkábban egy régebbi API Enumeration-t ad. A Collections.list metódus az Enumeration-t Colleciton-né transzformálja:

```
Enumeration e = oldMethod (arg);
newMethod (Collections.list(e));
```

Visszafelé való kompatibilitás 1.

- Ez ekkor kerül előtérbe, ha az új API által visszaadott értéket egy régi API által definiált metódusnak kell átadni.
- Ha az új API egy Collection-t ad vissza, a régi pedig objektumtömböt vár:

```
Collection c = newMethod ();
oldMethod (c.toArray ());

Collection<String> c = newMethod ();
oldMethod ((String [])c.toArray (new String[0]));
```

- Ha a régi API Vector-t vár

```
Collection c = newMethod;
oldMethod (new Vector (c));
```

Visszafelé való kompatibilitás 2.

- Ha a régi API Hashtable-t vár

```
Map m = newMethod ();
oldMethod (new Hashtable (m));
```

- Ha a régi API Enumeration objektumot vár. Ez nem általános eset, de azért sokszor előfordulhat. Erre való a Collectons.enumeration metódus. Ez a statikus factory metódus veszi a collection-t és egy olyan Enumerator-ta ad vissza, mely képes felsorolni a Collection elemeit.

```
Collection c = newMethod ();
oldMethod (Collections.enumeration (c));
```