

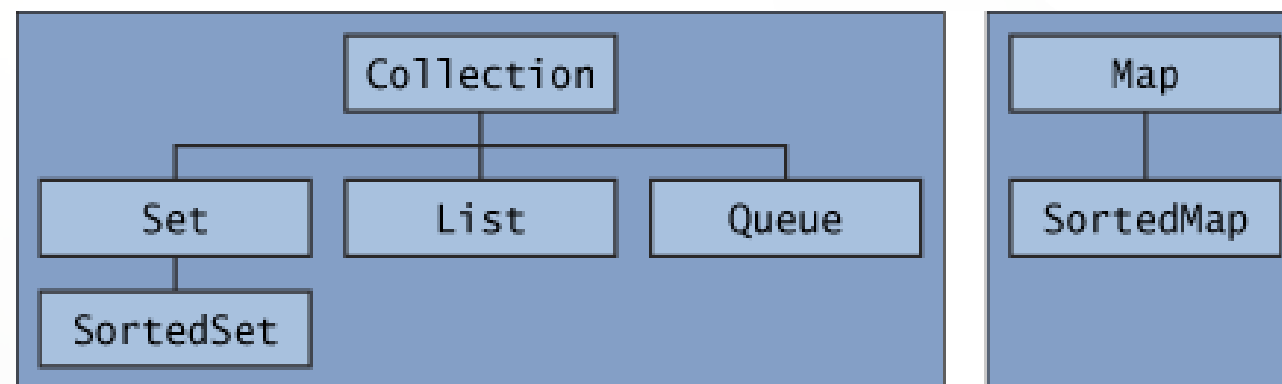
Java Collections Framework (JCF)

Gyűjtemény keretrendszer

Simon Károly
simon.karoly@codespring.ro

- **Collection** (gyűjtemény/kollekció/tároló/konténer): objektum, amely más objektumok egy „természetes” csoportosulásának (natural grouping) egy egységben történő (összefoglaló jellegű) tárolását szolgálja. Lehetőséget biztosít a tárolt elemek keresésére/ kinyerésére és manipulálására.
- **JCF**: a gyűjtemények használatát segítő keretrendszer.
- A JCF részei:
 - **Interfészek**: absztrakt adattípusok, melyek gyűjtemények megvalósítástól (implementációtól) független ábrázolását szolgálják.
 - **Implementációk**: a gyűjtemény interfészek konkrét megvalósításai.
 - **Algoritmusok**: gyűjteményekkel kapcsolatos hasznos műveletek (keresés, rendezés stb.). Ezeket a műveleteket az interfészeket implementáló objektumokon végezhetjük (egy adott metódus egy gyűjtemény interfész különböző implementációi esetében egyaránt érvényes).

- Az alapvető gyűjteményeknek megfelelő interfészek hierarchiája (a JCF alapja):



- A Collection interfész deklarációja:
`public interface Collection<E>...`
- Minden interfész generikus típus paraméterrel rendelkezik (lásd Java generics). A Collection példányok deklarációjánál meg lehet (és tanácsos) megadni a konkrét típust. Ez lehetővé teszi a fordítási idejű ellenőrzést, segít kiszűrni a hibalehetőségeket és a kinyerésnél megkímél a manuális átalakításoktól (casting).

- **Collection:** a gyűjtemény hierarchia gyökere, a legáltalánosabb interfész, akkor használjuk ha maximális általánosságra törekszünk (pl. kollekciók átadása paraméterként), nincs direkt implementációja a JCF-ben.
- **Set:** halmaz (Collection, amelyben nincsenek duplikált elemek).
- **List:** rendezett gyűjtemény, amelyben a pozíció (az index egy pozitív egész) segítségével hivatkozhatunk az elemekre.
- **Queue:** sor, tipikusan feldolgozásra váró elemek tárolása (speciális műveletekkel), legtöbbször FIFO elv szerint rendezett elemekkel (nem feltétlenül, kivétel pl. a prioritási sor), ahol az először eltávolítandó elem a sor elején van (a FIFO esetében az új elem a végére lesz beszúrva, de más rendezés is alkalmazható).
- **Map:** kulcs-érték párok tárolása (ahol a kulcs egyedi).
- **SortedSet:** az elemeket növekvő sorrendben tárolja.
- **SortedMap:** az elemeket a kulcsok növekvő sorrendjében tárolja.

- ```
public interface Collection<E> extends Iterable<E> {
 // Basic operations
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(E element); //optional
 boolean remove(Object element); //optional
 Iterator<E> iterator();
 // Bulk operations (több elemet érintő metódusok)
 boolean containsAll(Collection<?> c);
 boolean addAll(Collection<? extends E> c); //optional
 boolean removeAll(Collection<?> c); //optional
 boolean retainAll(Collection<?> c); //optional
 void clear(); //optional
 // Array operations
 Object[] toArray();
 <T> T[] toArray(T[] a);
}
```

- Az add és remove metódusok visszatérítési értéke true, ha volt változás a gyűjteményben (pl. halmaz, vagy nem létező elem esetén nem biztos).
- A konkrét implementációk esetében egyes metódusok kivételeket dobhatnak, pl.:
  - UnsupportedOperationException – az opcionális metódusok esetében (erre vonatkozik az optional megjegyzés) megtörténhet, hogy egy adott implementáció nem ad lehetőséget az illető műveletre.
  - NullPointerException – pl. ha az illető implementáció nem ad lehetőséget null elem beillesztésére.
  - IllegalArgumentException – pl. az illető elem valamilyen tulajdonsága nem teszi lehetővé a művelet végrehajtását (az illető implementáció szerint).
  - IllegalStateException – az illető művelet nem engedélyezett a gyűjtemény aktuális állapotában (az illető implementációnak megfelelően).
  - ClassCastException – az elem típusa (osztálya) miatt nem történhet meg a művelet végrehajtása.
  - ArrayStoreException – nem történhet meg a tömbbe konvertálás (típus inkompatibilitás miatt).



- For-each ciklus:

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

- Iterátorok:

```
public interface Iterator<E> {
 boolean hasNext();
 E next();
 void remove(); //optional
}
```

- Iterátort érdemes használni, ha törölni is akarunk (de ez csak akkor biztonságos, ha a bejárással egy időben más törlés nem történik), pl. szűrésnél (ahol a for-each nem használható, mert elrejtí az iterátort, nem ad lehetőséget törlésre), vagy több kollekció párhuzamos bejárásánál.
- Példa:

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

- ```
public interface Set<E> extends Collection<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element); //optional  
    boolean remove(Object element); //optional  
    Iterator<E> iterator();  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); //optional  
    boolean removeAll(Collection<?> c); //optional  
    boolean retainAll(Collection<?> c); //optional  
    void clear(); //optional  
    // Array Operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```


- `Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);`
- `Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);`
- `Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);`
- ```
import java.util.*;
public class FindDups {
 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);
 }
}
```

- ```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element); //optional  
    boolean add(E element); //optional  
    void add(int index, E element); //optional  
    E remove(int index); //optional  
    boolean addAll(int index,  
                   Collection<? extends E> c); //optional  
  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    // Iteration  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
    // Range-view  
    List<E> subList(int from, int to);  
}
```

- ```
public interface ListIterator<E> extends Iterator<E> {
 boolean hasNext();
 E next();
 boolean hasPrevious();
 E previous();
 int nextIndex();
 int previousIndex();
 void remove(); //optional
 void set(E e); //optional
 void add(E e); //optional
}
```
- Példa:  

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

- ```
public interface Map<K,V> {  
    // Basic operations  
    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();  
    // Bulk operations  
    void putAll(Map<? extends K, ? extends V> m);  
    void clear();  
    // Collection Views  
    public Set<K> keySet();  
    public Collection<V> values();  
    public Set<Map.Entry<K,V>> entrySet();  
    // Interface for entrySet elements  
    public interface Entry {  
        K getKey();  
        V getValue();  
        V setValue(V value);  
    }  
}
```

- ```
public interface Queue<E> extends Collection<E> {
 E element();
 boolean offer(E e);
 E peek();
 E poll();
 E remove();
}
```
- Bounded queue: az elemek száma (kapacitás) korlátozott.
- add/remove/element – kivételeket generálnak  
(add – IllegalStateException,  
remove/element – NoSuchElementException)
- offer/poll/peek – az offer false-ot fordít vissza ha nem lehetséges (kapacitás), a poll/peek null értéket adnak vissza.

- Az `ArrayList` elemei, amennyiben implementálják a `Comparable` interfészt, rendezhetőek:

```
Collections.sort(l);
```

- Az elemek rendezhetőek, ha implementálják a `Comparable` interfészt:

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

- Visszafordított érték: negatív egész, ha az `o` kisebb; 0, ha egyenlőek; pozitív egész, ha nagyobb.

- Példa:

```
import java.util.*;  
public class Name implements Comparable<Name> {  
    private final String firstName, lastName;  
    ...  
    public int compareTo(Name n) {  
        int lastCmp = lastName.compareTo(n.lastName);  
        return (lastCmp!=0 ? lastCmp :  
                firstName.compareTo(n.firstName));  
    }  
}
```


- Ha nem az alapértelmezett („természetes”) módon akarjuk rendezni az elemeket, vagy a Comparable interfészt nem implementáló objektumokat akarunk rendezni.
- ```
public interface Comparator<T> {
 int compare(T o1, T o2);
}
```
- Példa: az alkalmazottakat (Employee) általában név szerint rendezzük (így van implementálva a compareTo metódus), de adott esetben szükségünk lehet régiség szerinti rendezésre.

```
...static final Comparator<Employee> SENIORITY_ORDER =
 new Comparator<Employee>() {
 public int compare(Employee e1, Employee e2) {
 return e2.hireDate().compareTo(e1.hireDate());
 }
 };
```

```
...
List<Employee>e = new ArrayList<Employee>(employees);
Collections.sort(e, SENIORITY_ORDER);
System.out.println(e);
```

# A SortedSet és SortedMap interfészek

- ```
public interface SortedSet<E> extends Set<E> {  
    // Range-view  
    SortedSet<E> subSet(E fromElement, E toElement);  
    SortedSet<E> headSet(E toElement);  
    SortedSet<E> tailSet(E fromElement);  
    // Endpoints  
    E first();  
    E last();  
    // Comparator access (returns null for natural order)  
    Comparator<? super E> comparator();  
}
```
- ```
public interface SortedMap<K, V> extends Map<K, V>{ Comparator<?
 super K> comparator();
 SortedMap<K, V> subMap(K fromKey, K toKey);
 SortedMap<K, V> headMap(K toKey);
 SortedMap<K, V> tailMap(K fromKey);
 K firstKey();
 K lastKey();
}
```

- Implementáció típusok:
  - Általános célú
  - Speciális – speciális esetekben alkalmazzuk (megkötések, spec. viselkedés)
  - Konkurens – több végrehajtási szál esetére tervezve (egy szál esetén a teljesítmény rovására), a `java.util.concurrent` csomag részei
  - Csomagoló (wrapper) – a többi implementációval (ált. az általános célúval) együtt használjuk (extra funkcionálisok, korlátozások)
  - Kényelmi – tipikusan gyártó metódusokon keresztül érhetőek el, az általános célúak alternatívájaként, speciális esetekben (pl. singleton halmaz)
  - Absztrakt – vázlatos implementációk, az egyéni implementációk támogatására

- List
  - ArrayList – változó méretű tömböt használ.
  - LinkedList – duplán láncolt listát használ.
- Map
  - HashMap – hasító tábla, változó méretű tömbök segítségével implementálva.
  - LinkedHashMap – a HashMap leszármazottja, duplán láncolt listát használ (a bejárás a beillesztés sorrendjében történik).
- Set
  - HashSet – hasító tábla alapú (konkrétan HashMap példányt használ).
  - LinkedHashSet – a HashSet-ből származtatott, hasító táblát (HashMap példányt) és duplán láncolt listát (LinkedList példányt), a bejárás a beillesztés sorrendjében történik (ellentétben a HashSet –tel, ahol a sorrendre nincs garancia).
- SortedMap
  - TreeMap – rendezett, piros fekete fán alapul.
- SortedSet
  - TreeSet – TreeMap alapú.
- Queue
  - LinkedList – FIFO.
  - PriorityQueue – érték szerinti rendezés.

- Az interfészekben deklarált összes metódust implementálják.
- Megengedik a null elemeket.
- Egyik sem szinkronizált (thread-safe) (szakítás a múlttal), de mindegyik átalakítható szinkronizált gyűjteménybe a megfelelő wrapper implementációk alkalmazásával. A `java.util.concurrent` package további interfészeket és implementációkat biztosít.
- Mindegyik serializálható és mindegyik biztosít clone metódust.
- **Interfészekben gondolkodjunk!** (A konkrét implementáció általában csak a teljesítmény szempontjából számít, kiválasztása a körülmények függvényében történik)

- A HashSet gyors, a TreeSet rendezett, a LinkedHashSet bizonyos értelemben átmenetet képez (a Map implementációk esetében hasonló a helyzet).
- Az ArrayList általában gyorsabb (hozzáférés az elemekhez konstans időben), de sok iterálás/törlés/lista elejére történő beillesztés esetén előnyösebb lehet a LinkedList (konstans időben történnek, míg az ArrayList esetében lineáris). A LinkedList a Queue interfészt is implementálja.
- Általában: változó méretű tömb használata esetén (pl. ArrayList): ha nem (jól) határozzuk meg (nem tudjuk) a kapacitást (tárolandó elemek számát) gondot jelenthet. Ha túl nagy → fölöslegesen nő a komplexitás (bizonyos műveletek lineáris idejűek), ha túl kicsi → a növelés másolással jár (és ettől nő a komplexitás) (megjegyzés: System.arraycopy() (native) használata).
- ```
public void ensureCapacity(int minCapacity) {  
    modCount++;  
    int oldCapacity = elementData.length;  
    if (minCapacity > oldCapacity) {  
        Object oldData[] = elementData;  
        int newCapacity = (oldCapacity * 3) / 2 + 1;  
        if (newCapacity < minCapacity)  
            newCapacity = minCapacity;  
        elementData = Arrays.copyOf(elementData, newCapacity);  
    }  
}
```


- Wrapper: a feldolgozást továbbítja (egy adott gyűjteményhez), de további funkciókat biztosít (Decorator minta).
- Szinkronizálás, módosítások letiltása (Unmodifiable Wrappers), típusellenőrző wrapperek (Check Interface Wrappers, a statikus ellenőrzés kikerülésének (pl. legacy code alkalmazásának esetén) megelőzése) .

- Szinkronizálás:

- A Collections osztály statikus metódusai (Factory minta):

```
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> list);  
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);  
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);  
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

- `List<Type> list =`

```
    Collections.synchronizedList(new ArrayList<Type>());
```

- Az iterálás szinkronizálását manuálisan kell megoldani (a több léptetés műveletet egyetlen (atomi) műveletként kell szinkronizálni):

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

- Tömb (array) lista (list) nézete:

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

- Több másolatot tartalmazó lista:

```
List<Type> list =  
    new ArrayList<Type>(Collections.nCopies(1000, (Type)null);  
(az eredmény egy lista 1000 null elemmel)
```

- Egyetlen meghatározott elemből álló lista:

```
job.values().removeAll(Collections.singleton(LAWYER));  
(egy elem minden előfordulását törölni akarjuk egy listából)
```

- Üres gyűjtemények

```
tourist.declarePurchases(Collections.emptySet());  
(a metódus gyűjteményt vár, de nincsen amit átadni)
```

- Set: EnumSet (az elemek (azonos) enum típusúak, egy bit sorozatban tárolhatóak), CopyOnWriteArraySet (a változásokat előidéző műveletknél másolat készül, nincs szükség blokkolásra).
- List: CopyOnWriteArrayList.
- Map: EnumMap, WeakHashMap (gyenge kulcs referenciák, a kulcs-elem párok felszabadíthatóak, ha a kulcsra kívülről nincs referencia), IdentityHashMap.

Szinkronizált implementációk (java.util.concurrent)

- Map → ConcurrentMap (interface) → ConcurrentHashMap
- Queue → BlockingQueue (interface) → LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, DelayQueue, SynchronousQueue

- A Collections osztály statikus metódusai
 - `public static void shuffle(List l)`
 - `public static void sort(List l) //merge sort`
 - `public static void sort(List l, Comparator c)`
 - `public static void swap(List l, int i, int j)`
 - `public static void rotate(List l, int distance)`
 - `public static void reverse(List l)`
 - `public static Object min(Collection c)`
 - `public static int binarySearch(List l, Object o)`
 - `public static int binarySearch(List l, Object o, Comparator c)`
 - `public static void copy(List dest, List src)`
 - `public static void fill(List l, Object o)`

```
private static class MyArrayList<T>
    extends AbstractList<T> {

    private final T[] a;

    MyArrayList(T[] array) {
        a = array;
    }
    public T get(int index) {
        return a[index];
    }
    public T set(int index, T element) {
        T oldValue = a[index];
        a[index] = element;
        return oldValue;
    }
    public int size() {
        return a.length;
    }
}
```

- **AbstractCollection**
- **AbstractSet**
- **AbstractList**
- **AbstractSequentialList**
- **AbstractQueue**
- **AbstractMap**

- Vector, Hashtable: az előző Java verzióknak is részei voltak (ezek a verziók még nem tartalmazták a JCF -ot), a JCF bevezetése után „beleillesztették” őket a keretrendszerbe.
- Tulajdonképpen az ArrayList és HashMap szinkronizált változatainak tekinthetők (de: nem engedik meg a null értékeket, és bár az iterátorok biztonságosak (fail-safe) (iterálás közbeni változtatások), a metódusok által visszatérített Enumeration-ok nem azok).
- Legacy code (kompatibilitással kapcsolatos meggondolások, generics):

```
class LegacyCode{  
  
    public static List.getItems(){  
        List list = new ArrayList();  
        list.add(new Integer(1));  
        list.add("ketto");  
        return list;  
    }  
}
```

```
class NaiveClient{  
  
    public void processItems(){  
        List<Integer> list =  
            LegacyCode.getItems();  
        int s =0;  
        for (int i:list) s+=i;  
    }  
}
```