Collections

Collections

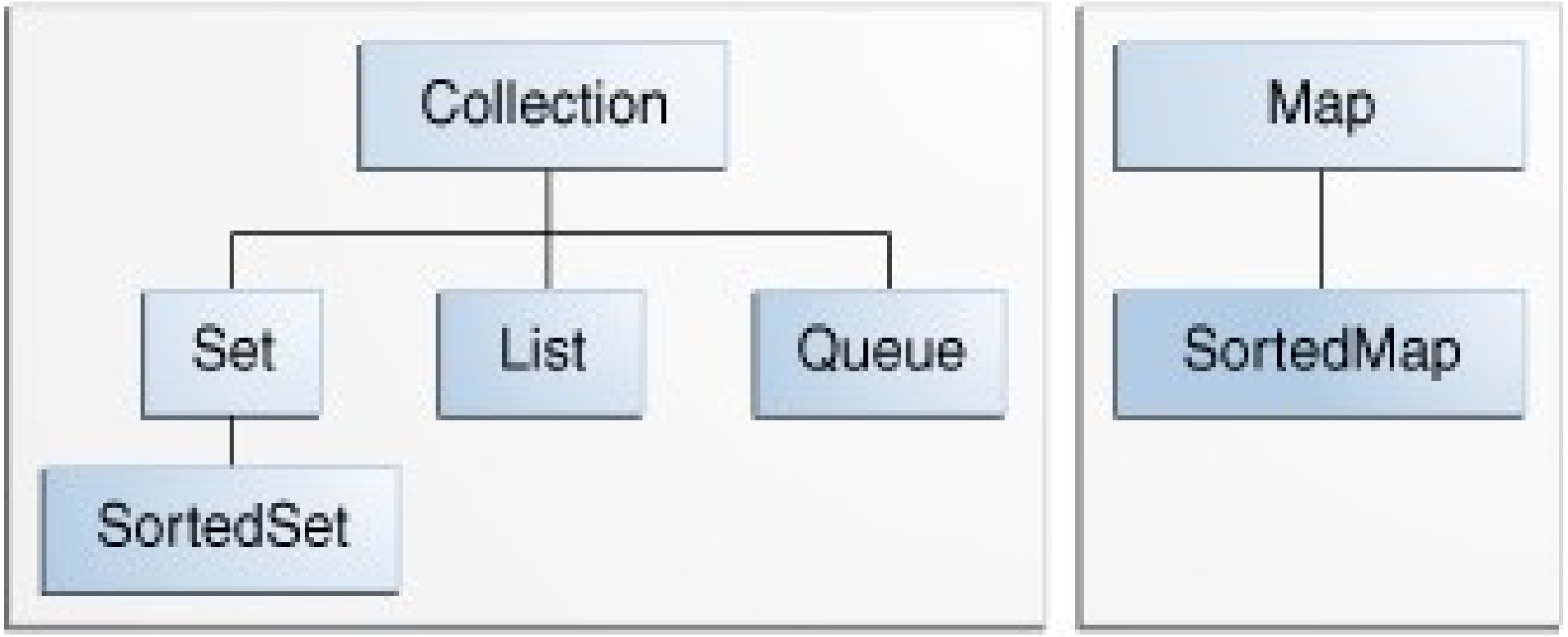
Collections

- Les collections són simplement objectes que agrupen diferents elements en una única unitat. També són anomenades *contenidors*.
- Les collections s'utilitzen per emmagatzemar, obtenir, manipular i transmetre dades agregades.
- Representen dades agregades del món físic com poden ser un llistat telefònic, un safata de correu electrònic o un conjunt de cartes.

Collections Framework

- El bastiment de collections està compost per:
  - **Interfaces**: són tipus de dades abstractes que representen les collections, i permeten manipular-les independentment de quina sigui la seva implementació.
  - **Implementations**: són implementacions concretes de les diferents interfícies. Essencialment, són estructures de dades reutilitzables.
  - **Algoritmes**: usats en les implementacions per dur a terme càlculs computacionals útils com ara l'ordenació d'elements d'una collection, la cerca d'un element, etc. Els algoritmes són polimòrfics (poden usar-se en implementacions de diferents tipus de collections). Són reutilitzables.

Collection Interfaces



Collection Interfaces

- Un Map no és una Collection real.
- Totes les collections interfaces són genèriques:

  `public interface Collection<E>....`
- La sintaxi <E> ens indica que la interfície és genèrica.
- Però quan el programador declari una Collections, hauria d'especificar el tipus d'objecte que la collection contindrà.
- Les dades contingudes en una Collection es coneixen com els seus *elements*.

Collection Interfaces

- **Collection**: l'arrel de la jerarquia d'interfícies. No té una implementació concreta. Només hi ha implementacions de les interfícies que en deriven.
- **Set**: una collection (no ordenada) que no admet elements duplicats. Modela el concepte matemàtic de conjunt i s'usa per a representar conjunts d'elements (sense repetits) com ara les matèries que fa un estudiant o els processos que corren a un sistema.

Collection Interfaces

- **List**: és una *ordered collection or sequence.* Els elements es troben ordenats. Pot contenir elements repetits o duplicats.

  Una List dóna control sobre on està ubicat cada element, on s'inserta i es pot accedir als elements mitjançant un índex enter.
- **Queue**: una collection que manté múltiples elements en llista d'espera per a ser processats (FIFO, LIFO...)

Collection Interfaces

- **Map**: una collection que mapa claus amb valors (keys to values). No pot contenir claus duplicades i cada clau mapa com a màxim a un únic valor.
- **SortedSet**: un Set ordenat i que manté els seus elements en un ordre ascendent. Té mètodes addicionals respecte del Set, que permeten aprofitar la ordenació dels elements. S'usen per conjunts de dades que de forma natural estan ordenades, com ara una llista de paraules (ordenades per ordre alfabètic).

Collection Interfaces

- **Sorted Map**: és un Map que manté els seus mapatges segons un ordre ascendent en les seves claus. Similar al SortedSet. S'usen en dades ordenades de parelles clau/valor, com ara diccionaris o llistats telefònics.

The Collection Interface

- **Conversion Constructor**: Totes les interfícies de tipus Collection tenen un constructor que pren com a argument una Collection.
- Aquest constructor inicialitza la nova Collection amb tots els elements de la collection que li passem com a argument, sigui quin sigui el subtipus i la implementació.
- En altres paraules, ens permet la **conversió** d'un tipus de collection a un altre:

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

on c és una collection qualsevol.

The Collection Interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();
```

The Collection Interface

```
    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The Collection Interface

- Els mètodes Add i Remove retornen True quan després de ser cridats la Collection s'ha modificat, i False en cas contrari.
- **Recorrent una Collection**:

    1 - Usant el for-each

    2 - Usant un Iterator

Recorrent una Collection amb for-each

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

Recorrent una Collection amb un Iterator

La interfície Iterator és:

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

Recorrent una Collection amb un Iterator

- HasNext retorna true si la iteració té més elements, i el mètode next retorna el proper element de la iteració.
- El mètode remove elimina el darrer element retornat per next. Només es pot cridar una sola vegada per cada vegada que es crida next. I només és segur usar-lo dins d'una iteració.
- Quan vulguem eliminar elements usarem un Iterator (amb el mètode remove) i mai un for-each.
- Usarem un iterator (i no un for-each) quan vulguem recórrer diferents collections en paral·lel.

Recorrent una Collection amb un Iterator

- Exemple:

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

És un mètode polimòrfic, que vol dir que funciona amb qualsevol tipus de Collection.

Collection Interface Bulk Operations

- Els mètodes Bulk permeten operar sobre la collection en el seu conjunt. Per exemple:

  `containsAll` : retorna true si la collection de destí conté tots els elements de la collection passada com a argument.

  `clear`: elimina tots els elements de la collection.
- Consulteu la API per a la resta d'operacions.

Collection Interface Bulk Operations

- Exemple, donada una Collection c:

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

  on Collections.singleton() és un mètode estàtic que retorna un Set immutable que conté només l'element especificat.

  Aquest exemple eliminaria totes les aparicions de l'element e que contingués la collection c.

Collection Interface Array Operations

- La Collection Interface proveeix del mètode `toArray` per poder treballar amb APIs antigues que esperen un array enlloc d'una collection.
- Podrem transformar els continguts d'una collection a array.
- La manera més simple, sense arguments, crea un array d'`Objects`.
- La manera més complexe permet fer la crida proveint un array o escollir el tipus d'array en temps d'execució.

Collection Interface Array Operations

- Exemple, donada una Collection c. Volca el contingut de c a un array de tipus Object la mida del qual és igual al nombre d'elements de c:

```
Object[] a = c.toArray();
```

Collection Interface Array Operations

- Exemple, donada una Collection c. Volca el contingut de c a un array de tipus Object la mida del qual és igual al nombre d'elements de c:

```
Object[] a = c.toArray();
```

Collection Interface Array Operations

- Ara suposa que sabem d'antuvi que la Collection c només conté strings (potser perquè c és de tipus Collection<String>). El següent exemple volca el contingut de c a un array nou d'strings la mida del qual és idèntica al nombre d'elements de c:

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

The Set Interface

- Un Set és una Collection que **no pot contenir elements duplicats**.
- Un Set <u>només</u> té mètodes heredats de Collection, afegint-hi la restricció que els elements duplicats estan prohibits.
- Dues instàncies (objectes) de Set són iguals si contenen els mateixos elements. Les comparacions es poden fer sempre encara que les implementacions de la interfície Set siguin diferents (és a dir, es poden fer comparacions entre dos subtipus de Set).

The Set Interface

```
public interface Set<E> extends
Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();
```

The Set Interface

```
    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E>
c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    void clear(); // optional
    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);   }
```

Set Implementations

- Les tres implementacions de Set més generals i que cal conèixer són HashSet, TreeSet i LinkedHashSet.
- **HashSet**: emmagatzema les dades a una estructura de dade de tipus Hash Table (també anomenades taules de dispersió) *(nota: no és objectiu d'aquest curs aprendre les diferents estructures de dades subjacents a les diferents implementacions de les collections, però si voleu ampliar coneixements podeu buscar a Internet pels conceptes Hash Table o taula de dispersió).*

Set Implementations

- La implementació HashSet és la que ofereix major rendiment (menor cost computacional de l'algorisme o estructura de dades subjacent); emperò, no ofereix cap garantia sobre en quin ordre es farà cada iteració.
- **TreeSet**: emmagatzema els elements en una estructura de dades de tipus Tree (arbre), que és significativament més lenta que la Hash Table. És una collection que preserva l'ordre.

Set Implementations

- **LinkedHashSet**: implementació basada en una estructura de dades que consisteix en una llista enllaçada (o encadenada) a una taula de dispersió, de manera que s'ordenen els elements segons l'ordre en el qual van ser introduits al Set. El seu cost és només un xic superior al del HashSet.

Set Implementations

- Exemple, donada una Collection c, volem crear una altra Collection amb els mateixos elements però amb tots els duplicats (elements repetits) eliminats:

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

Set Implementations

- El següent exemple fa el mateix, però amb la diferència que preserva l'ordre dels elements en la Collection c original:

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

Set Implementations

- Exemple amb un mètode genèric:

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

Set Interface Basic Operations

1) Operacions bàsiques, que són les pròpies de les Collections (o contenidors): *add*, *remove*, *contains*, *size*, *isEmpty* i *iterator*.

2) Operacions massives, que bàsicament estenen les anteriors sobre grups d'elements: *addAll*, *removeAll*, *containsAll* (rep contenidors com a paràmetres), *equals* (compara el conjunt amb un objecte) i *clear* (per a destruir el contingut del conjunt).

3) Operacions sobre vectors (arrays) per a transferir el contingut d'un conjunt a un vector: *toArray*.

Set Interface – Example FindDups

- L'exemple FindDups (veure exemples adjunts) imprimeix les paraules duplicades detectades a una llista de paraules que hem passat com a argument. També imprimeix el nombre total de paraules diferents i una llista de totes les paraules un cop eliminats els duplicats.

Set Interface – Example FindDups

- **Important**:

  Noteu com al codi d'aquest exemple, es refereix a la Collections segons el tipus de la seva interfície, que és Set en aquest cas, enlloc del tipus de la seva implementació (HashSet):

  ```
  Set<String> s = new HashSet<String>();
  ```

- Aquesta pràctica de programació **és molt recomanable**, ja que ens dóna la flexibilitat de canviar implementacions simplement canviant el constructor. Així evitem haver de canviar el tipus de totes les variables i paràmetres si es vol canviar d'implementació.

Set Interface – Example FindDups

- Així, si a l'exemple FindDups volem imprimir ara tots els elements de forma ordenada (alfabèticament) ho podrem aconseguir canviant la implementació escollida, i això es farà simplement posant ara TreeSet (per exemple) enlloc de HashSet:

```
Set<String> s = new TreeSet<String>();
```

The List Interface

- Una **List** és un contenidor (o Collection) ordenat. A vegades se l'anomena, també, seqüència.
- A banda dels mètodes heredats de la interfície Collection, la interfície List proveeix mètodes per a:

Accés posicional: manipula elements en base a la seva posició a la llista.

Cerca: cerca un element en una llista i retorna quina és la seva posició.

Iterator: estén l'Iterator de Collection, afegint-li propietats específiques d'una llista.

Operacions de rang: afecten a un rang d'elements.

The List Interface

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    // optional
    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);
```

The List Interface

```
// 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);
}
```

The List Interface

- En el cas de les implementacions de List, el mètode `remove` sempre elimina de la llista **la primera ocorrència** de l'element especificat.
- Els mètodes `add` i `addAll` sempre afegeixen nous alements **al final de la llista**.
- En el següent exemple veiem la concatenació de dues llistes:

```
list1.addAll(list2);
```

The List Interface

- Si volem concatenar dues llistes sense perdre una referència a la original, podem fer el següent:

```
List<Type> list3 = new
ArrayList<Type>(list1);
list3.addAll(list2);
```

The List Interface

- Igual que amb els Set, dues llistes es poden comparar independentment de quina sigui la seva implementació.
- Així, podrem comparar, per exemple, un ArrayList amb un LinkedList.
- Els mètodes que s'usen per comparar collections són `equals` i `hashCode`.

Exemples amb operacions d'accés posicional i de cerca

- Per a intercanciar dos valors indexats a una List:

```
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);
}
```

Exemples amb operacions d'accés posicional i de cerca

- El següent mètode es podem trobar a la classe estàtica Collections. Realitza permutacions dels elements de la llista a l'atzar:

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

Exemples amb operacions d'accés posicional i de cerca

- Aquests dos últims exemples són polimorfics, que vol dir que es poden usar amb qualsevol List, independentment de quina sigui la seva implementació.
- L'exemple NB adjunt anomenat Shuffle fa ús d'aquest darrer mètode per reordenar a l'atzar les paraules que es passin com a argument en l'execució de l'exemple (veure exemple).

Exemples amb operacions d'accés posicional i de cerca

- L'exemple NB anomenat Shuffle2 fa el mateix que Shuffle, però el codi és més curt perquè aprofitem el mètode asList, de la classe estàtica Arrays.
- El mètode Arrays.asList permet un array ser vist com una List.
- Veure exemple Shuffle2 i comparar amb Shuffle.

ListIterator Interface

- La interfície ListIterator proveeix dels mètodes d'Iterator (hasNext, next i remove) i addicionalment, com que les llistes són ordenades, proveeix d'altres mètodes que treuen profit de la ordenació dels elements.
- Per exemple, tenim els mètodes hasPrevious i previous, que són equivalents a hasNext i next.

ListIterator Interface

```
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    }
```

ListIterator Interface

- Si volem iterar cap enrere en una List farem:

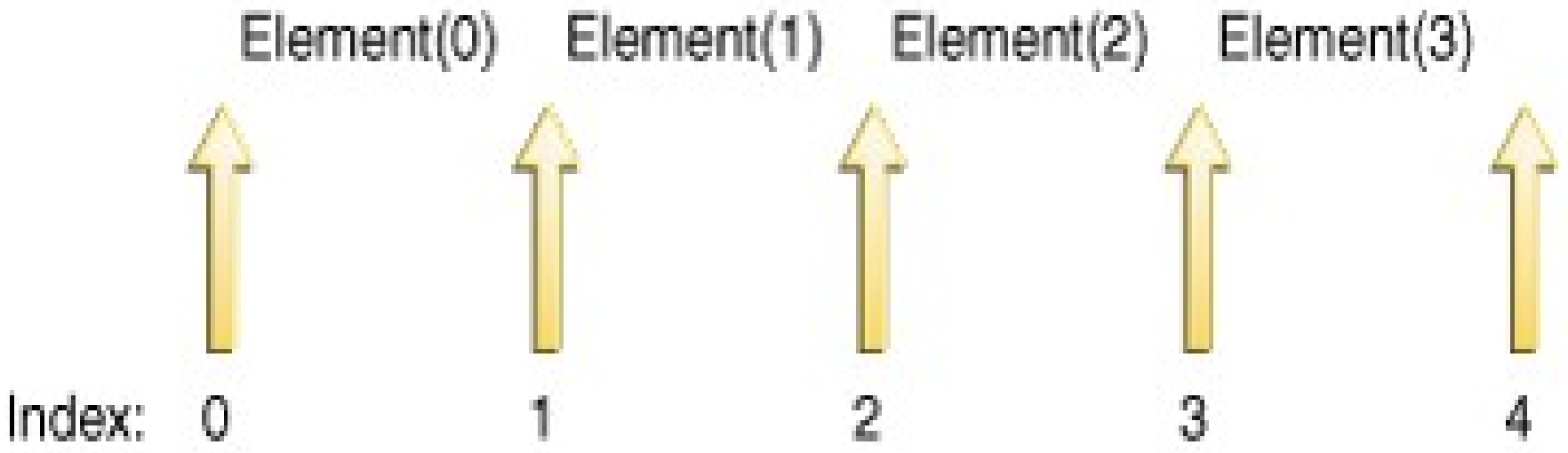
```
for (ListIterator<Type> it =
list.listIterator(list.size());
it.hasPrevious(); ) {
    Type t = it.previous();
    ...
}
```

El mètode listIterator

- El mètode listIterator, que retorna un ListIterator, vist a l'exemple anterior té dues signatures (sobrecàrregues). Una sense arguments, que retorna un ListIterator posicionat al principi de la llista. L'altra, amb un integer com a argument, que posiciona el ListIterator segons l'índex especificat per l'integer. L'índex es refereix a l'element que seria retornat per la primera crida feta a next. Una primera crida feta a previous, doncs, retornaria un element l'índex del qual serà index-1. *A una llista de mida n hi ha n+1 valors vàlids de l'índex, des de 0 fins a n*.

El mètode listIterator

- Intuitivament, els n+1 valors vàlids de l'índex corresponen als n+1 "espais" que es poden trobar entre els n elements, des de l'espai anterior al primer element fins al posterior al darrer element.



ListIterator – El cursor

- La primera crida a previous retorna el mateix element que la última crida a next.
- La primera crida a next, després d'una seqüència de crides a previous, retorna el mateix element que la última crida de previous.
- Això és degut a la posició que té el cursor. La posició del cursor s'indica amb l'índex i es pot obtenir amb els mètodes `nextIndex` (que retorna l'índex de l'element que seria retornat amb `next`) i `previousIndex` (que retorna l'índex de l'element que seria retornat amb `previous`).

ListIterator – El cursor

- D'aquesta manera, l'índex retornat per nextIndex és sempre una unitat major que l'índex retornat per previousIndex.
- Així, una crida previousIndex quan el cursor està situat just abans del primer element de la llista, retorna -1.
- Una crida a nextIndex quan el cursor està situat just després del darrer element, retorna list.size().

IndexOf implementation

```
public int indexOf(E e) {
    for (ListIterator<E> it = listIterator(); it.hasNext(); )
        if (e == null ? it.next() == null : e.equals(it.next()))
            return it.previousIndex();
    // Element not found
    return -1;
}
```

ListIterator – Altres mètodes

Volem destacar els mètodes remove, add i set.

- El mètode set sobreescriu el darrer element retornat per next o per previous, per l'element especificat.
- El mètode remove elimina el darrer element retornat per next o per previous.
- El següent exemple usa el mètode set per a substituir totes les ocorrències d'un element concret per un altre valor:

ListIterator – Altres mètodes

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

ListIterator – Altres mètodes

- El mètode Add insereix un nou element a la llista immediatament abans de la posició actual del cursor. Veiem un exemple (què fa i com ho fa?):

```
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); }}}
```

ListIterator – Operacions de rang

- A l'exemple NB Deal podem observar com es fan ús d'algunes operacions de rang. This example generates hands from a normal 52-card deck. It takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand.
- Un possible resultat d'executar Deal és:

```
% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4
of spades, king of diamonds]

[4 of diamonds, ace of clubs, 6 of clubs,
jack of hearts, queen of hearts]

[7 of spades, 5 of spades, 2 of diamonds,
queen of diamonds, 9 of clubs]

[8 of spades, 6 of diamonds, ace of spades, 3
of hearts, ace of hearts]
```

The Map Interface

- Un Map és un objecte que mapa claus (keys) amb valors (values).
- A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction.

The Map Interface

```
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);    }}
```

The Map Interface

- The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet, tal com hem descrit anteriorment.

Map – Mètodes bàsics

- Les operacions (mètodes) bàsiques d'un Map, són: put, get, containsKey, containsValue, size, i isEmpty.
- A l'exemple NB Freq generem una taula de freqüències de cadascuna de les paraules que passem com a argument. Aquesta taula mapa cada paraula amb un enter que representa la freqüència, és a dir, el nombre de vagades que la paraula apareix en la llista de paraules passada com a argument en l'execució del programa.

Map – Mètodes bàsics

- Parem atenció a la línia:

  ```
  m.put(a, (freq == null) ? 1 : freq + 1);
  ```

  En aquest cas, el segon argument és una expressió condicional que té l'efecte d'assignar una freqüència d'1 si la paraula mai s'ha vist abans, o de sumar-li 1 al valor actual de la freqüència si la paraula ja s'ha vist abans.

- El resultat del programa en executar

```
java Freq if it is to be it is up to me to delegate
```

és:

```
8 distinct words:
{be=1, delegate=1, if=1, is=2, it=2, me=1,
to=3, up=1}
```

Map – Diferents implementacions

- Si vulguéssim que les paraules apareguessin en ordre alfabètic, simplement canviaríem:

```
Map<String, Integer> m = new HashMap<String,
Integer>();
```

per

```
Map<String, Integer> m = new TreeMap<String,
Integer>();
```

Obtindrem ara:

```
8 distinct words:
```

```
{be=1, delegate=1, if=1, is=2, it=2, me=1,
to=3, up=1}
```

Map – Diferents implementacions

- Si pretenem que les paraules apareguin, a la sortida, en el mateix ordre que a l'entrada, la implementació a usar seria:

```
Map<String, Integer> m = new
LinkedHashMap<String, Integer>();
```

Resultat:

```
8 distinct words:

{if=1, it=2, is=2, to=3, be=1, up=1, me=1,
delegate=1}
```

Map Interface

- Igual que amb Set i List, Map permet comparar diferents Maps independentment de la implementació usada per cadascun. Els mètodes que s'usen per dur a terme les comparacions són equals i hashCode.

Conversions entre Maps

- By convention, all general-purpose Map implementations provide constructors that take a Map object and initialize the new Map to contain all the key-value mappings in the specified Map.

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

Map – Operacions massives (bulk)

- The clear operation removes all the mappings from the Map.
- The putAll operation is the Map analogue of the Collection interface's addAll operation. Its obvious use is dumping one Map into another.

Map – Collection Views

- Recordem que Map no és realment una Collection. Emperò, ofereix mètodes per tal que el Map pugui ser vist com si fos una Collection, i poder recorrer el Map igual a com ho fem amb una Collection:

keySet — the Set of keys contained in the Map.

values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.

entrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

Map – iterar pels índex

- Amb for-each:

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

- Amb Iterator:

```
// Filter a map based on some
// property of its keys.
for (Iterator<Type> it =
m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

Map – iterar pels valors

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

Map – Collection Views

- The Collection views support element removal in all its many forms — remove, removeAll, retainAll, and clear operations, as well as the Iterator.remove operation.

- The Collection views do not support element addition under any circumstances. It would make no sense for the keySet and values views, and it's unnecessary for the entrySet view, because the backing Map's put and putAll methods provide the same functionality.

Multimaps

- A multimap is like a Map but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to **use a Map whose values are List instances as a multimap**.
- Podeu veure l'exemple NB Anagrams, on un anagrama és un grup de paraules o totes contenen els mateixos caràcters però en diferents ordres.

  The program takes two arguments on the command line: (1) the name of the dictionary file and (2) the minimum size of anagram group to print out.

Object Ordering

A List l may be sorted as follows:

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

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String and Date both implement the Comparable interface. Comparable implementations provide a natural ordering for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement Comparable.

Object Ordering

- Classes Implementing Comparable (adjuntar taula)

The SortedSet Interface

- A **SortedSet** is a Set that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a Comparator provided at SortedSet creation time. In addition to the normal Set operations, the SortedSet interface provides operations for the following:

  Range view — allows arbitrary range operations on the sorted set

  Endpoints — returns the first or last element in the sorted set

  Comparator access — returns the Comparator, if any, used to sort the set

The SortedSet Interface

```
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
    Comparator<? super E> comparator();  }
```

The SortedMap Interface

- A **SortedMap** is a Map that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of the SortedMap creation.
- The SortedMap interface provides operations for normal Map operations and for the following:

Range view — performs arbitrary range operations on the sorted map

Endpoints — returns the first or the last key in the sorted map

Comparator access — returns the Comparator, if any, used to sort the map

The SortedMap Interface

```
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();
}
```

Exercicis

1 - Write a program that prints its arguments in random order. Do not make a copy of the argument array.

2 - Take the FindDups example and modify it to use a SortedSet instead of a Set. Specify a Comparator so that case is ignored when sorting and identifying set elements.

3 - Write a method that takes a List<String> and applies String.trim to each element. Per fer-ho necessitaràs recorrer la llista. Write a program that demonstrates that the method actually works!

Implementacions

- Per a ampliar coneixements sobre les diferents implementacions de les diverses interfícies de Collections, podeu consultar:

  http://docs.oracle.com/javase/tutorial/collections/implementations/index.html