Collections and Generics

1. Overview

The learning objectives for this laboratory session are:

- To understand how to use collections and generics
- Acquire knowledge concerning the use of important classes and interfaces included in the package `java.util`
- Hands-on experience in using Java collections.

A *collection* (aka container) is an object which groups several elements in a single unit. Collections are used to store, retrieve, manipulate and communicate aggregated data. They usually represent data items which form a natural group, such as a handful of playing cards (a card collection), an email folder (a message collection) or a telephone directory (a mapping from names to telephone numbers).

2. The Java Collections framework

A *framework* is a unified architecture for representing and manipulating collections. All the frameworks in JCF (Java Collection Framework) contain:

- **Interfaces:** Abstract data types representing collections and which enable programmers to manipulate collections independent of how these are implemented.
- **Implementations:** concrete implementations of collections interfaces. These are, in essence, reusable data structures.
- **Algorithms:** methods which perform useful operations, such as storing and searching, on/in objects implementing interfaces. Algorithms are *polymorphic:* i.e., the same method may be used on different implementations of the corresponding collection interface. Algorithms are reusable functionality.

2.1. Benefits of Java Collections Framework (JCF)

The main benefits resulted from using Java Collections Framework are:

- **Reduced programming effort:** by providing data structures and algorithms, useful in application development; by facilitating unrelated APIs interoperability.
- **Increased program speed and quality:** implementations are interchangeable, and thus programs can be adjusted by changing collections implementations.
- **Favoring software reuse:** new data structures which comply with standard collections interfaces are naturally reusable; this is also true for algorithms which operate on objects implementing those interfaces.

When writing applications we can concentrate our efforts on the problem at hand, and not on the ways to represent and manipulate data.

2.2. JCF Hierarchy

The interfaces in JCF form a hierarchy, as shown in Figure 1.

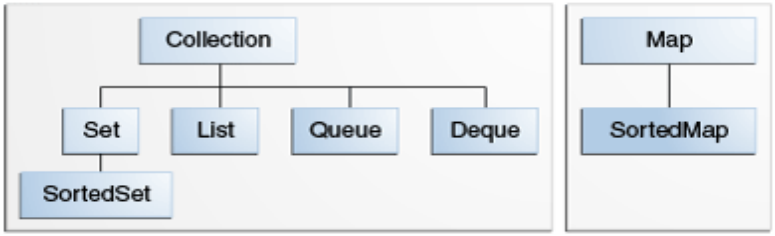

Figure 1. Main collection interfaces

2.3. JCF Summary

The following list enumerates the main JCF components.

- `List` – Extends *Collection*. Elements can be accessed sequentially or based on their index.
- `Map` – Stores key-value pairs, directly accessible based on their key. Figure 2 shows a mapping from persons to favorite colors.
- `SortedMap` - Extends *Map*, by adding in-order access.
- `Set` - Extends *Collection*. Contains only unique values.
- `SortedSet` - Extends *Set*; elements can be accessed in order.
- `Deque` – Double ended queue, used for both *stack* (LIFO) and *queue* (FIFO) operations.
- `Collections` – a class which contains a set of static methods for working with data structures.

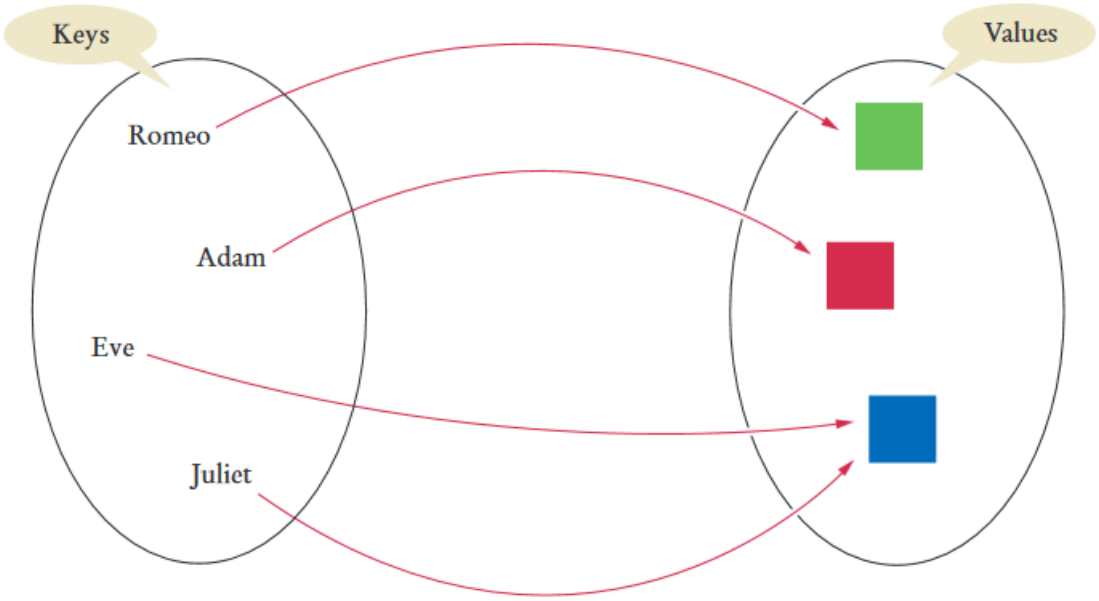

Figure 2. A mapping example

The `Collection` interface models a group of objects called elements. It is intended to facilitate the use of collections at a maximum generality. There are three categories of methods in the interface definition (see comment lines below).

```
public interface Collection<E> {
	// Basic operations at element level
	boolean isEmpty();
	boolean contains(E e);
	boolean add(E e);                       // Optional
	boolean remove(E e);                    // Optional
	Iterator<E> iterator();
	// Collection level operations
	int size();
	boolean containsAll(Collection<E> c);
	boolean addAll(Collection<E> c);        // Optional
	boolean removeAll(Collection<E> c);     // Optional
	boolean retainAll(Collection<E> c);     // Optional
	void clear(); // Optional
	// Conversion operations to a single-dimensional array
	Object[] toArray();
	E[] toArray(E[] a);
}
```

The `Set` interface models the mathematical notion for a set. A set cannot have duplicates. The `Set` interface defines the same methods as the `Collection` interface.

2.4. Important Classes and Interfaces

In what follows, the most useful classes are bolded.

```
AbstractCollection<E> implements Collection<E>, Iterable<E>
    AbstractList<E> implements List<E>
        ArrayList<E> implements RandomAccess
        AbstractSequentialList<E>
            LinkedList<E> implements Deque<E>
        Vector<E> implements RandomAccess<E>  // Synchronizat equivalent of ArrayList
            Stack<E>  // Adds push(), pop(), and peek()
    AbstractSet<E> implemets Set<E>
        HashSet<E>
            LinkedHashSet<E>
        TreeSet<E> implements SortedSet<E>
        EnumSet<E>   // Bitset implementation for the Enum class.
    AbstractQueue<E> implements Queue<E>
        PriorityQueue<E>
    ArrayDeque<E> implements Queue<E> Deque<E>
```

2.5. `Map` Links a Key to a Value

```
AbstractMap<K, V> implements Map<K, V>
    HashMap<K, V>
        LinkedHashMap<K, V> // Key can be iterated in the order of insertion
TreeMap<K, V> implementeaza SortedMap<K, V>
    EnumMap<K, V>          // Kye must be of the same Enum classm.
    WeakHashMap<K, V>      // Special use - Keys are weak references[1].
    IdentityHashMap<K, V>  // Special use - Keys must by identical.

Map.Entry<K, V>       // Map key/value pair.
```

2.6. Interfaces

```
Iterator<E>           // Interface needs hasNext(), next(), ?remove()
    ListIterator<E>   // Interface
Comparator <T>         // Interface needs compare() and equals()
// the following interfaces in java.lang are usually utilized in collections.
Iterable<T>          // Interface neeeds iterator()
Comparable<T>        // Interface neeeds compareTo()
```

2.7. Concrete Classes and Interfaces

Some of the most useful classes. Utility interfaces are omitted (such as `Cloneable` and `Serializable`).

Class	***Implementation***
The most used classes	
`ArrayList`	Sequence of values stored in a resizable array
`LinkedList`	Sequence of values stored in a linked list
`HashMap`	Key/value pairs in a hash table
`TreeMap`	Key/value pairs in a balanced binary tree

[1] A *weak reference*, is one which is not strong to force an object reside în memory. Weak reference allow the garbage collector to determine an objects accessibility.

HashSet	Unique value, stored in a hash table. Implements Set.
TreeSet	Unique Value, stored in a balanced binary tree. Implements Set.
Interfaces	
Collection	Methods common to all data structures
List	Fundamental methods for List. Implemented by ArrayList and LinkedList.
Map	Fundamental methods for Map (mapping). Implemented by HashMap and TreeMap.
Map.Entry	Key/Value pairs in Set returned by Map.entrySet().
Set	Fundamental methods for Set. Implemented by HashSet and TreeSet.
Iterator	Methods for "forward" iteration (from the first to the last element)
ListIterator	Additional methods for going backward.
Specialized classes	
BitSet	Expandable bit array.
LinkedBlockingDeque	May have a fixed upper limit. May block the ability to get an element till an element is added.
LinkedHashMap	Hash table in which elements can be accesses also in the order they have been added.
LinkedHashSet	Hash table in which elements can be accesses also in the order they have been added.
WeakHashMap	Hash table which uses weak references.
Preferences	For persistent program options.
Properties	Pre-Java 2, compare to Preferences
Older classes for which a replacement exists	
HashTable	Older, synchronized version of a HashMap.
Vector	Older, synchronized version of a ArrayList, still used.
Deprecated classes	
Dictionary	Abstract, deprecated class. Do not use it.

2.8. Interface Implementations

	Implementations			
Interface	**Array**	**Balanced Tree**	**Linked list**	**Hash Table**
List	ArrayList		LinkedList	
Map		TreeMap		HashMap
Set		TreeSet		HashSet
Deque	ArrayDeque		LinkedList	

2.9. Key-Value Pairs

Key-value pairs are stored in *mappings*.

2.9.1. Map Interfaces

- *Map* implemented by HashMap and TreeMap
- *SortedMap* implemented by TreeMap.
- *Map.Entry* which describes the access methods for key-value pairs

2.9.2. Classes implementing `Map`

Some classes implement the `Map` interface, including `HashMap`, `TreeMap`, `LinkedHashMap`, `WeakHashMap`, `ConcurrentHashMap`, and `Properties`. The most useful class is `HashMap`.

- `java.util.HashMap` is implemented using a hash table. Access in O(1). Entries are not sorted
- `java.util.LinkedHashMap` is implemented using a hash table. Access in O(1). Entries are sorted either in insertion order, or in last access order – this is useful when implementing the caching LRU (least recently used).
- `java.util.TreeMap` is implemented using a balanced binary tree. Access in O(log N). Unsorted entries.

2.9.3. Methods in the `Map` Interface

Some of the most useful methods in the `Map` interface are summarized below. *m* is a `Map`, *b* is a `boolean`, *i* is an `int`, *set* is a `Set`, *col* is a `Collection`, *key* is an `Object` used as a key to store a value, *val* is an `Object` storead as a value associated with the key.

Result	Method	Description
	Addition of a key-value pair to a mapping	
obj =	*m*.**put**(*key*, *val*)	Creates a mapping from *key* to *val*. Returns the value previously associated with that key (or `null`).
	m.**putAll**(*map2*)	Adds all key-value entries of another mapping, *map2*.
	Removal of key-value pairs form a mapping	
	m.**clear**()	Removes all elements in a mapping
obj =	*m*.**remove**(*key*)	Deletes the mapping form *key* to anything. Returns the value previously associated with the key (or `null`).
	Retrieval of information from a mapping	
b =	*m*.**containsKey**(*key*)	Returns `true` if *m* contains a key *key*
b =	*m*.**containsValue**(*val*)	Returns `true` if *m* contains *val* as one of the values
obj =	*m*.**get**(*key*)	Returns the value corresponding to *key*, or `null` if there is no mapping. If `null` was stored as a value, then use `containsKey` to check if there is a mapping.
b =	*m*.**isEmpty**()	Returns `true` if *m* doea not contain mappings (pairs).
i =	*m*.**size**()	Returns the number of mappings in *m*.
	Retrieving all keys, values, or key-value pairs (necessary for iteration)	
set =	*m*.**entrySet**()	Returns set of `Map`.Entry values for all mappings.
set =	*m*.**keySet**()	Returns `Set` of keys.
col =	*m*.**values**()	Returns a Collection view of the values in *m*.

2.9.4. Methods of the `Map.Entry` interface

Each element is a mapping and has a *key* and a value *value*. Every key-value pair is saved in an `java.util.Map.Entry` object. The set of these entries can be obtained by calling the `entrySet()` method of the mapping. Iteration over a mapping is achieved by iterating over this set.

In the following table, *me* stands for a `Map.Entry` object.

Result	Method	Description
obj =	*me*.**getKey**()	Returns the key in the pair.
obj =	*me*.**getValue**(*key*)	Returns the value in the pair.
obj =	*me*.**setValue**(*val*)	*Optional* operation which might not be supported by all `Map.Entry` objects. Sets the value in the pair, thus modifying the `Map` which has the pair. Returns the original value.

2.9.5. Constructors of the class HashMap

HashMap has the following constructors:

Result	Constructor	Description
hmap =	new HashMap()	Creates a new HashMap with an initial default capacity of 16 and a 0.75 load factor.
hmap =	new HashMap(*initialCapacity*)	Creates a new HashMap having the specified initial (int) capacity
hmap =	new HashMap(*initialCapacity*, *loadFactor*)	Creates a new HashMap with the specified initial capacity and which will not exceed the specified load factor (float).
hmap =	new HashMap(*mp*)	Creates a new HashMap with elements from the *Map mp*

2.9.6. Methods of the interface SortedMap

The *SortedMap* interface is used by TreeMap and adds methods which reflect the fact that a TreeMap is sorted.

Result	Method	Description
comp =	comparator()	Returns the comparator used for comparing keys. Null if the natural order is used (e.g. for String).
obj =	firstKey()	Key of the first element (in sorted order).
obj =	lastKey()	Key of the last element (in sorted order).
smp =	headMap(*obj*)	Returns a *SortedMap* containing all the elements which are smaller than *obj*.
smp =	tailMap(*obj*)	Returns a *SortedMap* containing all the elements which are greater of equal to *obj*.
smp =	subMap(*fromKey*, *toKey*)	Returns a *SortedMap* containing all the elements which are greater of equal to *fromKey* and smaller than *toKey*.

3.2.1. Constructors of the class TreeMap

TreeMap implements the methods of the interfaces *Map* and *SortedMap*. Different from HashMap, TreeMap maintans the balanced binary tree in sorted order by key. If the key has a natural order (as it is the case with String) it is okay, but often you will have to supply a Comparator object which should tell how the keys are compared. It has the following constructors:

Result	Constructor	Description
tmap =	new TreeMap()	Creates a new TreeMap. Keys sorted by natural order.
tmap =	new TreeMap(*comp*)	Creates a new TreeMap using the comparator *comp* to sort keys.
tmap =	new TreeMap(*mp*)	Creates a new TreeMap from *Map mp* using its natural order.
tmap =	new TreeMap(*smp*)	Creates a new TreeMap from *SortedMap smp* using the order of keys in *smp*.

2.10. The class Collections

Some of the utility **static** methods, of the java.util.Collections are briefly presented in the following.

Let us assume that we have written the following declarations, where T stands for a class or interface type.

```
int i;
List<T> listC;   // List of comparable objects.
List<T> list;    // Any kind of list. Does not have to be Comparable.
Comparator <T> comp;
```

```
T key;
T t;
Collection<T> coll;  // Any kind of Collection (List, Set, Deque).
Collection<T> collC; // Collection implementing Comparable.
```

Rearranging – Sorting, mixing, . . .		
	`Collections.sort(listC)`	Sorts `listC`. Elements must be *Comparable<T>*. Stable sort, O(N log N).
	`Collections.sort(list, comp)`	Sorts `list` using a comparator.
	`Collections.shuffle(list)`	Puts the elements in `list` in random order.
	`Collections.reverse(list)`	Reverses the elements in `list`.
Searching		
`i =`	`Collections.binarySearch(listC, key)`	Searches for `key` in `list`. Returns the index of the element or a negative value if it does not find it. Uses binary search.
`i =`	`Collections.binarySearch(list, key, comp)`	Searches for `key` in `list` using the `Comparator` comp.
`t =`	`Collections.max(collC)`	Returns the *Comparable* object in `collC`, having the maximum value.
`t =`	`Collections.max(coll, comp)`	Returns the maximum valued object in `coll`, using `Comparator comp`.
`t =`	`Collections.min(collC)`	Returns the maximum valued *Comparable* object in `collC`.
`t =`	`Collections.min(coll, comp)`	Returns the minimum valued object in `coll`, using `Comparator comp`.

Therea are others, these are just a few.

Searching can be implemented in the classes implementing interfaces, as well

All *`List`* and *`Set`* classes implements the method `contains()`, which performs a linear search on lists (O(N)), a binary search on TreeSets (O(log N)), and a hash-based one on HashSets (O(1)).

Mappings define `get()` to look for a key. For `HashMap` search is O(1), and for `TreeMap` search is O(log N).

Sorting may be implicit in a class which implements the interfaces

Data in `TreeSet` and keys in `TreeMap` maintained sorted. These collections are implemented as binary trees, and thus insertion and searching are both O(log N). One may use an iterator to retrieve data (`TreeSet`s) or keys (`TreeMap`s) in sorted order. Either the class of an element must be `Comparable`, or a `Comparator` must be provided.

3. Comparators

3.1. Defining your own sort order

The main use of comparators is passing them as arguments to something that sorts, either one of the default sort methods, or a data structure that sorts by default (e.g. `TreeSet` or `TreeMap`).

3.2. `java.util.Comparator` Interface

`java.util.Comparator` interface can be used to create objects to be passed to sort methods or data structures which sort. A `Comparator` must define a `compare` function which receives as arguments two `Object` and returns -1, 0, or 1 (less than, equal, greater than).

Comparators are not needed if there exists a natural sort order
Comparators are not needed for primitive values arrays or arrays or collections of objectswhich have a natural order (such as, e.g. String, BigInteger, etc.)

String also has a predefined comparator for case insensitive sort, String.CASE_INSENSITIVE_ORDER .

3.2.1. Example of using `Comparator` objects

In the following example, the files in a directory are read, and then they are sorted in two ways.

```
// File: arrays/filelist/Filelistsort.java
// Purpose: Listing the contents of the default (home) directory of a user
//          Demonstrates using comparators to sort the same array
//          using two different criteria.
// Author: Fred Swartz 2006-Aug-23  Public domain.

import java.util.Arrays;
import java.util.Comparator;
import java.io.*;

public class Filelistsort {

    //=========================================================== main
    public static void main(String[] args) {
        //... Creates the sorting comparators.
        Comparator<File> byDirThenAlpha = new DirAlphaComparator();
        Comparator<File> byNameLength   = new NameLengthComparator();

        //... Creates a File object for user directory.
        File dir = new File(System.getProperty("user.home"));
        File[] children = dir.listFiles();

        System.out.println("Files by directory, then alphabetically ");
        Arrays.sort(children, byDirThenAlpha);
        printFileNames(children);

        System.out.println("Files by length of their name (longest first)");
        Arrays.sort(children, byNameLength);
        printFileNames(children);
    }

    //============================================== printFileNames
    private static void printFileNames(File[] fa){
        for (File oneEntry : fa) {
            System.out.println("   " + oneEntry.getName());
        }
    }
}


/////////////////////////////////////////////////// DirAlphaComparator
// To sort directories first, then alphabetically.
class DirAlphaComparator implements Comparator<File> {

    // Comparator interface requires to define the method compare.
    public int compare(File filea, File fileb) {
        //... Sort directories before files,
        //    otherwise alphabetically, case insensitive.
        if (filea.isDirectory() && !fileb.isDirectory()) {
            return -1;

        } else if (!filea.isDirectory() && fileb.isDirectory()) {
            return 1;

        } else {
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
```

```
}


////////////////////////////////////////////// NameLengthComparator
// To sort by file/directory name length (longest first).
class NameLengthComparator implements Comparator<File> {

    // Comparator interface requires defining compare method.
    public int compare(File filea, File fileb) {
        int comp = fileb.getName().length() - filea.getName().length();
        if (comp != 0) {
            //... daca lungimile sunt diferite, am terminat.
            return comp;
        } else {
            //... daca lungimile sunt egale, sorteaza alfabetic.
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}
```

4. Generics

Usage of generic classes is common, writing generic classes is less common. Generics are mainly a way to allow library authors to write something which enables users to adapt them to their own types.

4.1. Basic problem – restricted or totally open types

When you need to write something which works well with object form may classes or interfaces passed as parameters, then you have a problem. When selecting a type T, then you may use only objects of that type or its subclasses. If you need something more general, you often have to go up the inheritance hierarchy to `Object`, which works for all types, and thus is totally general. This is the way Java collections were written up to Java 5 – they often had `Object` as a parameter type. It was very convenient for implementors, but less useful for users.

Constraining a type. For most of the data structures there is only one type which should actually be used; for instance, you have an `ArrayList` of `String`, or an `ArrayList` of `Date`, but you seldom mix them. Indeed, it is considered bad style to mix them. Although the library methods working with `Object` allow the use and the addition of different types, by mistake.

4.2. Static vs dynamic typing

Static/strong typing. One of the attractive elements in Java is that it has what is called "*strong typing*" – the type of the variables (and other elements) is declared, and the values assigned to the variable must be of that type. This generates more work when writing a program, to set them in order, but the compiler error messages are a better solution than allowing the code to perform incorrect runtime assignments.

Weak/dynamic typing is used in some languages (e.g. Ruby), which allow variables to be assigned different types of values, which later have the type of the last assigned value. The people who proposed this solution say that they did that in order to relieve the programmer to take care to correctly specify types in the source code which would speed up coding, and with (Test-Driven Development)any wrong assignments would be discovered and corrected rapidly.

Java uses static/strong typing, and the introduction of generics allows for even stronger typing.

4.3. Examples which compare old style with generics

Non-generic example

```
// Tipical use before Java 5
List greetings = new ArrayList();
greetings.add("We come in peace.");
```

Same example using generics

```
// Same example with generics.
List<String> greetings = new
ArrayList<String>();
greetings.add("We come in peace.");
```

```
greetings.add("Take me to your
leader.");
greetings.add("Resistance is futile.");

Iterator it = greetings.iterator();
while (it.hasNext()) {
    String aGreeting =
(String)it.next();
    attemptCommunication(aGreeting);
}
```

```
greetings.add("Take me to your leader.");
greetings.add("Resistance is futile.");

Iterator<String> it = greetings.iterator();
while (it.hasNext()) {
    String aGreeting = it.next();  // No
downcast.
    attemptCommunication(aGreeting);
}
```

Specification of the type of elements in a collection has good consequences:

- An attempt to add something of a wrong type results in a compilation error.
- Getting elements does not need a downcast, even though our example dos not show how often this thing helps.
- Type specification provides more documentation.

4.4. Type Parameter Naming - T, U, ...

Even though there are many names possible for parameter types, traditionally these are written using letters of the sequence T, U, V, ... Other capitals are used with specific meaning, e.g. K for a *key* type, E an *element* type.

4.5. Reading type parameters

Notation	Meaning
`List<T>`	`List` of elements of type T (T is a *concrete type*)
`List<?>`	`List` of any type (? is an *unbounded wildcard*)
`List<? super T>`	`List` of any type (? Is a *bounded wildcard* – super-type of T)
`List<? extends T>`	`List` of any type (? Is a *bounded wildcard* – sub-type of T)
`List<U extends T>`	`List` of any type (U must be a super-type of T)

4.6. Where you CANNOT use generic types in the definition of a generic class/method

Limitations, not all obvious:

- You cannot create objects of type T.
- You cannot create arrays of type T.
- You cannot use them for statics.

4.7. Implementation with type erasure of generic types in Java

For compatibility with JVM (Java Virtual Machine) implementations, generics in Java are seen only by the compiler. At runtime, all the generic information was "erased" and replaced by `Object`. This is a clever way to assure compatibility, but also the cause of several complications, e.g. forbidding arrays of generic types.

4.7.1. An example generic class: a generic matrix

The following code implements a `GenericMatrix` class and a `IntegerMatrix` subclass.

```
public class IntegerMatrix extends GenericMatrix<Integer> {
  @Override /** Add two integers */
  protected Integer add(Integer o1, Integer o2) {
    return o1 + o2;
```

```
  }

  @Override /** Multiply two integers */
  protected Integer multiply(Integer o1, Integer o2) {
    return o1 * o2;
  }

  @Override /** Specify zero for an integer */
  protected Integer zero() {
    return 0;
  }
}

public abstract class GenericMatrix<E extends Number> {
  /** Abstract method for adding two elements of the matrices */
  protected abstract E add(E o1, E o2);

  /** Abstract method for multiplying two elements of the matrices */
  protected abstract E multiply(E o1, E o2);

  /** Abstract method for defining zero for the matrix element */
  protected abstract E zero();

  /** Add two matrices */
  public E[][] addMatrix(E[][] matrix1, E[][] matrix2) {
    // Check bounds of the two matrices
    if ((matrix1.length != matrix2.length) ||
        (matrix1[0].length != matrix2[0].length)) {
      throw new RuntimeException(
        "The matrices do not have the same size");
    }

    E[][] result =
      (E[][])new Number[matrix1.length][matrix1[0].length];

    // Perform addition
    for (int i = 0; i < result.length; i++)
      for (int j = 0; j < result[i].length; j++) {
        result[i][j] = add(matrix1[i][j], matrix2[i][j]);
      }

    return result;
  }

  /** Multiply two matrices */
  public E[][] multiplyMatrix(E[][] matrix1, E[][] matrix2) {
    // Check bounds
    if (matrix1[0].length != matrix2.length) {
      throw new RuntimeException(
        "The matrices do not have compatible size");
    }

    // Create result matrix
    E[][] result =
      (E[][])new Number[matrix1.length][matrix2[0].length];

    // Perform multiplication of two matrices
    for (int i = 0; i < result.length; i++) {
      for (int j = 0; j < result[0].length; j++) {
        result[i][j] = zero();

        for (int k = 0; k < matrix1[0].length; k++) {
          result[i][j] = add(result[i][j],
            multiply(matrix1[i][k], matrix2[k][j]));
        }
      }
```

```
    }

    return result;
  }

  /** Print matrices, the operator, and their operation result */
  public static void printResult(
      Number[][] m1, Number[][] m2, Number[][] m3, char op) {
    for (int i = 0; i < m1.length; i++) {
      for (int j = 0; j < m1[0].length; j++)
        System.out.print(" " + m1[i][j]);

      if (i == m1.length / 2)
        System.out.print("  " + op + "  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m2.length; j++)
        System.out.print(" " + m2[i][j]);

      if (i == m1.length / 2)
        System.out.print("  =  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m3.length; j++)
        System.out.print(m3[i][j] + " ");

      System.out.println();
    }
  }
}

public class TestIntegerMatrix {
  public static void main(String[] args) {
    // Create Integer arrays m1, m2
    Integer[][] m1 = new Integer[][]{{1, 2, 3}, {4, 5, 6}, {1, 1, 1}};
    Integer[][] m2 = new Integer[][]{{1, 1, 1}, {2, 2, 2}, {0, 0, 0}};

    // Create an instance of IntegerMatrix
    IntegerMatrix integerMatrix = new IntegerMatrix();

    System.out.println("\nm1 + m2 is ");
    GenericMatrix.printResult(
      m1, m2, integerMatrix.addMatrix(m1, m2), '+');

    System.out.println("\nm1 * m2 is ");
    GenericMatrix.printResult(
      m1, m2, integerMatrix.multiplyMatrix(m1, m2), '*');
  }
}
```

5. Iterators

Collections `List` and `Set` provide *iterators*, which are objects enabling a sequential traversal of the whole collection. The interface `java.util.Iterator<E>` specifies a one way traversal, and `java.util.ListIterator<E>` specifies a two way traversal (form the beginning to the end and vice-versa). `Iterator<E>` is a replacement for the older `Enumeration` class which was used before collections were added to Java.

5.1. Creating an `Iterator`

Iterators are created by invoking the *`iterator()`* or *`listIterator()`* method of a `List`, `Set`, or other data collection with iterators.

5.2. Methods of an `Iterator`

`Iterator` defines three methods, one of them optional.

Result	Method	Description
b =	*it*.**hasNext**()	`true` if there are more elements to traverse.
obj =	*it*.**next**()	Returns the next object. I a generic list is accessed, then the iterator returns something of the type of the list. Pre-generic iterators always returned the type `Object`, and thus usually a downcast was needed.
	it.remove()	Removes the most recent element that was returned by `next`. Not all collections support `delete`. An *UnsupportedOperationException* will be thrown if the collection does not support *remove()*.

5.2.1. An example with generics

An iterator could be used like this:

```
ArrayList<String> alist = new ArrayList<>();
// . . . Add Strings to alist

for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    String s = it.next();  // No need for downcasting
    System.out.println(s);
}
```

5.2.2. Previous example with for-each

```
for (String s: alist) {
    System.out.println(s);
}
```

5.2.3. Example pre Java 5, with an explicit iterator and downcasting

In Java versions < 5, an iterator could be used like this:

```
ArrayList alist = new ArrayList(); // has the type Object.
// . . . Add Strings to alist

for (Iterator it = alist.iterator(); it.hasNext(); ) {
    String s = (String)it.next();  // Downcasting is needed pre Java 5.
    System.out.println(s);
}
```

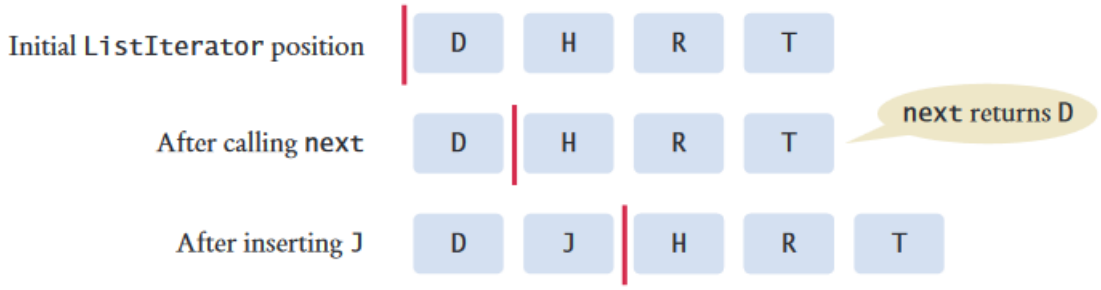

Figure 1. A conceptual view of a ListIterator. Note the red-marked positions

5.3. ListIterator Methods

`ListIterator` is implemented only by classes which implement the `List` interface (`ArrayList`, `LinkedList`, and `Vector`). `ListIterator` provides the following:

Result	Method	Description
Forward iteration		
b =	*it*.**hasNext**()	`true` if there is a next element in the collection.
obj =	*it*.**next**()	Returns the next element.

Backward iteration		
b =	*it*.**hasPrevious**()	true if there is a previous element in the collection.
obj =	*it*.**previous**()	Returns the previous element.
Getting the index of an element		
i =	*it*.**nextIndex**()	Returns the index of the element that would be returned by a call to *next()*.
i =	*it*.**previousIndex**()	Returns the index of the element that would be returned by a call to *previous()*.
Optional modifier methods. UnsupportedOperationException is thrown if unsupported.		
	it.add(*obj*)	Inserts *obj* in the collection at the "current" position (before the next element which would be returned by next(), and after an element which would be returned by previous()).
	it.set()	Replaces the "current" element (the most recent element returned by a call to next() or previous()).
	it.remove()	Removes the "current" element (the most recent element returned by a call to next() or previous()).

5.4. How NOT to do

Question: What does the following loop do? Note the mix of iterator and index.

```
ArrayList<String> alist = new ArrayList<String>();
// . . . Add Strings to alist

int i = 0;
for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    System.out.println(alist.get(i++));
}
```

Answer: Throws an exception when the end it goes beyond the end.

When hasNext() returns true, the only way to advance an iterator is by calling next(). But the element was obtained with get(),and thus the iterator doesn not advance. hasNext() will always be true true (because there is a first element),and, finally get() will request for something beyond the end of ArrayList. Use **either** the scheme with iterator:

```
for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
    System.out.println(it.next());
}
```

or the one with indexing, but never mix them:

```
for (int i=0; i < alist.size(); i++) {
    System.out.println(alist.get(i));
}
```

6. Application Sample which Use JCF

6.1. An example from Java Tutorial

This example is from Oracle Java Tutorial.
Develop a program which reads a text file, with the name given as the first command line argument, in a List. Then the program should print random lines from the file, the number of lines being specified by the second command line argument. Write the program such a way that a correctly sized collection is allocated at once instead of being gradually expanded as the file is read. Hint: to find the number of lines in the file, use java.io.File.length get the size of the file, then divide this size by an average assumed line length..
Solution: Because we access the list at random, we will use ArrayList. We estimate the number of lines dividing by 50 the file size. Then we double the value we get, as it is more efficient to over-estimate than to under-estimate.

```
import java.util.*;
```

```
import java.io.*;

public class FileList {
    public static void main(String[] args) {
        final int assumedLineLength = 50;
        File file = new File(args[0]);
        List<String> fileList =
            new ArrayList<String>((int)(file.length() / assumedLineLength) * 2);
        BufferedReader reader = null;
        int lineCount = 0;
        try {
            reader = new BufferedReader(new FileReader(file));
            for (String line = reader.readLine(); line != null;
                    line = reader.readLine()) {
                fileList.add(line);
                lineCount++;
            }
        } catch (IOException e) {
            System.err.format("Cannot read %s: %s%n", file, e);
            System.exit(1);
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {}
            }
        }
        int repeats = Integer.parseInt(args[1]);
        Random random = new Random();
        for (int i = 0; i < repeats; i++) {
            System.out.format("%d: %s%n", i,
                    fileList.get(random.nextInt(lineCount - 1)));
        }
    }
}
```

This program spends more time reading the file, thus pre-allocating the `ArrayList` has minor impact on its performance. The specification of the initial capacity in advance is more likely to be useful when your program creates `ArrayList` objects without interleaved input/output operations.

6.2. An Example HashSet Usage

The following example uses `HashSet` to count the keywords in a Java program.

```
import java.util.*;
import java.io.*;

public class CountKeywords {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a Java source file: ");
    String filename = input.nextLine();

    File file = new File(filename);
    if (file.exists()) {
      System.out.println("The number of keywords in " + filename
        + " is " + countKeywords(file));
    }
    else {
      System.out.println("File " + filename + " does not exist");
    }
  }
```

```
  public static int countKeywords(File file) throws Exception {
    // Array of all Java keywords + true, false and null
    String[] keywordString = {"abstract", "assert", "boolean",
        "break", "byte", "case", "catch", "char", "class", "const",
        "continue", "default", "do", "double", "else", "enum",
        "extends", "for", "final", "finally", "float", "goto",
        "if", "implements", "import", "instanceof", "int",
        "interface", "long", "native", "new", "package", "private",
        "protected", "public", "return", "short", "static",
        "strictfp", "super", "switch", "synchronized", "this",
        "throw", "throws", "transient", "try", "void", "volatile",
        "while", "true", "false", "null"};

    Set<String> keywordSet =
      new HashSet<String>(Arrays.asList(keywordString));
    int count = 0;

    Scanner input = new Scanner(file);

    while (input.hasNext()) {
      String word = input.next();
      if (keywordSet.contains(word))
        count++;
    }

    return count;
  }
}
```

6.3. Example TreeMap Usage

The following example uses a TreeMap to count word occurrences.

```
import java.util.*;

public class CountOccurrenceOfWords {
  public static void main(String[] args) {
    // Set text in a string
    String text = "Good morning. Have a good class. " +
      "Have a good visit. Have fun!";

    // Create a TreeMap to hold words as key and count as value
    Map<String, Integer> map = new TreeMap<String, Integer>();

    String[] words = text.split("[ \n\t\r.,;:!?(){}]");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

      if (key.length() > 0) {
        if (!map.containsKey(key)) {
          map.put(key, 1);
        }
        else {
          int value = map.get(key);
          value++;
          map.put(key, value);
        }
      }
    }

    // Get all entries into a set
    Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
```

```
    // Get key and value from each entry
    for (Map.Entry<String, Integer> entry: entrySet)
      System.out.println(entry.getKey() + "\t" + entry.getValue());
  }
}
```

7. How to Choose a Collection?

You can use, as a guide, the following steps:

1. Determine the value access mode. How can you access individual values? You have more options:
   - Values are accessed using an integer position. Use an `ArrayList`
   - Values are accessed by a key, which is not part of the object. Use a `Map`.
   - Values are accessed only at one end. Use a queue (for FIFO) or s stack (for LIFO).
   - You do not need to access individual position values. Refine your choice in steps 3 and 4
2. Determine the type of elements or the type of key/value pairs
   For a list or a set, determine the type of elements you want to store. For instance, if the collection s a set of books, then the element type is `Book`. Similarly, for a mapping, you find the types of keys and associated values. If you want to look for books by ID you may use
   `Map <Integer, Book>` or `Map <String, Book>`, depending on the type of ID.
3. Find if the order of elements or the order of keys is important. You have several possibilities:
   - Elements or keys must be sorted. Use a `TreeSet` or a `TreeMap`. Go to step 6.
   - Elements must be retrieved in insertion order. Use `LinkedList` or `ArrayList`.
   - It only matters to be able to visit all elements. If you have chosen `Map` at step 1 then use `HashMap` and go to step 5.
4. Find what operations must be efficient on the collection. You have the following possibilities:
   - Element retrieval must be efficient. Use a `HashSet`.
   - It must be efficient to add or delete elements at the beginning, at end, or in the current position. Choose `LinkedList`
   - You insert or delete only at the end, or you collect just a few elements, and thus speed does not matter. Use `ArrayList`.
5. For `HashSet` and `Map decide whether you need to implement hashcode` and `equals`. You have the following possibilities:
   - If the elements or the keys belong to an already implemented class, then check if the class has its own `hashcode` and `equals`. This is true for Java library classes such as `String, Integer, Rectangle,` etc.
   - If not, decide if you must compare elements by identity. This is the case in which you do not construct two distinct elements having the same contents. If you do, then you need to implement `hashcode` and `equals`. The implementation in `Object` is enough.
   - Otherwise you must implement your own methods.
6. If you use a binary tree, then decide if you have to provide a comparator. Check the class of the element set or the key set. Do they implement `Comparable`? If yes, then does the `compareTo` method sort in correct order? If yes, the there is nothing to do. Again, that is the case with Java library classes. If not, then you have to implement the `Comparable` interface or declare a class which implements the `Comparator` interface.

8. Lab Tasks

1. Study and execute the code presented in this paper, in order to understand the way the types presented work and how they are used.

2. Create, using the `IntegerMatrix` as a model, matrices of other types (`DoubleMatrix, LongMatrix` etc.) and check that they perform correctly.

3. Make changes in the provided code (e.g. add methods and types) and notice the effect of those changes.