

# The Collections Framework

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

Johannes Kepler University, Linz, Austria

[Wolfgang.Schreiner@risc.jku.at](mailto:Wolfgang.Schreiner@risc.jku.at)

<http://www.risc.jku.at>

## Generics

A generic is a type (class/interface) parameterized over another type.

```
// T is a type parameter
public class Box<T>
{
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Objects of type `Box<T>` encapsulate values of some type *T*.

## Type Instantiation

A generic is to be instantiated before use.

```
Box<String> sbox = new Box<String>();  
sbox.set("text");  
String s = sbox.get();
```

```
Box<Integer> ibox = new Box<Integer>();  
ibox.set(5);          // autoboxing:   ibox.set(new Integer(5));  
int i = ibox.get();  // autounboxing: int i = ibox.get().intValue();
```

The instantiation type must be a class (not a primitive type).

## Generic Interfaces

Also interfaces may be generic.

```
public interface Pair<A,B>
{
    public A getFirst();
    public B getSecond();
}
```

A generic may also have multiple parameters.

## Generic Interfaces (Continued)

```
public class PairClass<A,B> implements Pair<A,B>
{
    private A first;
    private B second;

    public PairClass(A first, B second)
    { this.first = first; this.second = second; }

    public A getFirst() { return first; }
    public B getSecond() { return second; }
}
```

```
Pair<String,Integer> p = new PairClass<String,Integer>("key", 7);
String s = p.getFirst();
int i = p.getSecond(); // autounboxing
```

## Implementation of Generics

Java implements generics by “type erasure”.

- JVM only represents the “raw type” of a generic `Box<T>`.
  - Java: multiple instances `Box<String>` and `Box<Integer>`.
  - JVM: single raw type `Box` with `T` set to `Object`.
- No runtime information about the instantiation of `T`.
  - It is illegal to create an instance of `T`.

```
public class Generic<T>
{ public void method() { T x = new T(); } // illegal }
```

- It is illegal to create an array of parameterized types.

```
Box<Integer>[] array = new Box<Integer>[10]; // illegal
```

However, we may create collections of parameterized types.

## The Collections Framework

A library for the uniform processing of multiple types of “containers”.

- Three kinds of entities:
  - **Interfaces:** abstract data types that denote the collections.
  - **Implementations:** classes that provide concrete implementations for the interfaces.
  - **Algorithms:** methods that operate on interfaces independently of their implementation.
- Two kinds of interfaces and associated classes:
  - **Collections:** based on (interfaces derived from) interface `Collection<E>`; objects represent certain kinds of “collections” of “elements” of type  $E$ .
    - \* Lists, sets, queues/deques.
  - **Maps:** based on interface `Map<K, V>`; objects represent “maps” of “keys” of type  $K$  to “values” of type  $V$ .

We are now going to investigate both “collections” and “maps”.

## Collections

```
public interface Collection<E> extends Iterable<E> {
    boolean add(E e);                // add e to this collection
    boolean addAll(Collection<E> c); // add all elements of c
    void clear();                    // make collection empty
    boolean contains(Object o);      // is o in this collection?
    boolean containsAll(Collection<E> c); // are all elems of c in this?
    boolean isEmpty();               // is this collection empty?
    boolean remove(Object o);        // remove o from this collection
    boolean removeAll(Collection<E> c); // remove all elems of c
    int size();                       // number of elements
}
```

```
Collection<String> c = ...
for (String e : c) System.out.println(e);
```

**We may iterate over all elements of a collection.**

## Sequenced Collections

```
public interface SequencedCollection<E> extends Collection<E>
{
    void addFirst(E e); // add e as first element to this collection
    void addLast(E e); // add e as last element to this collection
    E getFirst();      // get first element of this collection
    E getLast();       // get last element of this collection
    E removeFirst();   // remove first element from this collection
    E removeLast();    // remove last element from this collection
    SequencedCollection<E> reversed();
                        // get reverse-ordered view of this collection
}
```

```
SequencedCollection<String> seq = ...
SequencedCollection<String> list = new ArrayList<String>(seq.reversed());
```

The elements of a sequenced collection occur in a well-defined order.

## Lists

```
public interface List<E> extends SequencedCollection<E> {
    E get(int index);           // get element at index >= 0
    E set(int index, E element); // set element at index >= 0
    void add(int index, E element); // insert element at index >= 0
    int indexOf(Object o);      // index of first occurrence of o (or -1)
    int lastIndexOf(Object o); // index of last occurrence of o (or -1)
    List<E> subList(int from, int to); // sublist view of range [from,to[
}
```

```
String[] a = new String[...];
List<String> l = new ArrayList<String>(Arrays.asList(a));
...
l.subList(from, to).clear();
```

The type of ordered collections whose elements can be accessed by indices; conversion from arrays to lists (and vice versa) are possible.

## List Implementations

```
// implementation by an array
public class ArrayList<E> ... implements List<E>, ... {
    public ArrayList();
    public ArrayList(Collection<E> c);
    ...;
}
// implementation by a linked list
public class LinkedList<E> ... implements List<E>, ... {
    public LinkedList();
    public LinkedList(Collection<E> c);
    ...;
}
```

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

**Arbitrary collections may be converted to lists.**

## Sets

```
public interface Set<E> extends Collection<E> { ... }  
public interface SequencedSet<E> extends SequencedCollection<E>, Set<E> { ... }
```

```
// implementation by a hash table  
public class HashSet<E> ... implements Set<E>, ... {  
    public HashSet();  
    public HashSet(Collection<E> c);  
    ...;  
}
```

```
// implementation by a tree  
public class TreeSet<E> ... implements SequencedSet<E>, ... { ... }
```

```
// implementation by a hash table and a linked list  
public class LinkedHashSet<E> ... implements SequencedSet<E>, ... { ... }
```

The type of collections without duplicate elements.

## Set Implementations

### ● `HashSet<E>`

- Hash table, i.e., array that stores every element at a position computed from the element by application of a “hash function” that maps the elements to an integer (the “hash code”).
- Very fast but with unpredictable iteration order.
- $E$  must overwrite `int hashCode()` such that “equal” elements yield the same hash codes.

### ● `TreeSet<E>`

- Binary tree, i.e., dynamic data structure that generalizes linked lists in that every node has two successor nodes.
- Slower but with natural iteration order (e.g. integers in increasing order).
- $E$  must implement interface `Comparable`, i.e., provide a method `int compareTo(E x)`.

### ● `LinkedHashSet<E>`

- Combination of a hash table and a linked list to which new elements are added.
- Fast and iteration returns elements in insertion order.
- $E$  must overwrite `int hashCode()` such that “equal” elements yield the same hash codes.

## Example

```
Collection<String> c = ...;  
Set<String> set = new TreeSet<String>(c);  
for (String s : set) System.out.println(s);
```

Prints all strings in a collection in alphabetic order (with duplicates removed).

## Queues and Deques

```
public interface Queue<E> extends Collection<E> {  
    public E remove(); // retrieve and remove head  
    public E element(); // retrieve head without removing it  
}
```

```
public interface Deque<E> extends Queue<E>, SequencedCollection<E> {  
    ...  
}
```

In a queue; elements are added to one end and removed from the other; in a deque (“doubly-ended queue”), elements can be added/removed to/from either end by the sequenced collection operations.

## Queue and Dequeue Implementations

```
// implementation by an array
public class ArrayDeque<E> ... implements Deque<E>, ...
{
    public ArrayDeque();
    public ArrayDeque(Collection<E> c);
    ...;
}
```

```
// implementation by a linked list
public class LinkedList<E> ... implements List<E>, Deque<E>, ...
{
    public LinkedList();
    public LinkedList(Collection<E> c);
    ...;
}
```

## Example

```
int n = ... ;
Queue<Integer> queue = new LinkedList<Integer>();
for (int i = 0; i < n; i++)
    queue.add(i);
while (!queue.isEmpty())
{
    int i = queue.remove();
    System.out.println(i);
}
```

Add  $n$  integers  $0, \dots, n - 1$  to a queue and print them out in the order in which they were inserted.

## Maps

```
public interface Map<K,V> {
    public V put(K key, V val); // maps key to value, returns previous value (or null)
    public V get(K key);      // returns value to which key is mapped (or null)
    ...

    // set of keys, collection of values, set of key/value pairs
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Entry<K,V>> entrySet();

    public static interface Entry<K,V> { // the type of a key/value pair
        public K getKey();
        public V getValue();
    }
}
```

**Access to values via keys or by iterating over the map elements.**

## Sequenced Maps

```
public interface SequencedMap<K,V> extends Map<K,V>
{
    ...
    // *sequenced* set of keys, collection of values, set of key/value pairs
    public SequencedSet<K> sequencedKeySet();
    public SequencedCollection<V> sequencedValues();
    public SequencedSet<Entry<K,V>> sequencedEntrySet();

    // get reverse-ordered view of this map
    SequencedMap<K,V> reversed();
}
```

Maps whose entries occur in some well-defined order.

## Map Implementations

```
// implementation by a hash table
public class HashMap<K,V> ... implements Map<K,V>, ... {
    public HashMap();
    public HashMap(Map<K,V> m);
}
// implementation by a tree
public class TreeMap<K,V> ... implements SequencedMap<K,V>, ... {
    public TreeMap();
    public TreeMap(Map<K,V> m);
}
// implementation by a hash table and a linked list
public class LinkedHashMap<K,V> ... implements SequencedMap<K,V>, ... {
    public LinkedHashMap();
    public LinkedHashMap(Map<K,V> m);
}
```

Same properties as the corresponding  $\text{Set}\langle E \rangle$  implementations.

## Example

```
// maps words to number of occurrences in their text
Map<String,Integer> map = new TreeMap<String,Integer>();

// reads words from individual lines and puts them in map
while (true) {
    String word = Input.readString();
    if (!Input.isOkay()) break;
    Integer count = map.get(word);
    if (count == null) map.put(word, 1);
    else map.put(word, count+1);
}

// prints words and their occurrences in lexicographic order
for (Map.Entry<String,Integer> entry : map.entrySet())
    System.out.println(entry.getKey() + ":" + entry.getValue());

... // to be continued
```

## Generic Methods

```
public interface Collection<E> extends Iterable<E>
{
    ...
    <T> T[] toArray(T[] a); // copy this collection to a
}

// copy elements of collection c to array a
Collection<String> c = ...
String[] a = c.toArray(new String[c.size()]);
```

Example of a generic method parameterized over types.

## Generic Methods

```
package java.util;
public class Collections {
    ...
    public static <T> boolean addAll(Collection<T> c, T[] a);
}

// add all elements of array a to list l
String[] a = new String[...];
...
List<String> l = new LinkedList<String>();
l.add("word");
Collections.addAll(l, a);
```

Operations that provide a bridge between collections and arrays.

## Algorithms

Generic methods in class `java.util.Collections`.

```
// reverses the order of the elements in the list  
public static <T> void reverse(List<T> list);
```

```
// replaces all elements of the list by the denoted value  
public static <T> void fill(List<T> list, T value);
```

```
// copies all elements from src into dest replacing original elements  
// if dest is longer than src, the remaining elements remain unchanged  
public static <T> void fill(List<T> dest, List<T> src);
```

```
// swaps in list elements at position i and j  
public static <T> void swap(List<T> list, int i, int j);
```

## Algorithms

```
// returns position of key in list sorted according to natural order of T
public static <T> int binarySearch(List<T> list, T key);
```

```
// returns position of key in list T sorted according comparator c
public static <T> int binarySearch(List<T> list, T key, Comparator<T> c);
```

```
// sorts list in ascending order according to the natural order of T
public static <T> void sort(List<T> list);
```

```
// sorts list in ascending order according to comparator c
public static <T> void sort(List<T> list, Comparator<T> c);
```

```
public interface Comparator<T> {
    // returns -1, 0, 1, if o1 is less than, equal, or greater than o2
    public int compare(T o1, T o2);
}
```

## Example

```
public class GreaterCounter
    implements Comparator<Map.Entry<String,Integer>>
{
    // returns -1, if entry1 has a greater counter than entry2
    public int compare(Map.Entry<String,Integer> entry1,
        Map.Entry<String,Integer> entry2)
    {
        Integer value1 = entry1.getValue();
        Integer value2 = entry2.getValue();
        return value2.compareTo(value1);
    }
}
```

Comparing the values of map entries.

## Example (Continued)

```
// maps words to number of occurrences in their text
Map<String,Integer> map = new TreeMap<String,Integer>();

... // example continued

// create duplicate of entry set as a list
List<Map.Entry<String,Integer>> entries =
    new ArrayList<Map.Entry<String,Integer>>(map.entrySet());

// sort list in descending order of the word counters
Collections.sort(entries, new GreaterCounter());

// print words in descending order of the number of their occurrences
for (Map.Entry<String,Integer> entry : entries)
    System.out.println(entry.getKey() + ":" + entry.getValue());
```