

Session 08

Collections

(<http://docs.oracle.com/javase/tutorial/collections/index.html>)

Objectives

- Collections Framework (package `java.util`):
 - List: ArrayList, Vector → Duplicates are agreed
 - Set: HashSet, TreeSet → Duplicates are not agreed
 - Map: HashMap, TreeMap
 - Queue: LinkedList, PriorityQueue
 - Deque: LinkedList, ArrayDeque

The Collections Framework

- The Java 2 platform includes a new *collections framework*.
- A *collection* is an object that represents a group of objects.
- The Collections Framework is a unified architecture for representing and manipulating collections.
- The collections framework as a whole is not threadsafe.

The Collections Framework...

- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

Collection Interfaces

- java.lang.**Iterable**<T>
 - java.util.**Collection**<E>
 - java.util.**List**<E>
 - java.util.**Queue**<E>
 - java.util.**Deque**<E>
 - java.util.**Set**<E>
 - java.util.**SortedSet**<E>
 - java.util.**NavigableSet**<E>
- java.util.**Map**<K,V>
 - java.util.**SortedMap**<K,V>
 - java.util.**NavigableMap**<K,V>

Methods declared in these interfaces can work on a list containing elements which belong to arbitrary type. T: type, E: Element, K: Key, V: Value

Details of this will be introduced in the topic Generic

3 types of group:

List can contain duplicate elements

Set can contain distinct elements only

Map can contain pairs <key, value>. Key of element is data for fast searching

Queue, Deque contains methods of restricted list.

Common methods on group are: Add, Remove, Search, Clear,...

Common Methods of the interface Collection

Method	Description	
<code>add(Object x)</code>	Adds x to this collection	<p>Elements can be stored using some ways such as an array, a tree, a hash table.</p> <p>Sometimes, we want to traverse elements as a list → We need a list of references → Iterator</p>
<code>addAll(Collection c)</code>	Adds every element of c to this collection	
<code>clear()</code>	Removes every element from this collection	
<code>contains(Object x)</code>	Returns true if this collection contains x	
<code>containsAll(Collection c)</code>	Returns true if this collection contains every element of c	
<code>isEmpty()</code>	Returns true if this collection contains no elements	
<code>iterator()</code>	Returns an Iterator over this collection (see below)	
<code>remove(Object x)</code>	Removes x from this collection	
<code>removeAll(Collection c)</code>	Removes every element in c from this collection	
<code>retainAll(Collection c)</code>	Removes from this collection every element that is not in c	
<code>size()</code>	Returns the number of elements in this collection	
<code>toArray()</code>	Returns an array containing the elements in this collection	

The Collection Framework...

Central Interfaces

- `java.util.Collection<E>`
 - `java.util.List<E>`
 - `java.util.Queue<E>`
 - `java.util.Deque<E>`
 - `java.util.Set<E>`
 - `java.util.SortedSet<E>`
 - `java.util.NavigableSet<E>`
 - `java.util.Map<K,V>`
 - `java.util.SortedMap<K,V>`
 - `java.util.NavigableMap<K,V>`

Common Used Classes

- `java.util.ArrayList<E>`
- `java.util.Vector<E>`
- `java.util.HashSet<E>`
- `java.util.TreeSet<E>`
- `java.util.HashMap<K,V>`
- `java.util.TreeMap<K,V>`

Store: Dynamic array
Use index to access an element.

Store: Specific structure/tree
Use iterator to access elements

java.lang.Comparable interface

keySet()
values()

Use
iterator

A TreeSet will stored elements using ascending order. Natural ordering is applied to numbers and lexicographic (dictionary) ordering is applied to strings.

If you want a TreeSet containing your own objects, you must implement the method `compareTo(Object)`, declared in the Comparable interface.

Lists

- A List keeps its elements in the order in which they were added.
- Each element of a List has an index, starting from 0.
- Common methods:
 - **void add(int index, Object x)**
 - **Object get(int index)**
 - **int indexOf(Object x)**
 - **Object remove(int index)**

Classes Implementing the interface List

- ArrayList
- Vector (like ArrayList but it is **synchronized**)
- LinkedList: *linked lists can be used as a stack, queue, or double-ended queue (deque)*

List Implementing Classes

```
Vector vec = new Vector();
for (int i = 101; i <= 110; i++) {
    vec.add(i);
}
for (int i = 0; i < vec.size(); i++) {
    System.out.println(vec.get(i));
}
//or using Iterator
/*
    Iterator iter = vec.iterator();
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }
*/
```

Using the Vector class

java.util.**Vector**<E> (implements java.lang.Cloneable,
java.util.List<E>, java.util.RandomAccess, java.io.Serializable)

The Vector class is obsolete from Java 1.6 but it is still introduced because it is a parameter in the constructor of the javax.swing.JTable class, a class will be introduced in GUI programming.

```
import java.util.Vector;
class Point {
    int x,y;
    Point() { x=0; y=0; }
    Point(int xx, int yy) {
        x=xx; y=yy;
    }
    public String toString() { return "[" + x + "," + y + "];"}
}
public class UseVector {
    public static void main(String[] args) {
        Vector v = new Vector();
        v.add(15);
        v.add("Hello");
        v.add(new Point());
        v.add(new Point(5,-7));
        System.out.println(v);
        v.remove(2);
        System.out.println(v);
        for (int i=0;i<v.size();i++) System.out.print(v.get(i) + ", ");
        System.out.println();
    }
}
```

Output - Chapter08 (run)

run:

[15, Hello, [0,0], [5,-7]]

[15, Hello, [5,-7]]

15, Hello, [5,-7],

Sets

- Lists are based on an ordering of their members. Sets have no concept of order.
- A Set is just a cluster of references to objects.
- Sets may **not** contain **duplicate** elements.
- Sets use the `equals()` method, not the `==` operator, to check for duplication of elements.

```
void addTwice(Set set) {
    set.clear();
    Point p1 = new Point(10, 20);
    Point p2 = new Point(10, 20);
    set.add(p1);
    set.add(p2);
    System.out.println(set.size());
}
```

will print out 1, not 2.

Sets...

- Set extends Collection but does not add any additional methods.
- The two most commonly used implementing classes are:
 - **TreeSet**
 - Guarantees that the sorted set will be in ascending element order.
 - $\log(n)$ time cost for the basic operations (add, remove and contains).
 - **HashSet**
 - Constant time performance for the basic operations (add, remove, contains and size).

TreeSet and Iterator

- Ordered Tree – Introduced in the subject Discrete Mathematics
- Set: Group of different elements
- TreeSet: Set + ordered tree, each element is called as node
- Iterator: An operation in which references of all nodes are grouped to make a linked list. Iterator is a way to access every node of a tree.
- Linked list: a group of elements, each element contains a reference to the next

TreeSet = Set + Tree

The result may be:

```
Random r = new Random();
TreeSet myset = new TreeSet();
for (int i = 0; i < 10; i++) {
    int number = r.nextInt(100);
    myset.add(number);
}
//using Iterator
Iterator iter = myset.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

7
27
36
41
43
46
49
57
75
83

Using the TreeSet class & Iterator

```
import java.util.TreeSet;
import java.util.Iterator;
public class UseTreeSet {
    public static void main (String[] args){
        TreeSet t= new TreeSet();
        t.add(5); t.add(2); t.add(9);t.add(30); t.add(9);
        System.out.println(t);
        t.remove(9);
        System.out.println(t);
        Iterator it= t.iterator();
        while (it.hasNext())
            System.out.print(it.next() + ", ");
        System.out.println();
    }
}
```

Output - Chapter08 (run)

run:
[2, 5, 9, 30]
[2, 5, 30]
2, 5, 30,

A TreeSet will stored elements using ascending order. Natural ordering is applied to numbers and lexicographic (dictionary) ordering is applied to strings.

If you want a TreeSet containing your own objects, you must implement the method `compareTo(Object)`, declared in the `Comparable` interface.

Hash Table

- In array, elements are stored in a contiguous memory blocks → Linear search is applied → slow, binary search is an improvement.
- Hash table: elements can be stored in a different memory blocks. The index of an element is determined by a function (hash function) → Add/Search operation is very fast ($O(1)$).



The hash function f may be:

`'S'*10000+'m'*1000+'i'*100+'t'*10+'h' % 50`

49	
14	Brown
9	Hoa
5	Smith
0	Line1

HashSet = Set + Hash Table

```
Random r = new Random();  
HashSet myset = new HashSet();  
for (int i = 0; i < 10; i++) {  
    int number = r.nextInt(100);  
    myset.add(number);  
}  
//using Iterator  
Iterator iter = myset.iterator();  
while (iter.hasNext()) {  
    System.out.println(iter.next());  
}
```

The result may be:



84
55
7
76
77
95
94
12
91
44

HashSet or TreeSet?

- If you care about iteration order, use a Tree Set and pay the time penalty.
- If iteration order doesn't matter, use the higher-performance Hash Set.

How to TreeSet ordering elements?

- Tree Sets rely on all their elements implementing the interface `java.lang.Comparable`.

`public int compareTo(Object x)`

- Returns a positive number if the current object is “greater than” x, by whatever definition of “greater than” the class itself wants to use.

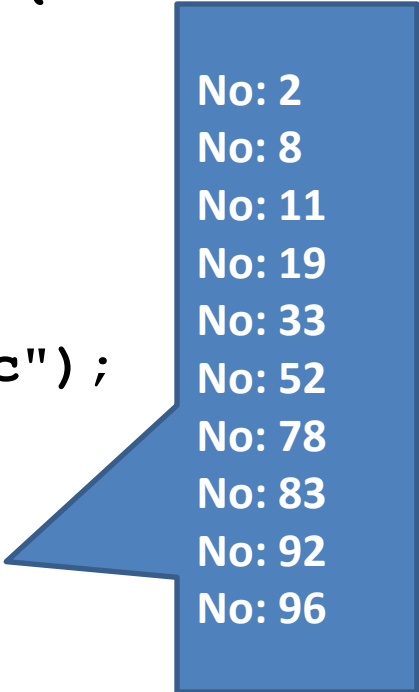
How to TreeSet ordering elements?

```
class Student implements Comparable{  
    int no;  
    ...  
    public int compareTo(Object o) {  
        Student st = (Student) o;  
        if(no > st.getNo())  
            return 1;  
        else if(no == st.getNo())  
            return 0;  
        else  
            return -1;  
    }  
    . . .  
}
```

Comparing 2 students
based on their IDs (
field no)

How to TreeSet ordering elements?

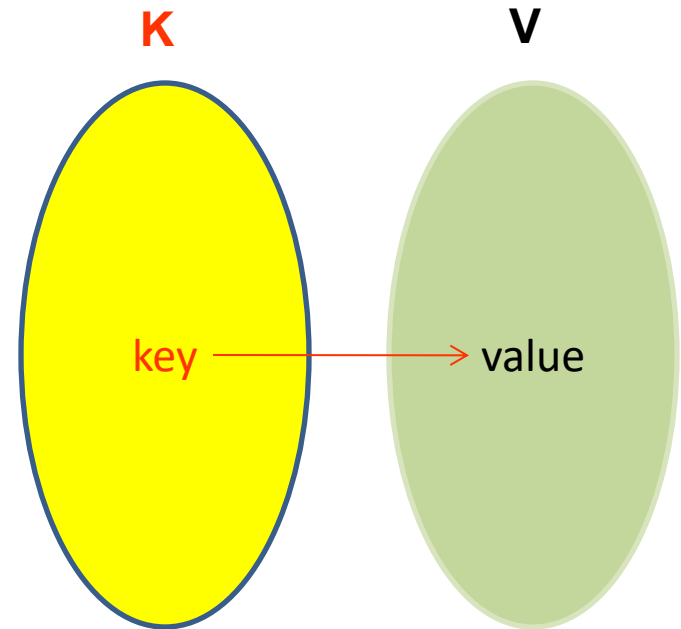
```
public static void main(String[] args) {  
    Random r = new Random();  
    TreeSet myset = new TreeSet();  
    for (int i = 0; i < 10; i++) {  
        int no = r.nextInt(100);  
        Student st = new Student(no, "abc");  
        myset.add(st);  
    }  
    //using Iterator  
    Iterator iter = myset.iterator();  
    while (iter.hasNext()) {  
        Student st = (Student)iter.next();  
        System.out.println("No: " + st.getNo());  
    }  
}
```



No: 2
No: 8
No: 11
No: 19
No: 33
No: 52
No: 78
No: 83
No: 92
No: 96

Maps

- Map doesn't implement the `java.util.Collection` interface.
- A Map combines *two* collections, called keys and values.
- The Map's job is to associate exactly one value with each key.
- A Map like a dictionary.
- Maps check for key uniqueness based on the `equals()` method, not the `==` operator.
- IDs, Item code, roll numbers are keys.
- The normal data type for keys is `String`.



Each element: `<key,value>`

Maps..

- Java's two most important Map classes:
 - HashMap (mapping keys are unpredictable order – hash table is used, hash function is pre-defined in the Java Library).
 - TreeMap (mapping keys are natural order)-> all keys must implement Comparable (a tree is used to store elements).

HashMap

```
public static void main(String[] args) {
    HashMap mymap = new HashMap();
    mymap.put(1, "One");
    mymap.put(2, "Two");
    mymap.put(3, "Three");
    mymap.put(4, "Four");
    //using Iterator
    Iterator iter = mymap.keySet().iterator();
    while (iter.hasNext()) {
        Object key = iter.next();
        System.out.println(key + ": " + mymap.get(key));
    }
}
```

//output
1: One
2: Two
3: Three
4: Four

Key: integer, value: String

Using HashMap class & Iterator

```

1 import java.util.HashMap;
2 import java.util.Iterator;
3 public class UseHashMap {
4     public static void main(String[] args){
5         HashMap h = new HashMap();
6         h.put("Sáu Tấn", "Huỳnh Anh Tuấn");
7         h.put("Bình Gà", "Nguyễn Tấn Sầu");
8         h.put("Ba Địa", "Trần Mai Hoà");
9         System.out.println(h);
10        h.put("Sáu Tấn", "Nguyễn Văn Tuấn");
11        System.out.println(h);
12        h.remove("Bình Gà");
13        System.out.println(h);
14        Iterator it = h.keySet().iterator();
15        while (it.hasNext())
16        { String key= (String)(it.next());
17          String value = (String)(h.get(key));
18          System.out.println(key + ", " + value);
19        }
20    }
21 }

```

Key: String, value: String

Output - Chapter08 (run)

```

run:
{Ba Địa= Trần Mai Hoà, Sáu Tấn=Huỳnh Anh Tuấn, Bình Gà=Nguyễn Tấn Sầu}
{Ba Địa= Trần Mai Hoà, Sáu Tấn=Nguyễn Văn Tuấn, Bình Gà=Nguyễn Tấn Sầu}
{Ba Địa= Trần Mai Hoà, Sáu Tấn=Nguyễn Văn Tuấn}
Ba Địa, Trần Mai Hoà
Sáu Tấn, Nguyễn Văn Tuấn
BUILD SUCCESSFUL (total time: 1 second)

```

Interface Queue and Deque



- Interfaces for restricted list (limited manipulation), programmers can not access an arbitrary element but elements at the beginning or the end of the list only.
- **Deque**: A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually **pronounced "deck"**. Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

Interface Queue



public interface **Queue**<E> extends [Collection](#)<E>

boolean [add](#)([E](#) e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an `IllegalStateException` if no space is currently available.

[E](#) [element](#)() Retrieves, but does not remove, the head of this queue.

boolean [offer](#)([E](#) e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

[E](#) [peek](#)() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

[E](#) [poll](#)() Retrieves and removes the head of this queue, or returns null if this queue is empty.

[E](#) [remove](#)() Retrieves and removes the head of this queue.

Classes:

- `java.util.AbstractQueue<E>` (implements `java.util.Queue<E>`)
 - `java.util.PriorityQueue<E>` (implements `java.io.Serializable`)

Interface Deque...



public interface **Deque**<E> extends [Queue](#)<E>

IN addition to methods inherited from the interface Queue, some methods are declared:

Summary of Deque methods

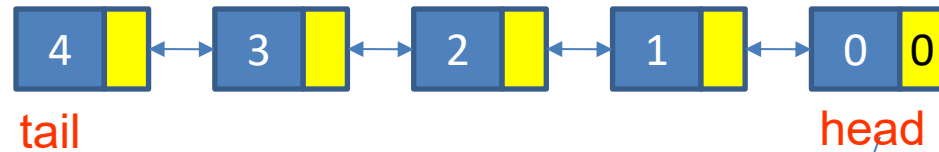
	First Element (Head)		Last Element (Tail)	
	<i>Throws exception</i>	<i>Special value</i>	<i>Throws exception</i>	<i>Special value</i>
Insert	<code>addFirst(e)</code>	<code>offerFirst(e)</code>	<code>addLast(e)</code>	<code>offerLast(e)</code>
Remove	<code>removeFirst()</code>	<code>pollFirst()</code>	<code>removeLast()</code>	<code>pollLast()</code>
Examine	<code>getFirst()</code>	<code>peekFirst()</code>	<code>getLast()</code>	<code>peekLast()</code>

Classes

java.util.[LinkedList](#)<E> (implements java.lang.[Cloneable](#), java.util.[Deque](#)<E>, java.util.[List](#)<E>, java.io.[Serializable](#))

java.util.[ArrayDeque](#)<E> (implements java.lang.[Cloneable](#), java.util.[Deque](#)<E>, java.io.[Serializable](#))

Queue/Deque Demo.



```

import java.util.LinkedList;
public class DequeDemo {
    public static void main(String args[]){
        int N=5;
        // 3 list are the same
        LinkedList list1= new LinkedList();
        LinkedList list2= new LinkedList();
        LinkedList list3= new LinkedList();
        for (int i=0; i<N; i++) {
            list1.add(i); list2.add(i); list3.add(i);
        }
        // Access list1 as a queue
        while(!list1.isEmpty()) System.out.print(list1.remove() + ",");
        System.out.println();
        // Access list2 from it's head
        while(!list2.isEmpty()) System.out.print(list2.removeFirst()+ ",");
        System.out.println();
        // Access list2 from it's tail
        while(!list3.isEmpty()) System.out.print(list3.removeLast()+ ",");
        System.out.println();
    }
}

```

run:

0,1,2,3,4,
0,1,2,3,4,
4,3,2,1,0,

Summary

- The Collections Framework
 - The ***Collection*** Super interface and Iteration
 - Lists
 - Sets
 - Maps
 - Support Classes
 - Collections and Code Maintenance