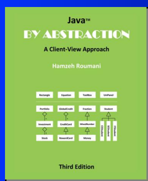


Chapter 10

The Collection Framework

Outline

10.1.1 The Interfaces

List	Set	Map

`add(element)`
`remove(element)`
`get(index)`
`iterator()`
	`add(element)`
	`remove(element)`
	`iterator()`
	`...`
		`add(key, value)`
		`remove(key)`
		`get(key)`
		`keySet(): Set`

Sequence	**Set**	**Pairs**
Duplicates are OK and the positional order is significant	Duplicates are not allowed and order is insignificant	A pair is (key,value) where key is unique

10.1.2 The Classes

List ○	Set ○	Map ○
add(element) remove(element) get(index) iterator()	add(element) remove(element) iterator() ...	add(key, value) remove(key) get(key) keySet(): Set

ArrayList LinkedList	HashSet TreeSet	HashMap TreeMap

The two classes that implement each interface are equivalent in the client's view. The only visible diff is performance (running time).

 Java By Abstraction 10-4

The Classes, cont.

ArrayList LinkedList	HashSet TreeSet	HashMap TreeMap

- Declare using the interface, not the class
- Use LinkedList only if your app tends to add or remove elements at index 0
- Use TreeSet/Map only if you want to keep the elements sorted
- Specify the type of the elements that you intend to store in the collection

Example: A list of strings

```
List<String> bag = new ArrayList<String>();
```

 Java By Abstraction 10-5

10.1.3 Revisiting Generics

All classes in the framework support generics. By specifying the type (between < and >) the client ensures:

- No rogue element can be inserted
- No casting is needed upon retrieval

Example:

```
List<Stock> bag = new ArrayList<Stock>();
// bag.add("Hello"); will not compile!
bag.add(new Stock(".ab"));
Stock s = bag.get(0); // no cast!
```

 Java By Abstraction 10-6

10.2.1 API Highlights

- Use add to add elements to lists and sets:

```
List<Date> list = new ArrayList<Date>();
Set<String> set = new HashSet<String>();
list.add(new Date());
set.add("Hello");
```

- Use put to add an element to a map

```
Map<Integer, String> map;
map = new HashMap<Integer, String>();
map.put(55, "Clock Rate");
```


API Highlights

- Use remove to delete from lists and sets:

```
boolean done = set.remove("Adam");
```

Note that remove returns `false` if the specified element was not found and returns `true` otherwise.

- To delete a map element given its key:

```
String gone = map.remove(55);
```

Note that remove in maps returns the value of the element that was removed or null if the specified key was not found.

API Highlights

The elements of lists are indexed (starting from 0). Hence, but only for lists, we can also add and delete based on the position index:

- To insert x at position 5:

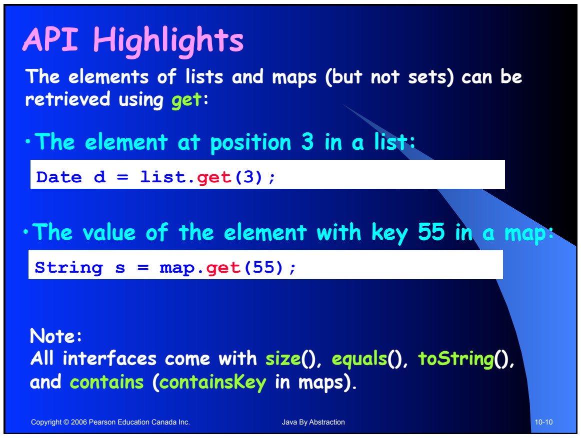
```
list.add(5, x);
```

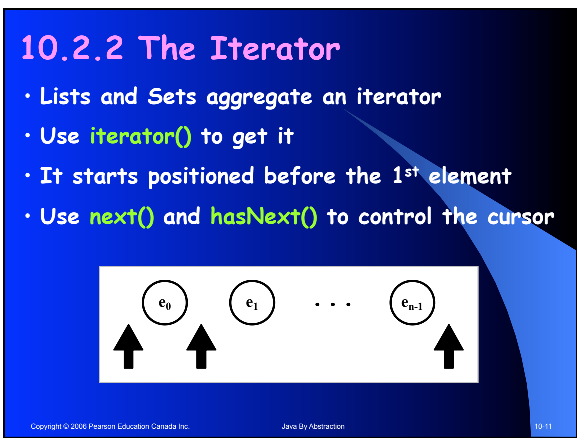
This will work only if the list has at least 5 elements, and it will adjust the indices of all elements after position 5, if any.

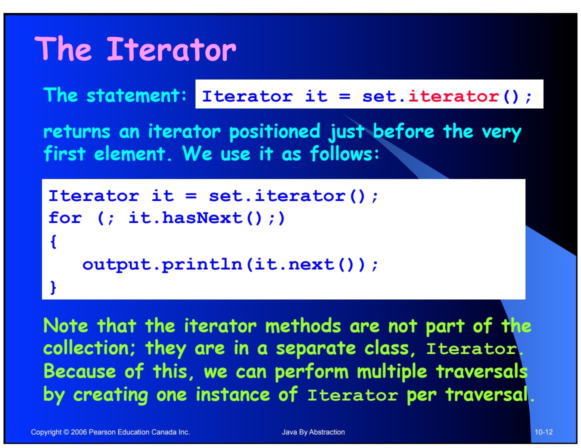
- To delete the element at position 5:

```
list.remove(5);
```

This will work only if the list has at least 6 elements.


API Highlights
The elements of lists and maps (but not sets) can be retrieved using get:
•The element at position 3 in a list:
Date d = list.get(3);
•The value of the element with key 55 in a map:
String s = map.get(55);
Note:
All interfaces come with size(), equals(), toString(), and contains (containsKey in maps).
Copyright © 2006 Pearson Education Canada Inc.
Java By Abstraction
10-10


10.2.2 The Iterator
• Lists and Sets aggregate an iterator
• Use iterator() to get it
• It starts positioned before the 1st element
• Use next() and hasNext() to control the cursor
e0
e1
. . .
en-1
Copyright © 2006 Pearson Education Canada Inc.
Java By Abstraction
10-11


The Iterator
The statement: Iterator it = set.iterator();
returns an iterator positioned just before the very first element. We use it as follows:
Iterator it = set.iterator();
for (; it.hasNext();)
{
output.println(it.next());
}
Note that the iterator methods are not part of the collection; they are in a separate class, Iterator. Because of this, we can perform multiple traversals by creating one instance of Iterator per traversal.
Copyright © 2006 Pearson Education Canada Inc.
Java By Abstraction
10-12

The Iterator and Generics

The Iterator class supports generics; i.e. we can obtain a type-aware iterator as follows:

```
Iterator<String> it = set.iterator();
```

To benefit from this, let us rewrite the loop of the previous slide so it prints the elements capitalized:

```
Iterator<String> it = set.iterator();
for (; it.hasNext();)
{
   String tmp = it.next();
   output.println(tmp.toUpperCase());
}
```


The Iterator in Maps

The Map interface has no iterator() method but we can obtain a set of the map's keys:

```
public Set<K> keySet()
```

And by iterating over the obtained set, we can, in effect, iterate over the map's elements:

```
Iterator<Integer> it = map.keySet().iterator();
for (; it.hasNext();)
{
   int key = it.next();
   String value = map.get(key);
   output.println(key + " --> " + value);
}
```


10.2.3 Searching and Sorting

Searching

One simple (albeit inflexible) way to search a collection is to use the contains method (containsKey in maps). It determines if an element in the collection is equal to a given value and returns true or false accordingly.

```
output.print("Enter a word to look for: ");
String lookFor = input.nextLine();
output.println(set.contains(lookFor));

output.print("Enter a key to look for: ");
int findMe = input.nextInt();
output.println(map.containsKey(findMe));
```

Searching, cont.

For applications that require more than a simple yes/no, we use traversal-based searches. For example, find out if a given key is present in a map and output its value:

```
output.print("Enter a key to look for: ");
int find = input.nextInt();
Iterator<Integer> it = map.keySet().iterator();
boolean found = false;
Integer key = null;
for (; it.hasNext() && !found;)
{
   key = it.next();
   found = key.equals(find);
}
if (found) output.println(map.get(key));
```


Sorting

The Collections class has the method:

```
static void sort(List<T> list)
```

It rearranges the elements of the list in a non-descending order. It works if, and only if, the elements are comparable; i.e. one can invoke the compareTo method on any of them passing any element as a parameter.

Recall that compareTo (in String) returns an int whose sign indicates < or > and whose 0 value signals equality.

Sorting, cont.

To ensure that compareTo can be invoked, we require that T (the element's class) implements Comparable<T>, an interface with only one method: compareTo(T).

Note:
Requiring that T implements Comparable<T> is too strong. It is sufficient if T extends some class S that implements Comparable<S>. The sort method states this requirement in its API as follows:

```
<T extends Comparable<? super T>>
```

A Sorting Example:

Write a program that creates a list of a few Fractions and then sort them.

```
List<Fraction> list;
list = new ArrayList<Fraction>();
list.add(new Fraction(1,2));
list.add(new Fraction(3,4));
list.add(new Fraction(1,3));

output.println(list);
Collections.sort(list);
output.println(list);
```


Sorting Sets and Maps

The sort method accepts only lists. What if we needed to sort a set?

```
Set<Fraction> set;
set = new HashSet<Fraction>();
set.add(new Fraction(1,2));
set.add(new Fraction(3,4));
set.add(new Fraction(1,3));
output.println(set);
```

A minor modification to the above program will make its output sorted …

Sorting Sets and Maps, cont.

Simply use TreeSet instead of HashSet.

The same technique applies to maps: use TreeMap instead of HashMap to keep the map's elements sorted on their keys.

Note:
Using a tree-implementing class for sets and maps is conceptually different from using the sort methods for lists. The former keeps the elements sorted at all times. The latter sort will not persist after adding or removing elements.

Sorting and Binary Search

The main advantage of sorting is speeding up the search. When the elements are sorted, you don't have to visit all of them to determine if a given value is present in the collection or not.

```
int binarySearch(List list, T value)
```

The method searches for value in list and returns its index if found and a negative number otherwise

Note: Unlike exhaustive search (which is linear), binary search has a complexity of O(lgN).

10.2.4 Summary of Features

LIST	SET	MAP
Adding Elements		
boolean `add(E e)` void `add(int index, E e)`	boolean `add(E e)`	V `put(K key, V value)`
Removing Elements		
boolean `remove(E e)` E `remove(int index)`	boolean `remove(E e)`	V `remove(K key)`
Accessing an Element		
E `get(int index)`	*none*	V `get(K key)`
Searching the Elements		
boolean `contains(E o)`	boolean `contains(E o)`	boolean `containskey(K key)`
Traversing the Elements		
Iterator `iterator()` *invoke on it:* E `next()` boolean `hasNext()`	Iterator `iterator()` *invoke on it:* E `next()` boolean `hasNext()`	Iterator `keySet().iterator()` *invoke on it:* E `next()` boolean `hasNext()`
Other methods (available in all three interfaces)		
equals, size, toString		
Algorithms for lists only (`static` methods in the `Collections` class)		
binarySearch, copy, fill, reverse, shuffle, sort		

List-Only Utilities in `Collections`

- **Sort / binarySearch**
  We covered these
- **shuffle**
  Rearranges the elements randomly
- **reverse**
  Rearranges the elements in reverse order
- **copy**
  Returns a deep copy of the collection
- **fill**
  Populates all the elements with a given value

10.3 Applications

Read the three applications in sections 10.3.1-3.

Here, we will outline five different applications that utilize various features of the Collection Framework:

Example 1

- Create a set of random dates.
- Let all of them be in this year and set their times to 0 (h, min, se,ms).
- Make the set sorted
- Serialize it as RanDateA.dat
- Make a second in RanDateB.dat

Example 2

- Write an app that de-serializes the two files created in Example 1.
- Print the two sets side by side (in two columns)

Example 3

- Write an app that determines the common dates in the above two files.
- The app should generate a sorted intersection set.

Hint: use `contains`

Example 4

- Let the sizes of the above two sets be n and m. Assuming that the contains method has a linear complexity, what is the complexity of your app?
- Can you rewrite the app to make it linear with complexity O(n+m)?

Here is a problem of a similar nature:
Find the celebrity in a room of n persons. Everyone knows the celebrity but s/he does know anyone. You are allowed to ask n questions of the form: "do you know that person?"

Example 5

This application uses the Supplier and Item classes of type.lib.

As a demonstration, the following fragment creates a supplier named Loblaws with address Toronto, and an item named Corn Flakes with item number df102 and price $1.75:

```
Supplier s = new Supplier("Loblaws", "Toronto")
Item i = new Item("df102","Corn Flakes",1.75));
```

The following fragment creates a map and store the above supplier/item pair in it:

```
Map map = new TreeMap();
map.put(supplier, item);
```

Example 5, cont.

- Create a map<Supplier, Item> containing the following supplier, item pairs:

SUPPLIER & ADDRESS		ITEM#	DESCRIPTION	PRICE
Loblaws	Toronto	df102	Corn Flakes	1.75
Dominion	Toronto	df453	Lindt Chocolate	5.75
Loblaws	Toronto	df102	Corn Flakes	1.75
IGA	Markham	ef777	Ice Cream	3.25
IGA	Maple	df102	Corn Flakes	1.75

- Output the map using its default toString method. How come it has 5 elements (even when two of the suppliers are the same)?

Copyright © 2006 Pearson Education Canada Inc. Java By Abstraction 10-31

Example 5, cont.

- Output the map using an iterator over the keys.
- Create a "reversed" or map with all the the distinct items as keys. For each, the value is a list of suppliers who supply this item.
- Output the inverted map using an iterator over its keys and an indexed iterator over the supplier list of each item.

Copyright © 2006 Pearson Education Canada Inc. Java By Abstraction 10-32