Java By Abstraction: Chapter 10

The Collection Framework

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

Review

- Collection: an aggregate that can hold a varying number of elements
- Interface: an entity that defines mandatory features (methods, attributes, etc.) of all classes that implement it

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

Duplicates are OK and the positional order is significant

Set

Duplicates are not allowed and order is insignificant

Pairs

A pair is (key,value) where key is unique

The Classes

```
List ○
add(element)
remove(element)
get(index)
iterator()
...
```

```
Set ○
add(element)
remove(element)
iterator()
...
```

```
Map ○
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).

Which Class to Use?

- ArrayList: good if you want elements to be indexed
- LinkedList: only efficient if elements added to the beginning of the list
- TreeSet or TreeMap: good if you want to access elements in sorted order
- HashSet or HashMap: more efficient than TreeSet/Map if you don't need sorted order

Generics

- Java is strongly typed
  - Object type mismatch checked at compile-time
- Collections prior to Java 1.5:
  - Create for elements of a specific type
  - Create to contain elements of type Object, then cast
- Java 1.5 introduced generics:
  - Specify element type in angled brackets
    e.g.: <String>

Generics in the API

- Generics allows the element type to be defined at compilation
- API needs a placeholder to describe parameter types and/or return types
- Placeholder names are arbitrary
- Typical placeholders: T, E, K, V, P, C
- Example: add(int index, E element)

Declaring and Initializing Collections

- Declare reference using the interface, but initialize object using the class

```
List<String> bag = new ArrayList<String>
```

Using Collections without Generics

- Don't do it
- Generics allows for type checking at compile-time
- With generics:
  - List<String> bag = new ArrayList<String>();
- Without generics:
  - List bag = new ArrayList();
- Compiler warning without generics:
  - Note: *ClassName*.java uses unchecked or unsafe operations.
    Note: Recompile with -Xlint:unchecked for details.

API Highlights (1)

- 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 (2)

- 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 (3)

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 (4)

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);
```

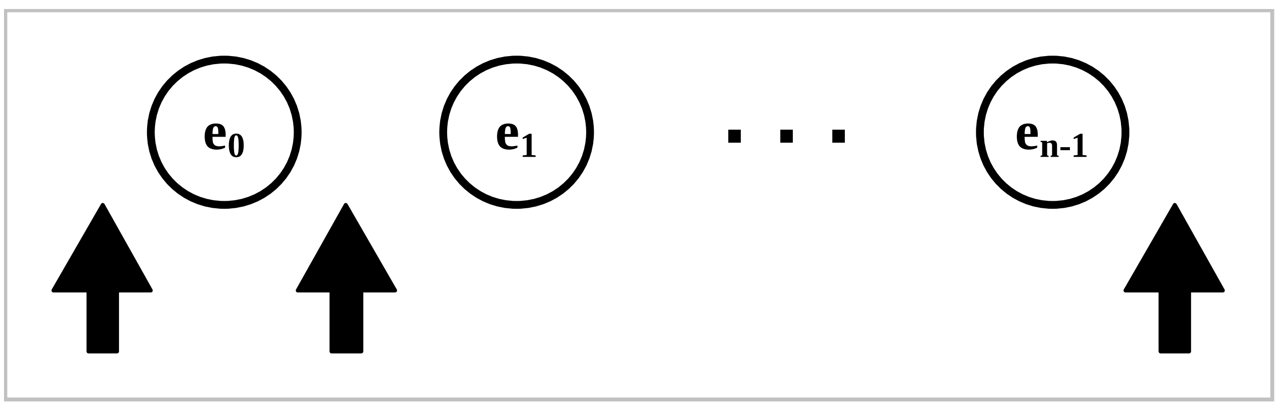
- The value of the element with key 55 in a map:

```
String s = map.get(55);
```

Note:
All interfaces come with size(), equals(), toString(), and contains (containsKey in maps).

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



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());
}
```

Iterating over a Map

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);
}
```

Sorting Collections

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.

Implementing Comparable

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>>
```

Sorting a List Collection

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 non-List Collections

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 non-List Collections

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.

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).

Applications

- Read, study, and work through the application exercises in section 10.3