The java collection framework

Pieter van den Hombergh
Richard van den Ham

Fontys Hogeschool voor Techniek en Logistiek

February 8, 2018

Collection Zoo

The basic collections, well known in programming frameworks are:

LIST An ordered collection of elements. The elements are ordinarily ordered in **insertion** or **add** order.

STACK A Last in First out ordered collection. The collection can only be access at the "top". You know that one already.

QUEUE A First in First Out ordered data structure. The elements are added (typically called tail) only at one end and can be retrieved from the other end (called head). (NL:Wachtrij, DE:Warteschlange).

Collection Zoo 2

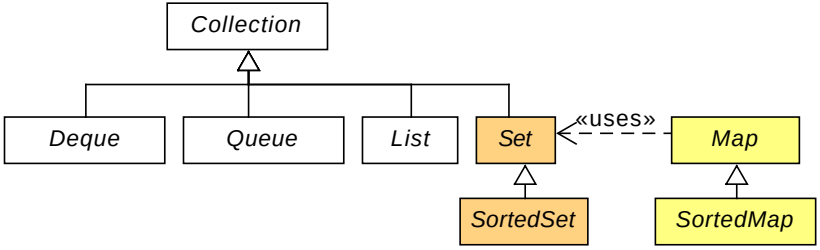
BAG An unordered collection. Things can go in and come out. The bag provides no stricter constraints on storage.

SET A collection in which every element is unique. It is mostly comparable to a mathematical set (NL:Verzameling, DE:Menge). A set will typically provide method that support the mathematical concepts of union, difference and add.

MAP otherwise known as a dictionary. A map provides a "mapping" from a key to a value, where you can use the key to look up (get) the value from the map. By definitions, the keys in a MAP form a set. Typically this key-set can be obtained from the map.

Java Collection Framework

- A collections framework is a unified architecture for representing and manipulating collections. Collections frameworks contain the following: interfaces, implementations, algorithms.
- The top most interface is `Collection`. This interface declares almost all operations[1] of all collections, leaving some of the operations optional.
  - For one it declares that all collections should be **Iterable**, which allows the use of the for-each construct as in `for(E e : collection)`.
- The core collection interfaces



[1] methods

Collection methods and iterators

Methods

- The Collection interface provides basic adding and removing operations like: `addAll`, `add`, `remove`, `removeAll`
- The interface provides query operations like: `size`, `contains`, `isEmpty`
- The interface provide the `toArray()` method
- Also the `forEach()` method is provided, instead of using an iterator you can apply lambda expressions to iterate over the collection.

Iterators

- Each collection is `Iterable`
- Obtain an iterator, example: `Iterator<> iterator = collection.iterator();`

Traversing Collections

There are three ways to traverse collections:

using aggregate operations

```
list.forEach(e -> System.out.println(e));
```

with the for-each construct

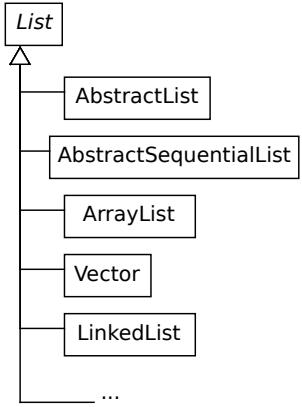
```
for (Object o : c) {
  System.out.println(o);
}
```

using Iterators

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

Rich set of implementations

- Interfaces have multiple implementations, because one cannot always make "one size to fit all".
- E.g. The `List` has the following implementing classes: **`AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`, `LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, and `Vector`**.



[2] If you implemented it as is shown in our examples

Which implementation is the best for you?

Some criteria (incomplete list):

- Thread Safety
- Access Method
- Performance
- Functionality

Read the documentation, which gives you important information like:

- `AbstractList`: skeletal implementation of random access list
- `AbstractSequentialList`: skeletal implementation of sequential access list
- `ArrayList` vs `Vector`: Vector is thread safe. Different resizing strategies
- `CopyOnWriteArrayList`: thread safe variant of ArrayList.
- `ArrayList` vs `LinkedList`: random vs sequential acces

Example

Suppose, you have a `Collection <String> c`, which may be a List, a Set, or another kind of Collection.

```
Collection<String> c = new HashSet<>();
```

This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

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

Symbiosis between Set and Map

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. The keys of a Map are stored in a Set, which guarantees the uniqueness of the keys.

Java platform contains three general purpose implementations of the Map interface:

- `HashMap` which (doesn't retain insertion order),
- `LinkedHashMap` (iteration order == insertion order)
- `TreeMap` iteration order is sorted order by key (using a comparator or natural order.).

Their behavior and performance are precisely analogous to HashSet, LinkedHashSet and TreeSet which are used to store the keys.

The Comparator interface

The Comparator interface can be used to compare the objects of a class that doesn't implement the Comparable interface. Demo example SimpleCollection, see course website.

Comparator:

```
public class AreaComparator implements Comparator<GeometricObject> {
     /* returns -1 if area of o1 is smaller than area of o2
        returns 1 if area of o1 is greater than area of o2
        returns 0 in all other cases */
    @Override
    public int compare(GeometricObject go1, GeometricObject go2) {
        if (go1.getArea() < go2.getArea()) return -1;
        if (go1.getArea() > go2.getArea()) return 1;
        return 0;
    }
}
```

Example call:

```
        Circle c = new Circle(10);
        Rectangle r = new Rectangle(5,5);
        AreaComparator ac = new AreaComparator();
        if (ac.compare(r, c)<0) {System.out.println("smaller");}
```

The Comparable interface

In case you are designer of the class you could use the Comparable interface.
Comparable:

```
public class Rectangle implements GeometricObject, Comparable<↩
    GeometricObject> {

    @Override
    public int compareTo(GeometricObject go) {
        // still to implement
        if (getArea() < go.getArea()) return -1;
        if (getArea() > go.getArea()) return 1;
        return 0;
```

Example call:

```
        Rectangle r1 = new Rectangle(5,5);
        Rectangle r2 = new Rectangle(3,7);
        if (r1.compareTo(r2)<0) {System.out.println("smaller");}
```

What's the difference? Compare Comparable to Comparator!

Links to more examples, book chapters

Example with java 8 aggregate functions.
See: http://docs.oracle.com/javase/tutorial/collections/interfaces/map.html

For more easy to understand collection and stream examples see:
http://www.mkyong.com/java8/java-8-foreach-examples

Book by Y. Daniel Lang.
Chapter 20, paragraphs 20.1-20.6 Lists, Stacks, Queues and Priority Queues

Questions and links

Not all understood?
Study
Java collection Turorial and

Questions?

Questions or remarks?