ESS 201 Programming II Java
Term 1, 2018-19

Collections

T K Srikanth
International Institute of Information Technology, Bangalore

Collections

A unified architecture for representing and manipulating different kinds of “collections”

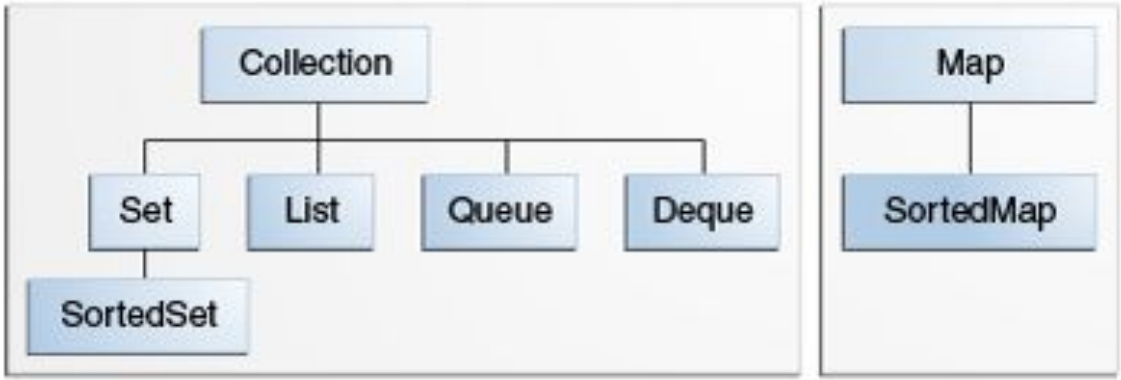
Defines:

- Interfaces: of common collections mechanisms
- Implementations: concrete implementations of some common types
- Algorithms: implementations of useful operations that work on all collections

All these are defined in a *generic* way, so that they can be used for collections of any specific type of objects, and still provide strong type checking.

Thus, we can get re-use of API’s, efficient implementations and algorithms,

The Collections interfaces



All these are ***interfaces***

Each provides a specific kind of functionality, and adds new interfaces to that of the parent interface

Collections Interface

- Collection: root of one hierarchy. Defines most common interfaces of all collections of elements of a given type: size, add, remove, iterate etc
- Set: Collection that has no duplicates
- List: Collection with implied order, and notion of “position” or “index” of an element
- Queue: general notion of queue with control over order of placement and removal of elements, and additional methods for placement, removal etc.
- Deque: Can add and insert at both ends. Can be used as a stack (LIFO) or normal queue (FIFO)
- Map: maps keys to values, stores key-value pairs (e.g. hash table). Cannot hold duplicate keys

Collection interfaces

Common methods:

```
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element);
boolean remove(Object element);
void clear();
```

Also methods such as containsAll, removeAll, addAll, retainAll,

Collection interfaces

Can create an instance of a collection from any other collection

```
Collection<String> c = ....

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

(or)

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

Will create a list of Strings, initialized with the elements in c

(Note that the contents of the result and input may not be the same nor the same order - for example creating a Set from a List

Collections and arrays

Arrays are not Collections

However, convenience methods allow you to convert back and forth

```
Object[] a = c.toArray();
```

or , if you know the type of c, and want to make the array type explicit:

```
String[] a = c.toArray();
```

To initialize an existing array

```
String[] a = c.toArray(new String[c.size()]);
```

Static method of Arrays Array.asList returns a List **wrapper** on an existing array

```
List<String> ls = new ArrayList<String>(Arrays.asList(a));
```

Iterating through a collection

Common mechanisms for iterating through the elements of a collection , without needing to know anything about the implementation: whether it is a Set, LinkedList, ArrayList, Map, etc.

for-each:

Given collection c containing elements of type T

```
for(T elem: c) {
    // do something with the element elem
}
```

Useful if we are only processing some aspect of each element and not modifying the c. Note that the order in which elements are visited would depend on the implementation and sub-type of Collection used

Iterators

A general notion of iteration: create an object that encapsulates information about the collection, the current position, the element at the current position and whether we have reached end-of-collection.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Can create an Iterator object for any collection and then use it to walk through the collection. Cannot modify the collection - except can remove (removes the last element that was returned by next()

Iterators

To print out elements of a list of Book (or any collection of Book)

```
for (Iterator<Book> it = c.iterator(); it.hasNext(); ) {
    Book b = it.next();
    System.out.println(b);
}
```

For example, to remove null values from a list c of Book objects:

```
for (Iterator<Book> it = c.iterator(); it.hasNext(); )
    if (it.next() == null)
        it.remove();
```

List Interface

Provides ability to get/set/add elements at a specific location (index)

For example, a generic method to swap elements of any kind of list

(we'll discuss generics later in more detail)

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

List Iterators

ListIterator extends Iterator and provides ways to traverse a List forwards or backwards - adds interfaces hasPrevious() and previous()

```
for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {
    Type t = it.previous();
    ...
}
```

listIterator(list.size()) -- positions iterator at end of list.

Also has nextIndex() and previousIndex() - index of elements that would have been returned by the subsequent calls to next() or previous();

Algorithms

Collections framework contain a useful list of common algorithms (as static methods) that are intended to be robust and efficient

Examples: sort, reverse, swap, binarySearch, ….

For some of these, we need a way to “compare” elements - on the lines of the equals method we saw earlier

Comparing elements of a collection

To make a type comparable, implement the Comparable interface

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

returns -ve, 0, or +ve depending on whether the object is < == or > o

Or, implement a Comparator, and pass an instance of this class to a method such as sort

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Sorting using customized comparators

```
class BankAccount implements Comparable<BankAccount> {
    …
    public int compareTo(BankAccount other) {
        return amount - other.amount;
    }
    private float amount;
}
```

If we have an ArrayList<BankAccount> accounts,
we can now use Collections.sort(accounts);

Customized comparators

If we need options for multiple ways of comparing elements of a collection, we can implement Comparators and pass an instance of the appropriate ones to a sort method.

E.g. imagine we had class Rectangle with length and height, and we wanted to sort sometimes based on length and sometimes based on area.

We implement 2 different Comparators - LengthCompare and AreaCompare which would do the appropriate comparisons

Thus, if we have:

```
ArrayList<Rect> rects;
….
Collections.sort(rects, new LengthCompare());
Or
Collections.sort(rects, new AreaCompare());
```

Customized Comparators

```
public class LengthCompare implements Comparator<Rect> {
    public int compare(Rect r1, Rect r2) {
        return r1.length - r2.length;
    }
}

public class AreaCompare implements Comparator<Rect> {
    public int compare(Rect r1, Rect r2) {
        return r1.getArea() - r2.getArea();
    }



}
```

Common concrete classes on Collections

ArrayList

Stack

LinkedList - also implements interface Queue

PriorityQueue: highest priority element (largest value) will be removed first. Can specify a Comparator in the constructor to control order/priroty

HashSet implements Set

TreeSet implements SortedSet

Maps

Associate keys to values. Keys must be unique: one-to-one or many-to-one maps.

Methods to put(key, value), get(key) to find value for a given key, and containsKey(key) to check if key exists

HashMap implements Map

So, HashMap<Course, Teacher> can keep track of courses that a teacher has(assuming only one teacher per course)

TreeMap implements SortedMap - maintains key-value is sorted order. Uses a red-black tree implementation and guarantees O(log n) for insert/find

Generic Classes and Methods

Mechanism for implementing classes/methods that can work on different classes, but still provide **compile-time** type safety.

E.g. we can implement a generic method printArray that can iterate through and print an array with any specific type of element

Instead of

```
static void printArray(Integer[] ints)
static void printArray(Book[] books)
static void printArray(String[] strings)
```

We can have

```
static <T> printArray(T[] elems)
```

and pass it any of ints, books, strings...

Generic print array - example

```
public static <T> void printArray(T[] arr) {
    for(T elem: arr) {
        System.out.println(elem);
    }
    System.out.println();
}
```

Works on any class T for which toString is defined - i.e. any sub-class of Object!
Note: generic methods can only be defined for non-primitive types. For primitives, use wrapper classes

Bounding the type parameters

What if the generic method implementation uses methods of a certain type - e.g. a method that can compute the average of an ArrayList<Double> or ArrayList<Integer> - in general ArrayList<Number>

I.e. we want a method:

```
static double average(ArrayList<Number> nums) {// assume nums non-zero length
    double sum = 0.0;
    for(number n: nums) {
        sum += n;
    }
    return sum/nums.size();
}
```

Bounding type parameters

What happens when we call this with a list that is of type

ArrayList<Integer> or ArrayList<Double>? Why?

ArrayList<Integer> is not a sub-class of ArrayList<Number>. Why?

Can we pass in an ArrayList<Book>?

Hence, need a way to say that we can pass in any arraylist, so long as the elements are of any sub-type of Number.

```
static double average(ArrayList<? extends Number> nums){ … }
```

Called a wildcard type-parameter. Can also use:

```
static <T extends Number> double average(ArrayList<T> nums) { … }
```

Generic Classes

We can implement classes that can have flexibility in the type of objects they handle. Collections are examples of this - you can have a Set of any type of elements, and be able to apply its methods consistently.

Consider a class Point in 2D. Depending on the context, the coordinates could be in float or integer units (e.g. a continuous space or a pixel-based screen). Yet, most of the operations we perform on these would be “generic” in nature:

- Distance, closest point of a list of points to a given point, etc.

Can we implement this once and re-use it for both scenarios - float and int coordinate spaces?

Generic Classes

```
public class Point<T> {
        Point(T x, T y) { … }
        public static Double dist(Point<T> p2) {  … }
        public Point<T> closest(ArrayList<Point<T>> points) { … }
        …
        private T x, y;
}
```

And use this as

```
Point<Integer> pi = new Point<Integer>(3,4);
Point<Double> pd = new Point<Double>(3.0, 4.3);
```

Note: to be safe, we should strictly define this as

```
        public class Point<T extends Number> { …. }
```