03 - Generics and Collections

< Previous | Next >

§3.1 Collection API

- Set, SortedSet, Map and SortedMap are all interfaces not classes.
- All name-value maps implement java.util.Map and all collections implement java.util.Collection.
- When using the standard collection interfaces some operations may throw an UnsupportedOperationException (unchecked).
- java.util.HashSet allows null value to be stored.
- Sets implementing the SortedSet interface such as TreeSet, keep the elements sorted.
- The order of elements is not defined in HashSet. So while retrieving elements, it can return them in any order. Remember that, TreeSet does store elements in their natural sorted order.
- A List keeps the elements in the order they were added. A List may have non-unique elements.
- A Set keeps unique objects without any order or sorting.
- A SortedSet keeps unique elements in their natural order.
- There are no interfaces like OrderedSet or OrderedList.
- The Java Collections framework consist of all the descendants of both java.util.Collection and java.util.Map.
- Type erasure means that a compiled java class does not contain any of the generic information that is present in the java file.
- Reification is just the opposite of type erasure. Here, all the type information is preserved in the byte code. In Java, arrays are reified.
- Remember that HashMap supports adding null key as well as null values but ConcurrentHashMap does not. Inserting a null key or null in a ConcurrentHashMap will throw a NullPointerException.
- HashMap allows both - the key and the value to be null. But Hashtable does not.
- There is no DuplicateKeyException!
- Collections API has two separate trees of classes/interfaces - java.util.Collection and java.util.Map. A Collection (such as a Set or a List) stores objects, while a Map stores key-value pairs.
- Both LinkedList and PriorityQueue classes implement Queue interface.
- The java.lang.Comparable interface has only one method: `public int compareTo(T o)`. The java.lang.Comparator interface has only one method: `public int compare(T o, T o)`.
- Remember that any time you want to sort a collection of items, you need code that "sorts" the elements. equals method can only tell whether two objects are equal or not but it cannot tell which object comes before and which after. To figure this out, either the objects themselves should be able to tell that (by implementing java.lang.Comparable interface) or you need to use a separate java.util.Comparator instance that can compare the objects. If no Comparator is supplied to the sorted() method, at run time, when the sorted method tries to cast Book object to Comparable, it will fail (throw an exception).
- The List.subList method returns a view backed by the original list. It doesn't change the existing list. Remember that, however, if you modify the sub list, the changes will be visible in the original list.
- Of all the collection classes of the java.util package, only Vector and Hashtable are Thread-safe. java.util.Collections class contains a synchronizedCollection method that creates thread-safe instances based on collections which are not. For example: `Set s = Collections.synchronizedSet(new HashSet());`
- From Java 1.5 onwards, you can also use a new Concurrent library available in java.util.concurrent package, which contains interfaces/classes such as ConcurrentMap/ConcurrentHashMap. Classes in this package offer better performance than objects returned from Collections.synchronizedXXX methods.
- Arrays.binarySearch() method returns the index of the search key, if it is contained in the list; otherwise, (-(insertion point) - 1).

- `Map.Entry<K,V> pollFirstEntry()` - Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
- `Map.Entry<K,V> pollLastEntry()` - Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
- `NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)` - Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey. Note that the tailmap is backed by the original map so change to the tailmap affects the original map as well.
- For Collections.sort(List) method to work, the elements of the passed List must implement Comparable interface. Here, Book does not implement Comparable and therefore this line will fail to compile. It is important to understand that compilation failure occurs because books is declared as `List<Book>`. If it were declared as just `List`, compilation would succeed and it would fail at run time with a ClassCastException.
- It is not permissible for a map to contain itself as a key. But it can contain itself as a value.
- All keys in a map are unique.
- Not all Map implementations keep the keys sorted. Only TreeMap does. Implementations of SortedMap keep the entries sorted.
- While all the keys in a map must be unique, multiple identical values may exist. Since values are not unique, the values() method returns a Collection instance and not a Set instance.
- A LinkedHashMap is a linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map.
- In the exam, you will be asked to order words that are different only in case. You need to remember a simple rule to answer such questions: `spaces < numbers < uppercase < lowercase`.

 # TreeSet:

- Note that TreeSet is an ordered set that keeps its elements in a sorted fashion. When you call the add() method, it immediately compares the element to be added to the existing elements and puts the new element in its appropriate place. Thus, the foremost requirement of a TreeSet is that the elements must either implement Comparable interface (which has the compareTo(Object) method) and they must also be **mutually comparable** or the TreeSet must be created with by passing a Comparator (which has a compare(Object, Object)) method).
- If you add objects to a TreeSet that do not implement Comparable interface, the program will compile fine without any warning. The compiler knows nothing about this requirement of TreeSet since it is an application level requirement and not a language level requirement. However, it throws a ClassCastException at runtime when you add the first element itself.

 # NavigableSet:

- A NavigableSet is a SortedSet extended with navigation methods reporting closest matches for given search targets. Methods lower, floor, ceiling, and higher return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning null if there is no such element. Since NavigableSet is a SortedSet, it keeps the elements sorted.
  - `E ceiling(E e)` Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
  - `E floor(E e)` Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
  - `E higher(E e)` Returns the least element in this set strictly greater than the given element, or null if there is no such element.
  - `E lower(E e)` Returns the greatest element in this set strictly less than the given element, or null if there is no such element.

- TreeSet is a NavigableSet and so it supports subSet() method :

`NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)` - Returns a view of the portion of this set whose elements range from fromElement to toElement. The returned subset is backed by the original set. So if you insert or remove an element from the subset, the same will be reflected on the original set.

- Since the subset is created using a range (fromElement to toElement), if you insert an element, the element that you are inserting must fall within that range. Otherwise an IllegalArgumentException is thrown with a message "key out of range.".

 # NavigableMap:

- A NavigableMap is a SortedMap (which in turn extends Map) extended with navigation methods returning the closest matches for given search targets. Methods lowerEntry, floorEntry, ceilingEntry, and higherEntry return Map. Entry objects associated with keys respectively less than, less than or equal, greater than or equal, and greater than a given key, returning null if there is no such key. Similarly, methods lowerKey, floorKey, ceilingKey, and higherKey return only the associated keys. All of these methods are designed for locating, not traversing entries.
- TreeMap is a class that implements NavigableMap interface. ConcurrentSkipListMap is the other such class.

 # Deque:

A Deque can act as a Queue as well as a Stack:

1. Since Queue is a FIFO structure (First In First Out i.e. add to the end and remove from the front), it has methods `offer(e)/add(e)` (for adding an element to the end or tail) and `poll()/remove()` (for removing an element from the front or head) for this purpose. Note that offer and add are similar while poll and remove are similar. **`remove()` is a queue method that removes the element from the front.**
2. Since Stack is a LIFO structure (Last In First Out i.e. add to the front and remove from the front), it provides methods `push(e)` and `pop()` for this purpose, where push adds to the front and pop removes from the front.

Besides the above methods, Deque also has variations of the above methods. But it is easy to figure out what they do:

- `pollFirst()/pollLast()` - poll is a Queue method. Therefore pollFirst and pollLast will remove elements from the front and from the end respectively.
- `removeFirst()/removeLast()` - These are Deque specific methods. They will remove elements from the front and from the end respectively. These methods differ from pollFirst/pollLast only in that they throw an exception if this deque is empty.
- `offerFirst(e)/offerLast(e)` - offer is a Queue method. Therefore offerFirst and offerLast will add elements to the front and to the end respectively.
- `addFirst(e)/addLast(e)` - add is a Queue method. Therefore addFirst and addLast will add elements to the front and to the end respectively.
- `peek(), peekFirst()` : return the first element from the front of the queue but does not remove it from the queue.
- `peekLast()` : returns the last element from the end of the queue but does not remove it from the queue.
- `element()` : retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque). This method differs from peek only in that it throws an exception if this deque is empty.
- Notice that there is no `pushFirst(e)` and `pushLast(e)`
- You may wonder why there are multiple methods for the same thing such as offer(e) and add(e). Well, they are not exactly same. add(e) throws an exception if the element cannot be added to the queue because of lack of capacity, while offer(e) does not. There are similar differences in other methods but they are not too important for the exam.

§3.2 Generic class creation and usage

- You cannot embed a diamond operator within another generic class instantiation. Thus, `new HashMap<String, List<>>` is invalid because of `List<>`.