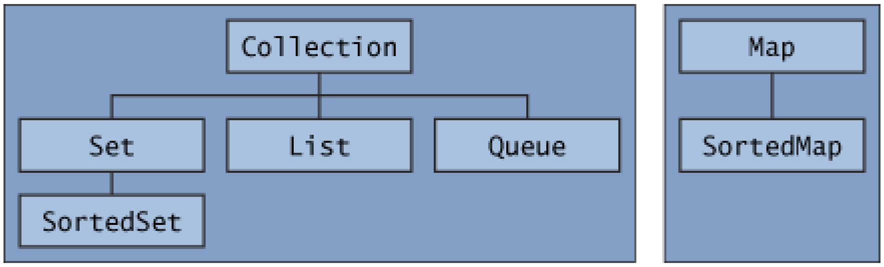
Software Development

« SCJP – Study Resources
SCJP – Generics »

SCJP – Java Collections Cheat Sheet

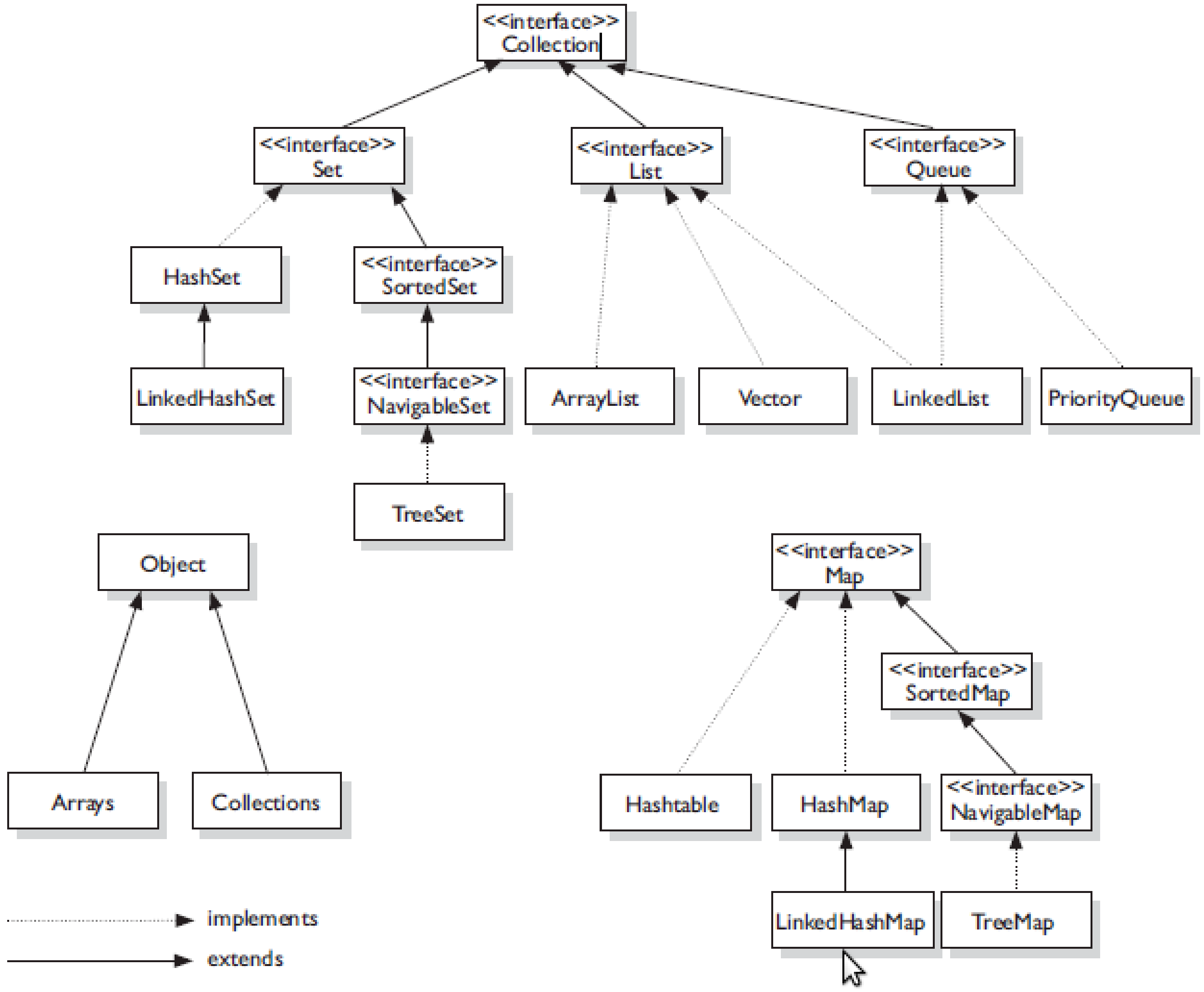
This is just a compilations of some notes together with my own notes during the preparation for SCJP exam. I expect to be the first post on the same subject, one for each chapter or topic.
[[pageindex]]

Interfaces



Implementations

		Implementations				
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
Interfaces	**Set**	HashSet		TreeSet		LinkedHashSet
	List		ArrayListVector		LinkedList	
	Map	HashMapHashtable		TreeMap		LinkedHashMap



Class	Map	Set	List	Ordered	Sorted
`HashMap`	x			No	No
`Hashtable`	x			No	No
`TreeMap`	x			Sorted	By *natural order* or custom comparison rules
`LinkedHashMap`	x			By insertion order or last access order	No
`HashSet`		x		No	No
`TreeSet`		x		Sorted	By *natural order* or custom comparison rules
`LinkedHashSet`		x		By insertion order	No
`ArrayList`			x	By index	No
`Vector`			x	By index	No
`LinkedList`			x	By index	No
`PriorityQueue`				Sorted	By to-do order

Collection Interfaces

- **Collection** - A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.

Interfaces	Main methods>
• Set - Extends the `Collection` ○ The familiar set abstraction. ○ No duplicate elements permitted. ○ At most one null element. ○ May or may not be ordered.interface.	• `boolean add(E e)` • `boolean remove(Object o)` • `boolean contains(Object o)` • `isEmpty()` • `size()`
• SortedSet - Extends Set ○ A set whose elements are automatically sorted, either in their natural ordering (see the `Comparable` interface), or by a `Comparator`object provided when a `SortedSet` instance is created. ○ first() and last() methods throws NoSuchElementExecption when set is empty	• `Comparator<? super E> comparator()` • `E first()` • `E last()` • `SortedSet<E> headSet(E toElement)` • `SortedSet<E> subSet(E fromElement, E toElement)` • `SortedSet<E> tailSet(E fromElement)`
• NavigableSet - Extends SortedSet ○ Navigation methods reporting closest matches for given search targets. ○ A `NavigableSet` may be accessed and traversed in either ascending or descending order. ○ All methods that receives a parameter can throw exceptions: ▪ ClassCast if arg type are different ▪ NPE if arg is null	• `E lower(E e)` • `E floor(E e)` • `E higher(E e)` • `E ceiling(E e)` • `E pollFirst()` • `E pollLast()` • `SortedSet<E> subSet(E fromElement, E toElement)` • `SortedSet<E> headSet(E toElement, boolean inclusive)` - Method withoud inclusive flag available from SortedSet interface • `SortedSet<E> tailSet(E fromElement, boolean inclusive)` - Method withoud inclusive flag available from SortedSet interface
-	-
• Map- ○ A mapping from keys to values. ○ Each key can map to at most one value.	• `V put(K key, V value) -` ○ replaces the old value is replaced by the specified value ○ returns the previous value associated with key • `V get(Object key)` • `V remove(Object key)`

	◦ returns the previous value associated with key • boolean containsKey(Object key) • Set<Map.Entry<K,V>> entrySet() • Set<K> keySet() • Collection<V> values()
• SortedMap- Extends Map ◦ A map whose mappings are automatically sorted by key, either in the keys' natural ordering or by a comparator provided when a SortedMap instance is created. Extends the Map interface.	• K firstKey() - firstEntry only in NavigableMap • K lastKey() - lastEntry only in NavigableMap • SortedMap<K,V> headMap(K toKey) • SortedMap<K,V> tailMap(K fromKey) • SortedMap<K,V> subMap(K fromKey, K toKey)
• NavigableMap - Extends SortedMap ◦ navigation methods returning the closest matches for given search targets. ◦ May be accessed and traversed in either ascending or descending key order. ◦ All methods that receives a parameter can throw exceptions: ▪ ClassCast if arg type are different ▪ NPE if arg is null	• Map.Entry<K,V> lowerEntry(K key) • Map.Entry<K,V> floorEntry(K key) • Map.Entry<K,V> higherEntry(K key) • Map.Entry<K,V> ceilingEntry(K key) • K lowerKey(K key) • K floorKey(K key) • K higherKey(K key) • K ceilingKey(K key) • Map.Entry<K,V> firstEntry() • Map.Entry<K,V> lastEntry() • NavigableSet<K> descendingKeySet() • NavigableMap<K,V> descendingMap() • Map.Entry<K,V> pollFirstEntry() • Map.Entry<K,V> pollLastEntry()
-	-
• List - Extends the Collection. ◦ Ordered collection, also known as a sequence. ◦ Duplicates are generally permitted. ◦ Allows positional access.	• boolean add(E e) • boolean remove(Object o) • E get(int index) • boolean contains(Object o) • int indexOf(Object o) • int size() • List<E> subList(int fromIndex, int toIndex)
• Queue - Extends Collection ◦ A collection designed for holding elements prior to processing. ◦ Provide additional insertion, extraction, and inspection operations.	• Throws exception: ◦ boolean add(E e) ◦ E remove() ◦ E element()

  - Each of these methods exists in two forms:
    - one throws an exception if the operation fails,
    - the other returns a special value (either `null` or `false`, depending on the operation).

- Returns null or false:
  - `boolean offer(E e)`
  - `E poll()`
  - `E peek()`

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- Deque - Extends Queue
  - A linear collection that supports element insertion and removal at both ends.
  - The name *deque* is short for "double ended queue" and is usually pronounced "deck".
  - When a deque is used as a queue, FIFO (First-In-First-Out) behavior results.
  - Deques can also be used as LIFO (Last-In-First-Out) stacks.

	First Element (Head)		**Last Element (Tail)**	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

- Both in a queue or stack view, the elements are always read/removed from the head of the list: element/peek/poll for queue, pop/peek for the stack.
  - In the queue elements are added to the end of the list: add/offer
  - In the stack elements are added to the head of the list: push

The methods inherited from the `Queue` interface are precisely equivalent to `Deque`

`Queue` Method	**Equivalent `Deque` Method**
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

Stack methods are precisely equivalent to `Deque` methods as indicated in the table below:

Stack Method	**Equivalent `Deque` Method**
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`

`peek()`	`peekFirst()`

General-Purpose Implementations

• **HashSet** ◦ Implements Collection<E>, Set<E>	• This class permits the `null` element. • Fast access, assures no duplicates, provides no ordering.
• **LinkedHashSet** ◦ Implements Collection<E>, Set<E>	• Runs nearly as fast as `HashSet`. • No duplicates; iterates by insertion order.
• **TreeSet** - Implements Collection<E>, NavigableSet<E>, Set<E>, SortedSet<E>	• not syncronized • No duplicates; iterates in sorted order.
-	
• **ArrayList** ◦ Implements Collection<E>, List<E>	• Fast iteration and fast random access
• **Vector** ◦ Implements Collection<E>, List<E>	• synchronised • It's like a slower ArrayList, but it has synchronized methods.
• **LinkedList** ◦ Implements Collection<E>, Deque<E>, List<E>, Queue<E>	• May provide better performance than the `ArrayList` implementation if elements are frequently inserted or deleted within the list. • Good for adding elements to the ends, i.e., stacks and queues.
• **PriorityQueue** ◦ Implements Collection<E>, Queue<E>	• The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time • unbounded. its capacity grows automatically. • does not permit null elements • The *head* of this queue is the *least* element • The Iterator provided in method iterator() is not guaranteed to traverse the elements of the priority queue in any particular order.

• **Stack** - Extends Collection<E>, List<E> ◦ Deque interface should be used in preference to the legacy **Stack** class	• `E pop()` • `E push(E item)` • `E peek()`
-	
• **HashMap** ◦ Implements Map<K,V>	• Essentially an unsynchronized `Hashtable` that supports `null` keys and values. • Fastest updates (key/values); allows one null key, many null values.
• **Hastable** ◦ Implements Map<K,V>	• Like a slower HashMap (as with Vector, due to its synchronized methods). • No null values or null keys allowed
• **LinkedHashMap** ◦ Implements Map<K,V>	• Predictable iteration order. • Runs nearly as fast as `HashMap`. • Allows one null key, many null values.
• **TreeMap** ◦ Implements Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>	• A sorted map • Ascending element order

Notes

- As of Java 6 TreeSets and TreeMaps have new navigation methods like floor() and higher().

- Sorting can be in natural order, or via a Comparable or many Comparators.
- Implement Comparable using compareTo(); provides only one sort order.
- To be sorted and searched, a List's elements must be comparable.
- To be searched, an array or List must first be sorted.
- Every method that invokes a comparator with different type will throw a ClassCastException. For instance, a subset invoke on a sortedMap with different types as arguments will throw a ClassCastExecption

- A TreeSet sorts its elements.
  - By default, it will try to sort the elements in their natural order. For this to happen, it is necessary that they implements Comparable.
  - Therefore, all types in the set must either be comparable with each other, or the comparator given to the TreeSet be able to compare the elements.
  - If a TreeSet<Number> already has one int inside it, all other numbers added must be of type int (or castable to int, e.g. short).This means long, float, etc cannot be added.
- When adding to a hashmap, if an items hashcode and equals are equal to another item already in the map, the new item will replace the old one.

- Prior to using Collections.binarySearch, it is necessary to first ensure the collection is sorted, which can be achieved by calling Collections.sort(). If the collection isn’t sorted, the results are undefined.
- If a collection is sorted using a comparator, it is critical that the binarySearch method also be called with the same comparator.

- Polling’ is the term used to mean retrieve and remove from the collection. The TreeSet interface has pollFirst() and pollLast() methods. Similarly, TreeMap has pollFirstEntry() and pollLastEntry().
- ‘Peeking’ is the term used to mean retrieve an object from a collection, without removing it.

- HashMap is unsynchronized and can have null keys and values. Conversely, HashTable is synchronized and cannot have null keys and values.
- map.keySet().add("A"); //will throw an UnsupportedOperationException, it's not implemented
    - map.keySet().remove(1);   // this works fine and remove the entry from the map.

- Generic API have the read only methods receiving Objects as arg, instead of E type. Example on the List<E> class: int lastIndexOf(Object)
- if MyClass extends HashSet<Person> then the overrride of the add must be add(Person) and not add(Object)

Utility methods

Method	Class	Explanation	static?	return
`reverse(List<?> list)`	Collections	reverses the order of elements in a List.	static	void
`reverseOrder()`	Collections	Returns a comparator that imposes the reverse of the natural ordering	static	`Comparator<T>`
`asList(T... a)`	Arrays	Returns a fixed-size list backed by the specified array.	static	`List<T>`
`toArray()`	Collection interface	Returns an array containing all of the elements in this collection.	no	`Object[]`
`toArray(T[] a)`	Collection interface	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.	no	`T[]`

LinkedList

- add,remove and get throws an exception (for instance if element not found)
    - offer, poll and peek returns null (for instance if element not found)
- stack interface is only: push, pop and peek
- queue interface is: offer, poll and peek
- deque interface is: offer First and Last, poll First and Last, peek First and Last. It's a double linked list !!
- sublist does not affect the list given as parameter. It returns the modified list.

TreeSet / TreeMap

- from SortedSet interface: first(), last(), tailSet(), headSet() and subSet(). From a natural order sequence!!
- headSet() work exclusive until the given e. tailSet() work inclusive.

- lower() returns the element less than the given element, and floor() returns the element less than or equal to the given element.
    - Similarly, higher() returns the element greater than the given element, and ceiling() returns the element greater than or equal to the given element.
- subSet work inclusive at begin at exclusive and the end.
    - subMap() from NavigableMap returns an instance of SortedMap, not NavigableMap as expected.
- subset(a, z) throws a IllegalArgumentException if z is greater than a
- backed collections: adding elements to the subset will add the element also to the main data set
    - adding a element out of range will throw a java.lang.IllegalArgumentException
- `lower`,`floor`,`ceiling`, and`higher` are methods from navigableSet and from NavigableMap
- TreeMap doesn't implement Iterable, so cannot it's not possible to iterate over the entries, only from the keySet or valuesSet
- add() throws NPE with a null arg if set is working with natural order, or if the comparator doesn't allows null objects.
    - **This only happens at second insert, because only that time the comparator is invoked.**

LinkedHashMap

- As in the TreeMap and other Map implementation, it's not possible to iterate over the entries. First we need to get the values and then we already can iterate since this is a Collection. Of course with the LinkedHashMap (that is order, and not sorted), the iteration over the values gives elements ordered by the insertion order.

PriorityQueue

- PriorityQueue it's iterable, it's a queue and it's a collection
- The iteration order given by the iterator is undefined, depends of the implmentation.
    - poll and peek should be used
- It have the main method from a queue: offer(), poll(), peek(). Also the one from collection: add, remove, get...
- Can be sorted by natural order or by any other custom order: The constructor could receive a Comparator.

Arrays, Sort and Search

- don't forget to check if the compareTo() method is public.

Natural Order

- Implement Comparable using compareTo(); provides only one sort order.
- compareTo must implement the ascending order:
    - return this.value - arg.value;
- a comparator in a natural order is:
    - compare(a, b) { return a.compareTo(b))
- reverse order:
    - *public int compareTo(Human h) {*
      return h.age.compareTo(this.age);
      }

Sort

- void sort(Object[] a)   - **sorts by natural order**

  - Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
  - All elements in the array must implement the `Comparable`interface.
    - Comparable interface: *int compareTo(T o)*
  - Can throw a `ClassCastException` if there are different element types in the array
- void sort(Object[] a, Comparator c) - **sorts by the given comprator**
  - Comparator interface: *int compare(T o1, T o2)*

Search

- binarySearch(Object[] a, Object key) // Searches using the binary search algorithm. l.
  - If it is not sorted, the results are undefined.
  - returns the object index

- binarySearch(Object[] a, Object key, Comparator c)
  - Comparator interface: *int compare(T o1, T o2)*
  - Attempting to search an array or collection, which is not sorted will cause an unpredictable search result.
  - The binarySearch() method gives meaningful results only if it uses the same Comparator as the one used to sort the array. Other way the result is: -1

-

Equals and Hashcode

- default (Object) implementation for equals and hashcode methods supports their contracts .
- public void equals(**Object** o)

Collections

- Collections methods sort(), reverse(), and binarySearch() do not work on Sets.
  - all of these have List as args, and so gives compilation error for Sets

Important notes to remimder

- **Four basic flavors of collections include**
  - **Lists,**
  - **Sets,**
  - **Maps,**
  - **Queues.**
  - Map IS NOT a Collection, all others are

- Don't forget to verify that
  - for Collection - **add**()
  - for Map - **put**()
- LinkedList is a (implements) List and a Queue
  - Queue is a interface

My java collections source code examples

Here are the eclipse project with some code that I did to practice. For now it isn't real exercises, is only some code to understand APIs usage.

scjp_CodeExamplesAndExercises.zip

Sources: *- SCJP Study Guide, McGrawHill*
- oracle.com - Collections Framework
- JonathanGiles.net

Tags: cheat card, cheat sheet, collections, java, ocjp, scjp

This entry was posted on Monday, December 27th, 2010 at 4:59 pm and is filed under Uncategorized. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

5 Responses to "SCJP – Java Collections Cheat Sheet"

1. *SCJP resources – How To ...* Says:
   May 27th, 2011 at 8:53 am

   [...] collection of resources, information, examples, compilation of topics and some cheat sheets for preparation for java certification [...]

2. *PROG Java vraagje - Pagina 2 - 9lives - Games Forum* Says:
   January 18th, 2012 at 6:43 pm

   [...] SCJP – Java Collections Cheat Sheet | Software Development [...]

3. *Lavinia* Says:
   January 23rd, 2012 at 4:33 pm

   Great job! Thanks!

4. *vinodh* Says:
   September 2nd, 2012 at 5:33 pm

   Hi,
   thanks a lot for this. cos i am preparing for interview.
   no other cheat sheet is as elaborate as this.
   regards
   vinodh

5. *N* Says:
   March 24th, 2016 at 4:21 pm

   Thanks for this! Very comprehensive - using it for interview prep!

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

-
- Search for:
  Search
- G+

- ## Recent Posts
  - Parallel Streams
  - A Note On Java EE Testing
  - Java 8 resources
  - IIOP standalone EJB3 client lookup to Glassfish 3.1.2
  - Want Better Estimates? Stop Estimating!

Software Development is proudly powered by WordPress
Entries (RSS) and Comments (RSS).