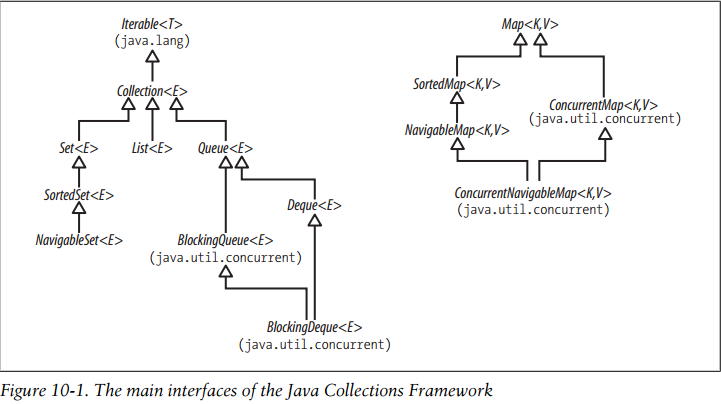
**Book : Java Generics and Collection**

**Chapter 11**

****

**Iterable interface:**

**public interface** Iterable<T> {  
 */\*\*  
 \* Returns an iterator over elements of type {****@code*** *T}.  
 \*  
 \** ***@return*** *an Iterator.  
 \*/* Iterator<T> iterator();  
  
 */\*\*  
 \* Performs the given action for each element of the {****@code*** *Iterable}  
 \* until all elements have been processed or the action throws an  
 \* exception. Unless otherwise specified by the implementing class,  
 \* actions are performed in the order of iteration (if an iteration order  
 \* is specified). Exceptions thrown by the action are relayed to the  
 \* caller.  
 \*  
 \** ***@implSpec*** *\* <p>The default implementation behaves as if:  
 \* <pre>{****@code*** *\* for (T t : this)  
 \* action.accept(t);  
 \* }</pre>  
 \*  
 \** ***@param action*** *The action to be performed for each element  
 \** ***@throws*** *NullPointerException if the specified action is null  
 \** ***@since*** *1.8  
 \*/* **default void** forEach(Consumer<? **super** T> action) {  
 Objects.*requireNonNull*(action);  
 **for** (T t : **this**) {  
 action.accept(t);  
 }  
 }  
  
 */\*\*  
 \* Creates a {****@link*** *Spliterator} over the elements described by this  
 \* {****@code*** *Iterable}.  
 \*  
 \** ***@implSpec*** *\* The default implementation creates an  
 \* <em><a href="Spliterator.html#binding">early-binding</a></em>  
 \* spliterator from the iterable's {****@code*** *Iterator}. The spliterator  
 \* inherits the <em>fail-fast</em> properties of the iterable's iterator.  
 \*  
 \** ***@implNote*** *\* The default implementation should usually be overridden. The  
 \* spliterator returned by the default implementation has poor splitting  
 \* capabilities, is unsized, and does not report any spliterator  
 \* characteristics. Implementing classes can nearly always provide a  
 \* better implementation.  
 \*  
 \** ***@return*** *a {****@code*** *Spliterator} over the elements described by this  
 \* {****@code*** *Iterable}.  
 \** ***@since*** *1.8  
 \*/* **default** Spliterator<T> spliterator() {  
 **return** Spliterators.*spliteratorUnknownSize*(iterator(), 0);  
 }  
}

forEach() is added in java 8 , so we can use list.forEach(cosnumer). Consumer takes an argument and don’t return any value. So we can use forEach on collection and process it.

splitIterator() is also added in java 1.8, it’s trySplit() method split the whole data into sutiable chunks.

If we want to optimize the performance, we should override this method as per our need.

Parallel stream internally uses splitIterator for parallel processing and breaks input into suitable parallel chunks.

**In java1.5 Collection interface was made to extend Iterable interface, so for each loop can be used with any list, set and queue.**

class Counter implements Iterable<Integer> {

private int count;

public Counter(int count) { this.count = count; }

public Iterator<Integer> iterator() {

return new Iterator<Integer>() {

private int i = 0;

public boolean hasNext() { return i < count; }

public Integer next() { i++; return i; }

public void remove(){ throw new UnsupportedOperationException(); }

Since Counter class is implementing Iterable interface, for each loop can be used on counter class.

int total = 0;

for (int i : new Counter(3)) {

total += i;

}

assert total == 6;

while doing for each or iterating over a collection, if any changes are made to source collection, Concurrent modification exception will come.

Its just that if u want to remove an element from collection, use iterator to do it else it throws same concurrent modification exception.

**Example 1**

List<Integer> list = **new** ArrayList();  
list.add(1);  
list.add(2);  
list.add(3);  
Iterator<Integer> iterator = list.iterator();  
**while**(iterator.hasNext()){  
 System.***out***.println(iterator.next());  
 list.add(6); //concurrent modification exception  
}

While iterating list is changed

**Example 2**

List<Integer> list = **new** ArrayList();  
list.add(1);  
list.add(2);  
list.add(3);  
Iterator<Integer> iterator = list.iterator();  
**while**(iterator.hasNext()){  
 System.***out***.println(iterator.next());  
 list.remove(1); //concurrent modification exception  
  
}

While iterating list is changed

**Example 3**

List<Integer> list = **new** ArrayList();  
list.add(1);  
list.add(2);  
list.add(3);  
Iterator<Integer> iterator = list.iterator();  
**while**(iterator.hasNext()){  
 System.***out***.println(iterator.next());  
 iterator.remove();  
}

Works fine because removal is through iterator

**The four basic structures**

**Arrays:** Continuous memory allocation of same data type. Element retrieval is fast by index but insertion and removal is slow as they may require adjusting of elements.

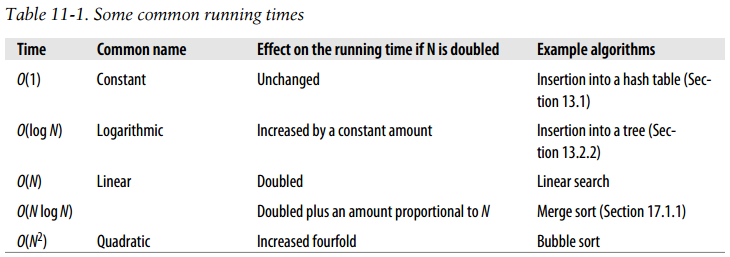
Insertion(if array size needs to be increased)

**LinkedList: Non continuous memory allocation. Retrieval is slow as always need to go by head but insertion/deletion is fast as only link reference needs to be updated**

**HashTables:** it stores elements based on hash value not index. Insertion/deletion is fast, accessing by hashvalue is fast relatively fast but there is no support for index based access.

**Tree:** it also stores element by content and in sorted order. Insertion/deletion, access is relatively fast.

**C**

****

**Synchronization and Legacy Collection:**

In java 1.0, there was fully synchronized colletions : Vector, HashTable.

But in JDK 1.2 changes were made due to performance cost and their non synchronized methods were added: ArrayList, HashMap etc

In java 1.5 , same policy was implemented for StringBuffer(thread safe and hence fast) and StringBuilder(thread safe) was introduced.

Collections.synchronized() wraps the collection to synchronized collection.

**Using Synchronized Collections Safely:**

Stack stack = new SynchronizedArrayStack(new ArrayStack());

...

// don't do this in a multi-threaded environment

if (!stack.isEmpty()) {

stack.pop(); // can throw IllegalStateException

}

Because client code is not synchronized, so we have to ensure that client code is synchronized and on the same lock used by synchronized wrapper.

So

synchronized(stack){

if (!stack.isEmpty()) {

stack.pop();

}

}

**Concurrent Collections Mechanisms:**

1. **CopyOnWriteArrayList or CopyOnWriteArraySet:** There is no lock for reading data. But whenever any changes are made to list, its copy is created and new element is added to that new copy. Its iterator is fail safe as since it is iterating over an old copy, in case new element is added simultaneously, so newly added element will not be in the iterator and hence it don’t throws concurrent modification exception. Write method has lock , so that no two threads can write simultaneously.After write array refrence is changed to point to new elementscopy and since it volatile, reading thread always follows happens before relationship.So if at same time

One thread is reading and other writing, reading thread will get array refrence from main memory as it is using volatile .(Happens before)

1. **ConcurrentLinkedQueue and ConcurrentSkipListMap:** It implements Compare and Swap(CAS) algorithm.These instructions are issued to processor.

Since while iterating, if any thread writes, so new value will not be there in iterator as iterator is working on a copy it fetches from memory.

1. **ConcurrentHashMap,Blocking Queue:** They implements Renetrant locks.

Before Java 1.6 Blocking Queue’s iterator was not thread safe but in 1.6 it is made fail safe as it also works on a copy.so it is weekly inconsistent.

*/\*\*  
 \* Returns an iterator over the elements in this queue. The  
 \* iterator does not return the elements in any particular order.  
 \*  
 \* <p>The returned iterator is  
 \* <a href="package-summary.html#Weakly"><i>weakly consistent</i></a>.  
 \*  
 \** ***@return*** *an iterator over the elements in this queue  
 \*/***public** Iterator<E> iterator() {  
 **return new** Itr(toArray());  
}

*/\*\*  
 \* Returns an array containing all of the elements in this queue.  
 \* The returned array elements are in no particular order.  
 \*  
 \* <p>The returned array will be "safe" in that no references to it are  
 \* maintained by this queue. (In other words, this method must allocate  
 \* a new array). The caller is thus free to modify the returned array.  
 \*  
 \* <p>This method acts as bridge between array-based and collection-based  
 \* APIs.  
 \*  
 \** ***@return*** *an array containing all of the elements in this queue  
 \*/***public** Object[] toArray() {  
 **final** ReentrantLock lock = **this**.**lock**;  
 lock.lock();  
 **try** {  
 **return** Arrays.*copyOf*(**queue**, **size**);  
 } **finally** {  
 lock.unlock();  
 }  
}

**Chapter 12 : The Collection Interface**

Collection interface has some core methods declaration.

1. **Adding Elements:**

boolean add(E e) // add the element e

boolean addAll(Collection<? extends E> c) // add the contents of c

1. **Removing Elements:**

boolean remove(Object o) // remove the element o

void clear() // remove all elements

boolean removeAll(Collection<?> c) // remove the elements in c

boolean retainAll(Collection<?> c) // remove the elements \*not\* in c

1. **Querying the contents of collection:**

boolean contains(Object o) // true if o is present

boolean containsAll(Collection<?> c) // true if all elements of c

// are present in the collection

boolean isEmpty() // true if no elements are present

int size() // return the element count (or

// Integer.MAX\_VALUE if that is less)

Because Integer has a fixed range, so when elements are more than 2 millions, it’s better to return a fixed value rather than random values.

1. **Making Collection’s content available for Further Processing:**

Iterator<E> iterator() // return an Iterator over the elements

Object[] toArray() // copy contents to an Object[]

<T> T[] toArray(T[] t) // copy contents to a T[] (for any T)

Collection<String> cs = ...

String[] sa = cs.toArray(new String[0]);

This version of toArray is used to get array of a specific data type rather than Object type.

Equivalent

private static final String[] EMPTY\_STRING\_ARRAY = new String[0];

Collection<String> cs = ...

String[] sa = cs.toArray(EMPTY\_STRING\_ARRAY);

-----

List<Object> l = Array.asList("zero","one",2);

String[] a = l.toArray(new String[0]); // run-time error Array Stored Exception as array 2 is not String

This can be used with reference type only.

One drawback of this design is that it does not work with arrays of primitive type:

List<Integer> l = Array.asList(0,1,2);

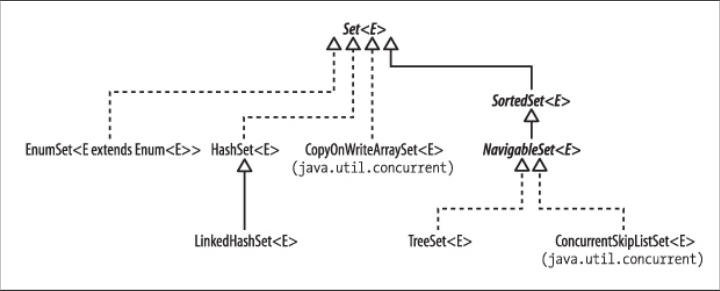
int[] a = l.toArray(new int[0]); // compile-time error

**Chapter 13 Sets**

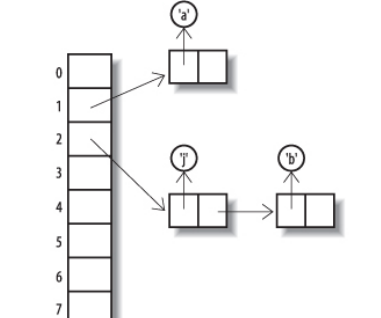
1. Set is a collection of unique items i.e. it doesn’t contain duplicates.
2. Set has the same methods as Collection , it just override them.

**Implementing Set:**

There are six concrete implementation of set:



**HashSet :**



1. HashSet internally implements HashTable. It saves elelement as key and a dummy object(new Object()) as value
2. HashTable is an array in which elements are stored on basis of hash value not index.
3. When there are more elements having same hash code, a link list is created at that index called bucket and elements are strored in liked list.
4. The default initial capacity of hashmap is 16 ie array of 16 size.
5. Its load factor value is .75 i.e. when 16\*.75 = 12 indexes are used, it will be resized.
6. **static final int *TREEIFY\_THRESHOLD*** = 8; if bins count on an index is greater than 8 , link list may be converted to binary tree.
7. **static final int *UNTREEIFY\_THRESHOLD*** = 6; if bin count on an index is less than 6, tree will be converted back to link list.
8. **static final int *MIN\_TREEIFY\_CAPACITY*** = 64; if the length of table is less than 64, hashmap will be resized rather than treefy.
9. This is added in java 8 to speedup performance as traversal of link list in worst case is o(n) while for binary tree , it is o(log n)
10. The chief attraction for implementging hashset using hashmap is that its constant time for basic operations like adding,remove.

To add()/remove(), hashcode of element is calculated and using that hash, array index can be calculated , so insertion/deletion is in a constant time.

1. Disadvantage is that to iterate, need to iterate each bucket, so cost = array size + bucket length.
2. There is no way to maintain insertion order.

**Hash Value:**

1. **public** V put(K key, V value) {  
    **return** putVal(*hash*(key), key, value, **false**, **true**);  
   }

Hashmap recalculates hash of hash to ensure that elements are spread across the buckets.

**static final int** hash(Object key) {  
 **int** h;  
 **return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

This is helpful in case of floats as floats are represented using special binary representation where some bits represent value after decimal and before.

Binary value of 2, 18 and 215 floats all have the same lower bits.

1000000000000000000000000000000

1000001100100000000000000000000

1000011010101110000000000000000

So in this case all will go into the same bucket as when their hashcode is calculated so they will be divided by 16 i.e.(number % 16 == number & 15) and 15 = (01111), so all will have the same index.

To avoid this h >>> 16 will right shift the bits , so that lower bits will be ignored and higher bits can be used.

In case of Integers, String this rehashing does not have any effect.

1. Hash Code implementation on Object class

**public native int** hashCode();

It returns the memory address where object is stored. Usually Every class should overrides Hashcode.

**public class** Person {  
  
 **private** Integer **id**;  
 **private** String **name**;  
  
 **public** Person(Integer id, String name) {  
 **this**.**id** = id;  
 **this**.**name** = name;  
 }  
  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public void** setId(Integer id) {  
 **this**.**id** = id;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
}

@Test  
**public void** testAbc() {  
  
 Person p1 = **new** Person(1, **"vishal"**);  
 Person p2 = **new** Person(1, **"vishal"**);  
 System.***out***.println(p1.hashCode());//22158529  
 System.***out***.println(p2.hashCode());//22002071  
}

Though they have same values but still their hash value is different as they are returning memory addresses.

1. Hash Code of Integers

**public static int** hashCode(**int** value) {  
 **return** value;  
}

Integer value itself is returned i.e. 1 for 1, 2 for 2 etc.

1. Hashcode of String

**public int** hashCode() {  
 **int** var1 = **this**.hash;  
 **if**(var1 == 0 && **this**.value.length > 0) {  
 **char**[] var2 = **this**.value;  
  
 **for**(**int** var3 = 0; var3 < **this**.value.length; ++var3) {  
 var1 = 31 \* var1 + var2[var3];  
 }  
  
 **this**.hash = var1;  
 }  
  
 **return** var1;  
}

or

int hash = 0;

for (char ch : str.toCharArray()) {

hash = hash \* 31 + ch;// 31 is prime odd

}

Hash value of each character is its UNICODE value which is multiplied by 31 everytime. This is do so that even anagram gets different hahscode.

Multiplication is by 31 is random , an odd value is selected so that odd value is generated which will be divided by an even value(because size of hahsmap array increases in order of power of 2 i.e. 2,4,8,16,32 etc.) otherwise if too many even values hashcode is generated on dividing them by an even value , all will get 0 index.

**HashSet Summary:**

**1)**Ordering: no because elements are saved on basis of hashcodes

2) Random Access: no indexing

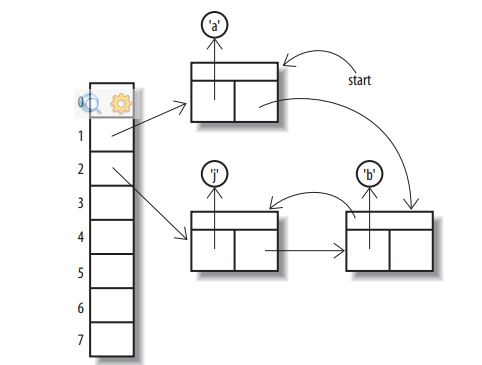
3)Key-Value: no

4)Duplicate: Sets are unique

5)Null Elements: Since it is unique, only one null value can exist and hence true. If it saves duplicate elements , then on saving two null, it wont be able to distinguish which null to retrun.

6)Thread Safe: No.Fail fast iterator

**LinkedHashSet:**

****

1. It maintains the insertion onrder.
2. It’s link list nodes have two pointers before, next.
3. Therefore link list can be traversed in insertion order.
4. Its iterator’s next() has constant time
5. But since maintaining two pointers requires more memory.

**Summary:**

**1)**Ordering: yes.It maintains insertion order because of two pointers

2) Random Access: no indexing

3)Key-Value: no

4)Duplicate: Sets are unique

5)Null Elements: Since it is unique, only one null value can exist and hence true. If it saves duplicate elements , then on saving two null, it wont be able to distinguish which null to retrun.

6)Thread Safe: No.Fail fast iterator

**CopyOnWriteArraySet:**

1. It is thread safe as whenever an element is inserted into set, it creates an copy of existing array +1 and then inserts new element into it.
2. So in that case if an thread is iterating at the same time, it will iterate over the copy at that time and my be the newly inserted element will not there. So it is weekly consistent.
3. It is backed by copyonWriteArrayList

**static final class** COWIterator<E> **implements** ListIterator<E> {  
 */\*\* Snapshot of the array \*/* **private final** Object[] **snapshot**;  
 */\*\* Index of element to be returned by subsequent call to next. \*/* **private int cursor**;  
  
 **private** COWIterator(Object[] elements, **int** initialCursor) {  
 **cursor** = initialCursor;  
 **snapshot** = elements;  
 }

Its iterator is working on a copy at that time. Since no instance of this array should be modified, remove method of iterator is not implemented. Remove() method is not present in iterator as iterator may itself working on an old copy.

**public void** add(**int** index, E element) {  
 **final** ReentrantLock lock = **this**.**lock**;  
 lock.lock();  
 **try** {  
 Object[] elements = getArray();  
 **int** len = elements.**length**;  
 **if** (index > len || index < 0)  
 **throw new** IndexOutOfBoundsException(**"Index: "**+index+  
 **", Size: "**+len);  
 Object[] newElements;  
 **int** numMoved = len - index;  
 **if** (numMoved == 0)  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 **else** {  
 newElements = **new** Object[len + 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }  
 newElements[index] = element;  
 setArray(newElements);  
 } **finally** {  
 lock.unlock();  
 }  
}

**Summary:**

1)Ordering: Yes, as they are backed by copyonwriteArrayList

2) Random Access: no indexing

3)Key-Value: no

4)Duplicate: Sets are unique

5)Null Elements: Since it is unique, only one null value can exist and hence true. If it saves duplicate elements , then on saving two null, it wont be able to distinguish which null to retrun.

6)Thread Safe: Yes

\*copy collections can accept null as it works on array.

**EnumSet :**

1. They contains enum of only one type else give compilation error.
2. It save enums as bits vector , so highly optimized. It could be only saved as bits because size and type of enum is fixed and cant be changed at runtime.
3. It has two implementations : RegularEnumSet and JumboEnumSet
4. Regular enum set stores **private long elements** = 0L;ie long is of 64 bits
5. Jumbo enum set stores **private long elements**[];
6. These classes are can be accessed within the package only, so there is no way to access them directly and can be accessed by EnumSet only.
7. If the elements are less than 64, enum set creates RegularEnum Set else Jumbo enum set
8. Its iterator is not thread safe but don’t throws the concurrent modification exception and will be weakly consistent.
9. EnumSet can be initialized through different factory methods not directly
10. EnumSet<Color> yellow = EnumSet.*of*(Color.***RED***,Color.***BLUE***);
11. Creates enum set of two enums

<E extends Enum<E>> EnumSet<E> of(E e)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)

1. EnumSet<Color> yellow = EnumSet.*allOf*(Color.**class**);//will create enum set with all the enums in enum class eg (Red,Green,Blue)
2. EnumSet. *noneOf*(Color.**class**); //empty enum set of type Color []
3. They maintain natural ordering

**Summary:**

1)Ordering: Yes, because of Vector

2) Random Access: no indexing

3)Key-Value: no

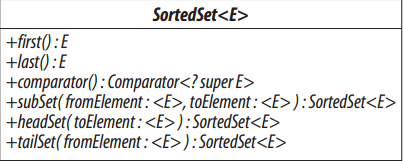
4)Duplicate: Sets are unique

5)Null Elements: Cant insert null as it only saves element of an enum type and null is not enum.

6)Thread Safe: No. To obtain a thread safe *Collections.synchronizedSet(EnumSet.noneOf(MyEnum.class)); can be used*

7)Iteator is weekly consistent and does not throw concurrent modification exception.

**SortedSet interface:**



First() : first element

Last() : last element

Comparator :

**Range View Methods:**

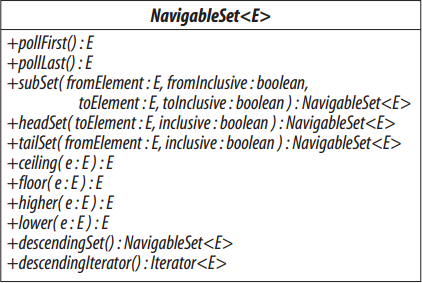
Subset : returns sub set inclusive left and exclusive right

Headset: returns element less than exclusive right element i.e. specified element

Tailset : returns element greater than specified element inclusive specified element

**Navigable Interface:** It was introduced in java 1.6 to supplement the deficiencies in sorted set.

It provides closest matching view of a given value.



Higher: strictly higher

Floor: equal or high

Lower : strictly lower

Ceiling : equal or high

Now it supplements the sorted interface

Subset(): u can specify inclsive from and inclusive to true/false

Headset() : u can specify inclusive Boolean value

Tailset(): u can specify inclusive booelan

We also have descendingIterator and descendingSet()

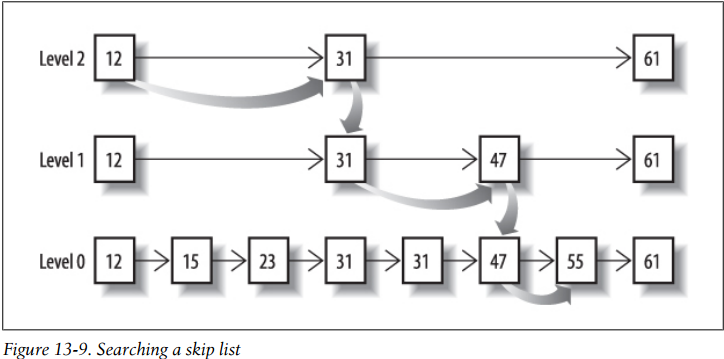
**TreeSet:**

1. It is sorted set
2. It maintains red black tree, so elements are sorted and complexity for insertion/removal is o(logn).Balanced Binary Tree
3. 1)Ordering: Yes as it is implementing ConcurrentNavigable, so sorted on natural ordering if no, comparator specified.
4. 2) Random Access: no indexing
5. 3)Key-Value: no
6. 4)Duplicate: Sets are unique
7. 5)Null Elements: No, because it is sorted and null cant be compared
8. 6)Thread Safe: No
9. 7)Iteator fail fast and does throw concurrent modification exception.

**ConcurrentSkipListSet:**

**public class** ConcurrentSkipListMap<K,V> **extends** AbstractMap<K,V>  
 **implements** ConcurrentNavigableMap<K,V>, Cloneable, Serializable

1. It was introduced in java 1.6.
2. Skip List is a set of linked lists.



3) Due to skip lists, insertion,deletion,traversal all operations occur in logn time.

4) Base level has all elements. When a new element is inserted in base level depending on factor.5 i.e. half of the elements should be in upper level , the same element may be inserted in upper level too.

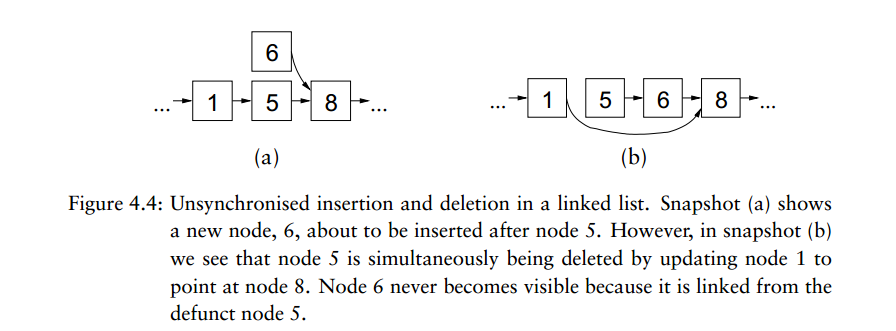
5) To remove an element, it is removed from each level where it is present.

6) To traverse, it starts from top level, then if next element is less than the searched number, it is moved to the next on same level else drop to the bottom level.

7) It provides concurrent operations as they implement lock free linked lists

**Lock Free List Algorithm:**

1. Problem: simultaneous insertion/deletion



**Cause:** Because insertion after a node updates node’s next pointer while deletion of a node updates nodes previous nodes next pointer

To fix it deleted node is marked as X, so if concurrent insertion/deletion, since the same node is having mark, no insertion operation will occur on it.

**Search:** if the left node is marked, search will restart.

If there is a sequence of marked nodes b/w left and right node, all will be ignored and left nodes next pointer will be updated to point to the right node.

**Deletion:** In first step, nodes next pointer will be updated with null value, so after that any subsequent operations on that node will ignored. We do so, that in case concurrent delete/insert , since now delete is in its first phase, node will be inserted .

In phase 2 , node will be marked.

In phase 3, node next pointer will be updated.

**Update:** Check for value, if found update it atomically, if not insert a new node at base level.

**Summary:**

1)Ordering: Yes as it is implementing ConcurrentNavigable, so sorted on natural ordering if no, comparator specified.

2) Random Access: no indexing

3)Key-Value: no

4)Duplicate: Sets are unique

5)Null Elements: No, almost all concurrent collection don’t insert null as they don’t able to identify null element on null retrun by contains in case element not present.

6)Thread Safe: yes

7)Iteator is weekly consistent and does not throw concurrent modification exception.

**Ques) Why Concurrent Maps or collection don’t accept null ?**

**Ans)** Map get(key) returns null if key is not in map. It also returns null , if value for key is null.

So to check whether key is mapped or not, we can use contains function in non concurrent collection.

But in case of concurrent collections, collection may changed in between these two calls. So suppose if key present contains returns true, now key is not present…..get(key) returns null, so we cant decide this null belongs to which case.

**Q) Predict the output :**

Ans)

NavigableSet<String> base = **new** TreeSet<String>(Collections.*reverseOrder*());  
Collections.*addAll*(base, **"b"**, **"a"**, **"c"**);  
  
NavigableSet<String> set1 = **new** TreeSet<String>(base);  
System.***out***.println(set1);//since base is of NavigabeSet type, its insertion order will be maintained and **public** TreeSet(SortedSet<E> s) {  
 **this**(s.comparator());  
 addAll(s);  
} will be called.

So output : c,b,a //descending

NavigableSet<String> base = **new** TreeSet<String>(Collections.*reverseOrder*());  
Collections.*addAll*(base, **"b"**, **"a"**, **"c"**);  
  
NavigableSet<String> set1 = **new** TreeSet<String>((Set)base);  
System.***out***.println(set1);

Because now it is of Set type, so Treeset without comparator will be called and output : natural ordering a,b,c

**Chapter 14 Queue**

Queue interface has following basic methods

1. To add element :

Boolean offer() : insert element at the end of queue and if full returns false

Boolean add() : insert element at the end of queue and if queue if full throws IllegalStateException

1. Remove element :

E Remove(): remove item from head and if queue is empty throws *NoSuchElementException*

E Poll() : remove from head and if empty returns null

1. Retrieve element : These does not remove element but returns element from top

Peek(): returns null if queue is empty

Element() : returns top element throws *NoSuchElementException*

PTR: Because these methods returns null if queue is empty, you should avoid using null them. Queue interface discouraged the use of null and that’s the reason only LinkedList(Legacy) allows null.

**Implementing Queue**

1. **Priority Queue :**
2. It is an unbounded queue and implemented using priority heap.

To determine the highest priority element, it uses natural ordering i.e. comparable or comparator if specified.

1. Since it requires comparison, it don’t accept null element.
2. Its head returns least element with respect to ordering.
3. If multiple elements have same priority, any arbitrary element can come.
4. Its iterator does not guarantee order.Though u can sort is using Arrays.sort(pq.toArray()) //Collections.sort() works on list only
5. It is not thread safe , if want use PriorityBlockingQueue.
6. Offer(),add(),poll(),remove() : Since on adding element, it requires rearranging heap, so complexity is o(log n)
7. Remove(Object), contains(Object): As the whole heap traversal is required , complexity is o(n) time.
8. Size : Since a counter is always updated, so constant time o(1)
9. Peek() and element() : complexity is o(1) as they simply return top element

**Implementation**

1. It is based on priority heap i.e. Min Heap.
2. Heap is binary search tree with two differences
3. Parent of all node must be greater than both of its child.
4. It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array

Summary :

Ordering : Elements are not stored in any order.It just that poll() or peek() element will return root element i.e. the smallest one

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: cant as null cant be compared

Thread Safe : No

Iterator : returns element in random order as heap does not stored them in sorted order

**Ques) Why Heap is preferred to implement Priority Queue over BST ?**

Ans)

A Self Balancing Binary Search Tree like [AVL Tree](http://www.geeksforgeeks.org/avl-tree-set-1-insertion/), [Red-Black Tree,](http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/) etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally O(1), but can be easily implemented in O(1) by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding inorder predecessor or successor.
2. Inserting an element is naturally O(Logn)
3. Removing maximum or minimum are also O(Logn)
4. Decrease key can be done in O(Logn) by doing a deletion followed by insertion. See [this](http://geeksquiz.com/how-to-implement-decrease-key-or-change-key-in-binary-search-tree/) for details.

**So why is Binary Heap Preferred for Priority Queue?**

* Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.
* Although operations are of same time complexity, constants in Binary Search Tree are higher.
* We can build a Binary Heap in O(n) time. Self Balancing BSTs require O(nLogn) time to construct.
* Binary Heap doesn’t require extra space for pointers.
* Binary Heap is easier to implement.
* There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in Θ(1) time

**ConcurrentLinkedQueue:**

1. It is non blocking thread safe as it uses CAS operations on link list nodes.
2. It does not accept null due to concurrent operations
3. It is unbounded and poll() returns in FIFO order.

Summary :

Ordering : As link list , so insertion order

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: no as concurrent operations

Thread Safe : Yes

Iterator : Weekly consistent, returns insertion order

**Blocking Queue interface**

It has four variants of methods:

*Throws exception*

*Special value*

*Blocks*

*Times out*

*<td><b>Insert</b></td>  
\* <td>{****@link*** *#add add(e)}</td>  
\* <td>{****@link*** *#offer offer(e)}</td>  
\* <td>{****@link*** *#put put(e)}</td>  
\* <td>{****@link*** *#offer(Object, long, TimeUnit) offer(e, time, unit)}</td>  
\* </tr>*

*<tr>  
\* <td><b>Remove</b></td>  
\* <td>{****@link*** *#remove remove()}</td>  
\* <td>{****@link*** *#poll poll()}</td>  
\* <td>{****@link*** *#take take()}</td>  
\* <td>{****@link*** *#poll(long, TimeUnit) poll(time, unit)}</td>  
\* </tr>*

*<tr>  
\* <td><b>Examine</b></td>  
\* <td>{****@link*** *#element element()}</td>  
\* <td>{****@link*** *#peek peek()}</td>  
\* <td><em>not applicable</em></td>  
\* <td><em>not applicable</em></td>  
\* </tr>*

**Retrieving or Querying the Contents of the Queue**

int drainTo(Collection<? super E> c)

// removes elements from queue and insert them to collection . It is efficient than multiple poll()

int drainTo(Collection<? super E> c, int maxElements)

// clear at most the specified number of elements into c

int remainingCapacity()

// return the number of elements that would be accepted

// without blocking, or Integer.MAX\_VALUE if unbounded

**Implementing Blocking Queue**

**LinkedBlockingQueue:**

* 1. It is a thread safe, FIFO order based on Linked node structures.
  2. It may be bounded/unbounded
  3. It uses Reentarnt locks : put lock(not full Condition) and take lock(not empty)

Summary :

Ordering : As link list , so insertion order FIFO

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: no as concurrent operations

Thread Safe : Yes

Iterator : Weekly consistent, returns insertion order

**Array Blocking Queue:**

1. It is a liner representation of a circular array in which start and last index of arrays are logically adjacent.
2. Whenever an element is inserted , tail pointer is increased
3. On removal , head pointer is increased
4. When any of the pointer reaches last index, then its next index will be 0
5. When both are at same index , means queue is full

The reason of using circular array rather than liner array is that in normal array insertion at end will o(n)/removal from beginning needs shifting of elements.

In case of circular array , this can be achieved in constant time.

It also has two locks.

Summary :

Bounded

Ordering : Circular array insertion order FIFO

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: no as concurrent operations

Thread Safe : Yes

Iterator : Weekly consistent, returns insertion order

**Priority Blocking Queue:**

It is the blocked version of priority queue.returns least priority element

Summary :

Bounded

Ordering : no order

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: no as concurrent operations

Thread Safe : Yes

Iterator : weekly consistent as it work on snapshot.

**Delay Queue:**

1) It is a special version of Priority Blocking Queue.

2) It maintains element as per delay.

3) It works for items implementing Delayed interface which has long getDelay(TimeUnit unit) method.

4) Poll() will return the element with longest delay in case , if no ones delay is expired, returns null.

5)Peek() returns the element which is going to be pull first based on its delay

**Since it is extending PriorityBlockedQueue. It has all properties of it.**

Summary :

Ordering : Based on Delay

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: no as concurrent operations

Thread Safe : Yes

Iterator :

weekly consistent as it work on snapshot.

Example:

|  |
| --- |
| public class DelayObject implements Delayed { |

|  |  |
| --- | --- |
| 10 | private String data; |

|  |  |
| --- | --- |
| 11 | private long startTime; |

|  |  |
| --- | --- |
| 12 |  |

|  |  |
| --- | --- |
| 13 | public DelayObject(String data, long delay) { |

|  |  |
| --- | --- |
| 14 | this.data = data; |

|  |  |
| --- | --- |
| 15 | this.startTime = System.currentTimeMillis() + delay; |

|  |  |
| --- | --- |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |

|  |  |
| --- | --- |
| 18 | @Override |

|  |  |
| --- | --- |
| 19 | public long getDelay(TimeUnit unit) { |

|  |  |
| --- | --- |
| 20 | long diff = startTime - System.currentTimeMillis(); |

|  |  |
| --- | --- |
| 21 | return unit.convert(diff, TimeUnit.MILLISECONDS); |

|  |  |
| --- | --- |
| 22 | } |

|  |  |
| --- | --- |
| 23 |  |

|  |  |
| --- | --- |
| 24 | @Override |

|  |  |
| --- | --- |
| 25 | public int compareTo(Delayed o) { |

|  |  |
| --- | --- |
| 26 | if (this.startTime < ((DelayObject) o).startTime) { |

|  |  |
| --- | --- |
| 27 | return -1; |

|  |  |
| --- | --- |
| 28 | } |

|  |  |
| --- | --- |
| 29 | if (this.startTime > ((DelayObject) o).startTime) { |

|  |  |
| --- | --- |
| 30 | return 1; |

|  |  |
| --- | --- |
| 31 | } |

|  |  |
| --- | --- |
| 32 | return 0; |

|  |  |
| --- | --- |
| 33 | } |

|  |  |
| --- | --- |
| 34 |  |

|  |  |
| --- | --- |
| 35 | @Override |

|  |  |
| --- | --- |
| 36 | public String toString() { |

|  |  |
| --- | --- |
| 37 | return "{" + |

|  |  |
| --- | --- |
| 38 | "data='" + data + '\'' + |

|  |  |
| --- | --- |
| 39 | ", startTime=" + startTime + |

|  |  |
| --- | --- |
| 40 | '}'; |

|  |  |
| --- | --- |
| 41 | } |

|  |  |
| --- | --- |
| 42 | } |

**SynchronousQueue:**

* 1. It don’t have any capacity.
  2. It only accept element when consumer thread is ready to consume it.
  3. Its iterator will be always empty as there is no element. It is like an empty collection.
  4. There is no ordering.
  5. It works on special CSP and ADA instructions

1. public class SynchronousQueueDemo{
2. public static void main(String args[]) {
3. final SynchronousQueue<String> queue = new SynchronousQueue<String>();
4. Thread producer = new Thread("PRODUCER") {
5. public void run() {
6. String event = "FOUR";
7. try {
8. queue.put(event); // thread will block here
9. System.out.printf("[%s] published event : %s %n", Thread
10. .currentThread().getName(), event);
11. } catch (InterruptedException e) {
12. e.printStackTrace();
13. }
14. }
15. };
16. producer.start(); // starting publisher thread
17. Thread consumer = new Thread("CONSUMER") {
18. public void run() {
19. try {
20. String event = queue.take(); // thread will block here
21. System.out.printf("[%s] consumed event : %s %n", Thread
22. .currentThread().getName(), event);
23. } catch (InterruptedException e) {
24. e.printStackTrace();
25. }
26. }
27. };
28. consumer.start(); // starting consumer thread
29. }
30. }
31. Output:
32. [CONSUMER] consumed event : FOUR
33. [PRODUCER] published event : FOUR
34. If you have send the output carefully then you would have noticed that order of events are reversed. Seems [CONSUMER] thread is consuming data even before [PRODUCER] thread has produced it. This happens because by default SynchronousQueue doesn't guarantee any order, but it has a fairness policy, which if set to true allows access to threads in FIFO order. You can enable this fairness policy by passing true to [overloaded constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) of SynchronousQueue i.e. new SynchronousQueue(boolean fair).  
      
    if we comment the consumer.start(), we wont see the output as producer thread will be blocked.

**Deque**

1)It is double ended queue.

2)It differs with queue in one more thing is that queue does not have any order for elements present in it but it has elements present in their insertion order. That’s why it don’t have any Priority Queue implementation.

3)If elements are removed and inserted at the same end, then it will behave like stack (LIFO)

4) It has queue methods with first and last suffix.

*<td></td>  
\* <td ALIGN=CENTER><em>Throws exception</em></td>  
\* <td ALIGN=CENTER><em>Special value</em></td>  
\* <td ALIGN=CENTER><em>Throws exception</em></td>  
\* <td ALIGN=CENTER><em>Special value</em></td>  
\* </tr>  
\* <tr>  
\* <td><b>Insert</b></td>  
\* <td>{****@link*** *Deque#addFirst addFirst(e)}</td>  
\* <td>{****@link*** *Deque#offerFirst offerFirst(e)}</td>  
\* <td>{****@link*** *Deque#addLast addLast(e)}</td>  
\* <td>{****@link*** *Deque#offerLast offerLast(e)}</td>  
\* </tr>  
\* <tr>  
\* <td><b>Remove</b></td>  
\* <td>{****@link*** *Deque#removeFirst removeFirst()}</td>  
\* <td>{****@link*** *Deque#pollFirst pollFirst()}</td>  
\* <td>{****@link*** *Deque#removeLast removeLast()}</td>  
\* <td>{****@link*** *Deque#pollLast pollLast()}</td>  
\* </tr>  
\* <tr>  
\* <td><b>Examine</b></td>  
\* <td>{****@link*** *Deque#getFirst getFirst()}</td>  
\* <td>{****@link*** *Deque#peekFirst peekFirst()}</td>  
\* <td>{****@link*** *Deque#getLast getLast()}</td>  
\* <td>{****@link*** *Deque#peekLast peekLast()}</td>  
\* </tr>*

*\* <td ALIGN=CENTER> <b>Stack Method</b></td>  
\* <td>{****@link*** *#push push(e)}</td>  
\* <td>{****@link*** *#pop pop()}</td>*

**Implementing Deque**

**ArrayDeque:** It has circular array implementation that’s why insertion/deletion can be done in constant time.

Summary :

Ordering : Insertion

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: no

Thread Safe : No

Iterator : fail fast

**LinkedList:** It is legacy implementation of Dequqe. It is the only queue accepting null.

It has two pointers next and previous.

Summary :

Ordering : Insertion

Random Access: No index based access

Key-Value : no

Duplicate : yes can be

Null: Yes

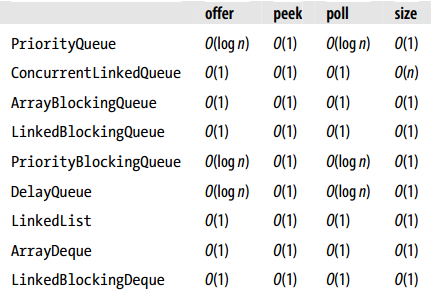
Thread Safe : No

Iterator : fail fast

**Blocking Deque interface**

It adds below methods to Deque

*<td></td>  
\* <td ALIGN=CENTER><em>Throws exception</em></td>  
\* <td ALIGN=CENTER><em>Special value</em></td>  
\* <td ALIGN=CENTER><em>Blocks</em></td>  
\* <td ALIGN=CENTER><em>Times out</em></td>  
\* </tr>  
\* <tr>  
\* <td><b>Insert</b></td>  
\* <td>{****@link*** *#addFirst addFirst(e)}</td>  
\* <td>{****@link*** *#offerFirst(Object) offerFirst(e)}</td>  
\* <td>{****@link*** *#putFirst putFirst(e)}</td>  
\* <td>{****@link*** *#offerFirst(Object, long, TimeUnit) offerFirst(e, time, unit)}</td>  
\* </tr>  
\* <tr>  
\* <td><b>Remove</b></td>  
\* <td>{****@link*** *#removeFirst removeFirst()}</td>  
\* <td>{****@link*** *#pollFirst pollFirst()}</td>  
\* <td>{****@link*** *#takeFirst takeFirst()}</td>  
\* <td>{****@link*** *#pollFirst(long, TimeUnit) pollFirst(time, unit)}</td>  
\* </tr>  
\* <tr>  
\* <td><b>Examine</b></td>  
\* <td>{****@link*** *#getFirst getFirst()}</td>  
\* <td>{****@link*** *#peekFirst peekFirst()}</td>  
\* <td><em>not applicable</em></td>  
\* <td><em>not applicable</em></td>  
\* </tr>  
\* <tr>  
\* <td ALIGN=CENTER COLSPAN = 5> <b>Last Element (Tail)</b></td>  
\* </tr>  
\* <tr>  
\* <td></td>  
\* <td ALIGN=CENTER><em>Throws exception</em></td>  
\* <td ALIGN=CENTER><em>Special value</em></td>  
\* <td ALIGN=CENTER><em>Blocks</em></td>  
\* <td ALIGN=CENTER><em>Times out</em></td>  
\* </tr>  
\* <tr>  
\* <td><b>Insert</b></td>  
\* <td>{****@link*** *#addLast addLast(e)}</td>  
\* <td>{****@link*** *#offerLast(Object) offerLast(e)}</td>  
\* <td>{****@link*** *#putLast putLast(e)}</td>  
\* <td>{****@link*** *#offerLast(Object, long, TimeUnit) offerLast(e, time, unit)}</td>  
\* </tr>  
\* <tr>  
\* <td><b>Remove</b></td>  
\* <td>{****@link*** *#removeLast() removeLast()}</td>  
\* <td>{****@link*** *#pollLast() pollLast()}</td>  
\* <td>{****@link*** *#takeLast takeLast()}</td>  
\* <td>{****@link*** *#pollLast(long, TimeUnit) pollLast(time, unit)}</td>  
\* </tr>  
\* <tr>  
\* <td><b>Examine</b></td>  
\* <td>{****@link*** *#getLast getLast()}</td>  
\* <td>{****@link*** *#peekLast peekLast()}</td>  
\* <td><em>not applicable</em></td>  
\* <td><em>not applicable</em></td>  
\* </tr>  
\* </table>*



ConcurrentLinkedQueue : size() it calculated not maintained using pointer as it uses CAS operations.

PriorityQueue and Delay queue which is a type of Priority queue has poll and offer logn time as it requires rearrangement.

Ques)How to decide which queue to use ?

Ans) ask whether we want concurrent access or not, order ?

If not concurrent ….order FIFO ….ArrayDeque…priority order…..Priority Deque or Delay

If we don’t want blocking methods…use ConcurrentLinkedQueue

**Chapter 15 List**

1)List is different from set as it can contains duplicate

2)It is different from Queue as it can maintains insertion order and elements can be accessed by index.

**Positional Access**

void add(int index, E e) // add element e at given index

boolean addAll(int index, Collection<? extends E> c)

// add contents of c at given index

E get(int index) // return element with given index

E remove(int index) // remove element with given index

E set(int index, E e) // replace element with given index by e

**Search** It returns -1 if element is not present

int indexOf(Object o) // return index of first occurrence of o

int lastIndexOf(Object o) // return index of last occurrence of o

**RangeView**

List<E> subList(int fromIndex, int toIndex)

Includes from index and exclude toIndex

On changing to sub list , as sub list are just pointers to the original list , changes will be reflected to original.

While on changing(insertion/removal) to original list , accessing sub list will throw Concurrent modification Exception as sub list is working on pointers to original list and that pointers will no longer valid.

*List<Integer> list = new ArrayList();  
for (int i = 0; i < 10; i++) {  
 list.add(i);  
}  
List<Integer> subList = list.subList(3, 5);  
System.out.println("original list" + list);  
System.out.println("view list" + subList);  
  
System.out.println("Changes to sub list");  
subList.add(10);  
System.out.println("original list" + list);  
System.out.println("view list" + subList);  
  
System.out.println("Changes to source list");  
list.remove(new Integer(10));  
System.out.println("original list" + list);  
System.out.println("view list" + subList);*

Output :

original list[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

view list[3, 4]

Changes to sub list

original list[0, 1, 2, 3, 4, 10, 5, 6, 7, 8, 9]

view list[3, 4, 10]

Changes to source list

original list[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.ArrayList$SubList.checkForComodification(ArrayList.java:1231)

at java.util.ArrayList$SubList.listIterator(ArrayList.java:1091)

at java.util.AbstractList.listIterator(AbstractList.java:299)

at java.util.ArrayList$SubList.iterator(ArrayList.java:1087)

at java.util.AbstractCollection.toString(AbstractCollection.java:454)

at java.lang.String.valueOf(String.java:2982)

at java.lang.StringBuilder.append(StringBuilder.java:131)

at net.autodata.batch.ABCTest.main(ABCTest.java:28)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:497)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)

While in case of subset as sub set doesn’t work on indexes , any changes to sub set will reflected in original and changes to original will be reflected in sub set.

SortedSet<Integer> set = **new** TreeSet<>();  
**for** (**int** i = 0; i < 10; i++) {  
 set.add(i\*2);  
}  
SortedSet<Integer> subList = set.subSet(3, 5);  
System.***out***.println(**"original list"** + set);  
System.***out***.println(**"view list"** + subList);  
  
System.***out***.println(**"Changes to sub list"**);  
subList.add(3);  
System.***out***.println(**"original list"** + set);  
System.***out***.println(**"view list"** + subList);  
  
System.***out***.println(**"Changes to source list"**);  
set.remove(3);  
System.***out***.println(**"original list"** + set);  
System.***out***.println(**"view list"** + subList);

**Output:**

original list[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

view list[4]

Changes to sub list

original list[0, 2, 3, 4, 6, 8, 10, 12, 14, 16, 18]

view list[3, 4]

Changes to source list

original list[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

view list[4]

**List Iterator:** It is special Iterator present with List interface.

It can traverse in both directions. It can move forward as well as backward.

void add(E e); // insert the specified element into the list

boolean hasPrevious(); // return true if this list iterator has further

// elements in the reverse direction

int nextIndex(); // return the index of the element that would be

// returned by a subsequent call to next

E previous(); // return the previous element in the list

int previousIndex(); // return the index of the element that would be

void set(E e); // replace the last element returned by next or

// previous with the specified element

}

set() and remove() operations work on last returned item.

Eg 1,2,3

Iterator initially cursor = 0,next() =1 item returned is 1 and cursor = 1,so remove will remove item 1.

Set will also change 1 to new value but if I add , so it will add to cursor position i.e. index 1 ,….new element will be inserted final 1,new, 2,3

Eg :

**public static void** main(String[] args) {  
  
 ArrayList<Integer> list = **new** ArrayList<>();  
 **for** (**int** i = 0; i < 3; i++) {  
 list.add(i);  
 }  
 System.***out***.println(list);  
 *//print elemenst* ListIterator<Integer> listIterator = list.listIterator();  
 **while** (listIterator.hasNext()) {  
 Integer element = listIterator.next();  
 **if**(element.equals(1)){  
 listIterator.add(100);  
 }  
 }  
 System.***out***.println(list);  
}

output:

[0, 1, 2]

[0, 1, 100, 2]

**List interface implementation**

**1)Array List :**

It is backed by an array as it requires index based access.

Adding/removing will require shifting of element while get() can be done in o(1) .

**Ques) Why it is not implemented using Circular Array ?**

Ans) Circular Array has insertion/removal complexity in o(1) time only if it is done at index 0 else not. So if our list has less index based operations and more start index, we can use ArrayDeque.

Summary :

Ordering : Insertion

Random Access: index based access

Key-Value : no

Duplicate : yes can be

Null: YES

Thread Safe : No

Iterator : fail fast

**Linked List:** Already covered in queue. It is doubly ended queue. It has both methods. As Its list implementation it adds element on index bases but as Dequqe, it has addFirst() and addLast()

**CopyOnWriteArrayList:** For every write a copy is created, so use when we have less write and more reads.CopyOnwriteArraySet internally uses CopyOnWriteArrayList methods.

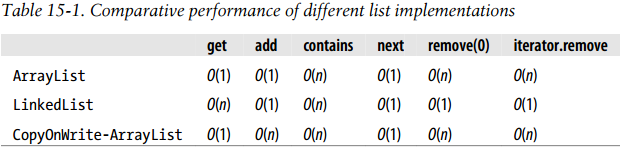
The class CopyOnWriteArraySetin fact delegates all of its operations to an instance of

CopyOnWriteArrayList, taking advantage of the atomic operations addIfAbsentand

addAllAbsentprovided by the latter to enable the Setmethods addand addAllto avoid

introducing duplicates to the set. In addition to the two standard constructors (see

Section 12.3),



Summary :

Ordering : Insertion

Random Access: index based access

Key-Value : no

Duplicate : yes can be

Null: YES

Thread Safe :YES

Iterator : Weeky Consistent

**Chapter 16 Map**

It is part of collection framrework i.e. it is in java.util package

It is the only Collection that does not extend Collection class reason being it works on key-value pair combination.

**Adding:**

V put(K key, V value) // add or replace a key-value association

// return the old value (may be null) if the

// key was present; otherwise returns null

void putAll(Map<? extends K,? extends V> m)

// add each of the key-value associations in

// the supplied map into the receiver

**Removing:**

void clear() // remove all associations from this map

V remove(Object key) // remove the association, if any, with the

// given key; returns the value with which it

// was associated, or null

**Querying the contents of map**

V get(Object k) // return the value corresponding to k, or

// null if k is not present as a key

boolean containsKey(Object k) // return true if k is present as a key

boolean containsValue(Object v) // return true if v is present as a value

int size() // return the number of associations

boolean isEmpty() // return true if there are no associations

As with the size method of Collection, the largest element count that can be reported

is Integer.MAX\_VALUE. Due to limitation of Integer range

**Providing Collection Views of the Keys, Values, or Associations:**

Set<Map.Entry<K, V>> entrySet() // return a Set view of the associations

Set<K> keySet() // return a Set view of the keys

Collection<V> values() // return a Collection view of the values

These functions are backed by original map, so the changes done here will be reflected back to the Map.

**Best Practice :** Remove element from the Map based on values

HashMap<Integer,String> map = **new** HashMap();  
map.put(1,**"1"**);  
map.put(2,**"2"**);  
map.put(3,**"3"**);  
map.put(4,**"4"**);  
  
Collection<String> collection = map.values();  
collection.removeAll(Collections.*singleton*(**"1"**));  
  
System.***out***.println(collection);  
System.***out***.println(map);

System.***out***.println(collection); //”2”,”3”,”4”  
System.***out***.println(map);//[{2,”2”},}{3,”3”},{4,”4”}]

Map.valuess() will define a collection which will inherit the AbstarcatCollection , and will override iterators etc to work on values.

**Map Implementation**

**HashMap:**

Hashmap is already discussed in HashSet.

Summary :

Ordering : no as elements are traversed as per bucket

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

Null: YES

Thread Safe :NO

Iterator : fail fast

**LinkedHashMap:**

It also works in same manner as Hashamp but it maintains insertion order as it has two pointers head and tail and each element points to it next element as discucssed in LinkedHashset.

It has a difference from LinkedHashset as it has a property called access order. By default it is false.

If it is set to true then the most get() element will be moved to the tail.

It also has a method called removeEldestEntry() . This method is called after put() and if returning true,so the oldest entry will be removed. By default it is retruning false.

Summary :

Ordering : insertion order and access order

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

Null: YES

Thread Safe :NO

Iterator : fail fast

**Ques) Implement LRU cache ?**

**Ans)** LRU cache removes the least recently used element from cache.

**public class** ABCTest {  
  
 **public static void** main(String[] args) {  
  
 LRU<Integer, Integer> lru = **new** LRU(4);  
  
 **for** (**int** i = 0; i < 5; i++) {  
 lru.put(i, i);  
 }  
  
 lru.get(4);  
 lru.get(3);  
 lru.put(5,5);  
  
 System.***out***.println(lru); //output: {2=2, 4=4, 3=3, 5=5}  
 }  
  
 **static class** LRU<K, V> **extends** LinkedHashMap<K, V> {  
  
 **private int maxEntries**;  
  
 **public** LRU(**int** maxEntries) {  
 **super**(16, .75f, **true**);  
 **this**.**maxEntries** = maxEntries;  
 }  
  
 **protected boolean** removeEldestEntry(Map.Entry<K, V> eldest) {  
 **return** size() > **maxEntries**;  
 }  
  
 }  
}

**WeakHashMap:**

* 1. Ordinary maps hold string reference of the key objects.
  2. So if key reference is null or key is not reachable , still value will be in map.
  3. To overcome this problem, WeakHashMap is introduced in which elements are stored as WeakReference, so as key becomes unreachable, it is removed from map.
  4. Before most of its operations, it check which values have been removed. Whenever GC collects any memory, it put this information on ReferenceQueue.
  5. This is good for usage where we have resource as key , so releasing them will fix memory leak problem.
  6. Its drawback is that it works on key, while in many cases value objects may hold large memory.

Example:

**public class** ABCTest {  
  
 **public static void** main(String[] args) {  
  
 WeakHashMap<Person, Integer> map = **new** WeakHashMap<>();  
 net.autodata.batch.ABCTest.Person person = **new** net.autodata.batch.ABCTest.Person(123);  
 map.put(person,123);  
  
 person = **null**;  
  
 System.*gc*();  
  
 System.***out***.println(map); //output : {} empty map  
 }  
  
 **static class** Person {  
  
 **private** Integer **id**;  
  
 **public** Person(Integer id) {  
 **this**.**id** = id;  
 }  
  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
  
 Person person = (Person) o;  
  
 **return id** != **null** ? **id**.equals(person.**id**) : person.**id** == **null**;  
  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return id** != **null** ? **id**.hashCode() : 0;  
 }  
 }  
}

**Source Code:**

**private static class** Entry<K,V> **extends** WeakReference<Object> **implements** Map.Entry<K,V> {  
 V **value**;  
 **final int hash**;  
 Entry<K,V> **next**;  
  
 */\*\*  
 \* Creates new entry.  
 \*/* Entry(Object key, V value,  
 ReferenceQueue<Object> queue,  
 **int** hash, Entry<K,V> next) {  
 **super**(key, queue);  
 **this**.**value** = value;  
 **this**.**hash** = hash;  
 **this**.**next** = next;  
 }  
  
 @SuppressWarnings(**"unchecked"**)  
 **public** K getKey() {  
 **return** (K) WeakHashMap.*unmaskNull*(get());  
 }  
  
 **public** V getValue() {  
 **return value**;  
 }  
  
 **public** V setValue(V newValue) {  
 V oldValue = **value**;  
 **value** = newValue;  
 **return** oldValue;  
 }  
  
 **public boolean** equals(Object o) {  
 **if** (!(o **instanceof** Map.Entry))  
 **return false**;  
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;  
 K k1 = getKey();  
 Object k2 = e.getKey();  
 **if** (k1 == k2 || (k1 != **null** && k1.equals(k2))) {  
 V v1 = getValue();  
 Object v2 = e.getValue();  
 **if** (v1 == v2 || (v1 != **null** && v1.equals(v2)))  
 **return true**;  
 }  
 **return false**;  
 }  
  
 **public int** hashCode() {  
 K k = getKey();  
 V v = getValue();  
 **return** Objects.*hashCode*(k) ^ Objects.*hashCode*(v);  
 }  
  
 **public** String toString() {  
 **return** getKey() + **"="** + getValue();  
 }  
}

Summary :

Ordering : no

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

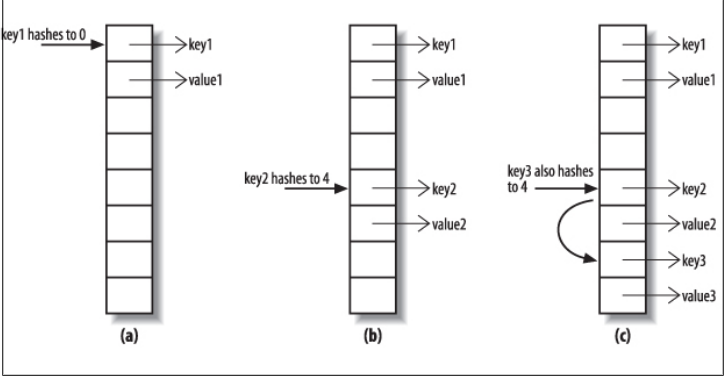
Null: YES

Thread Safe :NO

Iterator : fail fast

**Identity Hash Map**

1. It is different from HashMap as it not works on equal but the identity (==)
2. For it, two objects are equal if they are physically same.
3. It can be used while processing a graph as to find that graph nodes are same, one way is to compare node,its child etc. But if simply check that they are same node using identity will be easy option.
4. It works on linear probing rather than chaining.



1. Linear probing is faster than chaining as accessing array elements are faster as they are sequential in memory while in link list memory location is not near by
2. Inserting in link list will require creating new node i.e. not in case of array

**Ques) Why linear probing it is not used Hashmap and other collections. ?**

**Ans)**Reason is hashmap stores hash, key and value.It compares hash before equals as equals is expensive operation.So if go by this array will be full soon.While in indentity hash map , reference is placed directly on memory as it not requires hash.

**Ques)** **public IdentityHashMap()**

**public IdentityHashMap(Map<? extends K,? extends V> m)**

**public IdentityHashMap(int expectedMaxSize)**

**Why it not has initial load define method ?**

**Ans)**User cant define load factor as wrong value will unnecessary cause resizing and may spoil array

Summary :

Ordering : no

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

Null: YES

Thread Safe :NO

Iterator : fail fast

There is also a wekIdentityHashMap having qualities of both map.

**EnumMap:**

1. It has an array for Keys and a separate arrays for values.
2. Array index is calculated based on ordinal position.
3. Iterator iterates in natural order of enums that is the order in which enums are saved

**public class** EnumMap<K **extends** Enum<K>, V> **extends** AbstractMap<K, V>  
 **implements** java.io.Serializable, Cloneable  
{  
 */\*\*  
 \* The <tt>Class</tt> object for the enum type of all the keys of this map.  
 \*  
 \** ***@serial*** *\*/* **private final** Class<K> **keyType**;  
  
 */\*\*  
 \* All of the values comprising K. (Cached for performance.)  
 \*/* **private transient** K[] **keyUniverse**;  
  
 */\*\*  
 \* Array representation of this map. The ith element is the value  
 \* to which universe[i] is currently mapped, or null if it isn't  
 \* mapped to anything, or NULL if it's mapped to null.  
 \*/* **private transient** Object[] **vals**;

**public** V put(K key, V value) {  
 typeCheck(key);  
  
 **int** index = key.ordinal();  
 Object oldValue = **vals**[index];  
 **vals**[index] = maskNull(value);  
 **if** (oldValue == **null**)  
 **size**++;  
 **return** unmaskNull(oldValue);  
}

Summary :

Ordering : Enum natural order

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

Null: no as only enum can be saved

Thread Safe :NO

Iterator : weekly consistent

Example :

**public class** ABCTest {  
  
 **enum** Season {***SUMMER***,***WINTER***,***AUTUMN***};  
  
 **public static void** main(String[] args) {  
  
 EnumMap<Season,Integer> seasonEnumMap = **new** EnumMap(Season.**class**);  
 seasonEnumMap.put(Season.***AUTUMN***,2);  
 seasonEnumMap.put(Season.***WINTER***,1);  
 seasonEnumMap.put(Season.***SUMMER***,0);  
  
 seasonEnumMap.entrySet().forEach(i -> System.***out***.println(i));  
  
 System.***out***.println(**"Removing Enum Set"**);  
 seasonEnumMap.remove(Season.***WINTER***);  
  
 seasonEnumMap.entrySet().forEach(i -> System.***out***.println(i));  
 }  
}

output:

SUMMER=0

WINTER=1

AUTUMN=2

Removing Enum Set

SUMMER=0

AUTUMN=2

**SortedMap Interface**

1. It ensures that elements are stored in a sorted manner based on comparator if provided , if not based on natural ordering.
2. The equals method must be consistent with equals.
3. **Getting the First and Last Elements**

K firstKey()

K lastKey()

1. **Retrieving the Comparator**

Comparator<? super K> comparator()

This method returns the map’s key comparator if it has been given one, instead of

relying on the natural ordering of the keys. Otherwise, it returns null.

1. **Finding Subsequences**

SortedMap<K,V> subMap(K fromKey, K toKey)

SortedMap<K,V> headMap(K toKey)

SortedMap<K,V> tailMap(K fromKey)

Like sorted set, from and to value need not be present in map. It includes from , excludes to and changes done either in map or view will be reflected in each other.

**Navigable Map**

1. It is same as Navigable set

**Getting the First and Last Elements**

Map.Entry<K,V> pollFirstEntry()

Map.Entry<K,V> pollLastEntry()

Map.Entry<K,V> firstEntry()

Map.Entry<K,V> lastEntry()

**Getting Range Views**

NavigableMap<K,V> subMap(

K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)

NavigableMap<K,V> headMap(K toKey, boolean inclusive)

NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)

Like Navigable Set , they also have arguments to specify to include from and to.

**Getting Closest Matches**

Map.Entry<K,V> ceilingEntry(K Key)

K ceilingKey(K Key)

Map.Entry<K,V> floorEntry(K Key)

K floorKey(K Key)

Map.Entry<K,V> higherEntry(K Key)

K higherKey(K Key)

Map.Entry<K,V> lowerEntry(K Key)

K lowerKey(K Key)

**Navigating the Map**

NavigableMap<K,V> descendingMap() // return a reverse-order view of the map

NavigableSet<K> descendingKeySet() // return a reverse-order key set

**Navigable Map Implementation**

1. **TreeMap:** It is similar to TreeSet. It implementes Red Black tree and hence get(),put() and remove() perform in logn time.

**Summary :**

Ordering : Sorted order

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

Null: no as null cant be compared

Thread Safe :No

Iterator : fail fast

**Concurrent Map Interface**

**ConcurrentHashMap:**

1. It is better to use it than Collections.synchronized
2. For read operations, it does not block.
3. For write , it blocks at bucket level. Each bucket has its own write lock.
4. To calculate size, it has to hold lock on all buckets,getting their size and then add, so this operation is expensive.

For read operations, it uses voliatile

**Summary :**

Ordering : no

Random Access: yes, we can access value for a key

Key-Value : yes

Duplicate : no duplicate key,if placed old value for key will be replaced

Null: No as concurrent operations

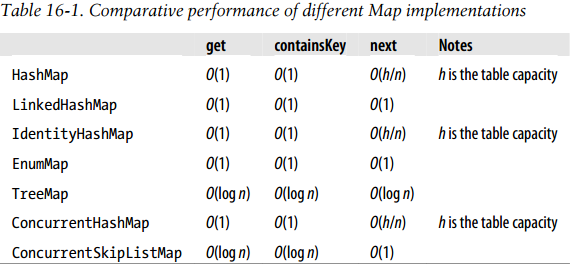
Thread Safe : Yes

Iterator : Weekly consistent

**Concurrent Navigable Map interface**

It inherits both ConcurrentMap and NavigableMap. Its implementation is ConcurrentSkipListMap.

**ConcurrentSkipListMap:** same as concurrentskiplist set. Only difference is that in case if key is already present, it will replace its value.



**Chapter 17 Collections Class**

**Changing the Order of List Elements**

1. void reverse(List<?> list)

// reverse the order of the elements

Complexity : o(n)

1. void rotate(List<?> list, int distance)

// rotate the elements of the list; the element at index

// i is moved to index (distance + i) % list.size()

Complexity : o(n)

1. void shuffle(List<?> list)

// randomly permute the list elements

1. void shuffle(List<?> list, Random rnd)

// randomly permute the list using the randomness source rnd

It uses swap(index,Random index) …it generates random index using specified Random class

Complexity : o(n)

1. <T extends Comparable<? super T>> void sort(List<T> list)

// sort the supplied list using natural ordering

<T> void sort(List<T> list, Comparator<? super T> c)

// sort the supplied list using the supplied ordering

Complexity : It copies element to array, then uses merge sort : o(nlogn)

1. void swap(List<?> list, int i, int j)

// swap the elements at the specified positions

Complexity: o(constant time)

**Changing the contents of a list:**

<T> void copy(List<? super T> dest, List<? extends T> src)

// copy all of the elements from one list into another

<T> void fill(List<? super T> list, T obj)

// replace every element of list with obj

<T> boolean replaceAll(List<T> list, T oldVal, T newVal)

// replace all occurrences of oldVal in list with newVal

**Finding Extreme Values in Collection:**

<T extends Object & Comparable<? super T>>

T max(Collection<? extends T> coll) // return the maximum element

// using natural ordering

<T> T max(Collection<? extends T> coll, Comparator<? super T> comp)

// return the maximum element

// using the supplied comparator

<T extends Object & Comparable<? super T>>

T min(Collection<? extends T> coll) // return the minimum element

// using natural ordering

<T> T min(Collection<? extends T> coll, Comparator<? super T> comp)

// return the minimum element

// using the supplied comparator

**Finding Specific Values in a List**

<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

// search for key using binary search

<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

// search for key using binary search

int indexOfSubList(List<?> source, List<?> target)

// find the first sublist of source which matches target

int lastIndexOfSubList(List<?> source, List<?> target)

// find the last sublist of source which matches target

**Collection Factories:**

**Empty Collection**

<T> List<T> emptyList() // return the empty list (immutable)

<K,V> Map<K,V> emptyMap() // return the empty map (immutable)

<T> Set<T> emptySet() // return the empty set (immutable)

**Collection with Single Element:**

<T> Set<T> singleton(T o)

// return an immutable set containing only the specified object

<T> List<T> singletonList(T o)

// return an immutable list containing only the specified object

<K,V> Map<K,V> singletonMap(K key, V value)

// return an immutable map, mapping only the key K to the value V

**Create n number of copies of an Object:**

<T> List<T> nCopies(int n, T o)

// return an immutable list containing n references to the object o

**Wrappers:**

The collection class provides wrapper objects that modify the behavior of standard collections classes in three ways: synchronized, unmodifiable,checked

These function works on Proxy objects. They create proxy objects using the nested classes defined inside the Collections class. Eg

**public static** <E> Collection<E> checkedCollection(Collection<E> c,  
 Class<E> type) {  
 **return new** CheckedCollection<>(c, type);  
}

**static class** CheckedCollection<E> **implements** Collection<E>, Serializable {  
 **private static final long *serialVersionUID*** = 1578914078182001775L;  
  
 **final** Collection<E> **c**;  
 **final** Class<E> **type**;

**Collections.synchronized() :** It holds the lock on whole collection.

<T> Collection<T> synchronizedCollection(Collection<T> c);

<T> Set<T> synchronizedSet(Set<T> s);

<T> List<T> synchronizedList(List<T> list);

<K, V> Map<K, V> synchronizedMap(Map<K, V> m);

<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);

<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m);

**Collections.unmodifiable():**Adding or removing will throw Exception

<T> Collection<T> unmodifiableCollection(Collection<? extends T> c)

<T> Set<T> unmodifiableSet(Set<? extends T> s)

<T> List<T> unmodifiableList(List<? extends T> list)

<K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)

<T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s)

<K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m)

**Checked Collections:** They ensure that all elements in collection should of the specified type.

static <E> Collection

checkedCollection(Collection<E> c, Class<E> elementType)

static <E> List

checkedList(List<E> c, Class<E> elementType)

static <E> Set

checkedSet(Set<E> c, Class<E> elementType)

static <E> SortedSet

checkedSortedSet(SortedSet<E> c, Class<E> elementType)

static <K, V> Map

checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)

static <K, V> SortedMap

checkedSortedMap(SortedMap<K, V> c, Class<K> keyType,Class<V> valueType)

**Other Methods:**

**addAll**

<T> boolean addAll(Collection<? super T> c, T... elements)

// adds all of the specified elements to the specified collection.

**asLifoQueue**

<T> Queue<T> asLifoQueue(Deque<T> deque)

// returns a view of a Deque as a Last-in-first-out (Lifo) Queue.

Recall from Chapter 14 that while queues can impose various different orderings on

their elements, there is no standard Queueimplementation that provides LIFO ordering.

Dequeueimplementations, on the other hand, all support LIFO ordering if elements are

removed from the same end of the dequeue as they were added. The method asLifo

Queueallows you to use this functionality through the conveniently concise Queuein-terface.

**Disjoint**

boolean disjoint(Collection<?> c1, Collection<?> c2)

// returns true if c1 and c2 have no elements in common

**Enumeration**

<T> Enumeration<T> enumeration(Collection<T> c)

// returns an enumeration over the specified collection

**frequency**

int frequency(Collection<?> c, Object o)

// returns the number of elements in c that are equal to o

**list**

<T> ArrayList<T> list(Enumeration<T> e)

// returns an ArrayList containing the elements returned by the specified Enumeration

**newSetFromMap**

<E> Set<E> newSetFromMap(Map<E, Boolean> map)

// returns a set backed by the specified map.

As we saw earlier, many sets (such as TreeSetand NavigableSkipListSet) are imple-mented by maps, and share their ordering, concurrency, and performance character-istics. Some maps, however (such as WeakHashMapand IdentityHashMap) do not have

standard Setequivalents. The purpose of the method newSetFromMapis to provide

equivalent Setimplementations for such maps. The method newSetFromMapwraps its

argument, which must be empty when supplied and should never be subsequently

accessed directly. This code shows the standard idiom for using it to create a weak

HashSet, one whose elements are held via weak references:

Set<Object> weakHashSet = Collections.newSetFromMap(

new WeakHashMap<Object, Boolean>());

**reverseOrder**

<T> Comparator<T> reverseOrder()

// returns a comparator that reverses natural ordering