Collection Notes Part 2

1. Introduction to Collections Framework

- A collections framework is a unified architecture for representing & manipulating collections, enabling collections to be manipulated independently of implementation details. All collections frameworks contain the following:
  - **a) Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In Object oriented language, interfaces generally form a hierarchy.
  - **b) Implementations:** These are the concrete implementations of the collections interfaces. In essence, they're reusable data structures.
  - **c) Algorithms:** These are the methods that perform useful computations, such as searching & sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic i.e. same method can be used on many different implementations of the appropriate collection interface.

Apart from the Java collections framework, the best – known examples of collections frameworks are the C++ Standard Template library (STL).

Benefits of the Java Collections Framework

1. **Reduces programming effort**: By providing useful data structures & algorithms so you don't have to write them yourself.
2. **Increases program speed & quality:** This collections framework provides high – performance, high – quality implementations of useful data structures & algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing our own data structures, you'll have more time to devote to improving program's quality & performance.
3. **Allows interoperability between unrelated**: The collection interfaces are the vernacular by which APIs pass collections back & forth. If my network administration API furnished a collection of node names & if your GUI toolkit expects a collection of column headings, our APIs will interoperated seamlessly, even though they were written independently.
4. **Reduces the effort required to learn & to use new APIs**
5. **Reduces the effort required to design & implement:** Designers & Implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
6. **Fosters software reuse:** Fosters software reuse by providing a standard interface for collections & algorithms with which to manipulate them.

2. Array

- An array is an indexed collection of fixed no. of homogenous data elements.
- The main advantage of Array is we can represent multiple values by using single variable so that readability of the code will be improved.
- Limitations of Array are
  - **a)** Fixed in size i.e., once we create an array, there is no chance of increasing/decreasing the size based on our requirement. Due to this, to use Array's concept, compulsory we should know the size in advance which may not possible always.
  - **b)** Array can hold only homogenous datatype elements. (Though we can use **Object** type array)
  - **c)** Array concept is not implemented based on some standard data Structure & hence readymade method support is not available i.e., for every requirement, we have to write the code explicitly which increases complexity of programming.

- To overcome above limitations of Array, we should go for **Collection** concept.
- Advantages of Collection are
  - **a)** Collections are growable in nature i.e., based on our requirement we can increase/decrease the size.
  - **b)** Collections can hold both homogenous & heterogeneous elements.

c) Every collection class is implemented based on some standard data Structure hence for every requirement, readymade method support is available.

Array Vs Collection

No.	Array	Collection
1.	Fixed in size.	Growable in nature.
2.	With respect to memory, Arrays are not recommended to use.	With respect to memory, Collections are recommended to use.
3.	With respect to performance, Arrays are recommended to use.	With respect to performance, Collections are not recommended to use.
4.	Arrays can hold only homogenous datatype elements.	Collections can hold both homo & heterogeneous datatype elements.
5.	No underlying data structure hence no readymade methods are available.	Every collection class has underlying data structure hence readymade methods are available.
6.	Arrays can hold both primitive & objects.	Collections can hold only object types but not primitive.

ArrayList Vs LinkedList

No.	ArrayList	LinkedList
1.	ArrayList is the best choice if our frequent operation is retrieval operation because it implements RandomAccess interface.	LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
2.	It is worst choice if our frequent operation is insertion/ deletion in the middle due to shift operation.	It is the worst choice if our frequent operation is retrieval operation.
3.	In ArrayList, the elements will be stored in consecutive memory locations & hence retrieval operation will become easy.	In LinkedList, the elements won't be stored in consecutive memory locations & hence retrieval operation will become difficult.

ArrayList Vs Vector

No.	ArrayList	Vector
1.	Every method present in the ArrayList is non synchronized.	Every method present in the vector is synchronized.
2.	At a time, multiple threads are allowed to operate on ArrayList object & hence it is not thread – safe.	At a time, only on thread is allowed to operate on vector object & hence it is thread – safe.
3.	Relatively performance is high because threads are not required to wait to operate on ArrayList object.	Relatively performance is low because threads are required to wait to operate on Vector object.
4.	Introduced in 1.2 V & it is non – legacy.	It is introduced in 1.0 V & it is legacy.

3 Cursors of Java Collections Framework

- If we want to get Objects one by one from a collection, then we should go for cursor.
- There are 3 types of cursors available in Java

1. Enumeration Interface
2. Iterator Interface
3. ListIterator Interface

1. Enumeration Interface

- We can use Enumeration to get objects one by one from legacy collection object.
- We can create Enumeration object by using element () method of Vector class.

- **Methods**
  1. **boolean hasMoreElements ()** – tests if the enumeration contains more elements.
  2. **E nextElement ()** – returns the next element of the enumeration.

- **Limitations of Enumeration**
  1. We can apply enumeration concept only for legacy classes & it is not universal cursor.
  2. By using Enumeration, we can get only read access & we can't perform remove operation.

- To overcome these limitations, we should go for Iterator.

2. Iterator Interface

- We can apply Iterator concept for any collection object; hence it is Universal Cursor.
- By using **Iterator** we can perform both read & remove operations.
- We can create Iterator object by using iterator () method present in Collection interface.

- **Methods**
  1. **boolean hasNext ()** – returns true if the iteration has more elements.
  2. **E next ()** – returns the next element in the iteration.
  3. **void remove ()** – removes from the underlying collection the last element returned by the iterator.

- **Limitations of Enumeration**
  1. By using Enumeration & Iterator, we can always move only towards forward direction not backward direction. So, these are single direction cursors not bidirectional cursor.
  2. By using Iterator, we can get only read & remove operation & we can't perform replacement or addition of new objects.

- To overcome these limitations, we should go for **ListIterator**.

3. ListIterator Interface

- It is child interface of Iterator; hence all methods of Iterator by default available to the ListIterator.
- ListIterator is a bidirectional cursor.
- By using ListIterator, we can perform replacement & addition of new objects along with read & remove operations.
- We can create ListIterator by using listIterator () method of List interface.

- **Methods**
  1. **void add (E e)** – inserts the specified element into the list.
  2. **boolean hasNext ()** – returns true if the list iterator has more elements when traversing the list in forward direction.
  3. **boolean hasPrevious ()** – returns true if the list iterator has more elements when traversing the list in reverse direction.
  4. **E next ()** – returns the next element in the list & advances the cursor position.
  5. **int nextIndex ()** – returns the index of the element that would be returned by a subsequent call to next ().
  6. **E previous ()** – returns the previous element in the list & moves the cursor position backwards.

7. **int previousIndex ()** – returns the index of the element that would be returned by a subsequent call to previous ().
8. **void remove ()** – removes from the list the last element returned by next () or previous ().
9. **void set (E)** – replaces the last element returned by next () or previous () with the specified element.

- The most powerful cursor is ListIterator but its limitation is it is applicable only for List objects.

No.	Property	Enumeration	Iterator	ListIterator
1.	Where we can apply?	Only for legacy classes	For any Collection objects	Only for List objects
2.	Is it legacy?	Yes	No	No
3.	Movement	Forward direction	Forward direction	Bidirectional
4.	Allowed operations	Only read	Read, Remove	Read, Remove, Add, Replace
5.	How can we get?	By using element () method of Vector class.	By using iterator() method of Collection interface.	By using listIterator() method of List interface.

List Vs Set

No.	List	Set
1.	Duplicates are allowed.	Duplicates are not allowed.
2.	Insertion order preserved.	Insertion order not preserved.

HashSet Vs LinkedHashSet

No.	HashSet	LinkedHashSet
1.	The underlying data structure is Hashtable.	The underlying data structure is a combination of Hashtable & LinkedList.
2.	Insertion order not preserved.	Insertion order preserved.
3.	Introduced in 1.2 version.	Introduced in 1.4 version.

TreeSet Vs HashSet Vs LinkedHashSet

No.	Property	TreeSet	HashSet	LinkedHashSet
1.	Underlying Data Structure	Balanced Tree	Hashtable	Hashtable + LinkedList
2.	Duplicate objects	Not allowed	Not allowed	Not allowed
3.	Insertion order	Not Preserved	Not Preserved	Preserved
4.	Sorting order	Applicable	Not applicable	Not applicable
5.	Heterogeneous object	Not allowed (default)	Allowed	Allowed
6.	Null Acceptance	Not allowed	Not allowed	Not allowed

Comparable & Comparator Interface

- If we are depending on default natural sorting order, compulsory the objects should be
  1. Homogenous
  2. Comparable
     otherwise, we will get Runtime exception saying ClassCastException.

- An Object is said to be **Comparable** if corresponding class implements **Comparable** interface.
  e.g., String class & all wrapper classes already implements Comparable interface but StringBuffer class doesn't implement Comparable interface; hence we will get ClassCastException.

- **Comparable** interface is present in **java.lang package** & it contains only 1 method

  i.e.

  ```
  int compareTo (Object obj)
  ```

  ```
  obj1.compareTo (obj2)
  ```

- **Method explanation**

  obj1 – The object which is to be inserted.
  obj2 – The object which is already inserted.
  - This method returns negative no. if obj1 has to come before obj2.
  - This method returns positive no. if obj1 has to come after obj2.
  - This method returns zero if obj1 equals to obj2 i.e., duplicate.

- If we're depending on default natural sorting order, then while adding objects into the TreeSet, JVM will call compareTo () method.
- If we are not satisfied with default natural sorting or default natural sorting is not available then we should go for customized sorting by using **Comparator** interface.

- Comparator interface is present in java.util package & it defines 2 methods

  a) int compare (Object obj1, Object obj2)
  - This method returns negative no. if obj1 has to come before obj2.
  - This method returns positive no. if obj1 has to come after obj2.
  - This method returns zero if obj1 equals to obj2 i.e., duplicate.

  b) boolean equals (Object obj)

- Whenever we're implementing **Comparator** interface, compulsory we should provide implementation only for compare () method & we're not required to provide implementation for equals () method because it is already available to our class from Object class through Inheritance.

Comparable Vs Comparator

No.	Comparable	Comparator
1.	It is meant for default natural sorting order.	It is meant for customized sorting order.
2.	Present in java.lang package.	Present in java.util package.
3.	It defines only one method i.e., **compareTo ()**	It defines 2 methods i.e., **compare ()** & **equals ()**
4.	String & all wrapper classes implements Comparable interface.	The only implemented classes of Comparator interface are Collator, RuleBasedCollator

HashMap Vs Hashtable

No.	HashMap	Hashtable
1.	Every method present in HashMap is non – synchronized.	Every method present in Hashtable is synchronized
2.	At a time, multiple threads are allowed to operate on HashMap object & hence Not thread safe.	At a time, only one thread is allowed to operate on Hashtable & hence it is thread – safe.
3.	Relatively performance is high because threads are not required to wait to operate on HashMap object.	Relatively performance is low because threads are required to wait to operate on Hashtable object.
4.	Null is allowed for both key & value	Null is not allowed for keys & values otherwise we will get NullPointerException.
5.	Introduced in 1.2 version & it's not legacy	Introduced in 1.0 version & it's legacy

HashMap Vs LinkedHashMap

No.	HashMap	LinkedHashMap
1.	The underlying data structure is Hashtable.	The underlying data structure is a combination of Linked list & Hashtable (Hybrid DS)
2.	Insertion order is not preserved & it's based on hashcode of keys	Insertion order is preserved.
3.	Introduced in 1.2 version	Introduced in 1.4 version

Note: LinkedHashSet & LinkedHashMap are commonly used for developing Cache based applications

Utility classes: Collections & Arrays

1. Collections class [java.util]

- Collections class is one of the utility classes in Java Collections Framework.
- Collections class defines several utility methods for Collection objects like Sorting, Searching, Reversing etc.
- This class contains only static methods that operate on or return collections.
- Collections class basically contains 3 fields that can be used to return immutable entities:
    1. static List EMPTY_LIST
    2. static Set EMPTY_SET
    3. static Map EMPTY_MAP
- Collections class methods:
    1. Static <T> Queue<T> asLifoQueue (Deque<T> deque): returns a view of a Deque as a Last-In-First-Out (LIFO) Queue.
    2. Static <T> int binarySerach(List<T> list, T key): searches the specified list for the specified object using the binary search algorithm.
    3. Static <T> void copy (List<? Extends T> dest, List<? Extends T> src): copies all of the elements from one list to another.
    4. Other methods like max, min, reverse, reverseOrder, rotate, sort, swap etc.

2. Arrays class [java.util]

- This class contains various methods for manipulating arrays (such as sorting & searching).
- This class also contains a static factory that allows arrays to be viewed as lists.
- Arrays specific methods like binarySearch, sort etc.