***What are Collections(JDK1.2) in Java?***

1. ***Collections are nothing but group of objects stored in well-defined manner.***
2. ***It basically provides an*** *architecture* ***to store and manipulate group of objects.***
3. *Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion*
4. *Java collection is a single unit of all Objects.*
5. *It is part of Java Collections Framework (JCF).*
6. *Collection framework contains Group of several classes and interfaces which can be used to represent group of individual objects as a single entity.*

***🡪Earlier, Arrays are used to represent this group of objects. But, arrays are not re-sizable. size of the arrays is fixed.***

***🡪Size of the arrays cannot be changed once they are defined. This causes lots of problem while handling group of objects.***

***🡪 To overcome this drawback of arrays, Collection framework or simply collections are introduced in java from JDK 1.2***



***Iterable Vs Collection***



*Collection is subiterface of Iterable Interface.*

* ***Iterable (Interface)***
* ***Definition****: Iterable<T> is the root interface in Java for all collection-like data structures.*
* ***Purpose****: It represents a group of objects that can be* ***iterated over*** *using a for-each loop.*
* ***Methods****: It has only* ***one method***

*This method returns an Iterator object, which helps in traversing the elements.*

***Since Iterable is a root interface, all Collection classes implement it****.*

* ***Collection (Interface)***
* ***Definition****: Collection<E> is a subinterface of Iterable<E> that* ***extends*** *its functionality.*
* ***Purpose****: It defines a* ***more advanced set of operations*** *like adding, removing, and checking elements.*
* ***Methods****: It provides multiple methods like add(), remove(), isEmpty(), size(), contains(), etc*

***🡪 Subinterfaces:***

* + List<E> (Ordered, duplicates allowed)
  + Set<E> (No duplicates)
  + Queue<E> (FIFO processing

***Relationship Between Iterable and Collection***

* ***All Collection classes are Iterable*** *because Collection<E> extends Iterable<E>.*
* ***Not all Iterable implementations are Collection****. For example, Iterable can be implemented in custom data structures that do not support add() or remove()*

***Why do we use Collections?***

*✅* ***Reduces code complexity*** *– Provides built-in methods for handling data.  
✅* ***Performance optimization*** *– Various implementations optimized for different use cases.  
✅* ***Scalability*** *– Can handle large data efficiently.  
✅* ***Thread safety (for some classes like Vector, Hashtable)*** *– Useful for concurrent programming.*

***Real-Time Applications of Java Collections***

***1) Banking & Finance Applications***

* ***Use Case:*** *Transaction Processing System*
* ***Example:*** *Banks store millions of transactions in a List or Queue, process them sequentially, and remove processed transactions.*
* ***Collection Used:***
  + *List<Transaction> (to store transactions in order)*
  + *Queue (to process transactions in FIFO order)*

***2) E-commerce Applications***

* ***Use Case:*** *Shopping Cart System*
* ***Example:*** *List is used to store cart items, and Map is used to maintain inventory.*
* ***Collection Used:***
  + *List<Product> (to store cart items)*
  + *HashMap<Product, Integer> (to store product stock availability)*

***3) Social Media Applications***

* ***Use Case:*** *Managing Friend Lists & User Connections*
* ***Example:*** *Set is used to store unique followers or friends.*
* ***Collection Used:***
  + *HashSet<User> (to ensure no duplicate followers)*
  + *Map<User, List<Post>> (to store user posts)*

*4)* ***Ride-Sharing Applications (Uber, Lyft)***

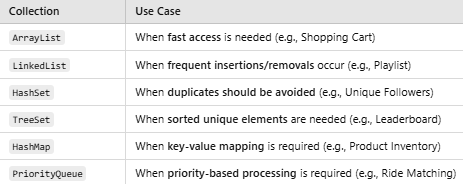
* ***Use Case:*** *Matching Riders with Drivers*
* ***Example:*** *PriorityQueue is used to assign nearest drivers to riders.*
* ***Collection Used:***
  + *PriorityQueue<Driver> (to find nearest available driver)*
  + *HashMap<Rider, Driver> (to store active rides)*

*5)* ***Healthcare Management System***

* ***Use Case:*** *Managing Patient Records*
* ***Example:*** *TreeMap is used to store patient records sorted by ID.*
* ***Collection Used:***
  + *TreeMap<Integer, Patient> (to keep records sorted by ID)*

***Why to use Collections in Real time applications?***

*✅* ***Efficient Data Handling*** *– Handles large datasets efficiently.  
✅* ***Pre-built Functionalities*** *– No need to write custom data structures.  
✅* ***Scalability*** *– Works well with growing data.  
✅* ***Better Performance*** *– Optimized algorithms for searching, sorting, and iteration.  
✅* ***Flexible Data Structures*** *– Supports Lists, Sets, Maps, and Queues.*



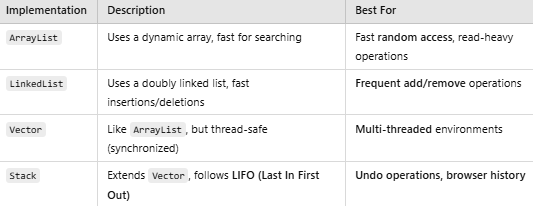
***Collection Framework 🡪 1) List Interface***

*List interface represents an ordered and sequential collection of Objects.*

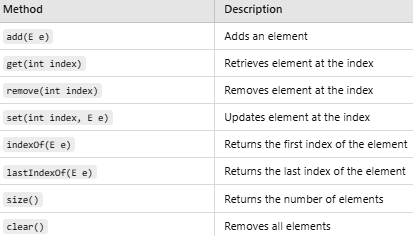
*Features of List:-*

*✅* ***Ordered Collection*** *– Maintains insertion order.   
✅* ***Allows Duplicates*** *– Elements can be repeated.  
✅* ***Indexed Access*** *– Elements can be accessed via an index.  
✅* ***Allows Null Values*** *– null elements can be stored.  
✅* ***Resizable*** *– Unlike arrays, lists grow dynamically.*

***Classes which implement Lists are:-***



***Methods Of List Interface are:-***



***Types of Data That Can Be Stored in ArrayList and List in Java***

*In Java,* ***both ArrayList and List*** *(which is an interface) can store* ***any type of data****, including:  
✅* ***Primitive Wrapper Classes*** *(e.g., Integer, Double, Boolean)  
✅* ***Objects of Custom Classes*** *✅* ***Strings*** *(String)  
✅* ***Null Values*** *✅* ***Other Collections*** *(e.g., List, Set, Map)*

***List(Interface)🡪 1)ArrayList (Class)***

*ArrayList is a resizable array implementation of the List interface in Java's Java Collections Framework (JCF). It stores elements in a dynamic array and allows random access using an index*

***Key Features of ArrayList:-***

*✅* ***Dynamic resizing*** *– It automatically increases size when needed.  
✅* ***Ordered collection*** *– Maintains insertion order.  
✅* ***Allows duplicates*** *– Same elements can be added multiple times.  
✅* ***Fast random access*** *– Uses an* ***array****, so accessing an element via index is fast (O(1)).  
✅* ***Allows null values*** *– null elements can be stored.  
✅* ***Not thread-safe*** *– Not synchronized by default.*

*Initial Size (default size ) of ArrayList is 🡪 10.*

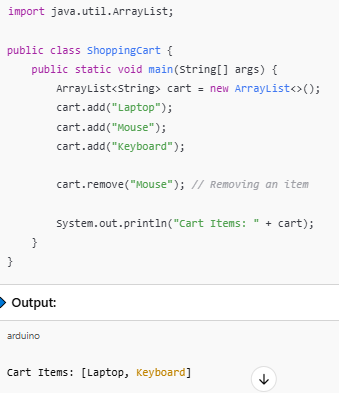
**The size of the ArrayList grows based on load factor and current capacity.**

* The Load Factor is a measure to decide when to increase its capacity. The default value of load factor of an ArrayList is 0.75f
* ArrayList in Java expands its capacity after each threshold which is calculated as the product of *current capacity* and *load factor* of the ArrayList instance.

*When the user inserts element upto 75% of the size then load factor gets invoked and automatically size of arraylist is doubled.* ***newCapacity = loadFactor\*oldCapacity***

***🡪*** *It is fast and widely used when we want to fetch or access any data , but slow in terms of data manipulation like inserting, deleting data in between.*

* *It creates a new copy everytime when we do any manipulation (insert, update, delete) due to which overhead increases and performance becomes slow.*
* *It uses* ***dynamic array*** *Data Structure.*



***List(Interface)🡪 2)LinkedList (Class)***

*🡪 Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the Abstract List class and implements List.*

*🡪* *Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the Abstract List class and implements List*

***Key Features of LinkedList***

*✅* ***Doubly Linked List*** *– Each element (node) stores references to both* ***previous*** *and* ***next*** *elements.  
✅* ***Efficient Insertions & Deletions (O(1))*** *– Adding/removing elements is faster than ArrayList.  
✅* ***Slow Random Access (O(n))*** *– Accessing an element requires traversal from the beginning.  
✅* ***Allows Duplicates & Null Values*** *– Elements can be repeated, and null values are allowed.  
✅* ***Implements List and Deque Interfaces*** *– Supports* ***FIFO (Queue) & LIFO (Stack) operations.***



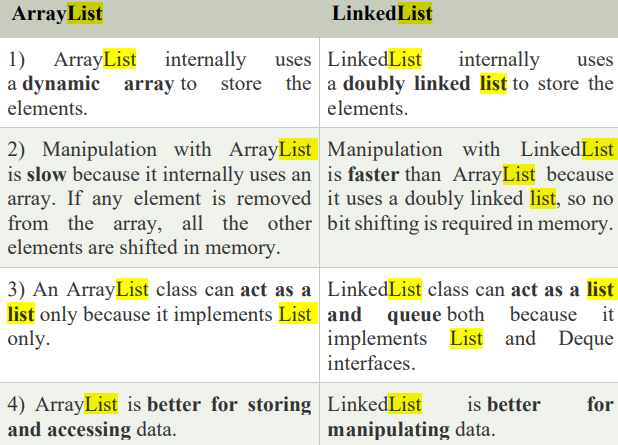
*🡪 Linked List does not have any default capacity because , as it follows node structure, so each new element gets added on new node dynamically and size gets increased automatically upon insertion of elements.*

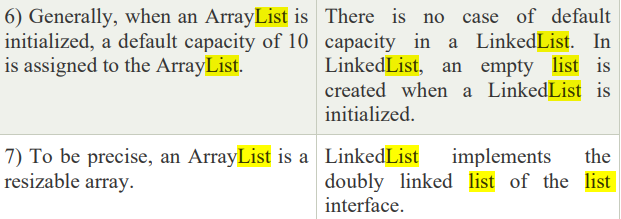
*Each* ***node*** *in a LinkedList consists of:*

* ***Data (element)***
* ***Reference to the next node***
* ***Reference to the previous node***

***Memory Overhead of LinkedList***

*Each node requires extra memory for the* ***next*** *and* ***previous*** *references. This makes LinkedList more* ***memory-intensive*** *than ArrayList.*





***Generic Collections***

***Definition***

*A* ***generic collection*** *is a collection that works with* ***typed elements****, meaning you specify the* ***type of objects*** *the collection will store when creating it. This provides* ***type safety****, ensuring that only objects of the specified type can be added to the collection.*

***Key Features***

* ***Type-safe****: The compiler ensures that only elements of a specified type can be added.*
* ***No need for casting****: When you retrieve an element from the collection, you don't need to cast it.*

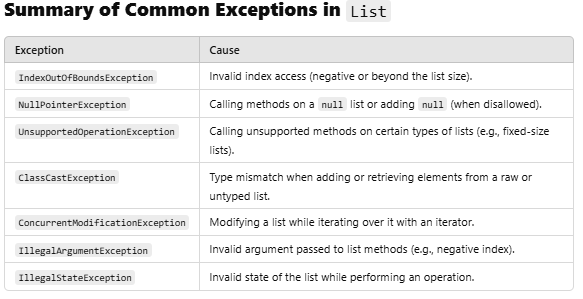
***Non-Generic Collections***

***Definition***

*A* ***non-generic collection*** *is a collection that can store* ***any type of objects*** *(it doesn't enforce a specific type). The type is not specified when creating the collection. As a result, elements of* ***different types*** *can be added.*

***Key Features***

* ***No Type Safety****: You can add any type of object to the collection, leading to potential runtime errors.*
* ***Old Java Collections (pre-Java 5)****: These collections were used before generics were introduced in Java 5.*



***Fail-Fast Collections***

*A collection is said to be* ***fail-fast*** *if it immediately throws a* ***ConcurrentModificationException*** *when it detects that the collection has been modified while it is being iterated.*

***Characteristics of Fail-Fast Collections:***

* ***Throws ConcurrentModificationException****: When an iterator detects a structural modification during iteration (modification of the list's elements), it throws a ConcurrentModificationException.*
* ***Common Fail-Fast Collections****: Collections like ArrayList, HashMap, LinkedList, and HashSet are fail-fast.*

***How Fail-Fast Works:***

* *Fail-fast behavior is triggered when the* ***modCount*** *field (modification count) of the collection is different from the* ***expected modCount*** *when an iterator is used.*
* *If the collection is modified (i.e., elements are added, removed, or changed in structure) while an iterator is traversing it, the next operation on the iterator will throw a ConcurrentModificationException.*
* *To know whether the collection is structurally modified or not, fail-fast iterators use an internal flag called modCount which is updated each time a collection is modified.Fail-fast iterators checks the modCount flag whenever it gets the next value (i.e. using next() method), and if it finds that the modCount has been modified after this iterator has been created, it throws ConcurrentModificationException.*

***Note :****If you remove an element via Iterator remove() method, exception will not be thrown. However, in case of removing via a particular collection remove() method, ConcurrentModificationException will be thrown. Below code snippet will demonstrate this:*

***Fail-Safe Collections***

*Fail-Safe this term used to demonstrate the difference between Fail Fast and Non-Fail Fast Iterator. These iterators make a copy of the internal collection (object array) and iterates over the copied collection. Any structural modification done to the iterator affects the copied collection, not original collection. So, original collection remains structurally unchanged.*

* *Fail-safe iterators allow modifications of a collection while iterating over it.*
* *These iterators don’t throw any Exception if a collection is modified while iterating over it.*
* *They use copy of original collection to traverse over the elements of the collection.*
* *These iterators require extra memory for cloning of collection. Ex : ConcurrentHashMap, CopyOnWriteArrayList*
* ***Difference between Fail Fast Iterator and Fail Safe Iterator*** *The major difference is fail-safe iterator doesn’t throw any Exception, contrary to fail-fast Iterator.This is because they work on a clone of Collection instead of the original collection and that’s why they are called as the fail-safe iterator.*

|  |  |  |
| --- | --- | --- |
| ***Base of Comparison*** | ***Fail Fast Iterator*** | ***Fail Safe Iterator*** |
| ***Exception*** | *It throws a ConcurrentModificationException in modifying the object during the iteration process.* | *It does not throw Exception.* |
| ***Clone Object*** | *No clone object is created during the iteration process.* | *A copy or clone object is created during the iteration process.* |
| ***Memory utilization*** | *It requires low memory during the process.* | *It requires more memory during the process.* |
| ***Modification*** | *It does not allow modification during iteration.* | *It allows modification during the iteration process.* |
| ***Performance*** | *It is fast.* | *It is slightly slower than Fail Fast.* |
| ***Examples*** | *HashMap, ArrayList, Vector, HashSet, etc* | *CopyOnWriteArrayList, ConcurrentHashMap, etc.* |

***Collection Framework 🡪 2) Set Interface***

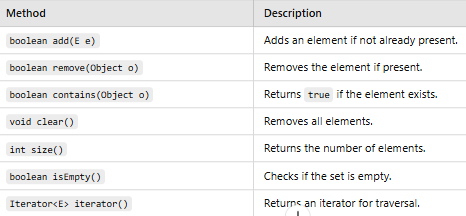
*The Set Interface is present in [java.util](https://www.geeksforgeeks.org/java-util-package-java/) package and extends the*[*Collection interface*](https://www.geeksforgeeks.org/collections-in-java-2/)*. It is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the mathematical set. This interface adds a feature that restricts the insertion of the duplicate elements.*

* ***No Specific Order:*** *Does not maintain any specific order of elements (Exceptions: LinkedHashSet and TreeSet).*
* ***Allows One Null Element:*** *Most Set implementations allow a single null element.*
* ***Implementation Classes:***[*HashSet*](https://www.geeksforgeeks.org/hashset-in-java/)*, [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/" \t "_blank) and [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/" \t "_blank).*
* ***Thread-Safe Alternatives****: For thread-safe operations, use [ConcurrentSkipListSet](https://www.geeksforgeeks.org/concurrentskiplistset-in-java-with-examples/" \t "_blank)or wrap a set using Collections.synchronizedSet().*

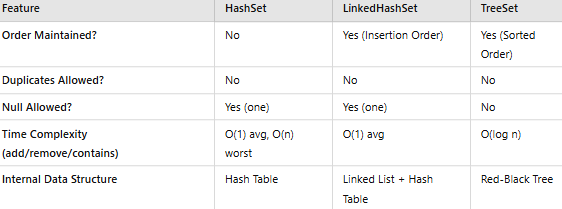
***Characteristics of Set Interface***

* *No Duplicate Elements – A Set does not allow duplicate elements. If a duplicate is added, the existing one is retained.*
* *Unordered Collection – Does not maintain insertion order (except for LinkedHashSet).*
* *Allows Null Values – Only one null value is permitted.*
* *Implements Collection Interface – Inherits methods from the Collection interface.*

***Important methods of Set Interface***



***Difference between Set Implementations***



***When to Use Which Set Implementation?***

* ***Use HashSet*** *when fast operations are needed and ordering doesn’t matter.*
* ***Use LinkedHashSet*** *when you want insertion order to be preserved.*
* ***Use TreeSet*** *when sorting is needed but performance is not a major concern.*

***(We use same iteration process as Lists for Set and its implications as well)***

***HashSet:--***

*Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface*

***The important points about Java HashSet class are:***

*o HashSet stores the elements by using a mechanism called hashing.*

*o HashSet contains unique elements only.*

*o HashSet allows null value.*

*o HashSet class is non synchronized.*

*o HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode. o HashSet is the best approach for search operations.*

*o The initial default capacity of HashSet is 16, and the load factor is 0.75.*

***LinkedHashSet:-***

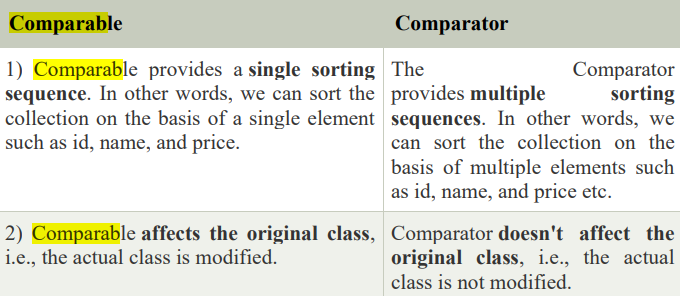
*Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface. The* ***Important points about the Java LinkedHashSet class are:***

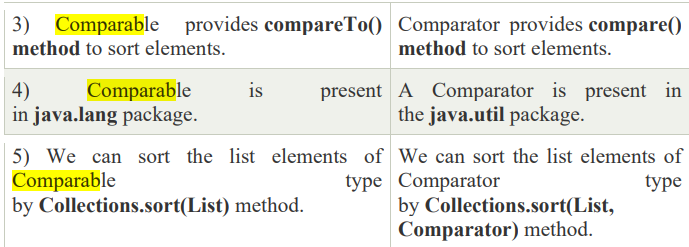
*o Java LinkedHashSet class contains unique elements only like HashSet.*

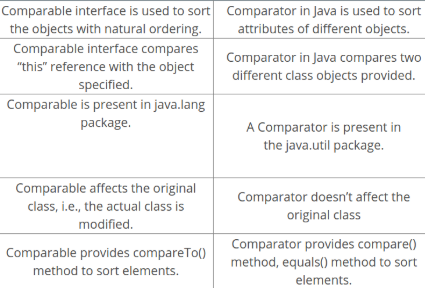
*o Java LinkedHashSet class provides all optional set operations and permits null elements.*

*o Java LinkedHashSet class is non-synchronized.*

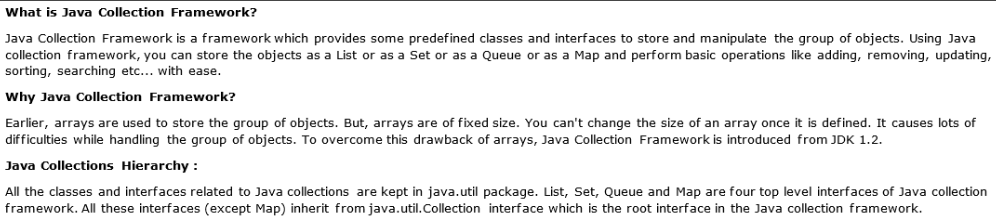
*o Java LinkedHashSet class maintains insertion order.*

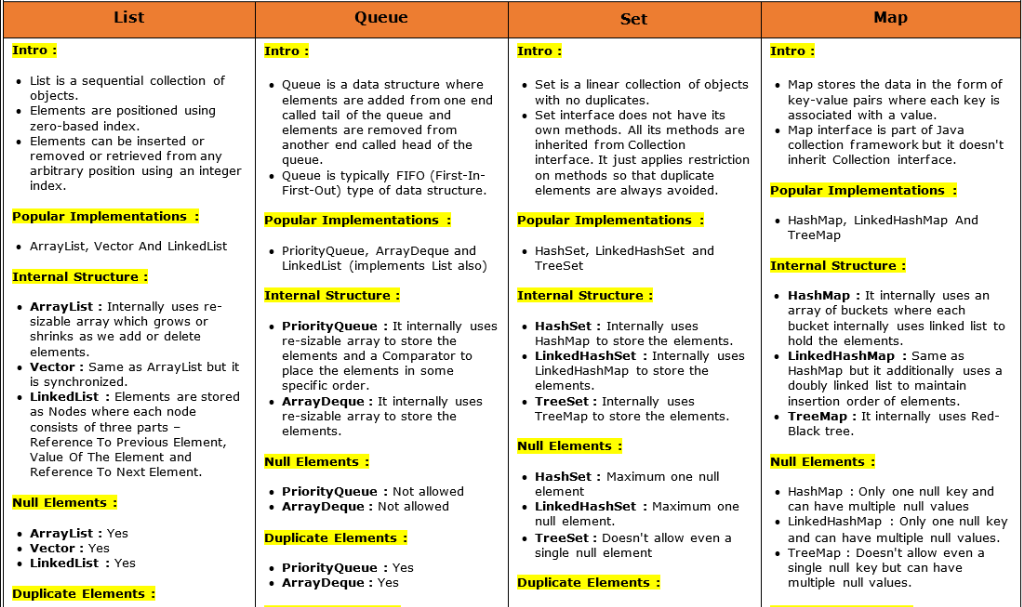


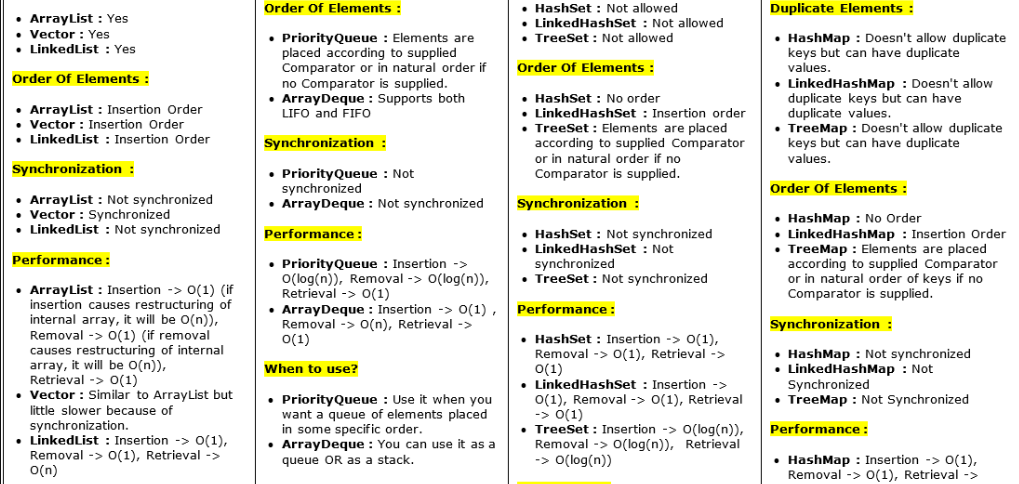




* [*https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/*](https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/) *(For Code)*









***Arrays Vs Collections***

1. *Collections are dynamic in Size(Growable)*
2. *Collections can store Homogenous/Heterogenous type of data elements*
3. *Every Collection class is implemented based on Standard data structure. As a programmer we don’t need to write code for operations on data , we need to use inbuilt methods.*
4. *For every requirement , readymade method support is available.*
5. *Collections cannot hold Primitive Data type they can only hold Objects and if we want to store Primitive data’s then need to use Wrapper classes ,Arrays can hold both primitive and Objects*
6. *With respect to memory arrays are not recommend to use and so we should use Collection when memory is important.*
7. *With respect to performance arrays are recommended to use so we should use Collection when performance is important.*

*--------------------------------------------------------------------------------------------------------------------------*

***Difference between Collection & Collections?***

***Collection (Interface)->****If we want to represent group of individual objects as single entity then use Collection*

***Collections (class)*** *🡪It is utility class . Present in* ***java.util*** *package .It defines several utility methods for Collection objects->* ***.sort() method , .searching(), etc***

***List (1.2):-***

*1)It is a child interface of Collection.*

*2)Duplicates and* ***multiple null******are allowed****, Insertion order*  ***followed***

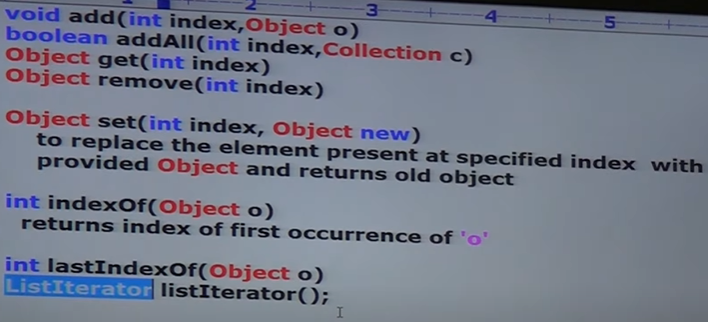
*3)* ***ArrayList(1.2),******LinkedList(1.2)*** *are classes of List interface*

*4)* ***Vector****(1.0),* ***Stack****(1.0) are Legacy (old generation) classes of List interface*

*5) Insertion order preserved on base of index and differentiate duplicate objects with index.*

*6)Hence, index plays important role in List.*

*7)List interface defines following specific methods*



***ArrayList:-***

1. *Underlying data structure is* ***Resizeable / growable array***
2. *Duplicates are* ***allowed/ insertion order is preserved****.*
3. *ArrayList can hold* ***heterogenous*** *objects*
4. *ArrayList can have “* ***null*** *“ objects.*
5. *ArrayList is* ***best choice*** *for* ***data retrieving , fetching operations****. Because it implements RandomAccess Interface.*
6. *It is* ***worst choice*** *for* ***insertion / deletion***  *operation in between.*

* ***Except TreeSet and TreeMap Heterogenous objects are allowed everywhere.***
  + ***Constructors:-*** 
    1. ***ArrayList list=new ArrayList();***

*Creates an empty arraylist object with default initial capacity =10*

*Once ArrayList reaches its max capacity then a new ArrayList Object gets created with newCapacity=(currentCapacity\*3/2) + 1. (before java 8)*

*From Java8-> loadfactor =0.75 i.e 75 %*

*If 8th element is added then automatically (currentCapacity\*2)🡪10\*2=20*

* + 1. ***ArrayList list=new ArrayList(int initialCapacity).***

*Creates an empty ArrayList object with specified initial Capacity.*

* + 1. ***ArrayList list=new ArrayList(Collection c)***

*Creates an equivalent ArrayList Object for the given Collection.*

* ***Usually we can use collections to transfer and hold Objects from one location to another Location(Container). To provide support for this requirement , Every Collection class by default implements Serializable Interface & Cloneable Interface***
* ***ArrayList*** *&* ***Vector*** *classes implements* ***RandomAccess*** *Interface.*

***🡪 So that any random elements can be accessed with same speed.***

***RandomAccess Interface:-*** *java.util Package, It does not contain method as it is Marker Interface. Where the required ability will be automatically provided by JVM.*

***Difference between ArrayList and Vector***

|  |  |
| --- | --- |
| ***ArrayList*** | ***Vector*** |
| *Every method is* ***non****-* ***synchronized*** | *Most of method are synchronized.* |
| *At a time multiple thread are allowed to operate on object. So It is not thread-safe* | *At a time only one thread is allowed to operate on object. So It is thread-safe* |
| *Performance is High than Vector because multiple threads work simultaneously* | *Performance is Low because threads are required to wait for next operation* |

***By default ArrayList is non synchronized:-***

***🡪But we can get synchronized version of ArrayList Object by using synchronized method of Collections class.***

*ArrayList list= new ArrayList();*

*List l1=Collections.synchronized(list);*

***public static List synchronized(List l); 🡪****Method of Collections class*

*Similarly we can get synchronized version of Set & Map Obejcts*

***public static Set synchronizedSet(Set s);***

***public static Map synchronizedMap(Map m);***

***LinkedList:-***

1. *Underlying data structure is* ***DoublyLinked*** *List for implementation of LinkedList.*
2. *Insertion* ***order is preserved, duplicate objects are allowed****.*
3. *Heterogenous Objects* ***are allowed****.*
4. *“ null “ insertion* ***is possible****.*
5. ***LinkedList*** *implements* ***Serializable****,* ***Cloneable*** *but not* ***Random Access*** *Interface*
6. *Best choice if frequent operation* ***is Insertion and deletion.***
7. *Worst Choice if frequent operation is* ***retrieval, fetching operation.***

***Constructors:-***

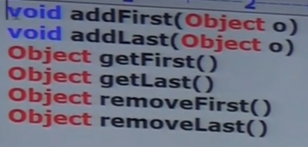
* + 1. ***LinkedList list=new LinkedList();***

*Creates and empty linkedlist Object.*

* + 1. ***LinkedList list=new LinkedList(Collection c)***

*Creates an equivalent Linked List object for Collection*

*LinkedList class have specific methods:- Usually we can LinkedList to develop Stack(LIFO) and Queue(FIFO). To provide support for this requirement LinkedList class defines following specific methods,*



***Vectors:-***

1. *Underlying data structure is* ***Resizeable / growable array***
2. *Duplicates are* ***allowed/ insertion order is preserved****.*
3. *Vectors can hold* ***heterogenous*** *objects*
4. *Vectors can have “* ***null*** *“ objects.*
5. *Vectors implements* ***Serializable****,* ***Cloneable*** *but not* ***Random Access*** *Interface*

***Constructors:-***

* 1. ***Vector v=new Vector(); 🡪*** *Default Initial Capacity****-10***

*New capacity=(currentCapacity\*2)*

* 1. ***Vector v=new Vector(int initialCapacity)***

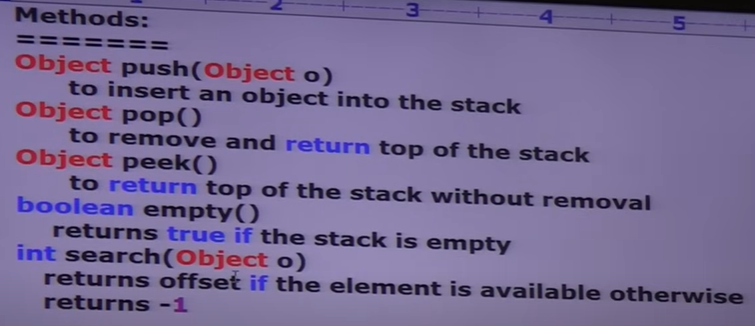
*Creates an empty Vector object with specified initial Capacity*

* 1. ***Vector v=new Vector(int initialCapacity, int incrementalCapacity)***
  2. ***Vector v=new Vector(Collection c)***

*Creates an equivalent Vector object for given Collection. It is meant for interconversion between Collection Objects.*

***Stack:-***

1. *Follows FIFO (First In First Out ) Order*
2. *It is child class of Vector class.*
3. *Methods of Stacks:-*



***What are Cursors?***

*If we want to get Objects one-by-one and perform operations on them then we should go for Cursors.*

*There 3 type of Cursors available in JAVA:-*

* 1. ***Enumerations(v1.0) 🡪Only for Legacy Classes***
  2. ***Iterator(v1.2)***
  3. ***ListIterator(v1.2)***

***Enumerations:-***

*We can use enumeration to get objects one-by one from Legacy Collection Object.(Vectors)*

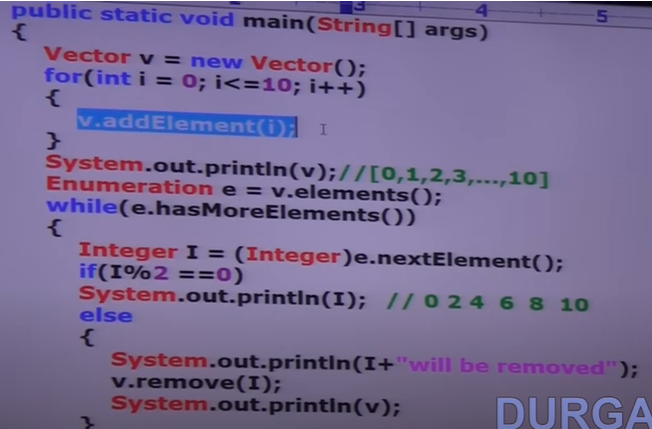
*We can create Enumeration object by using elements() method of Vector class*

***Enumeration e=v.elements(); 🡪 v = Vector object***

***It has only 2 methods:-***

***public boolean hasMoreElements();***

***public Object nextElement();***



***Limitations of Enumerations:-***

* + - *Can only be used for Legacy Classes.*
    - *It Is not a universal cursor.*
    - *By using enumeration we can only get Read access, we cannot perform remove operation. To overcome above limitations we should go for Iterator.*

***Iterator (Interface):-***

1. *Iterator is universal cursor.*
2. *Read and remove operations can be performed using Iterator.*
3. *Can be used on any Collection object.*
4. *We can create Iterator object by* ***using .iterator() method of Collection interface***

*public Iterator iterator();*

***Iterator itr=collection.iterator(); 🡪Stores all collection object in itr***

***It has only 3 methods:-***

***public boolean hasNext();***

***public Object next();***

***public void remove(); 🡪removes elements from itr collection***

***Limitations of Iterator:-***

* *By using enumeration and iterator we can always move only towards forward direction and we cant move towards backward direction.*
* *These are unidirectional cursors and not bidirectional cursor.*
* *By using Iterator we can perform only Read/Remove operations.*
* *We cannot perform add/ replace operations*
* *To overcome above limitations we should go for Iterator.*

***ListIterator (Interface):-***

1. *We can move either forward/backward direction, hence it is bidirectional Cursor.*
2. *By using* ***ListIterator****, we can perform* ***replacement*** *and* ***addition*** *of new objects, in addition to* ***Read****/****Remove*** *operations.*
3. *It is an (child) interface which extends (parent)****Iterator*** *interface.*

***public ListIterator listIterator();***

***ListIterator itr=collection.listIterator();***

1. *It is the most powerful Cursor ,* ***but its limitation are :- they are applicable only for List Objects.***

***Forward Traversal Methods***

1. *public boolean hasNext() – Checks if there is a next element.*
2. *public Object next() – Returns the next element.*
3. *public int nextIndex() – Returns the index of the next element.*

***Backward Traversal Methods***

1. *public boolean hasPrevious() – Checks if there is a previous element.*
2. *public Object previous() – Returns the previous element.*
3. *public int previousIndex() – Returns the index of the previous element.*

***Modification Methods***

1. *public void remove() – Removes the last returned element.*
2. *public void add(Object o) – Adds an element at the current position.*
3. *public void set(Object o) – Replaces the last returned element.*

***Set (1.2):-***

*1)It is child interface of Collection.*

*2)Duplicates are* ***not allowed****, No Insertion order is* ***not followed***

*3)****HashSet(1.2),******LinkedHashSet(1.4)*** *are classes of Set Interface*

*4)****SortedSet(1.2)*** *is child interface of Set 🡪 Duplicates are* ***not allowed,*** *Objects inserted with some* ***sorting order.***

***NavigableSet(1.6)*** *🡪 It is child interface of* ***SortedSet****.*

*It contains several methods for navigation purpose.*

***TreeSet🡪(1.2)*** *It is implementation class of* ***NavigableSet***

*5) Set interface doesn’t contain any new method and we need to use only Collection interface methods.*

***HashSet(v1.2):-***

1. *Underlying data structure for HashSet is HashTable.*
2. *Duplicates are* ***not allowed****, No Insertion order is* ***not followed***
3. *All objects will be inserted based on* ***Hashcode*** *of objects*
4. *Only one “ null “ obejct allowed, as duplicates are not allowed.*
5. *Heterogenous Objects* ***are allowed****.*
6. *Wherever we are having search operation, then we should use HashSet.*

***Note:-***

*In hashSet duplicates are not allowed if we are trying to insert duplicates , then we wont get any Compile, runtime errors, add () method simply returns false.*

***Constructors:-***

1. ***HashSet hSet=new HashSet();***

*Creates an empty HashSet object* ***default capacity->16 Load Factor-> 0.75***

1. ***HashSet hSet=new HashSet(int initialCapacity);***

*Creates an empty HashSet object with specified capacity*

1. ***HashSet hSet=new HashSet(int initialCapacity, float fillRatio);***

*Creates an empty HashSet object with specified capacity ,customized loadFactor(****fillRatio).***

1. ***HashSet hSet=new HashSet(Collection obj);***

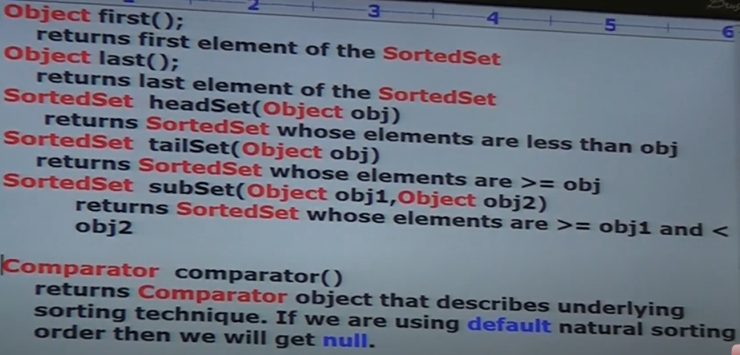
*It is meant for inter-conversion between Collection objects*

***LinkedHashSet(v1.4):-***

* 1. *Underlying dataStructure used is combination of* ***HashTable & LinkedList***
  2. *It is child class of HashSet and exactly same as HashSet incl constructors& methods.*
  3. *Insertion* ***Order is preserved.***
  4. *In general, we can use LinkedHashSet to* ***develop Cache based applications*** *where* ***duplicates are not allowed, insertion order is preserved****.*
  5. *All hash based collections have same constructors.*

***SortedSet():-***

1. *SortedSet is child interface of Set*
2. *If we want to represent group of individual objects according to some Sorting Order without duplicates then we should use SortedSet.*
3. *SortedSet has 6 different methods which are only applicable for Sorted Set.*



***TreeSet:-***

1. *Uses Balanced Tree as underlying data structure.*
2. *Duplicate objects* ***are not allowed****.*
3. *Insertion order* ***not preserved****.*
4. *Heterogenous objects* ***are not allowed,*** *otherwise we will get RE saying* ***ClassCastException***
5. *Only one “ null “ insertion allowed.*
6. *TreeSet implements Serializable, Cloneable but not RandomAccess.*
7. *All objects are inserted based on some Sorting Order maybe default natural, customized sorting order.*
8. *If we are depending on default natural sorting order then the objects should be homogenous and comparable, otherwise we will get ClassCast Exception.*
9. *An object is said comparable if corresponding class implements Comparable interface.*
10. *String, all wrapper classes already implement Comparable, but StringBuffer doesn’t.*

***Constructors:-***

1. ***TreeSet tSet=new TreeSet(); 🡪*** *Creates an empty TreeSet object where elements are inserted according to natural sorting order.*
2. ***TreeSet tSet=new TreeSet(Comparator); 🡪*** *Creates an empty TreeSet object where elements inseted according to customized sorting order.*
3. ***TreeSet tSet=new TreeSet(Set s);***
4. ***TreeSet tSet=new TreeSet(SortedSet sortSet);***
5. *Until v1.6 null is allowed as first element to empty TreeSet but v1.7 onwards, null is not allowed, even as first element. “ null “ such type of story not applicable for TreeSet from v1.7 onwards*
6. *JVM iternally calls compareTo () while we are inserting elements in TreeSet*

***Note:-*** *1)All the above Interfaces, classes meant for representing group of objects.*

*2) If we want to represent group of objects in* ***Key-Value*** *pair then use Map*

***Map (1.2):-***

*1)Map is* ***not*** *child Interface of Collection.*

*2)If we want to represent group of objects as* ***Key-Value pair****, then we should go for Map.*

*3)Both Key and Value are objects only, each key-value pair is called as entry , thus called as entry objects.*

*4****) Duplicate Keys are not allowed, but Values can be duplicated***

*5)****HashMap(1.2)*** *is child class of Map Interface*

*🡪* ***LinkedHashMap(1.4)*** *is child class of HashMap*

*6)****WeakHashMap(1.2), IdentityHashMap(1.4)*** *is child class of Map Interface*

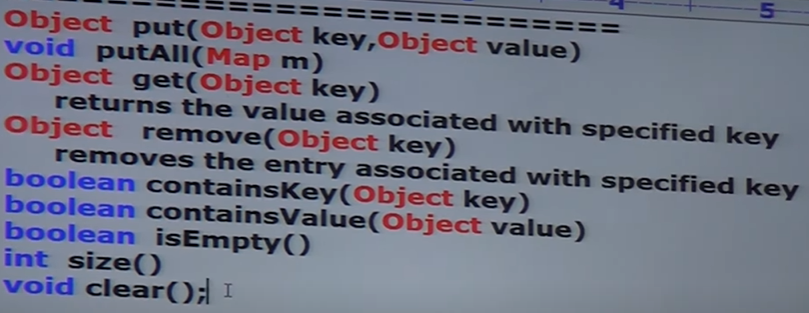
*7)****SortedMap(1.2)*** *is child interface of Map 🡪 Objects inserted with some* ***sorting order of keys.****Sorting will be based on keys and not on value.*

***NavigableMap(1.6)*** *🡪 It is child interface of* ***SortedMap****.*

*It contains several methods for navigation purpose.*

***TreeMap🡪(1.2)*** *It is implementation class of* ***NavigableMap***

***Map Interface Methods:-***



*Set* ***keyset****(); 🡪 To get set of keys*

*Collection* ***values****(); 🡪 To get collection of values*

*Set* ***entrySet****(); 🡪 To get whole key-value pair(entrySet)*

***Entry (Interface):- It is nested interface which is present in Map interface***

*1)Object* ***getKey****()*

*2)Object* ***getValue****()*

*3) Object* ***setValue****(Object newObj)*

*\* All these methods are part of entry interface*

***HashMap:-***

*1) Underlying data structure is* ***HashTable***

*2)Insertion* ***order is not preserved*** *and based on hashcode of keys*

*3)****Duplicate keys not allowed*** *but* ***values can be duplicated***

*4)Heterogenous* ***objects are allowed for key & value***

*5) “ null “ is allowed for key only once.*

*6)” null “ is allowed for values ‘n’ no of times*

*7) Implements* ***Serializable & Cloneable*** *but not RandomAccess*

*8)It is best choice if frequent operation is* ***search operation***

***9)JVM uses .equals() to identify duplicate keys***

***Constructors:-***

***1) HashMap hMap =new HashMap ();***

*Creates an empty* ***HashMap*** *object* ***default capacity->16 Load Factor-> 0.75***

***2) HashMap hMap =new HashMap (int initialCapacity);***

*Creates an empty* ***HashMap*** *object with specified capacity*

***3) HashMap hMap =new HashMap (int initialCapacity, float fillRatio);***

*Creates an empty* ***HashMap*** *object with specified capacity ,customized loadFactor(****fillRatio).***

***4) HashMap hMap=new HashMap (Map m);***

*It is meant for inter-conversion between Collection objects*

|  |  |
| --- | --- |
| ***HashMap*** | ***HashTable*** |
| *Methods are not synchronized* | *All methods are synchronized* |
| *At a time multiple threads can operate* | *At a time only one thread is allowed to operate* |
| *It is not thread-safe* | *It is thread-safe* |
| *Relatively performance is high , because threads are not required to wait.* | *Performance is low , because threads are required to wait to operate on HT object* |
| *“ null “ is allowed for key & values* | *“ null “ is not allowed for key & value otherwise we will get NullPointer Exception* |
| *Introduced in jdk 1.2* | *Introduced in jdk 1.0 . So it is Legacy* |

* ***How to get synchronized version of HashMap Object?***
* *Collections class contains synchronizedMap(Map m) method.*

*Map m1=Collections.synchronizedMap(m)*

|  |  |
| --- | --- |
| ***HashMap*** | ***LinkedHashMap*** |
| ***Underlying DS It is based on HashTable*** | ***Underlying DS Based on LinkedList & Hashtable*** |
| ***Insertion order is not preserved, based on Hashcode of keys*** | ***Insertion order is preserved*** |
| ***Introduced in 1.2v*** | ***Introduced in 1.4v*** |

***LinkedHashMap:-***

* 1. ***It is child class of hashmap.***
  2. ***Exactly same as Hashmap including methods and constructors, except above differences.***
  3. ***LinkedHashSet & LinkedHashMap are commonly used for developing Cache based applications.***

***IdentityHashMap©:-***

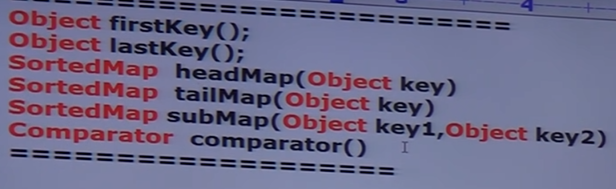
* + 1. *It is exactly same as HashMap including methods and constructors, except following difference.*
    2. *In case of normal* ***HashMap******JVM will use .equals() to identify duplicate keys****, which is meant for* ***content comparison****.*
    3. *But in case of* ***IdentityHashMap******JVM will use == to identify duplicate keys****, which is meant for* ***reference comparison(address comparison****)*

***WeakHashMap©:-***

1. *It is exactly same HashMap, except the following difference.*
2. *In case of* ***HashMap****, even though object doesn’t have any reference* ***it is not eligible for GC*** *if it is associated with* ***HashMap****, i.e* ***HashMap dominates Garbage collector****.*
3. *But in case of* ***WeakHashMap****, if object doesn’t contain any references* ***it is eligible for GC*** *eventhough object associated with* ***WeakHashMap****,* ***i.e Garbage collector dominates WeakHashMap.***

***SortedMap(I) :-***

1. *It is child interface of Map*
2. *If we want to represent group of key-value pairs according to some sorting order of keys, then we should go for SortedMap. Sorting is based on the key and not on the value.*
3. *Methods of SortedMap:-*



***TreeMap:-***

1. *Underlying data structure used is* ***RedBlackTree****.*
2. *Insertion order is not preserved and it is based on some sorting order of keys.*
3. *Duplicate keys are not allowed, but values can be duplicated.*
4. *If we are depending on* ***default natural sorting order****, then* ***keys should******be homogenous and comparable****, otherwise we will get* ***RuntimeException :ClassCastException****.*
5. *If we are defining* ***our own sorting by Comparator*** *then* ***keys need not be Homogenous and comparable****, we can take Heterogenous and non-comparable objects as well.*
6. *Whether we are depending on* ***natural sorting order or customized sorting order, there are no restrictions for values****, we can take heterogenous non-comparable objects also.*
7. *For* ***non-empty TreeMap*** *if we are trying to insert entry with null key, then we will get* ***RuntimeException:NullPointerException***
8. *For* ***empty TreeMap*** *as the* ***first entry with null key is allowed****, but after inserting that entry, if we are trying to insert any other entry then we will get* ***RuntimeException:NullPointerException.***
9. *The above null acceptance rule applicable until v1.6 only, from v1.7 null is not allowed for key.*
10. *Values can have multiple* ***null*** *values, irrespective of the verson.*

***Constructors:-***

1. ***TreeMap tMap=new TreeMap (); 🡪*** *Creates an empty TreeMap object where elements are inserted according to natural sorting order.*
2. ***TreeMap tMap =new TreeMap (Comparator); 🡪*** *Creates an empty TreeMap object where elements inserted according to customized sorting order.*
3. ***TreeMap tMap =new TreeMap (Map m);***
4. ***TreeMap tMap =new TreeMap (SortedMap sortMap);***

***Queue (1.5):-***

*1)It is a child interface of Collection.*

*2) If we want to store group of objects* ***Prior to processing*** *then we will use Queue.*

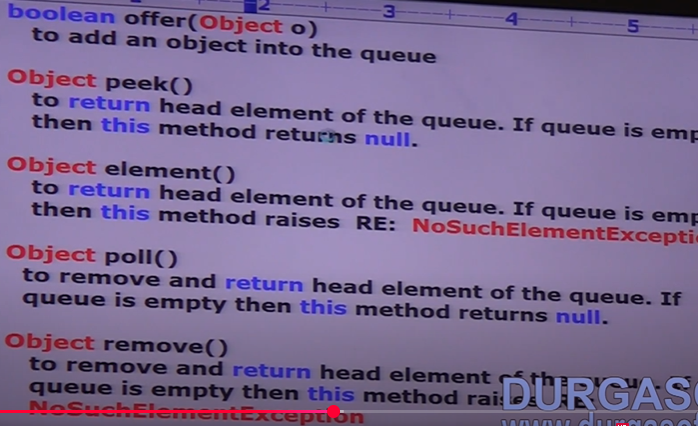
*3)Usually follows* ***FIFO order(First-In-First-Out)*** *but based on our requirement we can implement our own Priority Order.*

*4)****Eg:-****Before sending a mail, we need to store all Mail’ID in Data Structure, because in the order we stored mail, in that order only mail should be sent so there we can use QUEUE.*

*5)****Priority Queue, BlockingQueue*** *are child classes of Queue.*

*6)From v1.5 LinkedList also extends Queue interface, LinkedList based implementation of Queue follows FIFO.*

*7)Queue has 5 specific methods:-*



***PriorityQueue:-***

1. *If we want to represent group of individual objects* ***prior to processing*** *, according to some Priority, then we should go for Priority Queue.*
2. *Priority can be DNSO or CSO defined by Comparator.*
3. ***Insertion order is not preserved*** *and it based on some priority.*
4. ***Duplicate objects are not allowed****.*
5. *If we are depending on DNSO compulsory objects should be Homogenous and Comparabale, otherwise we will get* ***RuntimeException:ClassCastException****.*
6. *If we are defining our own sorting by comparator then objects need not be* ***Homogenous and comparable****.*
7. *“ null “ is not allowed even as the first element.*

***Constructors:-***

1. ***PriorityQueue queue=new PriorityQueue();***

***🡪****Creates an empty priority queue with default initialCapacity=11, all objects will be inserted according to DNSO.*

1. ***PriorityQueue queue=new PriorityQueue(int initialCapacity,Comparator c)***
2. ***PriorityQueue queue=new PriorityQueue(SortedSet s)***
3. ***PriorityQueue queue=new PriorityQueue(Collection c)***

***SortingOrder:-***

*1)****Comparable****(I)🡪 Used for default natural sorting order.*

*2)****Comparator****(I) 🡪Used for customized sorting order.*

***Cursors:-***

*Cursors are data Fetching techniques to get elements of collection one-by-one.*

*a) Enumerations(1.0 -Legacy Interface) (Interface)*

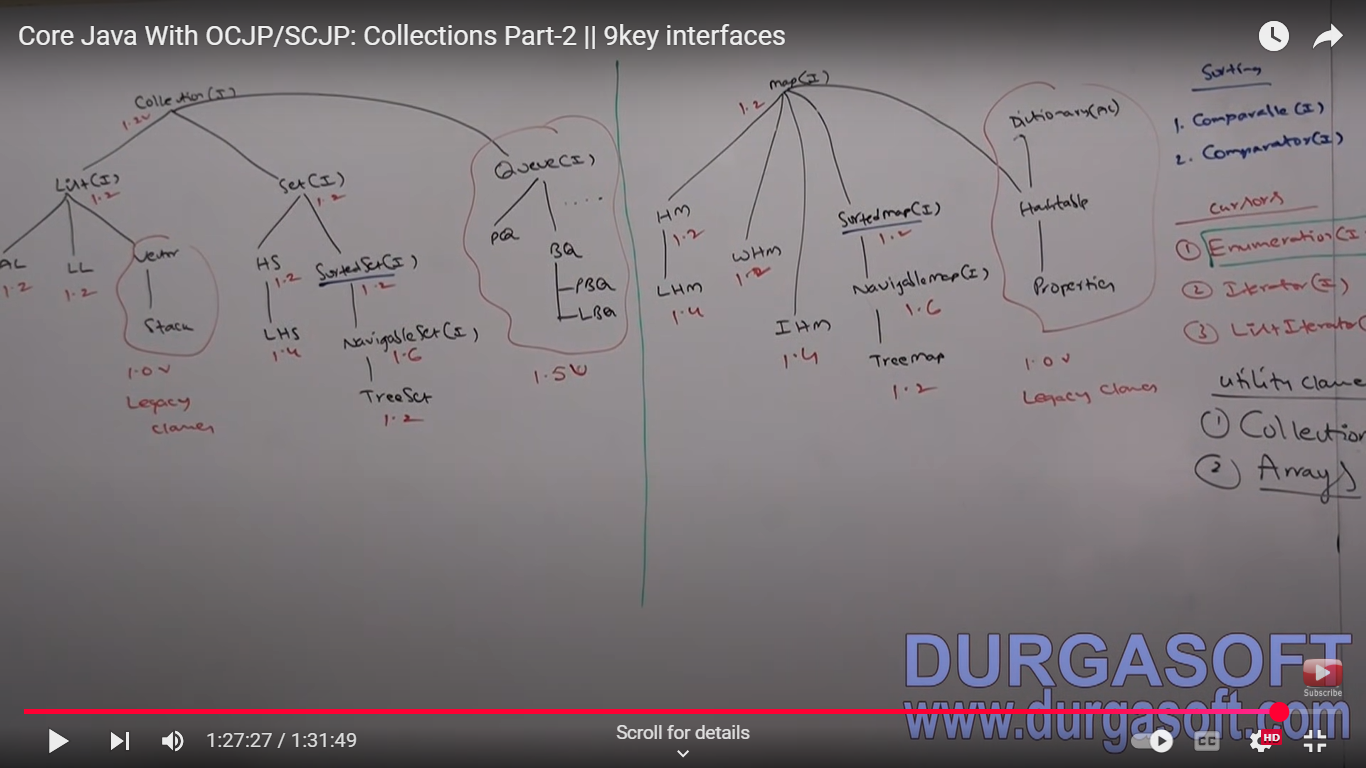
*b) Iterator (Interface)*

*c) listIterator (Interface)*

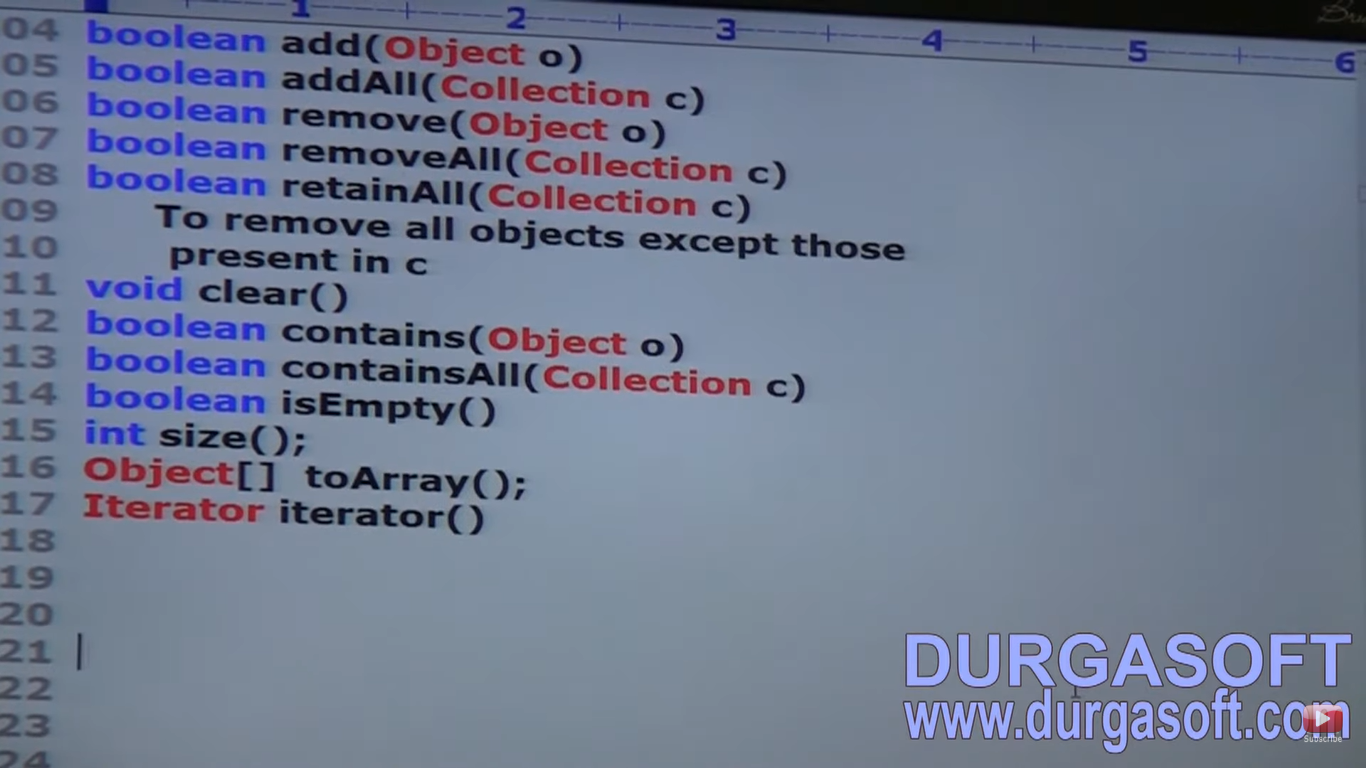
***Utility Classes:-***

*1)Collections*

*2)Arrays*



***Methods of Collection Interface:-***



*Which concrete class implements Collection directly?*

*🡪No concrete class implements Collection directly.*

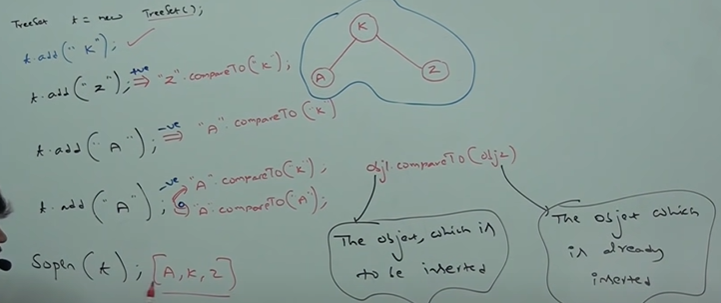
***Comparable (Interface):-***

1. *It is present in java.lang package.*
2. *Used for default natural sorting order.*
3. ***String and all Wrapper Classes*** *implement Comparable interface*
4. *If we are depending on default natural sorting order , then while adding objects into TreeSet then JVM calls compareTo().*

*4. It is used to compare the current object (often referred to as this) with another object passed as an argument (referred to as obj*

***public int compareTo( Object obj) obj1.compareTo(obj2)***

* ***Returns +ve*** *if obj1 comes* ***after*** *obj2* ***obj1 > obj2***
* ***Returns -ve*** *if obj1 comes* ***before*** *obj2* ***obj1 < obj2***
* ***Returns 0*** *if obj1 & obj2 are* ***equal obj1==obj2***



* *If default natural sorting order is not available or if we are not satisfied with default natural sorting order, then we can go for customized sorting by using* ***comparator****.*

***Comparator:-***

* 1. *It is meant for customized sorting order.*
  2. *Present in java.util package.*
  3. *Whenever we are implementing Comparator interface, then compulsory we need to provide implementation for compare(Object obj1, Object obj2) method.*
  4. *Only 2 classes which implement Comparator are 🡪*

***Collator, RuleBasedCollator (GUI Based)***

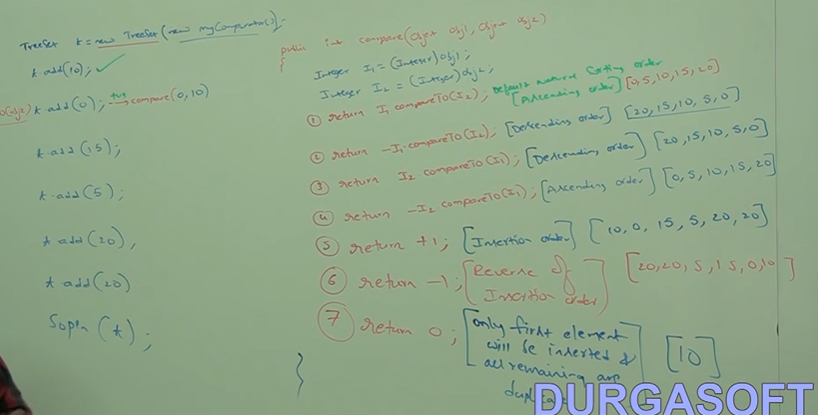
* 1. *It defines 2 methods: -*
* ***public int compare(Object obj1, Object obj2****)*

***🡪 Returns +ve*** *if obj1 comes* ***after*** *obj2* ***obj1 > obj2***

***🡪 Returns -ve*** *if obj1 comes* ***before*** *obj2* ***obj1 < obj2***

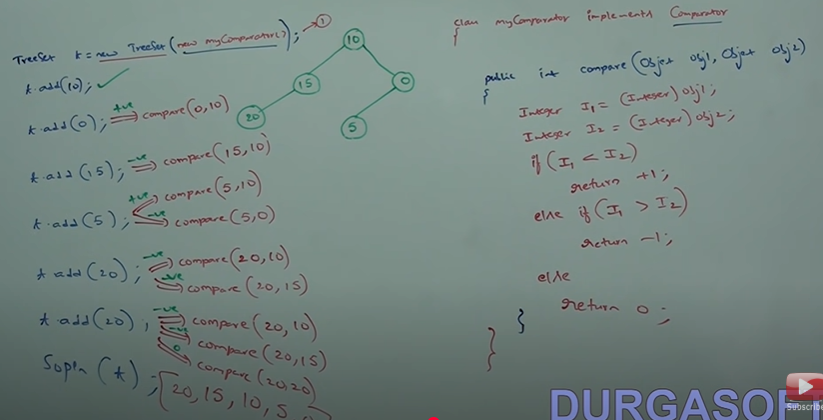
***🡪 Returns 0*** *if obj1 & obj2 are* ***equal obj1==obj2***

***Various possible implementations of compare()***



***🡪 public boolean equals(Object obj)***

*If we want to print objects in descending order*



*If we don’t pass comparator object at line 1 then JVM internally calls compareTo() which is meant for default natural sorting order.*

***(1) Collections class defines the following binary search methods:-***

***Public static int binarySearch(List l, Object target) 🡪returns index***

*🡪If the list is sorted according to default natural sorting order, then we should use above method.*

***Public static int binarySearch(List l,Object target,Comparator c) 🡪 returns index***

*🡪We have to use this method if the list is sorted according to Customized sorting order.*

***Conclusions:-***

*1) The above search methods will internally use binarySearch algorithm, successful search returns index, unsuccessful search returns insertion point.*

*2)Insertion point is the location where we can place target element in the sorted list.*

*3)Before calling binarySearch() compulsory list should be sorted otherwise we will get unpredictable results.*

*4)If the list is sorted using comparator then , at the time of search operation also we have to pass same comparator object, otherwise we will get unpredictable results.*

*5)If the object we are searching isn’t present in sorted list, then it will return the index according to alphabetical/numerical order where it can be placed.*

*For eg:- list=[A,D,G,Q,Z]*

*Sop(Collections.binarySearch(list,”C”));*

*🡪 -2 as C comes between A and D*

***(2)Collections class defines the reversing methods:-***

***Public static void reverse(List l);***

*Collections class defines above method to reverse order of elements of List.*

*Whereas, we can use reverseOrder() to get reversed Comparator*

***Comparator c1=Collections.reverseOrder(List l);***

*InterviewQuestions:-*

[*https://github.com/learning-zone/java-basics/blob/master/collections-questions.md*](https://github.com/learning-zone/java-basics/blob/master/collections-questions.md)

*--------------------------------------------------------------------------------------------------------------------------------------*

***Arrays:-***

* *It is an utility class to define several utility methods for Array Objects.*

*🡪Arrays class defines following sort methods to sort primitive and Object type Arrays:-*

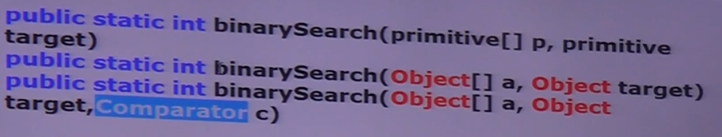
***public static void sort(Primitive[] p) 🡪Sort DNSO.***

***public static void sort(Object[] o) 🡪 Sort DNSO.***

***public static void sort(Object[]o , Comparator c)***

*We can sort primitive [] only based on DNSO, whereas Object[] on DNSO AND CSO.*

*🡪Arrays class defines the following binarySearch methods:-*



*All rules of Arrays class binarySearch() method are exactly same as Collections class binarySearch().*

*🡪Conversion of Arrays to List:-*

***Public static List asList(Object[] o)***

*🡪This method wont create an independent list object for existing array we are getting List view. List l=Arrays.asList(s); -🡪String[] s={“A”,”Z”,”B”}*

*1) By using Array reference we perform any change then automatically that change will be reflected to List.*

*2)By using List reference if we perform any change then change will automatically reflected to Array.*

*3)By using List reference we cant perform any operation which varies/changes the size of array otherwise we will get* ***RuntimeException:UnsupportedOperationException.***

*l.add(“M”) 🡪Invalid*

*l.remove(1) 🡪Invalid*

*l.set(1,”N”) 🡪Valid because it replaces and hence size doesn’t gets changed.*

*4)By using lIst reference we are not allowed to replace with Heterogenous objects otherwise we will get* ***RuntimeException:ArrayStoreException****. This occurs when we try to store heterogenous objects in homogenous array*