JAVA COLLECTIONS

NEED OF COLLECTIONS

------------------------------

**Arrays:**

An array is an indexed collection of fixed number of homogeneous data elements

Advantages:

The main advantage of Arrays is we can represent multiple values with a single variable.

So, that the reusability of the code will be improved.

Limitations:

1. Arrays are fixed in size i.e., once we created an array with some size there is no chance of increasing or decreasing its size based on our requirement. Hence to use arrays compulsory we need to know the size in advance which is always not possible.
2. Arrays can hold only homogeneous data elements.

Example:

Student[] s = new Student[100];

s[0] = new Student(); (Correct)

s[1] = new Customer(); (Incorrect)

But we can resolve this problem using the Object arrays.

Object[] o = new Object[100];

o[0] = new Student(); (Correct)

o[1] = new Customer(); (correct)

1. Arrays concept is not implemented based on some standard data structure hence readymade methods are not available for every requirement we have to write code explicitly. Which is complexity of programming.

To overcome the limitations of arrays we should go for Collections.

1. Collections are growable in nature i.e., Based on requirement we can increase or decrease the size.
2. Collections can hold both homogeneous and heterogeneous elements.
3. Every Collection class is implemented based on some standard data structure. Hence readymade method support is available for every requirement.

DIFFERENCE BETWEEN ARRAYS AND COLLECTIONS

|  |  |
| --- | --- |
| Arrays | Collections |
| Arrays are fixed in the size. | Collections are growable in nature i.e., based on requirement we can increase or decrease. |
| With respect to memory, Arrays are not recommendable to use. | With respect to memory, Collections are recommended to use. |
| With respect to Performance, Arrays are recommended to use. | With respect to performance, Collections are not recommended to use. |
| Arrays can hold only homogeneous data type elements. | Collections can hold both homogeneous and heterogeneous data type elements. |
| There is no underlying data structure for arrays and hence readymade methods support is not available. | Every collection class is implemented based on some standard data structure. Hence readymade methods support is available for every requirement. |
| Arrays can hold both primitive and objects types. | Collections can hold only objects but not primitives. |

COLLECTION AND COLLECTION FRAMEWORK

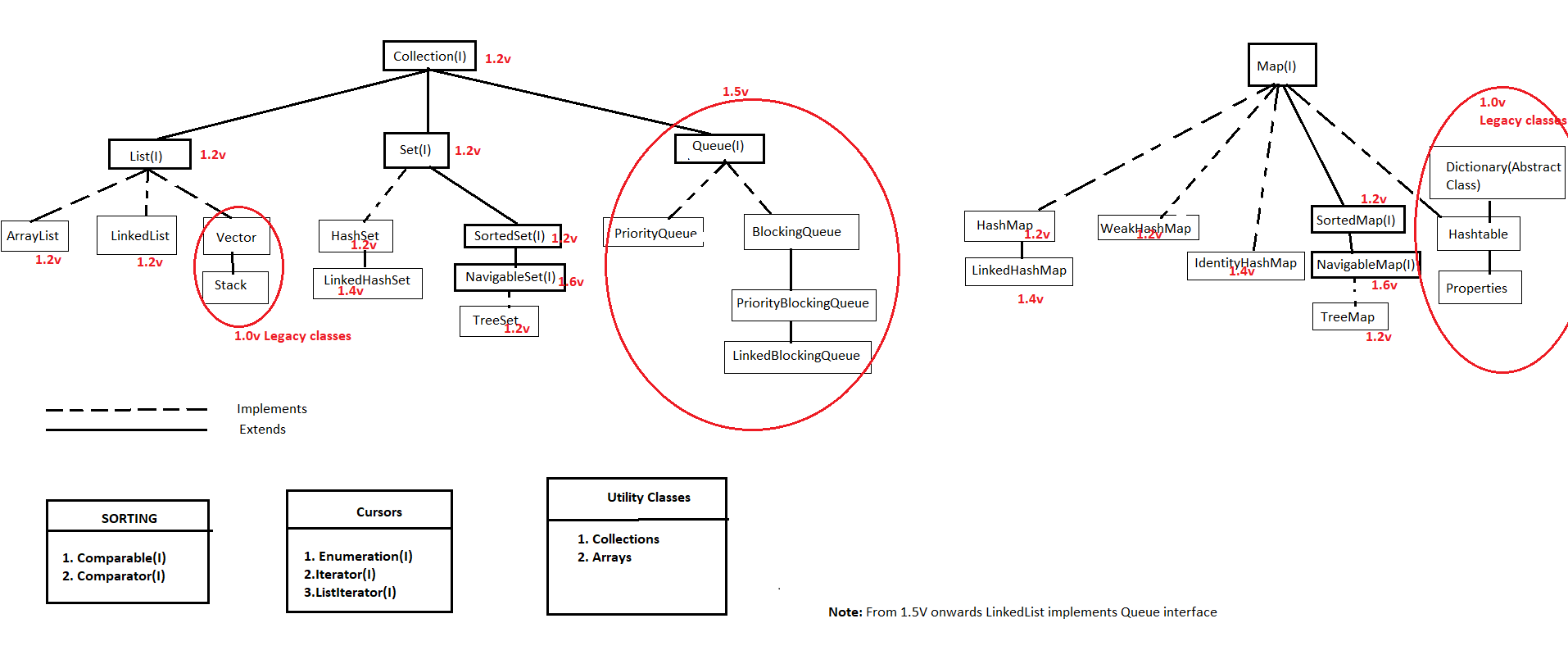
Collection: If we want to represent a group of individual objects as a single entity then we should go for collection.

Collection Framework: It defines classes and interfaces for grouping the individual objects as a single entity.

Java C++

1. Collection 1. Container
2. Collection Framework 2. STL (Standard Template Library)

9-KEY INTERFACES OF COLLECTION FRAMEWORK



1. Collection(1.2v):

* If we want to represent a group of individual objects as a single entity then we should go for Collection.
* Collection interface defines the most common methods which are applicable for every collection object.
* In general Collection interface is considered as root interface of collection framework.

Note: There is no concrete class which implements collection interface directly.

Difference between Collection and Collections?

Collection: Collection is an interface which can be used to represent group of individual objects as a single entity.

Collections: Collections is a utility class present in the java.util.package to define several utility methods (like sorting, searching ...) for Collection objects.

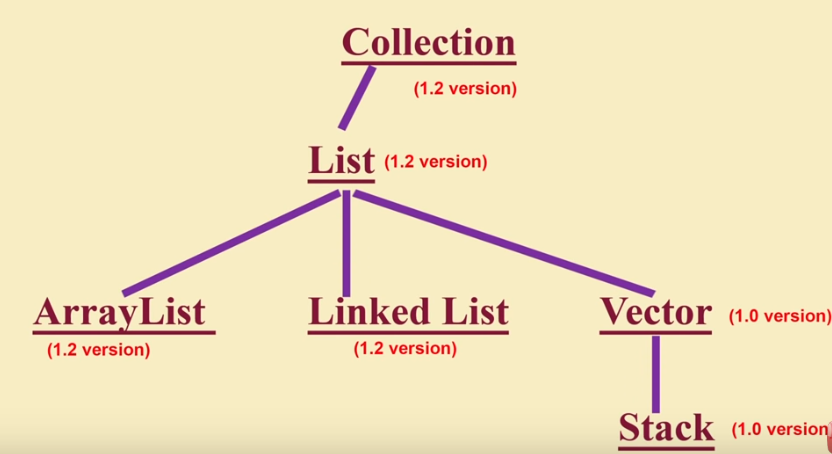
Important Methods of Collection interface:

* boolean add(Object o) : adds an object
* boolean addAll(Collection c): adds a collection of objects
* boolean remove(Object o) : removes an object
* boolean removeAll(Collection c) : removes all specified collection of objects
* boolean retainAll(Collection c) : retains all the objects of a specified collection
* void clear() : clears all
* boolean contains(Object o) : verifies whether an object is available or not
* boolean containsAll(Collection c) : verifies whether all specified objects of collection are available or not.
* boolean isEmpty() : verifies whether collection is empty or not
* int size() : gives size of the collection
* Object[] toArray() : converts collection to array
* Iterator iterator() : iterates through the elements

Note:

Collection interface don't have any method to retrieve object.

2. List(1.2v):



* List is child interface of Collection.
* If we want to represent a group of individual objects as a single entity where
  + duplicates are allowed
  + insertion order preserved

then we should go for List.

* We can differentiate duplicates by using index.
* We can preserve insertion order by using index, hence index plays an important role in List interface.

List Interface Methods:

* boolean add(int index, Object o) - adds object at specified index
* boolean addAll(int index, Collection c) - adds all objects of the given collection in the specified index
* Object get(int index) - gets the object in the specified index
* Object remove(int index) - removes object at the specified index
* Object set(int index, Object new) - replaces the object in the specified index with new object
* int indexOf(Object o) - gives the first index of the specified object
* int lastIndexOf(Object o) - gives the last index of the specified object
* ListIterator listIterator() - iterates through the objects in the list

Implementation classes :

1. ArrayList (1.2v)
2. LinkedList (1.2v)
3. Vector(1.0v) and Stack(1.0v).

Vector and Stack are called Legacy (coming from old generations) Classes.

ArrayList:

* The underlined data structure is resizable or growable Array.
* Duplicates are allowed.
* Insertion order is preserved.
* Heterogeneous objects are allowed [except TreeSet and TreeMap everywhere heterogeneous objects are allowed].
* Null insertion is Allowed.

ArrayList Constructors:

1. ArrayList al = new ArrayList();

* Creates an empty ArrayList object with default initial capacity 10.
* Once the ArrayList reaches its max capacity a new ArrayList will be created with new capacity = (Current Capacity \* 3/2) + 1.

1. ArrayList al = new ArrayList(int initialCapacity);

* Creates an empty ArrayList object with the specified initial capacity.

1. ArrayList al = new ArrayList(Collection c);

* Creates an ArrayList by converting the specified collection of objects.
* Usually we use collections to hold and transfer objects from one place to another place, to provide support for this requirement every collection class implements ***Serializable*** and ***Cloneable***interfaces.
* ArrayList and Vector classes implements the *RandomAccess* interface so that we can access any random element with the same speed.

RandomAccess:

* It is an interface present in java.util package.
* It doesn't contain any methods and it is a Marker interface.

Best Use:

* If our frequent operation is retrieval then ArrayList is the best choice.

Worst Use:

* ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle (because several shift operations are performed for insertion or removal)

**Difference between ArrayList and Vector**

|  |  |
| --- | --- |
| ArrayList | Vector |
| Every method present in ArrayList is not-synchronized. | Every method present in Vector is Synchronized |
| At a time, multiple threads are allowed to operate on ArrayList object and hence ArrayList is not thread safe. | At a time only one thread is allowed to operate on Vector object and hence Vector is thread safe. |
| Threads are not required to wait to operate on ArrayList object, hence relatively high performance. | Threads are required to wait to operate on Vector object and hence relatively low performance. |
| Introduced in 1.2v, not a legacy class. | Introduced in 1.0v. legacy class. |

How to get synchronized version of ArrayList?

* By default ArrayList object is non-synchronized but we can get synchronized version of ArrayList by using Collections class synchronizedList() method.

*public static List synchronizedList(List I);*

ArrayList al = new ArrayList(); //Non-Synchronized

List l = Collections.synchronizedList(al); //Synchronized

* Similarly, we can get the synchronized versions of Set, Map objects by using the following methods of Collections class

public static Set synchronizedSet(Set s);

public static Map synchronizedMap(Map m);

LinkedList:

* The underlying data structure is double Linked List.
* Insertion order is preserved.
* Duplicates are allowed.
* Heterogeneous Objects are allowed.
* Null insertion is possible.
* LinkedList implements Serializable and Cloneable interfaces but not RandomAccess interface.

Best Use:

* LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.

Worst Use:

* LinkedList is the worst choice if our frequent operation is retrieval operation.

Usually we can use LinkedList to implement the Stacks and Queues. To provide support for this requirement, LinkedList class defines following specific methods:

* boolean addFirst(Object o)
* boolean addLast(Object o)
* Object getFirst()
* Object getLast()
* Object removeFirst()
* Object removeLast()

LinkedList Constructors:

1. LinkedList ll = new LinkedList();

* Creates an empty LinkedList object

1. LinkedList ll = new LinkedList(Collection c);

* Creates an equivalent LinkedList Object for the given Collection.

Differences between ArrayList and LinkedList

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| It is the best choice if our frequent operation is retrieval. | It is the best choice if our frequent operation is insertion and deletion. |
| It is the worst choice if our frequent operation is insertion and deletion. | It is the worst choice if our frequent operation is retrieval. |
| Underlying data structure is resizable or growable Array. | Underlying data structure is double LinkedList. |
| ArrayList implements RandomAccess interface | LinkedList doesn't implement RandomAccess interface. |

Vector:

* The underlying data structure is resizeable or growable Array.
* Duplicate objects are allowed.
* Insertion order is maintained.
* Null insertion is allowed.
* Heterogeneous objects are allowed.
* Vector class implements Serializable and Cloneable Interfaces.
* Vector class implements RandomAccess.
* Most methods of vector class are synchronized.Hence Vector object is thread safe.

Best Use:

* if the retrieval is the frequent operation.

Vector Specific Methods:

* Void addElement(Object o)
* boolean removeElement(Object o)
* Void removeElementAt(Object o)
* Void removeAllElements()
* Object elementAt(int index)
* Object firstElement()
* Object lastElement()
* int size()
* int capacity()
* Enumeration elements()

Vector Constructors:

1. Vector v = new Vector();

* Creates an empty vector object with initial capacity of 10, once reaches its max capacity a new vector object will be created with new capacity = 2 \* current capacity.

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

* Creates an empty vector object with the specified initial capacity.

1. Vector v = new Vector(int initialCapacity, int incrementalCapacity);

* Creates an empty vector object with the specified initial capacity and increases by specified incrementalCapacity once its max capacity is reached.

1. Vector v = new Vector(Collection c);

* Creates an equivalent Vector object for the given collection.

Stack:

* Stack is a child class of Vector.
* It is specially designed for Last In First Out (LIFO) order.

Stack class Methods:

* Object push(Object o)
* inserts an object to the stack
* Object pop()
* to remove and return the top of the stack.
* Object peek()
* to return the top of the stack without removing.
* int search(Object obj)
* If the specified object is available it returns its offset from the top of the stack else returns -1.
* boolean empty()
* checks if the stack is empty.

Stack class constructors:

1. Stack s = new Stack();

THREE CURSORS IN JAVA

* If we want to get object one by one from the collection, then we should go for the cursors.
* There are three types of cursors available in java.

1. Enumeration
2. Iterator
3. ListIterator
4. Enumeration:

* It is introduced in java 1.0v (for Legacy)
* We can use enumeration to get object one by one from old collection Objects (Legacy Collections).
* We can create enumeration object by using elements() method of Vector class.

public Enumeration elements();

Ex: Enumeration e = v.elements();

Enumeration Methods:

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

Limitations of Enumeration:

* Enumeration concept is available only for legacy classes and hence it is not a universal cursor.
* By using Enumeration we can get only read access and we can't perform remove operation.

Note: To overcome these limitations we should go for the Iterator cursor.

1. Iterator:

* We can apply Iterator concept for any Collection object hence it is universal cursor.
* By using Iterator, we can perform both read and remove operations.
* We can create Iterator object by using iterator() method of Collection interface.

public Iterator iterator();

Ex: Iterator itr = c.iterator(); where c is any Collection Object.

Iterator Methods:

* Iterator interface defines three methods:
  + public boolean hasNext();
  + public Object next();
  + public void remove();

Iterator Limitations:

* By using Enumeration and Iterator we can move only forward direction and we cannot move to the backward direction, and hence these are single directional cursors.
* By using Iterator cursor we can perform only read and remove operations but we cannot perform replacement of new Objects.

Note: To overcome above limitations of Iterator we should go for ListIterator.

1. ListIterator:

* By using the ListIterator we can move either forward direction or backward direction, and hence ListIterator is bi-directional cursor.
* By using ListIterator, we can perform replacement and addition of new object in addition to read and remove operations.
* We can create the ListIterator object by using the listIterator() method of List Interface

public ListIterator listIterator();

Ex: ListIterator li = l.listIterator(); where l is any List Object.

ListIterator Methods:

* Forward direction
  + public boolean hasNext()
  + public Object next()
  + public int nextIndex()
* backward direction
  + public boolean hasPrevious()
  + public Object previous()
  + public int previousIndex()
* other capability methods
  + public void remove()
  + public void set(Object new)
  + public void add(Object new)

ListIterator Limitations:

* ListIterator is the most powerful cursor but its limitation is, it is applicable only for List implemented class objects and it is not a universal cursor.

Comparison of Enumeration,Iterator and ListIterator cursor

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Enumeration | Iterator | ListIterator |
| Applicable | Only Legacy classes | Any Collection Class | Only List Classes |
| Movement | Forward only | Forward only | Both forward & backward (Bi-directional) |
| Accessibility | Read only | Read and Remove | Read, Remove, Replace and Addition |
| How to get it? | elements() method of Vector | iterator() of Collection(I) | listIterator() of List(I) |
| Mathods | Has only two methods:   1. hasMoreElements() 2. nextElement() | Has three methods:   1. hasNext(), 2. next() 3. remove() | Has 9 methods:   1. hasNext() 2. next() 3. nextIndex() 4. hasPrevious() 5. previous() 6. previousIndex() 7. remove() 8. set(Object new) 9. add(Object o) |
| Is it Legacy | Yes(1.0v) | No(1.2v) | No(1.2v) |

Implementation classes for three cursors:

Vector v = new Vector();

Enumeration e = v.elements();

Iterator i = v.iterator();

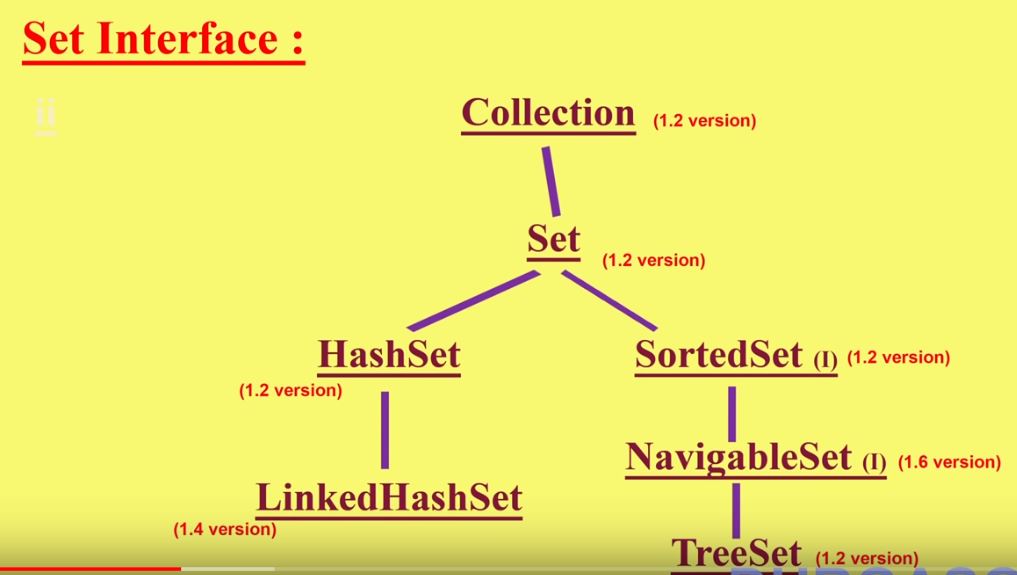
ListIterator li = v.listIterator();

System.out.println(e.getClass().getName()); //java.util.Vector$1

System.out.println(i..getClass().getName()); //java.util.Vector$Itr

System.out.println(li..getClass().getName()); //java.util.Vector$ListItr

3.Set(1.2v):



* Set is the child interface of Collection.
* If we want to represent group of individual objects as a single entity where
  + duplicates are not allowed
  + insertion order is not preserved

then we should go for Set.

* Set interface doesn't contain any new methods. So, we have to use only Collection interface methods.

Differences Between List and Set

|  |  |
| --- | --- |
| List | Set |
| Duplicates are allowed | Duplicates are not allowed |
| Insertion order preserved | Insertion order not preserved |

Implementation classes: HashSet(1.2v), LinkedHashSet(1.4v)

HashSet:

* The underlying data structure is Hashtable.
* Duplicates are not allowed. If we're trying to insert the duplicates, we won't get any compile or runtime errors. add() method simply returns false.
* insertion order is not preserved and all objects will be inserted based on the hash-code of objects.
* Null insertion is allowed.
* Heterogeneous objects are allowed.
* It implements serializable and cloneable interfaces but not RandomAccess interface.

Best Use:

* HashSet is the best choice if our frequent operation in search operation.

HashSet Constructors:

1. HashSet hs = new HashSet();

* Creates an empty HashSet object with default initial capacity of 16 & default fill ratior/load Factor 0.75.

1. HashSet hs = new HashSet(int initialCapacity);

* Creates an empty HashSet object with the specified initial capacity.

1. HashSet hs = new HashSet(int initialCapacity, float loadFactor);

* Creates an empty HashSet object with the specified initial capacity & specified load factor or fill ratio.

1. HashSet hs = new HashSet(Collection c);

* For inter conversion between collections objects.

*Load factor/Fill ratio:* After loading the how much factor, a new HashSet object will be created, that factor is called load factor/fill ratio.

LinkedHashSet:

* LinkedHashSet is the child class of HashSet.
* It is introduced in 1.4v.
* It is exactly same as HashSet except the following differences:

|  |  |
| --- | --- |
| HashSet | LinkedHashSet |
| Underlying data structure is Hashtable | Underlying data structure is Hashtable + LinkedList (This is a hybrid data structure) |
| Insertion order is not preserved | Insertion order is preserved. |
| Introduced in 1.2v | Introduced in 1.4v |

Note:

* LinkedHashSet is the best choice to develop cache based applications, where duplicates are not allowed and insertion order must be preserved.

4. SortedSet(1.2v):

* It is the child interface of Set interface.
* If we want to represent a group of individual objects as a single entity where
  + duplicates are not allowed
  + but objects should be inserted according to some sorting order

then we should go for SortedSet.

SortedSet Specific Methods:

* Object first()
* returns first element of the SortedSet
* Object last()
* returns the last element of the SortedSet
* SortedSet headSet(Object obj)
* returns the SortedSet whose elements are < obj
* SortedSet tailSet(Object obj)
* returns the SortedSet whose elements are >= obj
* SortedSet subSet(Object obj1, Object obj2)
* returns the SortedSet whose elements are >=obj1 and <obj2
* Comparator comparator()
* returns Comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null.

Notes:

1. Default natural sorting order for numbers is Ascending and for Strings Alphabetical order.
2. We can apply above mentioned methods only on SortedSet implemented class objects. That is on the TreeSet object.

5. NavigableSet(1.6v):

* NavigableSet is the child interface of SortedSet.
* It defines several methods for navigation purposes.
* It is the 1.6v enhancement of Collection Framework

NavigableSet Methods:

* Object floor(Object o)
* Returns the greatest element in this set less than or equal to the given element, or *null* if there is no such element.
* Object lower(Object o)
* Returns the greatest element in this set strictly less than the given element, or *null* if there is no such element.
* Object ceiling(Object o)
* Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
* Object higher(Object o)
* Returns the least element in this set strictly greater than the given element, or null if there is no such element.
* Object pollFirst()
* Retrieves and removes the first (lowest) element, or returns null if this set is empty.
* Object pollLast()
* Retrieves and removes the last (highest) element, or returns null if this set is empty.
* NavigableSet<Object> descendingSet()
* Returns a reverse order view of the elements contained in this set.

Example: NavigableSet Demo

|  |
| --- |
| public class NavigableSet\_Practice {  public static void main(String[] args) {    TreeSet<Integer> ts = new TreeSet<Integer>();  ts.add(1000);  ts.add(2000);  ts.add(3000);  ts.add(4000);  ts.add(5000);    System.out.println(ts);  System.out.println(ts.floor(2000));  System.out.println(ts.lower(3000));  System.out.println(ts.ceiling(4000));  System.out.println(ts.higher(4000));  System.out.println(ts.pollFirst());  System.out.println(ts.pollLast());  System.out.println(ts.descendingSet());  System.out.println(ts);  }  }  OUTPUT:  [1000, 2000, 3000, 4000, 5000]  2000  2000  4000  5000  1000  5000  [4000, 3000, 2000]  [2000, 3000, 4000] |

Implementation class: TreeSet(1.2v)

TreeSet:

* The underlying data structure for TreeSet is Balanced Tree.
* Duplicate objects are not allowed.
* Insertion order is not preserved, but all objects will be inserted according to some sorting order.
* Heterogeneous objects are not allowed. If we are trying to insert heterogeneous objects then we will get runtime exception saying ClassCastException.
* Null insertion is allowed, but only once.

TreeSet Constructors:

1. TreeSet t = new TreeSet();

* Creates an empty TreeSet object where elements will be inserted according to customized sorting order.

1. TreeSet t = new TreeSet(Comparator c);
2. TreeSet t = new TreeSet(SortedSet s);
3. TreeSet t = new TreeSet(Collection c);

Null Acceptance:

1. For empty TreeSet as the first element null insertion is possible. But after inserting that null if we are trying to insert any other element, we will get NullPointerException.
2. For non-empty TreeSet if we're trying to insert null then we will get NullPointerException.

Note:

1. If we are depending on natural sorting order then objects should be homogeneous and comparable. Otherwise we will get runtime exception saying ClassCastException.
2. An object said to be comparable if and only if the corresponding class implements java.lang.Comparable interface.
3. String class and all wrapper classes already implements Comparable interface. But StringBuffer doesn't implement comparable interface.

Comparable Interface:

* This interface is present in java.lang package. It contains only one method - compareTo()

public int compareTo(Object o);

Example:

obj1.compareTo(obj2);

* returns (-)ve if obj1 comes before obj2.
* returns (+)ve if obj1 comes after obj2.
* returns 0 if both obj1 and obj2 are equal.
* If we depending on default natural sorting order, internally JVM will call compareTo() method while inserting objects to the TreeSet. Hence the objects should be comparable.

TreeSet t = new TreeSet();

t.add("B");

t.add("Z"); // "Z".compareTo("B"); +ve

t.add("A"); // "A".compareTo("B"); -ve

System.out.println(t); //[A,B,Z]

Note:

1. If we are not satisfied with the default natural sorting order or if the default natural sorting order is not already available then we can define our own customized sorting using Comparator.
2. Comparable meant for default natural sorting order whereas Comparator meant for customized sorting order.

Comparator Interface:

* We can use comparator interface to define our own sorting order (customized sorting order).
* Comparator interface present in java.util package.
* It defines two methods - 1. compare() & 2.equals()

1. compare()

public int compare(Object obj1, Object obj2);

* returns (-) ve if obj1 has to come before obj2.
* returns (+) ve if obj1 has to come after obj2.
* returns 0 if obj1 & obj2 both are equal.

1. equals()

public boolean equals();

Note:

* Whenever we're implementing the Comparator interface, compulsory we should provide implementation for compare() method.
* And implementing equals() method is optional, because it is already available in every java class from Object class through inheritance.

Example: Program to insert Integer objects into the TreeSet where the sorting order is descending order.

|  |
| --- |
| public class TreeSetPractice {  public static void main(String[] args) {  TreeSet<Integer> t = new TreeSet<Integer>(new myComparator());    t.add(10);  t.add(0);  t.add(20);  t.add(20);  t.add(-1);    System.out.println(t);  }  }  //Sorting order in descending by implementing Comparator interface  class myComparator implements Comparator<Object>{  @Override  public int compare(Object arg0, Object arg1) {    Integer i1 = (Integer) arg0;  Integer i2 = (Integer) arg1;    if(i1 < i2)  return +1;  else if(i1 > i2)  return -1;  else  return 0;  }    } |

* If we are not passing the comparator object then internally JVM will call compareTo() method which meant for default natural sorting order (ascending order).
* If we are passing the comparator object then JVM will call compare() method which is meant for customized sorting.

Example 1: Customized sorting integers

|  |
| --- |
| public class TreeSetPractice {  public static void main(String[] args) {  TreeSet<Integer> t = new TreeSet<Integer>(new myComparator());    t.add(10);  t.add(0);  t.add(20);  t.add(20);  t.add(-1);    System.out.println(t);  }  }  //Sorting order in descending by implementing Comparator interface  class myComparator implements Comparator<Object>{  @Override  public int compare(Object arg0, Object arg1) {    Integer i1 = (Integer) arg0;  Integer i2 = (Integer) arg1;  //return i1.compareTo(i2); [-1, 0, 10, 20] ascending order  //return -i1.compareTo(i2); [20, 10, 0, -1] descending order  //return i2.compareTo(i1); [20, 10, 0, -1] descending order  //return -i2.compareTo(i1); [-1, 0, 10, 20] ascending order  //return +1; [10, 0, 20, 20, -1] insertion order  //return -1; [-1, 20, 20, 0, 10] reverse of insertion order  return 0;// [10] (only first element is inserted but all the other elements are considered as duplicates)    }  } |

Example 2: customized sorting Strings

|  |
| --- |
| public class TreeSet\_StringSorting {  public static void main(String[] args) {    TreeSet t = new TreeSet(new myComparsion());  t.add("Ashok");  t.add("Kumar");  t.add("Kumar");  t.add("Kanneganti");  t.add("India");    System.out.println(t);  }  }  class myComparsion implements Comparator{  @Override  public int compare(Object o1, Object o2) {    String s1 = (String) o1;  String s2 = o2.toString();    //return s1.compareTo(s2); [Ashok, India, Kanneganti, Kumar] ascending order  //return -s1.compareTo(s2); [Kumar, Kanneganti, India, Ashok] descending order  //return s2.compareTo(s1); [Kumar, Kanneganti, India, Ashok] descending order  //return +1; [Ashok, Kumar, Kumar, Kanneganti, India] insertion order  //return -1; [India, Kanneganti, Kumar, Kumar, Ashok] reverse of insertion of order  return 0; // [Kumar] (only first element is inserted but all the other elements are considered as duplicates)    }  } |

Example 3: Customized sorting StringBuffer

|  |
| --- |
| public class TreeSet\_StringSorting {  public static void main(String[] args) {    TreeSet t = new TreeSet(new myComparsion());  t.add(new StringBuffer("Ashok"));  t.add(new StringBuffer("Kumar"));  t.add(new StringBuffer("Kumar"));  t.add(new StringBuffer("Kanneganti"));  t.add(new StringBuffer("India"));    System.out.println(t);  }  }  class myComparsion implements Comparator{  @Override  public int compare(Object o1, Object o2) {    String s1 = o1.toString(); //convert the StringBuffer to String by using toString() method  //String s2 = (String)o2; //ClassCastException is thrown if StringBuffer externally type casted  String s2 = o2.toString();    //return s1.compareTo(s2); [Ashok, India, Kanneganti, Kumar] ascending order  //return -s1.compareTo(s2); [Kumar, Kanneganti, India, Ashok] descending order  //return s2.compareTo(s1); [Kumar, Kanneganti, India, Ashok] descending order  //return +1; [Ashok, Kumar, Kumar, Kanneganti, India] insertion order  //return -1; [India, Kanneganti, Kumar, Kumar, Ashok] reverse of insertion of order  return 0; // [Ashok] (only first element is inserted but all the other elements are considered as duplicates)    }  } |

Note:

* If we're defining our own sorting by Comparator, the objects need not be comparable.

Example 3: Sorting a TreeSet containing String and StringBuffer based on customized sorting

|  |
| --- |
| /\* Program to sort the strings based on length and if lengths are equal then sort based on Alphabets\*/  public class TreeSet\_SortingBasedOnStringLength {  public static void main(String[] args) {    TreeSet t = new TreeSet(new myComp());  t.add("AA");  t.add(new StringBuffer("ABCD"));  t.add(new String("XX"));  t.add(new StringBuffer("RBA"));  t.add("XYZX");  t.add("ORX");  System.out.println(t);  }  }  class myComp implements Comparator{    @Override  public int compare(Object o1, Object o2){    String s1 = o1.toString();  String s2 = o2.toString();    int l1 = s1.length();  int l2 = s2.length();    if(l1 < l2)  return -1;  else if(l1 > l2)  return +1;  else  return s1.compareTo(s2); //if string lengths are equal then sort in Alphabetical order    }  } |

Note:

1. If we are depending on default natural sorting order then the objects should be homogeneous and comparable otherwise we will get runtime exception saying ClassCastException.
2. But if we are defining our own sorting by comparator then objects need not be homogeneous and comparable. We can insert heterogeneous and non-comparable objects also.

Comparable vs Comparator Interface

1. For pre-defined comparable classes like String default natural sorting order already available. If we are not satisfied with that, we can define our own sorting by Comparator object.
2. For pre-defined non-comparable classes like StringBuffer default natural sorting order is not already available. We can define required sorting by implementing Comparator operator.
3. For own classes (like Employee, Customer, Student), the person who is writing the classes, he is responsible to define default natural sorting order by implementing Comparable interface.

The person who is using our class, if he is not satisfied with default natural sorting order, then he can define his own sorting order by implementing Comparator interface.

Example:

|  |
| --- |
| public class TreeSet\_UserDefineClass {  public static void main(String[] args) {  Employee e1 = new Employee(200, "Sham");  Employee e2 = new Employee(50, "Ashok");  Employee e3 = new Employee(100, "Ramesh");  Employee e4 = new Employee(50, "Ashok");  Employee e5 = new Employee(150, "Kumar");    //Creating an empty TreeSet object with default sorting order  TreeSet t = new TreeSet();  t.add(e1);  t.add(e2);  t.add(e3);  t.add(e4);  t.add(e5);  System.out.println(t);    //Creating an empty TreeSet object with customized sorting order  TreeSet t1 = new TreeSet(new myCustomizedSort());  t1.add(e1);  t1.add(e2);  t1.add(e3);  t1.add(e4);  t1.add(e5);  System.out.println(t1);    }  }  //This is user defined class implementing Comparable interface  class Employee implements Comparable{  private int id;  private String name;  public Employee(int id, String name) {  this.id = id;  this.name=name;  }  public String toString() {  return this.getName() +"-"+this.id;  }  @Override  public int compareTo(Object emp) {  int id1 = this.id;  Employee e = (Employee)emp;  int id2 = e.id;    if(id1 < id2)  return -1;  else if(id1 > id2)  return +1;  else  return 0;  }  public String getName() {  return name;  }  }  // Customized sorting the employee name in the Alphabetical order  class myCustomizedSort implements Comparator{  @Override  public int compare(Object o1, Object o2) {  Employee e1 = (Employee) o1;  Employee e2 = (Employee) o2;    String s1 = e1.getName();  String s2 = e2.getName();    return s1.compareTo(s2);  }    }  OUTPUT:  [Ashok-50, Ramesh-100, Kumar-150, Sham-200]  [Ashok-50, Kumar-150, Ramesh-100, Sham-200] |

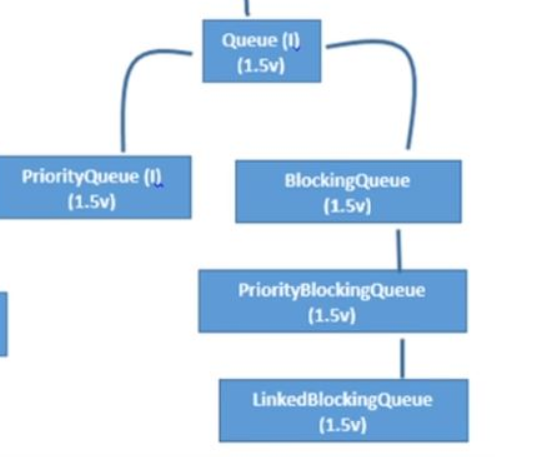
Comparison between Comparable and Comparator Interfaces

|  |  |
| --- | --- |
| Comparable | Comparator |
| It is meant for default sorting order. | It is meant for customized sorting order. |
| Present in java.lang package. | Present in java.util package. |
| Contains only one method:   1. compareTo() | Contains only two methods:   1. compare() 2. equals() |
| All wrapper classes and String class implements comparable interface. | The only implemented classes of Comparator interface are Collator and RuleBasedCollator. |

Comparison of Set(I) implemented classes

|  |  |  |  |
| --- | --- | --- | --- |
| Property | HashSet | LinkedHashSet | TreeSet |
| Underlying Data Structure | Hashtable | Hashtable+Linkedlist | Balanced Tree |
| Insertion order | Not Preserved | Preserved | Not Preserved |
| Sorting order | Not Applicable | Not Applicable | Applicable |
| Heterogeneous Objects | Allowed | Allowed | Not Allowed |
| Duplicate Objects | Not Allowed | Not Allowed | Not Allowed |
| Null Acceptance | Allowed.Only once | Allowed.Only once | For empty TreeSet as first element null is allowed and in all other cases We will get NullPointerException. |

6. Queue(1.5v):



* Queue is the child interface of Collection.
* If we want to represent a group of individual objects prior to processing then we should go for Queue.
* First In First out concept is the queue concept.
* Usually Queue follows First In First Out order but based on our requirement, we can implement our own priority order also(Priority Queue).
* From 1.5V onwards LinkedList class also implements Queue interface.
* LinkedList based implementation of Queue always follows First In First out order

Queue Interface Specific Methods:

* boolean offer(Object o)
* To add object into the queue.
* Object peek()
* To return head element of the queue. If queue is empty then this method should return null.
* Object element()
* To return head element of the queue. If the queue is empty then this method throws a RuntimeException saying - *NoSuchElementException*.
* Object poll()
* To remove and return head element of the queue. If the queue is empty then this method should return null.
* Object remove()
* To remove and return head element of the queue. If the queue is empty then this method throws a RuntimeException - *NoSuchElementException*.

Implementation classes :

PriorityQueue(1.5v)

BlockingQueue(1.5v) - PriorityBlockingQueue(1.5v) & LinkedBlockingQueue(1.5v)

PriorityQueue:

* If we want to represent a group of individual objects prior to processing according to some priority then we should go for PriorityQueue.
* The priority can be either default natural sorting order or customized sorting order defined by Comparator.
* Insertion order is not preserved and it is based on some priority.
* Duplicate objects are not allowed.
* If we are depending on Default natural sorting order compulsory the objects should be Homogeneous and Comparable. Otherwise, we’ll get RuntimeException saying *ClassCastException*.
* If we are defining our own sorting by Comparator then objects need not be Homogeneous and Comparable.
* Null is not allowed even as the first element also.

PriorityQueue Constructors:

1. PriorityQueue pq = new PriorityQueue();

* Creates an empty PriorityQueue with default initial capacity 11 and all objects will be inserted according to default natural sorting order.

1. PriorityQueue pq = new PriorityQueue(int initialCapacity);

* Creates an empty PriorieryQueue with the specified initial capacity.

1. PriorityQueue pq = new PriorityQueue(int initialCapacity, Comparator c);

* Creates an empty PriorityQueue with specified initial Capacity and all objects will be inserted according to specified customized sorting order.

1. PriorityQueue pq = new PriorityQueue(Sorted Set);
2. PriorityQueue pq = new PriorityQueue(Collection c);

Example:

|  |
| --- |
| public class PriorityQueue\_Practice {  public static void main(String[] args) {  PriorityQueue pq = new PriorityQueue();    System.out.println(pq.peek()); //null  //System.out.println(pq.element()); //NullPointerException  System.out.println(pq.poll()); //null  //System.out.println(pq.remove()); //NoSuchElementException  for(int i=1;i<=10;i++){  pq.offer(i);  }  System.out.println(pq); //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  System.out.println(pq.peek()); //1  System.out.println(pq.element()); //1  System.out.println(pq.poll()); //1  System.out.println(pq.remove()); //2  System.out.println(pq); //[3, 4, 6, 8, 5, 9, 7, 10]  }  } |
| Note: Some platforms won’t provide proper support for Thread Priorities and PriorityQueues |

Example: Program for customized priority

|  |
| --- |
| public class PriorityQueue\_Practice1 {  public static void main(String[] args) {    PriorityQueue pq = new PriorityQueue(15,new myCompar());    pq.offer("A");  pq.offer("Z");  pq.offer("L");  pq.offer("B");    System.out.println(pq); //[z,L,B,A]  }  }  class myCompar implements Comparator{  @Override  public int compare(Object arg0, Object arg1) {    String s1 = arg0.toString();  String s2 = arg1.toString();    return s2.compareTo(s1);  }  } |

Note:

* All the above interfaces (Collection, List, Set, SortedSet, NavigableSet, Queue) meant for representing a group of individual objects.
* If we want to represent a group of objects as key value pair then we should go for Map interface.

7. Map(1.2v):

* Map is not the child interface of Collection.
* If we want to represent a group of objects as key value pairs then we should go for Map.
  + Both key and value are Objects.
  + Duplicate keys are not allowed.
  + Values can be duplicated.
* Each key-value pair is called 'Entry'. Hence map is considered as collection of entry objects.

Map interface methods:

* Object put(Object key, Object value);
* to add one key-value pair to the map, if the key is already represent then the old value will be replaced with the new value and returns the old value.

Example:

System.out.println(map.put(1, "Kumar"));

System.out.println(map.put(2, "Kanneganti"));

System.out.println(map.put(1, "Ashok"));

Output:

null

null

Kumar

* void putAll(Map m);
* Object get(Object key);
* Object remove(Object key);
* boolean containsKey(Object key);
* boolean containsValue(Object value);
* boolean isEmpty();
* int size();
* void clear();
* Set keySet();
* returns set of keys from the map
* Collection values();
* returns collection of values from the map
* Set entrySet();
* returns set of the entries(key-value pairs) from the map.

Note: keySet(), values() & entrySet() are the collection views of Map

Entry Interface:

A Map is a group of key-value pairs and echa key-value pair is called an entry. Hence Map is considered as a collection of Entry objects. Without existing of Map object there is no chance of existing entry object.Hence Entry interface is defined inside Map interface.

Entry interface methods:

* Object getKey();
* Object getValue();
* Object setValue(Object new); //Replaces old value with new value by using this method.

These three above methods are entry specific methods and applied on entry object only.

Implementation classes:

HashMap(1.2v)

LinkedHashMap(1.4v)

WeakHashMap(1.2v)

IdentityHashMap(1.4v)

Dictionary(Abstract Class) ←-------- Hashtable ----------> Properties (all from 1.0v called Legacy classes)

Hashtable(1.0v) implements Dictionary(Abstract Class 1.0v)

Properties(1.0v) is the child class of Hashtable(1.0v).

HashMap:

* Underlying data structure is Hashtable.
* Insertion order is not preserved and it is based on hashcode of keys.
* Duplicate keys are not allowed but values can be duplicated.
* Heterogeneous objects are allowed for both key and value.
* Null is allowed for key (Only once)
* Null is allowed for values (any number of times)
* HashMap implements Serializable and Clonable interfaces but not RandomAccess.

Best Use:

* HashMap is the best choice if our frequent operation is search operation

HashMap Constructors:

1. HashMap hm = new HashMap()

* Creates an empty HashMap object with default initial capacity 16 and default fill ratio 0.75.

1. HashMap hm = new HashMap(int initialCapacity)

* Creates an empty HashMap object with specified initial capacity and default fill ratio 0.75.

1. HashMap hm = new HashMap(int initialCapacity, float fillratio)

* Creates an empty HashMap object with specified initial capacity and specified fill ratio.

1. HashMap hm = new HashMap(Map m)

Example:

|  |
| --- |
| public class HashMapPractice {  public static void main(String[] args) {    HashMap map = new HashMap();    map.put(100,"Rajesh");  map.put(101,"Kumar");  map.put(102,"Kanneganti");    System.out.println(map); //{100=Rajesh, 101=Kumar, 102=Kanneganti}  System.out.println(map.put(100,"Ashok")); //Rajesh    Set set = map.keySet();  System.out.println(set); //[100, 101, 102]    Collection values = map.values();  System.out.println(values); //[Ashok, Kumar, Kanneganti]    Set entries = map.entrySet();  System.out.println(entries); //[100=Ashok, 101=Kumar, 102=Kanneganti]    Iterator itr = entries.iterator();  while(itr.hasNext()){  Map.Entry m = (Map.Entry)itr.next();  System.out.println(m.getKey() + " " + m.getValue()); // key-value pair  if(m.getValue().equals("Kumar")){  System.out.println(m.setValue("Rajesh")); //Kumar  }  }  }  } |

Differences between HashMap and Hashtable

|  |  |
| --- | --- |
| HashMap | Hashtable |
| Every method present in HashMap is not synchronized. | Every method present in Hashtable is synchronized. |
| At a time multiple threads are allowed to operate on HashMap object and hence it is not thread safe. | At a time only one thread is allowed to operate on a Hashtable object and hence it is thread safe. |
| 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 objects. |
| Null is allowed for both key and value | Null is not allowed for keys and values. Otherwise, we will get NullPointerException. |
| Introduced in 1.2v and it is not legacy | Introduced in 1.0v and it is Legacy |

How to get synchronized version of HashMap Object?

By default, HashMap is not synchronized but we can get synchronized version of HashMap by using synchronizedMap() method of Collections class.

HashMap m = new HashMap(); //*non-synchronized*

Map m1 = Collections.synchronizedMap(m) //*synchronized.*

LinkedHashMap:

* It is the child class of HashMap.
* It is exactly same as HashMap (including methods and constructors) except the following differences:

|  |  |
| --- | --- |
| HashMap | LinkedHashMap |
| Underlying data structure is Hashtable | Underlying data structure is a combination of LinkedList and Hashtable (Hybrid Data structure) |
| Insertion order is not preserved and it is based on hashcode of keys. | Insertion order is preserved. |
| Introduced in 1.2v. | Introduced in 1.4v. |

Best use:

* LinkedHashSet and LinkedHashMap are commonly used for developing chache based applications.

Difference between == operator and .equals() method

In general == operator meant for reference comparison (address comparison). whereas .equals() method meant for content comparison.

Example:

|  |
| --- |
| Integer I1 = new Integer(10);  Integer I2 = new Integer(10);  System.out.println(I1 == I2) // false  System.out.println(I1.equals(I2)) // true |

IdentityHashMap:

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

Example:

|  |
| --- |
| public class HashMapPractice {  public static void main(String[] args) {    Integer i1 = new Integer(10);  Integer i2 = new Integer(10);    //HashMap  HashMap map = new HashMap();  map.put(i1, "Ashok");  map.put(i2, "Kanneganti");    System.out.println(map); //{10=Kanneganti}    //IdentityHashMap  IdentityHashMap map1 = new IdentityHashMap();  map1.put(i1, "Ashok");  map1.put(i2, "Kanneganti");    System.out.println(map1); //{10=Ashok, 10=Kanneganti}  }  } |

WeakHashMap:

* It is exactly same as HashMap except the following difference:
  + In the case of HashMap even though object doesn't have any reference it is not eligible for Garbage Collection if it is associated with HashMap i.e., HashMap dominates Garbage Collector.
  + But in the case of WeakHashMap, if the object doesn't contain any references it is eligible for Garbage Collection even though object associated with WeakHashMap i.e., Garbage Collector dominates WeakHashMap.

Example: HashMap with Garbage Collector

|  |
| --- |
| public class WeakHashMap {  public static void main(String[] args) {  Temp t = new Temp();    HashMap hashmap = new HashMap();  hashmap.put(t, "Ashok");  System.out.println(hashmap);    t = null; //reference set to null  System.gc();  System.out.println(hashmap);  }  }  class Temp{    public String toString(){  return "temp";  }    public void finalize(){  System.out.println("finalize() method called");  }  }  In the above example, temp object is not eligible for Garbage Collection because it is associated with HashMap.  So, the output is:  {temp=Ashok}  {temp=Ashok} |

In the above program if we replace HashMap with WeakHashMap then the temp object is eligible for Garbage Collection (See the below program)

Example: WeakHashMap with GC

|  |
| --- |
| public class WeakHashMap\_Practice {  public static void main(String[] args) {  Temp t = new Temp();    WeakHashMap hashmap = new WeakHashMap();  hashmap.put(t, "Ashok");  System.out.println(hashmap);    t = null; //reference set to null  System.gc();  System.out.println(hashmap);  }  }  class Temp{    public String toString(){  return "temp";  }    public void finalize(){  System.out.println("finalize() method called");  }  }  OUTPUT:  {temp=Ashok}  {}  finalize() method called |

8. SortedMap(1.2v):

* SortedMap is the child interface of Map.
* If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.
* Sorting is based on the key but not based on value.

SortedMap Interface Methods:

It defines the following specific methods -

* Object firstKey();
* Object lastKey();
* SortedMap headMap(Object Key);
* SortedMap tailMap(Object Key);
* SortedMap subMap(Object key1, Object key2);
* Comparator comparator();

Example:

|  |
| --- |
| public class Map\_Practice {  public static void main(String[] args) {    TreeMap t = new TreeMap();    t.put(101, "A");  t.put(103, "B");  t.put(104, "C");  t.put(107, "D");  t.put(125, "E");  t.put(136, "F");    System.out.println(t); //{101=A, 103=B, 104=C, 107=D, 125=E, 136=F}  System.out.println(t.firstKey()); //101  System.out.println(t.lastKey()); //136  System.out.println(t.headMap(107)); //{101=A, 103=B, 104=C}  System.out.println(t.tailMap(107)); //{107=D, 125=E, 136=F}  System.out.println(t.subMap(103, 125)); //{103=B, 104=C, 107=D}  }  } |

9. NavigableMap(1.6v):

* NavigableMap is the child interface of SortedMap.
* It defines several methods for navigation purposes.
* It is the 1.6v enhancement of Collection Framework.

NavigableMap Methods:

* K floorKey(K k)
* Returns the greatest key less than or equal to the given key, or null if there is no such key.
* K lowerKey(K k)
* Returns the greatest key strictly less than the given key, or null if there is no such key.
* K ceilingKey(K k)
* Returns the least key greater than or equal to the given key, or null if there is no such key.
* K higherKey(K k)
* Returns the least key strictly greater than the given key, or null if there is no such key.
* Map.Entry<K,V> pollFirstEntry()
* Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
* Map.Entry<K,V> pollLastEntry()
* Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
* NavigableMap<K,V> descendingMap()
* Returns a reverse order view of the mappings contained in this map.

Example: NavigableMap Program

|  |
| --- |
| public class NavigableMap\_Practice {  public static void main(String[] args) {    TreeMap<String,String> tm = new TreeMap<String,String>();    tm.put("b", "bangalore");  tm.put("e", "epurupalem");  tm.put("a", "andhra");  tm.put("f","florida");  tm.put("c", "chirala");  tm.put("d", "delhi");  System.out.println(tm);  System.out.println(tm.floorKey("d"));  System.out.println(tm.floorEntry("d"));  System.out.println(tm.lowerKey("d"));  System.out.println(tm.lowerEntry("d"));  System.out.println(tm.ceilingKey("d"));  System.out.println(tm.ceilingEntry("d"));  System.out.println(tm.higherKey("d"));  System.out.println(tm.higherEntry("d"));  System.out.println(tm.firstKey());  System.out.println(tm.firstEntry());  System.out.println(tm.lastKey());  System.out.println(tm.lastEntry());  System.out.println(tm.pollFirstEntry());  System.out.println(tm.pollLastEntry());  System.out.println(tm.descendingMap());  System.out.println(tm);  }  }  OUTPUT:  {a=andhra, b=bangalore, c=chirala, d=delhi, e=epurupalem, f=florida}  d  d=delhi  c  c=chirala  d  d=delhi  e  e=epurupalem  a  a=andhra  f  f=florida  a=andhra  f=florida  {e=epurupalem, d=delhi, c=chirala, b=bangalore}  {b=bangalore, c=chirala, d=delhi, e=epurupalem} |

Implementation class : TreeMap(1.2v)

TreeMap:

* Underlying data structure is Red-Black Tree.
* Insertion order is not preserved and it is based on some sorting order of keys.
* Duplicate keys are not allowed but values can be duplicated.
* Heterogeneous objects value acceptance:

Keys -

* If we are depending on default natural sorting order then keys should be Homogeneous and Comparable. Otherwise, we will get runtime exception saying ClassCastException.
* If we are defining our own sorting by Comparator then keys need not be Homogeneous and Comparable.We can take Heterogeneous non-comparable objects also.

Values -

* Values can be Heterogeneous object values if default or customized sorting followed.
* Null acceptance:

Key -

For Java version >= 1.7

* Null is not accepted.

For Java version <= 1.6

* for non-empty TreeMap, if we're inserting an entry with null key we get NullPointerException.
* for empty TreeMap, we can insert an entry will null key but will throw NullPointerException if we try to insert another value.

Values -

we can insert multiple null values.

TreeMap Constructors:

1. TreeMap t = new TreeMap();
2. TreeMap t = new TreeMap(Comparator c);
3. TreeMap t = new TreeMap(Map m);
4. TreeMap t = new TreeMap(SortedMap sm);

Hashtable:

* Underlying data structure for Hashtable is Hashtable.
* Inserting order is not preserved and it is based on Hashcode of keys.
* Duplicate keys are not allowed but values can be duplicated.
* Heterogeneous objects are allowed for both keys and values.
* Null is not allowed for both key and value. Otherwise we'll get runtime exception saying NullPointerException.
* It implements Serializable and Cloneable interfaces but not RandomAccess interface.
* Every method present in Hashtable is synchronized and Hashtable object is thread safe.

Best use:

* Hashtable is the best choice if our frequent operation is search operation.

Hashtable Constructors:

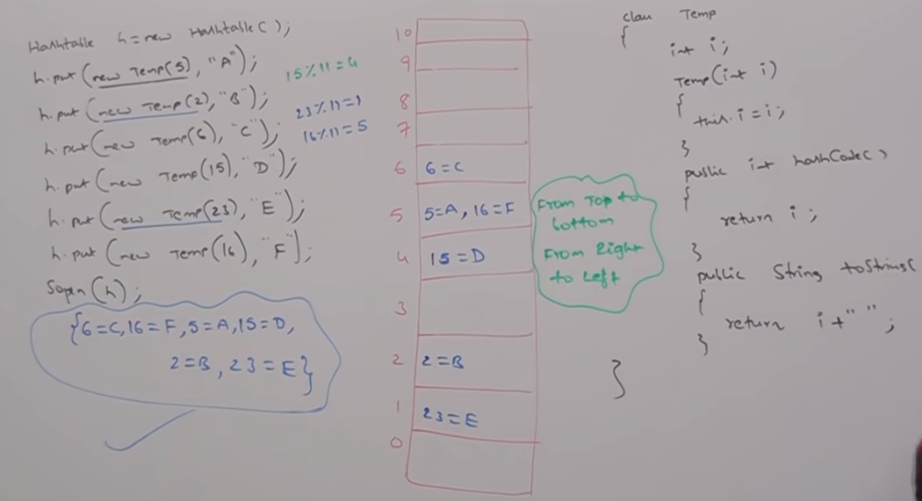
1. Hashtable h = new Hashtable();

* Creates an empty Hashtable object with default initial capacity 11 and default fill ratio 0.75.

1. Hashtable h = new Hashtable(int initialCapacity);
2. Hashtable h = new Hashtable(int initialCapacity, float fillratio);
3. Hashtable h = new Hashtable(Map m);

Example:

|  |
| --- |
| public class Hashtable\_Practice {  public static void main(String[] args) {  Hashtable h = new Hashtable();    h.put(new TempClass(5), "A"); //hashcode = 5  h.put(new TempClass(2), "B"); //hashcode = 2  h.put(new TempClass(6), "C"); //hashcode = 6  h.put(new TempClass(15), "D"); //hashcode = 15%11 = 4  h.put(new TempClass(23), "E"); //hashcode = 23%11 = 1  h.put(new TempClass(16), "F"); //hashcode = 16%11 = 5  //h.put(new TempClass(17), null); NullPointerException is thrown    System.out.println(h);  }  }  class TempClass{    int i;    public TempClass(int i){  this.i=i;  }    public int hashCode(){  return i;  }    public String toString(){  return i+"";  }  }  OUTPUT:  {6=C, 16=F, 5=A, 15=D, 2=B, 23=E} |

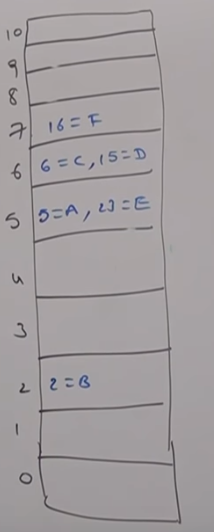


In the Hashtable the values are stored as c at 6, (A,F) at 5, D at 4, B at 2, E at 1. When you print the Hashtable the values are printed from top to bottom and right to left.

If we change the hashCode() method of TempClass as below

|  |
| --- |
| public int hashCode(){  return i%9;  } |

And the values are stored in the Hashtable as follows



Output also changes as follows:

{16=F, 15=D, 6=C, 23=E, 5=A, 2=B}

Properties:

In our program if anything which changes frequently (like username, password etc) are not recommended to hard code in Java program. If there is any change to reflect that change recompilation,rebuild and redeploy application are required even sometimes server restart also required which creates a big business impact to the client.

To overcome this problem, properties file is used. Such type of variable things we have to configure in properties file. From that properties file we have to read into java program and we can use those properties.

The main advantage of this approach is if there is a change in properties file to reflect that change, we just need to re-deploy is enough.

We can use Java properties object to hold properties which are coming from Properties file.

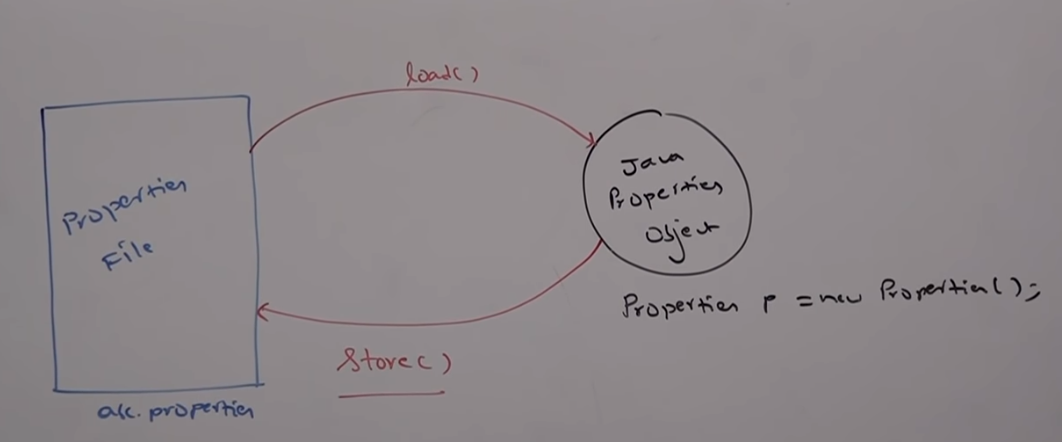
In normal map (like HashMap, Hashtable, TreeMap), key and value can be any type but in case of Properties key and value should be String type.

Properties Constructor:

1. Properties p = new Properties();

Properties Methods:

* String setProperty(String pName, String pValue);
* To set a property with new value and it returns the old value of the property.
* String getProperty(String pName);
* To get value associated with the specified property.
* Enumeration propertyNames();
* Void load(InputStream is);
* To load properties from properties file into Java Properties object.
* Void store( OutputStream os, String comment);
* To store properties from Java Properties object into Properties file.



Example:

abc.properties file:

|  |
| --- |
| userName=ashok  pwd=Sec@124 |

Java program:

|  |
| --- |
| public class Properties\_Practice {  public static void main(String[] args) throws IOException {  FileInputStream fis = new FileInputStream("C:\\Users\\Ashok\\workspace\\LearningJavaConcepts\\src\\Collections\\abc.properties");  Properties prop = new Properties();  prop.load(fis);  System.out.println(prop.getProperty("userName"));  System.out.println(prop.getProperty("pwd"));  prop.setProperty("KAK", "K Ashok Kumar");  FileOutputStream fos = new FileOutputStream("C:\\Users\\Ashok\\workspace\\LearningJavaConcepts\\src\\Collections\\abc.properties");  prop.store(fos, "Changes added by Ashok");  }  }  OUTPUT:  ashok  Sec@124 |

abc.properties file:

|  |
| --- |
| #Changes added by Ashok  #Sun Apr 28 17:13:33 EDT 2019  KAK=K Ashok Kumar  userName=ashok  pwd=Sec@124 |

1.6V enhancements in Collection Framework:

As part of 1.6v enhancement, the following two concepts introduced in CollectionFramework:

1. NavigableSet
2. NavigableMap

Explanation is provided in the respective interfaces.

Utility Classes

There are two utility classes:

1. Collections
2. Arrays

Collections:

* Collections class defines several utility methods for Collection objects like sorting, searching, reversing, etc.

Sorting elements of List:

Collections class defines the following two sort methods -

1. Public static void sort(List l);

* To sort based on Default Natural Sorting Order.
* In this case, List should compulsory contain Homogeneous and Comparable objects, otherwise we will get runtimeexception saying *ClassCastException*.
* List should not contain null otherwise, we will get *NullPointerException*.

1. Public static void sort(List l, Comparator c);

* To sort based on Customized sorting order.

Example: Program of Collections class Default sorting Order

|  |
| --- |
| public class CollectionsSorting\_Practice {  public static void main(String[] args) {    ArrayList al = new ArrayList();    al.add("Z");  al.add("A");  al.add("N");  al.add("E");  //al.add(new Integer(10)); //ClassCastException thrown calling sort() method of Collections class  //al.add(null); //NullPointerException thrown on calling sort() method of Collections class    System.out.println(al); //[Z,A,N,E]  Collections.sort(al);  System.out.println(al); //[A,E,N,Z]  }  } |

Example: Program of Collections class Customized Sorting Order

|  |
| --- |
| public class CollectionsSorting\_Practice1 {  public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add("Z");  al.add("A");  al.add("N");  al.add("E");  System.out.println(al); //[Z, A, N, E]  Collections.sort(al,new myCompars());  System.out.println(al); //[Z, N, E, A]  }  }  class myCompars implements Comparator {  public int compare(Object arg0, Object arg1) {  String s1 = arg0.toString();  String s2 = arg1.toString();  return s2.compareTo(s1);  }  } |

Searching Elements of List:

Collections class defines the following binarySearch() methods:

1. public static int binarySearch(List l, Object target);

* If the List is sorted according to default natural sorting order then we have to use this method.

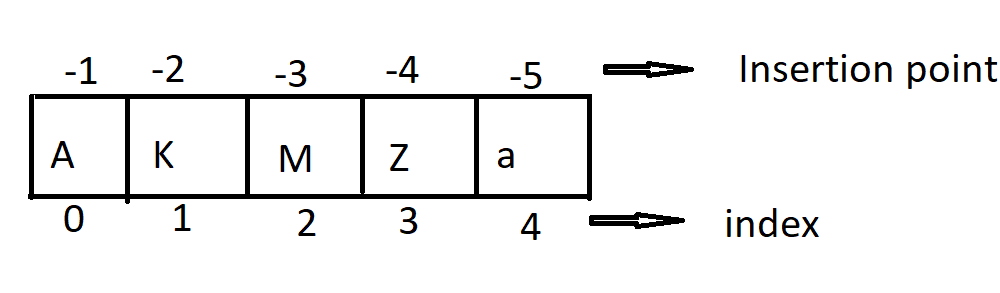
1. public static int binarySearch(List l, Object target, Comparator c);

* This method is used if the List is sorted according to customized sorting order.

Conclusions:

The above search methods internally will use binary search algorithm.

* Successful search return index
* Unsuccessful search insertion point (insertion point is the location where we can place the target element in the sorted list.)
* Before calling binarysearch method compulsory list should be sorted.Otherwise, we will get unpredictable results.
* If the list is sorted according to comparator then at the time of search operation also we have to pass same comparator object. Otherwise, we will get unpredictable results.



Example: Collections binarysearch with default sorting

|  |
| --- |
| public class CollectionsBinarySearch\_Practice {  public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add("Z");  al.add("A");  al.add("M");  al.add("K");  al.add("a");    System.out.println(al);  Collections.sort(al);  System.out.println(al);  System.out.println(Collections.binarySearch(al, "M"));  System.out.println(Collections.binarySearch(al, "J"));  }  } |

Example: Program to perform binary search for the list sorted according to Comparator

|  |
| --- |
| public class CollectionsBinarySearch\_Practice1 {  public static void main(String[] args) {    ArrayList al = new ArrayList();  al.add(15);  al.add(0);  al.add(20);  al.add(10);  al.add(5);    System.out.println(al);  Collections.sort(al,new myIntComparator());  System.out.println(al);  System.out.println(Collections.binarySearch(al,10,new myIntComparator()));  System.out.println(Collections.binarySearch(al,11,new myIntComparator()));  //System.out.println(Collections.binarySearch(al,11)); //unpredictable  }  }  class myIntComparator implements Comparator{  @Override  public int compare(Object o1, Object o2) {  Integer i1 = (Integer) o1;  Integer i2 = (Integer) o2;  return i2.compareTo(i1);  }  }  OUTPUT:  [15, 0, 20, 10, 5]  [20, 15, 10, 5, 0]  2  -3  -6 |

Note:

For the List of ‘n’ elements in the case of binarySearch() method

* Successful result range: 0 to n-1
* Unsuccessful result range: -(n+1) to -1
* Total result range: -(n+1) to (n-1)

Example:

List of 3 elements

-1 -2 -3 -4

|  |  |  |
| --- | --- | --- |
| A | K | Z |

0 1 2

* Successful result range: 0 to 2
* Unsuccessful result range: -4 to -1
* Total result range: -4 to 2

Reversing elements of List:

* Collections class defines the following reverse() method to reverse elements of List

public static void reverse(List l);

Example:

|  |
| --- |
| public class CollectionsReverse {  public static void main(String[] args) {    ArrayList al = new ArrayList();    al.add(15);  al.add(0);  al.add(20);  al.add(10);  al.add(5);    System.out.println(al); //[15, 0, 20, 10, 5]  Collections.reverse(al);  System.out.println(al); //[5, 10, 20, 0, 15]  }  } |

reverse() vs reverseOrder()

We can use reverse() method to reverse order of elements of List.

Whereas, we can use reverseOrder() to get reversed comparator

Comparator c1 = Collections.reverseOrder(Comparator c);

Arrays:

* Arrays class is an utility class to define several utility methods for array objects or array.

1. Sorting elements of Array:

Arrays class defines the following sort methods to sort elements of primitive and object type arrays’

* public static void sort(primitive[] p);
* To sort primitive array elements according to default natural sorting order.
* public static void sort(Object[] obj);
* To sort Object array elements according to default natural sorting order.
* public static void sort(Object[] obj,Comparator c);
* To sort Object array elements according to customized sorting order.

Example: Default Natural Sorting of Primitive Arrays and Object Arrays

|  |
| --- |
| public class ArraysDNSO\_Practice {  public static void main(String[] args) {    int[] a = new int[] {8,1,7,2,5};  System.out.print("Array Elements Before Sorting: ");  for(int i : a)  System.out.print(" "+i);  System.out.println();  Arrays.sort(a);  System.out.print("Array ELements After Sorting: ");  for(int j : a)  System.out.print(" "+j);  System.out.println();  System.out.print("Array before sorting: ");  String[] s = {"A","Z","B"};  for(String str : s)  System.out.print(" "+str);  System.out.println();  Arrays.sort(s);  System.out.print("Arrays after sorting: ");  for(String ss : s)  System.out.print(" "+ss);    }  }  OUTPUT:  Array Elements Before Sorting: 8 1 7 2 5  Array ELements After Sorting: 1 2 5 7 8  Array before sorting: A Z B  Arrays after sorting: A B Z |

Example: Customized sorting of Object Array

|  |
| --- |
| public class ArraysCSO\_Practice {  public static void main(String[] args) {    String[] str = {"Z","C","A"};  System.out.print("Objects array before sorting: ");  for(String s : str)  System.out.print(" "+s);  System.out.println();    System.out.print("Object array after default sorting: ");  Arrays.sort(str);  for(String s : str)  System.out.print(" "+s);  System.out.println();    System.out.print("Object arrat after Customized sorting: ");  Arrays.sort(str,new myArrayCompar());  for(String s : str)  System.out.print(" "+s);    }  }  class myArrayCompar implements Comparator{  @Override  public int compare(Object arg0, Object arg1) {  String s1 = arg0.toString();  String s2 = arg1.toString();  return s2.compareTo(s1);  }    }  OUTPUT:  Objects array before sorting: Z C A  Object array after default sorting: A C Z  Object array after Customized sorting: Z C A |

Note:

We can sort primitive arrays only based on default natural sorting order whereas we can sort object arrays either based on default natural sorting order or based on customized sorting order.

1. Searching elements of array

Arrays class defines the following binary search methods.

* public static int binarySearch(primitive[] p, primitive target);
* public static int binarySearch(Object[] obj, Object target);
* public static int binarySearch(Object[] obj, Object target, Comparator c);

Note:

All rules of Arrays class binarySearch methods are exactly same as Collections class binarySearch methods.

Example:

|  |
| --- |
| public class ArraysBinarySearch {  public static void main(String[] args) {    String[] str = {"Z","A","R","C"};    System.out.print("Elements in Array before sorting: ");  for(String i : str)  System.out.print(" "+i);  System.out.println();    Arrays.sort(str);  System.out.print("Elements in Array after sorting: ");  for(String i : str)  System.out.print(" "+i);  System.out.println();  System.out.println("index of element 'C' in Array: "+Arrays.binarySearch(str, "C"));  System.out.println("Insetion point of Element 'S' in array: "+Arrays.binarySearch(str,"S"));  System.out.println();    System.out.print("Elements in Array After customized sorting: ");  Arrays.sort(str, new myArrayCSO());  for(String s : str)  System.out.print(" "+s);  System.out.println();  System.out.println("index of element 'C' in Array: "+Arrays.binarySearch(str, "C",new myArrayCSO()));  System.out.println("Insetion point of Element 'S' in array: "+Arrays.binarySearch(str,"S",new myArrayCSO()));  System.out.println();  }  }  class myArrayCSO implements Comparator{  @Override  public int compare(Object o1, Object o2) {    String s1 = o1.toString();  String s2 = o2.toString();  return s2.compareTo(s1);  }  }  OUTPUT:  Elements in Array before sorting: Z A R C  Elements in Array after sorting: A C R Z  index of element 'C' in Array: 1  Insetion point of Element 'S' in array: -4  Elements in Array After customized sorting: Z R C A  index of element 'C' in Array: 2  Insetion point of Element 'S' in array: -2 |

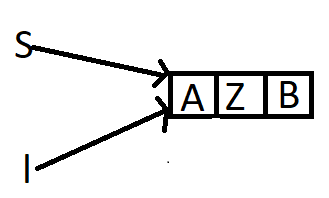
1. Conversion of Array to List

public static List asList(Object[] o)

Strictly speaking this method won’t create an independent List object for the existing Array we are getting List view.

String[] s = {“A”,”Z”,”B”};

List l = Arrays.asList(s);



1. By using Array reference if we perform any change automatically that change will be reflected to the List. Similarly by using the List refer if we perform any change that change will be reflected automatically to the array.

Ex:

S[o] = “K”;

l = {“K”,”Z”,”B”};

1. By using List reference, we cannot perform any operation which varies the size, otherwise we will get RuntimeException *UnsupportedOperationException*.

Ex:

l.add(“M”);

l.remove(1);

The above two operations are not supported and ‘*UnsupportedOperationException*’ exception is thrown.

l.set(1,”N”); replaces the element at index 1 with “N”

1. By using List reference we’re not allowed to replace with Heterogeneous objects otherwise we will get RuntimeException saying *ArrayStoreException.*

Ex:

l.set(1,new Integer(10)); // *ArrayStoreException* thrown

Example:

|  |
| --- |
| public class ArrayAsList\_Practice {  public static void main(String[] args) {    String[] str = {"Z","A","B"};    List l = Arrays.asList(str);  System.out.println(l); //[Z, A, B]    str[0] = "K";  System.out.println(l); //[K, A, B]    l.set(1, "R");  for(String s : str)  System.out.print(" "+s);  System.out.println();    //l.add("O"); /\* UnsupportedOperationException thrown \*/  //l.remove(1); /\* UnsupportedOperationException thrown \*/  //l.set(0, new Integer(10)); /\* ArrayStoreException thrown \*/  }  } |