**Q) How to represent a data in Java?**

1. Primitive types.
2. Primitive Arrays.
3. User Defined Objects.
4. User Defined Object Array.
5. Object Type Object Array.
6. Collections.

**1) Primitive Types**

We can represent only one element data and only one type of data

**Example**

Therefore we go for primitive Array

**2) Primitive Arrays**

1.) This array is fixed in size.

a) We can’t store more than the size

b) If we store fewer items than the specified size memory will be gone waste

2.) We can store only homogeneous elements

i.e. nothing but elements of Unique data type.

**Example: +**

Int[] enos = new int[1000];

double fees = new double[1000];

**3) User Defined Objects**

Here we store Heterogeneous or homogneous data items.

**Example:**

But if we want to define multiple employees information we have to go for user defined data Object Array

**4) User defined data Object Array**

Here we can represent multiple employees’ information

Example:

Employee[] employees= new Employee[10];

Disadvantages

* 1. But this array is fixed in size.
  2. We can’t store heterogeneous Objects in this array.

Thus we go for Object type Object Array

**5) Object Type Object Array**.

Here we can store heterogeneous or homogneous Objects also.

Example:

Disadvantages

1. But this array is fixed in size.
2. There is no predefined method support. ie to navigate, add, delete, insert etc…

Thus to Overcome these Problem we go for Collections

**6) Collections**

* Collections are growable in nature. i.e. based on run time requirement we can increase or decrease the size of collection. Collections can hold both homogeneous and heterogeneous data elements.
* For every requirement ready mate method support is available. Hence being a programmer we have to know how to use the predefined methods, But we are not responsible to define its functionality.

**Differences between arrays and collections**

**Arrays** **Collections**

1. Fixed in size 1. Growable
2. With respect to memory arrays 2. With respect to memory collections are recomm..

Are not recommended to use.

1. With respect to performance arrays 3. Less performance than arrays. But we use it.

Are best choice.

1. Can hold only homogeneous data 4. Both homogeneous & heterogeneous.

Elements.

1. No ready mate method support 5. For every requirement method support is there.
2. Arrays can hold both primitives and 6. Only objects.

Objects.

**Collection Framework**

It defines several classes and interfaces which can be used to represent a group of objects as single entity.

Java Collections Collection framework

C++ Container STL(Standard Template Library)

**9 key interfaces of collection framework**

1. **Collection**

* If we want represent a group of individual objects as single entity then we should go for **Collection** interface.
* It acts as root interface for entire collection framework.
* It defines several methods which can be applicable on any **Collection** implemented class objects.
* There is no concrete class which implement **Collection** interface directly.

**Q)**What is the difference between **Collection** and **Collections?**

**Collection** is an interface which can be used for representing a group of individual objects as a single entity. But **Collections** is a utility class present in java.util package to define several utility methods.

1. **List**

* It is the child interface of **Collection.** It can be used for representing a group of individual objects as a single entity where
  + insertion order is preserved
  + duplicate objects are allowed
* **ArrayList** and **LinkedLIst** classes implements List interface.

Collection(1.2)

List(I)(1.2)

ArrayList(c)(1.2) LinkedList(c)(1.2) Vector(c)(1.0)

Stack(c)(1.0) { legacy classes from old versions}

* Vector and Stack classes reengineered in 1.2 version to fit into new collection framework.

**3.Set**

* Child interface of Collection.
* It can be used for representing a group of individual objects as a single entity where
* Duplicate objects are not allowed
* Insertion order not preserved.
* **HashSet** (c)and L**inkedHashSet(c)** classes implements Set interface.

**4.SortedSet**

* It is the child interface of Set(I) it can representing a group of unique objects where all objects are inserted according to some sorting order.

**5.NavigableSet**

* It is the child interface of **SortedSet** it can be used to define several methods for navigation purposes.

Collection(I)(1.2)

Set(I)(1.2)

HashSet(C)(1.2) SortedSet(I)((1.2)

LinkedHashSet(c)(1.4) NavigableSet(I)(1.6)

TreeSet(c)(1.2)

1. **Queue (1.5)**

* Prior to processing if we want to represent a group of individual objects then we should go for Queue interface.
* It is the child interface of C**ollection** and introduced in 1.5 version.

Collection(I)

Queue(I)(1.5)

PriorityQueue(c) BlockingQueue(c)

PriorityBlockingQueue(c)

LinkedBlockingQueue(c)

* All the Queue related classes are from 1.5 version.
* All the above interface (Collection, List, Set, SortedSet, NaviagableSet and Queue) can be used to represent group of individual objects. If we want to represent a group of objects as key value pairs then we should go for **Map**(I).

1. **Map**

* It is not child interface of C**ollection** it can be used to represent a group of objects as key value pairs.

1. **SortedMap**

* It is child of **Map**, it can be used to represent a group of objects as key value pairs where all entries are inserted according to some sorting order.

1. **NavigableMap**

* Child of **SortedMap**(1.6) defines several methods for navigation purpose.

Map(I)(1.2)

SortedMap(I)(1.2)

NavigableMap(I)(1.6)

TreeMap(c)(1.2)

Map(I)(1.2)

Dictionary(c)(1.0)

HashMap(1.2)(c) IdentityHashMap(1.4)(c) WeakHashMap(c)(1.4) HashTable(c)(1.0)

LikedHashMap(1.4) Properties(c)(1.0)

Lexicographically

(Legacy classes)

**Q.) Which are the legacy classes/interfaces in collection framework.**

1. vector(c)

2.Stack(c)  
3.Hashtable(c)

4.Properties(c)

5.Dictionary(AS)

6.Enumeration(I)

**Collection(I)**

* This can be used for representing a group of individual objects. This interface defines the most common general methods which can be applicable on any Collection objects.

1. Boolean add(Object o)
2. Boolean addAll(Collection c)
3. Boolean remove(Object o)
4. Boolean removeAll(Collection c)
5. Void clear()
6. retainAll(Collection c)
7. Boolean contains(Object o)
8. Boolean containsAll(Collection c)
9. Int size()
10. Boolean isEmpty()
11. Object[] toArray()
12. Iterator iterator()

**List(I)**

* It is the child interface of collection. It can used to represent a group of individual objects where insertion order is preserved and duplicate objects are allowed.
* By using indexes we can preserve insertion order and we can differentiate duplicate objects.
* The interface adds position-oriented operations.
* The position-oriented operations include the ability to insert an element, get an element, as well as remove or change an element.

1. Boolean add(int index, Object o)
2. Boolean addAll(int index, Collection c)
3. Object remove(int index)
4. Int indexOf(Object obj)
   * Return index of first occurance
5. Int lastIndexOf(Object obj)
   * Return index of first occurance
6. ListIterator listIterator()
7. Object get(int index)
8. Object set(int index, Object new)
   * At index old object will be deleted and new object will be stored in that position.

**ArrayList(C)**

* The underlying data structure is resizable array or grow able array.
* Insertion order is preserved.
* Duplicate objects are allowed.
* Heterogeneous objects are allowed.
* **null** insertion is possible.

**Constructors**

1. ArrayList al = new ArrayList()
   * It creates an empty ArrayList object with default capacity ‘10’.
   * Once it reaches max capacity then a new array list object is created with new capacity

**newCapacity = (oldCapacity \* 3)/2 + 1**

1. ArrayList al=new ArrayList(int initialCapacity)
   * Creates an empty ArrayList object with the specified initialCpacity.
2. ArrayList al=new ArrayList(Collection c)
   * For inter conversion between Collection objects.

Example:

**import** java.util.ArrayList;

**public** **class** ArrayList1 {

**public** **static** **void** main(String[] params) {

ArrayList arrayList= **new** ArrayList();

arrayList.add("A");

arrayList.add(10);

arrayList.add("A");

arrayList.add(**null**);

System.*out*.println(arrayList);// [A, 10, A, null]

arrayList.remove(2);

System.*out*.println(arrayList);// [A, 10, null]

arrayList.add(2,"M");

arrayList.add("N");

System.*out*.println(arrayList);// [A, 10, M, null, N]

}

}

* Usually we can use **Collection** object for storing and transporting data across the network. To support this requirement every collection framework class implements **Serializable,** and **Clonable** interfaces.
* **ArrayList** and **Vector** implements **RandomAccess** interface so that we can access any element randomly with the same speed.

ArrayList list = new ArrayList();

sop(list instanceof Serializable); // true

sop(list instanceof Cloneable); // true

sop(list instanceof RandomAccess); // true

sop(list instanceof Collection); // true

sop(list instanceof List); // true

* **ArrayList** is best suitable, if our requirement is frequent retrieval operation.
* **ArrayList** is the worst choice if our requirement is frequent adding or removing elements in the based on the index.

Q.) How can get synchronized version of **ArrayList** class.

* **Collections** class defines the following method to get synchronized version of any List object.

p**ublic static List synchronizedList(List l)**

**Example:**

ArrayList al = new ArrayList();

List syscAl= Collections.synchronizedList(al);

* Similary we can get synchronized version of **Set** and **Map** objects by the following methods.

**public static Set synchronizedSet(Set set)**

**public static Map synchronizedMap(Map map)**

**public static SortedSet synchronizedSortedSet(SortedSet ss)**

**public static SortedMap synchronizedSortedMap(SortedMap sm)**

**public static Collection synchronizedCollection(Collection c)**

**LinkedList**

* Underlying data structure is double linkedlist.
* Insertion order is preserved.
* Duplicate objects are allowed.
* Heterogeneous objects are allowed
* **null** insertion is allowed.
* It is the Best choice if our requirement is frequent insertion or deletion or modification based on the position.
* Worst choice if frequent operation is retrival.
* **LinkedList** class implements **Serializable** an **Clonable** interfaces but not **RandomAccess**.

Constructor

1. LinkedList ll = new LinkedList();
   * It creates an empty LinkedList object.
2. LinkedList ll = new LinkedList(Collection c)
   * Create an equivalent LinkedList object for the passed Collection object.
   * This constructor is for inter conversion between Collection object.

* In general **LinkedList** class can be used for implementing **Stacks** and **Queues**. To support this type of requirement **LinkedList** contains the following six specific methods.

1. Void addFirst(Object o);
2. Void addLast(Object o);
3. Object removeFirst();
4. Object removeLast()
5. Object getFirst()
6. Object getLast();

* These methods only for LinkedList. These methods we don’t we use anywhere else other than this class.

**Example:**

**import** java.util.LinkedList;

**public** **class** LinkedList {

**public** **static** **void** main(String[] args) {

LinkedList linkedList = **new** LinkedList();

linkedList.add("rao");

linkedList.add(30);

linkedList.add(**null**);

linkedList.add("david");

linkedList.set(0, "software");

linkedList.add(0,"venky");

linkedList.removeLast();

linkedList.addFirst("ccc");

System.*out*.println(linkedList);//[ccc, venky, software, 30, null]

}

}

**Vector**

* Underlying data structure is Resizable array or grow able array.
* Insertion order is preserved.
* Duplicate objects are allowed.
* Heterogeneous Objects are allowed.
* null insertion is possible.
* **Vector** class implements **Serializable ,Clonable, RandomAccess, List, Collection** interfaces.
* Best suitable if frequent operation is retrieval
* Worst choice if frequent operation is position based insertion or deletion or updation.
* Every method is synchronized hence **Vector** object by default thread safe.

**Important methods of Vector**

* For adding objects
  + Void add(Object obj) //from Collection
  + Void add(int index, Object o) // form List
  + Void addElement(Object obj) // from Vector
* For removing elements.
  + Object remove(Object o) // Collection
  + Object remove(int index) // List
  + Object removeElement(Object o) // Vector
  + Object removeElementAt(int index) //Vector
  + Object clear() //Collection
  + removeAllElements() //Vector
* For retrieving Objects
  + Object get(int index) //List
  + Object elementAt(int index) //Vector
  + Object firstElement()//Vector
  + Object lastElement() //Vector
* Other common methods
  + Int capacity()//Vector
  + Int size() //Collection
  + Enumeration elements() //Vector
* Constructors
  + Vector v = new Vector()
    - Create new Vector object with default initial capacity 10. Whenever reaches max capacity then a new vector object will be created with
      * New capacity= 2\*currentCapacity;
* Vector v = new Vector(int initialCapacity);
* Vector v = new Vector(int initialCapacity, int incrementalCapacity)
* Vector v = new Vector(Collection c)

**Example:**

**package** com.lit.collection;

**import** java.util.Vector;

**public** **class** Vector1 {

**public** **static** **void** main(String[] args) {

Vector vector = **new** Vector();

System.*out*.println(vector.capacity());//10

**for** (**int** i = 0; i <= 10; i++) {

vector.add(i);

}

System.*out*.println(vector.capacity());//20

vector.addElement("A");

System.*out*.println(vector.capacity());//20

System.*out*.println(vector);// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]

}

}

\*\*\*Difference between ArrayList and Vector\*\*\*

|  |  |
| --- | --- |
| ArrayLlist | Vector |
| 1. No method is synchronized. 2. ArrayList is not Thread safe. 3. Performance is high. 4. It is introduced in 1.2 version and it is   Non legacy. | 1. Every method is synchronized 2. Vector is thread safe. 3. Performance is low. 4. It is introduced in 1.0 version and it   Is legacy. |

**Stack**

* It is the child of Vector , contains only one constructor

i.e. Stack stack = new Stack();

**Important method of stack**

* Object push(Object obj)
  + Insert an object into the stack at the end
* Object pop();
  + To remove and returns top element of the stack.
* Object peek();
  + To return top of the stack without removal
* boolean empty()
  + Returns true if the stack is empty otherwise returns false
* int search(Object obj)
  + Returns offset of the element if it is

available. Otherwise it returns -1.

* Stack follows LIFO(Last In First Out)

**Example:**

**package** com.lit.collection;

**import** java.util.Stack;

**public** **class** Stack1 {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Stack stack = **new** Stack();

stack.push("A");

stack.push("B");

stack.push("C");

System.*out*.println(stack);// [A, B, C]

System.*out*.println(stack.search("A"));//3

System.*out*.println(stack.search("Z"));//-1

}

}

**Types of Cursors**

* If we want to retrieve the objects of a Collection one by one we should go for cursors. There are three types of Cursors available in java.
  + Enumeration
  + Iterator
  + ListIterator

**Enumeration**:

* It has introduced in 1.0 version and hence it is considered as legacy interface. We can get **Enumeration** object by using **elements()** method of **Vector**.
  + **Enumeration e= vector.elements()**
* Enumeration interface defines the following two methods.
  + public boolean hasMoreElements()
  + public Object nextElement()

Example:

**package** com.lit.collection;

**import** java.util.Enumeration;

**import** java.util.Vector;

**public** **class** Enumeration1 {

**public** **static** **void** main(String[] args) {

Vector vector = **new** Vector();

**for** (**int** i = 0; i <= 10; i++) {

vector.add(i);

}

System.*out*.println(vector);

Enumeration enumeration=vector.elements();

**while**(enumeration.hasMoreElements()){

Integer i=(Integer)enumeration.nextElement();

**if**(i%2==0){

System.*out*.println(i);

}

}

// Cursors won’t allow any other user to add or remove or update //elements of the Collection object.

//Using our own logic

// prob is this for loop came in 1.6 version b4 we used to Cursors

//some threading probs is there for enhanced for loop

**for** (Object object : vector) {

Integer i=(Integer)object;

**if**(i%2==0){

System.*out*.println(i);

}

}

System.*out*.println(vector);

}

}

Output:

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

0

2

4

6

8

10

0

2

4

6

8

10

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

Limitations of Enumeration

* Enumeration concept is applicable only for legacy classes and hence it is not a universal cursor.
* While performing iteration we are not allowed to remove useless objects. i.e. Through enumeration we will get only read access.
* To overcome these limitation sun people introduced **Iterator** concept in 1.2 versions.

**NOTE:** The implementation class of **Enumeration** is anonymous class.

**Iterator**

* This interface defines the following three methods
  + public boolean hasNext();
  + public Object next();
  + pulbic void remove();
* We can get **Iterator** object by using **iterator()** method of **Collection** interface.
  + Iterator itr= collection.iterator();
* We can get **Iterator** from any **Collection** object. Hence **Iterator** is called as universal cursor.
* Using remove() method of Iterator even we can modify the collection object.

Example:

**package** com.lit.collection;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** Iterator1 {

**public** **static** **void** main(String[] args) {

ArrayList arrayList = **new** ArrayList();

**for** (**int** i = 0; i <= 10; i++) {

arrayList.add(i);

}

System.*out*.println(arrayList);

Iterator iterator = arrayList.iterator();

**while** (iterator.hasNext()) {

Integer integer = (Integer) iterator.next();

**if** (integer % 2 == 0) {

System.*out*.println(integer);

} **else** {

iterator.remove();

}

}

System.*out*.println(arrayList);

}

}

Output:

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

0

2

4

6

8

10

[0, 2, 4, 6, 8, 10]

* Generics importance explanation.

Limitations of Iterator

* It is a single directional cursor. i.e. We can always move forward direction only. There is no chance of moving towards backward direction.
* While performing iteration we can perform only read and removal operations, there is no chance of performing replacement and addition of new objects. To overcome these limitations sun people introduced ListIterator concept.

**NOTE:** The implementation class of **Iterator** is **Itr**, which is an inner class in **AbstractList**.

* iterator() is defined in java.lang.Iterable. Iterable is the child of Collection interface. So Iterable is the super interface to all collections classes.

**ListIterator:**

* It is the child interface of Iterator.
* It is a bidirectional cursor. i.e we can move either forward or backward direction.
* While performing iteration we can replace exiting object with new object.
* We can add new Object also.
* We can get ListIterator by using listIterator () method.
  + - ListIterator listItr = list.listIterator();
    - ‘list’ is any **java.util.List** implemented class.

Mehods of ListIterator:

* public boolean hasNext();
* public Object next();
* public int nextIndex();
  + - If there is no next element it returns size of the list but not -1.
* public boolean hasPrevious();
* public Object previous();
* public int previousIndex();
  + - If there is no previous element then this method returns -1.
* public void remove();
* public void add(Object newObj);
* public void set(Object newObj);

Example:

**package** com.lit.collection;

**import** java.util.LinkedList;

**import** java.util.ListIterator;

**public** **class** ListIterator1 {

**public** **static** **void** main(String[] args) {

LinkedList linkedList= **new** LinkedList();

linkedList.add("balayya");

linkedList.add("chiru");

linkedList.add("venki");

linkedList.add("nag");

System.*out*.println(linkedList);

ListIterator listIterator= linkedList.listIterator();

**while**(listIterator.hasNext()){

String s =(String)listIterator.next();

**if**(s.equals("venki")){

listIterator.remove();

}

/\*if(s.equals("nag")){

listIterator.add("chaitu");

}

\*/

/\*if(s.equals("chiru")){

listIterator.set("charan");

}\*/

}

}

}

Output:

[balayya, venki, venki, nag]

[balayya, nag]

* It is most powerful cursor. But it is applicable for only List implemented class objects.

**NOTE :** The implementation class of **ListIterator** is **ListItr**, which is an inner class in **AbstractList**.

|  |  |  |  |
| --- | --- | --- | --- |
| Properties | Enumeration | Iterator | ListIterator |
| 1. Is it Legacy 2. It is Applicable for 3. Movement 4. How to get 5. Accessibility 6. Methods 7. Implementation class | Yes  Only legacy classes  Single direction(forward)  By using elements()  Only read access  2 methods  hasMoreElements()  nextElement()  Anonymous class | No  Any Collection Object  Single direction(forward)  Iterator()  Read and remove  3 methods hasNext()  Next()  Remove()  Itr | No  Any List object  Bidirectional  listIterator()  read , remove, replace, add  9 methods.  hasNext();  next();  remove();  previous();  hasPrevious();  add()  set()  previousIndex();  nextIndex();  ListItr |

**Set**

* It is the child interface of **Collection**.
* It can be used to represent a group of individual objects where
  + Duplicate objects are not allowed.
  + Insertion order is not preserved.
* Set interface doesn’t contain any new method we have to use only **Collection** interface methods.

**HashSet**

* Underlying data structure is **hashtable**.
* Duplicates objects are not allowed. If we are trying to insert any duplicate object we won’t get any compile time or runtime errors. But **add()** method simply returns **false**.
* Insertion order is not preserved because it is based on **hashCode()** of the objects.
* Heterogeneous objects are allowed. But not advisable.
* It implements **Serializable** and **Clonable** interface.
* Itis the best suitable to perform search operations.
* **null** insertion is possible. But not advisable.

Constructor

* HashSet hs= new HashSet()
  + Creates an empty HashSet object with default **initialCapacity** **16** and default **fill-ratio** **0.75**(**load factor**).
* HashSet hs = new HashSet(int initialCapacity)
  + Creates an empty HashSet object with specified

**initialCapacity** and default **load factor**(0.75)

* HashSet hs = new

HashSet(int initialCapacity, float fillRatio)

* + **fill ratio** can be **between 0 to 1**.
* HashSet hs = new HashSet(Collection c)
* The **load factor** is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
* When the number of entries in the hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the hash method.
* Generally, the default load factor (.75) offers a good tradeoff between time and space costs

Example:

**package** com.lit.collection;

**import** java.util.HashSet;

**public** **class** HashSet1 {

**public** **static** **void** main(String[] args) {

HashSet hashSet = **new** HashSet();

hashSet.add("A");

hashSet.add("B");

hashSet.add("C");

hashSet.add("D");

hashSet.add("Z");

hashSet.add(**null**);

hashSet.add(10);

System.*out*.println(hashSet);// [null, D, A, B, C, 10, Z]

System.*out*.println(hashSet.add("Z"));//false

System.*out*.println(hashSet);// [null, D, A, B, C, 10, Z]

}

}

**LinkedHashSet**

* It is the child class of **HashSet**.
* It is exactly same as **HashSet** except the following differences.

|  |  |
| --- | --- |
| **HashSet** | **LinkedHashSet** |
| 1. Insertion is not preserved 2. Underlying data structure is Hashtable 3. Introduced in 1.2 version | 1. Insertion is preserved  2. combination of LinkedList and HashTable  3. Introduced in 1.4 version |

* In the above program if we replacing **HashSet** with **LinkedHashSet**, elements will be inserted in “**insertion order”**.

Output:

[A, B, C, D, Z, null, 10]

false

[A, B, C, D, Z, null, 10]

* **LinkedHashSet** and **LinkedHashMap** classes are best suitable to implement cashing applications where duplicates are not allowed.
* Insertion oreder is preserved.

**SortedSet**

* It is the child interface of **Set.**
* If we want to represent a group of unique objects according to some sorting order then we should go for **SortedSet** Interface.
* **SortedSet** interface defines the following more specific methods.
  + Object first()
  + Object last()
  + **SortedSet** headset(Object obj)
    - Returns the **SortedSet** whose elements are less than Object obj.
  + **SortedSet** tailSet(Object obj)
    - Returns the **SortedSet** whose elements are greater than or equal to object obj.
  + SortedSet subset(Object obj1, Object obj2)
    - Returns the SortedSet whose elements are greater than or equal to obj1 but less than obj2.
  + Comparator comparator();
    - Returns **Comparator** object which describe underlying sorting technique.
    - If we are using natural sorting order then we will get null.

Example: {101,103,104,105,107,108}

1. first() ; //101
2. last() //108
3. headSet(105); //101,103,104
4. tailSet(105); //105,107,108
5. subset(103,107); //103,104, 105
6. comparator(); // null

**TreeSet:**

* It is the child of **SortedSet**
* The underlying data structure is **balanced tree**.
* Duplicate objects are not allowed
* Insertion order is not preserved. Because all the elements are inserting based on some sorting order.
* Heterogeneous objects are not allowed. Violation leads to runtime Exception saying **ClassCastException**.

Constructors

* TreeSet ts = new TreeSet();
  + Creates an empty Treeset object where sorting order is default natural sorting order.
  + For numbers default natural sorting order is increasing order.
  + For Strings default natural sorting order is it is alphabetical order.
* TreeSet ts = new TreeSet(Comparator c);
  + Creates an empty TreeSet object where the sorting

order is specified by comparator object. i.e. this constructor is for customized sorting.

* TreeSet ts = new TreeSet(Collection c);
* TreeSet ts = new TreeSet(SortedSet ss)

Example:

**package** com.lit.collection;

**import** java.util.TreeSet;

**public** **class** TreeSet1 {

**public** **static** **void** main(String[] args) {

TreeSet treeSet= **new** TreeSet();

treeSet.add("A");

treeSet.add("a");

treeSet.add("B");

treeSet.add("Z");

treeSet.add("L");

//treeSet.add(null); ClassCastException

// heterogeneous elements are not allowed in TreeSet and TreeMap

//treeSet.add(10); NullPointerException

System.*out*.println(treeSet);// [A, B, L, Z, a]

}

}

null Acceptence:

* 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 java.lang.ClassCastException**.**
* For non empty **TreeSet** if we are trying to insert(even first time) **null** we get java.lang.**NullPointerException.**

Example:

**package** com.lit.collection;

**import** java.util.TreeSet;

**public** **class** TreeSet2 {

**public** **static** **void** main(String[] args) {

TreeSet treeSet= **new** TreeSet();

treeSet.add(**new** StringBuffer("A"));

treeSet.add(**new** StringBuffer("B"));

treeSet.add(**new** StringBuffer("Z"));

treeSet.add(**new** StringBuffer("L"));

System.*out*.println(treeSet);

}

}

Output: RuntimeException

java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable

* **If** we are depending on natural sorting order compulsory

Object should be homogeneous and Comparable. Otherwise

We will get runtime exception saying ClassCastException.

* String objects and all wrapper class objects are by default comparable,

But User defined objects(Student) are not comparable. Hence we are

getting runtime Exception in the above case.

**java.lang.Comparable**

* This interfacedeclares only one method **compareTo(Object obj)**

**Methods**

* public int compareTo(Object obj)
  + obj1.compareTo(obje2)
    - obj1 is new object and obj2 is existing object
    - Return –ve if obj1 has to come before obj2
    - Return +ve if obj1 has to come after obje2
    - Return 0 if obj1 and obj2 are equal.
* If we are depending on natural sorting order then JVM internally calls compareTo() method to arrange elements according to sorting order.
* TreeSet ts = new TreeSet();
  + ts.add(“A”); //No comparison
  + ts.add(“Z”); // “Z”.compareTo(“A”);
    - Returns +ve , hence “Z” will be placed after “A”
    - Then the result is [A,Z]
  + ts.add(“L”); // “L”.compareTo(“A”);
    - Returns +ve ,hence “L” will be placed after “A”
    - now set has [A,L]

//Then “L”.compareTo(“Z”)

* + - Returns –ve, hence “L” will be placed before “Z”
    - Result [A,L,Z]
* If we want to insert all objects according to natural sorting order to we will go for **Comparable** interface.
* Once we define some sorting technique using **Comparable** interface, that cannot be changed.
* If we want to insert all objects according to our own customized sorting order. Then we should go for **Comparator** interface.

**java.lang.Comparator interface:**

* This is for defining customized sorting.
* In this interface contains the following two methods
  + public int **compare**(Object obj1, Object obj2)
    - Obj1 is new object and obj2 is existing object
    - Returns –ve if obj1 has to come before obj2
    - Returns +ve if obj1 has to come after obj2
    - Returns 0 if obj1 equal to obj2
  + Public boolean equals(Object obj);
* Whenever we are implementing **Comparator** we should provide implementation for compare() method. Implementing equals() method is optional becoz it is already available from Object class through inheritance.

Q.) Write a program to insert Integer object into TreeSet where the sorting order is descending order.

**package** com.lit.collection.comparator;

**import** java.util.Comparator;

**import** java.util.TreeSet;

**public** **class** Comparator1 {

**public** **static** **void** main(String[] args) {

TreeSet treeSet = **new** TreeSet(**new** MyComparator());

treeSet.add(10);

treeSet.add(0);

treeSet.add(15);

treeSet.add(5);

treeSet.add(20);

System.*out*.println(treeSet);

}

}

**class** MyComparator **implements** Comparator {

**public** **int** compare(Object o1, Object o2) {

Integer i1 = (Integer) o1;

Integer i2 = (Integer) o2;

**if** (i1 > i2) {

**return** -1;

} **else** **if** (i1 < i2) {

**return** +1000;

} **else** {

**return** 0;

}

}

}

NOTE:

* if we are not passing our own comparator to the TreeSet constructor then JVM internally calls compareTo() method which is meant for default natural sorting order.
* if we are passing comparator to the TreeSet Constructor object then JVM calls

compare() method which is meant for our own customised sorting order.

Inputs for TreeSet [10, 5, 20, 15, 0]

* If we are implementing compare() method as follows the corresponding outputs are
* return new.compareTo(old); Ascending order [0, 5, 10, 15, 20]
* return - new.compareTo(old); Descending order [20, 15, 10, 5, 0]
* return old.compareTo(new); Descending order [20, 15, 10, 5, 0]
* return - old.compareTo(new); Ascending order [0, 5, 10, 15, 20]
* return -1; Reverse of insertion order [0, 15, 20, 5, 10]
* return +1; Insertion order [10, 5, 20, 15, 0]
* return 0; Only first inserted element will be present and all

remaining considered as duplicate[10]

Q.) Write a program to insert String objects into the TreeSet where the sorting order is reverse of dictionary order.

package com.lit.collection.comparator;

import java.util.Comparator;

import java.util.TreeSet;

public class Comparator3 {

public static void main(String[] args) {

TreeSet treeSet = new TreeSet(new MyComparator3());

treeSet.add("B");

treeSet.add("Z");

treeSet.add("L");

treeSet.add("C");

treeSet.add("M");

System.out.println(treeSet);

}

}

class MyComparator3 implements Comparator {

public int compare(Object o1, Object o2) {

String string1= (String)o1;

String string2= (String)o2;

// return string1.compareTo(string2); alphabetical order. [B, C, L, M, Z]

// return -string1.compareTo(string2); reverse alphabetical order. [Z, M, L, C, B]

return 0;

}

}

Q.) Write a program to insert StringBuffer objects into the TreeSet where the sorting order is alphabetical order.

**package com.lit.collection.comparator;**

**import java.util.Comparator;**

**import java.util.TreeSet;**

**public class Comparator4 {**

**public static void main(String[] args) {**

**TreeSet treeSet = new TreeSet(new MyComparator4());**

**treeSet.add(new StringBuffer("A"));**

**treeSet.add(new StringBuffer("B"));**

**treeSet.add(new StringBuffer("Z"));**

**treeSet.add(new StringBuffer("L"));**

**System.out.println(treeSet);**

**}**

**}**

**class MyComparator4 implements Comparator {**

**public int compare(Object o1, Object o2) {**

**String string1= o1.toString();**

**String string2= o2.toString();**

**// return string1.compareTo(string2); alphabetical order. [A, B, L, Z]**

**// return -string1.compareTo(string2); reverse alphabetical order. [Z, L, B, A]**

**return 0;**

**}**

**}**

* Whenever we are defining our own sorting order then the objects

needs not be comparable and need not be homogeneous.

**Q.)**Write a program to insert and StringBuffer into the TreeSet where the sorting order is increasing length order. If two objects having the same length then consider their alphabetical order.

package com.lit.collection.comparator;

import java.util.Comparator;

import java.util.TreeSet;

public class Comparator5 {

public static void main(String[] args) {

TreeSet treeSet = new TreeSet(new MyComparator5());

treeSet.add("A");

treeSet.add(new StringBuffer("ABC"));

treeSet.add(new StringBuffer("AA"));

treeSet.add("XX");

treeSet.add("ABCD");

treeSet.add("A");

System.out.println(treeSet);

}

}

class MyComparator5 implements Comparator {

public int compare(Object o1, Object o2) {

String string1= o1.toString();

String string2= o2.toString();

int l1= string1.length();

int l2 = string2.length();

if(l1 > l2 ){

return +1;

}else if(l1<l2){

return -1;

}else{

return string1.compareTo(string2);

}

//return 0; when we don't want same length objects

}

}

**Output:**

[A, AA, XX, ABC, ABCD]

**Note:**

**1.** For predefined C**omparable** class objects like String, default natural sorting order is already available. if we don’t want that then we can go for customised sorting order by implementing **Comparator** interface.

**2**: For the predefined non C**omparable** objects like the **StringBuffer** default natural sorting order is not available. Even we can’t provide. We can define only our required sorting by implementing **Comparator**.

**3**: For our own customised classes(user defined objects) we can go for either **Comparable** or **Comparator**.

**Conclusion:** To define natural sorting order we have to use **Comparable** interface and if we are not satisfied with natural sorting order, we can go for **Comparator** to define customised sorting.

**Example:**

**package** com.lit.collection.mycomparator;

**import** java.util.Comparator;

**import** java.util.TreeSet;

**public** **class** CompComp {

**public** **static** **void** main(String[] args) {

TreeSet comparableTreeSet = **new** TreeSet();

comparableTreeSet.add(**new** Employee(10));

comparableTreeSet.add(**new** Employee(5));

comparableTreeSet.add(**new** Employee(6));

comparableTreeSet.add(**new** Employee(12));

System.*out*.println(comparableTreeSet);

// [emp-5, emp-6, emp-10, emp-12]

TreeSet comparatorTreeset = **new** TreeSet(**new** EmpComparator());

comparatorTreeset.add(**new** Employee(10));

comparatorTreeset.add(**new** Employee(5));

comparatorTreeset.add(**new** Employee(6));

comparatorTreeset.add(**new** Employee(12));

System.*out*.println(comparatorTreeset);

// [emp-12, emp-10, emp-6, emp-5]

}

}

**class** Employee **implements** Comparable {

**int** eno;

**public** Employee(**int** eno) {

**this**.eno = eno;

}

**public** String toString() {

**return** "emp-" + eno;

}

**public** **int** compareTo(Object o) {

Integer eno1 = **this**.eno;

Integer eno2 = ((Employee) o).eno;

/\*

\* if (eno1 < eno2) { return -1; } else if (eno1 > eno2) { return +1; }

\* else { return 0; }

\*/

**return** eno1.compareTo(eno2);

}

}

**class** EmpComparator **implements** Comparator{

@Override

**public** **int** compare(Object o1, Object o2) {

Integer i1= ((Employee)o1).eno;

Integer i2= ((Employee)o2).eno;

**return** i2.compareTo(i1);

// return 0;

}

}

**Comparison between Comparator and Comparable**

**Comparable Comparator**

1. **Present in java.lang 1. Java.uitl**
2. **Meant for natural sorting order 2. Meant for customised sorting**

**Order**

1. **Contains only compareTo() 3. compare() and equals()**
2. **String, Wrapper classes already 4. No predefined class**

**Implemented Comparable interface. Implemented Comparator.**

**Comparison between Set implemented classes.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| 1. **Underling data structure** 2. **Duplicates** 3. **Insertion order** 4. **Sorting order** 5. **Hetrogeneous objects** | **Hashtable**  not allowed  Not preserved    **not preserved**  **allowed** | **Hashtable + linkedList**  not allowed  **Preserved**  **Not preserved**  **allowed** | **Balanced Tree**  not allowed  **Not preserved**  **Preserved**  **Not allowed.** |

Note: if we are depending on natural sorting order in TreeSet then the objects should be homogeneous and Comparable otherwise we will get ClassCastException.

* If we are defining our own sorting order by Comparator then the objects need not be homeogeneous and need not be Comparable.

**Map**:

* If we want to represent a group of objects as key value pairs then we should go for Map interface.

Example:

Key value

101 sita

102 rama

103 lakshmana

104 oormila

* Each key value pair is considered as Entry
* Both key and values are objects.
* Key duplication is not allowed, but value can be duplicated.

**methods of Map interface**

1. Object put(Object key, Object value);
   1. If the specified key is not already available then this

method will add a new entry to the Map and returns null.

* 1. If the specified key is already available then the old

value replaced with new value and old value will be returned.

1. Void putAll(Map map)
2. Object get(Object key)
   1. Returns the value associated with the specified key.
   2. If key is not available then we will get null.
3. Object remove(Object key)
   1. Removes the entry associated with the specified

key and returns the corresponding value.

1. Boolean containsKey(Object key)
2. Boolean containsValue(Object value)
3. Boolean isEmpty()
4. Int size()
5. Void clear();
   1. Removes all the entries in the Map
6. Set keyset()
7. Collection values()
8. Set entrySet()
   1. 10, 11, 12 methods are called Collection views of the Map.

Entry interface

* A Map is nothing but a set of entries.
* Without existing Map object there is no chance of existing

Entry Object. Entry is an inner interface of Map interface

Interface Map{

Interface Entry{

Object getKey();

Object getValue();

Object setValue(Object newObject);

}

}

**HashMap**

* The underlying data structure is **Hash table**.
* Duplicate keys are not allowed, but values can be duplicated.
* Insertion order is not preserved. Because it depends up on hash code of the keys.
* Heterogeneous objects are allowed for both keys and values.
* **“null”** is allowed for both key and value.

**Constructors**

1. HashMap()
   1. Creates an empty HashMap object with default initial

capacity 16, and default fill-ratio 0.75.

1. HashMap( int initialCapacity)
2. HashMap(int initialCapacity, float fillRatio)
3. HashMap(Map map);

Example:

**package** com.lit.collection.map;

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Set;

**public** **class** HashMap1 {

**public** **static** **void** main(String[] args) {

HashMap hashMap = **new** HashMap();

hashMap.put("chiru", 100);

hashMap.put("balayya", 200);

hashMap.put("nag", 300);

hashMap.put("venki", 400);

System.*out*.println(hashMap);

// { chiru=100, balayya=200, nag=300, venki=400 }

hashMap.put("chiru", 500);

System.*out*.println(hashMap);

// {chiru=500, balayya=200, nag=300, venki=400}

Set set = hashMap.keySet();

System.*out*.println(set);

// [chiru, balayya, nag, venki]

Collection collection = hashMap.values();

System.*out*.println(collection);

// [500, 200, 300, 400]

Set entrySet = hashMap.entrySet();

Iterator entrySetItr = entrySet.iterator();

**while** (entrySetItr.hasNext()) {

Map.Entry entry = (Map.Entry) entrySetItr.next();

System.*out*.println(entry.getKey() + "......" + entry.getValue());

**if** (entry.getKey().equals("nag")) {

entry.setValue(600);

}

}

System.*out*.println(hashMap);

}

}

o/p:

* {chiru=100, balayya=200, nag=300, venki=400}
* {chiru=500, balayya=200, nag=300, venki=400}
* [chiru, balayya, nag, venki]
* [500, 200, 300, 400]
* chiru......500
* balayya......200
* nag......300
* venki......400
* {chiru=500, balayya=200, nag=600, venki=400}

**LinkedHashMap**:

* It is exactly same as HashMap except the following differences.

HashMap LinkedHashMap

1. Underlying data structure Hash table 1. Hash table + Linked List
2. Insertion is not preserved 2. Insertion order is preserved.
3. 1.2 version. 3. 1.4 version.

* In the above program if we replace HashMap with LinkedHashMap the following o/p we get.
* {chiru=100, balayya=200, nag=300, venki=400}
* {chiru=500, balayya=200, nag=300, venki=400}
* [chiru, balayya, nag, venki]
* [500, 200, 300, 400]
* chiru......500
* balayya......200
* nag......300
* venki......400
* {chiru=500, balayya=200, nag=600, venki=400}
* i.e. insertion order is preserved.

Note: LinkedHashSet and LinkedHashMap are best suitable to implement Cache applications

**IdentityHashMap:**

* It is exactly same as **HashMap** except the following difference.
  + In the case of **HashMap** to identify duplicate keys JVM always uses **equals()** method which is mostly meant for content comparison.
  + But in the case of **IndentityHashMap** JVM uses **‘==’** operator to identify duplicate keys which is always meant for reference comparison.

Example:

**package** com.lit.collection.map;

**import** java.util.HashMap;

**public** **class** IdentityHashMap1 {

**public** **static** **void** main(String[] args) {

HashMap hashMap = **new** HashMap();

Integer i1 = **new** Integer(10);

Integer i2 = **new** Integer(10);

hashMap.put(i1, "pavan");

hashMap.put(i2, "kalyan");

System.*out*.println(hashMap);// {10=kalyan}

}

}

* **If we are replacing HashMap with IdentityHashMap then i1 and i2 are not duplicate keys, because i1==i2 returns false.**
* In this case both entries will be stored in the Map and the output is
  + {10=kalyan, 10=pavan}

**WeakHashMap**

* It is exactly same as **HashMap** except the following difference.
  + In the case of **HashMap** an object is not eligible for

Garbage collection if it is associated with **HashMap** even

though it doesn’t have any external references. i.e. **HashMap** dominates garbage collector.

* + But in the case of **WeakHashMap** an object is eligible for

Garbage Collector if it doesn’t contain any external

references even it is associated with **WeakHashMap**. i.e

Garbage Collector dominates **WeakHashMap**.

Example:

**package** com.lit.collection.map;

**import** java.util.HashMap;

**import** java.util.WeakHashMap;

**public** **class** WeakHashMap1 {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

HashMap hashMap = new HashMap();

Temp temp = **new** Temp();

hashMap.put(temp, "sekhar");

System.*out*.println(hashMap);

temp = **null**;

System.*gc*();

Thread.*sleep*(5000);

System.*out*.println(hashMap);

}

}

**class** Temp {

**public** String toString() {

**return** "temp";

}

**public** **void** finalize() {

System.*out*.println("finalize method is called");

}

}

Output:  
{temp=sekhar}

{temp=sekhar}

* If we are replacing HashMap with WeakHashMap then the output is WeakHashMap.
* Then the output is .

{temp=sekhar}

finalize method is called

{}

**SortedMap**

* If we want to represent a group of entries according to some sorting order of keys then we should go for SortedMap.
* This interface defines the following six specific methods.

1. Object firstKey();
2. Object lastKey();
3. SortedMap headMap(Object key);
4. SortedMap tailMap(Object key);
5. SortedMap subMap(Object key1, Object key2);
6. Comparator comparator();

**Example:**

101 E

102 D

103 G

104 A

105 K

* firstKey();
  + 101
* lastKey();
  + 105
* headMap(104);
  + {101 = E, 102 = D, 103= G}
* tailMap(104);
  + {104=E, 105=G}
* subMap(102,104);
  + {102=D, 103=G}
* comparator();
  + null

**TreeMap**

* The underlying data structure is **RED Black Tree.**
* Insertion is not preserved.
* Duplicate objects are not allowed for keys. But the values can be duplicated.
* If we are depending on natural sorting order the keys

should be homogeneous and **Comparable** otherwise

we will get **ClassCastException**.

* If we defining our own sorting then the keys need not be **Comparable**.
* For the empty **TreeMap** first time **null** key is allowed. But after inserting that if we are trying to insert any other entry we will get **NullPointerException**.
* For non empty **TreeMap** if we are trying to insert an

entry with **null** key we get **NullPointerException**

* There are no restrictions on null values.

**Constructors:**

1. TreeMap()
   1. Creates an empty **TreeMap** where the sorting

order is natural sorting order.

1. TreeMap(Comparator c);
   1. **Cr**eates an empty TreeMap where the sorting

order is described by **Comparator** object.

1. TreeMap(Map map)
2. TreeMap(SortedMap map)

**Example:**

**package** com.lit.collection.map;

**import** java.util.TreeMap;

**public** **class** TreeMap1 {

**public** **static** **void** main(String[] args) {

TreeMap treeMap = **new** TreeMap();

treeMap.put(100, "ZZZ");

treeMap.put(103, "YYY");

treeMap.put(101, "XXX");

treeMap.put(109, 106);

// treeMap.put("FFF", "AA"); keys should be homogeneous

// ClassCastException

// treeMap.put(null, 108); keys should be homogeneous

// NullPointerException

treeMap.put(104, **null**);

System.*out*.println(treeMap);

//{100=ZZZ, 101=**XXX**, 103=YYY, 104=null, 109=106}

}

}

Sorting according to Comparator

**package** com.lit.collection.map;

**import** java.util.Comparator;

**import** java.util.TreeMap;

**public** **class** TreeMap2 {

**public** **static** **void** main(String[] args) {

TreeMap treeMap = **new** TreeMap(**new** MyComparator());

treeMap.put(100, "ZZZ");

treeMap.put(103, "YYY");

treeMap.put(101, "XXX");

treeMap.put(109, 106);

// treeMap.put("FFF", "AA"); keys should be homogeneous

// treeMap.put(null, 108); keys should be homogeneous

treeMap.put(104, **null**);

System.*out*.println(treeMap);

// {109=106, 104=null, 103=YYY, 101=**XXX**, 100=ZZZ}

}

}

**class** MyComparator **implements** Comparator {

@Override

**public** **int** compare(Object o1, Object o2) {

Integer i1 = (Integer) o1;

Integer i2 = (Integer) o2;

**return** i2.compareTo(i1);

}

}

**HashTable**

* The underlying data structure is **hash table**
* Heterogeneous objects are allowed for both keys and values.
* Duplicate keys are not allowed but values can be duplicated.
* Insertion order is not preserved it is based on **hashCode()** of the keys.
* **null** is not allowed for both key and value. Violation leads **NullPointerException.**
* All methods are **synchronized** hence **HashTable** object is always Thread safe.

Constructors:

1. HashTable();
   1. Creates an empty **HashTable** with default initial capacity 7, and fill-ratio 0.75.
2. HashTable(int initialCapacity)
3. HashTable (int initialCapacity, float fillRatio)
4. HashTable (Map map);

Example:

**package** com.lit.collection.map;

**import** java.util.Hashtable;

**public** **class** HashTable1 {

**public** **static** **void** main(String[] args) {

Hashtable hashtable = **new** Hashtable();

hashtable.put(**new** Temps(5), "A");

hashtable.put(**new** Temps(2), "A");

hashtable.put(**new** Temps(6), "A");

hashtable.put(**new** Temps(15), "A");

hashtable.put(**new** Temps(23), "A");

hashtable.put(**new** Temps(16), "A");

//hashtable.put("sekhar", null); NullPointerException

System.*out*.println(hashtable);

//{6=A, 16=A, 5=A, 15=A, 2=A, 23=A}

}

}

**class** Temps {

**int** i;

Temps(**int** i) {

**this**.i = i;

}

**public** String toString(){

**return** ""+i;

}

**public** **int** hashCode(){

**return** i;

}

}

* If we are taking hashCode method as follows
  + Public int hashCode(){ return i%8; }
    - Output={23=A, 15=A, 6=A, 5=A, 2=A, 16=A}

|  |  |
| --- | --- |
| HashMap | HashTable |
| 1. No method is synchronized 2. HashMap object is not thread safe. 3. Performance is high 4. Null is allowed as both key and value. 5. Introdeuce in 1.2 and not legacy | 1. All methods are synchronized 2. HashTable is Thread safe. 3. Performance is low 4. Null is not allowed for both key and value, otherwise NullPointerException 5. Introduced in 1.0 version and it is legacy. |

* We can get synchronized version of HashMap by using Collections class as follows.
  + HashMap hashMap = new HashMap();

Map syncHashMap = Collections.synchronizedMap(hashMap);

**Properties**

* If anything changing frequently like database usernames and passwords...etc, Those things we never allowed to hardcode inside java. Because for every modification if require rebuild the application followed by redeployment and restart of the server also. Which may impact client business?
* So such type of variable things we have to place inside properties file. For any change in the properties file reloading application is enough.
* **Properties** is child of **HashTable**. Here the both key and values must be String.

**Constructor**: Properties()

**Methods:**

1. **String getProperty(String key);**
   1. Returns the value associated with the specified property
   2. If the specified key is not available then we get **null**
2. **String setProperty(String name, String value);**
   1. To add a new property
   2. If the specified name is already available then old value will be replaced with new value.
3. **Enumeration propertyNames();**
4. **void load(InputStream is)**
   1. To load the properties from properties file to the java **Properties** object.
5. **void store(OutputStream os, String comment)**
   1. To update the properties file from Java **Properties** object.

Example:

**package** com.lit.collection.map;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.util.Properties;

**public** **class** Properties1 {

**public** **static** **void** main(String[] args) **throws** Exception {

Properties properties = **new** Properties();

FileInputStream fis = **new** FileInputStream("abc.properties");

properties.load(fis);

System.*out*.println(properties);

System.*out*.println(properties.get("venki"));

properties.setProperty("nag", "99999");

FileOutputStream fos = **new** FileOutputStream("abc.properties");

properties.store(fos, "this is my updates");

}

}

**Queue**

* If we want to represent a group of objects prior to processing then we should go for Queue.
* In general Queue follows FIFO, But based on our requirement we can change it.
* From 1.5 version onwards **LinkedList** also implements **Queue** interface.

Methods

1. boolean offer(Object obj)
   1. To add an object to the **Queue**.
2. Object peek()
   1. To return head element of the **Queue**.
   2. If **Queue** is empty this method returns **null**
3. Object element()
   1. Return head element of the **Queue** without removal , if **Queue** is empty this method raises runtime exception saying **NoSuchElementException**
4. Object poll()
   1. Remove and returns head element of the Queue.
   2. If the Queue is empty this method returns **null**.

**PriorityQueue:**

* We can use **PriorityQueue** to represent a group of objects prior to processing according some priority.
* Sorting order is preserved , it can be natural sorting order or customised sorting order.
* If we are depending on natural sorting order object should be homogeneous and **Comparable** otherwise we will get **ClassCastException**.
* If we are defining our own sorting order, by **Comparator** then the objects need not be homogeneous and **Comparable**.
* **null** insertion is not possible even it is the first element also. Violation leads to **NullPointerException.**

Constructor:

1. PriorityQueue pq = new PriorityQueue();
   1. Creates an empty PriorityQueue with default initialCpacity 11 and all the objects are arranged according to natural sorting order.
2. PriorityQueue pq = new PriorityQueue(int initialCapacity)
3. PriorityQueue pq = new PriorityQueue(int initialCapacity, Comparator c)
4. PriorityQueue pq = new PriorityQueue(Collection c)
5. PriorityQueue pq = new PriorityQueue(SortedSet s)

* Some platform may not provide the supports for PriorityQueue.

Example:

**package** com.lit.collection.map;

**import** java.util.PriorityQueue;

**public** **class** PriorityQueue1 {

**public** **static** **void** main(String[] args) {

PriorityQueue pq = **new** PriorityQueue();

// System.out.println(pq.peek()); null

// System.out.println(pq.element()); NSEE

**for**(**int** i=0;i<=10;i++){

pq.offer(i);

}

System.*out*.println(pq);

System.*out*.println(pq.poll());;

System.*out*.println(pq);

}

}

Output:

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

0

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

Example:

**package** com.lit.collection.map;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**public** **class** PriorityQueue2 {

**public** **static** **void** main(String[] args) {

PriorityQueue pq = **new** PriorityQueue();

pq.offer("A");

pq.offer("Z");

pq.offer("L");

pq.offer("B");

System.*out*.println(pq);

//[A, B, L, Z]

}

}

**class** PQComparator **implements** Comparator {

**public** **int** compare(Object o1, Object o2) {

String string1 = (String) o1;

String string2 = (String) o2;

**return** string2.compareTo(string1);

}

}

**1.6 version enhancements.**

Collection{I}

|

Set{I}

|

SortedSet{I}

|

NavigableSet{I}

|

TreeSet{I}

**NavigableSet**

* This interface is child interface of **SortedSet**.
* It defines several methods which provide support for navigation.

**Methods**

1. ceiling( e)
   1. Return the lowest element which is >= e
2. higher(e)
   1. Return the lowest value which is >e
3. floor(e)
   1. Returns the highest value which is <= e
4. lower(e)
   1. Returns the highest value which is <e
5. pollFirst();
   1. **remove and retu**rns first element.
6. **pollLast();**
   1. **remove and return last element.**
7. descendingSet();
   1. Returns NavigableSet in reverse order.

Example:

**package** com.lit.collection;

**import** java.util.TreeSet;

**public** **class** NavigableSet1 {

**public** **static** **void** main(String[] args) {

TreeSet<Integer> treeSet = **new** TreeSet<Integer>();

treeSet.add(1000);

treeSet.add(2000);

treeSet.add(3000);

treeSet.add(4000);

treeSet.add(5000);

System.*out*.println(treeSet);

System.*out*.println(treeSet.ceiling(2000));

System.*out*.println(treeSet.higher(2000));

System.*out*.println(treeSet.floor(3000));

System.*out*.println(treeSet.lower(3000));

System.*out*.println(treeSet.pollFirst());

System.*out*.println(treeSet.pollLast());

System.*out*.println(treeSet.descendingSet());

System.*out*.println(treeSet);

}

}

**Output:**

[1000, 2000, 3000, 4000, 5000]

2000

3000

3000

2000

1000

5000

[4000, 3000, 2000]

[2000, 3000, 4000]

**NavigableMap**

* **It is the child interface of SortedMap.**
* **It defines several methods to provide support for navigation.**

**Map**

**|**

**SortedMap**

**|**

**NavigableMap**

**|**

**TreeMap**

**Methods:**

1. **celingKey()**
2. **higherKey()**
3. **floorKey()**
4. **lowerKey()**
5. **pollFirstEntry()**
6. **pollLastEntry()**
7. **descendingMap();**

**Example:**

**package** com.lit.collection.map;

**import** java.util.TreeMap;

**public** **class** NavigableMap1 {

**public** **static** **void** main(String[] args) {

TreeMap<String,String> treeMap = **new** TreeMap<String,String>();

treeMap.put("b", "banana");

treeMap.put("c", "cat");

treeMap.put("a", "apple");

treeMap.put("d", "dog");

treeMap.put("g", "gun");

System.*out*.println(treeMap);

System.*out*.println(treeMap.ceilingKey("c"));

System.*out*.println(treeMap.higherKey("e"));

System.*out*.println(treeMap.floorKey("e"));

System.*out*.println(treeMap.lowerKey("e"));

System.*out*.println(treeMap.pollFirstEntry());;

System.*out*.println(treeMap.pollLastEntry());

System.*out*.println(treeMap.descendingMap());

System.*out*.println(treeMap);

}

}

**Output:**

{a=apple, b=banana, c=cat, d=dog, g=gun}

c

g

d

d

a=apple

g=gun

{d=dog, c=cat, b=banana}

{b=banana, c=cat, d=dog}

**Collections:**

* **It defines several utility methods for Collection objects.**

**Sorting a List**

1. **Public static void sort(List l)**
   1. **To sort according to natural sorting order.**
2. **Public static void sort(List l, Comparator c)**
   1. **To sort according to customized sorting order.**

Example:

**package** com.lit.collection;

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** CollectionsArrayList2 {

**public** **static** **void** main(String[] params) {

ArrayList arrayList= **new** ArrayList();

arrayList.add("L");

arrayList.add("K");

arrayList.add("A");

arrayList.add("N");

System.*out*.println(arrayList);//[L, K, A, N]

Collections.*sort*(arrayList);

System.*out*.println(arrayList);// [A, K, L, N]

}

}

Example:

**package** com.lit.collection;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**public** **class** CollectionsArrayList2 {

**public** **static** **void** main(String[] params) {

ArrayList arrayList = **new** ArrayList();

arrayList.add("L");

arrayList.add("K");

arrayList.add("A");

arrayList.add("A");

arrayList.add("N");

System.*out*.println(arrayList);// [L, K, A, N]

Collections.*sort*(arrayList);

Collections.*sort*(arrayList, **new** ListComparator());

System.*out*.println(arrayList);// [A, K, L, N]

}

}

**class** ListComparator **implements** Comparator {

@Override

**public** **int** compare(Object o1, Object o2) {

String string1 = (String) o1;

String string2 = (String) o2;

**return** string2.compareTo(string1);

}

}

**Output:**

[L, K, A, A, N]

[N, L, K, A, A]

**For Searching**

* **Public static int binarySearch(List l, Object obj);**
* **Public static int binarySearch(List l, Object obj, Comparator c);**
* **Before calling binary search method compalsury List should be sorted.**
* **Successful search return index. Unsuccessful search returns insertion point.**

**To Reversing the elements of a List**

* **public static void reverse(List list);**

**To synchronize**

**java.util.Arrays**

* **Defines several utility methods for arrays.**

**To sort**

1. **Public static void sort(primitiveArray[] a);**
   1. **By natural sorting order**
2. **Public static void sort(Object[] o)**
   1. **By natural sorting order**
3. **Public static viod sort(Object[], Comparator c)**
   1. **By customized sorting order**

**To search**

**Public static int binarySearch(primitiveArray[], primitiveElement)**