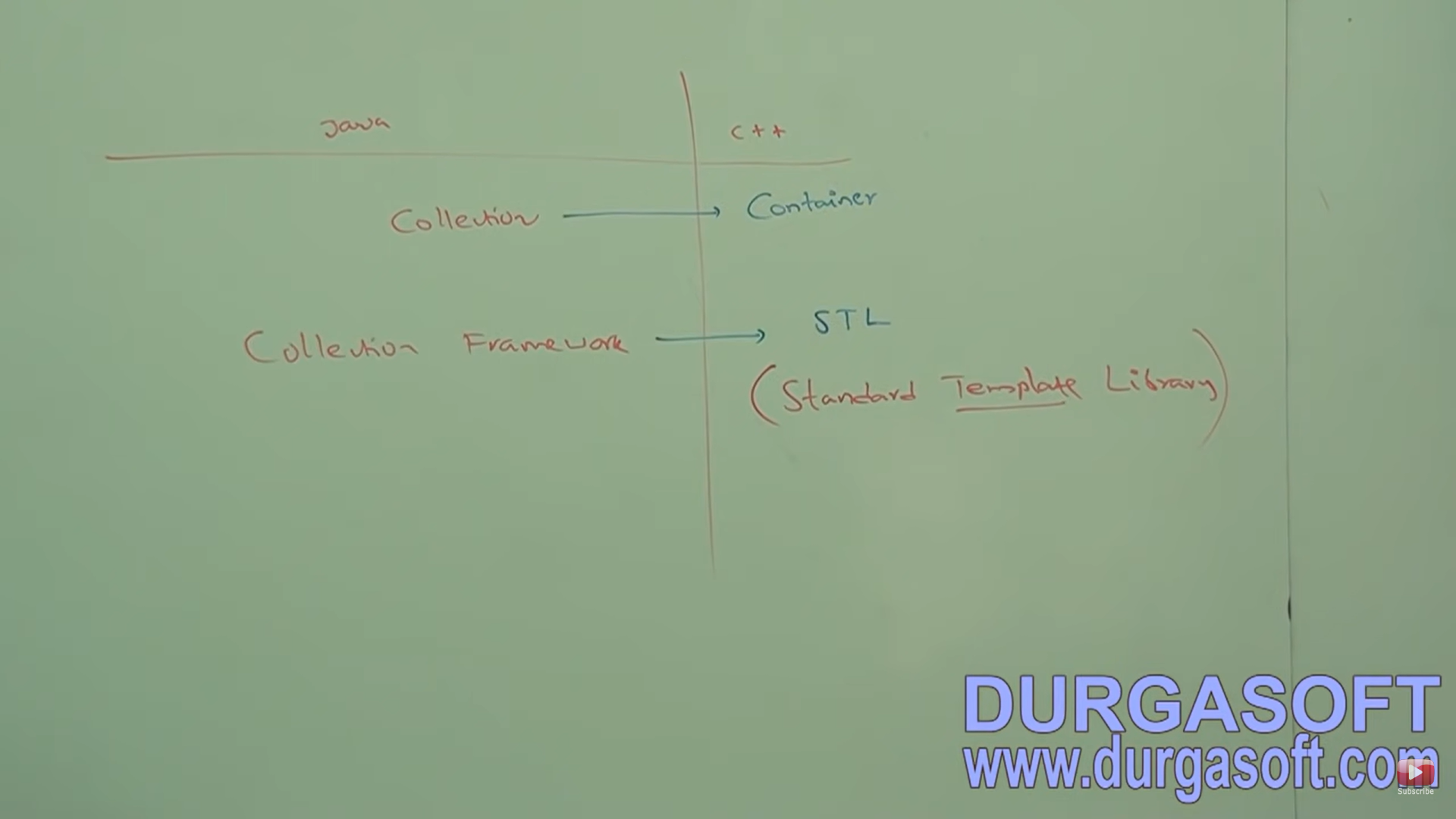
**Collections** are growable in nature.

Collections holds both homogeneous and non-homogeneous.

If we want to represent group of individual objects as a single entity than we should go with collection

Collection Framework: it cantinas several classes and interfaces which can be use to represent

a group of individuals as a single entity.



9 key interfaces of Collection Framework:

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap

Collection(I) : if we want to represent a group of individual object as a

Single entity than we go for Collection.

Collection interface define the most common methods which are available

Any collection objects. In general collection interface is consider as root interface

Of collection framework.

There is no concrete class which implements collection interface directly.

**Collection(I) vs Collections(class)**

Collection is an interface if we want to represent individual

Object as single entity the we should go for collection.

Collections is an utility class present in java.util package to define

Several utility methods for object like sorting searching etc.

**List(I):** it is the child interface of collection. If we want to represent group of

Individual object as a single entity were duplicates are allow and insertion order

Must be preserve then we should go for list.

List Implemented class

1. ArrayList
2. LinkedList
3. Vector
4. Stack

**Set(I):** it is the child interface of collection. If we want to represent group of

Individual object as a single entity were duplicates are not allowed and insertion order

not require then we should go for Set.

Set Implemented Class

1. HashSet
2. LinkedHashSet

**SotedSet(I):** it is the child interface of set. If we want to represent group of

Individual object as a single entity were duplicates are not allowed and insertion order

Require some sorted order then we should go for SortedSet.

**NavigableSet(I)** it is the child interface of SortedSet. It contains several methods for navigation purposes

Implemented class of navigableSet

1. TreeSet

**Queue(I):** it is the child interface of Collection

A person writing on a whiteboard

Description automatically generated

**Map(I)**

A picture containing text, whiteboard

Description automatically generated

**SortedMap(I):** sorted based on key.

**NavigableMap** it is child interface sortedMap. It Define several methods for navigation purposes

Implemented class:

1. TreeMap

**Collection(i) Methods:**

1. boolean add(Object o)
2. boolean addAll(Collection c)
3. boolean remove(Object o)
4. boolean removeAll(Collection c)
5. boolean retainAll(Collection c)
   1. To remove all objects except those present in c
6. void clear()
7. boolean contains(Objects o)
8. boolean containsAll(Collection c)
9. boolean isEmpty()
10. Int size();
11. Object[] toArray();
12. Iterator iterator()

There is no contrite class which implement collection interface directly

**List**

List Interface define the following specific methods

1. void add(int index, Object o)
2. boolean addAll(int index , Collection c)
3. Object get(int index)
4. Object remove(int index)
5. Object set(int index, Object new)

To replace the element, present at specified index with provided Object and returns old object

1. int indexOf(Object o)
   1. returns index of first occurrence of ‘o’
2. int LastIndexOf(object o)
3. ListIterator listIterator();

**ArrayList** :

**Constractor** :

1. ArrayList al = new ArrayList(); default size 10.
2. ArrayList al = new ArrayList(int intialCapacity);
3. ArrayList al = new ArrayList(Collection C);
   1. ArrayList al new ArrayList(collection c) create an equivalent ArrayList element for the given collection

**Methods** :

Set(index,object) to replace obejct at index

Usually be can use collection to hold and transfer object from one location to another location(container) to provide support for this requirement every collection class by default implement serializable and cloneable interfaces

RandomAccess Interface

**LinkedList**

**Constructor:**

1. LinkedList l = new LinkedList();
2. LinkedList l = new LinkedList(Collection c);

**Methods:**

1. add();
2. void addFirst(Object o);
3. void addLast(Object o);
4. Object getfirst();
5. Object getLast();
6. Object removeFirst();
7. Object removeLast();

**Vector**

Resizable

Insertion order

Duplicate

Heterogeneous

Null insertion

Implement serializable cloneable, random access,

Thread safe

**constructors**:

1. Vector v = new Vector();default size 10 , new capacity=cc\*2
2. Vector v = new Vector(int initalCapacity);
3. Vector v = new Vector(int initalCapacity,int incremental capacity)
4. Vector V = new Vector(collection c);

**Methods**:

1. addElement(Object o);
2. removeElement(Object o);
3. removeElementAt(int index);
4. removeAllElements();
5. Object elementAt(int index)
6. Object firstElement();
7. Object lastElement();
8. Int size();
9. Int capacity();
10. Enumeration elements();

**Stack**

It is child class of vector

Last in first out order.

**Constructor:**

1. Stack s = new Stack();

**Methods:**

1. push(Object o);
2. Object pop();
3. Object peek();
4. Bool empty();
5. int search(Object o); -1, return offset if the element is available otherwise return -1.

**The 3 cursors of java.**

Enumeration

Iterator

ListIterator

**Enumeration**

Enumeration e=v.elements();

While(e.hadMoreElements())

{

Integer I = (Integer)e.nextElement();

Sop(I);

}

**Iterator**

We can create iterator object by using iterator() of collection object

Ex

Iterator itr = o.iterator();

**Methods**

* + 1. Public Boolean hasNext();
    2. Public Object next();
    3. Public void remove();

**Limitation**

Single direction

Read and remove not add element not replace.

**ListIterator**

Bidirection

Read, remove, add and remove opetation done.

public ListIterator ListIterator();

Ex: ListIterator its = l.listIterator(); //l = Any list Object

IistIterator is child Interface of Iterator and hence all methods present in iterator by default available by listIterator.

**Methods** :

1. Public Boolean hasNext();
2. Public Boolean next();
3. Public int nextIndex();
4. Public Boolean hasPrevious();
5. Pubic Object previous();
6. Public int previousIndex();
7. Public void remove();
8. Public void add(object o);
9. Public void set(object);

**Limitation**:

Its applicable only for list objects.

A screenshot of a computer

Description automatically generated

**SET**

**HashSet**

1. Underling data structure is hash Table.
2. Duplicates are not allowed.
3. Insertion order is not preserved, and it is based on hash code of object.
4. Null insertion is possible only once.
5. Heterogenous object are allow;
6. Implements serializable and cloneable but not RandomAccess interface
7. HashSet is best choices of search operation.

**Constructors**:

1. HashSet h = new HashSet();
2. HashSet h = new HashSet(int initialCapacity);
3. HashSet h = new HashSet(int initialCap, float fillRatio);
4. HashSet h = new HashSet(Collection c);

**LinkedHashSet**

LinkedHashSet is child class of hashSet.

It’s is exactly same as HashSet Including constructors and methods excepts the following differences

1. Underling data structure is Linked List + Hash Table
2. Insertion order is preserved

LinkedHashSet h = new LinkedHashSet();

Common use LinkedHashSet to develop cache-based application

**SortedSet**

SortedSet is a child interface of set

If we want to represent a group of individual objects according to some sorting order without duplicate, then we should go for SortedSet.

**SortedSet Interface define the following specific method,**

1. Object first();
2. Object last();
3. SortedSet headSet(Object obj)
   1. Return SortedSet whose elements are less than object.
4. SortedSet tailSet(object obj)
   1. Returns SortedSet whose elements are >=obj
5. SortedSet subSet(object obj1,Object obj2)
   1. Returns SortedSet whose elements re >=object and <obj2
6. Comparator comparator()
7. Return Comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null.

**TreeSet**

Underlying data structure is balanced tree

1. Duplicate not
2. Inserting order not preserve
3. Not heterogenous objects are not allows
4. Only once null

**Constructor:**

1. TreeSet t = new TreeSet();
2. TreeSet t = new TreeSet(Comparator c);
   1. Create an empty TreeSet where the element will be insert according to customising sorted order specified by comparator object.
3. TreeSet t = new TreeSet(Collection c);
4. TreeSet t = new TreeSet(SortedSet s);

Null acceptance:

1 t.add(null);

Re:NPE

2 t.add(null);

t.add(“a”); return null pointer exception

import java.util.TreeSet;

public class TreeSetEx {

    public static void main(String args[])

    {

        TreeSet<String> t = new TreeSet<String>();

        t.add("A");

        t.add("B");

        t.add("C");

        System.out.println(t);

    }

}

If we are depending on default natural sorting order compulsory the object should be homogenous and comparable otherwise, we get classCastException

Comparable(Interface)

1. Java.lang package
2. compareTo();

obj1.compareTo(obj2);

|  |  |
| --- | --- |
| - | iff obj1 comes before obj2 |
| + | iff obj1 comes after obj2 |
| 0 | Iff obj1 & obj2 equal |

public class CompareableEx {

    public static void main(String args[]) {

        System.out.println("A".compareTo("Z"));

        System.out.println("A".compareTo("A"));

        System.out.println("Z".compareTo("A"));

    }

}

Comparable means for Default natural sorting Order

Where as

Comparator means for customized Sorting order.

Comparator present in java.util package

And its define two methods

compare()  and  equal

1. public int compare(object obj1,object obj2)
   1. Return  -ve  iff obj1 has to come before obj2
   2. Return  +ve iff obj1 has to come after obj2
   3. Return  0 iff obj1 and obj2 equal.
2. Public boolean equals(object obj)

Whenever we are implementing comparator interface compulsory we should provide implementation only for compare() and we are not required to implement equals methods because it is already available to our class from object class through inheritance.

Write a program to insert an integer object into the treeSet where the sorting order is descending order?

import java.util.Comparator;

import java.util.TreeSet;

import java.util.\*;

/\*\*

 \* MyComparator

 \*/

class MyComparator implements Comparator {

       public int compare(Object obj1 , Object obj2)

       {

            Integer a = (Integer)obj1;

            Integer b = (Integer)obj2;

            if(a<b)

                return 1;

            else if(a>b)

                return -1;

            else

                return 0;

// or

       // return -a.compareTo(b);

       }

}

public class ComparetorEx {

    public static void main(String[] args) {

        TreeSet<Integer> t = new TreeSet<Integer>(new MyComparator());//1

        t.add(15);

        t.add(51);

        t.add(25);

        t.add(53);

        t.add(45);

        System.out.println(t);

    }

}

At line 1

If be are passing comparator object then JVM will call compare method which is mean for customize sorting in this care out opt

51 53 45 25 15

By default it call compareTo();

MAP

–HashMap→linkedHashMap

–IdentityHashMap

–WeakHashMap

–SortedMap→ NavigableMap→ TreeMap

–HashTable→Properties

Map is not a child interface of collection.

If we want a group of objects as key value pairs we should go with Map.

Both keys and values are objects only

Duplicate keys are not allowed but values can be duplicates.

Each key value pairs is called Entry hence map is considered collection of entry object

Method:

1. Object put(object key, object value);
   1. To add one key value pair
   2. If the key is already present then old value will be replaced with new value and returns old value.
2. void putAll(Map m);
3. Object get(key)
4. Object remove(key)
5. boolean containsKey(key)
6. boolean containValue(value)
7. Boolean isEmpty()
8. Int size()
9. Void clear();

Collection views of MAP

1. Set keySet()
2. Collection values()
3. Set entrySet()

Entry Interface

Without an existing map object there is no chance of existing entry object hence entry interface is defined in map interface.

interface map

{

    interface Entry

    {

        Object getKey();

        Object getValue();

        Object setValue(Object newObj);

    }

}

Entry Specific methods and we can apply only on Entry objects.

**HashMap**

1. Underlying data structure is HASH TABLE
2. Insertion order not preserve
3. Duplicates keys are not allow but value can be duplicate
4. Heterogenous objects are allow
5. null is allows only once
6. null is allows for value any number of time
7. Implements Serializable cloneable and RandomAccess
8. Search operation

Constructor:

1. HashMap m = new HashMap(); default capacity 16 and default fill ratio 75%
2. HashMap m = new HashMap(int initialCapacity);
3. HashMap m = new HashMa[(int initialCapacity, float fillRatio)
4. HashMap m = new HashMap(Map m);

Example

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

public class HashMapDemo {

    public static void main(String[] args) {

        HashMap<Integer,String> h = new HashMap<Integer,String>();

        h.put(1,"January");

        h.put(2,"February");

        h.put(3,"March");

        h.put(4,"April");

        System.out.println(h);

        System.out.println(h.put(1,"Jan"));

        System.out.println(h);

        Set s = h.keySet();

        System.out.println(s);

        Set s1 = h.entrySet();

        System.out.println(s1);

        Iterator itr = s1.iterator();

        while(itr.hasNext())

        {

            Map.Entry m1 = (Map.Entry)itr.next();

            System.out.println(m1.getKey() + " = " + m1.getValue());

            if(m1.getValue().equals("Jan"))

                m1.setValue("January");

        }

        System.out.println(h);

    }

}

Difference between HashMap and HashTable

|  |  |
| --- | --- |
| HashMap | HashTable |
| 1. Methods are not synchronized 2. At a time multiple threads are allowed to operate on hashMap objects and hence it is not thread safe. 3. Relatable performance is high because threads are not required to wait to operate on hashmap objects. 4. Null is allow for both key and value 5. 1.2 version and its is not legacy | 1. Methods are Synchronized 2. At a time only one thread is allowed to operate on a hashMap object and hence it is thread safe. 3. Relatable performance is low because threads are required to wait to operate on hashTable objects. 4. Null is not allowed for key and value otherwise we will get null pointer exception 5. 1.0 version and it is legacy. |

How to get a Synchronized version of hashMap Object?

By default hashMap is not synchronized but we can get a synchronized version of hashMap by using the synchronizedMap method of collections class.

 HashMap<Integer,String> h = new HashMap<Integer,String>();

        Map map = Collections.synchronizedMap(h);

**LinkedHashMap**

It is a child class of HashMap. It is exactly same as hashMap including methods and constructor except the following differences

| LinkedHashMap | HashMap |
| --- | --- |
| 1. The underlying data structure is combination of linked List and hashTable 2. Insertion order is preserve | 1. The underlying data structure is hashTable. 2. Insertion order is not preserved and it is based on hash Code of key |

LinkedHashSet and LinkedHashMap are commonly used  for developing cache based applications.

**IdentityHashMap**

It is exactly same as HashMap including methods and constructor except the following differences

In the case of normal HashMap JVM will use .equals methods to identify duplicate key which is meant for contains comparator but in case of IdentityHashMap JVM will use == operator to identify duplicate key which is meant for address comparator.

Example

import java.util.HashMap;

import java.util.IdentityHashMap;

public class IdentityHashMapDemo {

    public static void main(String[] args) {

        HashMap<String,String> m = new HashMap<String,String>();

        m.put(new String("5"),"Roushan");

        m.put(new String("5"),"Ram");

IdentityHashMap<String,String> im = new IdentityHashMap<String,String>();

        im.put(new String("5"),"Roushan Kumar");

        im.put(new String("5"),"Ram Kumar");

        System.out.println(m);

        System.out.println(im);

    }

}

**WeakHashMap**

**SortedMap**

It is the child interface of map.

If we want to represent a group of objects as a group of key value pairs according to some sorting order of keys then we should go for SortedMap

Sorting is based on key but not based on value.

sortedMap define the following 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()

The underlying data structure is Red Black Tree. Insertion order is not preserved and it is based on some sorting order of keys. Duplicated keys are not allowed but values can be duplicated. If we are depending on default natural sorting order then keys should be homogenous and comparable otherwise we will get classCastException if we are defining our own sorting by comparator then keys need not be homogenous and comparable we can take heterogeneous non comparable objects also.

Constructors:

1. TreeMap t = new TreeMap();

For Default Natural Sorting Order

1. TreeMap t = new TreeMap(Comparator c);

For Customized Sorting Order

1. TreeMap t = new TreeMap(SortedMap m);
2. TreeMap t = new TreeMap(Map m);

Hashtable

1. Underlying data structure is Hashtable.
2. Insertion order is not preserved and it is based on the hash code of keys.
3. Duplicated keys are not allowed and values can be duplicated.
4. Heterogeneous objects are for both keys and values.
5. Null is not allowed for both keys and value otherwise we will get NPE.
6. It implements a serializable and cloneable interface but not random access.
7. Every methods present in hash table is synchronized and hence hashtable is thread safe
8. HashTable is the best choice if our frequent operation is search operation.

Constructors:

1. Hashtable ht = new Hashtable ();
2. Hashtable ht = new Hashtable (int initialCapacity);
3. Hashtable ht = new Hashtable (int initialCapacity, float fillRatio);
4. Hashtable ht = new Hashtable (map m);

import java.util.Hashtable;

public class HashTableDemo {

    public static void main(String[] args) {

        Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

//Constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).

        ht.put(5,"A");//insert in 5%11=5 place

        ht.put(2,"A");//insert in 2%11=2 place

        ht.put(6,"A");//insert in 6%11=6 place

        ht.put(15,"A");//insert in 15%11=4 place

        ht.put(23,"A");//insert in 23%11=1 place

        ht.put(16,"A");//insert in 16%11=5 place

        System.out.println(ht);

    }

}

Output

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

**Print top to bottom and right to left.**

|  |  |
| --- | --- |
| 10 |  |
| 9 |  |
| 8 |  |
| 7 |  |
| 6 | 6=A |
| 5 | 5=A,16=A |
| 4 | 15=A |
| 3 |  |
| 2 | 2=A |
| 1 | 23=A |
| 0 |  |

**Properties**

In normal map like hashmap treemap key and value any type but in the case of properties key and value should be string type

Constructor:

1. Properties p = new Properties();

Methods:

1. String setProperty(String pname, String pvalue);
2. String getProperty(String pname);
3. Enumeration propertyNames();
4. void load(InputStream is);
5. void store(OutputStream os,String comment)

Example:

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.util.Properties;

public class PropertiesDemo {

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

    {

        Properties p = new Properties();

        FileInputStream fis = new FileInputStream("app.properties");

        p.load(fis);

        System.out.println(p);

        if(p.getProperty("user").equals("Roushan559"))

        System.out.println("Hello Roushan");

        p.setProperty("url", "https://google.com");

        p.setProperty("path", "D:\\download");

        FileOutputStream fos = new FileOutputStream("app.properties");

        p.store(fos, "url update by Roushan Kumar");

    }

}

Output

{path=D:\download, pwd=Tiger@, user=Roushan559, url=https://google.com}

Hello Roushan

app.properties

#url update by Roushan Kumar

#Fri Feb 24 23:36:01 IST 2023

path=D\:\\download

pwd=Tiger@

user=Roushan559

url=https\://google.com

**Queue**

It is the child interface of collection

Usually queue follow first in first out order but based on our requirement we can implement our own priority order(priority Queue).

LinkedList based implementation of queue always follow first in first out order.

Queue interface specific methods.

Methods:

* 1. Boolean offer(Object o)
     1. To add an object into the queue
  2. Object peek()
     1. To return head element. If queue is empty, then returns null.
  3. Object element()
     1. To return head element of the queue. If queue is empty, then this method raises RE: NoSuchElementException
  4. Object poll()
     1. To remove and return head element of the queue. If queue is empty, then this method returns null.
  5. Object remove()
     1. To remove and return head element of the queue. If queue is empty, then this method raises RE: NoSuchElementException

**Priority Queue**

If we want to represent a group of individual objects prior to process according to some priority, then we should go for priority Queue. Insertion order is not preserved, and it based on some priority. Duplicates object are not allowing. If we are depending on default natural sorting order compulsory object should be homogenous and comparable otherwise, we get classCastException. If we are defining our own sorting by comparator then object need not be homogenous and comparable. Null is not allowed even first element also.

**Constructor**:

1. PriorityQueue q = new PriorityQueue(); // 11
2. PriorityQueue q = new PriorityQueue(int initialCapacity);
3. PriorityQueue q = new PriorityQueue(int initialCapacity, Comparator c);
4. PriorityQueue q = new PriorityQueue(Collection c);

**NavigableSet**

It is the child interface of sorted Set and it define several methods for navigation purpose

Methods:

* + - 1. floor(e)

it returns highest element which is <=e

* + - 1. lower(e)

it returns highest element which is <e

* + - 1. ceiling(e)

it returns lowest element which is >=e

* + - 1. higher(e)

it returns lowest element which is >e

* + - 1. pollFirst()

remove and return first element

* + - 1. pollLast()

remove and return last element

* + - 1. descendingSet()

it returns NavigableSet in reverse order

public class NavigableSetDemo {

    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);

        ts.add(6000);

        System.out.println(ts);

        System.out.println(ts.floor(3000));

        System.out.println(ts.lower(3000));

        System.out.println(ts.ceiling(3000));

        System.out.println(ts.higher(3000));

        System.out.println(ts.pollFirst());

        System.out.println(ts.pollLast());

        System.out.println(ts.descendingSet());

        System.out.println(ts);

    }

}

**NavigableMap**

Methods:

* + - * 1. floorKey(e)
        2. lowerKey(e)
        3. ceilingKey(e)
        4. higherKey(e)
        5. pollFirstEntry()
        6. pollLastEntry()
        7. descendingMap()

**Collections**

**Collections class define several utility methods for collection object like sorting searching reversing etc.**

**Soring element of list:**

Collections class define following two sort methods

public static void sort(list l)

public static void sort(list l, comparator c)

**Searching element of list:**

Collections class defines the following binary search methods

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

if the list is sorted according to default natural sorting order

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

we must use this method if the list is sorted according to customized sorting order

conclusion

the over search methods internally will use binary search algorithm. Successful search return index unsuccessful search return insertion point. Insertion point is the location where we can place target in sorted list. Before calling binary search method compulsory list should be sorted otherwise, we will get unpredictable result. If the list is sorted according to comparator then at the time of search operation also, we must pass same comparable object otherwise we get unpredictable result.

**Reversing elements of List**

Collections class defines the following reverse method to reverse element of List

public static void reverse(List l)

**Arrays**

Arrays class is a utility class to define several utility methods for array object.

**Sorting elements of array.**

Arrays class define the following sort methods to sort elements of primitive and object type array.

public static void sort(primitive[] p)

To sort according to Natural sorting Order

public static void sort(Object[] o)

To sort according to natural sorting order

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

To sort according to Customized sorting Order

Searching elements of array.

1. public static int binarySearch(primitive[] p, primitive target)

2. public static int binarySearch(Object[] a, Object target)

3. public static int binarySearch(Object[] a, Object target, Comparator c)

Conversion of Array to List

1. public static List asList(Object[] o)

Strictly speaking this method don’t create an independent list object for the exacting array we are getting list view.

By using array reference if we perform any change automatically the change will be reflected to list similarly by using list reference if we perform any change that change will we reflected automaticity to the array.

By using list reference, we can’t perform any operation which vary the size otherwise we will get runtime exception UnsupportedOperationException

by using list reference, we are not allowed to replace with heterogenous object otherwise we will get run time exception ArryaStoreException.