**Collections**

**Array:** An array is a collection of elements which are stored in continuous memory locations. Arrays can be used for storing both primitive types and objects.

**Example:**

int[] arr = new int[5];

double []marks = new double[10];

Student stu[] = new Student[25];

stu[0] = new Student(); //valid

stu[1] = new Employee(); //invalid

Employee []emp = new Employee[100];

emp[0] = new Employee(); //valid

emp[1] = new Student(); //invalid

In the Student array we can store only Student objects but we cannot store Employee objects and in the Employee array we can store only Employee objects but we cannot store Student objects.

To overcome the above problem we can create an array of Object class where we can store an object of any class.

Object obj[] = new Object[50];

obj[0] = new Object();

obj[1] = new Student();

obj[2] = new Employee();

**Limitations of array (or) disadvantages of array:**

1. The size of an array is fixed i.e., once the array is created, the size cannot be increased or decreased. Therefore either the memory may be wasted or the memory may not be sufficient.
2. To perform the operations like insertion, deletion, searching, sorting etc, the array concept does not provide any predefined methods. The programmer has to write their own logic to perform these operations.

To overcome the limitations of the array concept, the java soft people have come up with a concept called collections. The collections are introduced in java 1.2 version.

The collections are designed to store only objects.

**Collection object:** An object is said to be a collection object if it holds or stores a group of other objects.

group of

objects

Collection Object

Assume every object requires 2 bytes of memory. To store 4 objects we require 8 bytes of memory. The objects referred by s1, s2, s3 and s4 are stored two times, one time inside the collection object and one time outside the collection object, there by wasting the memory. To save the memory, the jvm instead of storing the objects directly, stores the references of the objects in the collection object.

references of a group

of other objects

Collection Object

**Collection object:** An object is said to be collection object if it holds or stores a group of references of other objects. The collection object can also be called as **container object**.

**Collection class:** A collection class is a class whose object can hold or store a group of other objects.

**Note:** collections cannot store primitive type values.

All the collection classes and interfaces are together called as **collection framework**. Collection framework is a library which is implemented in java.util(utility) package.

**The advantages of collection framework are:**

1. **Reduces programming effort** by providing data structures and algorithms so that we don’t have to write them on our own.
2. **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
3. **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
4. New data structures that conform to the standard collection interfaces are by nature **reusable**.

All the collection classes are classified into three categories**:**

1. **List:** This category can be used for storing a group of individual elements where the elements can be duplicated. List is an interface which cannot be instantiated and therefore we take the help of the implementation classes. The implementation classes of List interface are

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

1. **Set:** This category can be used for storing a group of individual elements where the elements cannot be duplicated. Set is an interface which cannot be instantiated and therefore we take the help of the implementation classes. The implementation classes of Set interface are

1) HashSet 2) LinkedHashSet 3) TreeSet

1. **Map:** This category can be used for storing the elements in the form of key-value pairs where the keys cannot be duplicated but the values can be duplicated. Map is an interface which cannot be instantiated and therefore we take the help of the implementation classes. The implementation classes of Map interface are

1) HashMap 2) LinkedHashMap 3) TreeMap 4) Hashtable

**List Category:** List category can be used for storing a group of individual objects. List category allows duplicates.

**ArrayList:**

* ArrayList is an implementation class of List interface.
* ArrayList can be used for storing individual objects.
* ArrayList allows duplicates.
* ArrayList allows null value.
* ArrayList is not synchronized.

**Creation of ArrayList:**

**Syntax:**

1. ArrayList<E> al = new ArrayList<E>();

The above syntax creates an empty list with the default initial capacity as 10.

1. ArrayList<E> al = new ArrayList<E>(int initialCapacity);

The above syntax creates an empty list with the specified capacity as the default initial capacity.

1. ArrayList<E> al = new ArrayList<E>(Collection c);

The above syntax creates a list with the elements that are available in the specified Collection.

Here, E represents the element data type

**Methods of ArrayList:**

1. **boolean add(Element obj):** This method is used to place the specified element at the end of List.
2. **void add(int index, Element obj):** This method is used to insert the specified element at the specified index position.
3. **boolean addAll(Collection c):** This method is used to append all the elements available in the specified collection into the list.
4. **boolean addAll(int index, Collection c):** This method is used to insert all the elements available in the specified collection into the list at the specified index position.
5. **boolean remove(Element obj):** This method is used to remove the first occurrence of the specified element.
6. **Element remove(int position):** This method is used to remove an element available at the specified index position.
7. **void clear()**: This method will remove all the elements available in the list.
8. **int size():** This method will return the count of the number of elements available in the list.
9. **boolean contains(element obj):** This method returns true if the specified element is available in the list.
10. **Element get(int position):** This method is used to access the element that is available in the specified index position.
11. **Element set(int position, Element obj):** This method is used to replace an element at the specified index position with the specified element.
12. **boolean isEmpty():** This method returns true if the list is empty.
13. **Object[] toArray():** This method converts a list into an array of objects.

**Program :**

import java.util.\*;

public class ArrayListDemo {

public static void main(String[] args) {

ArrayList<String> al = new ArrayList<String>();

al.add("Nokia");

al.add("Samsung");

al.add("Sony");

al.add("HTC");

al.add(3,"Motorla");

System.out.println("List : "+al);

al.remove("HTC");

al.remove(1);

System.out.println("List : "+al);

System.out.println("size : "+al.size());

Iterator it = al.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

**LinkedList**:

* LinkedList is an implementation class of List interface.
* LinkedList can be used for storing individual objects.
* LinkedList allows duplicates.
* LinkedList allows null value.
* LinkedList is not synchronized.

**Creation of LinkedList**:

**Syntax:**

1. LinkedList<E> ll = new LinkedList<E>();

The above syntax creates an empty list.

1. LinkedList<E> ll = new LinkedList<E>(Collection c);

The above syntax creates a list with the elements that are available in the specified Collection.

Here, E represents the element data type.

**Note:** LinkedList can use the same methods as specified in the ArrayList.

**Difference between Array List and Linked List:**

ArrayList is an implementation class which follows resizable array structure. ArrayList is faster in accessing the elements and slower in performing insertions and deletions. LinkedList is an implementation class which follows double linked list structure. LinkedList is slower in accessing the elements and faster in performing insertions and deletions.

**Program :**

import java.util.\*;

public class LinkedListDemo {

public static void main(String[] args) {

LinkedList<Student> ll = new LinkedList<Student>();

ll.add(new Student(34));

ll.add(new Student(12));

ll.add(new Student(56));

ll.add(new Student(90));

ll.add(new Student(78));

ll.add(2,new Student(45));

System.out.println("List : "+ll);

ll.remove(2);

System.out.println("List : "+ll);

Iterator it = ll.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

class Student {

int rollNo;

Student(int rollNo) {

this.rollNo = rollNo;

}

public String toString() {

return " "+rollNo;

}

}

**Vector:**

* Vector is an implementation class of List interface.
* Vector can be used for storing individual objects.
* Vector allows duplicates.
* Vector allows null value.
* Vector is synchronized.

**Creation of Vector:**

**Syntax:**

1. Vector<E> v = new Vector<E>();

The above syntax creates an empty list with the default initial capacity as 10.

1. Vector<E> v = new Vector<E>(int initialCapacity);

The above syntax creates an empty list with the specified capacity as the default initial capacity.

1. Vector<E> v = new Vector<E>(Collection c);

The above syntax creates a list with the elements that are available in the specified Collection.

Here, E represents the element data type.

**Program :**

import java.util.\*;

public class VectorDemo {

public static void main(String[] xyz) {

Vector<Integer> v = new Vector<Integer>();

v.add (new Integer(11));

v.add(new Integer(22));

v.add(new Integer(33));

v.add(44);//auto boxing

v.add(55);

v.add(66);

v.add(1,99);

System.out.println("List: "+v);

v.remove(new Integer(22));

v.remove(1);

System.out.print("List using for loop: ");

for(int i=0;i<v.size();i++){

System.out.print(v.get(i)+" ");

}

System.out.print("\nList using for each loop: ");

for(int x : v) {

System.out.print(x+" ");

}

ListIterator lit = v.listIterator();

System.out.print("\nforward direction: ");

while(lit.hasNext()){

System.out.print(lit.next()+ " ");

}

System.out.print("\nbackward direction: ");

while(lit.hasPrevious()){

System.out.print(lit.previous()+" ");

}

}

}

**Stack:**

* Stack is an implementation class of List interface.
* Stack can be used for storing individual objects.
* Stack allows duplicates.
* Stack allows null value.
* Stack is synchronized.
* Stack class can be used for implementing stack data structure.

**Creation of Stack:**

**Syntax:**

1. Stack<E> s = new Stack<E>();

The above syntax creates an empty list with the default initial capacity as 10.

Here, E represents the element data type.

**Methods of Stack:**

1. **Object push(Object):** This method is used to push the specified object into the top of the stack.
2. **Object pop():** This method is used to remove the object from the top of the stack.
3. **Object peek():** This method is used to access the top most element from the stack.
4. **boolean empty():** This method is used to check whether the stack is empty or not.
5. **int search(Object):** This method is used to check whether the specified element is available or not.

**Program :**

import java.util.Stack;

public class StackDemo {

public static void main(String[] args) {

Stack<Integer> s = new Stack<Integer>();

s.push(50);

s.push(20);

s.push(40);

s.push(60);

s.push(10);

s.push(30);

System.out.println("List: "+s);

System.out.println(s.pop());

System.out.println("List: "+s);

System.out.println(s.peek());

System.out.println("List: "+s);

}

}

**Differences between the List implementation classes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ArrayList | LinkedList | Vector | Stack |
| Ordered | Ordered by  Insertion | Ordered by  Insertion | Ordered by  Insertion | Ordered by  Insertion |
| Duplicates | Allowed | Allowed | Allowed | Allowed |
| Null value | Allowed | Allowed | Allowed | Allowed |
| Synchronized | Not Synchronized | Not Synchronized | Synchronized | Synchronized |
| Data Structure | Resizable Array | Double Linked List | Resizable Array | Resizable Array |
| Initial Capacity | 10 | ---- | 10 | 10 |

**Set Category:** Set category can be used for storing a group of individual objects. Set category does not allow duplicates.

**HashSet:**

* HashSet is an implementation class of Set interface.
* HashSet can be used for storing individual objects.
* HashSet does not allow duplicates.
* HashSet allows null value.
* HashSet is not synchronized.
* HashSet does not guarantee the order of insertion.

**Creation of HashSet:**

**Syntax:**

1. HashSet<E> hs = new HashSet<E>();

The above syntax creates an empty set with the default initial capacity as 16.

1. HashSet<E> hs = new HashSet<E>(int initialCapacity);

The above syntax creates an empty set with the specified capacity as the default initial capacity.

1. HashSet<E> hs = new HashSet<E>(Collection c);

The above syntax creates a set with the elements that are available in the specified Collection

Here, E represents the element data type.

**Methods of HashSet:**

1. **boolean add(Element obj):** This method is used to place the specified element into the set.
2. **boolean remove(Element obj):** This method is used to remove the specified element from the set if available.
3. **boolean contains(Element obj):** This method returns true if the specified element is available in the set.
4. **boolean isEmpty():** This method returns true if the set is empty.
5. **int size():** This method returns the count of the number of elements available in the set.
6. **void clear():** This method is used to remove all the elements from the set.

**Program :**

import java.util.\*;

public class HashSetDemo {

public static void main(String[] args) {

HashSet<Integer> hs = new HashSet<Integer>();

hs.add(56);

hs.add(23);

hs.add(67);

hs.add(45);

hs.add(12);

hs.add(34);

System.out.println("Set : "+hs);

hs.remove(67);

System.out.println("Set : "+hs);

Iterator it = hs.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

**LinkedHashSet:**

* LinkedHashSet is an implementation class of Set interface.
* LinkedHashSet can be used for storing individual objects.
* LinkedHashSet does not allow duplicates.
* LinkedHashSet allows null value.
* LinkedHashSet is not synchronized.
* LinkedHashSet guarantees the order of insertion.

**Creation of LinkedHashSet:**

**Syntax:**

1. LinkedHashSet<E> lhs = new LinkedHashSet<E>();

The above syntax creates an empty set with the default initial capacity as 16.

1. LinkedHashSet<E> lhs = new LinkedHashSet<E>(int initialCapacity);

The above syntax creates an empty set with the specified capacity as the default initial capacity.

1. LinkedHashSet<E> lhs = new LinkedHashSet<E>(Collection c);

The above syntax creates a set with the elements that are available in the specified Collection.

Here, E represents the element data type.

**Program :**

import java.util.\*;

class LinkedHashSetDemo {

public static void main(String[] args) {

LinkedHashSet<Integer>lhs = new LinkedHashSet<Integer>();

lhs.add(56);

lhs.add(23);

lhs.add(67);

lhs.add(45);

lhs.add(12);

lhs.add(34);

System.out.println("Set : "+lhs);

lhs.remove(67);

System.out.println("Set : "+lhs);

Iterator it = lhs.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

**TreeSet:**

* TreeSet is an implementation class of Set interface.
* TreeSet can be used for storing individual objects.
* TreeSet does not allow duplicates.
* TreeSet is not synchronized.
* TreeSet sorts the elements in natural order(ascending order).
* TreeSet allows null value in java 1.6 version if it contains only one element and java 1.7 version does not allow null value even if the TreeSet contains one element.

**Creation of TreeSet:**

**Syntax:**

1. TreeSet<E> ts = new TreeSet<E>();

The above syntax creates an empty set where the elements will be sorted in natural order.

1. TreeSet<E> ts = new TreeSet<E>(Comparator c);

The above syntax creates an empty set where the elements will be sorted according to the specified comparator.

1. TreeSet<E> ts = new TreeSet<E>(Collection c);

The above syntax creates a set with the elements that are available in the specified Collection.

Here, E represents the element data type.

**Program :**

import java.util.\*;

class TreeSetDemo {

public static void main(String[] args) {

TreeSet<Integer> ts = new TreeSet<Integer>();

ts.add(56);

ts.add(23);

ts.add(67);

ts.add(45);

ts.add(12);

ts.add(34);

System.out.println("Set : "+ts);

Iterator it = ts.iterator();

while(it.hasNext()){

System.out.println(it.next());

}

}

}

The TreeSet is a class which sorts the elements by default in natural order(ascending order). If we want to change the order of sorting, then we need to implement **Comparator** interface.

Comparator interface is available in java.util package and it contains 2 methods

1. public abstract int compare(Object o1, Object o2)
2. public abstract boolean equals(Object obj)

The default logic of compare() is to sort the elements in ascending order(natural order).

int compare(Object o1, Object o2) {

if(o1 < o2)

return –VE;

else if(o1 > o2)

return +VE;

else

return 0;

}

**Program :**

import java.util.\*;

class TreeSetDemo {

public static void main(String[] args) {

TreeSet<Integer> ts = new TreeSet<Integer>(new Comparator<Integer>() {

public int compare(Integer i1, Integer i2) {

if(i1 < i2)

return 12;

else if(i1 > i2)

return -12;

else

return 0;

}

});

ts.add(56);

ts.add(23);

ts.add(67);

ts.add(45);

ts.add(12);

ts.add(34);

System.out.println("Set : "+ts);

Iterator it = ts.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

}

TreeSet is a class which stores the elements in sorted order. To sort the elements in sorted order, we take the help of Comparator interface. The Comparator interface belongs to java.util package, which will decide whether the elements have to be sorted either in ascending order or descending order.

Two objects can be compared by the Comparator interface only if those objects are eligible for comparison. An object is said to be eligible for comparison when its corresponding class implements Comparable interface.

The Comparable interface is available in java.lang package and it contains only one method.

1. public int compareTo(Object obj)

**Program :**

import java.util.\*;

class Student implements Comparable {

int rollNo;

Student(int rollNo) {

this.rollNo = rollNo;

}

public int compareTo(Object obj) {

Student s = (Student)obj;

if(this.rollNo < s.rollNo)

return -12;

else if(this.rollNo>s.rollNo)

return 12;

else

return 0;

}

public String toString() {

return "" + rollNo;

}

}

public class TreeSetDemo {

public static void main(String[] args) {

TreeSet<Student> ts = new TreeSet<Student>();

ts.add(new Student(14));

ts.add(new Student(11));

ts.add(new Student(15));

ts.add(new Student(13));

ts.add(new Student(16));

ts.add(new Student(12));

System.out.println("Set : "+ts);

}

}

**Differences between the Set implementation classes**

|  |  |  |  |
| --- | --- | --- | --- |
|  | HashSet | LinkedHashSet | TreeSet |
| Ordered | Unordered | Ordered by Insertion | Sorted Order |
| Duplicates | Not Allowed | Not Allowed | Not Allowed |
| Null value | Allowed | Allowed | Allowed |
| Synchronized | Not Synchronized | Not Synchronized | Not Synchronized |
| Data Structure | Hashtable | Hashtable +  double linked list | Balanced Tree |
| Initial Capacity | 16 | 16 | ---- |

**Map Category:** Map category can be used for storing a group of objects in the form of key-value pairs. Map category does not allow keys to be duplicated whereas values can be duplicated.

**HashMap:**

* HashMap is an implementation class of Map interface.
* HashMap can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
* HashMap allows null in both keys and values.
* HashMap is not synchronized.
* HashMap does not guarantee the order of insertion.

**Creation of HashMap:**

**Syntax:**

1. HashMap<K,V> hm = new HashMap<K,V>();

The above syntax creates an empty map with the default initial capacity as 16.

1. HashMap<K,V> hm = new HashMap<K,V>(int initialCapacity);

The above syntax creates an empty map with the specified capacity as the default initial capacity.

1. HashMap<K,V> hm = new HashMap<K,V>(Map m);

The above syntax creates an empty map with the elements that are available in the specified Map.

Here, K represents the type of the key and V represents the type of the value.

**Methods of HashMap:**

1. **value put(Object key, Object value):** This method is used to place a key and a value as a pair into the map.
2. **value remove(Object key):** This method is used remove the specified key and it’s corresponding value.
3. **value get(Object key):** This method will return the value of the key that is specified.
4. **Set keySet():** This method returns all the keys available in the map in the form of a set.
5. **Collection values():** This method returns all the values available in the map in the form of a collection.
6. **void clear():** This method is used to remove all the key-value pairs from the map.
7. **int size():** This method will return the count of the number of key-value pairs available in the map.
8. **boolean containsKey(Object key):** This method returns true if the specified key is available in the map.
9. **boolean containsValue(Object value):** This method returns true if the specified value is available in the map.
10. **boolean isEmpty():** This method returns true if the map is empty.

**Program :**

import java.util.\*;

public class HashMapDemo {

public static void main(String[] args) {

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

hm.put("mnop",40);

hm.put("abcd",80);

hm.put("pqrs",70);

hm.put("qwer",60);

hm.put("stuv",50);

hm.put("ghij",30);

System.out.println("Elements : "+hm);

hm.remove("stuv");

System.out.println("Elements : "+hm);

Set s = hm.keySet();

System.out.println("keys : "+s);

Iterator it = s.iterator();

while(it.hasNext()) {

Object obj = it.next();

String str = (String) obj;

System.out.println(str+" "+hm.get(str));

}

Collection<Integer> c = hm.values();

System.out.println("values : "+c);

System.out.println(hm.containsKey("abcd"));

System.out.println(hm.containsValue(50));

}

}

**LinkedHashMap:**

* LinkedHashMap is an implementation class of Map interface.
* LinkedHashMap can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
* LinkedHashMap allows null in both keys and values.
* LinkedHashMap is not synchronized.
* LinkedHashMap guarantees the order of insertion.

**Creation of LinkedHashMap:**

**Syntax:**

1. LinkedHashMap<K,V> lhm = new LinkedHashMap<K,V>();

The above syntax creates an empty map with the default initial capacity as 16.

1. LinkedHashMap<K,V> lhm = new LinkedHashMap<K,V>(int initialCapacity);

The above syntax creates an empty map with the specified capacity as the default initial capacity.

1. LinkedHashMap<K,V> lhm = new LinkedHashMap<K,V>(Map m);

The above syntax creates an empty map with the elements that are available in the specified Map.

Here, K represents the type of the key and V represents the type of the value.

**Program :**

import java.util.\*;

public class LinkedHashMapDemo {

public static void main(String[] args) {

LinkedHashMap<String, Integer> lhm = new LinkedHashMap<String, Integer>();

lhm.put("mnop",40);

lhm.put("abcd",80);

lhm.put("pqrs",70);

lhm.put("qwer",60);

lhm.put("stuv",50);

lhm.put("ghij",30);

System.out.println("Elements : "+lhm);

lhm.remove("stuv");

Set s = lhm.keySet();

System.out.println("keys : "+s);

Iterator it = s.iterator();

while(it.hasNext()) {

Object obj = it.next();

String str = (String) obj;

System.out.println(str+" "+lhm.get(str));

}

Collection<Integer> c = lhm.values();

System.out.println("values : "+c);

System.out.println(lhm.containsKey("abcd"));

System.out.println(lhm.containsValue(50));

}

}

**TreeMap:**

* TreeMap is an implementation class of Map interface.
* TreeMap can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
* TreeMap is not synchronized.
* Map does not guarantee the order of insertion.
* TreeMap sorts the keys in natural order(ascending order).
* TreeMap allows null into keys in java 1.6 version, if it contains only one element and java 1.7 version does not allow null value even if the TreeSet contains one element. TreeMap allows null into value.

**Creation of TreeMap:**

**Syntax:**

1. TreeMap<K,V> tm = new TreeMap<K,V>();

The above syntax creates an empty map where the elements will be sorted in natural order.

1. TreeMap<K,V> tm = new TreeMap<K,V>(Comparator c);

The above syntax creates an empty map where the elements will be sorted according to the specified comparator.

1. TreeMap<K,V> tm = new TreeMap<K,V>(Map m);

The above syntax creates a map with the elements that are available in the specified Collection.

Here, K represents the type of the key and V represents the type of the value.

**Program :**

import java.util.\*;

public class TreeMapDemo {

public static void main(String[] args) {

TreeMap<String, Integer> tm = new TreeMap<String, Integer>();

tm.put("mnop",40);

tm.put("abcd",80);

tm.put("pqrs",70);

tm.put("qwer",60);

tm.put("stuv",50);

tm.put("ghij",30);

System.out.println("Elements : "+tm);

}

}

**Hashtable:**

* Hashtable is an implementation class of Map interface.
* Hashtable can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
* Hashtable does not allow null in both keys and values.
* Hashtable is synchronized.
* Hashtable does not guarantee the order of insertion.

**Creation of Hashtable:**

**Syntax:**

1. Hashtable<K,V> ht = new Hashtable<K,V>();

The above syntax creates an empty map with the default initial capacity as 11.

1. Hashtable<K,V> ht = new Hashtable<K,V>(int initialCapacity);

The above syntax creates an empty map with the specified capacity as the default initial capacity.

1. Hashtable<K,V> ht = new Hashtable<K,V>(Map m);

The above syntax creates an empty map with the elements that are available in the specified Map.

Here, K represents the type of the key and V represents the type of the value.

**Program :**

import java.util.\*;

public class HashtableDemo {

public static void main(String[] args) {

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

ht.put("mnop",40);

ht.put("abcd",80);

ht.put("pqrs",70);

ht.put("qwer",60);

ht.put("stuv",50);

ht.put("ghij",30);

System.out.println("Elements : "+ht);

Enumeration e = ht.keys();

while(e.hasMoreElements()) {

System.out.println(e.nextElement());

}

}

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HashMap | LinkedHashMap | TreeMap | Hashtable |
| Ordered | Unordered | Ordered by  Insertion | Sorted Order | Unordered |
| Duplicates keys  values | Not Allowed  Allowed | Not Allowed  Allowed | Not Allowed  Allowed | Not Allowed  Allowed |
| Null value | Allowed | Allowed | Allowed | Not Allowed |
| Synchronized | Not Synchronized | Not  Synchronized | Not Synchronized | Synchronized |
| Data Structure | Hashtable | Hashtable + Double Linked list | Red-Black Tree | Hashtable |
| Initial Capacity | 16 | 16 | ----- | 11 |

**Differences between the Map implementation classes**

**Cursors of collection framework:** The cursors available in java.util package can be used for accessing the elements one by one and perform some other operations. There are 3 cursors and they are:

1. Iterator
2. ListIterator
3. Enumeration

**Iterator:**

* Iterator is an interface.
* This cursor can be used for accessing the elements one by one.
* This cursor can be applied to all the classes that implement Collection interface.
* Iterator can be used for accessing the elements in forward direction only.
* Iterator can be used for performing an additional task like removing the elements.

Iterator interface contains 3 methods and they are**:**

1. boolean hasNext()
2. Object next()
3. void remove()

**Program :**

import java.util.\*;

public class IteratorDemo {

public static void main(String[] args) {

ArrayList<Integer> al = new ArrayList<Integer>();

al.add(45);

al.add(23);

al.add(67);

al.add(12);

al.add(56);

al.add(34);

al.add(78);

Iterator it = al.iterator();

while(it.hasNext()) {

Object obj = it.next();

Integer i = (Integer) obj;

if(i==34 | i==78)

it.remove();

}

System.out.println(al);

}

}

**ListIterator:**

* ListIterator is an interface.
* This cursor can be used for accessing the elements one by one.
* This cursor can be applied to all the classes that implement List interface. ListIterator can be used for accessing the elements in both forward and backward directions.
* ListIterator can be used for performing an additional tasks like removing, adding and replacing the elements.

ListIterator interface contains 9 methods and they are**:**

1. boolean hasNext()
2. Object next()
3. boolean hasPrevious()
4. Object previous()
5. int nextIndex()
6. int previousIndex()
7. void remove()
8. void set(Object)
9. void add(Object)

**Program :**

import java.util.\*;

public class ListIteratorDemo {

public static void main(String[] args) {

LinkedList<Integer> ll = new LinkedList<Integer>();

ll.add(45);

ll.add(23);

ll.add(67);

ll.add(12);

ll.add(56);

ll.add(34);

ll.add(78);

ListIterator<Integer> lit = ll.listIterator();

while(lit.hasNext()) {

Object obj = lit.next();

Integer i = (Integer) obj;

if(i==56)

lit.remove();

if(i==45)

lit.set(99);

if(i==12)

lit.add(22);

}

while(lit.hasPrevious()) {

System.out.println(lit.previous());

}

}

}

**Enumeration:**

* Enumeration is an interface.
* This cursor can be used for accessing the elements one by one.
* This cursor can be applied to all the legacy classes.
* Enumeration can be used for accessing the elements in forward direction only.

Enumeration interface contains 2 methods and they are**:**

1. boolean hasMoreElements()
2. Object nextElement()

**Program :**

import java.util.\*;

public class EnumerationDemo {

public static void main(String[] args) {

Vector<Integer> v = new Vector<Integer>();

v.add(45);

v.add(23);

v.add(67);

v.add(12);

v.add(56);

v.add(34);

v.add(78);

Enumeration e = v.elements();

while(e.hasMoreElements()) {

System.out.println(e.nextElement());

}

}

}

**Differences between the Cursors**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Iterator | ListIterator | Enumeration |
| Applicable to | Classes implementing Collection interface | Classes implementing List interface | Legacy classes |
| Accessibility | Forward direction | Forward and backward direction | Forward direction |
| Operations performed | Accessing  Removing | Accessing, Removing  Replacing, Adding | Accessing |
| Method to access | iterator() | listIterator() | elements() |
| Number of methods | 3 | 9 | 2 |

**Collections:** This class belongs to collection framework which consists exclusively of static methods that operate on collections.

**Methods of Collections class:**

1. **void sort(List):** This method is used to sort the elements available in the List class.
2. **int binarySearch(List, Object):** This method is used to perform binary search operation on the elements of the List.
3. **void reverse(List):** This method is used to reverse the elements available in the List.
4. **void swap(List, int index1, int index2):** This method is used to swap the elements available in the specified index positions.
5. **void copy(List, List):** This method is used to copy the elements of one List to another List.
6. **Object min(Collection):** This method will return the smallest element available in the specified Collection.
7. **Object max(Collection):** This method will return the biggest element available in the specified Collection.
8. **List synchronizedList(List):** This method is used to convert an unsynchronized list to synchronized list.
9. **Set synchronizedSet(Set):** This method is used to convert an unsynchronized set to synchronized set.
10. **Map synchronizedMap(Map):** This method is used to convert an unsynchronized map to synchronized map.

**Program :**

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] ar) {

ArrayList<Integer> al = new ArrayList<Integer>();

al.add(56);

al.add(23);

al.add(67);

al.add(89);

al.add(34);

al.add(78);

al.add(12);

al.add(45);

System.out.println("List : "+al);

Collections.reverse(al);

System.out.println("List : "+al);

Collections.sort(al);

System.out.println("List : "+al);

Collections.swap(al,1,4);

System.out.println("List : "+al);

System.out.println(Collections.min(al));

System.out.println(Collections.max(al));

List l = Collections.synchronizedList(al);

System.out.println("List : "+l);

}

}

**Arrays:** This class is a special class available in java.util package, using which we can perform some operations on the Arrays.

**Methods of Arrays:**

1. **void sort(Object[]):** This method can be used to sort the elements in the specified array in natural order.
2. **void sort(Object[], int index, int offset):** This method can be used to sort the elements in the specified range of the array in natural order.
3. **int binarySearch(Object[], Object):** This method can be used to search the specified array of objects for the specified value using the binary search algorithm.
4. **List asList(Object[]):** This method can be used to convert the elements of an array into a List.

**Program :**

import java.util.\*;

public class ArraysDemo {

public static void main(String[] ar) {

Integer[] arr = {3,6,1,7,2,4,8,5};

for(Integer x : arr) {

System.out.println(x);

}

Arrays.sort(arr);

for(Integer x : arr) {

System.out.println(x);

}

List l = Arrays.asList(arr);

System.out.println(l);

}

}

**NavigableSet:** This interface is introduced in java 1.6 version and it provides some special methods, which are designed for accessing the elements from the TreeSet class.

**Methods of NavigableSet:**

1. **Object lower(Object element):** This method will return the highest element which is smaller than the specified element.
2. **Object floor(Object element):** This method will return the highest element which is smaller than or equal to the specified element.
3. **Object higher(Object element):** This method will return the smallest element which is bigger than the specified element.
4. **Object ceiling(Object element):** This method will return the smallest element which is bigger than or equal to the specified element.
5. **Object pollFirst():** This method will delete the first element from the set.
6. **Object pollLast():** This method will delete the last element from the set.
7. **NavigableSet descendingSet():** This method will sort the elements in descending order.
8. **SortedSet subSet(Object begin, Object end):** This method will return a group of elements starting from the specified begin element upto the specified end element. The begin element will be included and the end element will be excluded.
9. **SortedSet headSet(Object end):** This method will return a group of elements starting from the first element upto the specified end element. The specified end element will be excluded.
10. **SortedSet tailSet(Object begin):** This method will return a group of elements starting from the specified begin element upto the last element. The specified begin element will be included.

**Program :**

import java.util.\*;

public class NavigableSetDemo {

public static void main(String[] args) {

TreeSet<Integer> ts = new TreeSet<Integer>();

ts.add(56);

ts.add(89);

ts.add(67);

ts.add(45);

ts.add(23);

ts.add(78);

ts.add(12);

ts.add(34);

System.out.println("set : "+ts);

System.out.println(ts.lower(56));

System.out.println(ts.floor(56));

System.out.println(ts.higher(56));

System.out.println(ts.ceiling(56));

System.out.println(ts.subSet(45,78));

System.out.println(ts.headSet(56));

System.out.println(ts.tailSet(56));

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

}

}

**NavigableMap:** This interface is introduced in java 1.6 version and it provides some special methods, which are designed for accessing the elements from the TreeMap class.

**Methods of NavigableMap:**

1. **Entry lowerEntry(Object key):** This method will return the highest key and its corresponding value which is smaller than the specified element.
2. **Entry floorEntry(Object key):** This method will return the highest key and its corresponding value which is smaller than or equal to the specified element.
3. **Entry higherEntry(Object key):** This method will return the smallest key and its corresponding value which is bigger than the specified element.
4. **Entry ceilingEntry(Object key):** This method will return the smallest key and its corresponding value which is bigger than or equal to the specified element.
5. **Object lowerKey(Object key):** This method will return the highest key which is smaller than the specified element.
6. **Object floorKey(Object key):** This method will return the highest key which is smaller than or equal to the specified element.
7. **Object higherKey(Object key):** This method will return the smallest key which is bigger than the specified element.
8. **Object ceilingKey(Object key):** This method will return the smallest key which is bigger than or equal to the specified element.
9. **Entry firstEntry():** This method will return the first entry available in the map.
10. **Entry lastEntry():** This method will return the last entry available in the map.
11. **Entry pollFirstEntry():** This method will delete the first entry available in the map.
12. **Entry pollLastEntry():**This method will delete the last entry available in the map.
13. **Entry descendingMap():**This method will sort the elements in descending order.
14. **Entry subMap(Object begin, Object end):** This method will return a group of elements starting from the specified begin element upto the specified end element. The begin element will be included and the end element will be excluded.
15. **Entry headMap(Object end):** This method will return a group of elements starting from the first element upto the specified end element. The specified end element will be excluded.
16. **Entry tailMap(Object begin):** This method will return a group of elements starting from the specified begin element upto the last element. The specified begin element will be included.

**Program :**

import java.util.\*;

public class NavigableMapDemo {

public static void main(String[] args){

TreeMap<Integer,Integer> tm = new TreeMap<Integer,Integer>();

tm.put(56,40);

tm.put(89,10);

tm.put(67,50);

tm.put(45,80);

tm.put(23,70);

tm.put(78,30);

tm.put(12,60);

tm.put(34,20);

System.out.println("map : "+tm);

System.out.println(tm.lowerEntry(56));

System.out.println(tm.floorEntry(56));

System.out.println(tm.higherEntry(56));

System.out.println(tm.ceilingEntry(56));

System.out.println(tm.lowerKey(56));

System.out.println(tm.floorKey(56));

System.out.println(tm.higherKey(56));

System.out.println(tm.ceilingKey(56));

System.out.println(tm.firstEntry());

System.out.println(tm.pollLastEntry());

System.out.println(tm.subMap(45,78));

System.out.println(tm.headMap(56));

System.out.println(tm.tailMap(56));

System.out.println(tm.descendingMap());

}

}

**StringTokenizer**: This class is used to break a string into tokens (pieces).

**Syntax**: StringTokenizer st = new StringTokenizer(String, delimiter);

**Program :**

import java.util.\*;

public class StringTokenizerDemo {

public static void main(String[] args) {

String str = "oneatwoathreeafour8five8six8seven#eight#nine#ten";

StringTokenizer st = new StringTokenizer(str,"a8#");

System.out.println(st.countTokens());

while (st.hasMoreTokens()) {

System.out.println(st.nextToken());

}

}

}

**Date:** The Date class represents a specific instance of time.

**Calendar:** The Calendar class is an abstract class. It provides some methods for converting an instance of a time to day, month, year, second, minute, hour etc.

We cannot create an object of Calendar class directly. We can get the object of Calendar class by using a static method called getInstance() of Calendar class.

**Program :**

import java.util.\*;

public class DateDemo {

public static void main(String[] args) {

Date d = new Date();

System.out.println("Date: "+d);

Calendar c = Calendar.getInstance();

int date = c.get(Calendar.DATE);

int month = c.get(Calendar.MONTH);

int year = c.get(Calendar.YEAR);

System.out.println("Date: "+date+ "/" +(++month)+ "/" +year);

int hour = c.get(Calendar.HOUR);

int minute = c.get(Calendar.MINUTE);

int second = c.get(Calendar.SECOND);

System.out.println("Time = "+hour+":"+minute+":"+second);

}

}

**Generics**

Generics is a concept introduced in the java 1.5 version. Generics are called as **parameterized types**.

The generics are represented by a pair of angular brackets (<>), called as diamond operator.

Generics are designed to provide compile time **type safety**, which will reduce the need for typecasting.

Generics are said to be **type erasures**, which means the generic type information will be available only up to the compilation time, once the compilation is done the generic type information will be erased.

**Procedure to create a generic method:** To create a generic method, we need to specify the generic type parameter before the return type of the method.

**Syntax:**

<E> returntype methodName() {

}

E represents the generic type parameter.

**Program :**

public class GenericMethod {

public <T> void display(T[] temp) {

for(T x : temp) {

System.out.println(x);

}

}

public static void main(String[] args) {

GenericMethod gm = new GenericMethod();

Integer[] iarr = {1,2,3,4,5};

gm.display(iarr);

Double[] darr = {1.2,2.3,3.4,4.5,5.6};

gm.display(darr);

String[] sarr = {"abc","def","ghi","stu","xyz"};

gm.display(sarr);

}

}

**Procedure to create Generic class:** Declare the Generic type parameter after the class declaration.

**Syntax:** class ClassName<E> {

}

**Program :**

class MyClass<T> {

T obj;

MyClass(T obj) {

this.obj = obj;

}

T getValue(){

return obj;

}

public void showType() {

System.out.println("type : " +obj.getClass().getName());

}

}

public class GenericDemo{

public static void main(String[] args){

Integer iobj = new Integer(123);

MyClass<Integer> mc1 = new MyClass<Integer>(iobj);

System.out.println("value : " +mc1.getValue());

mc1.showType();

Double dobj = new Double(34.5);

MyClass<Double> mc2 = new MyClass<Double>(dobj);

System.out.println("value : " +mc2.getValue());

mc2.showType();

String sobj = new String("java");

MyClass<String> mc3 = new MyClass<String>(sobj);

System.out.println("value : " +mc3.getValue());

mc3.showType();

}

}

**Procedure to create Generic interface:** Declare the Generic type parameter after the interface declaration.

**Syntax:** interface InterfaceName<E> {

}

**Program :**

interface MyInterface<A,B> {

void add(A a, B b);

}

public class GenericInterface1 implements MyInterface<String, String> {

public void add(String s1, String s2) {

System.out.println(s1+s2);

}

public static void main(String[] args) {

GenericInterface1 gi1 = new GenericInterface1();

gi1.add("hello","friends");

}

}

public class GenericInterface2 implements MyInterface<Integer, Integer> {

public void add(Integer i1, Integer i2) {

System.out.println(i1+i2);

}

public static void main(String[] args){

GenericInterface2 gi2 = new GenericInterface2();

gi2.add(12,34);

}

}