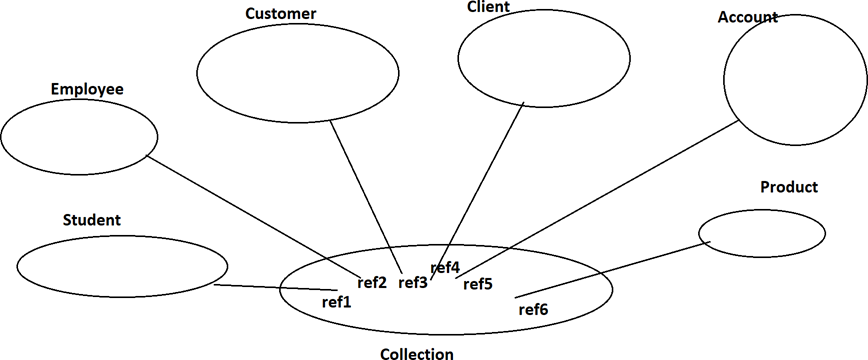
**Collection Framework**

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

Collection is an object, it is able to manage a group of other objects, here Collection is able to store the references of other Objects.



**Q)** **In Java applications , to manage a group of Objects we have already Arrays then what is the requirement to use Collections?**

**Ans:**

**—---**

1. Arrays are fixed size in nature, arrays are able to allow elements up to its max size, if we add any element over its size then JVM will raise an exception like java.lang.ArrayIndexOutOfBoundsException.

EX:

Employee[] emps = new Employee[5];

emps[0] = new Employee();

emps[1] = new Employee();

emps[2] = new Employee();

emps[3] = new Employee();

emps[4] = new Employee();

emps[5] = new Employee(); —--> java.lang.ArrayIndexOutOfBoundsException

Collections are dynamically Growable in nature, that is, in the case of Collections even if we add elements over its size there JVM will not raise any Exception , where Collections are able to allow extra elements by increasing its size automatically.

EX: ArrayList al = new ArrayList(5);

al.add(new Employee());

al.add(new Employee());

al.add(new Employee());

al.add(new Employee());

al.add(new Employee());

al.add(new Employee()); —-> No Exception, elements are allowed by increasing

ArrayList size dynamically.

2. Arrays are able to allow only Homogeneous elements , that is, arrays are able to allow only one type of elements, if we add any other type of element then the compiler will raise an error like Incompatible Types Error.

EX:

Employee[] emps = new Employee[12];

emps[0] = new Employee();

emps[1] = new Employee();

—---

emps[9] = new Employee();

emps[10] = new Product(); —-> Error: Incompatible Types.

Collection objects are able to allow Heterogeneous elements , that is, Collection objects are able to allow different types of elements .

EX: ArrayList al = new ArrayList();

al.add(new Employee());

al.add(new Student()); —--> No Error

al.add(new Customer());---> No Error

al.add(new Client()); —---> No Error

3. Arrays do not have predefined libraries to perform the operations like Sorting, if we want to perform sorting over the elements we must write logic for the sorting explicitly.

EX:

String[] str = {“AAA”,”FFF”,”BBB”,”EEE”,”CCC”,”DDD”};

We have to write for the algorithm up to 20 loc

Collections have predefined libraries to perform the operations like Sorting, no need to write logic explicitly.

EX:

TreeSet ts = new TreeSet();

ts.add(“AAA”);

ts.add(“FFF”);

ts.add(“BBB”);

ts.add(“EEE”);

ts.add(“CCC”);

ts.add(“DDD”);l

Sopln(ts); // [AAA, BBB, CCC, DDD, EEE, FFF]

4. Arrays are not more flexible than Collections.

5. If we know the number of elements and their type which we want to represent at the time of writing a program then it is suggestible to take Arrays , not Collection, because Arrays are able to provide better performance when compared with Collections. If we don't know the number of elements and their types at the time of writing a program then it is suggestible to take Collections.

6. Arrays are less API dependent, so debugging and testing are simple.

Collections are more API dependent, so debugging and testing are more difficult.

7. Arrays are able to improve Typedness, because arrays allow only Homogeneous elements.

Collections are able to reduce typedness , because Collections allow heterogeneous elements.

8. In case of Arrays we are able to perform Typesafe operations.

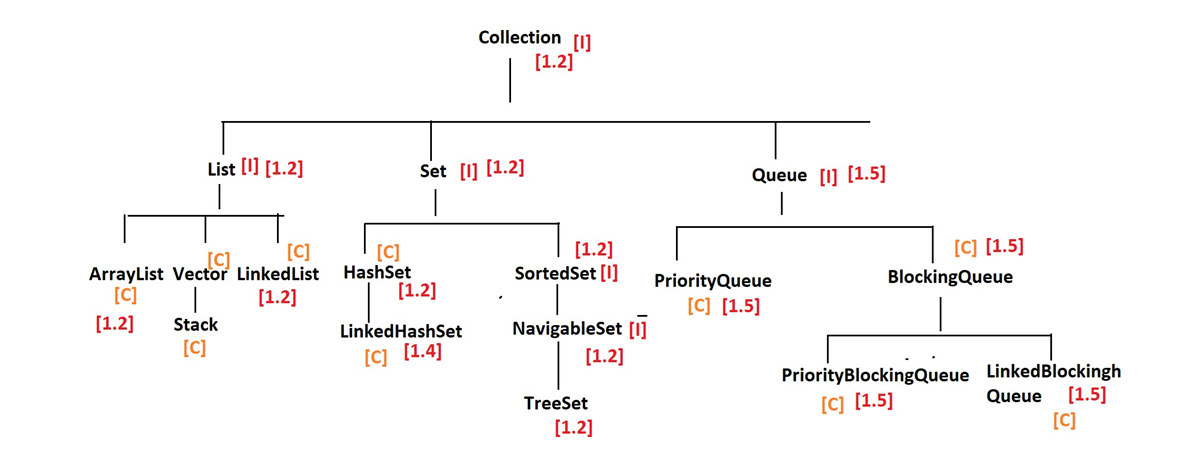
In the case of Collections we are able to perform Typeunsafe operations.

9. Arrays are Lightweight. Collections are Heavy weight.

To represent Collections in Java applications , JAVA has provided a set of predefined classes and interfaces in a separate package that is “java.util”.

Java has provided a set of predefined classes and interfaces to represent collections , here the set of predefined classes and interfaces called “Collection Framework”.

IN Java, Collection Framework has provided the following classes and interfaces.



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

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

"Collection is an "interface" which can be used to represent a group of

objects as a single entity. Where as "Collections is an utility class"

present in java.util package to define several utility methods for

Collection objects.

Collection--------------------interface

Collections------------------class

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

**—-------------------------------**

**Ans:**

**—---**

Collection is able to manage a group of individual objects.

EX: If we want to manage 100 Employee objects we have to use Collection.

Map is able to manage all the elements in the form of “Key-Value” pairs.

EX:

To manage the Student registry , where to manage Students data like Roll number and Student name we have to use Map.

To manage the Telephone Directory, where we have to represent Telephone number and customer details we have to use Map.

**Q) What are the differences between List and Set?**

**—----------------------------------**

**Ans:**

—---

1. List is index based, every element must be stored along with an index value.

Set is not index based, every element will not be stored with index, but every element will be stored on the basis of hashcode values of the objects.

2. List allows duplicate elements, recognizing the duplicate elements on the basis of indexes.

Set does not allow duplicate elements, because object hashcode values never be duplicated.

3. List follows insertion order.

Set does not follow insertion order.

Note: LinkedHashSet is able to follow insertion order.

4. Both List and Set are not following Sorting order.

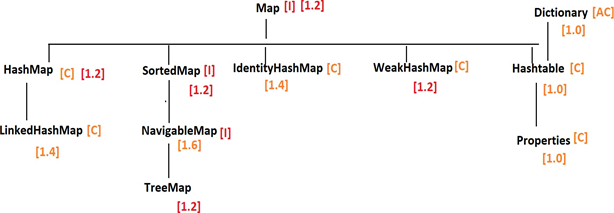
Note: All implementations of Set are not following Sorting order, but the Set implementations like SortedSet, NavigableSet and TreeSet are following Sorting order.

5. List allows heterogeneous elements. Set allows heterogeneous elements.

Note: SortedSet, NavigableSet and TreeSet are allowing Homogeneous elements only, not allowing heterogeneous elements.

6. List allows any number of null elements. Set allows only one null element.

Note: SortedSet, NavigableSet and TreeSet do not allow null elements.



**Collection:**

1. Collection is a root for all the Collections.

2. It has defined a number of methods to perform operations with the elements of the Collection objects.

**Methods:**

**1.public boolean add(Object obj)**

It will add the specified element to the Collection, if the element is added really then add() method will return true value, if the element is not added really then add() method will return false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.HashSet;

**public** **class** Main {

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

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

System.***out***.println(hashSet.add("AAA"));

hashSet.add("BBB");

hashSet.add("CCC");

hashSet.add("DDD");

System.***out***.println(hashSet.add("AAA"));

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

}

}

**2.** **public boolean addAll(Collection c):**

It is able to add all the elements of the specified Collection to the present collection. If the add operation is successful then addAll() method will return true value.

If the add operation is not successful then the addAll() method will return false value

. EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

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

hashSet.add("AAA");

hashSet.add("BBB");

hashSet.add("CCC");

hashSet.add("DDD");

hashSet.add("EEE");

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

Collection collection = **new** HashSet();

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

System.***out***.println(collection.addAll(hashSet));

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

System.***out***.println(collection.addAll(hashSet));

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

}

}

In the case of addAll() method, if at least one element is added then addAll() method will return true value, if all elements are not added then addAll() method will return false value.

**3.** **public boolean remove(Object obj)**

It is able to remove the specified element from the Collection, if the remove operation is successful then remove() method will return true value, if the remove operation is not successful then remove() method will return false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

System.***out***.println(collection1);

System.***out***.println(collection1.remove("BBB"));

System.***out***.println(collection1);

System.***out***.println(collection1.remove("BBB"));

System.***out***.println(collection1);

}

}

**4.** **public boolean removeAll(Collection collection):**

It is able to remove all the elements of the specified Collection from the original collection object. If removeALl operation is successful then removeALl() method will return true value otherwise false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

Collection collection2 = **new** ArrayList();

collection2.add("BBB");

collection2.add("CCC");

collection2.add("DDD");

System.***out***.println(collection2);

System.***out***.println(collection1.removeAll(collection2));

System.***out***.println(collection1);

System.***out***.println(collection1.removeAll(collection2));

System.***out***.println(collection1);

}

}

In removeAll operation, if at least one element is removed then removeALl() method will return true value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

Collection collection2 = **new** ArrayList();

collection2.add("BBB");

collection2.add("CCC");

collection2.add("XXX");

collection2.add("YYY");

System.***out***.println(collection2);

System.***out***.println(collection1.removeAll(collection2));

System.***out***.println(collection1);

}

}

**5.** **public boolean contains(Object obj)**

It is able to check whether the specified element exists or not in the Collection. If the specified element exists in the collection then contains() method will return true value, if the specified element does not exist in the collection then contains() method will return false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

System.***out***.println(collection1.contains("BBB"));

System.***out***.println(collection1.contains("ZZZ"));

}

}

**6.** **public boolean containsAll(Collection collection)**

It is able to check whether all the elements of the specified collection exist or not in the present Collection object. If all the elements exist then containsAll() method will return true value otherwise containsAll() method will return false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

Collection collection2 = **new** ArrayList();

collection2.add("BBB");

collection2.add("CCC");

collection2.add("DDD");

System.***out***.println(collection2);

System.***out***.println(collection1.containsAll(collection2));

}

}

In the case of containsAll method , if at least one element is not matched with all the elements of COllection1 then containsAll() method will return false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

Collection collection2 = **new** ArrayList();

collection2.add("BBB");

collection2.add("CCC");

collection2.add("XXX");

collection2.add("YYY");

System.***out***.println(collection2);

System.***out***.println(collection1.containsAll(collection2));

}

}

**7.** **public boolean retainAll(Collection c):**

It is able to remove all the elements of the Collection which are not matched with the elements of the provided Collection, if at least one element is removed from the collection then retainAll() method will return true value otherwise false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

Collection collection2 = **new** ArrayList();

collection2.add("BBB");

collection2.add("CCC");

collection2.add("DDD");

collection2.add("EEE");

System.***out***.println(collection2);

System.***out***.println(collection1.retainAll(collection2));

System.***out***.println(collection1);

System.***out***.println(collection1.retainAll(collection2));

System.***out***.println(collection1);

}

}

**8.** **public boolean isEmpty():**

It is able to check whether the provided Collection is empty or not, if the Collection is empty then isEmpty() method will return true value otherwise false value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

System.***out***.println(collection1.isEmpty());// false Collection collection2 = new ArrayList();

// System.out.println(collection2.isEmpty());//true

}

}

**9.** **public int size():**

It is able to return the size of the Collection that is the number of elements existing in the collection.

Ex:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

System.***out***.println(collection1.size());

}

}

**10.** **public void clear():**

It is able to remove all elements from the Collection: EX:

—-

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

collection1.clear();

System.***out***.println(collection1);

}

}

**11.** **public Object[] toArray():**

It is able to convert all elements from Collection to an Object[]. EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**public** **class** Main {

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

Collection collection1 = **new** ArrayList();

collection1.add("AAA");

collection1.add("BBB");

collection1.add("CCC");

collection1.add("DDD");

collection1.add("EEE");

collection1.add("FFF");

System.***out***.println(collection1);

Object[] elements = collection1.toArray();

**for** (Object obj : elements) {

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

}

}

}

**List**

**—------------**

1. It is an interface provided by JAVA in JDK1.2 version.

2. It is index based, it is able to store all elements as per indexing.

3. It allows duplicate elements.

4. It follows insertion order.

5. It does not follow sorting order.

6. It allows heterogeneous elements.

7. It allows any number of null elements.

8. It is a child interface to Collection interface.

**Methods:**

**1.** **public void add(int index, Object element):**

It is able to add the specified element at the specified index value.

Here the provided index value is in out of the indexes then JVM will raise an exception like “java.lang.IndexoutOfBoundsException”.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");

list.add("BBB");

list.add("CCC");

list.add("DDD");

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

list.add(4, "EEE");

list.add(5, "FFF");

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

//list.add(8, "GGG"); --> IndexOutOfBoundsException

}

}

**2.** **public void addAll(int index, Collection c):**

It is able to add all the elements of the specified Collection to the List at the specified index value. If the provided index value is in outside range of the List then JVM will raise an exception like java.lang.IndexOutOfBoundsException.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");

list.add("BBB");

list.add("CCC");

list.add("DDD");

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

List list1 = **new** ArrayList();

list1.add("XXX");// 0

list1.add("YYY");// 1

System.***out***.println(list1);

list1.addAll(2, list);

System.***out***.println(list1);

//list1.addAll(10, list);-->IndexOutOfBoundsException System.out.println(list1);

}

}

**3.** **public Object remove(int index)**

It is able to remove the element that existed at the specified index value, if the specified index is outside of the indexes range then JVM will raise an exception like java.lng.IndexoutOfBoundsException.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");

list.add("BBB");

list.add("CCC");

list.add("DDD");

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

System.***out***.println(list.remove(2));

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

//list.remove(10); --> IndexOutOfBoundsException

}

}

**4.** **public Object set(int index, Object element):**

It is able to set the specified element at the specified index value, If any element exists already at the specified index then JVM will replace the existing element with the new element.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");// 0

list.add("BBB");// 1

list.add("CCC");// 2

list.add("DDD");// 3 System.out.println(list); System.out.println(list.set(2, "**XXX**"));

// System.out.println(list);

}

}

**Q)** **What is the difference between add() method and set() method in List?**

**Ans:**

**—--**

In List, add() method is able to perform the following operations.

1. If the specified index value is the last index value then it will append a new element to the List as the last element.

2. If the specified index value is in the middle of List , that is, if any element exists at the specified index value then the add() method will insert the new element at the specified index and the existing element will be adjusted to the next index value.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");// 0

list.add("BBB");// 1

list.add("CCC");// 2

list.add("DDD");

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

list.add(4, "EEE");

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

list.add(2, "XXX");

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

}

}

set() method is able to override an element with the new element, It will not perform append operation to the list, it will not perform insertion operation, That is, in the case of set() method if any element is existed at the specified index then set() method will replace the existed element with new element, if no element is existed at the specified index value then JVM will raise an exception like java.lang.IndeOutOfBoundsException.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");// 0

list.add("BBB");// 1

list.add("CCC");// 2

list.add("DDD");// 3 System.out.println(list);

//list.set(4, "EEE"); --> IndexOutOfBoundsException

list.set(2, "XXX");

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

}

}

Note: add() method return type is void, but set() method return type is Object , that is, the element which is removed in override operation that element will be returned by set() method.

**6.** **public Object get(int index):**

It is able to get the element which is existing at the specified index value. If the provided index value is in outside of the indexes range then JVM will raise an exception like java.lang.IndexOutOfBoundsException.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");// 0

list.add("CCC");// 2

list.add("DDD");// 3 System.out.println(list);

//System.out.println(list.get(4));--> IndexOutOfBoundsException

}

}

**7.** **public int indexOf(Object element):**

It is able to return an index value where the first occurence of the specified element exists.

**8.** **public int lastindex(Object element):**

It is able to return an index value where the last occurence of the specified element exists.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");// 0

list.add("BBB");// 1

list.add("CCC");// 2

list.add("DDD");// 3

list.add("BBB");// 4

list.add("EEE");// 5

list.add("FFF");// 6

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

System.***out***.println(list.indexOf("BBB"));// 1 System.out.println(list.lastIndexOf("BBB"));//7

}

}

**ArrayList:**

**—------------**

1. It was introduced in JDK1.2 version

2. It is not deprecated/Legacy Collection.

3. It is an implementation class to List interface.

4. It is index based, it allows all the elements as per indexing.

5. It follows insertion order.

6. It does not follow Sorting order.

7. It allows heterogeneous elements.

8. It allows Duplicate elements.

9. It allows any number of null elements.

10. It is not Synchronized Collection

11. It allows more than one thread at a time.

12. It follows parallel execution over the threads.

13. It reduces application execution time.

14. It increases application performance.

15. It does not give guarantee for data consistency.

16. It is not a thread safe resource.

17. It is suggestible when we want to perform frequent retrieval operations.

18. Its default capacity is 10 elements.

19. Its incremental Capacity is “(currentCapacity\*3/2)+1”.

20. Its internal Data Structure is “Resizable Array”.

In Java applications, when we add an element over the max capacity of the ArrayList then ArrayList will perform the following actions.

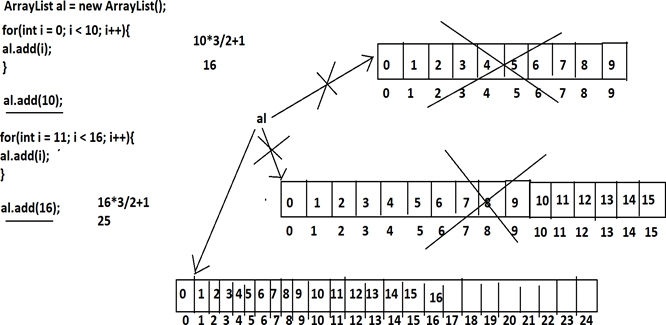
1. It will calculate Incremental capacity value with the incremental ratio. CurrentCapacity\*3/2+1

2. It will create a new Array of the incremental Capacity size.

3. It will copy all the elements from the old array to the new array.

4. ArrayList will change array references to new Array from old array.

5. It will make an eligible old Array for Garbage Collection.



**Constructors:**

**1. public ArrayList():**

It is able to create an empty ArrayList object with the initial capacity of 10 elements.

EX: ArrayList al = new ArrayList();

**2. public ArrayList(int capacity):**

It is able to create an empty Arraylist object with the specified default capacity.

EX:

ArrayList arrayList = new ArrayList(20); System.out.println(arrayList);

**3. public ArrayList(Collection collection)**

It is able to create an ArrayList object with all the elements of the specified Collection.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.List;

**public** **class** Main {

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

Collection collection = **new** HashSet();

collection.add("AAA");

collection.add("BBB");

collection.add("CCC");

collection.add("DDD");

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

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

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

}

}

Example program

==================

package com.codegnan.collections

import java.util.ArrayList;

import java.util.Collections;

public class ArrayListExample {

public static void main(String[] args) {

// Create an ArrayList to hold integers

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

// Add numbers to the ArrayList

numbers.add(10);

numbers.add(20);

numbers.add(30);

numbers.add(40);

numbers.add(50);

// Display the original list

System.out.println("Original List: " + numbers);

// Remove a number from the list

numbers.remove(Integer.valueOf(30)); // Remove by value

System.out.println("After removing 30: " + numbers);

// Check if the list contains a specific number

if (numbers.contains(20)) {

System.out.println("List contains 20.");

} else {

System.out.println("List does not contain 20.");

}

// Get the index of a specific number

int index = numbers.indexOf(40);

if (index != -1) {

System.out.println("40 is at index: " + index);

}

// Sort the list

Collections.sort(numbers);

System.out.println("Sorted List: " + numbers);

// Clear the list

numbers.clear();

System.out.println("List after clearing: " + numbers);

}

}

**LinkedList:**

**— ---------**

1. It was introduced in JDK1.2 version.

2. It is not a Legacy Collection.

3. It is an implementation class to List interface.

4. It is index based, it allows all the elements as per indexing.

5. It follows Insertion order.

6. It does not follow Sorting order.

7. It allows duplicate elements.

8. It allows Heterogeneous elements.

9. It allows more than one null element.

10. It is not a Synchronized Collection.

11. It is able to allow more than one thread at a time to access data.

12. It is following Parallel execution.

13. It reduces application execution time.

14. It increases application performance.

15. It is not a guarantee for Data Consistency.

16. It is not Thread Safe.

17. It is mainly for frequent insertions and deletions.

18. Its internal data structure is Double Linked List.

**Constructors:**

**— -----------**

**1. public LinkedList():**

It is able to create an empty LinkedList object.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.LinkedList;

**public** **class** Main {

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

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

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

}

}

**1.** **public LinkedList(Collection c)**

It is able to create a LinkedList object with all the elements of the specified Collection.

EX

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

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

arrayList.add("AAA");

arrayList.add("BBB");

arrayList.add("CCC");

arrayList.add("DDD");

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

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

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

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

Set set = **new** HashSet();

set.add("XXX");

set.add("YYY");

set.add("ZZZ");

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

LinkedList linkedList1 = **new** LinkedList(set);

System.***out***.println(linkedList1);

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

Queue queue = **new** PriorityQueue();

queue.add(111);

queue.add(222);

queue.add(333);

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

LinkedList linkedList2 = **new** LinkedList(queue);

System.***out***.println(linkedList2);

}

}

**Methods:**

**— ----------**

**1. public void addFirst(Object obj)**

It can be used to add the specified element as First element to the LinkedList.

**2. public void addLast(Object obj)**

It can be used to add the specified element as Last element.

**3. public Object getFirst()**

It can be used to get the First element from LinkedList.

**4. public Object getLast()**

It can be used to get Last Element from LinkedList.

**5. public Object removeFirst()**

It can be used to remove the first element from the LinkedList.

**6. public Object removeLast()**

It can be used to remove the last element from LinkedList.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

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

linkedList.add("AAA");

linkedList.add("BBB");

linkedList.add("CCC");

linkedList.add("DDD");

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

linkedList.addFirst("XXX");

linkedList.addLast("YYY");

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

System.***out***.println(linkedList.getFirst());

System.***out***.println(linkedList.getLast());

System.***out***.println(linkedList.removeFirst());

System.***out***.println(linkedList.removeLast());

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

}

}

OP:

[AAA, BBB, CCC, DDD]

[AAA, BBB, CCC, DDD]

[YYY, XXX, ZZZ]

[YYY, XXX, ZZZ]

[111, 222, 333]

[111, 222, 333]

OP:

—-

[AAA, BBB, CCC, DDD]

[XXX, AAA, BBB, CCC, DDD, YYY] XXX

YYY XXX YYY

[AAA, BBB, CCC, DDD]

**Q) What are the differences between ArrayList and LinkedList?**

**—--------------------------**

**Ans:**

**—---**

1. ArrayList is suggestible when we want to perform frequent retrieval operations. LinkedList is suggestible when we want to perform frequent insertion and deletions

2. ArrayList's internal data structure is “Resizable Array”. LinkedList internal Data structure is “Double LinkedList”.

3. Capacity manipulations are possible with ArrayList. Capacity Manipulations do not exist with LinkedList.

1. ArrayList is an implementation class to List, RandomAccess, Cloneable,Serializable LinkedList is an implementation class to List, Deque, Cloneable, Serializable

**Vector:**

**—--------**

1. It was introduced in the JDK1.0 version.

2. It is a Legacy Collection.

3. It is an implementation class to List interface.

4. It is index based, it allows all the elements as per indexing.

5. It follows insertion order.

6. It does not follow sorting order.

7. It allows duplicate elements.

8. It allows more than one null element.

9. It allows heterogeneous elements.

10. It is a Synchronized Collection.

11. It allows only one thread at a time.

12. It follows sequential execution.

13. It increases application execution time.

14. It reduces application performance.

15. It gives guarantee for Data consistency.

16. It is a thread safe collection.

17. It is suggestible when we have frequent retrieval operations.

18. Its initial capacity is 10 elements.

19. Its incremental capacity value is

New Capacity = 2\*CurrentCapacity;

20. Its internal Data Structure “Resizable Array”.

Constructors:

**1.** **public Vector():**

It is able to create an empty Vector object with 10 default capacity value.

EX:

**public** **class** Main {

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

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

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

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

}

}

2. public Vector(int capacity):

It is able to create an empty Vector object with the specified initial capacity.

EX:

package com.codegnan.collectionframeworks;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Iterator;

import java.util.List;

import java.util.ArrayList;

import java.util.Vector;

import java.util.ArrayList;

import java.util.Vector;

public class Main {

public static void main(String[] args) {

Vector vector = new Vector(20);

System.out.println(vector);

System.out.println(vector.capacity());

}

}

**3.** **public Vector(int capacity, int incrementalCapacity):**

It is able to create an empty Vector object with the specified capacity value and with the specified incremental capacity ratio

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Vector;

**public** **class** Main {

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

Vector vector = **new** Vector(5, 5);

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

**for** (**int** i = 0; i < 5; i++) {

vector.add(i);

}

vector.add(6);

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

**for** (**int** i = 7; i < 11; i++) {

vector.add(i);

}

vector.add(11);

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

}

}

5

10

15

**1.** **public Vector(Collection c):**

It is able to create a Vector object with all the elements of the specified Collection

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Vector;

**public** **class** Main {

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

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

arrayList.add("AAA");

arrayList.add("BBB");

arrayList.add("CCC");

arrayList.add("DDD");

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

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

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

}

}

**Methods:**

**1. public void addElement(Object element)**

It is able to add the specified element to the Vector.

**2. public Object elementAt(int index):**

It is able to return an element at the specified index value.

**3. public Object firstElement():**

It is able to get the first element from the Vector.

**4. public Object lastElement():**

It is able to get the last element from the Vector.

**5. public boolean removeElement(Object obj):**

It is able to remove the specified element from the Vector.

**6. public void removeElementAt(int index):**

It is able to remove an element at the specified index value.

**7. public void removeAllElements()**

It is able to remove all elements from Vector.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.Vector;

**public** **class** Main {

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

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

vector.addElement("AAA");

vector.addElement("BBB");

vector.addElement("CCC");

vector.addElement("DDD");

vector.addElement("EEE");

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

System.***out***.println(vector.elementAt(3));

System.***out***.println(vector.firstElement());

System.***out***.println(vector.lastElement());

System.***out***.println(vector.removeElement("BBB"));

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

System.***out***.println(vector.removeElement("BBB"));

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

vector.removeElementAt(2);

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

vector.removeAllElements();

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

}

}

OP:

[AAA, BBB, CCC, DDD, EEE] DDD

AAA EEE

true

[AAA, CCC, DDD, EEE]

false

[AAA, CCC, DDD, EEE] [AAA, CCC, EEE]

[]

**Q) What are the differences between ArrayList and Vector?**

**—-----------------------------------------------**

**Ans:**

**—---**

1. ArrayList was introduced in JDK1.2 version. Vector was introduced in JDK1.0 version.

2. ArrayList is not a Legacy Collection. Vector is a Legacy Collection.

3. ArrayList is not a Synchronized Collection. Vector is a Synchronized Collection.

4. ArrayList is able to allow more than one thread at a time. Vector is able to allow only one thread at a time.

5. ArrayList is able to follow Parallel execution over the threads. Vector is able to follow sequential execution over the threads.

6. ArrayList is able to reduce application execution time. Vector is able to increase application execution time.

7. ArrayList is able to increase application performance. Vector is able to reduce application performance.

8. ArrayList is not giving guarantees for Data consistency. Vector is giving guarantee for Data Consistency.

9. ArrayList is not Thread Safe. Vector is Thread safe.

10. ArrayList has the incremental capacity like below. NewCapacity = CurrentCapacity\*3/2+1

Vector incremental capacity like belo.

NewCapacity = 2\*CurrentCapacity

11. ArrayList does not allow customizations on incremental capacity value, we must follow the default implementation for the incremental capacity.

Vector allows customizations on incremental capacity , not mandatory to follow default implementation of incremental capacity value.

12. It is not possible to find the current capacity of the ArrayList, because ArrayList does not have a capacity() method.

It is possible to find the current capacity value of the Vector, because the Vector class has a capacity() method.

13. ArrayList has all the methods implementations which are declared in Collection interface and List interface, it does not have its own methods.

Vector has all the methods implementations which are declared in Collection and List interface and its own methods.

firstElement() lastElement() addElement() elementAt(int index)

removeElement(Object obj) removeElementAt(int index) removeAllElements() capacity()

**Stack:**

**— ------------**

1. It was introduced in the JDK1.0 version.

2. It is a Legacy Collection

3. It is a subclass of Vector.

4. To manage elements Stack is following “LIFO”.

**Constructors:**

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

**1. public Stack()**

It is able to create an empty Stack object.

**Methods:**

**1. public void push(Object obj)**

It is able to add the specified element to the stack as the top most element.

**2. public Object pop()**

It is able to read and remove the top of the stack.

**3. public Object peek():**

It is able to read the top of the stack.

**4. public int search(Object obj)**

It is able to search for the specified element in Stack, if the element exists in stack then its position value will be returned. If the element does not exist in the Stack then search() method will return -1 value.

EX:

—--

**package** com.codegnan.collectionframeworks;

**import** java.util.Stack;

**import** java.util.Stack;

**public** **class** Main {

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

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

stack.push("AAA");

stack.push("BBB");

stack.push("CCC");

stack.push("DDD");

stack.push("EEE");

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

System.***out***.println(stack.pop());

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

System.***out***.println(stack.peek());

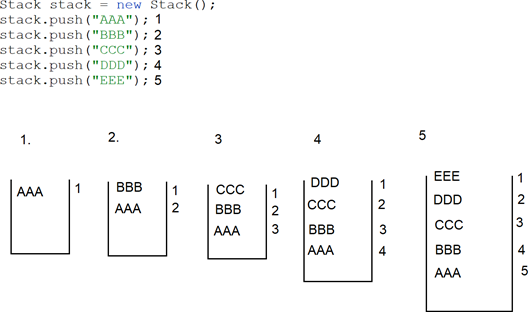
System.***out***.println(stack.search("AAA"));

System.***out***.println(stack.search("DDD"));

System.***out***.println(stack.search("EEE"));

}

}



**Iterators / Cursors in Collections:**

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

—

IN Java applications, when we pass a particular class reference variable as parameter to System.out.println() method, JVM will execute toString() method over the provided reference variable, to execute toString() method , first, JVM will search for toString() method in the respective class, if it is not existed in the respective class JVM will search for toString() method in the super class of the respective class, if no super class is existed explicitly then JVM will search toString() method in the common and default superclass that is java.lang.Object class.

In java.lang.Object class, the toString() method was implemented in such a way that to return a String containing “className@RefValue”.

In All Collection classes, the toString() method was overridden in such a way that to return a String containing all the elements of the respective Collection by enclosed with [].

EX:

—--

ArrayList al = new ArrayList(); al.add(“AAA”);

al.add(“BBB”);

al.add(“CCC”);

al.add(“DDD”); System.out.println(al); OP: [AAA,BBB,CCC,DDD]

In Collections , we want to read element by element , that is, we want to display element by element on command prompt, to read elements like this we have to use Cursors or Iterators.

IN Collections, there are three types of Iterators or Cursors.

1. Enumeration

2. Iterator

3. ListIterator

Enumeration:

—-------------------

1. It was introduced in the JDK1.0 version.

2. It is a Legacy Cursor.

3. It is applicable for only Legacy Collections to read elements.

4. To represent Enumeration , JAVA has provided a predefined interface in the form of java.util.Enumeration and its implementation class was provided by Java internally.

5. To get java.util.Enumeration object we have to use the following method. public Enumeration elements()

EX: Enumeration e = vector.elements();

Where elements() method will read all the elements from the Vector and all these elements must be stored in Enumeration object which are ready to read or iterate

Note: When an Enumeration object is created , automatically, a cursor will be created just before the first element.

To read elements from Enumeration we have to use the following steps for each and every element.

1. Check whether more elements exist or not from the current cursor position. public boolean hasMoreElements()

a. It will check if more elements exist or not from the current cursor position, if atleast next element is available then this method will return true value, if no next element exists then it will return false value.

2. Read the next Element and move the cursor to the next element position. public Object nextElement()

EX:

—-

**package** com.codegnan.collectionframework;

**import** java.util.ArrayList;

**import** java.util.Enumeration;

**import** java.util.Vector;

**public** **class** EnumerationExample {

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

// Create an ArrayList and add elements

ArrayList<String> names = **new** ArrayList<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

names.add("Diana");

// Convert ArrayList to Vector

Vector<String> vector = **new** Vector<>(names);

// Create an Enumeration from the Vector

Enumeration<String> enumeration = vector.elements();

// Iterate using Enumeration

System.***out***.println("Names in the list:");

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

String name = enumeration.nextElement();

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

}

}

}

**Drawbacks:**

1. It is applicable for only Legacy Collections.

2. It allows only read operation while iterating elements, it does not allow remove, replace operations.

3. It allows you to read all the elements in only forward direction, not in Backward Direction.

**Iterator:**

**— ------------**

1. It was introduced in JDK1.2 version.

2. It is not a Legacy Cursor.

3. It is also called “Global Cursor”, because it is applicable for all the Collections.

4. It allows read operation and Remove operations while iterating elements.

5. To represent Iterator , JAVA has provided a predefined interface in the form of java.util.Iterator.

6. To get an Iterator object in Java applications we have to use the following method from all the Collection classes.

**public Iterator iterator()**

EX: Iterator iterator = collection.iterator();

When we execute iterator() method over the collection reference , JVM will copy all the elements from Collection and JVM will store all the elements by creating an Iterator object .

When an Iterator object is created, automatically a cursor[Pointer] will be before the first element to read elements.

To read elements one by one from Iterator we have to use the following methods.

**1. public boolean hasNext():**

It will check whether the next element exists or not from the current cursor position, if the next element exists from the current cursor position then it will return true value, if no next element exists from the current cursor position then it will return false value.

2. If next element is existed then read next element: public Object next()

To remove an element from Collection while iterating elements we have to use the following method.

**public void remove()**

EX:

**package** com.codegnan.collectionframework;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** Main {

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

// Create an ArrayList

ArrayList<String> fruits = **new** ArrayList<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Cherry");

fruits.add("Date");

// Create an iterator

Iterator<String> iterator = fruits.iterator();

// Iterate through the ArrayList using the iterator

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

String fruit = iterator.next();

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

}

// Optionally, you can also remove items while iterating

iterator = fruits.iterator(); // Reset the iterator

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

String fruit = iterator.next();

**if** (fruit.equals("Banana")) {

iterator.remove(); // Remove "Banana"

}

}

// Display the remaining items

System.***out***.println("After removal:");

**for** (String fruit : fruits) {

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

}

}

}

Example-2

—--------

**package** com.codegnan.collectionframework;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** IteratorExample {

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

List<Integer> list = **new** ArrayList<>();

**for** (**int** i = 1; i <= 20; i++) {

list.add(i);

}

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

Iterator<Integer> iterator = list.iterator();

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

**int** number = iterator.next();

**if** (number % 2 != 0) {

iterator.remove();

}

}

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

}

}

**Drawbacks:**

1. It allows only read and remove operations while iterating elements, it does not allow replace operation.

2. It allows us to read elements in only forward direction, not possible in Backward direction.

**3. ListIterator:**

**— --------------------**

1. It was introduced in JDK1.2 version

2. It is not legacy cursor

3. It is applicable for only List implementations.

4. It allows to perform read, remove and replace operations while iterating elements.

5. It allows us to read elements in both forward and Backward directions.

6. To represent ListIterator Java has provided a predefined interface in the form of java.util.ListIterator

7. To create a ListIterator object we have to use the following method from List. public ListIterator listIterator()

EX: ListIterator listIterator = list.listIterator();

When we execute the above code, all elements will be copied to ListIterator from the List object and a cursor will be created before the first element.

To read elements in Forward direction we have to use the following methods.

public boolean hasNext()

public int nextIndex(): It will return next elements index value public Object next()

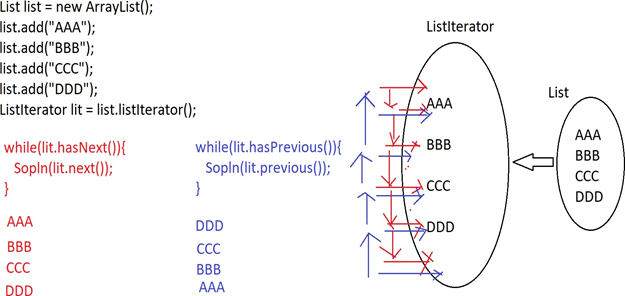
To read elements in Backward direction, we have to use the following methods. public boolean hasPrevious()

public int previousindex(): It will return previous elements index value public Object previous()

To remove an element from ListIterator we have to use the following method. public void remove()

To perform replacement operations over the elements while iterating elements we have to use the following method.

public Object set(Object newElement)



**package** com.codegnan.collectionframework;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.ListIterator;

**public** **class** ListIteratorExample {

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

List<String> fruits = **new** ArrayList<>();

fruits.add("APPLE");

fruits.add("BANANA");

fruits.add("orange");

fruits.add("mango");

fruits.add("kiwi");

fruits.add("pineapple");

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

ListIterator<String> fruitNames = fruits.listIterator();

System.***out***.println("Fruits in Forward Direction ");

**while** (fruitNames.hasNext()) {

System.***out***.println(fruitNames.nextIndex() + "---->" + fruitNames.next());

}

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

**while** (fruitNames.hasPrevious()) {

System.***out***.println("Fruits in Backward Direction : ");

System.***out***.println(fruitNames.previousIndex() + "----->" + fruitNames.previous());

}

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

**while** (fruitNames.hasNext()) {

String names = fruitNames.next();

**if** (names.equalsIgnoreCase("Banana")) {

fruitNames.set("grapes");

}

**if** (names.equalsIgnoreCase("kiwi")) {

fruitNames.add("mosambi");

}

**if** (names.equalsIgnoreCase("PINEAPPLE")) {

fruitNames.remove();

}

}

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

}

}

**Q) What are the differences between Enumeration, Iterator and ListIterator?**

**—--------------------------------**

**Ans:**

**—---**

1. Enumeration was introduced in JDK1.0 version.

Iterator and ListIterator were introduced in JDK1.2 version.

2. Enumeration is a Legacy Cursor.

Iterator and ListIterator are not Legacy Cursors.

3. Enumeration is an independent class, not related with Iterator and ListIterator. Iterator is parent to ListIterator.

4. Enumeration is applicable for only Legacy Collections. Iterator is applicable for all the Collections. ListIterator is applicable for only List implementations.

5. Enumeration is able to allow only Read operation while iterating elements.

Iterator is able to allow Read and Remove Operations while iterating elements.

ListIterator is able to allow Read, Remove, Insert and Replace operations while Iterating Elements.

6. Enumeration and Iterator are able to read elements in only Forward Direction.

ListIterator is able to read elements in both Forward and Backward Directions.

7. To get an Enumeration Object we have to use the following method. public Enumeration elements()

**EX: Enumeration e = vector.elements();**

To get an Iterator Object We have to use the following method. public Iterator iterator()

**EX: Iterator it = collection.iterator();**

To get a ListIterator object we have to use the following method. public ListIterator listIterator()

**EX: ListIterator lit = list.listIterator();**

8. To iterate Elements, Enumeration has provided the following methods.

**a. public boolean hasMoreElements()**

**b. public Object nextElement()**

To iterate elements, Iterator has provided the following methods.

**a. public boolean hasNext()**

**b. public Object next()**

**c. public void remove()**

To iterate elements, ListIterator has provided the following methods.

a. In Forward Direction:

public boolean hasNext()

public Object next()

public int nextIndex()

b. In Backward Direction:

public boolean hasPrevious()

public Object previous()

public int previousIndex()

c. For Manipulations with the elements:

public void add(Object obj)

public Object set(Object obj)

public void remove()

**Set :**

**—----**

1. It was introduced in JDK1.2 version.

2. It is not index based, it is able to manage all the elements as per their hashcode values.

3. It does not follow the Insertion order.

Note: LinkedHashSet is able to follow insertion order.

4. It does not follow Sorting Order.

Note: SortedSet, NavigableSet and treeSet are following Sorting order.

5. It allows Heterogeneous elements.

Note: SortedSet, NavigableSet and TreeSet are allowing only Homogeneous elements.

6. It does not allow duplicate elements.

7. It allows only one null element.

Note: SortedSet, NavigableSet and TreeSet are not allowing even a single null element.

Note: Set interface is not having new methods, it is having all the methods which are declared in Collection interface.

**HashSet:**

**— -------**

1. It was introduced in JDK1.2 version.

2. It is not a Legacy Collection.

3. It is an implementation class to Set interface.

4. It is not index based, it is able to manage all the elements on the basis of elements hashcode values.

5. It does not follow INsertion order.

6. It does not follow Sorting order.

7. It allows heterogeneous elements.

8. It allows only one null element.

9. It does not allow duplicate elements.

10. The default Initial capacity in HashSet is 16 elements.

11. The default Fill ratio / Load Factor for HashSet is 75%, that is , When HashSet is filled up to 75% the HashSet will increase its capacity automatically.

12. HashSet has a Hashtable is an internal data Structure.

13. HashSet is suggestible for frequent search operations.

14. It is not a Synchronized Collection.

15. It allows more than one thread at a time to access data.

16. It follows parallel execution of the threads.

17. It reduced application execution time.

18. It increases application Performance.

19. It is not giving guarantee for data consistency.

20. It is not Thread Safe.

**Constructors:**

**—--------------------**

**1.** **public HashSet():**

It is able to create an empty HashSet object with 16 elements capacity and 75% load factor.

EX:

**import** java.util.HashSet;

**public** **class** Main {

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

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

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

}

}

**2.** **public HashSet(int capacity)**

It is able to create an empty HashSet object with the specified capacity value and with the default load factor.

EX:

—-

**import** java.util.HashSet;

**public** **class** Main {

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

HashSet hashSet = **new** HashSet(20);

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

}

}

**3.** **public HashSet(int capacity, float loadFactor):**

It is able to create an empty HashSet object with the specified capacity value and with the specified loadFactor, where load Factor value is suggestible from 0 to 1.

EX:

**import** java.util.HashSet;

**public** **class** Main {

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

HashSet hashSet = **new** HashSet(20, 0.85f);

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

}

}

**4.** **public HashSet(Collection c):**

It is able to create a HashSet object with all the elements of the specified Collection.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("AAA");

list.add("BBB");

list.add("CCC");

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

HashSet hashSet1 = **new** HashSet(list);

System.***out***.println(hashSet1);

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

Set set = **new** HashSet();

set.add("XXX");

set.add("YYY");

set.add("ZZZ");

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

HashSet hashSet2 = **new** HashSet(set);

System.***out***.println(hashSet2);

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

Queue queue = **new** PriorityQueue();

queue.add(111);

queue.add(222);

queue.add(333);

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

HashSet hashSet3 = **new** HashSet(queue);

System.***out***.println(hashSet3);

}

}

OP:

[AAA, BBB, CCC]

[AAA, CCC, BBB]

[YYY, XXX, ZZZ]

[YYY, XXX, ZZZ]

[111, 222, 333]

[333, 222, 111]

EX:

—--

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

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

hashSet.add("AAA");

hashSet.add("BBB");

hashSet.add("CCC");

hashSet.add("DDD");

hashSet.add("EEE");

hashSet.add("FFF");

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

hashSet.add("BBB");

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

hashSet.add(**null**);

hashSet.add(**null**);

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

hashSet.add(10);

hashSet.add(22.22f);

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

}

}

Ex:

—----

**package** com.codegnan.collectionframework;

**import** java.util.HashSet;

**public** **class** Main {

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

// Create a HashSet to store fruit names

HashSet<String> fruits = **new** HashSet<>();

// Adding elements to the HashSet

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Cherry");

fruits.add("Date");

fruits.add("Banana"); // Duplicate element (will not be added)

// Display the HashSet

System.***out***.println("Fruits: " + fruits);

// Check if a specific element is in the HashSet

**if** (fruits.contains("Cherry")) {

System.***out***.println("Cherry is in the set.");

} **else** {

System.***out***.println("Cherry is not in the set.");

}

// Remove an element from the HashSet

fruits.remove("Date");

System.***out***.println("After removing Date: " + fruits);

// Iterate over the HashSet

System.***out***.println("Iterating over fruits:");

**for** (String fruit : fruits) {

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

}

// Get the size of the HashSet

System.***out***.println("Number of unique fruits: " + fruits.size());

// Clear the HashSet

fruits.clear();

System.***out***.println("After clearing, is the set empty? " + fruits.isEmpty());

}

}

**LinkedHashSet:**

**— --------------------**

**Q) What are the differences between HashSet and LinkedHashSet?**

**—----------**

**Ans:**

**—---**

1. HashSet was introduced in JDK1.2 version. LinkedHashSet was introduced in JDK1.4 version.

2. HashSet does not follow insertion order. LinkedHashSet follows insertion order.

EX:

**package** com.codegnan.collectionframework;

**import** java.util.HashSet;

**public** **class** Main {

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

// Create a HashSet to store fruit names

LinkedHashSet<String> fruits = **new** LinkedHashSet<>();

// Adding elements to the HashSet

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Cherry");

fruits.add("Date");

fruits.add("Banana"); // Duplicate element (will not be added)

// Display the HashSet

System.***out***.println("Fruits: " + fruits);

// Check if a specific element is in the HashSet

**if** (fruits.contains("Cherry")) {

System.***out***.println("Cherry is in the set.");

} **else** {

System.***out***.println("Cherry is not in the set.");

}

// Remove an element from the HashSet

fruits.remove("Date");

System.***out***.println("After removing Date: " + fruits);

// Iterate over the HashSet

System.***out***.println("Iterating over fruits:");

**for** (String fruit : fruits) {

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

}

// Get the size of the HashSet

System.***out***.println("Number of unique fruits: " + fruits.size());

// Clear the HashSet

fruits.clear();

System.***out***.println("After clearing, is the set empty? " + fruits.isEmpty());

}

}

**SortedSet:**

**— ----------------**

1. SortedSet was introduced in JDK1.2 version.

2. It is a direct child interface to Set interface.

3. In general Sets are not following Insertion order and Sorting order, but the main intention of the SortedSet is to arrange all the elements as per Sorting.

4. In general, Sets are able to allow heterogeneous elements, but SortedSet does not allow heterogeneous elements, it allows only Homogeneous elements in order to compare the elements and in order to arrange all the elements in sorting order.

5. In general, Set implementations are able to allow one null element, but SortedSet does not allow even a single null element, if we add null element to SortedSet then JVM will raise an exception like java.lang.NullPointerException.

6. In general, Sets are able to allow normal objects, but SortedSet is able to allow only Comparable Objects , that is the objects which are implementing java.lang.Comparable interface. If we add Non comparable Objects to SortedSet then JVM will raise an exception like java.lang.ClassCastException, in this case to avoid ClassCastException we have to use java.util.Comparator.

**Methods:**

**1. public Object first():**

It is able to return the First element from SortedSet.

**2. public Object last():**

It is able to return the last element from SortedSet.

**3. public SortedSet headSet(Object element):**

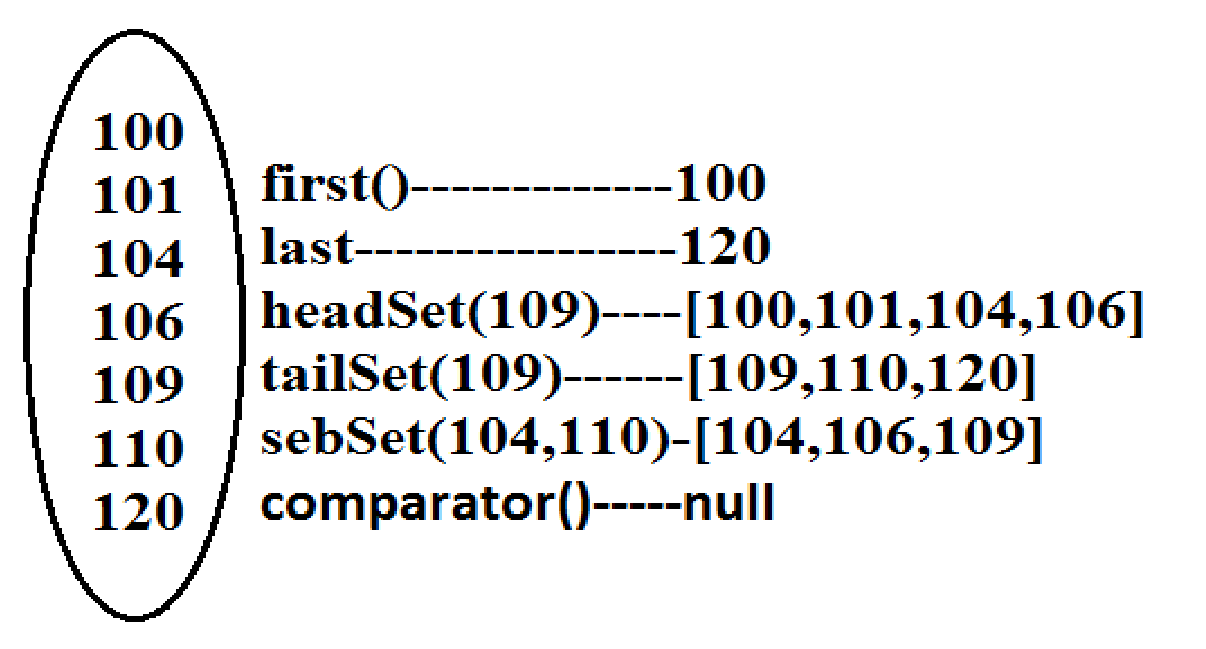
It is able to return all the elements in the form of SortedSet which are less than the specified element.

**4. public SortedSet tailSet(Object element):**

It is able to return all the elements in the form of SortedSet which are greater than or equal to the specified element.

**5. public SortedSet subSet(Object fromElement, Object toElement):**

It is able to return all the elements in the form of SortedSet which are greater than or equal to the specified fromELement and less than the specified toElement.



EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

SortedSet <String>sortedSet = **new** TreeSet<>();

sortedSet.add("FFF");

sortedSet.add("AAA");

sortedSet.add("EEE");

sortedSet.add("BBB");

sortedSet.add("DDD");

sortedSet.add("CCC");

System.***out***.println(sortedSet);// [AAA,BBB,CCC,DDD,EEE,FFF]

System.***out***.println(sortedSet.first());// AAA

System.out.println(sortedSet.last());//FFF

System.***out***.println(sortedSet.headSet("DDD"));// [AAA,BBB,CCC]

System.out.println(sortedSet.tailSet("DDD"));//[DDD,EEE,FFF]

System.***out***.println(sortedSet.subSet("BBB", "EEE"));// [BBB,CCC,DDD]

}

}

**NavigableSet:**

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

It was introduced in JDK1.6 version, it is a direct child interface to SortedSet interface, it has almost all the features of SortedSet and it has define some new methods apart from the SOrtedSet methods in order to provide navigation over the elements like finding ceiling element, floor element, descendingSet,.......

**Methods:**

**1. public NavigableSet descendingSet():**

It is able to generate a NavigableSet with all the elements in Descending order.

**2. public Object ceiling(Object obj):**

It is able to return the lowest element among all the elements which are greater than or equal to the specified element.

**3. public Object higher(Object element):**

It is able to return the lowest element among all the elements which are greater than the specified element.

**4. public Object floor(Object element):**

It is able to return the highest element among all the elements which are less than or equal to the specified element.

**5. public Object lower(Object element):**

It is able to return the highest element among all the elements which are less than the specified element.

**6. public Object pollFirst():**

It is able to return and remove the First element.

**7. public Object pollLast():**

It is able to return and remove the last element.

EX:

—--

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

NavigableSet<String> navigableSet = **new** TreeSet<>();

navigableSet.add("FFF");

navigableSet.add("AAA");

navigableSet.add("EEE");

navigableSet.add("BBB");

navigableSet.add("DDD");

navigableSet.add("CCC");

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

System.***out***.println(navigableSet.descendingSet());// [FFF,EEE, DDD,CCC,BBB,AAA]

System.***out***.println(navigableSet.ceiling("DDD"));//

System.***out***.println(navigableSet.higher("DDD"));// EEE System.out.println(navigableSet.floor("DDD"));// DDD

// System.out.println(navigableSet.lower("DDD"));// CCC

// System.out.println(navigableSet.pollFirst());// AAA

// System.out.println(navigableSet.pollLast());// FFF

// System.out.println(navigableSet);// [BBB,CCC,DDD,EEE]

}

}

Ex2

=======

**package** com.codegnan.collectionframework;

**import** java.util.NavigableSet;

**import** java.util.TreeSet;

**public** **class** Main {

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

// Create a NavigableSet using TreeSet

NavigableSet<Integer> numbers = **new** TreeSet<>();

// Adding elements to the NavigableSet

numbers.add(10);

numbers.add(20);

numbers.add(5);

numbers.add(15);

numbers.add(25);

// Display the NavigableSet

System.***out***.println("Numbers: " + numbers);

// Get the first (lowest) element

System.***out***.println("First (lowest) number: " + numbers.first());

// Get the last (highest) element

System.***out***.println("Last (highest) number: " + numbers.last());

// Get the greatest element less than or equal to 15

System.***out***.println("Greatest number less than or equal to 15: " + numbers.floor(15));

// Get the least element greater than or equal to 15

System.***out***.println("Least number greater than or equal to 15: " + numbers.ceiling(15));

// Get the greatest element less than 15

System.***out***.println("Greatest number less than 15: " + numbers.lower(15));

// Get the least element greater than 15

System.***out***.println("Least number greater than 15: " + numbers.higher(15));

// Get the head set (elements less than 15)

System.***out***.println("Head set (elements less than 15): " + numbers.headSet(15));

// Get the tail set (elements greater than or equal to 15)

System.***out***.println("Tail set (elements greater than or equal to 15): " + numbers.tailSet(15));

// Iterate through the NavigableSet in reverse order

System.***out***.println("Numbers in reverse order:");

**for** (Integer number : numbers.descendingSet()) {

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

}

// Remove an element

numbers.remove(20);

System.***out***.println("After removing 20: " + numbers);

}

}

**TreeSet:**

**—----------**

1. It was introduced in JDK1.2 version

2. It is not a Legacy Collection.

3. It is a direct implementation class to NavigableSet, where NavigableSet is a child interface to Set interface, where Set interface is child interface to Collection interface, Due to this inheritance architecture TreeSet must provide the implementations for all methods of Collection, Set, SortedSet and NavigableSet interfaces.

4. It is not index based.

5. It is not following insertion order.

6. It is following Sorting order.

7. It allows only homogeneous elements, if we add heterogeneous elements then JVM will raise an exception like java.lang.ClassCastException.

8. It does not allow duplicate elements.

9. It does not allow null elements, if we add null elements to the TreeSet then JVM will raise an exception like java.lang.NullPointerException.

10. It allows only Comparable elements, that is the elements which implement java.lang.Comparable interface. If we add Non comparable elements then JVM will raise an exception like java.lang.ClassCastException, in this context if we want to avoid ClassCastException then we have to use “java.util.Coparator”.

11. Its internal Data Structure is “Balanced Tree”.

12. It is not a Synchronized Collection.

13. It allows more than one thread at a time to access data.

14. It follows parallel execution over the threads.

15. It is able to reduce application execution time.

16. It is able to increase application performance.

17. It does not guarantee Data Consistency.

18. It is not a thread safe resource.

**Constructors:**

**1. public TreeSet():**

It is able to create an empty TreeSet object with the default natural sorting order.

EX:

------

**import** java.util.TreeSet;

**public** **class** Main {

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

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

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

}

}

**2. public TreeSet(Comparator c):**

It is able to create an empty TreeSet object with the explicit sorting order in the form of the provided Comparator object.

**3. public TreeSet(Collection c):**

It is able to create a TreeSet object with all the elements of the specified Collection in default natural sorting order.

EX:

---------

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

List list = **new** ArrayList();

list.add("CCC");

list.add("AAA");

list.add("DDD");

list.add("BBB");

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

TreeSet treeSet1 = **new** TreeSet(list);

System.***out***.println(treeSet1);

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

Set set = **new** HashSet();

set.add("AAA");

set.add("BBB");

set.add("CCC");

set.add("DDD");

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

TreeSet treeSet2 = **new** TreeSet(set);

System.***out***.println(treeSet2);

Queue queue = **new** PriorityQueue();

queue.add(5);

queue.add(3);

queue.add(4);

queue.add(2);

queue.add(1);

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

TreeSet treeSet3 = **new** TreeSet(queue);

System.***out***.println(treeSet3);

}

}

**4. public TreeSet(SortedSet ss):**

It is able to create a TreeSet object with all the elements of the specified SortedSet object.

EX:

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

SortedSet sortedSet = **new** TreeSet();

sortedSet.add("AAA");

sortedSet.add("BBB");

sortedSet.add("CCC");

sortedSet.add("DDD");

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

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

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

}

}

Ex:

========

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**class** Employee {

}

**public** **class** Main {

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

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

treeSet.add("FFF");

treeSet.add("AAA");

treeSet.add("EEE");

treeSet.add("BBB");

treeSet.add("DDD");

treeSet.add("CCC");

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

treeSet.add("BBB");

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

//treeSet.add(null);---> java.lang.NullPointerExcepotion

//treeSet.add(10); ---> java.lang.ClassCastException

//treeSet.add(new Employee()); ---> java.lang.ClassCastException

}

}

Example

=========

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

// Create a TreeSet to store integers

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

// Adding elements to the TreeSet

numbers.add(10);

numbers.add(20);

numbers.add(5);

numbers.add(15);

numbers.add(25);

numbers.add(20); // Duplicate element, will not be added

// Display the elements in the TreeSet (sorted order)

System.out.println("TreeSet elements: " + numbers);

// Remove an element

numbers.remove(15);

System.out.println("After removing 15: " + numbers);

// Check if an element exists

boolean contains10 = numbers.contains(10);

System.out.println("Does TreeSet contain 10? " + contains10);

// Get the first and last elements

Integer first = numbers.first();

Integer last = numbers.last();

System.out.println("First element: " + first);

System.out.println("Last element: " + last);

}

}

**Q) How TreeSet is providing all the elements in Sorting order?**

**—------------------------------------**

**Ans:**

**—---**

To provide all the elements in Sorting order, TreeSet is able to follow the following steps.

1. Create a Balanced Tree with all the elements of the TreeSet.

2. Retrieve all elements from the Balanced Tree by following In-Order Traversal.

To create Balanced Tree we have to use the following steps

1. Make the first value in TreeSet as Root Node.

2. For the second element onwards we have to use the following steps.

a. Access compareTo() method over the provided present element by passing all the previous elements of the TreeSet as parameters.

b. If compareTo() method returns -ve value then check whether any LEFT NODE exists or not, if left node exists then repeat the same process for left node, if no left node exists make the present element as left node.

c. If compareTo() method returns +ve value then check whether any RIGHT NODE exists or not, if any RIGHT NODE exists then repeat the same process for Right Node, if no Right Node exists then Make the present element as Right Node.

d. If compareTo() method returns 0 value then decide the present new element is a duplicate element and discard that element.

To retrieve all elements from Balanced Tree we have to use IN-Order Traversal: For every element, get the element as per Left-Root-Right approach.

Functionality of CompareTo() method:

compareTo() method exists in java.lang.Comparable interface, it has been implemented in a number of predefined classes like java.lang.String .

The String class has implemented the compareTo() method in the following way. It checks whether the provided values are in dictionary order or not.

EX: str1.compareTo(str2);

1. If str1 comes first in dictionary order when compared with str2 then compareTo() method will return -ve value.

2. If str2 comes first in dictionary order when compared with str1 then compareTo() method will return +ve value.

3. If str1 and str2 are at the same position in Dictionary Order then compareTo() method will return 0 value.

EX:

—--

**package** com.codegnan.collectionframeworks;

**public** **class** Test {

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

String str1 = "abc";

String str2 = "def";

String str3 = "abc";

System.***out***.println(str1.compareTo(str2));// abc.cTo(def)=> -ve

// System.out.println(str2.compareTo(str3));//def.cTo(abc)=>

// +ve System.out.println(str3.compareTo(str1));//abc.cTo(abc)=> 0

}

}

OP:

—-

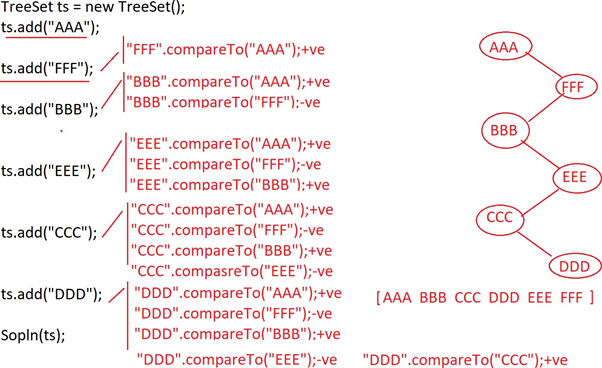
-3

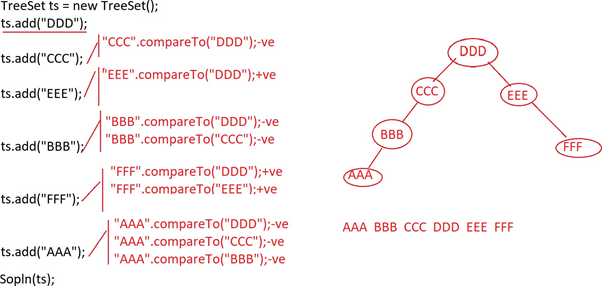
3

0

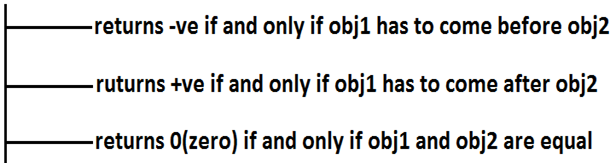
EX:







### **Comparable interface:**

Comparable interface present in java.lang package and contains only one method compareTo() method.  
 public int compareTo(Object obj);  
 Example:  
 **obj1.compareTo(obj2);** Diagram:  
  
  
 public class Main

{

public static void main(String[] args)

{

// Compare strings and print the results

System.out.println("A".compareTo("Z")); // -25

System.out.println("Z".compareTo("K")); // 15

System.out.println("A".compareTo("A")); // 0

// Uncommenting these lines will result in compilation errors or runtime exceptions

// System.out.println("A".compareTo(new Integer(10))); // Compile-time error

// System.out.println("A".compareTo(null)); // NullPointerException at runtime

}

}

If we are depending on default natural sorting order then internally JVM will use compareTo() method to arrange objects in sorting order.

Example 4:

import java.util.\*;

class Test

{

public static void main(String[] args)

{

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

t.add(10);

t.add(0);

t.add(15);

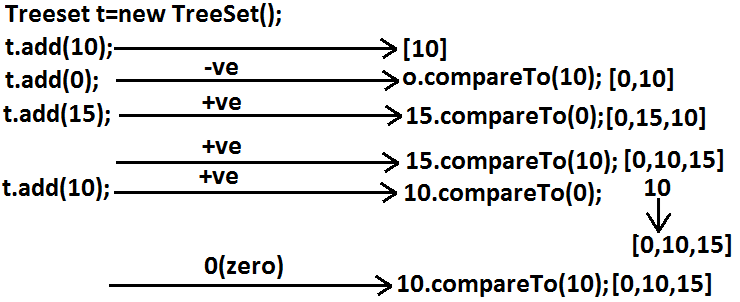
t.add(10);

System.out.println(t);//[0, 10, 15]

}

}

**compareTo() method analysis:**

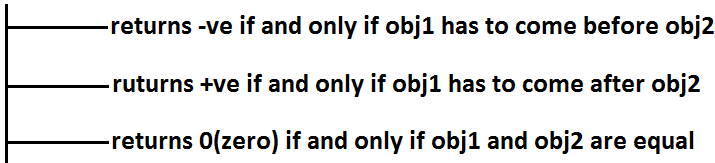
****

* If we are not satisfying with default natural sorting order (or) if default natural sorting order is not available then we can define our own customized sorting by Comparator object.
* Comparable meant for default natural sorting order.
* Comparator meant for customized sorting order.

**Comparator interface:**

Comparator interface present in java.util package this interface defines the following 2 methods.

**1) public int compare(Object obj1,Object Obj2);**

Diagram:  
  


**2) public boolean equals(Object obj);**

* Whenever we are implementing Comparator interface we have to provide implementation only for compare() method.
* Implementing equals() method is optional because it is already available from Object class through inheritance.

Requirement: **Write a program to insert integer objects into the TreeSet where the sorting order is descending order.**

Program:

import java.util.\*;

class Test

{

public static void main(String[] args)

{

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

t.add(10);

t.add(0);

t.add(15);

t.add(5);

t.add(20);

System.out.println(t);//[20, 15, 10, 5, 0]

}

}

class MyComparator implements Comparator

{

public int compare(Object obj1,Object obj2)

{

Integer i1=(Integer)obj1;

Integer i2=(Integer)obj2;

if(i1<i2)

return +1;

else if(i1 > i2)

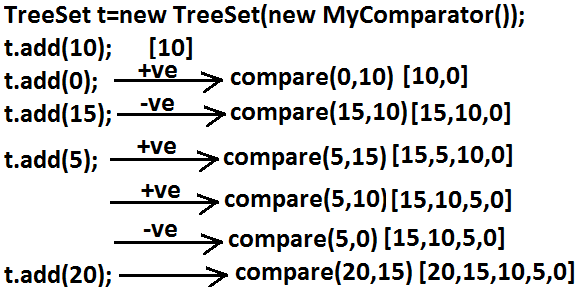
return -100;

else return 0;

}

}

* At line "1" if we are not passing Comparator object then JVM will always calls compareTo() method which is meant for default natural sorting order(ascending order)hence in this case the output is [0, 5, 10, 15, 20].
* At line "1" if we are passing Comparator object then JVM calls compare() method of MyComparator class which is meant for customized sorting order(descending order) hence in this case the output is [20, 15, 10, 5, 0].

***Diagram:  
  
***

Various alternative implementations of compare() method:

public int compare(Object obj1,Object obj2)

{

Integer i1=(Integer)obj1;

Integer i2=(Integer)obj2;

//return i1.compareTo(i2);//[0, 5, 10, 15, 20]

//return -i1.compareTo(i2);//[20, 15, 10, 5, 0]

//return i2.compareTo(i1);//[20, 15, 10, 5, 0]

//return -i2.compareTo(i1);//[0, 5, 10, 15, 20]

//return -1;//[20, 5, 15, 0, 10]//reverse of insertion order

//return +1;//[10, 0, 15, 5, 20]//insertion order

//return 0;//[10]and all the remaining elements treated as duplicate.

}

Requirement: **Write a program to insert String objects into the TreeSet where the sorting order is reverse of alphabetical order.**

Program:

import java.util.\*;

class TreeSetDemo

{

public static void main(String[] args)

{

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

t.add("Roja");

t.add("ShobaRani");

t.add("RajaKumari");

t.add("GangaBhavani");

t.add("Ramulamma");

System.out.println(t);//[ShobaRani, Roja, Ramulamma, RajaKumari, GangaBhavani]

}

}

class MyComparator implements Comparator

{

public int compare(Object obj1,Object obj2)

{

String s1=obj1.toString();

String s2=(String)obj2;

//return s2.compareTo(s1);

return -s1.compareTo(s2);

}

}

**Queue:**

**—-------**

1. It was introduced in JDK1.5 version.

2. It is a direct child interface to Collection interface.

3. It is not index based.

4. It is able to allow duplicate elements.

5. It is able to manage all the elements as per prior to the processing.

6. It is able to follow FIFO [First In First Out], but implementation classes may or may not follow FIFO.

7. It is able to follow natural ordering.

8. It is able to allow only Homogeneous elements.

9. It does not allow null elements.

10. It allows only Comparable elements, if we want to add non comparable elements then we must use “java.util.Comparator”.

**Methods :**

**—-----------------**

**1. public boolean offer(Object obj)**

It is able to add the specified element to Queue.

**2. public Object peek():**

It is able to read head element from Queue

**3. public Object element():**

It is able to read the head element of the Queue.

**Q)What is the difference between peek() method and element() method from Queue interface?**

**—---------------------------------------------**

**Ans:**

**—--**

If we access peek() method on an empty Queue then it will return null value, but if we access element() method on an empty queue then it will raise an exception like java.util.NoSuchElementException.

**4. public Object poll():**

It is able to read and remove the Head element from the Queue.

**5. public Object remove():**

It is able to remove the head element from Queue.

**Q) What is the difference between poll() method and remove() method?**

**—-----------------------------**

**Ans:**

**—---**

If we access the poll() method on an empty queue object then it will return null value.

If we access the remove() method on an empty queue object then it will raise an exception like java.util.NoSuchElementException.

**package** com.codegnan.collectionframeworks;

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public** **class** Main {

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

Queue <String>queue = **new** PriorityQueue<>();

queue.offer("AAA");

queue.offer("BBB");

queue.offer("CCC");

queue.offer("DDD");

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

System.***out***.println(queue.peek());

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

System.***out***.println(queue.element());

/\*

\* Queue queue1 = new PriorityQueue(); System.out.println(queue1.peek());

\* System.out.println(queue1.element());--> java.util.NoSuchElementException

\*/ System.***out***.println(queue.poll());

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

System.***out***.println(queue.remove());

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

/\*

\* Queue queue1 = new PriorityQueue(); System.out.println(queue1.poll());

\*

\* System.out.println(queue1.remove());---> java.util.NoSuchElementException

\*

\*/

}

}

**PriorityQueue:**

**—-----------------------**

1. It was introduced in JDK1.5 version

2. It is not a legacy collection.

3. It is an implementation class to Queue interface.

4. It is not index based.

5. It is able to allow duplicate elements.

6. It is able to manage all the elements prior to the processing as per elements priority.

7. It is able to follow natural order.

8. It is able to allow only Homogeneous elements.

9. It does not allow null elements.

10. It allows only Comparable elements, if we want to add non comparable elements then we must use “java.util.Comparator”.

11. It is an unbounded queue, it does not have any boundaries on sizes or capacities, if we add an element automatically its capacity will be increased.

12. Its internal data structure is Heap Array.

13. Its initial capacity is 11 elements.

14. It is not not synchronized

15. It allows more than one thread at a time to access data.

16. It follows parallel execution.

17. It reduces application execution time.

18. It improves application performance.

19. It is not a guarantee for Data Consistency.

20. It is not Thread safe.

**Constructors:**

**—---------------------**

**1. public PriorityQueue():**

It is able to create an empty Priority Queue with default natural ordering with the initial capacity is 11 .

EX:

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public** **class** Main {

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

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

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

}

}

**2. public PriorityQueue(int capacity):**

It is able to create an empty PriorityQueue with the specified capacity value.

EX:

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public** **class** Main {

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

PriorityQueue pq = **new** PriorityQueue(20);

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

}

}

**3. public PriorityQueue(Comparator c):**

It is able to create an empty PriorityQueue with Customized sorting logic.

EX:

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public** **class** Main {

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

PriorityQueue<Integer> pq = **new** PriorityQueue(Comparator.*reverseOrder*());

pq.add(5);

pq.add(2);

pq.add(6);

pq.add(1);

pq.add(4);

pq.add(3);

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

}

}

**4. public PriorityQueue(Collection c):**

It is able to create a PriorityQueue object with all the elements of the specified Collection.

EX:

**import** java.util.\*;

**public** **class** Main {

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

List <String>list = **new** ArrayList<>();

list.add("AAA");

list.add("BBB");

list.add("CCC");

list.add("DDD");

list.add("EEE");

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

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

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

}

}

**5. public PriorityQueue(SortedSet ss):**

It is able to create a PriorityQueue object with all the elements of the provided SortedSet.

EX:

**import** java.util.\*;

**public** **class** Main {

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

SortedSet <String>sortedSet = **new** TreeSet<>();

sortedSet.add("AAA");

sortedSet.add("BBB");

sortedSet.add("CCC");

sortedSet.add("DDD");

sortedSet.add("EEE");

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

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

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

}

}

**6. public PriorityQueue(PriorityQueue pq):**

It is able to create a PriorityQueue object with all the elements of the specified PriorityQueue object.

EX:

**import** java.util.\*;

**public** **class** Main {

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

PriorityQueue<String> pq = **new** PriorityQueue<>();

pq.add("AAA");

pq.add("BBB");

pq.add("CCC");

pq.add("DDD");

pq.add("EEE");

pq.add("FFF");

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

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

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

}

}

**Q) How does PriorityQueue work internally?**

**—----------------------------**

**Ans:**

**—---**

1. Create Min Heap Structure.

2. Get elements from MIN Heap Structure.

1. Create Min Heap Structure.

a. If the element is the first element then add that element as the root node.

b. If the element is not the first element, add that element as left not, if the next node exists then add that node as right node.

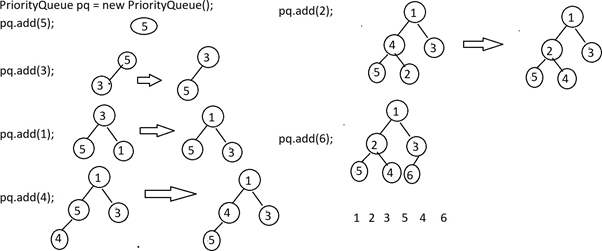
c. If the present node has both left node and right node then add the present element to the left node in the next level.

d. After adding the element to Min Heap then check if the present element is less than the parent then swap the elements, otherwise swap operation is not required.

2. Get the elements from Min Heap.

Read all elements from left to right at each and every level.

Note: If we remove an element from PriorityQueue the highest priority element will be removed first, that is the number with low value, after removing the element Min Heap Structure will be re-arranged.



**Deque:**

**— -----------**

1. Deque means Double ended Queue.

2. Deque was introduced in JDK1.6 version.

3. Deque is a child interface to Queue interface.

4. In general, Queues are able to follow FIFO, where insertion will be provided in the rear side and removal will be performed at Head side, but Deques are able to allow insertions and deletions at both the sides.

**Methods:**

**—--------**

**1. Public void addFirst(Object element):**

It is able to add the specified element as the first element in Deque.

**2. public boolean offerFirst(Object element)**

It is able to add the specified element as the first element.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>();

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

dq.addFirst("AAA");

dq.addFirst("BBB");

dq.addFirst("CCC");

dq.addFirst("DDD");

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

dq.offerFirst("EEE");

dq.offerFirst("FFF");

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

}

}

**Q) What is the difference between addFirst() method and offerFirst() method?**

**—-----------------------------------------**

**Ans:**

**------**

1. addFirst() method return type id void. offerFirst() method return type if boolean.

2. If the Deque is capacity restricted then addFirst() method will raise an exception.

Even though Deque is capacity restricted then the offerFirst() method will not raise any exception.

**3. public Object addLast(Object element):**

It can be used to add the specified element as Last element.

4**. public boolean offerLast(Object element):**

It can be used to add the specified element as last element in Deque.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>(4);

dq.addLast("AAA");

dq.addLast("BBB");

dq.addLast("CCC");

dq.addLast("DDD");

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

dq.offerLast("EEE");

dq.offerLast("FFF");

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

}

}

**Q) What is the difference between addLast() method and offerLast()?**

**—--------------------------------------**

**Ans:**

**—----**

1. addLast() method has void return type. offerLast() method will have boolean return type.

2. If the Deque is Capacity Restricted , if no space is available for adding an element then the addLast() method will raise an exception.

Even through the Deque is capacity restricted , if no space is available in Deque offerLast() method will not raise any exception.

**5. public Object getFirst():**

It can be used to get the First element from Deque.

**6. public Object peekFirst():**

It can be used to get the First element from Deque.

EX:

—--

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>(4);

dq.addLast("AAA");

dq.addLast("BBB");

dq.addLast("CCC");

dq.addLast("DDD");

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

System.***out***.println(dq.getFirst());

System.***out***.println(dq.peekFirst());

}

}

**Q)What is the difference between getFirst() and peekFirst() method?**

**—--------------------------------------------------------**

**Ans:**

**—---**

If we access the getFirst() method on an empty Deque then JVM will rase an exception like java.util.NoSuchElementException.

If we access peekFirst() method on an empty Deque then JVM will not raise any exception.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque <String>dq1 = **new** ArrayDeque<>(4);

System.***out***.println(dq1.peekFirst());

Deque dq2 = **new** ArrayDeque(4);

System.***out***.println(dq2.getFirst());// NoSuchElementException

}

}

**7. public Object getLast():**

It can be used to get the Last element from Deque.

**8. public Object peekLast():**

It can be used to get the last element.

EX:

—--

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque <String>dq = **new** ArrayDeque<>(4);

dq.addLast("AAA");

dq.addLast("BBB");

dq.addLast("CCC");

dq.addLast("DDD");

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

System.***out***.println(dq.getLast());

System.***out***.println(dq.peekLast());

}

}

Note: getLast() method will raise an exception when no element exists in the Deque, but, peekLast() method will return null value instead of providing an exception.

EX:

—--

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque dq = **new** ArrayDeque();

System.***out***.println(dq.peekLast());

System.***out***.println(dq.getLast());// java.util.NoSuchElementException

}

}

**9. public Object removeFirst():**

It is able to remove the first element from Deque.

**10. public Object pollFirst():**

It is able to remove the first element from Deque.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>();

dq.add("AAA");

dq.add("BBB");

dq.add("CCC");

dq.add("DDD");

dq.add("EEE");

dq.add("FFF");

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

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

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

System.***out***.println(dq.removeFirst());

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

}

}

Note: If we access pollFirst() method on an empty Deque then JVM will return null value, but if we access removeFirst() method on Deque then JVM will raise an exception like java.util.NoSuchElementException

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque dq = **new** ArrayDeque();

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

System.***out***.println(dq.removeFirst());// NoSuchElementException

}

}

**11. public Object removeLast():**

It can be used to remove the last element from Deque.

**12. public Object pollLast():**

It can be used to remove the last element from Deque.

EX:

—-

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>();

dq.add("AAA");

dq.add("BBB");

dq.add("CCC");

dq.add("DDD");

dq.add("EEE");

dq.add("FFF");

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

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

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

System.***out***.println(dq.removeLast());

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

}

}

Note: If we access pollLast() method on an empty Deque then JVM will return null value, but if we access removeLast() method on an empty Deque then JVM will raise an exception like java.util.NoSuchElementException .

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque dq = **new** ArrayDeque();

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

System.***out***.println(dq.removeLast());// java.util.NoSuchElementException

}

}

**13. Public boolean removeFirstOccurence(Object element)**

It is able to remove an element from its first occurrence in Deque.

**14. Public boolean removeLastOccurence(Object element)**

It is able to remove an element from its last occurrence in Deque.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String>dq = **new** ArrayDeque<>();

dq.add("AAA");

dq.add("BBB");

dq.add("CCC");

dq.add("DDD");

dq.add("BBB");

dq.add("FFF");

dq.add("GGG");

dq.add("BBB");

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

System.***out***.println(dq.removeFirstOccurrence("BBB"));

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

System.***out***.println(dq.removeLastOccurrence("BBB"));

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

}

}

**ArrayDeque:**

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

1. It was introduced in JDK1.6 version.

2. It is not a Legacy Collection.

3. It is an implementation class to the Deque interface.

4. It is not index based.

5. It is able to Follow Insertions and deletions at both sides of the Deque.

6. It is not following insertion order

7. It is not following the sorting order.

8. It is allowing duplicates.

9. It allows Heterogeneous element.

10. It does not allow null elements.

11. It allows directly Non Comparable elements.

12. Its initial capacity is 16 elements.

13. Its internal data structure is Resizable Array.

14. It is faster than Stack and Faster than LinkedList.

15. It is suggestible for frequent insertions and deletions .

16. It is not synchronized.

17. It allows more than one thread at a time.

18. It follows parallel execution over the threads.

19. It will reduce application execution time.

20. It will increase application performance.

21. It is not a guarantee for Data Consistency.

22. It is not a Thread Safe resource.

**Constructors**

**—---------------------**

**1. public ArrayDeque():**

It is able to create an ArrayDeque object with the initial capacity of 16 elements.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**public** **class** Main {

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

Deque <String>dq = **new** ArrayDeque<>();

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

}

}

**2. public ArrayDeque(int capacity):**

It is able to create an empty Deque object with the specified initial capacity.

EX:

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>(20);

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

}

}

**3. public ArrayDeque(Collection c)**

It is able to create an ArrayDeque with ball the elements of the specified Collection.

EX:

**import** java.util.ArrayDeque;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**public** **class** Main {

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

Collection <String>collection = **new** ArrayList<>();

collection.add("AAA");

collection.add("BBB");

collection.add("CCC");

collection.add("DDD");

Deque dq = **new** ArrayDeque(collection);

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

}

}

Ex

----

**package** com.codegnan.collectionframeworks;

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.concurrent.DelayQueue;

**class** A {

}

**public** **class** Main {

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

Deque<String> dq = **new** ArrayDeque<>();

dq.add("AAA");

dq.add("BBB");

dq.add("CCC");

dq.add("DDD");

dq.add("EEE");

dq.add("FFF");

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

dq.add("BBB");

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

dq.add(10);

dq.add(22.22f);

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

// dq.add(null);NullPointerException

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

dq.add(**new** A());

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

}

}

**Map:**

**—----**

1. It was introduced in JDK1.2 version.

2. It is not a child interface to Collection interface.

3. It is able to manage all the objects in the form of Key-Value pairs, where both Keys and Values must be objects.

4. In Maps, Keys must be unique, keys must not be duplicated but values may be duplicated.

5. In Maps, keys are able to allow only one null element , but values may allow more than one null element.

6. In Maps, by default, Keys and Values are able to allow heterogeneous elements.

7. Maps are able to follow Keys insertion order.

**Methods:**

**1. public void put(Object key, Object value)**

It is able to add the provided key-value to the Map object.

EX:

---------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "Durga");

map.put(102, "Ramesh");

map.put(103, "Mahesh");

map.put(104, "Ramu");

map.put(105, "Harsha");

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

}

}

**2. public void putAll(Map map):**

It is able to keep all the key-value pairs of the specified Map object to the present map object.

EX:

—-------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

Map<Integer, String> map1 = **new** HashMap<>();

map1.putAll(map);

System.***out***.println(map1);

}

}

**3. public Object get(Object key):**

It is able to get value on the basis of the provided key.

EX:

—--------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

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

System.***out***.println(map.get(102));

System.***out***.println(map.get(103)); // Returns null

}

}

**4. public Object remove(Object key):**

It is able to remove a key-value pair on the basis of the specified key.

EX:

—-------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

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

System.***out***.println("Removed: " + map.remove(102));

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

}

}

**5. public int size():**

It is able to return the size of the Map, that is the number of key-Value pairs.

EX:

—-------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

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

System.***out***.println("Size: " + map.size());

}

}

**6. public boolean isEmpty():**

It is able to check whether the map is empty or not , if the map is empty then it will return true value, if the provided map is not empty then it will return false value.

EX:

—----------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

System.***out***.println("Is map empty? " + map.isEmpty());

map.put(101, "malli");

System.***out***.println("Is map empty? " + map.isEmpty());

}

}

**7. public void clear():**

It is able to remove all key-values from Map.

EX:

—--------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "mall");

map.put(102, "Ramesh");

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

map.clear();

System.***out***.println("After clearing: " + map);

}

}

**8. public boolean containsKey(Object key):**

It is able to check whether the specified key exists at the keys side or not. If the specified key exists at the keys side then it will return true value else false value.

EX:

—-------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

System.***out***.println("Contains key 101? " + map.containsKey(101));

System.***out***.println("Contains key 103? " + map.containsKey(103));

}

}

**9.** **public boolean containsValue(Object value):**

It is able to check whether the specified value exists or not at the values side. If the provided value exists at the value side then it will return true value otherwise false value.

EX:

—-----

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

System.***out***.println("Contains value 'Ramesh'? " + map.containsValue("Ramesh"));

System.***out***.println("Contains value 'Nag'? " + map.containsValue("Nag"));

}

}

**10. public Set keySet():**

It is able to return all the keys in the form of a Set.

EX:

—---

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

System.***out***.println("Keys: " + map.keySet());

}

}

**11. public Collection values():**

It is able to return all values in the form of a Collection object.

EX:

—------

**package** com.codegnan.collectionframework;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer, String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

System.***out***.println("Values: " + map.values());

}

}

**Entry :**

It is able to represent a single key-value pair in Maps.

To represent Entry , Java has provided a predefined interface inside Map interface in the form of java.util.Map.Entry.

Methods:

1. **To get all Entry objects from Map we have to use the following method**.

public Set entrySet()

***2.*** ***To get the key from the Entry we have to use the following method.***

public Object getKey()

**3.** **To get value from the Entry we have to use the following method.**

public Object getValue()

4. **To set value in Entry we will use the following method**. public void setValue(Object value)

EX:

—--

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** Main {

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

Map<Integer,String> map = **new** HashMap<>();

map.put(101, "malli");

map.put(102, "Ramesh");

map.put(103, "Mahesh");

map.put(104, "Ramu");

map.put(105, "Harsha");

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

Set set = map.entrySet();

**for** (Object obj : set) {

Map.Entry entry = (Map.Entry) obj;

Object key = entry.getKey();

**if** ((Integer) key == 103) {

entry.setValue("XXX");

}

Object value = entry.getValue();

System.***out***.println(key + " >" + value);

}

}

}

**HashMap:**

**—--------------**

1. It was introduced in JDK1.2 version.

2. It is an implementation class to Map interface.

3. It is able to store all elements in the form of Key-Value pairs.

4. In Key-Value pairs, both Keys and Values are Objects.

5. In HashMap, Keys must be unique , they must not be duplicated and values may be duplicated.

6. It will not follow Keys Insertion order and Sorting Order.

7. It allows heterogeneous elements at both the keys side and values side.

8. It allows only one null element at the keys side, but values are able to allow any number of null elements.

9. It is not a Synchronized map.

10. It allows more than one thread.

11. It follows parallel execution over the threads.

12. It will reduce application execution time.

13. It will increase application performance.

14. It does not give guarantee for Data Consistency.

15. It is not Thread safe.

16. Its initial capacity is 16 elements.

17. Its initial load factor is 75%.

18. It is suggestible when we want to perform frequent search operations.

19. Its internal data structure is Hashtable.

**Constructors:**

**1. public HashMap():**

It is able to create an empty HashMap object with 16 initial capacity and with 75% load factor.

EX:

**import** java.util.HashMap;

**public** **class** Main {

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

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

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

}

}

**2. public HashMap(int capacity):**

It is able to create an empty HashMap with the specified capacity value and with the default load factor 75%.

EX:

**import** java.util.HashMap;

**public** **class** Main {

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

HashMap hashMap = **new** HashMap(20);

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

}

}

**3. public HashMap(int capacity, float loadFactor):**

It is able to create an empty HashMap object with the specified capacity value and with the specified load factor .

EX:

**import** java.util.HashMap;

**public** **class** Main {

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

HashMap hashMap = **new** HashMap(20, 0.80f);

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

}

}

**4. public HashMap(Map map):**

It is able to create a HashMap object with all the key-value pairs of the specified Map object.

EX:

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map<Integer,String>map = **new** HashMap<>();

map.put(1, "AAA");

map.put(2, "BBB");

map.put(3, "CCC");

map.put(4, "DDD");

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

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

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

}

}

Ex:

-------

**package** com.codegnan.collectionframeworks;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Main {

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

Map <Integer,String>map = **new** HashMap<>();

map.put(1, "AAA");

map.put(2, "BBB");

map.put(3, "CCC");

map.put(4, "DDD");

map.put(5, "EEE");

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

map.put(2, "XXX");

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

map.put(6, "CCC");

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

map.put(**null**, "FFF");

map.put(**null**, "GGG");

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

map.put(7, **null**);

map.put(8, **null**);

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

map.put(**new** StringBuffer("XXX"), **new** ~~Integer~~(1000));

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

}

}

**LinkedHashMap:**

**—---------------------**

**Q) What are the differences between HashMap and LinkedHashMap?**

**—---------------------------------------------**

**Ans:**

**—--------------------------**

1. HashMap was introduced in JDK1.2 version. LinkedHashMap was introduced in JDK1.4 version.

2. HashMap is not following insertion order. LinkedHashMap follows insertion order.

3. HashMap has Hashtable internally as a data structure. LinkedHashMap has Hashtable + LinkedList as internal Data structure.

Note: LinkedHashMap is a child class to HashMap.

Ex

----------

**package** com.codegnan.collectionframeworks;

**import** java.util.HashMap;

**import** java.util.LinkedHashMap;

**public** **class** Main {

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

HashMap<String,String> hashMap = **new** HashMap<>();

hashMap.put("AAA", "ABC");

hashMap.put("BBB", "BCD");

hashMap.put("CCC", "CDE");

hashMap.put("DDD", "DEF");

hashMap.put("EEE", "EFG");

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

LinkedHashMap<String,String> linkedHashMap = **new** LinkedHashMap<>();

linkedHashMap.put("AAA", "ABC");

linkedHashMap.put("BBB", "BCD");

linkedHashMap.put("CCC", "CDE");

linkedHashMap.put("DDD", "DEF");

linkedHashMap.put("EEE", "EFG");

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

}

}

**SortedMap:**

**— --------------**

1. It was introduced in JDK1.2 version.

2. It is not a Legacy Map.

3. It is a child interface to the Map interface.

4. It is able to store all elements in the form of Key-Value pairs.

5. In Key-Value pairs , both Keys and values are Objects.

6. In Key-Value pairs, keys must be unique and keys must not be duplicated, but values may be duplicated.

7. It does not follow Keys insertion order.

8. It follows the keys' sorting order.

9. It does not allow null elements at the keys side, if we add null at key then JVM will raise an exception like java.lang.NullPointerException.

10. It allows only Homogeneous elements at the keys side, but Values are able to allow heterogeneous elements.

11. It allows only Comparable elements at keys side, it is not allowing non comparable elements at keys side, if we want to add non comparable elements at keys side then we must use java.util.Copmparator.

**Methods:**

**— --------------**

**1. public Object firstKey():**

It is able to return the first key from the SOrtedMap.

**2. public Object lastKey():**

It is able to return the last key from the SortedMap.

**3. public SortedMap headMap(Object key):**

It is able to return a SortedMap object containing key-value pairs where all the keys must be less than the specified key.

**4. public SortedMap tailMap(Object key):**

It is able to return a SortedMap object containing key-value pairs , where all the keys must be greater or equal to the specified key.

**5. public SortedMap subMap(Object key1, Object k2):**

It is able to return a SortedMap containing key-value pairs , where all the keys must be greater than or equal to the specified key1 and less than specified key2.

EX:

--------

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

SortedMap<String,String> map = **new** TreeMap<>();

map.put("AAA", "111");

map.put("FFF", "666");

map.put("BBB", "222");

map.put("EEE", "555");

map.put("CCC", "333");

map.put("DDD", "444");

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

System.***out***.println(map.firstKey());

System.***out***.println(map.lastKey());

System.***out***.println(map.headMap("DDD"));

System.***out***.println(map.tailMap("DDD"));

System.***out***.println(map.subMap("BBB", "EEE"));

}

}

OP:

{AAA=111, BBB=222, CCC=333, DDD=444, EEE=555, FFF=666}

AAA

FFF

{AAA=111, BBB=222, CCC=333}

{DDD=444, EEE=555, FFF=666}

{BBB=222, CCC=333, DDD=444}

**NavigableMap:**

**— ---------------**

It was provided by JAVA in its JDK1.6 version, it is a child interface to SortedMap, that is

, it is following all the features of SortedMap but it has define some extra behaviors to perform navigation over the elements like Getting descending maps, finding ceiling elements, higher elements, floor elements , lower elements,.......

**Methods:**

**1. public NavigableMap descendingMap():**

It is able to generate a NavigableMap containing all key-value pairs , where all the key-value pairs must be in Descending order.

**2. public Object ceilingKey(Object key):**

It is able to return a lowest key among all the keys which are greater than or equal to the specified key.

**3. public Object higherKey(Object key):**

It is able to return the lowest key among all the keys which are greater than the specified key.

**4. public Object floorKey(Object key):**

It is able to return a higher key among all the keys which are less than or equal to the specified key.

**5. public Object lowerKey(Object key):**

It is able to return a higher key among all the keys which are less than the specified key.

**6. public Entry pollFirstEntry():**

It is able to remove the first entry from NavigableMap.

**7. public Entry pollLastEntry():**

It is able to remove the last Entry from NavigableMap.

Ex:

---

**package** com.codegnan.collectionframeworks;

**import** java.util.\*;

**public** **class** Main {

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

NavigableMap <String,String>map = **new** TreeMap<>();

map.put("AAA", 76);

map.put("FFF", 89);

map.put("BBB", 88);

map.put("EEE", 67);

map.put("CCC", 100);

map.put("DDD", 56);

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

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

System.***out***.println(map.ceilingKey("DDD"));// DDD

System.***out***.println(map.higherKey("DDD"));// EEE

// System.out.println(map.floorKey("DDD"));// DDD

// System.out.println(map.lowerKey("DDD"));//CCC

// System.out.println(map.pollFirstEntry());

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

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

}

}

OP:

—----------

{AAA=76, BBB=88, CCC=100, DDD=56, EEE=67, FFF=89}

{FFF=89, EEE=67, DDD=56, CCC=100, BBB=88, AAA=76} DDD

EEE DDD CCC AAA=76 FFF=89

{BBB=88, CCC=100, DDD=56, EEE=67}

**TreeMap:**

**— ----------**

1. It was introduced in JDK1.2 version

2. It is not a legacy Map.

3. It is a direct implementation class to NavigableMap, where NavigableMap is a child interface to SortedMap, where SortedMap is a child interface to Map interface, that is, TreeMap must implement all the methods of NavigableMap, SortedMap and Map.

4. It is able to manage all the elements in the form of Key-Value pairs, where both Keys and values must be objects.

5. In Key-Value pairs, Keys must be unique, keys must not be duplicated, but values may be duplicated.

6. TreeMap does not follow Insertion order.

7. TreeMap is able to follow Sorting order.

8. TreeMap is able to allow only Homogeneous elements at the keys side, but it allows heterogeneous elements at the values side.

9. TreeMap is not allowing null at the keys side, but it allows any number of null elements at the values side.

10. It is able to allow only Comparable elements at keys side, any type of elements are allowed at values side, if we want to add non comparable elements to TreeMap at keys side then we must use java.util.Comparator.

11. It is not a synchronized Map.

12. It allows more than one thread at a time to access data.

13. It follows parallel execution.

14. It will reduce application execution time.

15. It will improve application performance.

16. It is not a guarantee for Data Consistency.

17. It is not thread safe.

18. Its internal data structure is “Red-Black Tree”.

**Constructors:**

**1. public TreeMap()**

It is able to create an empty TreeMap with the default natural sorting order.

**2. public TreeMap(Comparator c):**

It is able to create an empty TreeMap object with the explicit sorting logic in the form of Comparator.

**3. public TreeMap(SortedMap map):**

It is able to create a TreeMap object with all the elements of the specified SortedMap.

**4. public TreeMap(Map map):**

It is able to create a TreeMap object with all the elements of the specified Map.

EX:

—-

**package** com.codegnan.collectionframeworks;

**import** java.util.TreeMap;

**class** A {

}

**public** **class** Main {

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

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

map.put("AAA", "111");

map.put("EEE", "222");

map.put("BBB", "333");

map.put("DDD", "444");

map.put("CCC", "555");

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

//map.put(null, "666"); --> NullPointerException

//map.put(10, 100); --> ClASScASTeXCEPTION

//map.put(new A(), "666");---> ClassCastException

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

}

}

**Q)Write a Java program to sort all the Student Names as per Student roll numbers in Descending order?**

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

**Ans:**

**package** com.codegnan.collectionframeworks;

**import** java.util.Comparator;

**import** java.util.TreeMap;

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

@Override

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

Integer in1 = (Integer) o1;

Integer in2 = (Integer) o2;

**int** val = 0;

**if** (in1 < in2) {

val = -100;

} **else** **if** (in1 > in2) {

val = 100;

} **else** {

val = 0;

}

**return** -val;

}

}

**public** **class** Main {

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

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

treeMap.put(111, "malli");

treeMap.put(666, "Anil");

treeMap.put(222, "Rakesh");

treeMap.put(555, "Deepak");

treeMap.put(333, "Venkatesh");

treeMap.put(444, "Mahesh");

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

}

}

Ex

-------

**package** com.codegnan.collectionframeworks;

**import** java.util.Comparator;

**import** java.util.TreeMap;

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

@Override

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

Integer in1 = (Integer) o1;

Integer in2 = (Integer) o2;

**int** val = 0;

**if** (in1 < in2) {

val = -100;

} **else** **if** (in1 > in2) {

val = 100;

} **else** {

val = 0;

}

**return** -val;

}

}

**public** **class** Main {

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

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

treeMap.put(111, "Durga");

treeMap.put(666, "Anil");

treeMap.put(222, "Rakesh");

treeMap.put(555, "Deepak");

treeMap.put(333, "Venkatesh");

treeMap.put(444, "Mahesh");

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

}

}

Ex

---

**package** com.codegnan.collectionframeworks;

**import** java.util.Comparator;

**import** java.util.TreeMap;

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

@Override

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

String state1 = (String) o1;

String state2 = (String) o2;

**int** val = state1.compareTo(state2);

**return** -val;

}

}

**public** **class** Main {

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

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

treeMap.put("Telangana", 1000);

treeMap.put("AP", 2000);

treeMap.put("Karnataka", 3000);

treeMap.put("UP", 4000);

treeMap.put("Delhi", 5000);

treeMap.put("Tamilnadu", 6000);

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

}

}

**Hashtable:**

**—---------**

**Q) What are the differences between HashMap and Hashtable?**

**—--------------------------------------------------------------------**

**Ans:**

**—---**

1. HashMap was introduced in JDK1.2 version. Hashtable was introduced in JDK1.0 version.

2. HashMap was not a legacy Map. Hashtable was a legacy Map.

3. HashMap is able to allow one null element at the keys side and any number of null elements are value side.

Hashtable is not allowing null elements at both keys and values. If we add null element then JVM will raise an exception like java.lang.NullPointerException

4. HashMap is not Synchronized. Hashtable is synchronized.

5. HashMap is able to allow more than one thread at a time to access data. Hashtable is able to allow only one thread at a time to access data.

6. HashMap is able to follow parallel execution. Hashtable is able to follow sequential execution.

7. HashMap is able to reduce application execution time. Hashtable is able to increase application execution time.

8. HashMap is able to increase application performance. Hashtable is able to reduce application performance.

9. HashMap is not giving guarantees for Data Consistency. Hashtable is a guarantee for Data consistency.

10. HashMap is not thread safe. Hashtable is Thread safe.

EX:

—---

**package** com.codegnan.collectionframeworks;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** Main {

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

HashMap hashmap = **new** HashMap();

hashmap.put("A", "AAA");

hashmap.put("B", "BBB");

hashmap.put("C", "CCC");

hashmap.put("D", "DDD");

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

hashmap.put(**null**, "EEE");

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

hashmap.put("E", **null**);

hashmap.put("F", **null**);

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

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

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

hashtable.put("A", "AAA");

hashtable.put("B", "BBB");

hashtable.put("C", "CCC");

hashtable.put("D", "DDD");

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

//hashtable.put(null, "EEE"); --> NullPointerException

//hashtable.put("E", null); >NullPointerException

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

}

}

**Conversions**:

* **List to Set**: Converts a list to a Set, removing duplicates.
* **Set to List**: Converts the Set back to a List.
* **List to Queue**: Converts a list to a Queue.
* **Queue to List**: Converts the Queue back to a List.
* **List to Deque**: Converts a list to a Deque.
* **Deque to List**: Converts the Deque back to a List.
* **Map Initialization**: Creates a Map with key-value pairs.
* **Map Keys to List**: Converts the keys of a Map to a List.
* **Map Values to List**: Converts the values of a Map to a List.
* **Map Entries to List**: Converts the entries of a Map to a List.
* **List to Array**: Converts a List to an array.
* **Set to Array**: Converts a Set to an array.
* **Queue to Array**: Converts a Queue to an array.
* **Deque to Array**: Converts a Deque to an array.
* **Map Keys to Array**: Converts the keys of a Map to an array.
* **Map Values to Array**: Converts the values of a Map to an array.
* **Using Streams**: Demonstrates converting a list to a set and a set back to a list using Java Streams.

**package** com.codegnan.collectionframework;

**import** java.util.ArrayDeque;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Deque;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.LinkedList;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Queue;

**import** java.util.Set;

**import** java.util.stream.Collectors;

**public** **class** CollectionAndArrayConversionExample {

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

// Initialize a List

List<String> list = Arrays.*asList*("apple", "banana", "cherry");

System.***out***.println("Original List: " + list);

// 1. Convert List to Set

Set<String> setFromList = **new** HashSet<>(list);

System.***out***.println("Set from List: " + setFromList);

// 2. Convert Set to List

List<String> listFromSet = **new** ArrayList<>(setFromList);

System.***out***.println("List from Set: " + listFromSet);

// 3. Convert List to Queue

Queue<String> queueFromList = **new** LinkedList<>(list);

System.***out***.println("Queue from List: " + queueFromList);

// 4. Convert Queue to List

List<String> listFromQueue = **new** ArrayList<>(queueFromList);

System.***out***.println("List from Queue: " + listFromQueue);

// 5. Convert List to Deque

Deque<String> dequeFromList = **new** ArrayDeque<>(list);

System.***out***.println("Deque from List: " + dequeFromList);

// 6. Convert Deque to List

List<String> listFromDeque = **new** ArrayList<>(dequeFromList);

System.***out***.println("List from Deque: " + listFromDeque);

// 7. Initialize a Map

Map<String, Integer> map = **new** HashMap<>();

map.put("one", 1);

map.put("two", 2);

map.put("three", 3);

System.***out***.println("Original Map: " + map);

// 8. Convert Map keys to List

List<String> keysList = **new** ArrayList<>(map.keySet());

System.***out***.println("List from Map keys: " + keysList);

// 9. Convert Map values to List

List<Integer> valuesList = **new** ArrayList<>(map.values());

System.***out***.println("List from Map values: " + valuesList);

// 10. Convert Map entries to List

List<Map.Entry<String, Integer>> entriesList = **new** ArrayList<>(map.entrySet());

System.***out***.println("List from Map entries: " + entriesList);

// 11. Convert List to Array

String[] arrayFromList = list.toArray(**new** String[0]);

System.***out***.println("Array from List: " + Arrays.*toString*(arrayFromList));

// 12. Convert Set to Array

String[] arrayFromSet = setFromList.toArray(**new** String[0]);

System.***out***.println("Array from Set: " + Arrays.*toString*(arrayFromSet));

// 13. Convert Queue to Array

String[] arrayFromQueue = queueFromList.toArray(**new** String[0]);

System.***out***.println("Array from Queue: " + Arrays.*toString*(arrayFromQueue));

// 14. Convert Deque to Array

String[] arrayFromDeque = dequeFromList.toArray(**new** String[0]);

System.***out***.println("Array from Deque: " + Arrays.*toString*(arrayFromDeque));

// 15. Convert Map keys to Array

String[] keysArray = keysList.toArray(**new** String[0]);

System.***out***.println("Array from Map keys: " + Arrays.*toString*(keysArray));

// 16. Convert Map values to Array

Integer[] valuesArray = valuesList.toArray(**new** Integer[0]);

System.***out***.println("Array from Map values: " + Arrays.*toString*(valuesArray));

// 17. Using Streams to convert List to Set

Set<String> setFromListStream = list.stream().collect(Collectors.*toSet*());

System.***out***.println("Set from List using Streams: " + setFromListStream);

// 18. Using Streams to convert Set to List

List<String> listFromSetStream = setFromListStream.stream().collect(Collectors.*toList*());

System.***out***.println("List from Set using Streams: " + listFromSetStream);

}

}

How To get Synchronized In CollectionFramework

============================================

To achieve synchronization in the Java Collections Framework, you can use the Collections.synchronizedList(), Collections.synchronizedMap(), or similar methods to create synchronized versions of collection objects. Here are examples for different collection types, including ArrayList, HashMap, and more.

### **1. Synchronized ArrayList**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class SynchronizedArrayListExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

List<String> synchronizedList = Collections.synchronizedList(list);

synchronizedList.add("Durga");

synchronizedList.add("Ramesh");

synchronizedList.add("Mahesh");

// Synchronized block to iterate over synchronized collection

synchronized (synchronizedList) {

for (String name : synchronizedList) {

System.out.println(name);

}

}

}

}

### **2. Synchronized HashMap**

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class SynchronizedHashMapExample {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<>();

Map<Integer, String> synchronizedMap = Collections.synchronizedMap(map);

synchronizedMap.put(1, "Durga");

synchronizedMap.put(2, "Ramesh");

synchronizedMap.put(3, "Mahesh");

// Synchronized block to iterate over synchronized collection

synchronized (synchronizedMap) {

for (Map.Entry<Integer, String> entry : synchronizedMap.entrySet()) {

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

}

}

}

}

### **3. Synchronized HashSet**

import java.util.Collections;

import java.util.HashSet;

import java.util.Set;

public class SynchronizedHashSetExample {

public static void main(String[] args) {

Set<String> set = new HashSet<>();

Set<String> synchronizedSet = Collections.synchronizedSet(set);

synchronizedSet.add("Durga");

synchronizedSet.add("Ramesh");

synchronizedSet.add("Mahesh");

// Synchronized block to iterate over synchronized collection

synchronized (synchronizedSet) {

for (String name : synchronizedSet) {

System.out.println(name);

}

}

}

}