**Collections Framework**

**What is Collection?**

Collections framework is nothing but handling individual Objects (Collection Interface) and Group of objects (Map interface).

We know only object can move from one network to another network.

A collections framework is a class library to handle group of Objects.

It is implemented by using java.util package.

It provides an architecture to store and manipulate group of objects.

All the operations that we can perform on data such as searching, sorting, insertion and deletion can be done by using collections framework because It is the data structure of Java.

The simple meaning of collections is single unit of Objects.

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

It provides the following sub interfaces:

**1) List** (Accept duplicate elements)

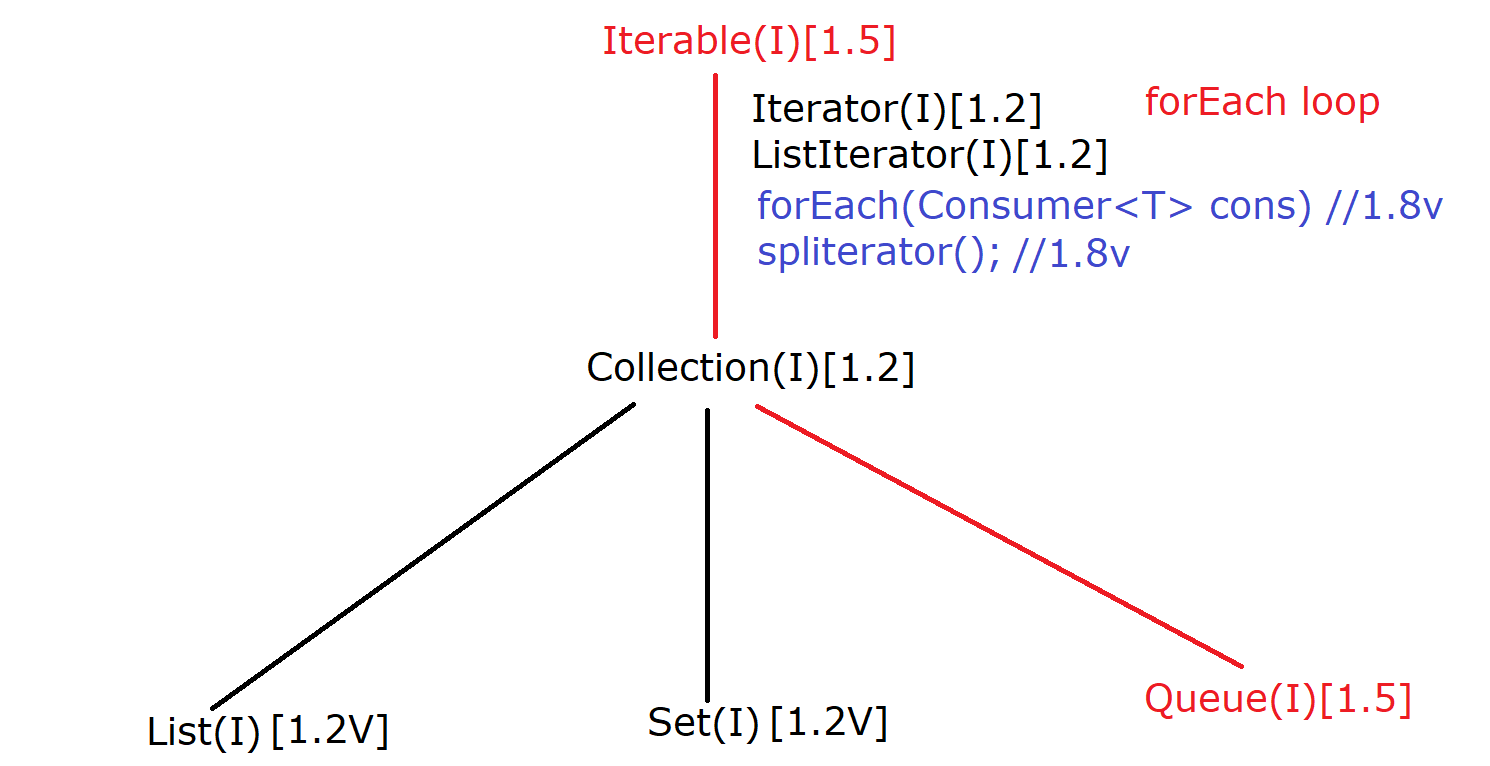
**2) Set** (Not accepting duplicate elements)

**3) Queue** (Storing and Fetching the elements based on some order i.e FIFO)

***Note:*** *Collection is a predefined interface available in java.util package whereas Collections is a predefined utility class which is available from JDK 1.2V which contains only static methods (Constructor is private)*

Collection Hierarchy

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



**Methods of Collection interface**:

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

a) **public boolean add(E element)** : It is used to add an item/element in the collection.

b) **public boolean addAll(Collection c)** : It is used to insert the specified collection elements in the existing collection(For merging the Collection)

c) **public boolean retainAll(Collection c)** :- It is used to retain all the elements from existing element. (Common Element)

d) **public boolean removeAll(Collection c)** :- It is used to delete all the elements from the existing collection.

e) **public boolean remove(Object element)** :- It is used to delete an element from the collection based on the object.

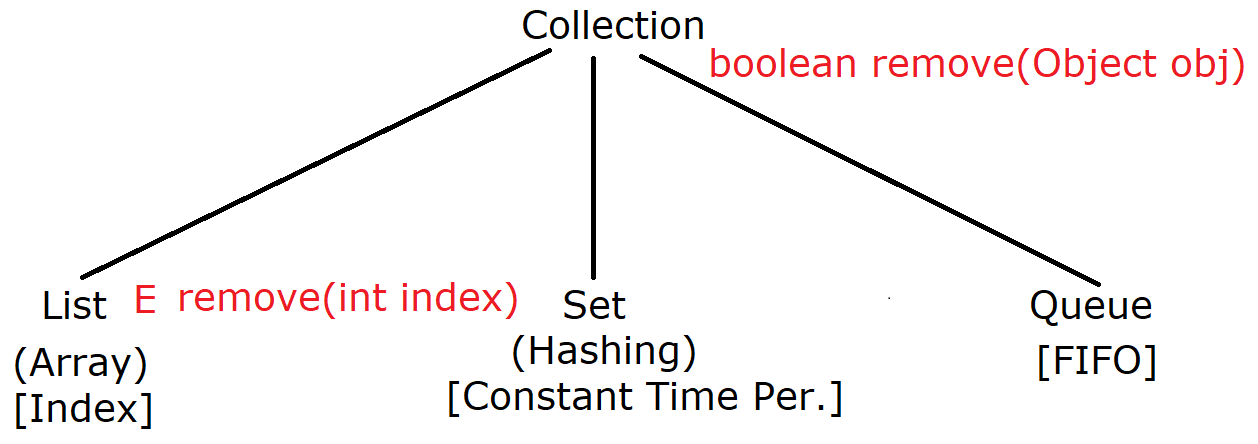
f) **public int size()** :- It is used to find out the size of the Collection [Total number of elements available]

g) **public void clear()** :- It is used to clear all the elements at once from the Collection.

All the above methods of Collection interface will be applicable to all the sub interfaces like List, Set and Queue.

----

**Method collection interface**

****

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

**List<E> interface**

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

It is the sub interface of Collection interface introduced from JDK 1.2V.

It is internally an Array so it stores the object in a sequence order by using index.

Here we can store the object by using index because List interface has provided add(int index, E element) method which will add the object based on the index position.

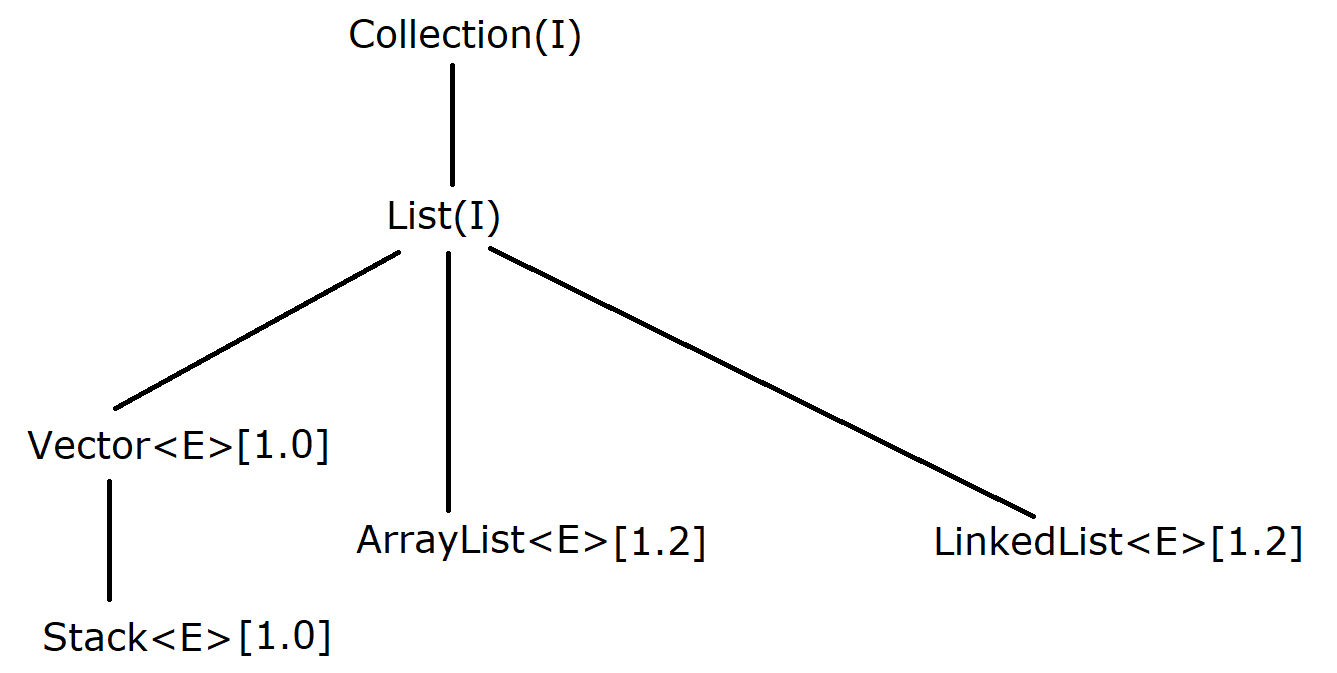
List interface can accept duplicate elements.

We can perform sorting operation directly by using **sort(Comparator<T> cmp)** method or by using Collections.sort(List<E> list) interface as a parameter.

We can iterate the elements of List interface by using Iterator and ListIterator interface.

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

**List interface Hierarchy**

****

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

**Behaviour of List interface Specific classes**

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

1) It stores the elements on the basis of index because internally it is using array concept.

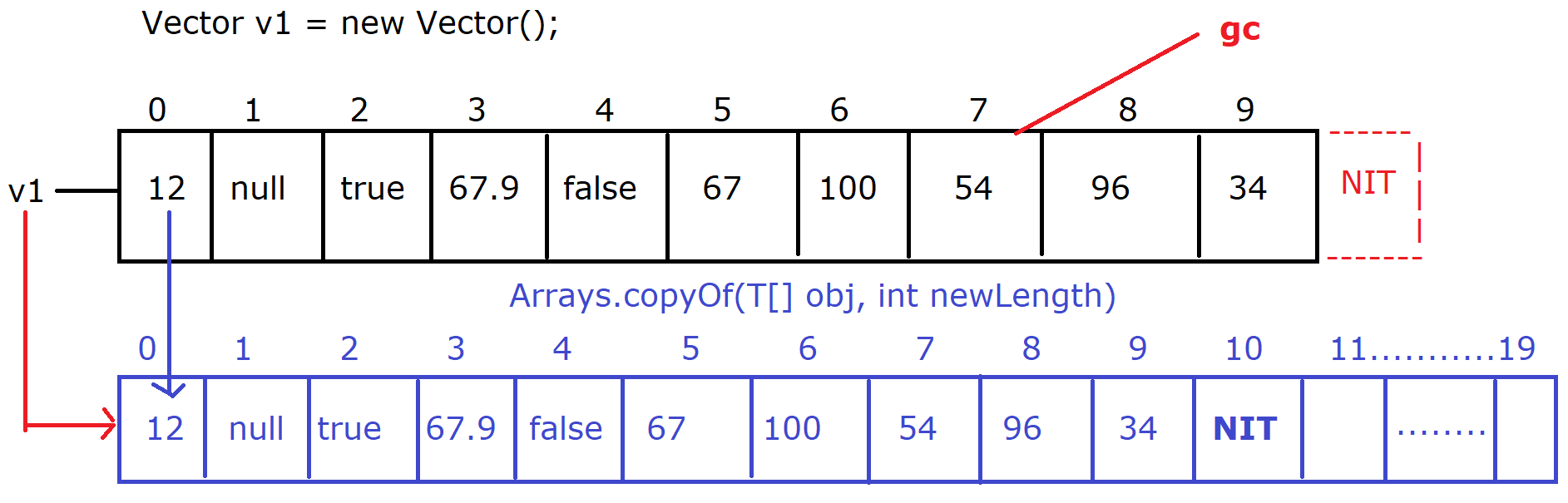
2) It can accept duplicate, homogeneous and heterogeneous elements.

3) It stores everything in the form of Object.

4) When we accept the collection classes without generic concept then compiler generates a warning message because It is unsafe object.

5) By using **generic (<>)** (JDK 1.5V) we can eliminate compilation warning and still we can take homogeneous as well as heterogeneous. (Vector<Object>)

6) In list interface few classes are dynamically Growable like Vector and ArrayList.



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

***List<E> interface Methods***

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

1) **public boolean isEmpty()** :- Verify whether List is empty or not

2) **public void clear(**) :- Will clear all the elements, Basically List will become empty.

3) **public int size()** :- To get the size of the Collections(Total number of elements are available in the collection)

4) **public void add(int index, Object o)** :- Insert the element based on the index position.

5) **public boolean addAll(int index, Collection c)** :- Insert the Collection based on the index position

6) **public Object get(int index)** :- To retrieve the element based on the index position

7) **public Object set(int index, Object o)** :- To override or replace the existing element based on the index position

8) **public Object remove(int index)** :- remove the element based on the index position

9) **public boolean remove(Object element)** :- remove the element based on the object element, It is the Collection interface method extended by List interface

10) **public int indexOf()** :- index position of the element

11) **public int lastIndex()** :- last index position of the element

12) **public Iterator iterator()** :- To fetch or iterate or retrieve the elements from Collection in forward direction only.

13) **public ListIterator listIterator()** :- To fetch or iterate or retrieve the elements from Collection in forward and backward direction.

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

**How many ways we can fetch OR retrieve the Object from the Collection:**

There are 9 ways to fetch OR retrieve the Collection Object :

1) By using toString() method of respective class.[JDK 1.0]

2) By using Ordinary for loop. [JDK 1.0]

3) By using for each loop.[JDK 1.5]

4) By using Enumeration<E> interface. [JDK 1.0]

5) By using Iterator<E> interface. [JDK 1.2]

6) By using ListIterator<E> interface. [JDK 1.2]

7) By using SplIteratot<E> interface. [JDK 1.8]

8) By using forEach(Consumer<T> cons) [JDK 1.8]

9) By using Method Reference [JDK 1.8]

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

***Note*** *Among all these 9 ways Enumeration<E>, Iterator<E>, ListIterator<E> and SplIterator<E> are the cursors so it can move from one direction to another.*

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

***Assignment*** *: WAP to describe all these 9 ways to fetch the Collection Object*

| **Sr.** | **Method** | **Version** | **Explanation (English)** | **Example (Java Code)** |
| --- | --- | --- | --- | --- |
| 1️⃣ | toString() | JDK 1.0 | Converts collection to string (e.g., [A, B, C]) | System.out.println(list); |
| 2️⃣ | Ordinary for loop | JDK 1.0 | Index-based loop (only for List) | for (int i = 0; i < list.size(); i++) { System.out.println(list.get(i)); } |
| 3️⃣ | Enhanced for-each loop | JDK 1.5 | Simplified loop for all collections | for (String s : list) { System.out.println(s); } |
| 4️⃣ | Enumeration<E> | JDK 1.0 | Legacy interface, used with Vector | Enumeration<String> e = vector.elements(); while (e.hasMoreElements()) { System.out.println(e.nextElement()); } |
| 5️⃣ | Iterator<E> | JDK 1.2 | Universal for all collections | Iterator<String> it = list.iterator(); while (it.hasNext()) { System.out.println(it.next()); } |
| 6️⃣ | ListIterator<E> | JDK 1.2 | Only for List, supports bidirectional traversal | ListIterator<String> lit = list.listIterator();  while (lit.hasNext()) { System.out.println(lit.next()); } |
| 7️⃣ | Spliterator<E> | JDK 1.8 | Supports parallel traversal | Spliterator<String> sp = list.spliterator(); sp.forEachRemaining(System.out::println); |
| 8️⃣ | forEach(Consumer<T>) | JDK 1.8 | Lambda with action | list.forEach(s -> System.out.println(s)); |
| 9️⃣ | Method Reference | JDK 1.8 | Shortcut for lambda (e.g., System.out::println) | list.forEach(System.out::println); |

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

***Enumeration***

-----------

It is a predefined interface available in java.util package from JDK 1.0 onwards(Legacy interface).

We can use Enumeration interface to fetch or retrieve the Objects one by one from the Collection because it is a cursor.

We can create Enumeration object by using elements() method of the legacy Collection class. Internally it uses anonymous inner class object.

public Enumeration elements();

***Enumeration interface contains two methods :***

1) **public Enumeration elements();** // Returns the Enumeration object

2) **public boolean hasMoreElements();** // Checks if more elements are available

3) **public Object nextElement();** // Returns next element from collection

+------------------------------+

| Hyderabad [0] | <== First element

| Kolkata [1] |

| Pune |

| Indore |

| Mumbai | <== Last element

+------------------------------+

***Note*** *It will only work with legacy Collections classes.*

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

***Iterator<E>***

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

It is a predefined interface available in java.util package available from 1.2 version.

It is used to fetch/retrieve the elements from the Collection in forward direction only because it is also a cursor.

It is also using private inner class i.e Itr class.

public Iterator iterator();

***Example***

**Iterator itr = listOfCity.iterator();**

*Now, Iterator interface has provided two methods*

**public boolean hasNext()** :- It will verify, the element is available in the next position or not, if available it will return true otherwise it will return false.

**public Object next()** :- It will return the collection object and move the cursor to the element object.

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

**ListIterator<E> interface**

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

It is a predefined interface available in java.util package and it is the sub interface of Iterator available from JDK 1.2v.

It is used to retrieve the Collection object in both the direction i.e in forward direction as well as in backward direction. Here the inner class name is LstItr class extends from Itr class.

**public ListIterator listIterator();**

***Example***

**ListIterator lit = fruits.listIterator();**

**1) public boolean hasNext()** :- It will verify the element is available in the next position or not, if available it will return true otherwise it will return false.

2) **public Object next()** :- It will return the next position collection object.

3) **public boolean hasPrevious**() :- It will verify the element is available in the previous position or not, if available it will return true otherwise it will return false.

4) **public Object previous()** :- It will return the previous position collection object.

***Note*** *Apart from these 4 methods we have* ***add(), set()*** *and* ***remove()*** *method in ListIterartor interface.*

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

**SplIterator interface**

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

It is a predefined interface available in java.util package from java 1.8 version.

It is a cursor through which we can fetch the elements from the Collection [Collection, array, Stream]

It is the combination of **hasNext()** and **next()** method.

It is using forEachRemaining(Consumer<T> cons) method for fetching the elements.

----

package com.ravi.collection;

import java.util.Enumeration;

import java.util.Iterator;

import java.util.ListIterator;

import java.util.Spliterator;

import java.util.Vector;

public class RetrievingCollectionObject

{

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Kolkata");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Mumbai");

System.out.println("1) By using toString() method :");

System.out.println(listOfCity.toString());

*2) By using Ordinary for loop :*

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

{

System.out.println(listOfCity.get(i));

}

*3) By using for Each loop :*

for(String city : listOfCity)

{

System.out.println(city);

}

*4) By using Enumeration interface* :

Enumeration<String> ele = listOfCity.elements();

while(ele.hasMoreElements())

{

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

}

*5) By using Iteartor interface :*

Iterator<String> itr = listOfCity.iterator();

itr.forEachRemaining(city -> System.out.println(city));

*6) By using ListIteartor interface :*

ListIterator<String> lstItr = listOfCity.listIterator();

System.out.println("IN FORWARD DIRECTION ..");

while(lstItr.hasNext())

{

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

}

System.out.println("IN BACKWARD DIRECTION ..");

while(lstItr.hasPrevious())

{

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

}

*7) By using SplIterator interface :*

Spliterator<String> splItr = listOfCity.spliterator();

splItr.forEachRemaining(city -> System.out.println(city));

*8) By using forEach() Method:*

listOfCity.forEach(city -> System.out.println(city));

*9) By using Method Reference:*

listOfCity.forEach(System.out::println);

}

}

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

***Q. How forEach(Consumer<T>) method works internally ?***

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

forEach() method is available from JDK 1.8V.

This forEach() method is used iterate the elements from the source.

*This forEach() method is available in the following 3 palces :*

*1) Iterable (JDK 1.5)*

*2) Map interface (JDK 1.2)*

*3) Stream interface (JDK 1.8)*

Here, we are talking about forEach(Consumer<T> cons) method of Iterable interface.

forEach(Consumer<T> cons) method of Iterable interface internally uses for each loop.

---

***Case 1:*** By using Anonymous inner class :

package com.ravi.collection;

import java.util.Vector;

import java.util.function.Consumer;

public class ForEachMethodInternalDemo1 {

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Kolkata");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Mumbai");

//Anonymous inner class for Consumer<T>

Consumer<String> cons = new Consumer<String>()

{

@Override

public void accept(String city)

{

System.out.println(city.toUpperCase());

}

};

listOfCity.forEach(cons);

}

}

---

***Case 2***

package com.ravi.collection;

import java.util.Vector;

import java.util.function.Consumer;

public class ForEachMethodInternalDemo1 {

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Kolkata");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Mumbai");

//Lambda Expression

Consumer<String> cons = city -> System.out.println(city.toUpperCase());

listOfCity.forEach(cons);

}

}

---

***Case 3***

package com.ravi.collection;

import java.util.Vector;

import java.util.function.Consumer;

public class ForEachMethodInternalDemo1 {

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Kolkata");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Mumbai");

listOfCity.forEach(city -> System.out.println(city.toUpperCase()));

}

}

---

***Case 4*** *[Method Reference]*

package com.ravi.collection;

import java.util.Vector;

import java.util.function.Consumer;

public class ForEachMethodInternalDemo1 {

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Kolkata");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Mumbai");

listOfCity.forEach(System.out::println);

}

}

**============================================**

**Working with List interface Specific classes**

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

As we know, in List interface we have 4 implemented classes which are as follows:

***1) Vector<E>***

***2) Stack<E>***

***3) ArrayList<E>***

***4) LinkedList<E>***

---------

**Vector<E>**

*public class Vector<E> extends AbstractList<E> implements List<E>, Serializable, Clonable, RandomAccess*

Vector is a predefined class available in java.util package under List interface.

Vector is always from java means it is available from jdk 1.0 version.

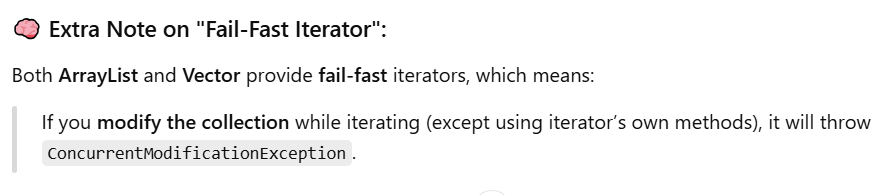
It can accept duplicate, null, homogeneous as well as hetrogeneous elements.

Vector and Hashtable, these two classes are available from jdk 1.0, remaining Collection classes were added from 1.2 version. That is the reason Vector and Hashtable are called legacy(old) classes.

The main difference between Vector and ArrayList is, ArrayList methods are not synchronized so multiple threads can access the method of ArrayList whereas on the other hand most the methods are synchronized in Vector so performance wise Vector is slow.

\*We should go with ArrayList when Threadsafety is not required on the other hand we should go with Vector when we need ThreadSafety for reterival operation.

Here Iterator is **Fail Fast Iterator**.



It stores the elements on index basis. It is dynamically growable with initial capacity 10. The next capacity will be 20 i.e double of the first capacity.

*new capacity = current capacity \* 2;*

It implements List, Serializable, Clonable, RandomAccess interfaces.

**Constructors in Vector**

*We have 4 types of Constructor in Vector*

1**) Vector v1 = new Vector()**;

It will create the vector object with default capacity is 10

2) **Vector v2 = new Vector(int initialCapacity);**

Will create the vector object with user specified capacity.

3) **Vector v3 = new Vector(int initialCapacity, int capacityIncrement);**

Eg :- Vector v = new Vector(1000,5);

*Initially It will create the Vector Object with initial capacity 1000 and then when the capacity will be full then increment by 5 so the next capacity would be 1005, 1010 and so on.*

4) **Vector v4 = new Vector(Collection c);**

We can achieve loose coupling

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

package com.ravi.vector;

import java.util.Collections;

import java.util.Vector;

public class VectorDemo {

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Bhubneswar");

listOfCity.add("Kolkata");

System.out.println("Before Sorting :"+listOfCity);

listOfCity.sort((s1,s2)-> s1.compareTo(s2));

System.out.println("After Sorting :"+listOfCity);

//Remove the element based on the index position

listOfCity.remove(2);

System.out.println(listOfCity);

//Remove based on the Object

listOfCity.remove("Kolkata");

System.out.println(listOfCity);

}

}

---

*//Vector Program on capacity*

package com.ravi.vector;

import java.util.\*;

public class VectorDemo1 {

public static void main(String[] args)

{

Vector<Integer> v = new Vector<>(100,10);

System.out.println("Initial capacity is :" + v.capacity());

for (int i = 0; i < 100; i++)

{

v.add(i);

}

System.out.println("After adding 100 elements capacity is :" + v.capacity());

v.add(101);

System.out.println("After adding 101th elements capacity is :" + v.capacity());

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

{

if(i%5==0)

{

System.out.println();

}

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

}

}

}

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

package com.ravi.vector;

*//Array To Collection*

import java.util.\*;

public class VectorDemo2

{

public static void main(String args[])

{

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

int x[]={22,20,10,40,15,58};

//Adding array values to Vector

for(int i=0; i<x.length; i++)

{

v.add(x[i]);

}

Collections.sort(v);

System.out.println("Maximum element is :"+Collections.max(v));

System.out.println("Minimum element is :"+Collections.min(v));

System.out.println("Vector Elements :");

v.forEach(y -> System.out.println(y));

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

Collections.reverse(v);

v.forEach(y -> System.out.println(y));

//How to convert Collection to Array

Object[] array = v.toArray();

System.out.println(Arrays.toString(array));

}

}

--------

**1️⃣ Method: toArray()** // Converts collection to array  
**Example:**

List<String> list = new ArrayList<>();

list.add("A");

list.add("B");

String[] arr = list.toArray(new String[0]);

**2️⃣ Method: Collections.max()** //Finds maximum element  
**Example:**

List<Integer> list = Arrays.asList(10, 25, 5, 40);

int max = Collections.max(list); //Output: 40

**3️⃣ Method: Collections.min() //**Finds minimum element  
**Example:**

List<Integer> list = Arrays.asList(10, 25, 5, 40);

int min = Collections.min(list); //Output: 5

**4️⃣ Method: Collections.reverse()** //Reverses order of list elements  
**Example:**

List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));

Collections.reverse(list); // Output: [C, B, A]

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

**How to work with Custom Object**

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

package com.ravi.vector;

import java.util.Vector;

record Employee(Integer id, String name, Double salary)

{

}

public class VectorDemo3

{

public static void main(String[] args)

{

Vector<Employee> listOfEmployees = new Vector<>();

listOfEmployees.add(new Employee(333, "Scott", 800D));

listOfEmployees.add(new Employee(555, "Smith", 1200D));

listOfEmployees.add(new Employee(111, "Alen", 1500D));

listOfEmployees.add(new Employee(222, "John", 1800D));

listOfEmployees.add(new Employee(444, "Martin", 900D));

listOfEmployees.forEach(System.out::println);

}

}

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

***Program that describes ArrayList is better than Vector in performance wise***

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

As we know ArrayList methods are not synchronized so multiple threads can access the method of ArrayList, on the other hand most of the methods are synchronized in Vector class.

java.lang.System class has provided a predefined static method called **currentTimeMillis()** through which we can get the current system time in millisecond.

public static native long **currentTimeMillis();**

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

package com.ravi.vector;

import java.util.ArrayList;

import java.util.Vector;

public class VectorDemo4

{

public static void main(String[] args)

{

long startTime = System.currentTimeMillis();

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

for(int i=0; i<1000000; i++)

{

al.add(i);

}

long endTime = System.currentTimeMillis();

System.out.println("Total Time taken by ArrayList class :"+(endTime - startTime)+" ms");

startTime = System.currentTimeMillis();

Vector<Integer> v1 = new Vector<Integer>();

for(int i=0; i<1000000; i++)

{

v1.add(i);

}

endTime = System.currentTimeMillis();

System.out.println("Total Time taken by Vector class :"+(endTime - startTime)+" ms");

}

}

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

**\*\*\*What is Fail Fast Iteartor ?**

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

While retrieving the object from the collection by using Itearor interface or for each loop, if at any point of time the original structure is going to modify after the creation of Itearator then we will get java.util.ConcurrentModificationExacption.

package com.ravi.vector;

import java.util.Iterator;

import java.util.Spliterator;

import java.util.Vector;

class Concurrent extends Thread //java.util.concurrent

{

private Vector<String> cities = null;

public Concurrent(Vector<String> cities)

{

super();

this.cities = cities;

}

@Override

public void run()

{

try

{

Thread.sleep(2000);

}

catch(InterruptedException e)

{

}

cities.add("Ameerpet");

}

}

public class FailFastIteartor

{

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

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Hyderabad");

listOfCity.add("Pune");

listOfCity.add("Indore");

listOfCity.add("Bhubneswar");

listOfCity.add("Kolkata");

Concurrent c1 = new Concurrent(listOfCity);

c1.start();

Iterator<String> itr = listOfCity.iterator();

while(itr.hasNext())

{

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

Thread.sleep(500);

}

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

Spliterator<String> spliterator = listOfCity.spliterator();

spliterator.forEachRemaining(System.out::println);

}

}

Here we will get java.util.ConcurrentModificationException.

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

package com.ravi.vector;

import java.util.Arrays;

import java.util.Collections;

import java.util.OptionalInt;

import java.util.Vector;

public class VectorDemo5

{

public static void main(String[] args)

{

Vector<String> listOfCity = new Vector<>();

listOfCity.add("Surat");

listOfCity.add("Pune");

listOfCity.add("Ahmadabad");

listOfCity.add("Vanaras");

Collections.sort(listOfCity);

listOfCity.forEach(System.out::println);

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

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

listOfNumbers.add(500);

listOfNumbers.add(900);

listOfNumbers.add(400);

listOfNumbers.add(300);

listOfNumbers.add(800);

listOfNumbers.add(200);

listOfNumbers.add(100);

System.out.println("Original Data...");

System.out.println(listOfNumbers);

System.out.println("Ascending Order...");

Collections.sort(listOfNumbers);

System.out.println(listOfNumbers);

System.out.println("Descending Order...");

Collections.sort(listOfNumbers, Collections.reverseOrder());

System.out.println(listOfNumbers);

//Converting Our Vector(Collection Object) into Array

Vector<String> listOfFruits = new Vector<>();

listOfFruits.add("Orange");

listOfFruits.add("Apple");

listOfFruits.add("Mango");

Object[] fruits = listOfFruits.toArray();

System.out.println(Arrays.toString(fruits));

}

}

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

package com.ravi.vector;

import java.util.Scanner;

import java.util.Vector;

public class VectorDemo6

{

public static void main(String[] args)

{

Vector<String> toDoList = new Vector<>();

Scanner scanner = new Scanner(System.in);

int choice;

do

{

System.out.println("To Do List Menu:");

System.out.println("1. Add Task");

System.out.println("2. View Tasks");

System.out.println("3. Mark Task as Completed");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

scanner.nextLine();

switch (choice)

{

case 1:

// Add Task

System.out.print("Enter task description: ");

String task = scanner.nextLine();

toDoList.add(task);

System.out.println("Task added successfully!\n");

break;

case 2:

// View Tasks

System.out.println("To Do List:");

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

{

System.out.println((i + 1) + ". " + toDoList.get(i));

}

System.out.println();

break;

case 3:

// Mark Task as Completed

System.out.print("Enter task number to mark as completed: ");

int taskNumber = scanner.nextInt(); //1

if (taskNumber >= 1 && taskNumber <= toDoList.size())

{

String completedTask = toDoList.remove(taskNumber - 1);

System.out.println("Task marked as completed: " + completedTask + "\n");

}

else {

System.out.println("Invalid task number!\n");

}

break;

case 4:

System.out.println("Exiting ToDo List application. Goodbye!");

break;

default:

System.out.println("Invalid choice. Please enter a valid option.\n");

}

}

while (choice != 4);

scanner.close();

}

}

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

**Enumeration interface has provided a method from java 9V called**

**asIterator(), the return type of this method is Iterator**

**interface.**

**public Iterator asIterator(); [Enumeration interface method]**

It is mainly used for Backward compatibility.

package com.ravi.vector;

import java.util.Enumeration;

import java.util.Iterator;

import java.util.Vector;

record Product(int productId, String productName)

{

}

public class VectorDemo7

{

public static void main(String[] args)

{

Vector<Product> listOfProduct = new Vector<>();

listOfProduct.add(new Product(111, "Laptop"));

listOfProduct.add(new Product(222, "Mobile"));

listOfProduct.add(new Product(333, "Camera"));

listOfProduct.add(new Product(444, "Bag"));

listOfProduct.add(new Product(555, "Watch"));

Enumeration<Product> ele = listOfProduct.elements();

Iterator<Product> asIterator = ele.asIterator();

asIterator.forEachRemaining(System.out::println);

}

}

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

Stack<E> :

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

public class Stack<E> extends Vector<E>

It is a predefined class available in java.util package. It is the sub class of Vector class introduced from JDK 1.0 so, It is also a legacy class.

It is a linear data structure that is used to store the Objects in LIFO (Last In first out) order.

Inserting an element into a Stack is known as push operation where as extracting an element from the top of the stack is known as pop operation.

It throws an exception called java.util.EmptyStackException, if Stack is empty and we want to fetch the element.

It has only one constructor as shown below

Stack s = new Stack();

Will create empty Stack Object.

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

Methods :

----------

public E push(Object o) :- To insert an element in the bottom of the Stack.

public E pop() :- To remove and return the element from the top of the Stack.

public E peek() :- Will fetch the element from top of the Stack without removing.

public boolean empty() :- Verifies whether the stack is empty or not (return type is boolean)

public int search(Object o) :- It will search a particular element in the Stack and it returns OffSet position (int value). If the element is not present in the Stack it will return -1

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

//Program to insert and fetch the elements from stack

package com.ravi.stack;

import java.util.\*;

public class Stack1

{

public static void main(String args[])

{

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

try

{

s.push(12);

s.push(15);

s.push(22);

s.push(33);

s.push(49);

System.out.println("After insertion elements are :"+s);

System.out.println("Fetching the elements using pop method");

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

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

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

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

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

System.out.println("After deletion elements are :"+s);//[]

System.out.println("Is the Stack empty ? :"+s.empty());

}

catch(EmptyStackException e)

{

e.printStackTrace();

}

}

}

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

//add(Object obj) is the method of Collection

package com.ravi.stack;

import java.util.\*;

public class Stack2

{

public static void main(String args[])

{

Stack<Integer> st1 = new Stack<>();

st1.add(10);

st1.add(20);

st1.forEach(x -> System.out.println(x));

Stack<String> st2 = new Stack<>();

st2.add("Java");

st2.add("is");

st2.add("programming");

st2.add("language");

st2.forEach(x -> System.out.println(x));

Stack<Character> st3 = new Stack<>();

st3.add('A');

st3.add('B');

st3.forEach(x -> System.out.println(x));

Stack<Double> st4 = new Stack<>();

st4.add(10.5);

st4.add(20.5);

st4.forEach(x -> System.out.println(x));

}

}

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

package com.ravi.stack;

import java.util.Stack;

public class Stack3

{

public static void main(String[] args)

{

Stack<String> stk= new Stack<>();

stk.push("Apple");

stk.push("Grapes");

stk.push("Mango");

stk.push("Orange");

System.out.println("Stack: " + stk);

String fruit = stk.peek();

System.out.println("Element at top: " + fruit);

System.out.println("Stack elements are : " + stk);

}

}

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

//Searching an element in the Stack

package com.ravi.stack;

import java.util.Stack; //

public class Stack4

{

public static void main(String[] args) //1 -1 false 2

{

Stack<String> stk= new Stack<>();

stk.push("Apple");

stk.push("Grapes");

stk.push("Mango");

System.out.println("Offset Position is : " + stk.search("Mango")); //1

System.out.println("Offser Position is : " + stk.search("Banana")); //-1

System.out.println("Is stack empty ? "+stk.empty()); //false

System.out.println("Index Position is : " + stk.indexOf("Mango")); //2

}

}

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

14-02-2025

-----------

ArrayList<E>

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

public class ArrayList<E> extends AbstractList<E> implements List<E>, Serializable, Clonable, RandomAccess

It is a predefined class available in java.util package under List interface from java 1.2v.

It accepts duplicate,null, homogeneous and hetrogeneous elements.

It is dynamically growable array.

It stores the elements on index basis so it is simillar to dynamic array.

Initial capacity of ArrayList is 10. The new capacity of Arraylist can be calculated by using the formula

new capacity = (current capacity \* 3)/2 + 1 [Almost 50% increment]

\*All the methods declared inside an ArrayList is not synchronized so multiple thread can access the method of ArrayList so performance wise it is good.

\*It is highly suitable for fetching or retriving operation when duplicates are allowed and Thread-safety is not required.

Here Iterator is Fail Fast Iteartor.

It implements List,Serializable, Clonable, RandomAccess interfcaes

Constructor of ArrayList :

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

In ArrayList we have 3 types of Constructor:

Constructor of ArrayList :

1) ArrayList al1 = new ArrayList();

Will create ArrayList object with default capacity 10.

2) ArrayList al2 = new ArrayList(int initialCapacity);

Will create an ArrayList object with user specified Capacity

3) ArrayList al3 = new ArrayList(Collection c)

We can copy any Collection interface implemented class data to the current object reference (Coping one Collection data to another)

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

package com.ravi.arraylist;

import java.util.ArrayList;

public class ArrayListDemo

{

public static void main(String[] args)

{

ArrayList<Integer> numbers = new ArrayList<>(100);

numbers.add(100);

numbers.add(200);

numbers.add(300);

numbers.add(400);

int sum = 0;

for (int number : numbers)

{

sum += number;

}

System.out.println("Sum of numbers: " + sum);

}

}

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

package com.ravi.arraylist;

import java.util.ArrayList;

import java.util.Collections;

record Customer(Integer custId, String custName, Double custSal)

{

}

public class ArrayListDemo1

{

public static void main(String[] args)

{

ArrayList<Customer> listOfCustomer = new ArrayList<>();

listOfCustomer.add(new Customer(111, "Scott", 800D));

listOfCustomer.add(new Customer(222, "Smith", 1200D));

listOfCustomer.add(new Customer(333, "Alen", 1800D));

listOfCustomer.add(new Customer(444, "Martin", 1500D));

listOfCustomer.add(new Customer(555, "John", 1300D));

listOfCustomer.forEach(System.out::println);

}

}

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

package com.ravi.arraylist;

//Program to merge and retain of two collection addAll() retainlAll()

import java.util.\*;

public class ArrayListDemo2

{

public static void main(String args[])

{

ArrayList<String> al1=new ArrayList<>();

al1.add("Ravi");

al1.add("Rahul");

al1.add("Rohit");

ArrayList<String> al2=new ArrayList<>();

al2.add("Pallavi");

al2.add("Sweta");

al2.add("Puja");

al1.addAll(al2);

al1.forEach(str -> System.out.println(str.toUpperCase()) );

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

ArrayList<String> al3=new ArrayList<>();

al3.add("Ravi");

al3.add("Rahul");

al3.add("Rohit");

ArrayList<String> al4=new ArrayList<>();

al4.add("Pallavi");

al4.add("Rahul");

al4.add("Raj");

al3.retainAll(al4);

al3.forEach(x -> System.out.println(x));

}

}

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

How to create Immutable List and Fixed length array :

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

1) Fixed length Array :

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

java.util.Arrays class has provided a predefined static method

asList(T ...x), It will create a fixed length array and the return type of this method is List interface.

In this fixed length array we can't perform add or remove opeartion otherwise we will get java.lang.UnsupportedOperationException but we can replace the element.

package com.ravi.arraylist;

import java.util.Arrays;

import java.util.List;

public class FixedLengthArray {

public static void main(String[] args)

{

//Fixed length Array

List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8);

//list.add(9); java.lang.UnsupportedOperationException

//list.remove(0); java.lang.UnsupportedOperationException

list.set(0, 100);

System.out.println(list);

}

}

2) Immutable List :

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

List interface has provided a predefined static method called

of(T ...x) available from java 9V.

It will create an immutable list, return type of this method is

List<E>. Once it is created after that we can't pefrom any kind of operation like add(), remove() or replace [set(int index, Object obj)] otherwise we will get java.lang.UnsupportedOperationException

package com.ravi.arraylist;

import java.util.List;

public class ImmutableList {

public static void main(String[] args)

{

List<Integer> immutable = List.of(1,2,3,4,5,6,7,8);

//immutable.add(9); java.lang.UnsupportedOperationException

//immutable.remove(0); java.lang.UnsupportedOperationException

//immutable.set(0, 90); java.lang.UnsupportedOperationException

System.out.println(immutable);

}

}

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

//Program to fetch the elements in forward and backward

//direction using ListIterator interface

package com.ravi.arraylist;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

import java.util.ListIterator;

public class ArrayListDemo3

{

public static void main(String args[])

{

List<String> listOfName = Arrays.asList("Rohit","Akshar","Pallavi","Sweta"); //Length is fixed

Collections.sort(listOfName);

//Fetching the data in both the direction

ListIterator<String> lst = listOfName.listIterator();

System.out.println("In Forward Direction..");

while(lst.hasNext())

{

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

}

System.out.println("In Backward Direction..");

while(lst.hasPrevious())

{

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

}

}

}

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

Serialization and Deserialization on ArrayList object :

//Serialization and De-serialization on ArrayList Object

package com.ravi.arraylist;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.util.ArrayList;

public class ArrayListDemo4

{

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

{

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

listOfIceCream.add("Vanila");

listOfIceCream.add("Strwbarry");

listOfIceCream.add("Butter Scotch");

//Serialization

var fos = new FileOutputStream("D:\\new\\IceCream.txt");

var oos = new ObjectOutputStream(fos);

try(oos; fos)

{

oos.writeObject(listOfIceCream);

System.out.println("Object Stored Successfully!!!");

}

catch(Exception e)

{

e.printStackTrace();

}

//De-Serialization

var fin = new FileInputStream("D:\\new\\IceCream.txt");

var ois = new ObjectInputStream(fin);

try(ois; fin)

{

ArrayList<String> list = (ArrayList<String>) ois.readObject();

System.out.println("List Data :"+list);

}

catch(Exception e)

{

e.printStackTrace();

}

}

}

Note : Here String and ArrayList both the classes implements java.io.Serializable so serailization is possible.

package com.ravi.arraylist;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.util.ArrayList;

record Employee(Integer employeeId, String employeeName) implements Serializable

{

}

public class ArrayListSerialization

{

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

{

ArrayList<Employee> listOfEmployees = new ArrayList<>();

listOfEmployees.add(new Employee(111, "A"));

listOfEmployees.add(new Employee(222, "B"));

listOfEmployees.add(new Employee(333, "C"));

listOfEmployees.add(new Employee(444, "D"));

listOfEmployees.add(new Employee(555, "E"));

String filePath = "D:\\new\\Employee.txt";

//Serialization

var fos = new FileOutputStream(filePath);

var oos = new ObjectOutputStream(fos);

try(fos; oos)

{

oos.writeObject(listOfEmployees);

System.out.println("Object data stored successfully");

}

catch(Exception e)

{

e.printStackTrace();

}

//De-Serialization

var fin = new FileInputStream(filePath);

var ois = new ObjectInputStream(fin);

try(fin; ois)

{

@SuppressWarnings("unchecked")

ArrayList<Employee> empList = (ArrayList<Employee>) ois.readObject();

empList.forEach(System.out::println);

}

catch(Exception e)

{

e.printStackTrace();

}

}

}

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

package com.ravi.arraylist;

import java.util.ArrayList;

import java.util.Collections;

public class ArrayListDemo5

{

public static void main(String[] args)

{

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

cities.add("Hyderabad");

cities.add("Delhi");

cities.add("Banglore");

cities.add("Chennai");

System.out.println("Before sorting: " + cities);

Collections.sort(cities);

System.out.println("After sorting (Ascending): " + cities);

Collections.sort(cities,Collections.reverseOrder());

System.out.println("After sorting (Descending): " + cities);

}

}

Collections class has provided static method reverseOrder(), return type is Comparator.

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

package com.ravi.arraylist;

//Program on ArrayList that contains null values as well as we can pass

//the element based on the index position

import java.util.ArrayList;

import java.util.LinkedList;

public class ArrayListDemo6

{

public static void main(String[] args)

{

ArrayList<Object> al = new ArrayList<>(); //Generic type

al.add(12);

al.add("Ravi");

al.add(12);

al.add(3,"Hyderabad");

al.add(1,"Naresh");

al.add(null);

al.add(11);

System.out.println(al); //12 Naresh Ravi 12 Hyderabad null 11

}

}

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

package com.ravi.arraylist;

import java.util.ArrayList;

import java.util.List;

record Professor(String name, String specialization)

{

}

class Department

{

private String departmentName;

private List<Professor> professors;

public Department(String departmentName)

{

this.departmentName = departmentName;

this.professors = new ArrayList<Professor>();//Composition

}

public void addProfessor(Professor prof)

{

this.professors.add(prof);

}

public String getDepartmentName()

{

return departmentName;

}

public List<Professor> getProfessors()

{

return professors;

}

}

public class ArrayListDemo7

{

public static void main(String[] args)

{

Department cs = new Department("Computer Science");

cs.addProfessor(new Professor("James", "Java"));

cs.addProfessor(new Professor("Scott", "Adv Java"));

cs.addProfessor(new Professor("Kishore Sir", "C language"));

System.out.println("Professors in :"+cs.getDepartmentName());

List<Professor> professors = cs.getProfessors();

professors.forEach(System.out::println);

}

}

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

package com.ravi.arraylist;

import java.lang.reflect.Array;

import java.util.ArrayList;

public class ArrayListDemo8

{

public static void main(String[] args)

{

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

original.add("BCA");

original.add("MCA");

original.add("BBA");

System.out.println("By using clone Method");

@SuppressWarnings("unchecked")

ArrayList<String> clonedAL = (ArrayList<String>) original.clone();

System.out.println(clonedAL);

System.out.println("By using Copy Constructor");

ArrayList<String> copyConstr = new ArrayList<String>(original);

System.out.println(copyConstr);

}

}

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

public List subList(int fromIndex, int toIndex) :

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

It is used to fetch/retrieve the part of the List based on the given index. The return type of this method is List, Here fromIndex is inclusive and toIndex is exclusive.

public boolean contains(Object element) :

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

It is used to find the given element object in the corresponsing List, if available it will return true otherwise false.

public boolean removeIf(Predicate<T> filter)

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

It is used to remove the elements based on boolean condition passed as a Predicate.

package com.ravi.arraylist;

import java.util.ArrayList;

import java.util.List;

public class ArrayListDemo9 {

public static void main(String[] args)

{

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

list.add(1);

list.add(2);

list.add(3);

list.add(4);

list.add(5);

list.add(6);

list.add(7);

list.add(8);

list.add(9);

list.add(10);

//public List subList(int fromIndex, int toIndex)

List<Integer> subList = list.subList(2, 5);

System.out.println(subList);

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

//public boolean contains(Object obj)

boolean contains = list.contains(99);

System.out.println(contains);

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

//public int indexOf(Object obj)

System.out.println(list.indexOf(1));

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

//public void removeIf(Predicate<T> p)

list.removeIf(num -> num%2==0);

list.forEach(System.out::println);

}

}

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

public void trimToSize() :

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

Used to reduce the capacity.

public void ensureCapacity(int minCapacaity):

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

Increase the capacity of the ArrayList to avoid frequent resizing.

The minCapacaity parameter will specify that ArrayList will definetly hold the number of elements specified in the parameter of ensureCapacity() method.

After using ensureCapacity() method, still it is dynamically growable.

package com.ravi.arraylist;

import java.util.ArrayList;

import java.util.RandomAccess;

public class ArrayListDemo10 {

public static void main(String[] args)

{

ArrayList<String> list = new ArrayList<>(100);

list.add("Java");

list.add("World");

//public void trimToSize()

list.trimToSize();

System.out.println("Trimmed List Size: " + list.size());

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

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

// public void ensureCapacity(int minCapacity)

//Increase the capacity of the ArrayList to avoid frequent resizing.

listOfNumber.add(999);

listOfNumber.ensureCapacity(100);

for (int i = 0; i < 50; i++)

{

listOfNumber.add(i);

}

System.out.println("List size: " + listOfNumber.size());

}

}

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

Time Complexity of ArrayList :

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

The time complexity of ArrayList to insert OR delete an element from the middle would be O(n) [Big O of n] because 'n' number of elements will be re-located so, it is not a good choice to perform insertion and deletion operation in the middle OR begning of the List.

On the other hand time complexity of ArrayList to retrieve an element from the List would be O(1) because by using get(int index) method we can retrieve the element randomly from the list. ArrayList class implements RandomAccess marker interface which provides the facility to fetch the elements Randomly.

[17-FEB-25]

In order to insert and delete the element in middle of the list frequently, we introduced LinkedList class.

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

It is a predefined class available in java.util package under List interface from JDK 1.2v.

It is ordered by index position like ArrayList except the elements (nodes) are doubly linked to one another. This linkage provide us new method for adding and removing the elements from the middle of LinkedList.

It stores the elements in non-contiguous memory location.

\*The important thing is, LikedList may iterate more slowely than ArrayList but LinkedList is a good choice when we want to insert or delete the elements frequently in the list.

From jdk 1.6 onwards LinkedList class has been enhanced to support basic queue operation by implementing Deque<E> interface.

LinkedList methods are not synchronized.

It inserts the elements by using Doubly linked List so insertion and deleteion is very easy.

ArrayList is using Dynamic array data structure but LinkedList class is using LinkedList (Doubly LinkedList) data structure.

At the time of searching an element, It will start searching from Head node OR tail node OR closer one based on the index.

\*\*Here Iterators are Fail Fast Iterator.

Constructor:

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

It has 2 constructors

1) LinkedList list1 = new LinkedList();

It will create a LinkedList object with 0 capacity.

2) LinkedList list2 = new LinkedList(Collection c);

Interconversion between the collection

Methods of LinkedList class:

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

1) void addFirst(Object o)

2) void addLast(Object o)

3) Object getFirst()

4) Object getLast()

5) Object removeFirst()

6) Object removeLast()

The time complexcity for insertion and deletion is O(1) The time complexcity for seraching O(n) because it serach the elemnts using node reference.

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

import java.util.LinkedList;

public class MyLinkedList

{

private static class Node

{

int item;

Node next;

public Node(int item)

{

this.item = item;

this.next = null;

}

}

public static void main(String[] args)

{

Node node1 = new Node(100);

Node node2 = new Node(200);

Node node3 = new Node(300);

node1.next = node2;

node2.next = node3;

System.out.println(node1.item);

System.out.println(node1.next.item);

System.out.println(node1.next.next.item);

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

}

}

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

package com.ravi.linked\_list;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

public class LinkedListDemo

{

public static void main(String args[])

{

LinkedList<Object> list=new LinkedList<>();

list.add("Ravi");

list.add("Vijay");

list.add("Ravi");

list.add(null);

list.add(42);

System.out.println("1st Position Element is :"+list.get(1));

//Iterator interface

Iterator<Object> itr = list.iterator();

itr.forEachRemaining(System.out::println); //JDK 1.8

}

}

Note : From this program, It is clear that LinkedList works on the basis of index.

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

package com.ravi.linked\_list;

import java.util.\*;

public class LinkedListDemo1

{

public static void main(String args[])

{

LinkedList<String> list= new LinkedList<>(); //generic

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

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

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

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

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

list.add("Item 7");//7

list.add("Item 9"); //10

list.add(0,"Item 0");//0

list.add(1,"Item 1"); //1

list.add(8,"Item 8");//8

list.add(9,"Item 10");//9

System.out.println(list);

list.remove("Item 5");

System.out.println(list);

list.removeLast();

System.out.println(list);

list.removeFirst();

System.out.println(list);

list.set(0,"Ajay"); //set() will replace the existing value

list.set(1,"Vijay");

list.set(2,"Anand");

list.set(3,"Aman");

list.set(4,"Suresh");

list.set(5,"Ganesh");

list.set(6,"Ramesh");

list.forEach(x -> System.out.println(x));

}

}

Note : From the above program it is clear that we can isert or delete an element in the LinkedList with good time complexity.

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

package com.ravi.linked\_list;

//Methods of LinkedList class

import java.util.LinkedList;

public class LinkedListDemo2

{

public static void main(String[] argv)

{

LinkedList<String> list = new LinkedList<>();

list.addFirst("Ravi"); // Rahul

list.add("Rahul");

list.addLast("Anand");

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

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

list.removeFirst();

list.removeLast();

System.out.println(list); //[Rahul]

}

}

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

19-02-2025

-----------

package com.ravi.linked\_list;

//ListIterator methods (add(), set(), remove())

import java.util.\*;

public class LinkedListDemo3

{

public static void main(String[] args)

{

LinkedList<String> city = new LinkedList<> ();

city.add("Kolkata");

city.add("Bangalore");

city.add("Hyderabad");

city.add("Pune");

System.out.println(city);

ListIterator<String> lt = city.listIterator();

while(lt.hasNext())

{

String cityName = lt.next();

if(cityName.equals("Kolkata"))

{

lt.remove();

}

else if(cityName.equals("Hyderabad"))

{

lt.add("Ameerpet");

}

else if(cityName.equals("Pune"))

{

lt.set("Mumbai");

}

}

city.forEach(System.out::println);

}

}

Here there is no ConcurrentModificationException because ListIterator is modifying the structure by it's own method hence there is no problem because it is internal structure modification.

--------------------------------------------------------------package com.ravi.linked\_list;

//Insertion, deletion, displaying and exit

import java.util.LinkedList;

import java.util.List;

import java.util.Scanner;

public class LinkedListDemo4

{

public static void main(String[] args)

{

List<Integer> linkedList = new LinkedList<>();

Scanner scanner = new Scanner(System.in);

while (true)

{

System.out.println("Linked List: " + linkedList); //[]

System.out.println("1. Insert Element");

System.out.println("2. Delete Element");

System.out.println("3. Display Element");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

switch (choice)

{

case 1:

System.out.print("Enter the element to insert: ");

int elementToAdd = scanner.nextInt();

linkedList.add(elementToAdd);

break;

case 2:

if (linkedList.isEmpty())

{

System.out.println("Linked list is empty. Nothing to delete.");

}

else

{

System.out.print("Enter the element to delete: ");

int elemenetToDelete = scanner.nextInt();

boolean remove = linkedList.remove(Integer.valueOf(elemenetToDelete));

if(remove)

{

System.out.println("Element "+elemenetToDelete+ " is deleted Successfully" );

}

else

{

System.out.println("Element "+elemenetToDelete+" not available is the LinkedList");

}

}

break;

case 3:

System.out.println("Elements in the linked list.");

linkedList.forEach(System.out::println);

break;

case 4:

System.out.println("Exiting the program.");

scanner.close();

System.exit(0);

default:

System.out.println("Invalid choice. Please try again.");

}

}

}

}

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

//LinkedList list = new LinkedList(Collection coll)

package com.ravi.linked\_list;

import java.util.Arrays;

import java.util.LinkedList;

import java.util.List;

public class LinkedListDemo5 {

public static void main(String[] args)

{

List<String> listOfName = Arrays.asList("Ravi","Rahul","Ankit", "Rahul");

LinkedList<String> list = new LinkedList<>(listOfName);

list.forEach(System.out::println);

}

}

package com.ravi.linked\_list;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

record Product(Integer productId, String productName)

{

}

public class LinkedListDemo6 {

public static void main(String[] args)

{

List<Product> listOfProduct = new LinkedList<Product>();

listOfProduct.add(new Product(1, "ApplePhone"));

listOfProduct.add(new Product(2, "MiPhone"));

listOfProduct.add(new Product(3, "VivoPhone"));

System.out.println("Is list empty :"+listOfProduct.isEmpty());

Iterator<Product> iterator = listOfProduct.iterator();

iterator.forEachRemaining(prod -> System.out.println(prod.productName().toUpperCase()));

Product product = listOfProduct.get(1);

System.out.println(product.productName());

}

}

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

Set<E> interface :

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

Set interface :

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

Set interface is the sub interface of Collection available from JDK 1.2V

Set interface never accept duplicate elements, Here internally equals(Object obj) method is working from the respective class.

Set interface does not maintain any order (because internally It does not use Array concept, Actually It uses hashing algorithm)

On Set interface we can't use ListIterator interface.

Set interface supports all the methods of Collection interface, few more methods were added from java 9v.

Set interface Hierarchy :

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

Hierarchy is available in the paint diagram [19-FEB]

What is hashing algorithm ?

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

What is hashing algorithm ?

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

Hashing algorithm is a technique through which we can search, insert and delete an element in more efficient way in comparison to our classical indexing approach.

Hashing algorithm, internally uses Hashtable data structute, Hashtable data structure internally uses Bucket data structure.

Here elements are inserted by using hashing algorithm so the time complaxity to insert, delete and search an element would be O(1).

It is more efficient than our classical array approach which works on the basis of index.

20-02-2025

-----------

HashSet<E> [UNORDERED, UNSORTED, NO DUPLICATES]

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

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Clonabale, Serializable

It is a predefined class available in java.util package under Set interface and introduced from JDK 1.2V.

It is an unsorted and unordered set.

It accepts hetrogeneous and homogeneous both kind of data.

It uses Hashtable data structure, default capacity is 16 that means 16 buckets will be created internally.

\*It uses the hashcode of the object being inserted into the Collection. Using this hashcode it finds the bucket location.

It doesn't contain any duplicate elements as well as It does not maintain any order while iterating the elements from the collection.

It can accept one null value.

HashSet methods are not synchronized.

HashSet is used for fast searching operation.

It has constant performance in all the operations like insert, delete and search.

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

It contains 4 types of constructors :

1) HashSet hs1 = new HashSet();

It will create the HashSet Object with default capacity is 16. The default load fator or Fill Ratio is 0.75 (75% of HashSet is filled up then new HashSet Object will be created having double capacity)

2) HashSet hs2 = new HashSet(int initialCapacity);

will create the HashSet object with user specified capacity.

3) HashSet hs3 = new HashSet(int initialCapacity, float loadFactor);

we can specify our own initialCapacity and loadFactor(by default load factor is 0.75)

4) HashSet hs4 = new HashSet(Collection c);

Interconversion of Collection.

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

//Unsorted, Unordered and no duplicates

import java.util.\*;

public class HashSetDemo

{

public static void main(String args[])

{

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

hs.add(67);

hs.add(89);

hs.add(33);

hs.add(45);

hs.add(12);

hs.add(35);

hs.add(null);

hs.forEach(num-> System.out.println(num));

}

}

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

import java.util.\*;

public class HashSetDemo1

{

public static void main(String[] argv)

{

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

hs.add("Ravi");

hs.add("Vijay");

hs.add("Ravi");

hs.add("Ajay");

hs.add("Palavi");

hs.add("Sweta");

hs.add(null);

hs.add(null);

hs.forEach(str -> System.out.println(str));

}

}

HashSet does not maintain any order while iterating the elements from the Collection.

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

package com.ravi.hash\_set\_demo;

import java.util.Arrays;

import java.util.HashSet;

public class HashSetDemo2 {

public static void main(String[] args)

{

Boolean values[] = new Boolean[5];

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

values[0] = hs.add(12);

values[1] = hs.add(12);

values[2] = hs.add("NIT");

values[3] = hs.add(new String("NIT"));

values[4] = hs.add(null);

System.out.println(Arrays.toString(values));

System.out.println("------------------------");

if(hs.contains("NIT"))

{

System.out.println("NIT Object is available");

}

else

{

System.out.println("NIT Object is not available");

}

hs.forEach(System.out::println);

}

}

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

package com.ravi.hash\_set\_demo;

import java.util.HashSet;

public class HashSetDemo {

public static void main(String[] args)

{

HashSet<String> hs1 = new HashSet<String>();

hs1.add(new String("India"));

hs1.add(new String("India"));

System.out.println(hs1.size()); //1

HashSet<StringBuffer> hs2 = new HashSet<StringBuffer>();

hs2.add(new StringBuffer("Hyd"));

hs2.add(new StringBuffer("Hyd"));

System.out.println(hs2.size()); //2

//reason for size mismatch

String str = new String("Ameerpet");

System.out.println(str.hashCode() +" : "+str.equals(null));

StringBuffer sb = new StringBuffer("S R nagar");

System.out.println(sb.hashCode() +" : "+sb.equals(null));

}

}

Note : We are getting different size while working with String and StringBuffer class because String class has overridden hashCode() and equals() method from Object class but StringBuffer class has not overridden hashCode() and equals()

method.

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

//add, delete, display and exit

import java.util.HashSet;

import java.util.Scanner;

public class HashSetDemo3

{

public static void main(String[] args)

{

HashSet<String> hashSet = new HashSet<>();

Scanner scanner = new Scanner(System.in);

while (true)

{

System.out.println("Options:");

System.out.println("1. Add element");

System.out.println("2. Delete element");

System.out.println("3. Display HashSet");

System.out.println("4. Exit");

System.out.print("Enter your choice (1/2/3/4): ");

int choice = scanner.nextInt();

switch (choice)

{

case 1:

System.out.print("Enter the element to add: ");

String elementToAdd = scanner.next();

if (hashSet.add(elementToAdd))

{

System.out.println("Element added successfully.");

}

else

{

System.out.println("Element already exists in the HashSet.");

}

break;

case 2:

System.out.print("Enter the element to delete: ");

String elementToDelete = scanner.next();

if (hashSet.remove(elementToDelete))

{

System.out.println("Element deleted successfully.");

}

else

{

System.out.println("Element not found in the HashSet.");

}

break;

case 3:

System.out.println("Elements in the HashSet:");

hashSet.forEach(System.out::println);

break;

case 4:

System.out.println("Exiting the program.");

scanner.close();

System.exit(0);

default:

System.out.println("Invalid choice. Please try again.");

}

System.out.println();

}

}

}

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

21-02-2025

-----------

Object class hashCode() and equals() Method :

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

public int hashCode()

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

It is a predefined non static method of Object class which is used to generate the hashcode of the Object.

hashCode of an object is useful to insert the Object in the Hashtable data structure.

hascode of an object is not meant for comparing two objects

If two Objects are having same hashcode then Object may be same or may not be same.

If two objects are same by using equals(Object obj) method then their hashcode MUST BE SAME.

//Program :

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

package com.ravi.hash\_code\_demo;

class Student

{

}

public class HashCodeDemo1

{

public static void main(String[] args)

{

Student s1 = new Student();

Student s2 = new Student();

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

}

Both the Student Objects are different as per equals(Object obj) method so the hashcode will be different which is

coming from Object class.

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

public boolean equals(Object obj) :

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

It is a predefined non static method of Object class which is

used to compare two objects based on the memory reference OR memory address as shown in the program below :

package com.ravi.equals;

class Customer

{

private int customerId;

private String customerName;

public Customer(int customerId, String customerName)

{

super();

this.customerId = customerId;

this.customerName = customerName;

}

}

public class EqualsDemo1 {

public static void main(String[] args)

{

Customer c1 = new Customer(111, "Scott");

Customer c2 = new Customer(111, "Scott");

System.out.println(c1.equals(c2));

}

}

Note : It will return false because equals(Object obj) method

of Object class is used to compare two objects based on the

memory reference/Memory address.

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

If we want to compare two Objects based on the content but not based on the memory address then we should override equals(Object obj) method from Object class as shown below

package com.ravi.equals;

class Customer

{

private int customerId;

private String customerName;

public Customer(int customerId, String customerName)

{

super();

this.customerId = customerId;

this.customerName = customerName;

}

@Override

public int hashCode()

{

return this.customerId;

}

@Override

public boolean equals(Object obj)

{

//1st Object data

int id1 = this.customerId;

String name1 = this.customerName;

//2nd Object data

Customer c2 = (Customer) obj;

int id2 = c2.customerId;

String name2 = c2.customerName;

if(id1 == id2 && name1.equals(name2))

{

return true;

}

else

{

return false;

}

}

}

public class EqualsDemo1 {

public static void main(String[] args)

{

Customer c1 = new Customer(111, "Scott");

Customer c2 = new Customer(111, "Scott");

System.out.println(c1.equals(c2));

System.out.println(c1.hashCode()+" : "+c2.hashCode());

}

}

WE SHOULD ALWAYS OVERRIDE HASHCODE AND EQUALS BOTH THE METHODS TOGETHER

All the Wrapper classes and String class have overriden both the methods.

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

package com.ravi.equals;

class Product

{

private Integer pid;

private String pname;

public Product(Integer pid, String pname) {

super();

this.pid = pid;

this.pname = pname;

}

@Override

public int hashCode()

{

return this.pid;

}

@Override

public boolean equals(Object obj)

{

if(obj instanceof Product)

{

Product p2 = (Product) obj;

if(this.pid == p2.pid && this.pname.equals(p2.pname))

{

return true;

}

else

{

return false;

}

}

else

{

System.err.println("Comparison is not possible!!");

return false;

}

}

}

public class EqualsDemo2 {

public static void main(String[] args)

{

Product p1 = new Product(111, "Scott");

Student s1 = new Student(111, "Scott");

System.out.println(p1.equals(s1));

System.out.println(p1.equals(null));

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

Product p2 = new Product(222, "Scott");

System.out.println(p1.equals(p2));

}

}

class Student

{

private int sid;

private String sname;

public Student(int sid, String sname) {

super();

this.sid = sid;

this.sname = sname;

}

}

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

Map interface :

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

As we know Collection interface is used to hold single Or individual object but Map interface will hold group of objects in the form key and value pair. {key = value}

Map<K,V> interface is not the part the Collection, It is a separate interface.

Before Map interface We have Dictionary<K,V>(abstract class) class and it is extended by Hashtable<K,V> class in JDK 1.0V

Map interface works with key and value pair introduced from 1.2V.

Here key and value both are objects.

Here key must be unique and value may be duplicate.

Each key and value pair is creating one Entry.(Entry is nothing but the combination of key and value pair)

public interface Map<K,V>

{

public interface Entry<K,V>

{

//key and value

}

}

How to represent this entry interface (Map.Entry in .java) [Map$Entry in .class]

In Map interface whenever we have a duplicate key then the old key value will be replaced by new key(duplicate key) value.

Map interface has defined forEach(BiConsumer cons) method to work with group of Objects.It does not extends Iterable interface.

Iterator and ListIterator we can't work directly using Map.

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

22-02-2025

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

Map interface Hierarchy :

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

Paint Digram [22-FEB-25]

Methods of Map interface :

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

1) Object put(Object key, Object value) :- To insert one entry in the Map collection. It will return the value of old Object key, if the key is already available(Duplicate key), If key is not available (new key) then it will return null.

2) Object putIfAbsent(Object key, Object value) :- It will insert an entry, if and only if, key is not available , if the key is already available then it will not insert the Entry to the Map Collection

3) Object get(Object key) :- It will return corresponding key value, if the key is not available then it will return null.

4) Object getOrDefault(Object key, Object defaultValue) :- To avoid null value this method has been introduced from JDK 1.8V, here we can pass some defaultValue to avoid the null value.

5) boolean containsKey(Object key) :- To Search a particular key

6) boolean containsValue(Object value) :- To Search a particular value

7) int size() :- To count the number of Entries.

8) remove(Object key) :- One complete entry will be removed.

9) void clear() :- Used to clear the Map

10) boolean isEmpty() :- To verify Map is empty or not?

11) void putAll(Map m) :- Merging of two Map collection

Methods of Map interface to convert the Map into Collection :

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

We have Map interafce methods through which we can convert Map interface into Collection interface which is known as collection views method.

1) public Set<Object> keySet() : It will retrieve all the keys.

2) public Collection values() : It will retrieve all the values.

3) public Set<Map.Entry> entrySet() : It will retrieve key and value both in a single object.

a) getKey()

b) getValue()

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

HashMap<K,V> [UNORDERED, UNSORTED, NO DUPLICATE KEY]

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

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Serializable, Clonable

It is a predefined class available in java.util package under Map interface available from JDK 1.2v.

It gives us unsorted and Unordered map. when we need a map and we don't care about the order while iterating the elements through it then we should use HashMap.

It inserts the element based on the hashCode of the Object key using hashing technique [hasing alogorithhm]

It does not accept duplicate keys but value may be duplicate.

It accepts only one null key(because duplicate keys are not allowed) but multiple null values are allowed.

HashMap is not synchronized.

Time complexcity of search, insert and delete will be O(1)

We should use HashMap to perform fast searching opeartion.

For eliminating duplicate keys in hashMap object we should compulsory follow the contract between hashcode and equals(Object obj) OR Use record

It contains 4 types of constructor

1) HashMap hm1 = new HashMap();

It will create the HashMap Object with default capacity is 16. The default load fator or Fill Ratio is 0.75 (75% of HashMap is filled up then new HashMap Object will be created having double capacity)

2) HashMap hm2 = new HashMap(int initialCapacity);

will create the HashMap object with user specified capacity

3) HashMap hm3 = new HashMap(int initialCapacity, float loadFactor);

we can specify our own initialCapacity and loadFactor(by default load factor is 0.75)

4)HashMap hm4 = new HashMap(Map m);

Interconversion of Map Collection

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

24-02-2025

-----------

hasCode() and equals(Object obj) method of String class :

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

String class has overridden hashCode() and equals(Object obj)

method so, String class equals(Object obj) method is meant for

content comparison.

Once String class equals(Object obj) method will return true then the hashCode of both the Object must be same.

Program 1:

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

public class StringClassHashCodeAndEquals {

public static void main(String[] args)

{

String s1 = new String("india");

String s2 = new String("india");

System.out.println(s1.equals(s2));

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

}

Program 02 :

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

public class StringClassHashCodeAndEquals {

public static void main(String[] args)

{

String s1 = "india";

String s2 = new String("india");

System.out.println(s1==s2);

System.out.println(s1.equals(s2));

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

}

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

\*\*\*\* How HashMap<K,V> works internally ?

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

a) While working with HashSet or HashMap every object must be compared because duplicate objects are not allowed.

b) Whenever we add any new key to verify whether key is unique or duplicate, HashMap internally uses hashCode(), == operator and equals method.

c) While adding the key object in the HashMap, first of all it will invoke the hashCode() method to retrieve the corresponding key hashcode value.

Example :- hm.put(key,value);

then internally key.hashCode();

d) If the newly added key and existing key hashCode value both are same (Hash collision), then only == operator is used for comparing those keys by using reference or memory address, if both keys references are same then existing key value will be replaced with new key value.

If the reference of both keys are different then only equals(Object obj) method is invoked to compare those keys by using state(data). [content comparison]

If the equals(Object obj) method returns true (content wise both keys are same), this new key is duplicate then existing key value will be replaced by new key value.

If equals(Object obj) method returns false, this new key is unique, new entry (key-value) will be inserted in the same Bucket by using Singly LinkedList

Note :- equals(Object obj) method is invoked only when two keys are having same hashcode as well as their references are different.

e) Actually by calling hashcode method we are not comparing the objects, we are just storing the objects in a group so the currently adding key object will be compared with its SAME HASHCODE GROUP objects, but not with all the keys which are available in the Map.

f) The main purpose of storing objects into the corresponding group to decrease the number of comparison so the efficiency of the program will increase.

g) To insert an entry in the HashMap, HashMap internally uses Hashtable data structure.

h) Now, for storing same hashcode object into a single group, hash table data structure internally uses one more data structure called Bucket.

i) The Hashtable data structure internally uses Node class array object.

j) The bucket data structure internally uses LinkedList data structure, It is a singlly linked list again implemented by Node class only.

\*k) A bucket is group of entries of same hash code keys.

l) Performance wise LinkedList is not good to serach, so from java 8 onwards LinkedList is changed to Binary tree to decrease the number of comparison within the same bucket hashcode if the number of entries are greater than 8.

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

\*\* equals() and hashCode() method contract :

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

Both the methods are working together to find out the duplicate objects in the Map.

\*If equals() method invoked on two objects and it returns true then hashcode of both the objects must be same.

Note : IF TWO OBJECTS ARE HAVING SAME HASH CODE THEN IT MAY BE SAME OR DIFFERENT BUT IF EQUALS(OBJECT OBJ) METHOD RETURNS TRUE THEN BOTH OBJECTS MUST RETURN SAME HASHCODE.

package com.ravi.map;

import java.util.HashMap;

public class HashMapInternal {

public static void main(String[] args)

{

HashMap<String,Integer> hm1 = new HashMap<>();

hm1.put("A", 1);

hm1.put("A", 2);

hm1.put(new String("A"), 3);

System.out.println("Size is :"+hm1.size());

System.out.println(hm1);

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

HashMap<Integer,Integer> hm2 = new HashMap<>();

hm2.put(128, 1);

hm2.put(128, 2);

System.out.println("Size is :"+hm2.size());

System.out.println(hm2);

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

HashMap<Object,Object> hm3 = new HashMap<>();

hm3.put("A", 1);

hm3.put("A", 2);

hm3.put(new String("A"), 3);

hm3.put(65, 4);

System.out.println("Size is :"+hm3.size());

System.out.println(hm3);

}

}

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

What will happen if we don't follow the contract ?

Case 1 :

--------

If we override only equals(Object obj)

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

If we override only equals(Object obj) method for content comparison

then same object (duplicate object) will have different hashcode (due to Object class hashCode()) hence same object (content wise) will move into two different buckets [Duplication].

Case 2 :

--------

If we override only hashCode() method

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

If we overrdie only hashCode() method then two objects which are having same hashcode (due to overriding) will go to same bucket but == operator and equals(Object obj) method of Object class, both will return false hence duplicate object will be inserted into same bucket by using Singly LinkedList.

So, the conclusion is, compulsory we need to override both the methods for removing duplicate elements.

package com.ravi.map;

import java.util.HashMap;

import java.util.Objects;

class Customer

{

private Integer customerId;

private String customerName;

@Override

public String toString() {

return "Customer [customerId=" + customerId + ", customerName=" + customerName + "]";

}

public Customer(Integer customerId, String customerName)

{

super();

this.customerId = customerId;

this.customerName = customerName;

}

@Override

public int hashCode() {

return Objects.hash(customerId, customerName);

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Customer other = (Customer) obj;

return Objects.equals(customerId, other.customerId) && Objects.equals(customerName, other.customerName);

}

}

public class HashMapInternalDemo1

{

public static void main(String[] args)

{

Customer c1 = new Customer(111, "Scott");

Customer c2 = new Customer(111, "Scott");

System.out.println(c1.hashCode()+" : "+c2.hashCode());

System.out.println(c1.equals(c2));

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

HashMap<Customer,String> map = new HashMap<>();

map.put(c1, "A"); //c1 = A ---> c1 = B

map.put(c2, "B");

System.out.println(map.size());

System.out.println(map);

}

}

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

Customer class we are using as a HashMap key so it must override

hashCode() and equals(Object obj) as well as at advanced level, It must be immutable class.

All the Wrapper classes and String class are immutable as well as

hashCode() and equals(Object obj) methods are overridden in these classes so perfectly suitable to becoming HashMap key.

so final conclusion is, In our user-defined class which we want to use as a HashMap key must be immutable and hashCode() and equals(Object obj) method must be overridden.

Instead of BLC class we can also use simply record because record is implicitly final and hashCode() and equals(Object obj) methods are overridden.

package com.ravi.map;

import java.util.HashMap;

record Manager(Integer id, String managerName)

{

}

public class HashMapInternalDemo2 {

public static void main(String[] args)

{

Manager m1 = new Manager(111,"Alen");

Manager m2 = new Manager(111,"Alen");

HashMap<Manager,String> map = new HashMap<>();

map.put(m1, "Ameerpet");

map.put(m2, "S R Nagar");

System.out.println(map.size());

}

}

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

Programs on HashMap<K,V>

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

package com.ravi.map;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map.Entry;

public class HashMapDemo1

{

public static void main(String[] args)

{

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

map.put(1, "Vanilla");

map.put(2, "Butterscotch");

map.put(3, "Chocolate");

map.put(4, "Cotton Candy");

System.out.println("HashMap: " + map); //{key = value}

String value = map.get(2);

System.out.println("Value for key 2: " + value);

value = map.getOrDefault(3, "Key is not available");

System.out.println("Value for key 3: " + value);

boolean hasKey = map.containsKey(3);

System.out.println("HashMap contains key 3: " + hasKey);

boolean hasValue = map.containsValue("Vanilla");

System.out.println("HashMap contains value 'Vanilla': " + hasValue);

map.remove(1);

System.out.println("HashMap after removing key 1: " + map);

System.out.println("Iterating through HashMap:");

for (HashMap.Entry<Integer, String> entry : map.entrySet())

{

System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());

}

System.out.println("Iterating through Iterator");

Iterator<Entry<Integer, String>> itr = map.entrySet().iterator();

itr.forEachRemaining(System.out::println);

System.out.println("Iterating through forEach(BiConsumer<T,U>)");

map.forEach((k,v)-> System.out.println("Key is :"+k+" Value is :"+v));

int size = map.size();

System.out.println("Size of HashMap: " + size);

map.clear();

System.out.println("HashMap after clearing: " + map); //{}

}

}

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

package com.ravi.map;

import java.util.HashMap;

public class HashMapDemo2

{

public static void main(String[] args)

{

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

studentRecords.put(101, "Scott");

studentRecords.put(102, "Smith");

studentRecords.put(103, "Martin");

studentRecords.put(104, "Aryan");

System.out.println("Student Records: " + studentRecords);

int searchId = 103;

String studentName = studentRecords.get(searchId);

if (studentName != null)

{

System.out.println("Student with ID " + searchId + " is " + studentName);

}

else

{

System.out.println("Student with ID " + searchId + " not found.");

}

System.out.println(studentRecords.put(103, "Rahul"));

System.out.println("Updated Records: " + studentRecords);

studentRecords.remove(104);

System.out.println("Records after removal: " + studentRecords);

int idToCheck = 101;

System.out.println("Does ID " + idToCheck + " exist? " + studentRecords.containsKey(idToCheck));

String nameToCheck = "Aryan";

System.out.println("Does name '" + nameToCheck + "' exist? " + studentRecords.containsValue(nameToCheck));

System.out.println("Iterating through records:");

for(HashMap.Entry<Integer, String> entry : studentRecords.entrySet())

{

System.out.println("ID: " + entry.getKey() + ", Name: " + entry.getValue());

}

studentRecords.clear();

System.out.println("All records cleared: " + studentRecords); //{}

}

}

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

package com.ravi.map;

import java.util.Collection;

import java.util.HashMap;

import java.util.Set;

public class HashMapDemo3

{

public static void main(String[] args)

{

HashMap<Integer,String> newmap1 = new HashMap<>();

HashMap<Integer,String> newmap2 = new HashMap<>();

newmap1.put(1, "OCPJP");

newmap1.put(2, "is");

newmap1.put(3, "best");

System.out.println("Values in newmap1: "+ newmap1);

newmap2.put(4, "Exam");

newmap2.putAll(newmap1);

System.out.println("Iterating through forEach()");

newmap2.forEach((k,v)->System.out.println(k+" : "+v));

System.out.println("All the Unique keys");

Set<Integer> setOfKeys = newmap2.keySet();

System.out.println(setOfKeys);

System.out.println("All the values");

Collection<String> values = newmap2.values();

System.out.println(values);

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

System.out.println("Loose Coupling for Merging one Map to another");

HashMap<Integer, String> hm1 = new HashMap<>();

hm1.put(1, "Ravi");

hm1.put(2, "Rahul");

hm1.put(3, "Rajen");

HashMap<Integer, String> hm2 = new HashMap<>(hm1);

System.out.println("Mapping of HashMap hm1 are : " + hm1);

System.out.println("Mapping of HashMap hm2 are : " + hm2);

}

}

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

package com.ravi.map;

import java.util.HashMap;

record Employee(Integer empId, String empName)

{

}

public class HashMapDemo4

{

public static void main(String[] args)

{

Employee e1 = new Employee(101,"Aryan");

Employee e2 = new Employee(102,"Pooja");

Employee e3 = new Employee(101,"Aryan");

Employee e4 = e2;

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

hm.put(e1,"Ameerpet");

hm.put(e2,"S.R Nagar");

hm.put(e3,"Begumpet");

hm.put(e4,"Panjagutta");

hm.forEach((k,v)-> System.out.println(k+" : "+v));

}

}

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

package com.ravi.map;

import java.util.HashMap;

public class HashMapDemo5

{

public static void main(String[] args)

{

// Create a HashMap to store book titles and their availability (true = available, false = borrowed)

HashMap<String, Boolean> library = new HashMap<>();

library.put("Core Java", true);

library.put("Advanced Java", true);

library.put("HTML", false);

library.put("JavaScript", true);

// Display the initial library status

System.out.println("Initial Library Status: " + library);

// Borrow a book

String bookToBorrow = "Advanced Java";

if (library.containsKey(bookToBorrow) && library.get(bookToBorrow))

{

library.put(bookToBorrow, false);

System.out.println(bookToBorrow + " has been borrowed.");

}

else

{

System.out.println(bookToBorrow + " is not available for borrowing.");

}

String bookToReturn = "HTML";

if (library.containsKey(bookToReturn) && !library.get(bookToReturn))

{

library.put(bookToReturn, true); // Mark the book as available

System.out.println(bookToReturn + "Book has been returned.");

}

else

{

System.out.println(bookToReturn + "Book is not borrowed.");

}

// Check the availability of a book

String bookToCheck = "JavaScript";

if (library.containsKey(bookToCheck))

{

String availability = library.get(bookToCheck) ? "available" : "borrowed";

System.out.println(bookToCheck + " Book is " + availability + ".");

}

else

{

System.out.println(bookToCheck + " is not in the library.");

}

//Display the final library status

System.out.println("Final Library Status:");

for (HashMap.Entry<String, Boolean> entry : library.entrySet())

{

String status = entry.getValue() ? "Available" : "Borrowed";

System.out.println("Book: " + entry.getKey() + ", Status: " + status);

}

}

}

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

LinkedHashSet<E> [It is the order version of HashSet]

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

public class LinkedHashSet extends HashSet implements Set, Clonable, Serializable

It is a predefined class in java.util package under Set interface and introduced from java 1.4v.

It is the sub class of HashSet class.

It is an orderd version of HashSet that maintains a doubly linked list across all the elements.

It internally uses Hashtable and LinkedList data structures.

We should use LinkedHashSet class when we want to maintain an order.

When we iterate the elements through HashSet the order will be unpredictable, while when we iterate the elements through LinkedHashSet then the order will be same as they were inserted in the collection.

It accepts hetrogeneous and null value is allowed.

It has same constructor as HashSet class.

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

import java.util.\*;

public class LinkedHashSetDemo

{

public static void main(String args[])

{

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

lhs.add("Ravi");

lhs.add("Vijay");

lhs.add("Ravi");

lhs.add("Ajay");

lhs.add("Pawan");

lhs.add("Shiva");

lhs.add(null);

lhs.add("Ganesh");

lhs.forEach(str -> System.out.println(str));

}

}

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

import java.util.\*;

public class LinkedHashSetDemo1

{

public static void main(String[] args)

{

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

linkedHashSet.add(10);

linkedHashSet.add(5);

linkedHashSet.add(15);

linkedHashSet.add(20);

linkedHashSet.add(5);

System.out.println("LinkedHashSet elements: " + linkedHashSet);

System.out.println("LinkedHashSet size: " + linkedHashSet.size());

int elementToCheck = 15;

if (linkedHashSet.contains(elementToCheck))

{

System.out.println(elementToCheck + " is present in the LinkedHashSet.");

}

else

{

System.out.println(elementToCheck + " is not present in the LinkedHashSet.");

}

int elementToRemove = 10;

linkedHashSet.remove(elementToRemove);

System.out.println("After removing " + elementToRemove + ", LinkedHashSet elements: " + linkedHashSet);

linkedHashSet.clear();

System.out.println("After clearing, LinkedHashSet elements: " + linkedHashSet); //[]

}

}

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

LinkedHashMap<K,V> [It maintains insertion order]

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

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

It is a predefined class available in java.util package under Map interface available from 1.4.

It is the sub class of HashMap class.

It maintains insertion order. It contains a doubly linked with the elements or nodes so It will iterate more slowly in comparison to HashMap.

It uses Hashtable and LinkedList data structure.

If We want to fetch the elements in the same order as they were inserted in the Map then we should go with LinkedHashMap.

It accepts one null key and multiple null values.

It is not synchronized.

It has also 4 constructors same as HashMap

1) LinkedHashMap hm1 = new LinkedHashMap();

will create a LinkedHashMap with default capacity 16 and load factor 0.75

2) LinkedHashMap hm1 = new LinkedHashMap(iny initialCapacity);

3) LinkedHashMap hm1 = new LinkedHashMap(int initialCapacity, float loadFactor);

4) LinkedHashMap hm1 = new LinkedHashMap(Map m);

import java.util.\*;

public class LinkedHashMapDemo

{

public static void main(String[] args)

{

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

l.put(1,"abc");

l.put(3,"xyz");

l.put(2,"pqr");

l.put(4,"def");

l.put(null,"ghi");

System.out.println(l);

}

}

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

SortedSet<E> interface :

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

It is the sub interface of Set<E> interface available from JDK 1.2V

As we know we can't perform sorting operation on HashSet and LinkedHashSet object by using Collections.sort(List<E> list)

In order to provide automatic sorting facility, java software people has provided SortedSet<E> interface.

It can sort the element either in default natural sorting order(java.lang.Comparable<E>) OR user defined sorting order (java.util.Comparator)

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

\*\*\*Difference between Comparable<E> and Comparator<E>

Available in paint Digram [24-FEB-25]

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

25-02-2025

-----------

Program on Comparable :

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

2 files :

----------

Employee.java

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

package com.ravi.comparable;

public record Employee(Integer id, String name) implements Comparable<Employee>

{

@Override

public int compareTo(Employee e2)

{

return Integer.compare(this.id, e2.id);

}

}

EmployeeComparable.java

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

package com.ravi.comparable;

import java.util.ArrayList;

import java.util.Collections;

public class EmployeeComparable {

public static void main(String[] args)

{

ArrayList<Employee> listOfEmployees = new ArrayList<>();

listOfEmployees.add(new Employee(333, "Scott"));

listOfEmployees.add(new Employee(222, "Alen"));

listOfEmployees.add(new Employee(111, "Smith"));

listOfEmployees.add(new Employee(444, "Zuber"));

System.out.println("Original data ");

listOfEmployees.forEach(System.out::println);

Collections.sort(listOfEmployees);

System.out.println("After Sorting based on the ID : ");

listOfEmployees.forEach(System.out::println);

}

}

Limitation of Comparable<T> interface :

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

1) Due to the current object support we need to modify the BLC

class so Lambda Expression will not work with Comparable even it is a functional interface.

2) We can write only one sorting logic.

3) It is a Functional interface but due to current object support we can write lambda expression

In avoid the above said problem, We introduced Comparator :

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

Program on Comparator :

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

package com.ravi.comparable;

public record Product(Integer pid, String pname, Double price)

{

}

package com.ravi.comparable;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

public class ProductComparator {

public static void main(String[] args)

{

ArrayList<Product> listOfProduct = new ArrayList<>();

listOfProduct.add(new Product(444, "Laptop", 84000D));

listOfProduct.add(new Product(333, "Camera", 54000D));

listOfProduct.add(new Product(111, "Mobile", 24000D));

listOfProduct.add(new Product(222, "HeadPhone", 4000D));

System.out.println("Original Data");

listOfProduct.forEach(System.out::println);

System.out.println("After Sorting based on the Price ");

//Anonymous inner class

Comparator<Product> cmpPrice = new Comparator<Product>()

{

@Override

public int compare(Product p1, Product p2)

{

return Double.compare(p1.price(), p2.price());

}

};

Collections.sort(listOfProduct, cmpPrice);

listOfProduct.forEach(System.out::println);

System.out.println("After Sorting based on the ID :");

//Passing Functional interface as a parameter

Comparator<Product> cmpId = (p1,p2)-> Integer.compare(p1.pid(), p2.pid());

Collections.sort(listOfProduct , cmpId);

listOfProduct.forEach(System.out::println);

System.out.println("After Sorting based on the Name :");

//Lambda

Collections.sort(listOfProduct, (s1,s2)-> s1.pname().compareTo(s2.pname()));

listOfProduct.forEach(System.out::println);

}

}

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

How to sort String data in descending order :

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

Here we can't modify the source code of String class, It is available in String.class file format so to provide descending order we should use Comparator as shown in the Program below :

package com.ravi.comparable;

import java.util.ArrayList;

import java.util.Collections;

public class StrinfDescending {

public static void main(String[] args)

{

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

al.add("Hyderabad");

al.add("Indore");

al.add("Ahmadabad");

al.add("Bombay");

Collections.sort(al, (s1,s2)-> s2.compareTo(s1));

System.out.println(al);

}

}

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

List interface sort(Comparator<T> cmp) :

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

package com.ravi.comparable;

import java.util.ArrayList;

import java.util.Collections;

public class StrinfDescending {

public static void main(String[] args)

{

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

al.add("Hyderabad");

al.add("Indore");

al.add("Ahmadabad");

al.add("Bombay");

al.sort((s1,s2)-> s1.compareTo(s2));

System.out.println(al);

}

}

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

TreeSet<E>

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

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Clonable, Serializable

It is a predefined class available in java.util package under Set interface available from JDK 1.2v.

TreeSet class uses Red Black tree data structure.

It will sort the elements in natural sorting order i.e ascending order in case of number , and alphabetical order or Dictionary order in the case of String. In order to sort the elements according to user choice, It uses Comparable/Comparator interface.

It does not accept duplicate and null value (java.lang.NullPointerException)

It does not accept non comparable type of Objects if we try to insert it will throw a runtime exception i.e java.lang.ClassCastException

TreeSet implements NavigableSet.

NavigableSet extends SortedSet.

It contains 4 types of constructors :

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

1) TreeSet t1 = new TreeSet();

create an empty TreeSet object, elements will be inserted in using Comparable.

2) TreeSet t2 = new TreeSet(Comparator c);

Customized sorting order.

3) TreeSet t3 = new TreeSet(Collection c);

loose coupling.

4) TreeSet t4 = new TreeSet(SortedSet s);

We can merge two TreeSet object to copy the data.

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

package com.ravi.comparable;

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args)

{

TreeSet<Object> ts1 = new TreeSet<>();

ts1.add(45);

ts1.add(23);

ts1.add(9);

ts1.add(99);

//ts1.add("NIT"); //Invalid

//ts1.add(null); //Invalid

System.out.println(ts1);

}

}

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

import java.util.\*;

public class TreeSetDemo1

{

public static void main(String[] args)

{

TreeSet<String> t1 = new TreeSet<>();

t1.add("Orange");

t1.add("Mango");

t1.add("Pear");

t1.add("Banana");

t1.add("Apple");

System.out.println("In Ascending order");

t1.forEach(i -> System.out.println(i));

TreeSet<String> t2 = new TreeSet<>();

t2.add("Orange");

t2.add("Mango");

t2.add("Pear");

t2.add("Banana");

t2.add("Apple");

System.out.println("In Descending order");

Iterator<String> itr2 = t2.descendingIterator(); //for descending order

itr2.forEachRemaining(x -> System.out.println(x));

}

}

Note :- descendingIterator() is a predefined method of TreeSet class which will traverse in the descending order and return type of this method is Iterator interface available from JDK 1.6

public Iterator descendingIterator()

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

//How to sort TreeSet by using Comparable :

package com.ravi.comparable;

import java.util.TreeSet;

record Customer(Integer cid, String cname) implements Comparable<Customer>

{

@Override

public int compareTo(Customer c2)

{

return Integer.compare(this.cid(), c2.cid());

}

}

public class TreeSetExample {

public static void main(String[] args)

{

TreeSet<Customer> ts2 = new TreeSet<>();

ts2.add(new Customer(333, "Scott"));

ts2.add(new Customer(111, "Zuber"));

ts2.add(new Customer(222, "Aryan"));

ts2.forEach(System.out::println);

}

}

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

package com.ravi.comparable;

import java.util.TreeSet;

record Prod(Integer id, String name)

{

}

public class TreeSetComparator {

public static void main(String[] args)

{

TreeSet<Prod> ts2 = new TreeSet<>((p1, p2)-> p1.name().compareTo(p2.name()));

ts2.add(new Prod(333, "Mobile"));

ts2.add(new Prod(222, "Camera"));

ts2.add(new Prod(111, "Laptop"));

ts2.forEach(System.out::println);

}

}

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

package com.ravi.comparable;

import java.util.ArrayList;

import java.util.TreeSet;

public class TreeSetExampleDemo {

public static void main(String[] args)

{

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

list.add("B.Tech");

list.add("M.Tech");

list.add("BCA");

list.add("MCA");

list.add("BA");

list.add("MA");

//ArrayList to TreeSet

TreeSet<String> ts1 = new TreeSet<>(list);

System.out.println(ts1);

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

TreeSet<Integer> ts2 = new TreeSet<>();

ts2.add(34);

ts2.add(45);

ts2.add(98);

ts2.add(12);

//TreeSet to ArrayList

ArrayList<Integer> listOfNum = new ArrayList<>(ts2);

System.out.println(listOfNum);

//TreeSet to TreeSet(SortedSet)

TreeSet<Integer> ts3 = new TreeSet<>(ts2);

System.out.println(ts3);

}

}

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

26-02-2025

----------

TreeSet Custom Sorting Order :

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

package com.ravi.treeset;

import java.util.TreeSet;

record Employee(Integer id, String name, Integer age)

{

}

public class CustomComparator {

public static void main(String[] args)

{

TreeSet<Employee> ts1 = new TreeSet<>((e1,e2)-> Integer.compare(e1.id(), e2.id()));

ts1.add(new Employee(201, "Zaheer", 24));

ts1.add(new Employee(101, "Aryan", 27));

ts1.add(new Employee(301, "Pooja", 26));

System.out.println("Sorted Based on the ID :");

ts1.forEach(i -> System.out.println(i));

TreeSet<Employee> ts2 = new TreeSet<>((e1, e2)-> e1.name().compareTo(e2.name()));

ts2.add(new Employee(201, "Zaheer", 24));

ts2.add(new Employee(101, "Aryan", 27));

ts2.add(new Employee(301, "Pooja", 26));

System.out.println("Sorted Based on the Name :");

ts2.forEach(i -> System.out.println(i));

}

}

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

TreeMap<K,V>

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

public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V> , Clonable, Serializable

It is a predefined class avaialble in java.util package under Map interface available for 1.2V.

It is a sorted map that means it will sort the elements by natural sorting order based on the key or by using Comparator interface as a constructor parameter.

It does not allow non comparable keys.(ClassCastException)

It does not accept null key but null value allowed.(NPE)

It uses Red black tree data structure.

TreeMap implements NavigableMap and NavigableMap extends SortedMap. SortedMap extends Map interface.

TreeMap contains 4 types of Constructors :

1) TreeMap tm1 = new TreeMap(); //creates an empty TreeMap

2) TreeMap tm2 = new TreeMap(Comparator cmp); //user defined soting logic

3) TreeMap tm3 = new TreeMap(Map m); //loose Coupling

4) TreeMap tm4 = new TreeMap(SortedMap m);

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

import java.util.\*;

public class TreeMapDemo

{

public static void main(String[] args)

{

TreeMap<Object,String> t = new TreeMap<>();

t.put(4,"Ravi");

t.put(7,"Aswin");

t.put(2,"Ananya");

t.put(1,"Dinesh");

t.put(9,"Ravi");

t.put(3,"Ankita");

t.put(5,null);

//t.put("six", "Xyz");

//t.put(null, "abc");

System.out.println(t);

}

}

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

import java.util.\*;

public class TreeMapDemo1

{

public static void main(String args[])

{

TreeMap map = new TreeMap();

map.put("one","1");

map.put("two",null);

map.put("three","3");

map.put("four",4);

displayMap(map);

map.forEach((k, v) -> System.out.println("Key = " + k + ", Value = " + v));

}

static void displayMap(TreeMap map)

{

Collection c = map.entrySet(); //Set<Map.Entry>

Iterator i = c.iterator();

i.forEachRemaining(x -> System.out.println(x));

}

}

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

import java.util.\*;

public class TreeMapDemo2

{

public static void main(String[] argv)

{

Map<String,String> map = new TreeMap<String,String>();

map.put("key2", "value2");

map.put("key3", "value3");

map.put("key1", "value1");

System.out.println(map);

}

}

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

Custom Sorting order by using TreeMap<K,V>

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

package com.ravi.treemap\_demo\_ex;

import java.util.TreeMap;

record Product(Integer pid, String pname)

{

}

public class TreeMapDemo3

{

public static void main(String[] args)

{

TreeMap<Product,String> tm1 = new TreeMap<>((p1, p2)-> p1.pid().compareTo(p2.pid()));

tm1.put(new Product(333, "Laptop"), "Hyderabad");

tm1.put(new Product(444, "Mobile"), "Pune");

tm1.put(new Product(111, "HeadPhone"), "Indore");

tm1.put(new Product(222, "Camera"), "Mumbai");

System.out.println("Sorting based on the Product Id in Ascending Order");

tm1.forEach((k,v)-> System.out.println(k+" : "+v));

TreeMap<Product,String> tm2 = new TreeMap<>((p1, p2)-> p2.pid().compareTo(p1.pid()));

tm2.put(new Product(333, "Laptop"), "Hyderabad");

tm2.put(new Product(444, "Mobile"), "Pune");

tm2.put(new Product(111, "HeadPhone"), "Indore");

tm2.put(new Product(222, "Camera"), "Mumbai");

System.out.println("Sorting based on the Product Id in Descending Order");

tm2.forEach((k,v)-> System.out.println(k+" : "+v));

}

}

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

Hashtable<K,V>

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

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable.

It is predefined class available in java.util package under Map interface from JDK 1.0.

Like Vector, Hashtable is also form the birth of java so called legacy class.

It is the sub class of Dictionary class which is an abstract class.

\*The major difference between HashMap and Hashtable is, HashMap methods are not synchronized where as Hastable methods are synchronized.

HashMap can accept one null key and multiple null values where as Hashtable does not contain anything as a null(key and value both). if we try to add null then JVM will throw an exception i.e NullPointerException.

The initial default capacity of Hashtable class is 11 where as loadFactor is 0.75.

It has also same constructor as we have in HashMap.(4 constructors)

1) Hashtable hs1 = new Hashtable();

It will create the Hashtable Object with default capacity as 11 as well as load factor is 0.75

2) Hashtable hs2 = new Hashtable(int initialCapacity);

will create the Hashtable object with specified capacity

3) Hashtable hs3 = new Hashtable(int initialCapacity, float loadFactor);

we can specify our own initialCapacity and loadFactor

4) Hashtable hs = new Hashtable(Map c);

Interconversion of Map Collection

import java.util.\*;

public class HashtableDemo

{

public static void main(String args[])

{

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

map.put(1, "Java");

map.put(2, "is");

map.put(3, "best");

map.put(4,"language");

//map.put(5,null);

System.out.println(map);

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

for(Map.Entry m : map.entrySet())

{

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

}

}

}

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

[Diagram for this Program : 26th FEB]

import java.util.\*;

public class HashtableDemo1

{

public static void main(String args[])

{

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

map.put(1,"Priyanka");

map.put(2,"Ruby");

map.put(3,"Vibha");

map.put(4,"Kanchan");

map.putIfAbsent(5,"Bina");

map.putIfAbsent(24,"Pooja");

map.putIfAbsent(26,"Ankita");

map.putIfAbsent(1,"Sneha");

System.out.println("Updated Map: "+map);

}

}

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

\*\*WeakHashMap<K,V>

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

public class WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>

It is a predefined class in java.util package under Map interface.It was introduced from JDK 1.2v onwards.

While working with HashMap, keys of HashMap are of strong reference type. This means the entry of map will not be deleted by the garbage collector even though the key is set to be null as well as Object is also not eligible for Garbage Collector.

On the other hand while working with WeakHashMap, keys of WeakHashMap are of weak reference type. This means the entry and corresponding object of a map is deleted by the garbage collector if the key value is set to be null because it is of weak type.

So, HashMap dominates over Garbage Collector where as Garbage Collector dominates over WeakHashMap.

It does not implements Cloneable and Serailizable because It is mainly used for inventory system where we need to manage the data and we can insert and delete object data frequently in the inventory.

It contains 4 types of Constructor :

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

1) WeakHashMap wm1 = new WeakHashMap();

Creates an empty WeakHashMap object with default capacity is 16 and load fator 0.75

2) WeakHashMap wm2 = new WeakHashMap(int initialCapacity);

3) WeakHashMap wm3 = new WeakHashMap(int initialCapacity, float loadFactor);

Eg:- WeakHashMap wm = new WeakHashMap(10,0.9);

capacity - The capacity of this map is 10. Meaning, it can store 10 entries.

loadFactor - The load factor of this map is 0.9. This means whenever our hashtable is filled up by 90%, the entries are moved to a new hashtable of double the size of the original hashtable.

4) WeakHashMap wm4 = new WeakHashMap(Map m);

package com.ravi.weakhashmap\_demo;

import java.util.WeakHashMap;

record Product(Integer id, String name)

{

@Override

public void finalize()

{

System.out.println("Product Object is eligible for GC");

}

}

public class WeakHashMapDemo1

{

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

{

Product p1 = new Product(111, "Laptop");

WeakHashMap<Product,String> map = new WeakHashMap<>();

map.put(p1, "Hyd");

System.out.println(map);

p1 = null;

System.gc();

Thread.sleep(5000);

System.out.println(map); //{}

}

}

//Program :

-----------

package com.ravi.weakhashmap\_demo;

import java.util.WeakHashMap;

record Product(Integer id, String name)

{

@Override

public void finalize()

{

System.out.println("Product Object is eligible for GC");

}

}

public class WeakHashMapDemo1

{

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

{

Product p1 = new Product(111, "Laptop");

WeakHashMap<Product,String> map = new WeakHashMap<>();

map.put(p1, "Hyd");

System.out.println(map);

p1 = null;

System.gc();

Thread.sleep(5000);

System.out.println(map); //{}

}

}

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

How to generate OR find out System hashcode value :

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

System class has provided a predefined native and static method called identityHashCode(Object obj), It is used to

generate System hashcode.

It accepts Object as a parameter and return type of this method is int.

If we don't override hashCode() method in the corresponding class then System generated Hashcode and Object class hashcode would be same.

public static native int identityHashCode(Object obj)

package com.ravi.identity\_hash\_code;

class Student

{

private Integer studentId;

}

public class SystemHashcode {

public static void main(String[] args)

{

String str = "india";

System.out.println("String class hashcode :"+str.hashCode());

int systemHashCode = System.identityHashCode(str);

System.out.println("System hash Code :"+systemHashCode);

System.out.println("========================");

Student s1 = new Student();

System.out.println("Object class hashCode :"+s1.hashCode());

int identityHashCode = System.identityHashCode(s1);

System.out.println("System hash Code :"+identityHashCode);

}

}

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

IdentityHashMap<K,V> [Comparing keys based on the Memory reference]

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

public class IdentityHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Clonable, Serializable.

It was introduced from JDK 1.4 onwards.

The IdentityHashMap uses == operator to compare keys.

As we know HashMap uses equals() and hashCode() method for comparing the keys based on the hashcode of the object it will serach the bucket location and insert the entry their only.

So We should use IdentityHashMap where we need to compare the keys by using reference or memory address instead of logical equality.

HashMap uses hashCode of the "Object key" to find out the bucket loaction in Hashtable, on the other hand IdentityHashMap does not use hashCode() method actually It uses System.identityHashCode(Object o)

IdentityHashMap is more faster than HashMap in case of key Comparison.

The default capacity is 32.

It has three constrcutors, It does not contain loadFactor specific constructor.

1) IdentityHashMap ihm1 = new IdentityHashMap();

Will create IdentityHashMap with default capacity is 32.

2) IdentityHashMap ihm2 = new IdentityHashMap(int initialCapacity);

Will create IdentityHashMap with user specified capacity

3) IdentityHashMap ihm3 = new IdentityHashMap(Map map)

package com.ravi.identity\_hash\_map;

import java.util.HashMap;

import java.util.IdentityHashMap;

public class IdentityHashMapDemo1

{

public static void main(String[] args)

{

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

hm.put("NIT", "Ameerpet");

hm.put(new String("NIT"), "Hyderabad");

System.out.println(hm.size());

System.out.println(hm);

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

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

ihm.put("NIT", "Ameerpet");

ihm.put(new String("NIT"), "Hyderabad");

System.out.println(ihm.size());

System.out.println(ihm);

}

}

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

27-02-2025

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

Methods of SortedSet interface :

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

public E first() :- Will fetch first element

public E last() :- Will fetch last element

public SortedSet headSet(int range) :- Will fetch the values which are less than specified range.

public SortedSet tailSet(int range) :- Will fetch the values which are equal and greater than the specified range.

public SortedSet subSet(int startRange, int endRange) :- Will fetch the range of values where startRange is inclusive and endRange is exclusive.

Note :- headSet(), tailSet() and subSet(), return type is SortedSet.

import java.util.\*;

public class SortedSetMethodDemo

{

public static void main(String[] args)

{

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

times.add(1205);

times.add(1505);

times.add(1545);

times.add(1600);

times.add(1830);

times.add(2010);

times.add(2100);

SortedSet<Integer> sub = new TreeSet<>();

sub = times.subSet(1545,2100);

System.out.println("Using subSet() :-"+sub);//[1545, 1600,1830,2010]

System.out.println(sub.first());

System.out.println(sub.last());

sub = times.headSet(1545);

System.out.println("Using headSet() :-"+sub); //[1205, 1505]

sub = times.tailSet(1545);

System.out.println("Using tailSet() :-"+sub); //[1545 to 2100]

}

}

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

By using above SortedSet<E> methods we can find the range of values but navigation among the element is not possible.

In order to provide navigation among the element, Java software people has introduced NavigableSet(I) interface from JDK 1.6V.

NavigableSet :

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

It is a predefined interface available in java.util package from JDK 1.6v

It is used to navigate among the elements, Unlike SortedSet which provides range of values. Here we can navigate among the values as shown below.

import java.util.\*;

public class NavigableSetDemo

{

public static void main(String[] args)

{

NavigableSet<Integer> ns = new TreeSet<>();

ns.add(1);

ns.add(2);

ns.add(3);

ns.add(4);

ns.add(5);

ns.add(6);

System.out.println("lower(3): " + ns.lower(3));//Just below than the specified element or null

System.out.println("floor(3): " + ns.floor(3)); //Equal less or null

System.out.println("higher(3): " + ns.higher(3));//Just greater than specified element or null

System.out.println("ceiling(3): " + ns.ceiling(3));//Equal or greater or null

}

}

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

Methods of SortedMap<K,V> interface :

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

1) firstKey() //first key

2) lastKey() //last key

3) headMap(int keyRange) //less than the specified range

4) tailMap(int keyRange) //equal or greater than the specified range

5) subMap(int startKeyRange, int endKeyRange) //the range of key where startKey will be inclusive and endKey will be exclusive.

return type of headMap(), tailMap() and subMap() return type would be SortedMap(I)

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

import java.util.\*;

public class SortedMapMethodDemo

{

public static void main(String args[])

{

SortedMap<Integer,String> map=new TreeMap<>();

map.put(100,"Amit");

map.put(101,"Ravi");

map.put(102,"Vijay");

map.put(103,"Rahul");

System.out.println("First Key: "+map.firstKey()); //100

System.out.println("Last Key "+map.lastKey()); //103

System.out.println("headMap: "+map.headMap(102)); //100 101

System.out.println("tailMap: "+map.tailMap(102)); //102 103

System.out.println("subMap: "+map.subMap(100, 102)); //100 101

}

}

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

Assignment for NavigableMap Methods :

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

1) ceilingEntry(K key)

2) ceilingKey(K key)

3) floorEntry(K key)

4) floorKey(K key)

5) higherEntry(K key)

6) higherKey(K key)

7) lowerEntry(K key)

8) lowerKey(K key)

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

Properties :

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

public class Properties extends Hashtable<K,V>

It is a legacy class and It represents a persistent set of properties.

It is subclass of Hashtable available in java.util package.

It is used to maintain the persistent data in the key-value form. It takes both key and value as a String format.

It is used to load properties file in our java application directly at runtime without compilation/deploymnet.

Constructors :

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

Commonly we are using this constructor :

Properties p1 = new Properties();

Creates an empty property list.

Methods :

----------

1) public void load(InputStream stream): Reads a property list (key and value pair) from the input byte stream.

2) public void load(Reader reader):Reads a property list (key and value pair) from the Character Oriented stream.

3) Object setProperty(String key, String value) : It Calls the Hashtable method put internally.

4) public String getProperty(String key) :Searches for the property with the specified key in this property list.

5) public void store(OutputStream out, String comments) : It

Writes this property list (key and element pairs) in this Properties table to the output stream.

6) public void store(Writer writer, String comments) : It

Writes this property list (key and element pairs) in this Properties table to the character stream.

Steps :

--------

1) Create a file whose extension must be .properties as shown below

db.properties

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

driver = oracle.jdbc.driver.OracleDriver

user = scott

password = ravi

PropertyDemo1.java

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

import java.io.\*;

import java.util.\*;

public class PropertyDemo

{

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

{

FileReader reader = new FileReader("D:\\new\\db.properties");

Properties p = new Properties();

p.load(reader);

String driver = p.getProperty("driver");

String userName = p.getProperty("user");

String password = p.getProperty("password");

System.out.println(driver);

System.out.println(userName);

System.out.println(password);

}

}

If we make changes in the properties file then directly (without compilation) we can take the the value in our java file so after any modification in the properties file we need not to re-compile/re-deploy our java program.

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

package properties;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Properties;

public class PropertiesDemo2 {

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

{

Properties properties = new Properties();

String filePath = "D:\\new\\data.properties";

var writer = new FileWriter(filePath);

try(writer)

{

properties.setProperty("book", "Java");

properties.setProperty("author", "James");

properties.setProperty("price", "1200");

properties.store(writer, "Book Properties set");

}

catch(Exception e)

{

e.printStackTrace();

}

//Reading the data from Properties file

var reader = new FileReader(filePath);

try(reader)

{

properties.load(reader);

System.out.println("Book Name is "+properties.getProperty("book"));

System.out.println("Author Name is "+properties.getProperty("author"));

System.out.println("Price Name is "+properties.getProperty("price"));

}

catch(Exception e)

{

e.printStackTrace();

}

}

}

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

Queue<E> interface

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

1) It is sub interface of Collection(I) available from JDK 1.5V hence it support all the methods of Collection interface.

2) It works in FIFO(First In first out)

3) It is an ordered collection.

4) In a queue, insertion is possible from last is called REAR where as deletion is possible from the starting is called FRONT of the queue.

5)In order to support Basis Queue operation, LinkedList class implements Deque and Deque interface exetnds Queue

interface.

28-02-2025

----------

PriorityQueue<E>

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

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

It is a predefined class in java.util package, available from Jdk 1.5 onwards.

It stores the elements using balanced binary heap tree, meaning the smallest element is at the head of the queue.

The elements of the priority queue are ordered according to their natural ordering (binary heap tree), or by using Comparator provided at queue construction time, depending on which constructor is used.

A priority queue does not permit null elements.

It provides natural sorting order so we can't take non-comparable objects(hetrogeneous types of Object)

The initial capacity of PriorityQueue is 11.

Constructor :

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

1) PriorityQueue pq1 = new PriorityQueue();

Will create PriorityQueue object with default capacity is 11, Elements will be inserted based on binary heap tree.

2) PriorityQueue pq2 = new PriorityQueue(int initialCapacity);

Will create PriorityQueue with user specified capacity

3) PriorityQueue pq3 = new PriorityQueue(int initialCapacity, Comparator cmp);

Will create PriorityQueue with user specified capacity and own userdefined order.

4) PriorityQueue pq3 = new PriorityQueue(Comparator cmp);

Will create PriorityQueue with user defined sorting order

5) PriorityQueue pq4 = new PriorityQueue(Collection c);

Loose coupling

Methods :-

----------

public boolean offer(E e) /public boolean add(E e) :- Used to add an element in the Queue

public E poll() :- It is used to fetch the elements from head of the queue, after fetching it will delete the element.

public E peek() :- It is also used to fetch the elements from head of the queue, Unlike poll it will only fetch but not delete the element.

public boolean remove(Object element) :- It is used to remove an element. The return type is boolean.

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

import java.util.PriorityQueue;

public class PriorityQueueDemo

{

public static void main(String[] argv)

{

PriorityQueue<Object> pq = new PriorityQueue<>();

pq.add("Orange");

pq.add("Apple");

pq.add("Mango");

pq.add("Guava");

pq.add("Grapes");

//pq.add(null); // Inavlid

//pq.add(23); //Invalid

System.out.println(pq);

}

}

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

import java.util.PriorityQueue;

public class PriorityQueueDemo1

{

public static void main(String[] argv)

{

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(11);

pq.add(2);

pq.add(4);

pq.add(6);

System.out.println(pq);

}

}

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

import java.util.PriorityQueue;

public class PriorityQueueDemo2

{

public static void main(String[] argv)

{

PriorityQueue<String> pq = new PriorityQueue<>();

pq.add("2");

pq.add("4");

pq.add("6");

System.out.print(pq.peek() + " "); // 6 9

pq.offer("1");

pq.offer("9");

pq.add("3"); // 2 2 3 4 4

pq.remove("1");

System.out.print(pq.poll() + " ");

if (pq.remove("2"))

{

System.out.print(pq.poll() + " ");

}

System.out.println(pq.poll() + " " + pq.peek()+" "+pq.poll());

}

}

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

package com.ravi.comparator;

import java.util.\*;

record Task(String name, Integer priority)

{

}

public class PriorityQueueDemo4

{

public static void main(String[] args)

{

PriorityQueue<Task> taskQueue = new PriorityQueue<>

((t1,t2)-> t1.priority().compareTo(t2.priority()));

taskQueue.add(new Task("Submit report", 4));

taskQueue.add(new Task("Find Bug", 2));

taskQueue.add(new Task("Write Program", 1));

taskQueue.add(new Task("Execute Program", 3));

while (!taskQueue.isEmpty())

{

System.out.println("Executing: " + taskQueue.poll());

}

}

}

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

Generics :

----------

What is the need of Generics ?

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

As we know our compiler is known for Strict type checking because java is a statically typed checked language.

The basic problem with collection is, It can hold any kind of Object.

ArrayList al = new ArrayList();

al.add("Ravi");

al.add("Aswin");

al.add("Rahul");

al.add("Raj");

al.add("Samir");

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

{

String s = (String) al.get(i);

System.out.println(s);

}

By looking the above code it is clear that Collection stores everything in the form of Object so here even after adding String type only we need to provide casting as shown below.

import java.util.\*;

class Test

{

public static void main(String[] args)

{

ArrayList al = new ArrayList();

al.add(12);

al.add(15);

al.add(18);

al.add(22);

al.add(24);

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

{

Integer x = (Integer) al.get(i);

System.out.println(x);

}

}

}

Note : Even we are accepting only Integer type of Object but still type casting is required.

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

import java.util.\*;

class Test2

{

public static void main(String[] args)

{

ArrayList t = new ArrayList(); //raw type

t.add("alpha");

t.add("beta");

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

{

String str =(String) t.get(i);

System.out.println(str);

}

t.add(1234);

t.add(1256);

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

{

String obj= (String)t.get(i); //we can't perform type casting here

System.out.println(obj);

}

}

}

Even after type casting there is no guarantee that the things which are coming from ArrayList Object is String only because we can add anything in the Collection as a result java.lang.ClassCastException

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

To avoid all the above said problem Generics came into picture from JDK 1.5 onwards

-> It deals with type safe Object so there is a guarantee of both the end i.e putting inside and getting outside.

Example:-

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

Now here we have a guarantee that only String can be inserted as well as only String will come out from the Collection so we can perform String related operation.

Advantages of Generics :

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

1) Type Safe Object (No Compilation warning)

2) No need of type casting

3) Strict compile time checking. (\*Type Erasure)

import java.util.\*;

public class Test3

{

public static void main(String[] args)

{

ArrayList<String> al = new ArrayList<>(); //Generic type

al.add("Ravi");

al.add("Ajay");

al.add("Vijay");

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

{

String name = al.get(i); //no type casting is required

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

}

}

}

//Program that describes the return type of any method can be type safe

//[We can apply generics on method return type]

package com.ravi.comparator;

import java.util.ArrayList;

import java.util.List;

record Dog(String name)

{

}

public class Test4 {

public static void main(String[] args)

{

Dog dog = getDogObject().get(0);

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

}

public static List<Dog> getDogObject()

{

List<Dog> listOfDogs = new ArrayList<Dog>();

listOfDogs.add(new Dog("A"));

listOfDogs.add(new Dog("B"));

listOfDogs.add(new Dog("C"));

return listOfDogs;

}

}

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

01-03-2025

-----------

Mixing Generic to non generic Type :

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

//Mixing generic with non-generic

import java.util.\*;

class Car

{

}

public class Test5

{

public static void main(String [] args)

{

ArrayList<Car> a = new ArrayList<>();

a.add(new Car());

a.add(new Car());

a.add(new Car());

ArrayList b = a; //assigning Generic to raw type

System.out.println(b);

}

}

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

//Mixing generic to non-generic

import java.util.\*;

public class Test6

{

public static void main(String[] args)

{

List<Integer> myList = new ArrayList<>();

myList.add(4);

myList.add(6);

myList.add(5);

UnknownClass u = new UnknownClass();

int total = u.addValues(myList);

System.out.println("The sum of Integer Object is :"+total);

}

}

class UnknownClass

{

public int addValues(List list) //generic to raw type OR

{

Iterator it = list.iterator();

int total = 0;

while (it.hasNext())

{

int i = ((Integer)it.next());

total += i; //total = 15

}

return total;

}

}

Note :-

In the above program the compiler will not generate any warning message because even though we are assigning type safe Integer Object to unsafe or raw type List Object but this List Object is not inserting anything new in the collection so there is no risk to the caller.

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

//Mixing generic to non-generic

import java.util.\*;

public class Test7

{

public static void main(String[] args)

{

List<Integer> myList = new ArrayList<>();

myList.add(4);

myList.add(6);

UnknownClass u = new UnknownClass();

int total = u.addValues(myList);

System.out.println(total);

}

}

class UnknownClass

{

public int addValues(List list)

{

list.add(5); //adding object to raw type

Iterator it = list.iterator();

int total = 0;

while (it.hasNext())

{

int i = ((Integer)it.next());

total += i;

}

return total;

}

}

Here Compiler will generate warning message because the unsafe object is inserting the value 5 to safe object.

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

\*Type Erasure

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

In the above program the compiler will generate warning message because the unsafe List Object is inserting the Integer object 5 so, the type safe Integer object is getting value 5 from unsafe type so there is a problem to the caller method.

By writing ArrayList<Integer> actually JVM does not have any idea that our ArrayList was suppose to hold only Integers.

All the type safe generics information does not exist at runtime. All our generic code is Strictly for compiler.

There is a process done by java compiler called "Type erasure" in which the java compiler converts generic version to non-generic type.

List<Integer> myList = new ArrayList<Integer>();

At the compilation time it is fine but at runtime for JVM the code becomes

List myList = new ArrayList();

Note :- GENERIC IS STRICTLY COMPILE TIME PROTECTION.

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

import java.util.\*;

public class TypeErasure

{

public static void main(String[] args)

{

}

public static void m1(List<Integer> list)

{

}

public static void m1(List<String> list)

{

}

}

Note : The above code will not compile due to type erasure, Java compiler will remove all the type parameter so at runtime JVM does not have any information.

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

Polymorphism with Array :

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

//Polymorphism with array

import java.util.\*;

abstract class Animal

{

public abstract void checkup();

}

class Dog extends Animal

{

@Override

public void checkup()

{

System.out.println("Dog checkup");

}

}

class Cat extends Animal

{

@Override

public void checkup()

{

System.out.println("Cat checkup");

}

}

class Bird extends Animal

{

@Override

public void checkup()

{

System.out.println("Bird checkup");

}

}

public class Test8

{

public static void checkAnimals(Animal ...animals)

{

for(Animal animal : animals)

{

animal.checkup();

}

}

public static void main(String[] args)

{

Dog []dogs={new Dog(), new Dog()};

Cat []cats={new Cat(), new Cat(), new Cat()};

Bird []birds = {new Bird(), new Bird()};

checkAnimals(dogs);

checkAnimals(cats);

checkAnimals(birds);

}

}

Note :-From the above program it is clear that polymorphism(Upcasting) concept works with array.

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

import java.util.\*;

abstract class Animal

{

public abstract void checkup();

}

class Dog extends Animal

{

@Override

public void checkup()

{

System.out.println("Dog checkup");

}

}

class Cat extends Animal

{

@Override

public void checkup()

{

System.out.println("Cat checkup");

}

}

class Bird extends Animal

{

@Override

public void checkup()

{

System.out.println("Bird checkup");

}

}

public class Test9

{

public void checkAnimals(List<Animal> animals)

{

for(Animal animal : animals)

{

animal.checkup();

}

}

public static void main(String[] args)

{

List<Dog> dogs = new ArrayList<>();

dogs.add(new Dog());

dogs.add(new Dog());

List<Cat> cats = new ArrayList<>();

cats.add(new Cat());

cats.add(new Cat());

List<Bird> birds = new ArrayList<>();

birds.add(new Bird());

birds.add(new Bird());

Test9 t = new Test9();

t.checkAnimals(dogs);

t.checkAnimals(cats);

t.checkAnimals(birds);

}

}

Note :- The above program will generate compilation error.

So from the above program it is clear that polymorphism does not work in the same way for generics as it does with arrays.

Example :

Parent [] arr = new Child[5]; //valid

Object [] arr = new String[5]; //valid

But in generics the same type is not valid

List<Object> list = new ArrayList<Integer>(); //Invalid

List<Parent> mylist = new ArrayList<Child>(); //Invalid

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

import java.util.\*;

public class Test10

{

public static void main(String [] args)

{

/\*ArrayList<Object> al = new ArrayList<String>(); [Compile time ]

ArrayList al = new ArrayList(); [Runtime, Type Erasure]

al.add("Ravi");\*/

Object []obj = new String[3]; //valid with Array

obj[0] = "Ravi";

obj[1] = "hyd";

obj[2] = 90; //java.lang.ArrayStoreException

System.out.println(Arrays.toString(obj));

}

}

Note :- Program will generate java.lang.ArrayStoreException because we are trying to insert 90 (integer value) into String array.

In Array we have an Exception called ArrayStoreException (Which protect us to assign some illegal value in the array at runtime) but the same Exception or such type of exception, is not available with Generics (due to Type Erasure) that is the reason in generics, compiler does not allow upcasting concept.

(It is a strict compile time protection)

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

WildCard (?)

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

With wildcard we have different cases :

<Dog> : We can assign only <Dog> object

<Animal> : We can assign only <Animal> Object

<?> : It is unknown type so many possibilities.

Here in this wildcard we can assign anything so many possibilities that means It can accept anything.

In order to provide some boundation now we have following two flavors :

a) <? extends Animal> :

This is upper bound here we can replace wild-card (?) with any class which

extends Animal but here there is a chance

of wrong collection in the future because

in future Animal can have more sub classes

so, we can't add any element in the Collection.

b) <? super Dog> :

This is called lower bound, Here we can replace wild card (?) with with any class which is super of Dog i.e Animal, Object. Here compiler knows that the only classes which are super of Dog are allowed so adding element in the collection is allowed.

package com.ravi.wild\_card;

import java.util.ArrayList;

class Animal

{

}

class Dog extends Animal

{

}

class Horse extends Animal

{

}

public class WildCardDemo1 {

public static void main(String[] args)

{

ArrayList<? super Dog> list = new ArrayList<Object>();

list.add(new Dog());

ArrayList<? extends Animal> list1 = new ArrayList<Horse>();

list1.add(new Horse()); //error [Not allowed]

}

}

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

import java.util.\*;

class Parent

{

}

class Child extends Parent

{

}

public class Test11

{

public static void main(String [] args)

{

//ArrayList<Parent> lp = new ArrayList<Child>(); //error

ArrayList<?> lp = new ArrayList<Child>();

ArrayList<Parent> lp1 = new ArrayList<Parent>();

ArrayList<Child> lp2 = new ArrayList<>();

System.out.println("Success");

}

}

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

//program on wild-card chracter

import java.util.\*;

class Parent

{

}

class Child extends Parent

{

}

public class Test12

{

public static void main(String [] args)

{

ArrayList<?> lp = new ArrayList<Parent>();

System.out.println("Wild card....");

}

}

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

import java.util.\*;

public class Test13

{

public static void main(String[] args)

{

List<? extends Number> list1 = new ArrayList<Long>();

List<? super String> list2 = new ArrayList<Object>();

List<? super Gamma> list3 = new ArrayList<Alpha>();

List list4 = new ArrayList();

System.out.println("yes");

}

}

class Alpha

{

}

class Beta extends Alpha

{

}

class Gamma extends Beta

{

}

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

class MyClass<T>

{

T obj;

public MyClass(T obj) //Student obj

{

this.obj=obj;

}

T getObj()

{

return this.obj;

}

}

public class Test14

{

public static void main(String[] args)

{

Integer i=12;

MyClass<Integer> mi = new MyClass<>(i);

System.out.println("Integer object stored :"+mi.getObj());

Float f=12.34f;

MyClass<Float> mf = new MyClass<>(f);

System.out.println("Float object stored :"+mf.getObj());

MyClass<String> ms = new MyClass<>("Rahul");

System.out.println("String object stored :"+ms.getObj());

MyClass<Boolean> mb = new MyClass<>(false);

System.out.println("Boolean object stored :"+mb.getObj());

Double d=99.34;

MyClass<Double> md = new MyClass<>(d);

System.out.println("Double object stored :"+md.getObj());

MyClass<Student> std = new MyClass<>(new Student(1,"A"));

System.out.println("Student object stored :"+std.getObj());

}

}

record Student(int id, String name)

{

}

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

package com.ravi.properties\_demo;

//E stands for Element type

class Fruit {

}

class Apple extends Fruit // Fruit is super, Apple is sub class

{

@Override

public String toString() {

return "Apple";

}

}

class Mango extends Fruit {

@Override

public String toString() {

return "Mango";

}

}

class Basket<E> //E is Fruit type

{

private E element;

public Basket(E element) //Fruit element = new Apple();

{

super();

this.element = element;

}

public E getElement()

{

return element;

}

}

public class Test15

{

public static void main(String[] args)

{

Basket<Fruit> basket = new Basket<>(new Apple());

Apple apple = (Apple) basket.getElement();

System.out.println(apple);

basket = new Basket<>(new Mango());

Mango mango = (Mango) basket.getElement();

System.out.println(mango);

}

}

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

//Generic Method

public class Test16

{

public static void main(String[] args)

{

Integer []intArr = {10,20,30,40,50};

printArray(intArr);

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

String []cities = {"Hyderabad", "Banglore", "Mumbai", "Kolkata"};

printArray(cities);

}

public static <T> void printArray(T[] array)

{

for(T element : array )

{

System.out.println(element);

}

}

}

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