Avdanced Java

Collection and Generics

1) Real life Thread implementation (Custom kafka model)
2) Collections
   - ArrayList class
   - LinkedList class
   - List interface
   - HashSet class
   - LinkedHashSet class
   - TreeSet class
   - PriorityQueue class
   - Map interface
   - HashMap class
   - LinkedHashMap class
   - TreeMap class
   - Hashtable class
   - Sorting
   - Comparable interface
   - Comparator interface
   - Properties class in Java
3) Generics
4) Comparable and Comparator

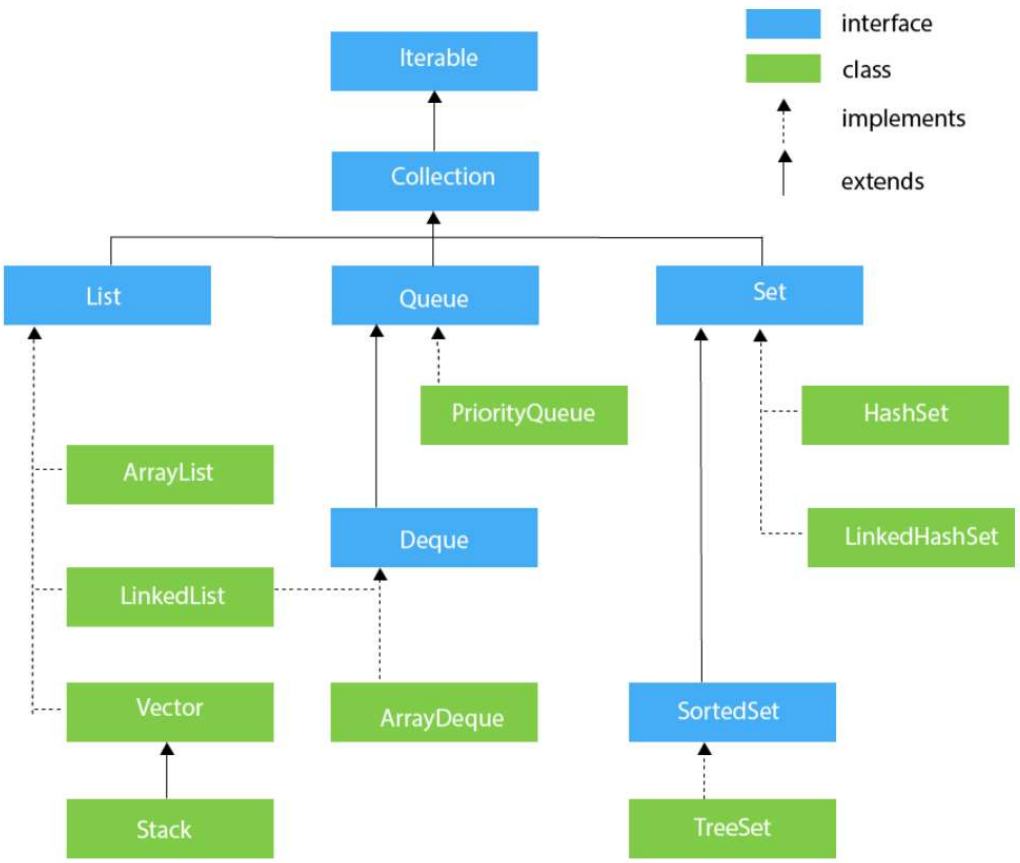
Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collection Hierarchy



Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.Iterator<T> iterator()

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3  = new Vector();
4. List <data-type> list4 = new Stack();
```

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Example

```
import java.util.*;
1.public class TestJavaCollection4{
2.public static void main(String args[]){
3.Stack<String> stack = new Stack<String>();
4.stack.push("Ayush");
5.stack.push("Garvit");
6.stack.push("Amit");
7.stack.push("Ashish");
8.stack.push("Garima");
9.stack.pop();
10.Iterator<String> itr=stack.iterator();
11.while(itr.hasNext()){
12.System.out.println(itr.next());
13.}
14.}
15.}
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

```
Queue<String> q1 = new PriorityQueue();
1.Queue<String> q2 = new ArrayDeque();
```

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Poll() : fetching the first element from queue and removing it
peek() : `Printing the top element of PriorityQueue` **without removing it .**

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

```
Deque d = new ArrayDeque();
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

```
Set<data-type> s1 = new HashSet<data-type>();
1.Set<data-type> s2 = new LinkedHashSet<data-type>();
2.Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

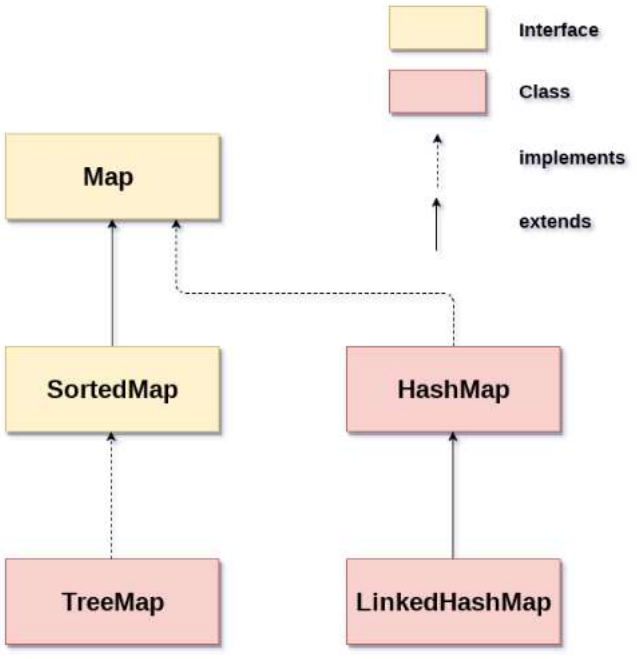
SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.



Map

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.

HashMap

Java **HashMap** class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.
HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface

LinkedHashMap class

LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

TreeMap class

TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

What is difference between HashMap and TreeMap?

HashMap	TreeMap
1) HashMap can contain one null key.	TreeMap cannot contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.

Sorting in Collection

We can sort the elements of:

1.String objects

2.Wrapper class objects

3.User-defined class objects

Collections class provides static methods for sorting the elements of a collection. If collection elements are of a Set type, we can use TreeSet. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): is used to sort the elements of List. List elements must be of the Comparable type.

Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns.

Example

```
class Student implements Comparable<Student>{

1.int rollno;
2.String name;
3.int age;
4.Student(int rollno,String name,int age){
5.this.rollno=rollno;
6.this.name=name;
7.this.age=age;
8.}
9.
10.public int compareTo(Student st){
11.if(age==st.age)
12.return 0;
13.else if(age>st.age)
14.return 1;
15.else
16.return -1;
17.}
18.}
```

Comparator interface

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified

	object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Difference between Comparable and Comparator

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Generics

Generics means parameterized types. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Example

```
import java.util.*;
1.class TestGenerics2{
2.public static void main(String args[]){
3.Map<Integer,String> map=new HashMap<Integer,String>();
4.map.put(1,"vijay");
5.map.put(4,"umesh");
6.map.put(2,"ankit");
7.
8.//Now use Map.Entry for Set and Iterator
```

```
9.Set<Map.Entry<Integer,String>> set=map.entrySet();
10.
11.Iterator<Map.Entry<Integer,String>> itr=set.iterator();
12.while(itr.hasNext()){
13.Map.Entry e=itr.next();//no need to typecast
14.System.out.println(e.getKey()+" "+e.getValue());
15.}
16.
17.}}
```

Generic Class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Example

```
class MyGen<T>{
1.T obj;
2.void add(T obj){this.obj=obj;}
3.T get(){return obj;}
4.}
```

Hashmap Internal Working

1) First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.

2) Then on next step, a hash value is calculated using key's hash code by calling its hashCode() method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written hashCode() functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function, and passed the object's hash code to this hash() function to bring hash value in range of array index size.

3) Now indexFor(hash, table.length) function is called to calculate exact index position for storing the Entry object.

4) Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [called bucket]. Answer is LinkedList. If you remember, Entry class had an attribute "next". This attribute always points to next object in chain. This is exactly the behavior of LinkedList.
So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.
If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.
What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.
In this way, HashMap ensure the uniqueness of keys.

https://www.geeksforgeeks.org/internal-working-of-hashmap-java/