**Abstract Class**

Partially Implemented class which may or may not contain the implementation of the object

* If abstract method is defined in a class, then class should be abstract
* Abstract class can not be instantiated
* If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

**Interfaces**

It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Java 8 changes related to interface

**Difference Between Abstract class and Interface**

| **Abstract class** | **Interface** |
| --- | --- |
| 1) Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. Since Java 8, it can have default and static methods also. |
| 2) Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| 4) Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| 5) The abstract keyword is used to declare abstract class. | The interface keyword is used to declare interface. |
| 6) An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
| 7) An abstract class can be extended using keyword "extends". | An interface can be implemented using keyword "implements". |
| 8) A Java abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |

**Generics Basic:** Class, Method

**Collection Framework**

It provided an architecture to perform operations on the collection of objects like add, update, search remove etc.

**Iterable Interface**

Parent interface of collection interface, it has only one method which is iterator();

**Mostly Used Methods of Collection framework**

1. **Public boolean add(E e)**It is used to insert an element in this collection.
2. **Public boolean addAll(Collection<? extends E> c) -** add Collection to another collection
3. **Public boolean remove(Object object) -** It is used to delete an element from collection
4. **Public int size()** - It returns the size of the collection
5. **Public void clear(**) - It removed all the elements from the collection
6. **Public boolean contains(Object element)** - It is used to search an element
7. **Public boolean isEmpty()** - It checks if the collection is empty
8. **Public Iterator iterator()** - It return an iterator()

**Iteration on Collection Object**

1. **By Using iterator class**

It has 3 methods, hasNext(), next(), remove()

1. **By using foreach loop**

**List Interface**

**ArrayList:**-> It used dynamic array to store the elements

-> It preserves the insertion order

-> It is non synchronized

-> Manipulation is little bit slower due to shifting of elements

**Methods**: add(), add(index,element), addAll(), clear(), get(index), isEmpty(), remove(index),set(index, element)

**Linked List:**

-> Java LinkedList class maintains insertion order.

-> Java LinkedList class is non synchronized.

-> In Java LinkedList class, manipulation is fast because no shifting needs to occur.

-> Java LinkedList class can be used as a list, stack or queue.

**Methods**: add(), add(index,element), addAll(), clear(), get(index), isEmpty(), remove(index),remove(), remove(Object),set(index, element), addFirst(),addLast(), getFirst(),getLast(),

**Array List vs Linked List Difference**

| **ArrayList** | **LinkedList** |
| --- | --- |
| 1) ArrayList internally uses a dynamic array to store the elements. | LinkedList internally uses a doubly linked list to store the elements. |
| 2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can act as a list only because it implements List only. | LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |
| 4) ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized. |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

**Set Interface**

**HashSet**

-> HashSet stores the elements by using a mechanism called **hashing.**

-> HashSet contains unique elements only

-> HashSet allows null value.

-> HashSet class is non synchronized.

-> It doesn’t maintain the insertion order

-> It is the best approach for search operation

**Methods:** add(E e), clear(), contains(), isEmpty(), remove(), size(), iterator()

**LinkedHashSet**

-> Java LinkedHashSet class contains unique elements only like HashSet

-> Java LinkedHashSet class maintains insertion order.

-> Java LinkedHashSet class provides all optional set operations and permits null elements.

-> Java LinkedHashSet class is non-synchronized.

-> It inherits the HashSet class and implements the Set interface.

**Methods:** Common as HashSet class

**TreeSet**

-> TreeSet class access and retrieval times are quiet fast.

-> TreeSet class doesn't allow null element.

-> TreeSet class is non synchronized.

-> TreeSet class maintains ascending order.

-> TreeSet class contains unique elements only like HashSet.

-> It internally uses self balancing BST

**Queue Interface  
Two Implementing classes:** LinkedList, PriorityQueue, ArrayBlockingQueue

**Methods**: add(), remove(), poll(), peek()

**DeQueue Interface**

ArrayDeque is the implementing class

Child interface of queue interface

Support operations from both end

**Methods**: add(), remove(), poll(), peek(), addFirst(),addLast(), peekFirst(),peekLast()

**Map Interface**

**HashMap**

-> allow null keys and values, only one null key

-> Values can be null

-> maintains no order

**Methods:** put(k,v), putIfAbsent(k,v), putAll(map), remove(Object), containsKey(Object), equals(Object), get(key), getOrDefault(key,defaultValue), isEmpty(), replace(k,v), size(),entrySet(),keySet()

**Iteration Over Map**

Map.Entry Interface, keySet()

**LinkedHashMap**-> Every property is same as hashMap except its insertion order

-> It maintains the insertion order

**Methods:** Same as HashMap

**TreeMap**

-> Every property is same as hashMap except its insertion order

-> It maintains the ascending order,

-> It cannot contain null values

**Methods**: floorKey(key), ceilingKey(key)

**HashTable**

It inherits Dictionary class and implements the Map interface.

-> It doesn't allow null key or value.

-> It is synchronized

**Methods**: Almost same as HashMap

**Difference between HashMap and HashTable**

| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| --- | --- |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code  Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

**Fail Fast and Non Fail Fast Iterator**

FailFast: - throw concurrent Modification exception e.g ArrayList, HashMap

Non Fail Fast: doesn’t throw exception e.g CopyOnWriteArrayList, ConcurrentHashMap

**Difference Between Vector and ArrayList**

| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| --- | --- |
| 2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity. | Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

**Collections class**

**Task: Study comparable and comparator interface**

**Functional Interfaces or Lambda Expression**

What are the functional interfaces

What is the lambda

**Benefits of lambda expression**

* Reduces lines of codes
* More readable, concise and maintainable code

**Important Rules of Lambda Expressions**

* If the body of lambda contains only single line then then curly braces are optional
* Compiler also insert the type of variable passed in arguments, hence type is optional

**Kinds of Functional Interfaces**

1. **Consumer -** Take input but doesn’t return anything
2. **Predicate -** Take input and return boolean value
3. **Function -** Take input and return one output
4. **Supplier -** Take no input still pass one output

**Streams API:**

Perform operation on collection of objects

Don’t change the original data structure

**Operations on Streams:**

**Intermediate Operation**

filter(Predicate ) - Filter the elements

sorted() - sort collection elements

distinct() - removes duplicate

map() - iterate through each and every element

**Terminal Operations:**

collect() - collect the returned value

min() - find min

max() - find max

anyMatch() - match with the predicate operation

allMatch() - check if condition applied to all elements successfully

findAny() - find any elements