Java-Unit-3

16 April 2025 06:03

Hello Welcome to learn java.

Unit 3: Collection Framework

Q1. What is a collection? Explain why generics are used ?

Collections Overview
The Java collection's framework provides a large set of readily usable general-purpose data structures and algorithms.
Collection framework standardizes the way in which group of objects are handled by our program.
These data structures and algorithms can be used with any suitable data type in a type-safe manner; this is achieved through the use of a language feature known as generics.

Interfaces:

Collection:
1. List
2. Set -->SortedSet -->NavigableSet
3. Queue -->Dequeue
4. Map

===
Q2. Differentiate between list, set and map?

List:
List is for containers that store a sequence of elements
You can insert the elements using an index, and retrieve the same element later (so that it maintains the insertion order).
You can store duplicate elements in a List.

Set interface:
List for containers that do not allow duplicate elements.
SortedSet interface:
It maintains the set elements in a sorted order.
NavigableSet interface:
It allows searching the set for the closest matches

Map interface:
It is for containers that map keys to values.
SortedMap interface - the keys are in a sorted order.
NavigableMap interface - allows you to search and return the closest match for given search criteria.
Note : Map hierarchy does not extend the Collection interface

===

Q3.What is list? Explain the list methods with an example program.

List in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface

List Interface Methods

The following are the methods of List Interface in Java -

Sr.No.	Method & Description
1	**void add(int index, Object obj)** Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	**boolean addAll(int index, Collection c)** Inserts all elements of **c** into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	**Object get(int index)** Returns the object stored at the specified index within the invoking collection.
4	**int indexOf(Object obj)** Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.

5	**int lastIndexOf(Object obj)** Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
6	**ListIterator listIterator( )** Returns an iterator to the start of the invoking list.
7	**ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index.
8	**Object remove(int index)** Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
9	**Object set(int index, Object obj)** Assigns obj to the location specified by index within the invoking list.
10	**List subList(int start, int end)** Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

```
import java.util.*;

// The Main class
public class Main {
  public static void main(String[] args) {
    // Creating two lists using List interface
    List < Integer > list1 = new ArrayList < Integer > ();
    List < Integer > list2 = new ArrayList < Integer > ();

    // Adding elements to list 1
    list1.add(0, 10);
    list1.add(1, 20);

    // Printing list 1
    System.out.println("list1 : " + list1);

    // Adding elements to list 2
    list2.add(10);
    list2.add(20);
    list2.add(30);

    // Adding all elements of list 2 in list 1
    list1.addAll(1, list2);

    // Printing list 2
    System.out.println("list2 : " + list2);

    // Removes an element from list 1 (from index 1)
    list1.remove(1);

    // Printing list 1
    System.out.println("list1 after removing an element: " + list1);

    // get() method
    System.out.println("list1 using get() : " + list1.get(2));

    // Replacing element
    list1.set(0, 50);

    // Printing the list 1
    System.out.println("list1 : " + list1);
  }
}
```

==

Q4.
What is set? Explain the set methods with an example program

A **Set** is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.
The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

Set Interface Methods

The methods declared by Set are summarized in the following table −

Sr.No.	Method & Description

1	**add( )** Adds an object to the collection.
2	**clear( )** Removes all objects from the collection.
3	**contains( )** Returns true if a specified object is an element within the collection.
4	**isEmpty( )** Returns true if the collection has no elements.
5	**iterator( )** Returns an Iterator object for the collection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the collection.
7	**size( )** Returns the number of elements in the collection.

```
import java.util.HashSet;
import java.util.Set;

public class SetDemo {

  public static void main(String args[]) {
    int count[] = {34, 22,10,60,30,22};
    Set<Integer> set = new HashSet<>();
    try {
      for(int i = 0; i < 5; i++) {
        set.add(count[i]);
      }
      System.out.println(set);
    }
    catch(Exception e) {}
  }
}
```

===

Q5. What is queue? Explain the queue methods with an example program

Queue Interface

The Queue interface is provided in java.util package and it implements the Collection interface. The queue implements FIFO i.e. First In First Out. This means that the elements entered first are the ones that are deleted first. A queue is generally used to hold elements before processing them. Once an element is processed then it is removed from the queue and next item is picked for processing.

Queue Interface Declaration

```
public interface Queue<E>
  extends Collection<E>
```

Queue Interface Methods

Following is the list of the important queue methods that all the implementation classes of the Queue interface implement –

Sr.No.	Method & Description
1	**boolean add(E e)** This method inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
2	**E element()** This method retrieves, but does not remove, the head of this queue.
3	**boolean offer(E e)** This method inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
4	**E peek()** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
5	**E poll()** This method retrieves and removes the head of this queue, or returns null if this queue is empty.
6	**E remove()** This method retrieves and removes the head of this queue.

Methods Inherited

This interface inherits methods from the following interfaces –

- java.util.Collection
- java.lang.Iterable

Classes that Implement Queue

The following are the classes that implement a Queue to use the functionalities of a Queue -

- LinkedList
- ArrayDeque
- PriorityQueue

Interfaces that Extend Queue

The following are the interfaces that extend the Queue interface -

- Deque
- BlockingQueue
- BlockingDeque

Example Program:

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueDemo {
   public static void main(String[] args) {
      Queue<Integer> q = new LinkedList<>();
      q.add(6);
      q.add(1);
      q.add(8);
      q.add(4);
      q.add(7);
      System.out.println("The queue is: " + q);
      int num1 = q.remove();
      System.out.println("The element deleted from the head is: " + num1);
      System.out.println("The queue after deletion is: " + q);
      int head = q.peek();
      System.out.println("The head of the queue is: " + head);
      int size = q.size();
      System.out.println("The size of the queue is: " + size);
   }
}
```

=

==

Q6. What is map? Explain the map methods with an example program.

Map Interface

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

Map Interface Methods

Sr.No.	Method & Description
1	**void clear( )** Removes all key/value pairs from the invoking map.
2	**boolean containsKey(Object k)** Returns true if the invoking map contains **k** as a key. Otherwise, returns false.
3	**boolean containsValue(Object v)** Returns true if the map contains **v** as a value. Otherwise, returns false.
4	**Set entrySet( )** Returns a Set that contains the entries in the map. The set contains objects of type **Map.Entry**. This method provides a set-view of the invoking map.
5	**boolean equals(Object obj)** Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	**Object get(Object k)** Returns the value associated with the key **k**.

7	**int hashCode( )** Returns the hash code for the invoking map.
8	**boolean isEmpty( )** Returns true if the invoking map is empty. Otherwise, returns false.
9	**Set keySet( )** Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10	**Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	**void putAll(Map m)** Puts all the entries from **m** into this map.
12	**Object remove(Object k)** Removes the entry whose key equals **k**.
13	**int size( )** Returns the number of key/value pairs in the map.
14	**Collection values( )** Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Classes that Implement Map

The following are the classes that implement a Map to use the functionalities of a Map -

- HashMap
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap

Interfaces that Extend Map

The following are the interfaces that extend the Map interface -

- SortedMap
- NavigableMap
- ConcurrentMap

Examples of Map Interface

Example 1

Map has its implementation in various classes like HashMap. Following is an example to explain map functionality –

```
import java.util.HashMap;
import java.util.Map;
public class CollectionsDemo {

   public static void main(String[] args) {
      Map<String, String> m1 = new HashMap<>();
      m1.put("Zara", "8");
      m1.put("Mahnaz", "31");
      m1.put("Ayan", "12");
      m1.put("Daisy", "14");

      System.out.println();
      System.out.println(" Map Elements");
      System.out.print("\t" + m1);
   }
}
```

===

Q7. Differentiate between array list, linked list?

Difference Between ArrayList and LinkedList in Java

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.
However, there are many differences between the ArrayList and LinkedList classes that are given below.

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	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in

memory.	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.

Example of ArrayList and LinkedList in Java

Let's see a simple example where we are using ArrayList and LinkedList both.

Example

```
import java.util.*;
class TestArrayLinked{
 public static void main(String args[]){

  List<String> al=new ArrayList<String>();//creating arraylist
  al.add("Ravi");//adding object in arraylist
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

  List<String> al2=new LinkedList<String>();//creating linkedlist
  al2.add("James");//adding object in linkedlist
  al2.add("Serena");
  al2.add("Swati");
  al2.add("Junaid");

  System.out.println("arraylist: "+al);
  System.out.println("linkedlist: "+al2);
 }
}
```

Output:

```
arraylist: [Ravi,Vijay,Ravi,Ajay]
linkedlist: [James,Serena,Swati,Junaid]
```

==

Q8. Differentiate between Hash set, linked hash set and Tree set?

Difference between HashSet and LinkedHashSet

Below are some key differences between HashSet and LinkedHashSet:

- HashSet is an unordered & unsorted collection of the data set, whereas the LinkedHashSet is an ordered and sorted collection of HashSet.
- HashSet does not provide any method to maintain the insertion order. Comparatively, LinkedHashSet maintains the insertion order of the elements.
- We can not predict the insertion order in HashSet, but we can predict it in LinkedHashSet.
- The LinkedHashSet extends the HashSet, so it uses a hashtable to store the elements. Moreover, it uses a doubly linked list to maintain the insertion order.
- The HashSet and LinkedHashSet both implement the Set interface.
- HashSet is slightly faster than the LinkedHashSet. But both provide almost similar performance,
- Both provide o(1) complicity for inserting, removing, retrieving the object.
- Both the HashSet and LinkedHashSet allows only one null objects.
- The LinkedHashSet requires more memory than the HashSet.
- The HashSet was introduced in Java 2, and the LinkedHashSet was introduced in Java 4.

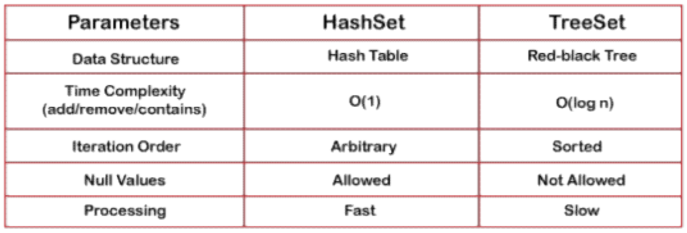
Consider the below tabular differences between HashSet and LinkedHashSet:

Property	HashSet	LinkedHashSet
Data structure	It uses a Hashtable to store the elements.	It uses a HashTable and doubly linked list to store and maintain the insertion order of the elements.
Technique to store the elements	Hashing	Hashing
Insertion Order	It does not provide any insertion order. We can	It provides an insertion order; we can predict the order of

	not predict the order of elements.	elements.
Null elements	It allows only one null element.	It also allows only one null element.
Memory	It requires less memory.	It requires more memory than HashSet.
Performance	It provides slightly faster performance than LinkedHashSet	It provides low performance than HashSet
Synchronized	Non-synchronized	Non-synchronized
Complexity for the insertion, removal, retrieval operations	O (1)	O (1)
Declaration	HashSet obj = new HashSet();	LinkedHashSet obj = new LinkedHashSet();
Extends	AbstractSet class	HashSet class
Implements	Set interface	Set interface
Initial Capacity	16	16
Package	java.util	Java.util

Difference Between HashSet and TreeSet

Parameters	HashSet	TreeSet
Ordering or Sorting	It does not provide a guarantee to sort the data.	It provides a guarantee to sort the data. The sorting depends on the supplied Comparator.
Null Objects	In HashSet, **only an element** can be null.	It does not allow null elements.
Comparison	It uses **hashCode()** or **equals()** method for comparison.	It uses **compare()** or **compareTo()** method for comparison.
Performance	It is **faster** than TreeSet.	It is **slower** in comparison to HashSet.
Implementation	Internally it uses **HashMap** to store its elements.	Internally it uses **TreeMap** to store its elements.
Data Structure	HashSet is backed up by a hash table.	TreeSet is backed up by a Red-black Tree.
Values Stored	It allows only **heterogeneous** value.	It allows only **homogeneous** value.



Parameters	HashSet	TreeSet
Data Structure	Hash Table	Red-black Tree
Time Complexity (add/remove/contains)	O(1)	O(log n)
Iteration Order	Arbitrary	Sorted
Null Values	Allowed	Not Allowed
Processing	Fast	Slow

HashSet Vs TreeSet

==

Q9. Differentiate between Hash map, linked hash map and Tree map?

Difference Between LinkedHashMap and HashMap

The LinkedHashMap is an alternative to HashMap with some additional features. The following are some major differences between LinkedHashMap and HashMap:

- The Major Difference between the HashMap and LinkedHashMap is the ordering of the elements. The LinkedHashMap provides a way to order and trace the elements. Comparatively, the HashMap does not support the ordering of the elements. In LinkedHashMap, if we iterate an element, we will get a key in the order in which the elements were inserted.
- The HashMap and LinkedHashMap both allow only one null key and multiple values.
- The HashMap extends AbstractMap class and implements Map interface, whereas the LinkedHashMap extends HashMap class and implements Map interface.
- Both LinkedHashMap and HashMap are non-synchronized, but they can be synchronized using the Collections.synchronizedMap() method.
- The HashMap uses a bucket to store the elements, which is an index of the array like bucket0 means index[0], bucket1 means index[1], and so on, of the array. Whereas the LinkedHashMap uses the same internal implementation as HashMap but, Apart from that, it also uses a doubly-linked through all of its entries. This linked list is useful for ordering the elements.
- The HashMap requires low memory than LinkedHashMap; because the LinkedHashMap uses the same implementation process as HashMap; additionally, it uses a doubly LinkedList to maintain the order of the elements.
- Both the LinkedHashMap and HashMap provides similar performance.

Comparison Table of LinkedHashMap and HashMap

Consider the below comparison table of HashMap and LinkedHashMap:

Property	HashMap	LinkedHashMap
Order of Iteration	No guarantee order	Insertion order

Implements (Interface)	Map	Map
Null key/values	Only one null key & multiple values	Only one null key & multiple values
Implementation	Buckets	Double-linked buckets
Synchronized	Non-synchronized	non-synchronized
Performance	Fast	Almost Similar to HashMap
Extends	AbstractMap class	HashMap class
Memory	Low Memory	More memory as compared to HashMap.
Thread-safety	Non-thread-safe	Non-thread-safe

The following table describes the differences between HashMap and TreeMap.

Basis	HashMap	TreeMap
Definition	Java **HashMap** is a hashtable based implementation of Map interface.	Java **TreeMap** is a Tree structure-based implementation of Map interface.
Interface Implements	HashMap implements **Map, Cloneable**, and **Serializable** interface.	TreeMap implements **NavigableMap, Cloneable**, and **Serializable** interface.
Null Keys/ Values	HashMap allows a **single** null key and **multiple** null values.	TreeMap does not allow **null** keys but can have **multiple** null values.
Homogeneous/ Heterogeneous	HashMap allows heterogeneous elements because it does not perform sorting on keys.	TreeMap allows homogeneous values as a key because of sorting.
Performance	HashMap is **faster** than TreeMap because it provides constant-time performance that is O(1) for the basic operations like get() and put().	TreeMap is **slow** in comparison to HashMap because it provides the performance of O(log(n)) for most operations like add(), remove() and contains().
Data Structure	The HashMap class uses the **hash table**.	TreeMap internally uses a **Red-Black** tree, which is a self-balancing Binary Search Tree.
Comparison Method	It uses **equals()** method of the **Object** class to compare keys. The equals() method of Map class overrides it.	It uses the **compareTo()** method to compare keys.
Functionality	HashMap class contains only basic functions like **get(), put(), KeySet()**, etc. .	TreeMap class is rich in functionality, because it contains functions like: **tailMap(), firstKey(), lastKey(), pollFirstEntry(), pollLastEntry()**.
Order of elements	HashMap does not maintain any order.	The elements are sorted in **natural order** (ascending).
Uses	The HashMap should be used when we do not require key-value pair in sorted order.	The TreeMap should be used when we require key-value pair in sorted (ascending) order.

==
Q10. What is natural ordering? How do you set it for a user defined objects? Explain with an example program?

==

Q11.What is Comparator. Explain the ordering of user defined objects with different comparators?

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

Java Comparator Interface

In Java, the **Comparator interface** is a part of java.util package and it defines the order of the objects of user-defined classes.

Methods of Comparator Interface

The Comparator interface defines two methods: **compare()** and **equals()**. The **compare()** method, shown here, compares two elements for order –

The compare() Method

```
int compare(Object obj1, Object obj2)
```

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.
By overriding compare(), you can alter the way that objects are ordered. For example, to sort in a reverse order, you can create a

comparator that reverses the outcome of a comparison

The equals() Method

The equals() method, shown here, tests whether an object equals the invoking comparator −

```
boolean equals(Object obj)
```

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

In this example, we're using Comparator interface to sort a custom object Dog based on comparison criterias.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Dog implements Comparator<Dog>, Comparable<Dog> {
   private String name;
   private int age;
   Dog() {
   }

   Dog(String n, int a) {
      name = n;
      age = a;
   }

   public String getDogName() {
      return name;
   }

   public int getDogAge() {
      return age;
   }

   // Overriding the compareTo method
   public int compareTo(Dog d) {
      return (this.name).compareTo(d.name);
   }

   // Overriding the compare method to sort the age
   public int compare(Dog d, Dog d1) {
      return d.age - d1.age;
   }

   @Override
   public String toString() {
      return this.name + "," + this.age;
   }
}

public class ComparatorDemo {

   public static void main(String args[]) {
      // Takes a list o Dog objects
      List<Dog> list = new ArrayList<>();

      list.add(new Dog("Shaggy", 3));
      list.add(new Dog("Lacy", 2));
      list.add(new Dog("Roger", 10));
      list.add(new Dog("Tommy", 4));
      list.add(new Dog("Tammy", 1));

      Collections.sort(list);   // Sorts the array list
      System.out.println("Sorted by name:");
      // printing the sorted list of names
      System.out.print(list);

      // Sorts the array list using comparator
      Collections.sort(list, new Dog());
      System.out.println(" ");

      System.out.println("Sorted by age:");
      // printing the sorted list of ages
      System.out.print(list);
   }
}
```

==

Q12. Explain the Following:

a. Iterator
b. List Iterator
c. Descending Iterator
d. Ennumeration
e. Entry Set

Iterator:

- The Java Iterator is an interface added in the Java Programming language in the Java 1.2 Collection framework. It belongs to java.util package.
- It is one of the Java Cursors that are practiced to traverse the objects of the collection framework.
- The Java Iterator is used to iterate the components of the collection object one by one.
- The Java Iterator is also known as the Universal cursor of Java as it is appropriate for all the classes of the Collection framework.
- The Java Iterator also supports the operations like READ and REMOVE.
- The methods names of the Iterator class are very simple and easy to use compared to the method names of Enumeration Iterator.
- **Forward-only traversal:** An Iterator allows sequential access to elements in a collection in a forward direction.
- **Read and remove operations:** It offers methods like next() to retrieve the next element and remove() to remove the last element returned by the next() method.
- **Limited functionality:** Unlike ListIterator, the Iterator interface does not support bidirectional traversal or modification of elements during iteration

List Iterator:

ListIterator

The ListIterator interface extends the Iterator interface and provides additional functionality specifically designed for lists. Here are the key characteristics of a ListIterator:
Bidirectional traversal: Unlike Iterator, ListIterator allows traversing the list in both forward and backward directions.
Read, remove, replace, and add operations: ListIterator supports all the operations of Iterator, along with methods like hasPrevious(), previous(), set(), and add(). These methods enable modifications to the list during iteration.

Iterator Vs. ListIterator

Feature	Iterator	ListIterator
Interface	Iterator is a more general-purpose interface.	ListIterator is a subinterface of Iterator and extends it, providing additional methods specific to lists.
Traversal Direction	Supports forward-only traversal.	Supports both forward and backward traversal.
Access to Elements	Read-only access to elements.	Read and write access to elements.
Methods	• boolean hasNext() • E next() • void remove() (removes last element returned by next()) • Not applicable	• boolean hasNext() • E next() • boolean hasPrevious() • previous() • int nextIndex() • int previousIndex() • void set(E e) (replaces last element returned by next() or previous()) • void add(E e) (inserts e into the list)
Applicability	Suitable for traversing any collection.	Specifically designed for lists (e.g., ArrayList, LinkedList).
Use Cases	When only forward traversal is required.	When both forward and backward traversal, along with element modification, are needed.
Supported Collections	Works with any collection that implements the Iterable interface.	Works with collections that implement the List interface.

Java Deque descendingIterator() Method

The descendingIterator() method of Java Deque Interface returns an iterator for the elements in the specified deque in a reverse sequence. The elements will return in a sequential order from first(head) to the last(tail).

Java Enums (Enumeration)

Java enum (Enumeration) is a data type that is used when we need to represent a fixed set of constants. Unlike C/C++, the enum in Java is more powerful. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters. The Java enum constants are static and final implicitly.

Java provides the **enum** keyword to define the enum data type. We can define an enum either inside the class or outside the class because it is similar to classes

Java IdentityHashMap entrySet() method

The entrySet() method of the Java IdentityHashMap class is used to get a set view of all the key-value pairs present in the calling IdentityHashMap.

===

Q13. Explain Auto boxing and Auto unboxing? With an example program?

Autoboxing and Unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

In method overloading, boxing and unboxing can be performed. There are some rules for method overloading with boxing:

- **Widening beats boxing**
- **Widening beats varargs**
- **Boxing beats varargs**

Simple Example of Autoboxing in java:

```

class BoxingExample1{
  public static void main(String args[]){
    int a=50;
        Integer a2=new Integer(a);//Boxing

        Integer a3=5;//Boxing

        System.out.println(a2+" "+a3);
 }
}

```

Test it Now

Output:50 5

Simple Example of Unboxing in java:

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

```

class UnboxingExample1{
  public static void main(String args[]){
    Integer i=new Integer(50);
        int a=i;

        System.out.println(a);
 }
}

```

Test it Now

Output:

50

==

Q.15 What is legacy class. Explain them?

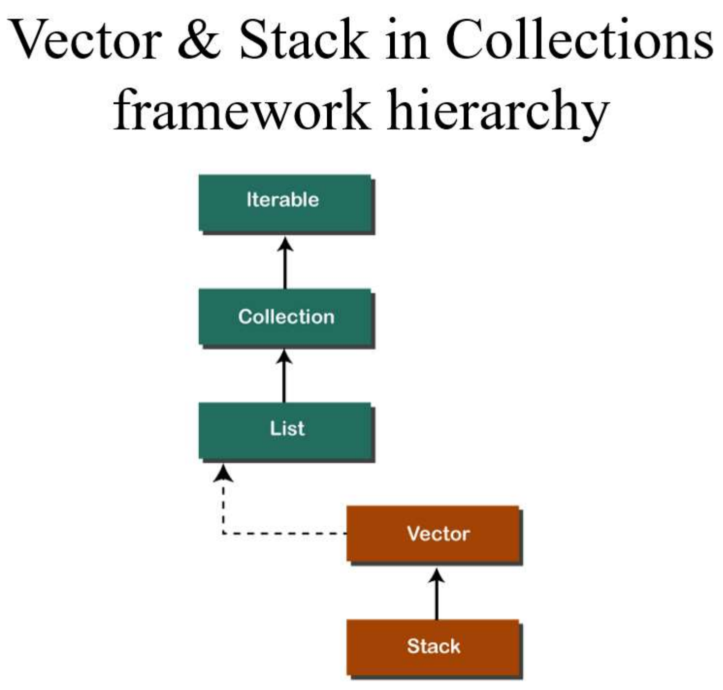
• In the early version of Java, we have several classes and interfaces which allow us to store objects.

• After adding the Collection framework in JSE 1.2, for supporting the collections framework, these classes were re-engineered.

• So, classes and interfaces that formed the collections framework in the older version of Java are known as Legacy classes.

All the legacy classes are synchronized.

The java.util package defines the following legacy classes:
Vector
Stack
Dictionary
HashTable
Properties



Screen clipping taken: 17-04-2025 05:28

Vector Class :

Vector is a special type of ArrayList that defines a dynamic array.
ArrayList is not synchronized while vector is synchronized.
Vector having default initial size of 10.
Constructors of Vectors:
Vector()
Vector(int size)
Vector(int size, int incr)
Vector(Collection c)

Stack Class:

Stack class extends Vector class, which follows the LIFO.
Constructor of Stack:
Stack()
Methods of Stack:

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.

pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Dictionary Class:

- The **Dictionary** class operates much like Map and represents the **key/value** storage repository.
- The **Dictionary class** is an abstract class that stores the data into the key/value pair.

Methods:
The put(K key, V value) method is used to add a key-value pair to the dictionary.
The elements() method is used to get the value representation in the dictionary.
The get(Object key) method is used to get the value mapped with the argumented key in the dictionary.
The isEmpty() method is used to check whether the dictionary is empty or not.
The keys() method is used to get the key representation in the dictionary.
The remove(Object key) method removes the data from the dictionary.
The size() method is used to get the size of the dictionary.

Hashtable class

The Hashtable class is similar to HashMap. It also contains the data into key/value pairs.
HashTable having default size 11.
Hashtable constructors:
1) Hashtable()
2) Hashtable(int size)
3) Hashtable(int size, float fillratio)
    The fillratio must be between 0.0 and 1.0.
4) Hashtable(Map< ? extends K, ? extends V> m)

Properties Class:

Properties class extends Hashtable class to maintain the list of values.
The list has both the key and the value of type string.
Constructors of Properties Class:

Method	Description
Properties()	It creates an empty property list with no default values.
Properties(Properties defaults)	It creates an empty property list with the specified defaults.

==

Q16. Explain the purpose of scanner class and its methods?

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.
The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.
The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.
The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as
nextInt(),
nextByte(),

nextShort(),
next(),
nextLine(),
nextDouble(),
nextFloat(),
nextBoolean(), etc.
To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Java Scanner Class Methods

The following are the list of Scanner methods:

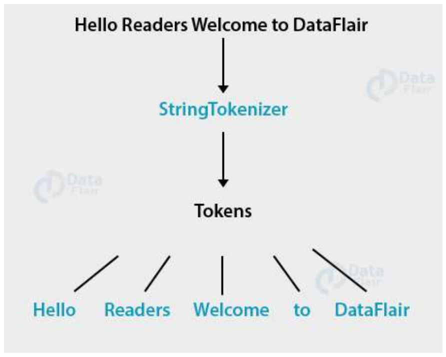
SN	Modifier & Type	Method	Description
1)	void	close()	It is used to close this scanner.
2)	pattern	delimiter()	It is used to get the Pattern which the Scanner class is currently using to match delimiters.
3)	Stream<MatchResult>	findAll()	It is used to find a stream of match results that match the provided pattern string.
4)	String	findInLine()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string	findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean	hasNext()	It returns true if this scanner has another token in its input.
7)	boolean	hasNextBigDecimal()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
8)	boolean	hasNextBigInteger()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
9)	boolean	hasNextBoolean()	It is used to check if the next token in this scanner's input can be interpreted as a Boolean using the nextBoolean() method or not.
10)	boolean	hasNextByte()	It is used to check if the next token in this scanner's input can be interpreted as a Byte using the nextBigDecimal() method or not.
11)	boolean	hasNextDouble()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextByte() method or not.
12)	boolean	hasNextFloat()	It is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
13)	boolean	hasNextInt()	It is used to check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
14)	boolean	hasNextLine()	It is used to check if there is another line in the input of this scanner or not.
15)	boolean	hasNextLong()	It is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong() method or not.
16)	boolean	hasNextShort()	It is used to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.
17)	IOException	ioException()	It is used to get the IOException last thrown by this Scanner's readable.
18)	Locale	locale()	It is used to get a Locale of the Scanner class.
19)	MatchResult	match()	It is used to get the match result of the last scanning operation performed by this scanner.
20)	String	next()	It is used to get the next complete token from the scanner which is in use.
21)	BigDecimal	nextBigDecimal()	It scans the next token of the input as a BigDecimal.
22)	BigInteger	nextBigInteger()	It scans the next token of the input as a BigInteger.
23)	boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
24)	byte	nextByte()	It scans the next token of the input as a byte.
25)	double	nextDouble()	It scans the next token of the input as a double.
26)	float	nextFloat()	It scans the next token of the input as a float.
27)	int	nextInt()	It scans the next token of the input as an Int.
28)	String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
29)	long	nextLong()	It scans the next token of the input as a long.
30)	short	nextShort()	It scans the next token of the input as a short.
31)	int	radix()	It is used to get the default radix of the Scanner use.
32)	void	remove()	It is used when remove operation is not supported by this implementation of Iterator.
33)	Scanner	reset()	It is used to reset the Scanner which is in use.
34)	Scanner	skip()	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>	tokens()	It is used to get a stream of delimiter-separated tokens from the Scanner object which is in use.

36)	String	toString()	It is used to get the string representation of Scanner using.
37)	Scanner	useDelimiter()	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner	useLocale()	It is used to sets this scanner's locale object to the specified locale.
39)	Scanner	useRadix()	It is used to set the default radix of the Scanner which is in use to the specified radix.

==

Q17. what is string tokenzier? Explain its methods with suitable examples

StringTokenizer class is used for creating tokens in Java.
It allows an application to break or split into small parts.
Each split string part is called Token



- There are 3 types of Constructors available in Java StringTokenizer:

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimeter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods:

- Following are 5 types of Methods available in Java StringTokenizer:
  - public boolean hasMoreTokens()
  - public String nextToken()
  - public int countTokens()
  - public Object nextElement()
  - public boolean hasMoreElements()

==

Q18. Explain the following:

a. Bit set
b. Date
c. Calendar
d. Observable Timer
e. Random Access

Bit Set:

BitSet creates an array of bits represented by boolean values.
The size of the array is flexible and can grow to accommodate additional bit as needed.
As it is an array, the index is zero-based and the bit values can be accessed only by non-negative integers as an index .
BitSet class Constructors:
BitSet()
BitSet(int nBits)

Methods:
void set(int index)
void set(int index, boolean v)
void set(int startIndex, int endIndex)
boolean get(int index)
boolean isEmpty( )
int size( )
int length( )
void clear( )
void and(BitSet bitSet)
void or(BitSet bitSet)
void xor(BitSet bitSet)

2. Random class
Random class is used to generate a stream of pseudorandom numbers.
Constructor:
Random(): This creates a new random number generator.

next()	Generates the next pseudorandom number.
nextBoolean()	Returns the next uniformly distributed pseudorandom boolean value from the random number generator's sequence
nextByte()	Generates random bytes and puts them into a specified byte array.
nextDouble()	Returns the next pseudorandom Double value between 0.0 and 1.0 from the random number generator's sequence
nextFloat()	Returns the next uniformly distributed pseudorandom Float value between 0.0 and 1.0 from this random number generator's sequence
nextInt()	Returns a uniformly distributed pseudorandom int value generated from this random number generator's sequence
nextLong()	Returns the next uniformly distributed pseudorandom long value from the random number generator's sequence.
nextBytes(byte[] bytes)	Generates random bytes and places them into a user-supplied byte array

- The class Date represents a specific instant in time, with millisecond precision.

- The Date class of java.util package implements Serializable, Cloneable and Comparable interface.

- It provides constructors and methods to deal with date and time with java.
  long getTime() : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object

- Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc.

- It inherits Object class and implements the Comparable interface.

- **public static Calendar getInstance():** This method is used with calendar object to get the instance of calendar according to current time zone set by java runtime environment