Collections in JAVA

https://www.youtube.com/watch?v=8SoPJYABryI

In Java, the Collection framework provides a set of interfaces, implementations, and algorithms to manipulate and store groups of objects. Collections are used to organize and manage data efficiently. This note aims to provide an easy-to-understand overview of the Collection framework in Java.

Key Concepts:

Interfaces:

The Collection framework includes several interfaces, such as List, Set, and Map, which define common behaviors and operations for collections.
List: An ordered collection that allows duplicate elements.
Examples include ArrayList, LinkedList, Stack and Queue.

Set: A collection that does not allow duplicate elements.
Examples include HashSet and TreeSet.

Map: A key-value pair association that does not allow duplicate keys.
Examples include HashMap and TreeMap.

In the context of the Java Collection framework, interfaces define common behaviors and operations that collections can implement. Here are some important interfaces from the Collection framework along with their methods:

Collection Interface:

boolean add(E element): Adds the specified element to the collection.
boolean remove(Object element): Removes the specified element from the collection.
boolean contains(Object element): Returns true if the collection contains the specified element.
int size(): Returns the number of elements in the collection.
Iterator<E> iterator(): Returns an iterator over the elements in the collection.
List Interface (extends Collection):

void add(int index, E element): Inserts the specified element at the specified position in the list.
E get(int index): Returns the element at the specified position in the list.
E set(int index, E element): Replaces the element at the specified position in the list with the specified element.

int indexOf(Object element): Returns the index of the first occurrence of the specified element in the list.
List<E> subList(int fromIndex, int toIndex): Returns a view of the portion of the list between the specified indexes.
Set Interface (extends Collection):

boolean add(E element): Adds the specified element to the set if it is not already present.
boolean remove(Object element): Removes the specified element from the set.
boolean contains(Object element): Returns true if the set contains the specified element.
int size(): Returns the number of elements in the set.
Iterator<E> iterator(): Returns an iterator over the elements in the set.

Map Interface:

V put(K key, V value): Associates the specified value with the specified key in the map.
V get(Object key): Returns the value to which the specified key is mapped in the map.
boolean containsKey(Object key): Returns true if the map contains the specified key.
boolean containsValue(Object value): Returns true if the map contains the specified value.
Set<K> keySet(): Returns a set of all keys in the map.
Collection<V> values(): Returns a collection of all values in the map.

These are just a few methods provided by the interfaces in the Collection framework. Each interface has additional methods specific to its purpose, allowing for various operations and manipulations of the collections. Implementations of these interfaces, such as ArrayList, LinkedList, HashSet, and HashMap, provide concrete implementations of these methods, enabling you to work with collections in your Java programs efficiently.

Implementations:

Java provides various implementations of the Collection interfaces, each with its unique characteristics and performance trade-offs. Some commonly used implementations include ArrayList, LinkedList, HashSet, TreeSet, HashMap, and TreeMap. It's important to choose the appropriate implementation based on the specific requirements of your program.

Implementations in the Collection framework refer to the concrete classes that implement the various interfaces defined in the framework. These implementations provide different ways to store and manipulate collections of objects. Here are some commonly used implementations in the Collection framework, along with their methods:

ArrayList:
Description: An implementation of the List interface that internally uses an array to store elements.

Key Methods:
add(E element): Appends the specified element to the end of the list.
remove(int index): Removes the element at the specified index from the list.
get(int index): Retrieves the element at the specified index.
set(int index, E element): Replaces the existing element with the provided element at the given index no
size(): Returns the number of elements in the list.
clear(): Clears the elements of the ArrayList

Sample program to demonstrate ArrayList:

```
import java.util.*;
//import java.util.ArrayList;
class JavaCollectionsArray
{
	public static void main(String args[])
	{
		ArrayList<String> l1=new ArrayList<>();
		l1.add("Ram");
		l1.add("Shyam");
		l1.add("Harsh");
		System.out.println(l1);		//display the elements of the ArrayList

		l1.add("Shubham");
		System.out.println(l1);		//display the elements of the ArrayList after adding Shubham to the list at the end

		l1.add(1,"Harshita");
		System.out.println(l1);		//display the elements of the ArrayList after inserting Harshita in the given(1) index position and shifts the rest to the right

		l1.remove(3);
		System.out.println(l1);		//display the elements of the ArrayList after removing the valu from the given(3) index position and shifts left

		l1.set(2, "Harshit");
		System.out.println(l1);		//display the elements of the ArrayList after replacing the value at the given(2) position
		System.out.println(l1.size());
```

```
		System.out.println(l1.get(1));	//display the elements of the ArrayList of the given(1) index no

		l1.clear();
		System.out.println(l1);			//display the elements of the ArrayList after clearing the ArrayList

	}
}
```

LinkedList:

Description: A doubly-linked list implementation of the List interface.

Key Methods:
addFirst(E element): Appends the specified element to the beginning of the list.
addLast(E element): Appends the specified element to the end of the list.
add(E element): Appends the specified element to the end of the list.
remove(int index): Removes the element at the specified index from the list.
get(int index): Retrieves the element at the specified index.
size(): Returns the number of elements in the list.

Sample program to demonstrate LinkedList:

```
import java.util.*;
//import java.util.LinkedList;
class JavaCollectionsLinkedList
{
	public static void main(String args[])
	{
		LinkedList<String> l1=new LinkedList<>();
		l1.add("Gaurav");
		l1.add("Geetansh");
		l1.add("Gauri");
		System.out.println(l1);		//display the elements of the LinkedList

		l1.addLast("Geetima");
		l1.addFirst("Shabnam");
		System.out.println(l1);		//display the elements of the LinkedList after adding Shubham to the list at the end

		l1.add(1,"Gursewak");
		System.out.println(l1);		//display the elements of the LinkedList after inserting Harshita in the given(1) index position and shifts the rest to the right
```

```
		l1.remove(3);
		System.out.println(l1);		//display the elements of the LinkedList after removing the valu from the given(3) index position and shifts left

		l1.removeFirst();
		System.out.println(l1);		//display the elements of the LinkedList after removing the valu from the given(3) index position and shifts left

		l1.removeLast();
		System.out.println(l1);		//display the elements of the LinkedList after removing the valu from the given(3) index position and shifts left

		l1.set(2, "Harshit");
		System.out.println(l1);		//display the elements of the LinkedList after replacing the value at the given(2) position
		System.out.println(l1.size());

		System.out.println(l1.get(1));	//display the elements of the LinkedList of the given(1) index no

		l1.clear();
		System.out.println(l1);			//display the elements of the LinkedList after clearing the ArrayList
	}
}
```

HashSet:

Description: An implementation of the Set interface that uses a hash table for storage.

Key Methods:
add(E element): Adds the specified element to the set if it is not already present.
remove(E element): Removes the specified element from the set.
contains(E element): Returns true if the set contains the specified element.
size(): Returns the number of elements in the set.

Sample program to demonstrate HashSet

```
import java.util.*;
class JavaCollectionsHashSet
{
	public static void main(String args[])
	{
		HashSet<String> set = new HashSet<>();
		   set.add("Apple");
```

```
		set.add("Banana");
		set.add("Orange");
		System.out.println(set); // display the elements of the HashSet

		set.add("Grapes");
		System.out.println(set); // display the elements of the HashSet after adding
Grapes to the set

		set.remove("Banana");
		System.out.println(set); // display the elements of the HashSet after removing
Banana

		System.out.println(set.contains("Apple")); // check if Apple is present in the
HashSet

		System.out.println(set.size()); // display the size of the HashSet

		set.clear();
		System.out.println(set); // display the elements of the HashSet after clearing
the set
}
```

TreeSet:
Description: An implementation of the Set interface that stores elements in a sorted tree structure.

Key Methods:
add(E element): Adds the specified element to the set while maintaining the sorted order.
remove(E element): Removes the specified element from the set.
first(): Returns the first (lowest) element in the set.
size(): Returns the number of elements in the set.

Sample program to demonstrate TreeSet:

```
import java.util.*;
class JavaCollectionsHashSet
{
	public static void main(String[] args)
	{
		TreeSet<String> set = new TreeSet<>();
		set.add("Apple");
		set.add("Banana");
		 set.add("Orange");
```

```
        System.out.println(set); // display the elements of the TreeSet

        set.add("Pineapple");
        System.out.println(set); // display the elements of the TreeSet after adding
Pineapple

        System.out.println(set.contains("Banana")); // check if the TreeSet contains
the element "Banana"

        set.remove("Orange");
        System.out.println(set); // display the elements of the TreeSet after removing
Orange

        System.out.println(set.size()); // get the size of the TreeSet

        System.out.println(set.first()); // get the first element in the TreeSet

        set.clear();
        System.out.println(set); // display the elements of the TreeSet after clearing it
    }
}
```

	HashSet	HashTree
Ordering	No particular order	Sorted order based on natural ordering or custom Comparator
Internal Data Structure	Hash table	Balanced binary search tree (red-black tree)
Performance	O(1) for add, remove, contains (on average)	O(log n) for add, remove, contains

Null Elements	Allows a single null element	Does not allow null elements
Sorting and Searching	Efficient searching	Efficient searching and efficient retrieval in sorted order
Use Cases	Unordered collection	Sorted collection

HashMap:

Description: An implementation of the Map interface that uses hash tables to store key-value pairs.

Key Methods:
put(K key, V value): Associates the specified value with the specified key in the map.
get(K key): Returns the value to which the specified key is mapped, or null if the key is not present.
remove(K key): Removes the mapping for the specified key from the map.
containsKey(K key): Returns true if the map contains the specified key.

Sample program to demonstrate HashMap:

```
import java.util.*;
class JavaCollectionsHashMap
{
	public static void main(String[] args)
	{
		HashMap<Integer, String> map = new HashMap<>();
		    map.put(1, "Apple");
		    map.put(2, "Banana");
		    map.put(3, "Orange");

		    System.out.println(map); // Display the elements of the HashMap

		    map.put(4, "Mango");
		    System.out.println(map); // Display the elements of the HashMap after
	adding Mango to the map
```

```
            map.put(2, "Grapes");
            System.out.println(map); // Display the elements of the HashMap after
    replacing the value at key 2 with Grapes

            map.remove(3);
            System.out.println(map); // Display the elements of the HashMap after
    removing the entry with key 3

            System.out.println(map.get(1)); // Display the value associated with key 1

            System.out.println(map.size()); // Display the size of the HashMap

            map.clear();
            System.out.println(map); // Display the elements of the HashMap after
    clearing it
        }
```

TreeMap:
Description: An implementation of the Map interface that stores key-value pairs in a sorted tree structure.

Key Methods:
put(K key, V value): Associates the specified value with the specified key in the map while maintaining the sorted order.
get(K key): Returns the value to which the specified key is mapped, or null if the key is not present.
remove(K key): Removes the mapping for the specified key from the map.
firstKey(): Returns the first (lowest) key in the map.

Sample program to demonstrate TreeMap:

```
import java.util.*;
class TreeMapExample
{
    public static void main(String[] args)
    {
        TreeMap<Integer, String> treeMap = new TreeMap<>();
            treeMap.put(3, "C");
            treeMap.put(1, "A");
            treeMap.put(4, "D");
```

```
        treeMap.put(2, "B");

        System.out.println(treeMap); // display the elements of the TreeMap

        treeMap.put(5, "E");
        System.out.println(treeMap); // display the elements of the TreeMap after
adding 'E' to the TreeMap

        treeMap.put(2, "Z");
        System.out.println(treeMap); // display the elements of the TreeMap after
replacing the value at key 2 with 'Z'

        treeMap.remove(4);
        System.out.println(treeMap); // display the elements of the TreeMap after
removing the entry with key 4

        System.out.println(treeMap.size()); // display the size of the TreeMap

        System.out.println(treeMap.get(3)); // display the value associated with key
3

        treeMap.clear();
        System.out.println(treeMap); // display the elements of the TreeMap after
clearing the TreeMap
    }

}
```

Features	**HashMap**	**TreeMap**
Implementation	Uses hash table data structure	Uses red-black tree data structure
Ordering	No guarantee on the iteration order	Sorted in ascending order of the keys

Null Keys/Values	Allows null keys and values	Does not allow null keys, values
Performance	Generally faster for most operations	Slower for insertions and deletions
Iteration	Order of iteration is not predictable	Iterates over the keys in sorted order
Memory Overhead	Less memory overhead	More memory overhead

Both HashMap and TreeMap are part of the Java Collections Framework and implement the Map interface, but they have different characteristics regarding ordering, performance, and memory usage. The choice between them depends on the specific requirements of your application.

These are just a few examples of the implementations available in the Collection framework. Each implementation provides different performance characteristics and is suitable for specific use cases. It's important to choose the appropriate implementation based on factors such as the expected number of elements, the need for sorting, and the required operations for your program.

Basic Operations:

Collections support fundamental operations, such as adding, removing, and retrieving elements.

Adding: Elements can be added to a collection using the add() method.
Removing: Elements can be removed from a collection using the remove() method.
Retrieving: Elements can be accessed from a collection using methods like get() or iterators.
Iterators:
Iterators allow you to traverse the elements of a collection sequentially. They provide methods like next(), hasNext(), and remove(). Iterators are commonly used in loops to process elements efficiently.

Algorithms:
The Collection framework includes several utility algorithms that operate on collections, such as sorting, searching, and shuffling. These algorithms provide a convenient way to perform common tasks without writing custom code.

Sorting: The Collections.sort() method can be used to sort elements in a collection.
Searching: The Collections.binarySearch() method allows you to search for an element in a sorted collection.
Shuffling: The Collections.shuffle() method randomly shuffles the elements in a collection.
Generics:
The Collection framework supports generics, which allow you to specify the type of elements a collection can contain. Generics provide compile-time type safety and help prevent runtime errors.
Example: List<String> names = new ArrayList<>();

Conclusion:
The Collection framework in Java provides a powerful set of tools for managing groups of objects. By understanding the key concepts, interfaces, implementations, and basic operations, you can effectively work with collections in your Java programs. It is important to choose the appropriate collection implementation based on your specific requirements and consider the performance trade-offs. Additionally, the provided utility algorithms and support for generics enhance code readability, type safety, and maintainability.

Question 1: Unique Elements
Write a Java program that takes an array of integers as input and returns a collection (Set) containing only the unique elements from the input array. Implement this program using collections in Java.

Question 2: Word Frequency Counter
Write a Java program that takes a string as input and counts the frequency of each word in the string. Return a collection (Map) where the keys are the unique words in the string, and the values are the corresponding frequencies. Implement this program using collections in Java.

Question 3: Student Grade Book
Write a Java program to create a grade book for students. The program should allow adding student names and their corresponding grades. Implement this using a collection (Map) where the student names are keys and the grades are values. The program should also provide functionality to retrieve the grade of a specific student and calculate the average grade of all students.

Question 4: Sorting Names

Write a Java program that takes a list of names as input and sorts them in alphabetical order. Implement this program using collections in Java.

Question 5: Palindrome Checker
Write a Java program that takes a string as input and checks whether it is a palindrome or not. Implement this program using collections in Java to store the characters of the string.

Question 6: Implement a phonebook application using HashMap in Java. The application should allow users to add contacts, search for contacts by name, and display all contacts in alphabetical order.

DFD:

1. Use a HashMap<String, String> to store the contacts, where the key is the contact name and the value is the phone number.
2. Implement methods to add contacts (addContact), search for contacts by name (searchContact), and display all contacts in alphabetical order (displayContacts).
3. When adding a contact, prompt the user for the name and phone number, and store them in the HashMap.
4. When searching for a contact, prompt the user for the name and retrieve the corresponding phone number from the HashMap.
5. To display all contacts in alphabetical order, retrieve all keys from the HashMap, sort them using the Collections.sort() method, and iterate over the sorted keys to display the contacts.

Question 7: Implement a shopping cart using ArrayList in Java. The shopping cart should allow users to add items, remove items, calculate the total price, and display the items in the cart.

DFD

1. Create a class Item with attributes like name, price, and quantity.
2. Use an ArrayList<Item> to store the items in the shopping cart.
3. Implement methods to add items (addItem), remove items (removeItem), calculate the total price (calculateTotalPrice), and display the items in the cart (displayCart).
4. When adding an item, prompt the user for the item details and create an instance of the Item class. Add the item to the ArrayList.
5. When removing an item, prompt the user for the item name and remove the corresponding item from the ArrayList.
6. To calculate the total price, iterate over the items in the ArrayList and sum up the prices.
7. To display the items in the cart, iterate over the ArrayList and print the details of each item.

Question 8: Implement a music playlist using LinkedList in Java. The playlist should allow users to add songs, remove songs, play the next song, and display the current playlist.

DFD:

1. Create a class Song with attributes like title, artist, and duration.
2. Use a LinkedList<Song> to represent the playlist.
3. Implement methods to add songs (addSong), remove songs (removeSong), play the next song (playNextSong), and display the current playlist (displayPlaylist).
4. When adding a song, prompt the user for the song details and create an instance of the Song class. Add the song to the LinkedList.

5. When removing a song, prompt the user for the song title and remove the corresponding song from the LinkedList.
6. To play the next song, remove the first song from the LinkedList and play it.
7. To display the current playlist, iterate over the LinkedList and print the details of each song.