Java Assignment -4

Q1->Discuss the concept of collections in Java. Write a code snippet to sort a List of strings in ascending and descending order using the Collections class.

ANSWER:

Concept of Collections in Java

In Java, **Collections** refer to a framework that provides an architecture for storing and manipulating groups of objects. This framework is known as the **Java Collections Framework (JCF)** and is present in the package java.util.

The Collection Framework provides:

1. Interfaces

These define the abstract data types.
Important interfaces include:

- **Collection**
- **List**
- **Set**
- **Queue**
- **Deque**
- **Map** (not a sub-interface of Collection but part of the framework)

2. Implementing Classes

These are concrete classes that implement the above interfaces:

- **ArrayList, LinkedList** (implement List)
- **HashSet, LinkedHashSet, TreeSet** (implement Set)
- **PriorityQueue, ArrayDeque** (implement Queue/Deque)
- **HashMap, TreeMap, LinkedHashMap** (implement Map)

3. Algorithms (Utility Methods)

The class **Collections** provides static utility methods like:

- sort()
- reverse()

- shufle()

- binarySearch()
- min(), max()
- fill(), copy()

These methods help perform common operations on Collection objects.

Advantages of Java Collections

- Dynamic size (unlike arrays)
- Ready-made data structures (list, set, map)
- Algorithms implemented and optimized
- High performance
- Easy data manipulation

CODE: Sorting List of Strings (Ascending & Descending)

```
import java.util.*;

public class Main2 {

   public static void main(String[] args) {

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

      names.add("Chirag");

      names.add("Sourav");

      names.add("Rohit");

      names.add("Zara");

      // Sort in Ascending Order

      Collections.sort(names);

      System.out.println("Ascending Order: " + names);

      // Sort in Descending Order

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

      System.out.println("Descending Order: " + names);

   }

}
```

Q2->Explain the use of byte streams in Java with examples of FileInputStream and FileOutputStream. Write a code snippet to read a binary file and write its contents to another binary file using these classes.

ANSWER: Byte streams read/write data in raw bytes (0–255).
Used for:

- Images
- Audio/video
- Executables
- Any binary file

CODE:
```
import java.io.*;

public class CopyBinaryFile {
   public static void main(String[] args) {
      try {
         FileInputStream fis = new FileInputStream("input.bin");
         FileOutputStream fos = new FileOutputStream("output.bin");
         int byteData = fis.read();
         while (byteData != -1) {
            fos.write(byteData);
            byteData = fis.read();
         }
         fis.close();
         fos.close();
         System.out.println("File copied successfully!");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Q3->Explain in detail the different types of Java I/O streams. Discuss Byte Streams, Character Streams, Buffered Streams, and their commonly used classes with examples.

ANSWER: Java I/O (Input/Output) is a mechanism that allows programs to read data from various sources (keyboard, files, network) and write data to different destinations (console, files, network).
The I/O system is based on streams, which represent a continuous flow of data.

Java provides two major categories of streams:

1. Byte Streams
2. Character Streams
3. Buffered Streams

1. Byte Streams

Byte streams are used to perform input and output of **8-bit raw binary data**.
These are best suited for handling:

- Images
- Audio/video files
- Executables
- Any non-text binary data

Byte stream classes extend the abstract classes:

- **InputStream**
- **OutputStream**

Common Byte Stream Classes->
a) FileInputStream

Used to read binary data from a file.

Example:
FileInputStream fin = new FileInputStream("data.bin");

int b = fin.read();
b) FileOutputStream

Used to write binary data to a file.

Example:
FileOutputStream fout = new FileOutputStream("copy.bin");

fout.write(65);

Example: Copying a Binary File

```
FileInputStream fin = new FileInputStream("pic.jpg");

FileOutputStream fout = new FileOutputStream("copy.jpg");

int data;

while ((data = fin.read()) != -1) {

   fout.write(data);
```

```
}

fin.close();

fout.close();
```

2. Character Streams

Character streams handle 16-bit Unicode characters.
These streams are used for reading and writing text files (like .txt, .csv, .xml).
Character Stream classes extend the abstract classes:

- Reader
- Writer

Common Character Stream Classes

a) FileReader

Used to read characters from a text file.

Example:

```
FileReader fr = new FileReader("notes.txt");

int ch = fr.read();
```

b) FileWriter

Used to write characters to a text file.

Example:

```
FileWriter fw = new FileWriter("output.txt");
fw.write("Hello Java");
```

Example: Copy Text File

```
FileReader fr = new FileReader("input.txt");

FileWriter fw = new FileWriter("output.txt");

int ch;

while ((ch = fr.read()) != -1) {

   fw.write(ch);

}

fr.close();

fw.close();
```

3. Buffered Streams

Buffered streams improve performance by reducing the number of direct disk accesses.
Instead of reading/writing one byte or character at a time, they use an internal buffer.

Buffered streams wrap other streams.

Buffered Byte Streams

a) BufferedInputStream

Used to read binary data efficiently.

b) BufferedOutputStream

Used to write binary data efficiently.

Example:

```
BufferedInputStream bin = new BufferedInputStream(new FileInputStream("image.png"));
BufferedOutputStream bout = new BufferedOutputStream(new
FileOutputStream("copy.png"));

int data;

while ((data = bin.read()) != -1) {

   bout.write(data);

}

bin.close();

bout.close();
```

Q4->Explain the Java Collections Framework architecture. Describe the core interfaces such as Collection, List, Set, Queue, and Map with examples of their implementations.

ANSWER:

The Java Collections Framework (JCF) is a unified architecture that provides standard ways to store, access, and manipulate groups of objects. It contains interfaces, classes, and algorithms to handle data structures more efficiently than arrays. The main goal of the framework is to reduce programming effort, increase performance, and provide reusable data structures.

1. Architecture of the Collections Framework

The architecture mainly consists of:

A. Core Interfaces

These define the basic behavior of different collection types. They are the foundation of the framework.

B. Implementation Classes

These are concrete classes that actually store the data, such as ArrayList, HashSet, etc.

C. Utility Classes

The Collections class provides commonly used algorithms like sorting, searching, reversing, and shufling.

2. Core Interfaces in JCF

2.1 Collection Interface

It is the root interface for most collection classes (except Map). It represents a group of objects, known as elements.

Child interfaces:

- List
- Set
- Queue

2.2 List Interface

- Stores elements in an ordered manner.
- Allows duplicate values.
- Elements can be accessed using indexes.

Implementations:

1. ArrayList – Dynamic array, fast in searching.
2. LinkedList – Uses nodes, fast in insertion and deletion.
3. Vector – Thread-safe version of ArrayList.

Example:

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

list.add("AI");

list.add("ML");

list.add("AI");
```

2.3 Set Interface

- Does not allow duplicates.
- Mostly unordered collections.

Implementations:

1. HashSet – Unordered and very fast.
2. LinkedHashSet – Maintains insertion order.
3. TreeSet – Elements are stored in sorted order.

Example:

```
Set<Integer> s = new HashSet<>();
s.add(10);
s.add(10); // ignored
```

2.4 Queue Interface

- Follows FIFO (First In First Out) rule.
- Used in scheduling, buffering, and processing tasks.

Implementations:

1. LinkedList – Can behave as a queue.
2. PriorityQueue – Stores elements according to priority.

Example:

```
Queue<String> q = new LinkedList<>();
q.add("Task1");
q.add("Task2");
```

2.5 Map Interface

- Stores data in key–value pairs.
- Each key is unique, values may repeat.
- Not a part of the Collection interface but an important part of JCF.

Implementations:

1. HashMap – Fast lookup, no order.

2. LinkedHashMap – Maintains insertion order.

3. TreeMap – Stores keys in sorted order.

Example:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Java");
map.put(2, "Python");
```

Q5->Write a Java program to demonstrate ArrayList operations such as adding, removing, searching, and updating elements.

CODE:

```
import java.util.*;
public class ArrayListDemo {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();
      list.add("Apple");
      list.add("Banana");
      list.add("Mango");
      list.remove("Banana");
      boolean found = list.contains("Mango");
      System.out.println("Is Mango present? " + found);
      list.set(0, "Orange");
      for (String s : list) {
         System.out.println(s);
      }
   }
}
```

NAME:KRIS

H ROLL

NO.:2401730158

COURSE:B.TECH CSE(AI/ML)