TW 1

Write a Java Program to demonstrate the implementation of stream classes in Java.

Problem statement:

Write a menu driven Java program to read contents of a file and :

print characters on the console – one character at a time

print the entire file

print contents to another file.

Objectives:

1. Binary Data Handling: Facilitate reading and writing operations for binary data files, such as images or multimedia content.

2. Byte-Oriented Approach: Work at the byte level, suitable for raw binary data, providing a byte-oriented I/O solution.

3. Stream-Based Model: Implement a stream-based model for sequential data processing, enhancing efficiency for large file operations.

4. Try-with-Resources: Support try-with-resources statement for automatic resource management, ensuring proper closing of streams.

5. Compatibility: Extend or implement stream-related classes/interfaces, allowing seamless integration with other stream-based classes in Java.

Theory:

`FileInputStream` and `FileOutputStream` are classes in Java that are used for reading from and writing to files, respectively. They are part of the Java I/O (Input/Output) package and are commonly used for handling file operations.

1. FileInputStream:

- `FileInputStream` is used for reading binary data from a file.

- It works with byte-oriented data and is typically used for reading images, audio, video, etc.

- It is a subclass of the `InputStream` class, which is the superclass for all classes representing input streams of bytes.

- To use `FileInputStream`, you create an instance by specifying the path of the file to be read. You can then read bytes from the file using various `read` methods.

```
try (FileInputStream fis = new FileInputStream("example.txt")) {
    int data;
    while ((data = fis.read()) != -1) {
        // Process the byte data
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

2. FileOutputStream:

- `FileOutputStream` is used for writing binary data to a file.

- It is a subclass of the `OutputStream` class, which is the superclass for all classes representing output streams of bytes.

- To use `FileOutputStream`, you create an instance by specifying the path of the file to be written. You can then write bytes to the file using various `write` methods.

```
try (FileOutputStream fos = new FileOutputStream("output.txt")) {
    byte[] data = "Hello, FileOutputStream!".getBytes();
    fos.write(data);
} catch (IOException e) {
    e.printStackTrace();
}
```

- The `try-with-resources` statement is used in the examples to automatically close the streams after use.

Both classes are essential for basic file I/O operations in Java. They provide a convenient way to read and write binary data, which is common when dealing with files that are not text-based, such as images or other non-text documents.

Program:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Scanner;

public class TW1FileIOOperations {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      try {
         int choice;
         do {
            System.out.println("\nMenu:");
            System.out.println("1. Print characters on the console (one character at a time)");
            System.out.println("2. Print the entire file");
            System.out.println("3. Print contents to another file");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();

            InputStream input = new FileInputStream("advJava\\lab\\TW1\\input.txt");
            OutputStream output = new FileOutputStream("advJava\\lab\\TW1\\output.txt");

            switch (choice) {
               case 1:
                  printCharacters(input);
                  break;
               case 2:
```

```
                String data = printEntireFile(input);
                System.out.println(data);
                break;
            case 3:
                printToFile(input, output);
                break;
            case 4:
                System.out.println("Exiting the program.");
                input.close();
                output.close();
                sc.close();
                break;
            default:
                System.out.println("Invalid choice. Please enter a valid option.");
                break;
        }
      } while (choice != 4);

   } catch (IOException e) {
      e.printStackTrace();
   }
}

private static void printCharacters(InputStream input) throws IOException {
   System.out.println("\nFile content is: ");
   int charValue = input.read();
   while (charValue != -1) {
      System.out.println((char) charValue);
      charValue = input.read();
   }
   System.out.println();
```

```
    }

    private static String printEntireFile(InputStream input) throws IOException {
        byte[] array = new byte[100];
        input.read(array);
        System.out.println("\nFile content is: ");
        String data = new String(array);
        return data;
    }

    private static void printToFile(InputStream input, OutputStream output) throws IOException {
        byte[] buffer = new byte[1];
        int bytesRead;
        while ((bytesRead = input.read(buffer)) != -1) {
            output.write(buffer, 0, bytesRead);
        }
        System.out.println("Contents printed to another file successfully.");
    }
}
```

Conclusion:

In summary, FileInputStream and FileOutputStream in Java provide byte-oriented input and output streams for efficient handling of binary data, allowing seamless read and write operations. They play a crucial role in tasks involving raw binary data processing, ensuring proper resource management with try-with-resources and complementing Java's broader stream-based I/O capabilities.

References:

Herbert Schildt and Dale Skrien, “Java Fundamentals A Comprehensive Introduction”, TMH.

Special Indian edition.

Output:

Menu:

1. Print characters on the console (one character at a time)

2. Print the entire file

3. Print contents to another file

4. Exit

Enter your choice: 1

File content is:

Hello I am dumb I took advanace java as a Professional elective Oo god please save me.

Menu:

1. Print characters on the console (one character at a time)

2. Print the entire file

3. Print contents to another file

4. Exit

Enter your choice: 2

File content is:

Hello

I am dumb

I took advanace java as a Professional elective

Oo god please save me.

Menu:

1. Print characters on the console (one character at a time)

2. Print the entire file

3. Print contents to another file

4. Exit

Enter your choice: 3

Contents printed to another file successfully.

Menu:

1. Print characters on the console (one character at a time)

2. Print the entire file

3. Print contents to another file

4. Exit

Enter your choice: 4

Exiting the program.

Input.txt →

Hello

I am dumb

I took advanace java as a Professional elective

Oo god please save me.

Output.txt →

Hello

I am dumb

I took advanace java as a Professional elective

Oo god please save me.

TW 4

Write a Java Program to demonstrate the implementation of File I/O operations in Java.

Problem Statement:

Write a Java program that :

Combines two files to a 3rd.

Displays the properties of the 3rd file – viz., path, absolute path, parent, check whether file exists – if exists whether the file is readable, writeable, a directory and its length.

Objectives:

1. FileInputStream:

   - Read raw binary data from files.

   - Enable byte-level reading for various file types.

2. FileOutputStream:

   - Write raw binary data to files.

   - Useful for tasks involving the manipulation of binary content.

3. File Class:

   - Offer functionalities for querying and handling properties of files and directories.

Theory:

FileInputStream:

- Purpose: Reads raw binary data from files.

- Functionality: Reads bytes from a file one at a time.

- Use Case: Suitable for reading binary files like images, audio, and video.

Example:

```
FileInputStream inputStream = new FileInputStream("example.txt");
int data;
while ((data = inputStream.read()) != -1) {
    // Process each byte of data
}
inputStream.close();
```

FileOutputStream:

- Purpose: Writes raw binary data to files.

- Functionality: Writes bytes to a file one at a time.

- Use Case: Ideal for creating or modifying binary files.

Example:

```
FileOutputStream outputStream = new FileOutputStream("output.txt");
byte[] data = "Hello, World!".getBytes();
outputStream.write(data);
outputStream.close();
```

File Class:

- Purpose: Represents a file or directory path in the system.

- Functionality: Provides methods for file manipulation, such as creating, deleting, and querying properties.

- Use Case: Used to interact with files and directories in the file system.

Example:

```
File file = new File("example.txt");
if (file.exists()) {
    // File exists, perform operations
} else {
    // File does not exist, handle accordingly
}
```

These classes are fundamental for file I/O operations in Java, enabling developers to read and write data to files and perform various file-related tasks.

Program:

```
package advJava.lab.TW2;
import java.io.*;


public class TW2CombineFilesAndDisplayProperties {
   public static void main(String[] args) {
      try {
         // Combine two files into a third file
         combineFiles("advJava\\lab\\TW2\\file1.txt", "advJava\\lab\\TW2\\file2.txt",
               "advJava\\lab\\TW2\\combinedFile.txt");


         // Display properties of the third file
         displayFileProperties("advJava\\lab\\TW2\\combinedFile.txt");
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
   // Combine two files into a third file
   private static void combineFiles(String inputFile1, String inputFile2, String outputFile) throws
IOException {
      try {
         InputStream input1 = new FileInputStream(inputFile1);
         InputStream input2 = new FileInputStream(inputFile2);
         OutputStream output = new FileOutputStream(outputFile);
         // Copy content from file1 to the output file
         byte[] buffer = new byte[1024];
         int bytesRead;
         while ((bytesRead = input1.read(buffer)) != -1) {
            output.write(buffer, 0, bytesRead);
         }
```

```
            // Copy content from file2 to the output file
            while ((bytesRead = input2.read(buffer)) != -1) {
                output.write(buffer, 0, bytesRead);
            }

            System.out.println("Files combined successfully!");

            input1.close();
            input2.close();
            output.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
    // Display properties of a file
    private static void displayFileProperties(String filePath) {
        File file = new File(filePath)
        if (file.exists()) {
            System.out.println("Path: " + file.getPath());
            System.out.println("Absolute Path: " + file.getAbsolutePath());
            System.out.println("Parent: " + file.getParent());
            System.out.println("File exists: " + file.exists());
            System.out.println("Is Readable: " + file.canRead());
            System.out.println("Is Writable: " + file.canWrite());
            System.out.println("Is a Directory: " + file.isDirectory());
            System.out.println("File Length: " + file.length() + " bytes");
        } else {
            System.out.println("File does not exist.");
        }
    }
}
```

Conclusion:

FileInputStream and FileOutputStream facilitate low-level, byte-oriented input and output operations, while the File class serves as a versatile tool for file and directory manipulation. Together, they contribute to effective file handling in Java, supporting various tasks involving binary data and file system interactions.

References:

Herbert Schildt and Dale Skrien, “Java Fundamentals A Comprehensive Introduction”, TMH.

Special Indian edition.

Output:

Files combined successfully!

Path: advJava\lab\TW2\combinedFile.txt

Absolute Path: E:\coding\codes\vscode.Java\advJava\lab\TW2\combinedFile.txt

Parent: advJava\lab\TW2

File exists: true

Is Readable: true

Is Writable: true

Is a Directory: false

File Length: 18 bytes

File1.txt →

Hello

File2.txt→

I am adv Java

combinedFile.txt →

HelloI am adv Java

TW2

Write a Java Program to demonstrate the implementation of Java's Type Wrappers.

Problem statement:

Write a menu driven java program to create an ArrayList of:

1. Integer
2. Float Of specified length.

Write a set of overloaded methods to "add" and / or "remove" elements from the arrays and another set of overloaded methods to perform linear search on the arrays, given the key element.

Create objects to demonstrate the above functionalities.

Objectives:

1. Encapsulate primitive data types as objects.

2. Facilitate object-oriented programming with primitive types.

3. Enable automatic conversion between primitives and objects.

4. Support the use of primitive types in collections.

5. Provide consistent methods for common operations on primitives.

6. Ensure compatibility with APIs expecting objects.

7. Maintain a uniform approach for different primitive types.

Theory:

In Java, type wrappers are classes that encapsulate primitive data types within objects. This allows primitive data types to be treated as objects, and they are particularly useful when working with classes that expect objects rather than primitives. Java provides a set of wrapper classes for each primitive data type. Here are the main type wrappers:

1. Integer (int):

   - Represents a 32-bit signed integer.

   - Provides methods for converting integers to strings and vice versa.

   - Allows operations like comparison and arithmetic on integers.

2. Double (double):

   - Represents a 64-bit double-precision floating-point.

   - Offers methods for converting doubles to strings and parsing strings to doubles.

   - Supports arithmetic operations and comparisons for doubles.

3. Character (char):

   - Represents a 16-bit Unicode character.

   - Provides methods for testing and converting characters.

4. Boolean (boolean):

   - Represents a boolean value (true or false).

   - Provides methods for logical operations and conversions.

5. Byte (byte):

   - Represents an 8-bit signed integer.

   - Supports conversions and operations on bytes.

6. Short (short):

- Represents a 16-bit signed integer.

- Provides methods for conversions and arithmetic on shorts.

7. Long (long):

- Represents a 64-bit signed integer.

- Offers methods for conversions and arithmetic on long integers.

8. Float (float):

- Represents a 32-bit single-precision floating-point.

- Provides methods for conversions and arithmetic on floats.

These wrapper classes are part of the `java.lang` package, and they provide a way to use primitive data types in situations where objects are required, such as in collections, generics, and other class-based scenarios. This process is known as autoboxing (converting a primitive type to its corresponding wrapper class) and unboxing (converting a wrapper class object to its primitive type).

Program:

```
import java.util.ArrayList;
import java.util.Scanner;

class Arrays {
   private ArrayList<Integer> intArrayList;
   private ArrayList<Float> floatArrayList;

   Arrays(int isize, int fsize) {
      intArrayList = new ArrayList<Integer>(isize);
      floatArrayList = new ArrayList<Float>(fsize);
      System.out.println(
            "Array of int and flaot has been initialized -> " + intArrayList.size() + ", " +
floatArrayList.size());
   }

   public ArrayList<Integer> addElements(int element) {
      intArrayList.add(element);
      return intArrayList;
   }

   public ArrayList<Integer> addElements(int position, int element) {
      intArrayList.add(position, element);
      return intArrayList;
   }

   public void removeElements(int element) {
      if (intArrayList.isEmpty())
         System.out.println("Array is empty....");
      else
         if(intArrayList.remove((Integer)element))
```

```
            System.out.println("Element has been removed.");
        else
            System.out.println("Element not found to be removed.");
}

public ArrayList<Float> addElements(float element) {
    floatArrayList.add(element);
    return floatArrayList;
}

public ArrayList<Float> addElements(int position, float element) {
    floatArrayList.add(position, element);
    return floatArrayList;
}

public void removeElements(float element) {
    if (floatArrayList.isEmpty())
        System.out.println("Array is empty....");
    else
        if(floatArrayList.remove(element))
            System.out.println("Element has been removed.");
        else
            System.out.println("Element not found to be removed.");
}

public int LinearSearch(int key) {
    for (int i = 0; i < intArrayList.size(); i++) {
        if (intArrayList.get(i) == key) {
            return i;
        }
    }
```

```
        return -1;
    }

    public int LinearSearch(float key) {
        for (int i = 0; i < floatArrayList.size(); i++) {
            if (floatArrayList.get(i) == key) {
                return i;
            }
        }
        return -1;
    }

    public void printIntArrayList() {
        System.out.println("Int Array List : " + intArrayList);
    }

    public void printFloatArrayList() {
        System.out.println("Int Array List : " + floatArrayList);
    }
}

public class TW3 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Arrays array = new Arrays(10, 10);

        while (true) {
            System.out.println(
                    "1.Add int\n2.Add float\n3.Remove int\n4.Remove float\n5.Linear int Search\n6.Linear
float Search\n7.Print int\n8.Print float\n9.exit.\nYour choice? : ");
            int c = sc.nextInt();
```

```
switch (c) {
    case 1:
        System.out.print("Enter integer to add into list =>");
        int ai = sc.nextInt();
        array.addElements(ai);
        break;

    case 2:
        System.out.print("Enter integer to add into list =>");
        float af = sc.nextFloat();
        array.addElements(af);
        break;

    case 3:
        System.out.print("Enter element to remove element from list =>");
        int ri = sc.nextInt();
        array.removeElements(ri);
        break;

    case 4:
        System.out.print("Enter element to remove element from list =>");
        int rf = sc.nextInt();
        array.removeElements(rf);
        break;

    case 5:
        System.out.print("Enter element to search element from list =>");
        int lsi = sc.nextInt();
        System.out.println("Index of element " + lsi + " : " + array.LinearSearch(lsi));
        break;
```

```
            case 6:
                System.out.print("Enter element to search element from list =>");
                float lsf = sc.nextInt();
                System.out.println("Index of element " + lsf + " : " + array.LinearSearch(lsf));
                break;

            case 7:
                array.printIntArrayList();
                System.out.println();
                break;

            case 8:
                array.printFloatArrayList();
                System.out.println();
                break;

            case 9:
                System.exit(0);
                break;

            default:
                sc.close();
                break;
        }
    }
  }
}
```

Conclusion:

In conclusion, Java's type wrappers enhance the flexibility of the language by allowing primitive data types to be treated as objects. This encapsulation promotes a more consistent and object-oriented approach to programming. The support for autoboxing and unboxing simplifies the conversion between primitives and objects, contributing to cleaner and more readable code. Additionally, compatibility with collections and standardized operations facilitates seamless integration into various aspects of Java programming, ensuring a well-rounded and versatile language feature.

References:

Herbert Schildt and Dale Skrien, “Java Fundamentals A Comprehensive Introduction”, TMH.

Special Indian edition.

Output:

Array of int and float has been initialized -> 0, 0

1.Add int

2.Add float

3.Remove int

4.Remove float

5.Linear int Search

6.Linear float Search

7.Print int

8.Print float

9.exit.

Your choice? : 1

Enter integer to add into list =>10

Your choice? : 1

Enter integer to add into list =>20

Your choice? : 1

Enter integer to add into list =>30

Your choice? : 2

Enter integer to add into list =>1.1

Your choice? : 2

Enter integer to add into list =>2.2

Your choice? : 2

Enter integer to add into list =>3.3

Your choice? : 5

Enter element to search element from list =>20

Index of element 20 : 1

Your choice? : 6

Enter element to search element from list =>3.3

Index of element 3.3 : 2

Your choice? : 7

Int Array List : [10, 20, 30]

Your choice? : 8

Int Array List : [1.1, 2.2, 3.3]

Your choice? : 3

Enter element to remove element from list =>10

Element has been removed.

Your choice? : 4

Enter element to remove element from list =>3.3

Element has been removed.

Your choice? : 7

Int Array List : [20, 30]

Your choice? : 8

Int Array List : [1.1, 2.2]

Your choice? : 9

TW 3

Write a Java Program to demonstrate the implementation of reading and writing binary operations in Java.

Problem Statement:

Write a Java Program to develop a text analysis and visualization tool that reads a large text document containing character data, processes the data using appropriate Java streams and file handling, and provides insightful statistics for Line and Word Count.

- Utilize streams to count the total number of lines and words in the document.
- Display the results to the user.
- Implement a feature to export the analysis results to a new file for further reference.

Objectives:

1. Text Analysis: Develop a Java program to perform text analysis on a large document using Java streams and file handling techniques.

2. Statistical Insights: Utilize streams to efficiently calculate and present insightful statistics, including the total number of lines and words in the document.

3. User Interaction: Implement a user-friendly interface to display the analysis results, providing a clear overview of the document's structure and content.

4. Export Feature: Enhance user experience by incorporating a feature to export the analysis results to a new file, allowing users to save and reference the obtained insights conveniently.

Theory:

Text analysis and visualization tools play a pivotal role in extracting meaningful insights from large textual datasets. Java, with its rich set of libraries and features, provides an ideal environment for developing robust applications in this domain. In the context of this program, the focus is on employing Java streams and file handling to create a text analysis tool with a user-friendly interface.

Java Streams for Efficient Data Processing:

Java streams are a powerful addition to the language, offering a functional programming paradigm for processing collections of data. In this program, streams are utilized to navigate through the document's characters, allowing for seamless and efficient computations of line and word counts. Streams provide a concise syntax that simplifies complex data processing tasks, contributing to the overall performance of the text analysis tool.

File Handling for Input and Output Operations:

Java's file handling capabilities are instrumental in reading the input text document and exporting the analysis results. The `java.nio` package provides a versatile set of classes that streamline file I/O operations. By reading the document as a stream of characters, the program ensures optimal memory usage and facilitates the smooth processing of large files. The export feature utilizes file writing operations to save the analysis results, enabling users to store and reference the insights at their convenience.

User Interface Design for Accessibility:

A crucial aspect of the program is its user interface, designed to present the analysis results in a comprehensible manner. Java's Swing or JavaFX libraries can be employed to create a graphical user interface (GUI) that allows users to interact with the tool seamlessly. The GUI not only displays the statistical insights but also provides an intuitive platform for users to initiate actions such as exporting results to a new file.

Export Feature for Enhanced Usability:

The inclusion of an export feature adds a layer of functionality to the text analysis tool. Users can save the obtained insights to a new file, ensuring that the results are accessible beyond the duration of a single session. This feature enhances the tool's usability and makes it a practical solution for users who may need to refer back to the analysis results or share them with others.

Program:

```
import java.io.FileReader;
import java.io.FileWriter;

public class TW5 {
   public static void main(String[] args) throws IOException {
      int space = 1;
      int newline = 1;

      File file = new File("advJava\\lab\\TW5\\input.txt");
      File output = new File("advJava\\lab\\TW5\\output.txt");

      FileReader file1 = new FileReader(file);
      FileWriter file2 = new FileWriter(output);

      int ch = file1.read();
      if (ch == -1) {
         space = 0;
         newline = 0;
      }

      while (ch != -1) {
         if (ch == 32) {
            space += 1;
         } else if (ch == 10) {
            newline += 1;
            space += 1;
         }
         ch = file1.read();
      }
```

```
        String str = new String("No. of words: " + space + "\n" + "No. of Lines: " + newline);
        System.out.println(str);

        file2.write(str);
        file2.flush();

        file1.close();
        file2.close();
    }
}
```

Output:

No. of words: 16

No. of Lines: 3

Input.txt:

Hello guys welcome to my channel.

please like and subasacribe.

I want to react 1 million..

Output.txt

No. of words: 16

No. of Lines: 3

Conclusion:

In conclusion, the text analysis and visualization tool leverage Java's streams and file handling capabilities to create an efficient and user-friendly application. The detailed use of streams ensures optimal data processing, while file handling operations facilitate seamless input and output interactions. The user interface design enhances accessibility, and the export feature adds a valuable dimension to the tool's usability. This program showcases how Java's features can be harnessed to develop a versatile and effective text analysis solution.

References:

Herbert Schildt and Dale Skrien, “Java Fundamentals A Comprehensive Introduction”, TMH.

Special Indian edition.

TW 5

Write a Java Program to demonstrate the implementation of Multithreading.

Problem statement:

Write a multithreaded java program to create a list of numbers and then sort the contents in ascending (say thread 1) and descending (say thread 2)

- since the task being performed by two threads are different
- you will need to have two different implementations of the run() method
- functionality for dynamically creating an array for the specified size, reading the array elements from the user, printing the array, sorting the array contents in ascending/descending order can be included in a single class.
- Definitions for thread classes will then invoke appropriate methods of the above class.
- The main thread will then spawn two child threads to sort the array contents.

Objectives:

1. Multithreading Implementation: Create threads for ascending and descending array sorting.

2. Array Handling: Dynamically create an array for user-specified size.

3. Sorting: Implement sorting methods for ascending and descending orders.

4. Thread Interaction: Ensure proper synchronization to avoid conflicts.

5. User Interaction: Prompt the user for array size and elements.

6. Output Presentation: Clearly display the sorted arrays for both threads.

Theory:

Multithreading is a programming paradigm that enables the concurrent execution of multiple threads within a single program, allowing tasks to run in parallel. In Java, multithreading can be achieved by either extending the `Thread` class or implementing the `Runnable` interface. Each thread represents an independent flow of execution with its own set of instructions, stack, and program counter. Threads share the same process resources, such as memory space, but operate independently, allowing for efficient utilization of available system resources.

The life cycle of a thread in Java involves several states, including new, runnable, blocked, waiting, and terminated. The `start()` method initiates a thread, and the `run()` method contains the code that will be executed concurrently. Synchronization is a critical aspect of multithreading to control access to shared resources and avoid data corruption. In Java, synchronization can be achieved through the `synchronized` keyword or by using the `Lock` interface.

Multithreading is particularly beneficial for tasks that can be divided into smaller, independent subtasks. It enhances program efficiency by leveraging parallelism, enabling multiple threads to execute different parts of the program simultaneously. However, developers must carefully manage synchronization to prevent race conditions and ensure data integrity. In summary, multithreading is a powerful technique that enhances the responsiveness and performance of Java programs by harnessing the capabilities of modern multicore processors.

Program:

```
import java.util.Scanner;

class Array {
   private int arr[];
   private int n;
   private Scanner s;

   Array() {
      System.out.println();
      s = new Scanner(System.in);
      System.out.print("Enter the size of array : ");
      n = s.nextInt();
      arr = new int[n];
      System.out.println("Allocated memory for array of size : " + arr.length);
   }

   Array(int n) {
      this.n = n;
      arr = new int[n];
      System.out.println("Allocated memory for array of size : " + arr.length);
   }

   void readArray() {
      s = new Scanner(System.in);
      System.out.print("Enter elements to sort : ");
      for (int i = 0; i < arr.length; i++) {
         arr[i] = s.nextInt();
      }
   }
```

```
void printArray() {
   System.out.print("Array elements are : ");
   for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
   }
   System.out.println();
}

void sortAsc() {
   int temp;
   for (int i = 0; i < arr.length; i++) {
      for (int j = i + 1; j < arr.length; j++) {
         if (arr[i] > arr[j]) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
   }
}

void sortDesc() {
   int temp;
   for (int i = 0; i < arr.length; i++) {
      for (int j = i + 1; j < arr.length; j++) {
         if (arr[i] < arr[j]) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
```

```
        }
    }
}

class AscThread extends Thread {
    Array a;
    AscThread() {
        a = new Array();
        a.readArray();
        a.printArray();
    }

    @Override
    public void run() {
        a.sortAsc();
        System.out.println("\nSorted Ascendging Array -->");
        a.printArray();
    }
}

class DescThread extends Thread {
    Array a;

    DescThread() {
        a = new Array();
        a.readArray();
        a.printArray();
    }
```

```
    @Override
    public void run() {
        a.sortDesc();
        System.out.println("\nSorted Descending Array -->");
        a.printArray();
    }
}

public class ArraySorting {
    public static void main(String[] args) {
        AscThread at = new AscThread();
        at.start();
        DescThread dt = new DescThread();
        dt.start();
    }
}
```

Output:

```
Enter the size of array : 5
Allocated memory for array of size : 5
Enter elements to sort : 5 4 3 2 1
Array elements are : 5 4 3 2 1
Sorted Ascendging Array -->
Array elements are : 1 2 3 4 5

Enter the size of array : 4
Allocated memory for array of size : 4
Enter elements to sort : 1 2 3 4
Array elements are : 1 2 3 4
Sorted Descending Array -->
Array elements are : 4 3 2 1
```

Conclusion:

In conclusion, the Java program effectively demonstrates the implementation of multithreading for array sorting. By creating separate threads for ascending and descending order sorting, the program showcases parallel execution, enhancing efficiency. Proper synchronization ensures accurate results, and user interaction features for array size and element input provide an interactive experience. The clear presentation of sorted arrays by each thread contributes to the program's effectiveness in showcasing multithreading capabilities for concurrent tasks.

References:

Herbert Schildt and Dale Skrien, “Java Fundamentals A Comprehensive Introduction”, TMH.

Special Indian edition.

TW 6

Write a Java Program to demonstrate the implementation of Synchronization in Java.

Problem Statement:

Write a java program to demonstrate how the standard operations on a bank account can be synchronized.

Objectives:

- Demonstrate Synchronization:

  - Showcase the use of synchronization in Java to manage concurrent access to shared resources.

- Simulate Bank Account Operations:

  - Implement standard bank account operations such as deposit and withdrawal.

- Ensure Thread Safety:

  - Ensure that multiple threads can interact with the bank account without conflicts or data inconsistencies.

- Illustrate Critical Sections:

  - Highlight critical sections within the program that require synchronization for proper execution.

Theory:

The program exemplifies synchronization in Java, a vital concept for managing concurrent access to shared resources, preventing data corruption and race conditions. In Java, synchronization is achieved by using the `synchronized` keyword, which can be applied to methods or code blocks. In this context, the `synchronized` keyword is used for the `deposit`, `withdraw`, and `check_balance` methods in the `BankAcct` class.

- Synchronization in Java:

Synchronization ensures that only one thread can access a shared resource at a time. When a method or block is declared as synchronized, Java guarantees that the method or block is atomic, preventing other threads from entering the same method or block concurrently. This is crucial for scenarios where multiple threads might try to modify shared data simultaneously, leading to unpredictable behavior without synchronization.

- Critical Sections:

The critical sections in the program are the `deposit` and `withdraw` methods of the `BankAcct` class. These methods involve operations on the shared resource (`balance`), and it is imperative to synchronize them to maintain data integrity. Without synchronization, multiple threads might interfere with each other, leading to incorrect results or data corruption.

- Implementation:

- Deposit_thread and Withdraw_thread:

  - These classes represent threads that perform deposit and withdrawal operations on the bank account.

  - They call the synchronized methods `deposit` and `withdraw` in the `BankAcct` class.

Code:

```
package advJava.lab.TW7;


class BankAcct {
   String cust_name;
   int cust_accno;
   static float balance;

   BankAcct(String cust_name, int cust_accno, float balance) {
      this.cust_name = cust_name;
      this.cust_accno = cust_accno;
      this.balance = balance;
   }

   synchronized void deposit(String name, int damt) {
      balance += damt;
      System.out.println(name + " Deposited Rs. " + damt);

      try {
         Thread.sleep(1000);
      } catch (InterruptedException ie) {
         System.out.println(ie);

      }
   }

   synchronized void withdraw(String name, int wamt) {
      if ((wamt <= balance) && (balance - wamt) > 1000) // checks
      {
         balance -= wamt;
         System.out.println("Amount of Rs. " + wamt + " successfully withdrawn by " + name);
```

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("Insufficient balance. Cannot withdraw");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                System.out.println(ie);
            }
        }
    }

    synchronized float check_balance() {
        return (balance);
    }
}// end of BankAccount class

class Deposit_thread extends Thread {
    Thread t;
    BankAcct BA;
    String name;
    int damt;

    Deposit_thread(String name, BankAcct BA, int damt) {
        this.BA = BA;
        this.name = name;
        this.damt = damt;
    }
```

```
    @Override
    public void run() {
        BA.deposit(name, damt);
        System.out.println("Updated balance is : " + BA.check_balance());
    }
} // end of Deposit_thread class

class Withdraw_thread extends Thread {
    // Thread t;
    BankAcct BA;
    String name;
    int wamt;

    Withdraw_thread(String name, BankAcct BA, int wamt) {
        this.BA = BA;
        this.name = name;
        this.wamt = wamt;
        // t.start();
    }

    @Override
    public void run() {
        BA.withdraw(name, wamt);
    }
} // end of Deposit_thread class

public class BankAccount {
    public static void main(String args[]) {
        BankAcct BA = new BankAcct("Rahul", 1000, 1000);
```

```
        Deposit_thread DT1 = new Deposit_thread("DT1", BA, 1000);
        Deposit_thread DT2 = new Deposit_thread("DT2", BA, 1000);
        Deposit_thread DT3 = new Deposit_thread("DT3", BA, 1000);
        Withdraw_thread WT1 = new Withdraw_thread("WT1", BA, 2000);
        Withdraw_thread WT2 = new Withdraw_thread("WT2", BA, 2000);
        Withdraw_thread WT3 = new Withdraw_thread("WT3", BA, 500);

        DT1.start();
        DT2.start();
        DT3.start();
        WT1.start();
        WT2.start();
        WT3.start();

    } // end of main
}// end of class
```

Output:

```
DT1 Deposited Rs. 1000
Insufficient balance. Cannot withdraw
Updated balance is : 2000.0
DT3 Deposited Rs. 1000
Insufficient balance. Cannot withdraw
Updated balance is : 3000.0
Amount of Rs. 500 successfully withdrawn by WT3
DT2 Deposited Rs. 1000
Updated balance is : 3500.0
```

Conclusion:

In conclusion, the program successfully demonstrates the importance of synchronization in a multithreaded environment, especially when dealing with shared resources like a bank account. The synchronization mechanisms employed in the program ensure that the bank account operations are thread-safe, maintaining data consistency even in a concurrent execution environment. The example provides valuable insights into handling critical sections and avoiding potential issues that might arise in a multithreaded scenario without proper synchronization.

References:

Herbert Schildt and Dale Skrien, “Java Fundamentals A Comprehensive Introduction”, TMH.

Special Indian edition.