|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Object Oriented Programming** | **Semester:** | **III** |
| **Date of Performance:** | **06 / 10 / 2023** | **Batch No:** | **A - 3** |
| **Faculty Name:** | **Pragya Gupta** | **Roll No:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **\_\_\_ / 25** |

|  |  |  |
| --- | --- | --- |
| **Writing Program (07)** | **Performance in lab**  **and Viva**  **(05 + 03)** | **Post lab questions, conclusion and completion**  **(03 + 02 + 05)** |
|  |  |  |

**Experiment No: 7**

**Title: I/O Function in Java**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| Learn the different types of I/O function. |

|  |
| --- |
| **COs to be achieved:** |
| **CO3:** Define exceptions and use I/O streams. |

|  |
| --- |
| **Tools used:** |
| 1. [Java IO - javatpoint](https://www.javatpoint.com/java-io) 2. [Java I/O Streams (programiz.com)](https://www.programiz.com/java-programming/io-streams) |

|  |
| --- |
| **Theory:** |
| (Byte stream class, char stream class and different Reading/writing methods)  In Java, there are two types of streams for handling I/O operations: byte streams and character streams. Byte streams are used to perform input and output of 8-bit bytes, whereas character streams are used to perform input and output for 16-bit Unicode.  Here is a breakdown of the concepts and examples for byte stream and character stream classes in Java:  Byte Stream Classes:   1. **InputStream and OutputStream**: These are the abstract classes representing input and output streams of bytes, respectively.   InputStream inputStream = new FileInputStream("input.txt");  OutputStream outputStream = new FileOutputStream("output.txt");   1. **FileInputStream and FileOutputStream**: These classes are used to read from and write to files, respectively, in the form of bytes.   Example for reading:  FileInputStream inputStream = new FileInputStream("input.txt");  int data;  while ((data = inputStream.read()) != -1) {  // Do something with the data  }  inputStream.close();  Example for writing:  FileOutputStream outputStream = new FileOutputStream("output.txt");  String str = "Hello, world!";  byte[] strToBytes = str.getBytes();  outputStream.write(strToBytes);  outputStream.close();  Character Stream Classes:   1. **Reader and Writer**: These are the abstract classes representing input and output streams for characters, respectively.   Example:  Reader reader = new FileReader("input.txt");  Writer writer = new FileWriter("output.txt");   1. **FileReader and FileWriter**: These classes are used to read from and write to files in the form of characters.   Example for reading:  FileReader reader = new FileReader("input.txt");  int data;  while ((data = reader.read()) != -1) {  // Do something with the data (as char)  char dataChar = (char) data;  }  reader.close();  Example for writing:  FileWriter writer = new FileWriter("output.txt");  String str = "Hello, world!";  writer.write(str);  writer.close();  Different Reading/Writing Methods:   1. **Reading and Writing Byte Streams:**   read(): Reads a byte of data from the input stream.  read(byte[] b): Reads some number of bytes from the input stream and stores them into the buffer array b.  write(int b): Writes the specified byte to the output stream.   1. **Reading and Writing Character Streams:**   read(): Reads a single character from the input stream.  read(char[] cbuf): Reads characters into an array.  write(String str): Writes a string to the output stream.  write(char[] cbuf): Writes a portion of an array of characters. |

|  |
| --- |
| **Code:** |
| 1. **Write a program that lists all the files in a directory including the files present in all its subdirectories as well. Get name/path of the directory from the user through standard input. [Hint: Use recursion]**   import java.io.File;  import java.util.\*;  public class exp7\_q1  {      public static void listFiles(String directoryName)      {          File directory = new File(directoryName);          File[] fileList = directory.listFiles();          if (fileList != null) {              for (File file : fileList) {                  if (file.isFile()) {                      System.out.println("File: " + file.getAbsolutePath());                  } else if (file.isDirectory()) {                      System.out.println("Directory: " + file.getAbsolutePath());                      listFiles(file.getAbsolutePath());                  }              }          }      }      public static void main(String[] args)      {          System.out.println("ketaki mahajan / A-3 / 16014022050");          Scanner sc = new Scanner(System.in);          System.out.print("\nenter the directory path: ");          String directoryPath = sc.nextLine();          listFiles(directoryPath);          sc.close();      }  }   1. **Write a program to read the contents of a file byte by byte and copy it into another file. Get names of the files from the user through standard input.**   import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  import java.util.Scanner;  public class exp7\_q2  {      public static void main(String[] args)      {          System.out.println("ketaki mahajan / A-3 / 16014022050");          Scanner sc = new Scanner(System.in);          System.out.print("\nenter source file name: ");          String sourceFileName = sc.nextLine();          System.out.print("enter destination file name: ");          String destinationFileName = sc.nextLine();          try (FileInputStream inputStream = new FileInputStream(sourceFileName);               FileOutputStream outputStream = new FileOutputStream(destinationFileName)) {              int byteRead;              while ((byteRead = inputStream.read()) != -1) {                  outputStream.write(byteRead);              }              System.out.println("file copied successfully");          } catch (IOException e) {              System.out.println("error occurred: " + e.getMessage());              e.printStackTrace();          }          sc.close();      }  }   1. **Write a program that appends data to the file using FileWriter class.**   **[Hint: Use a different constructor of FileWriter class]**  import java.io.FileWriter;  import java.io.IOException;  import java.util.\*;  public class exp7\_q3  {      public static void main(String[] args)      {          System.out.println("ketaki mahajan / A-3 / 16014022050");          Scanner sc = new Scanner(System.in);          System.out.print("\nenter file name: ");          String fileName = sc.nextLine();          try (FileWriter fileWriter = new FileWriter(fileName, true))          {              fileWriter.write("data is appended to  file.\n");              fileWriter.write("more appended data here\n");              System.out.println("data successfully appended to file");          } catch (IOException e) {              System.out.println("an error occurred: " + e.getMessage());              e.printStackTrace();          }      }  } |

|  |
| --- |
| **Output:** |
|  |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. **Explain the difference between FileInputStream and BufferedInputStream. Show an example in support of your answer.**   In Java, FileInputStream and BufferedInputStream are both classes used for input operations, but they serve different purposes and have different characteristics.   * **FileInputStream:**   FileInputStream is used for reading streams of raw bytes from a file.  It reads data byte by byte from the file without buffering.  This class is appropriate when dealing with smaller files or when the data needs to be processed byte by byte.   * **BufferedInputStream:**   BufferedInputStream is used for reading streams of bytes from a file, but it adds functionality for buffering input.  It reads data from a file into an internal buffer and then reads from the buffer, reducing the number of actual disk reads.  This class is suitable for reading large files efficiently as it minimizes the overhead associated with reading data from the disk.  Here's an example that demonstrates the difference between FileInputStream and BufferedInputStream:  import java.io.\*;  public class StreamComparisonExample {  public static void main(String[] args) {  try {  // Example using FileInputStream  FileInputStream fileInputStream = new FileInputStream("example.txt");  int data;  while ((data = fileInputStream.read()) != -1) {  System.out.print((char) data);  }  fileInputStream.close();  // Example using BufferedInputStream  BufferedInputStream bufferedInputStream = new BufferedInputStream(new FileInputStream("example.txt"));  int dataBuffered;  while ((dataBuffered = bufferedInputStream.read()) != -1) {  System.out.print((char) dataBuffered);  }  bufferedInputStream.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  In this example, both FileInputStream and BufferedInputStream are used to read the contents of a file named "example.txt". The FileInputStream reads the file byte by byte without any buffering, while the BufferedInputStream reads the file data into an internal buffer before reading from it, providing better performance for larger files.   1. **How many lines, words, and characters does a file have? Write a program for the same.**   import java.io.BufferedReader;  import java.io.FileReader;  import java.io.IOException;  public class exp7\_plq2  {      public static void main(String[] args)      {          System.out.println("ketaki mahajan / A-3 / 16014022050");          try {              BufferedReader reader = new BufferedReader(new FileReader("C:\\Users\\Ketaki Mahajan\\OneDrive\\Desktop\\copy\_file\_here.txt"));              String line;              int lineCount = 0;              int wordCount = 0;              int charCount = 0;              while ((line = reader.readLine()) != null) {                  lineCount++;                  String[] words = line.split("\\s+");                  wordCount += words.length;                  charCount += line.length();              }              reader.close();              System.out.println("\nnumber of lines: " + lineCount);              System.out.println("number of words: " + wordCount);              System.out.println("number of characters: " + charCount);          } catch (IOException e) {              e.printStackTrace();          }      }  }     1. **Explain in detail all the possible ways of taking inputs from the user.**  * **Scanner Class:** The Scanner class in Java is widely used to read user input. It provides methods like next(), nextInt(), nextLine(), etc., which can read various types of input from the user. The Scanner class is easy to use and is suitable for most simple input needs. * **BufferedReader with InputStreamReader:** Using BufferedReader with InputStreamReader allows reading character-based input from the user. It is more efficient for reading large amounts of text and provides methods like readLine() for reading entire lines at once. * **Command-Line Arguments:** Java allows you to pass arguments directly from the command line when executing a program. These arguments can be accessed within the main method as an array of strings. This method is suitable when you need to provide initial parameters to the program. * **Console Class (not recommended):** The Console class is used for reading sensitive data like passwords from the user. It provides methods like readPassword() for this purpose. However, it's not recommended for general input handling due to its limitations.   Each method has its own use cases, advantages, and limitations. The choice of method depends on the specific requirements of the program, including the type of input needed, the robustness required, and the potential sensitivity of the data being read. |

|  |
| --- |
| **Conclusion:** |
| Through this experiment, we gained a comprehensive understanding of various I/O functions, including byte stream and character stream operations. We explored file reading, writing, and copying processes, solidifying our grasp on handling file-based data within Java. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |