An Internship Report On:

***JAVA PROGRAMMING***

Prepared By**:**

*SHANMUGAPRIYA S  
INTERN ID: JP100022*

Submitted To:

***DATASOFTIXS***

On The Topic of:

***BANKING APPLICATION***

***TEXT FILE ANALYZER***

***TO-DO LIST APPLICATION***

****

**PROJECT – 1**

**QUESTION:**

**Basic Banking Application:**

* Objective: Create a console-based application to simulate simple banking operations.
* Key Features:
  + Allow users to check balance, deposit, and withdraw money.
  + Implement input validation to avoid negative balances.
  + Use classes and objects to represent accounts.
* Optional: Add a feature to transfer money between accounts.

**Objective:**

The primary objective of this project is to develop a console-based application that simulates fundamental banking operations. This application is designed to provide a simple and user-friendly interface where users can perform essential banking tasks such as checking their account balance, depositing funds, and withdrawing money.

**Abstract:**

The Basic Banking Application demonstrates core programming concepts, including object-oriented principles, user input handling, and data validation. Built as a console-based application, it focuses on simplicity and key functionalities.

Key features include:

* **Balance Inquiry:** Users can view their current account balance.
* **Deposit:** Users can add funds with validation to ensure positive amounts.
* **Withdrawal:** Robust validation prevents overdrafts and ensures valid transactions.
* **Transaction History:** A detailed log of deposits, withdrawals, and transfers provides transparency.
* **Money Transfer:** Allows secure transfers between accounts after necessary validations.
* **Error Handling:** Ensures a smooth experience by managing invalid inputs, such as non-numeric values or insufficient funds.

The program uses object-oriented principles by defining a class to represent accounts, encapsulating attributes like account number, holder name, and balance. Methods handle operations such as deposits, withdrawals, transfers, and history retrieval. Input validation and error handling enhance reliability and usability.

**Code:**

package project\_1;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Scanner;

class Account {

    private int accountId;

    private String accountHolder;

    private double balance;

    private static final String CUSTOMER\_CALL\_NUMBER = "11100011";

    public Account(int accountId, String accountHolder, double balance) {

        this.accountId = accountId;

        this.accountHolder = accountHolder;

        this.balance = balance;

    }

    public int getAccountId() {

        return accountId;

    }

    public String getAccountHolder() {

        return accountHolder;

    }

    public double getBalance() {

        return balance;

    }

    public void deposit(double amount) {

        if (amount > 0) {

            balance += amount;

            System.out.println("\u20b9" + amount + " deposited successfully. New Balance: \u20b9" + balance);

        } else {

            System.out.println("Invalid amount! Deposit amount must be positive.");

        }

    }

    public void withdraw(double amount) {

        if (amount > 0 && amount <= balance) {

            balance -= amount;

            System.out.println("\u20b9" + amount + " withdrawn successfully. Remaining Balance: \u20b9" + balance);

        } else if (amount > balance) {

            System.out.println("Insufficient balance! Cannot withdraw \u20b9" + amount);

        } else {

            System.out.println("Invalid amount! Withdrawal amount must be positive.");

        }

    }

    public void transfer(Account toAccount, double amount) {

        if (amount > 0 && amount <= balance) {

            this.balance -= amount;

            toAccount.balance += amount;

            System.out.println("\u20b9" + amount + " transferred successfully to Account ID: " + toAccount.getAccountId());

        } else if (amount > balance) {

            System.out.println("Insufficient balance! Cannot transfer \u20b9" + amount);

        } else {

            System.out.println("Invalid amount! Transfer amount must be positive.");

        }

    }

    @Override

    public String toString() {

        return "Account ID: " + accountId + ", Holder: " + accountHolder + ", Balance: \u20b9" + balance + ", Customer Call Number: " + CUSTOMER\_CALL\_NUMBER;

    }

}

public class project\_1 {

    private static HashMap<Integer, Account> accounts = new HashMap<>();

    private static List<String> transactionHistory = new ArrayList<>();

    private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {

        int choice;

        while (true) {

            System.out.println("\n--- Banking Application ---");

            System.out.println("1. Add Account");

            System.out.println("2. Check Balance");

            System.out.println("3. Deposit Money");

            System.out.println("4. Withdraw Money");

            System.out.println("5. Transfer Money");

            System.out.println("6. History");

            System.out.println("7. Delete Account");

            System.out.println("8. Exit");

            System.out.print("Enter your choice: ");

            try {

                choice = Integer.parseInt(scanner.nextLine());

                switch (choice) {

                    case 1:

                        addAccount();

                        break;

                    case 2:

                        checkBalance();

                        break;

                    case 3:

                        depositMoney();

                        break;

                    case 4:

                        withdrawMoney();

                        break;

                    case 5:

                        transferMoney();

                        break;

                    case 6:

                        showHistory();

                        break;

                    case 7:

                        deleteAccount();

                        break;

                    case 8:

                     System.out.println("FOR FURTHER ENQUIRY CONTACT OUR CUSTOMER CARE TOLL FREE NUMBER \n 1800 1200 1300 or 1800 1200 1400");

                        System.out.println("Thank you for using the banking application. Please come again.");

                        return;

                    default:

                        System.out.println("Invalid choice! Please choose a valid option.");

                }

            } catch (NumberFormatException e) {

                System.out.println("Invalid input! Please enter a valid number.");

            }

        }

    }

    private static void addAccount() {

        try {

            System.out.print("Enter New Account ID: ");

            int accountId = Integer.parseInt(scanner.nextLine());

            if (accounts.containsKey(accountId)) {

                System.out.println("Account ID already exists! Please use a different ID.");

                return;

            }

            System.out.print("Enter Account Holder Name: ");

            String accountHolder = scanner.nextLine();

            System.out.print("Enter Initial Balance: \u20b9");

            double balance = Double.parseDouble(scanner.nextLine());

            accounts.put(accountId, new Account(accountId, accountHolder, balance));

            System.out.println("Account added successfully!");

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter numeric values where required.");

        }

    }

    private static void deleteAccount() {

        try {

            System.out.print("Enter Account ID to delete: ");

            int accountId = Integer.parseInt(scanner.nextLine());

            if (accounts.containsKey(accountId)) {

                accounts.remove(accountId);

                System.out.println("Account with ID " + accountId + " deleted successfully.");

            } else {

                System.out.println("Account with ID " + accountId + " not found.");

            }

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter a valid account ID.");

        }

    }

    private static void checkBalance() {

        try {

            System.out.print("Enter Account ID: ");

            int accountId = Integer.parseInt(scanner.nextLine());

            Account account = getAccountById(accountId);

            if (account != null) {

                System.out.println(account);

            }

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter a valid account ID.");

        }

    }

    private static void depositMoney() {

        try {

            System.out.print("Enter Account ID: ");

            int accountId = Integer.parseInt(scanner.nextLine().trim());

            Account account = getAccountById(accountId);

            if (account != null) {

                System.out.print("Enter amount to deposit: \u20b9");

                double amount = Double.parseDouble(scanner.nextLine());

                account.deposit(amount);

                transactionHistory.add("Deposit: \u20b9" + amount + " into Account ID: " + accountId);

            }

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter numeric values for amount.");

        }

    }

    private static void withdrawMoney() {

        try {

            System.out.print("Enter Account ID: ");

            int accountId = Integer.parseInt(scanner.nextLine().trim());

            Account account = getAccountById(accountId);

            if (account != null) {

                System.out.print("Enter amount to withdraw: \u20b9");

                double amount = Double.parseDouble(scanner.nextLine());

                if (account.getBalance() >= amount) {

                    transactionHistory.add("Withdrawal: \u20b9" + amount + " from Account ID: " + accountId);

                } else {

                    transactionHistory.add("Transaction Failed: \u20b9" + amount + " from Account ID: " + accountId + " Insufficient Balance");

                }

                account.withdraw(amount);

            }

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter numeric values for amount.");

        }

    }

    private static void transferMoney() {

        try {

            System.out.print("Enter your Account ID: ");

            int fromAccountId = Integer.parseInt(scanner.nextLine());

            Account fromAccount = getAccountById(fromAccountId);

            if (fromAccount != null) {

                System.out.print("Enter the Receiver's Account ID: ");

                int toAccountId = Integer.parseInt(scanner.nextLine());

                Account toAccount = getAccountById(toAccountId);

                if (toAccount != null) {

                    System.out.print("Enter amount to transfer: \u20b9");

                    double amount = Double.parseDouble(scanner.nextLine());

                    if (fromAccount.getBalance() >= amount) {

                        transactionHistory.add("Transfer: \u20b9" + amount + " from Account ID: " + fromAccountId + " to Account ID: " + toAccountId);

                    } else {

                        transactionHistory.add("Transaction Failed: \u20b9" + amount + " from Account ID: " + fromAccountId + " Insufficient Balance");

                    }

                    fromAccount.transfer(toAccount, amount);

                }

            }

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter numeric values for account IDs or amount.");

        }

    }

    private static void showHistory() {

        try {

            if (transactionHistory.isEmpty()) {

                System.out.println("No transaction history available!");

            } else {

                System.out.println("\n--- Transaction History ---");

                for (String transaction : transactionHistory) {

                    System.out.println(transaction);

                }

            }

        } catch (NumberFormatException e) {

            System.out.println("Invalid input! Please enter numeric values for account IDs or amount.");

        }

    }

    private static Account getAccountById(int accountId) {

        if (accounts.containsKey(accountId)) {

         return accounts.get(accountId);

        } else {

         System.out.println("Account ID not found");

         return null;

        }

    }

}

**Code Explanation:**

**1. Package Declaration and Imports**

package project\_1;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Scanner;

* **package project\_1;**: Declares that this code belongs to the project\_1 package.
* **Imports**:
  + ArrayList and HashMap are used for dynamic data storage.
  + Scanner is used for reading user input from the console.

**2. Account Class**

**Attributes**

private int accountId;

private String accountHolder;

private double balance;

private static final String CUSTOMER\_CALL\_NUMBER = "11100011";

* **accountId**: Unique identifier for each account.
* **accountHolder**: Stores the name of the account holder.
* **balance**: Maintains the account's current balance.
* **CUSTOMER\_CALL\_NUMBER**: A constant for displaying customer support details.

**Constructor**

public Account(int accountId, String accountHolder, double balance) {

    this.accountId = accountId;

    this.accountHolder = accountHolder;

    this.balance = balance;

}

* Initializes the accountId, accountHolder, and balance attributes when creating a new account.

**Methods**

* **Getters**: Provide access to private attributes like accountId, accountHolder, and balance.
* **deposit(double amount)**:
  + Adds the specified amount to the account balance if positive.
  + Displays a success or error message based on validation.
* **withdraw(double amount)**:
  + Deducts the specified amount from the account if sufficient balance exists.
  + Handles overdraft prevention and invalid input.
* **transfer(Account toAccount, double amount)**:
  + Transfers money from the current account to another account after validation.
* **toString()**:
  + Provides a string representation of the account, including details like accountId, accountHolder, balance, and the customer call number.

**3. Main Class (project\_1)**

**Attributes**

private static HashMap<Integer, Account> accounts = new HashMap<>();

private static List<String> transactionHistory = new ArrayList<>();

private static Scanner scanner = new Scanner(System.in);

* **accounts**: Stores all accounts using accountId as the key and Account objects as the value.
* **transactionHistory**: Logs all transactions for future reference.
* **scanner**: Used to read user inputs for the program.

**main() Method**

public static void main(String[] args) { ... }

* The core loop of the program:
  + Displays a menu with options like adding an account, checking balance, depositing, withdrawing, transferring money, viewing history, deleting an account, and exiting.
  + Reads user input and executes the corresponding functionality via helper methods.

**4. Helper Methods**

**Add Account**

private static void addAccount() { ... }

* Reads account details (accountId, accountHolder, balance) from the user.
* Validates accountId uniqueness.
* Adds the new Account object to the accounts map.

**Delete Account**

private static void deleteAccount() { ... }

* Reads an accountId from the user.
* Removes the account from the accounts map if it exists, else displays an error message.

**Check Balance**

private static void checkBalance() { ... }

* Fetches and displays the details of an account based on the accountId.

**Deposit Money**

private static void depositMoney() { ... }

* Reads an accountId and deposit amount.
* Validates the amount and updates the account's balance.
* Logs the transaction in transactionHistory.

**Withdraw Money**

private static void withdrawMoney() { ... }

* Reads an accountId and withdrawal amount.
* Validates the amount and updates the account's balance if sufficient funds exist.
* Logs successful and failed transactions in transactionHistory.

**Transfer Money**

private static void transferMoney() { ... }

* Reads sender and receiver accountIds and the transfer amount.
* Validates the transaction and updates both accounts.
* Logs the transfer or failure in transactionHistory.

**Show Transaction History**

private static void showHistory() { ... }

* Displays all entries in transactionHistory.

**Get Account by ID**

private static Account getAccountById(int accountId) { ... }

* Searches for and returns an account from the accounts map using the accountId.

**5. Error Handling**

* The program uses try-catch blocks to handle invalid user inputs (e.g., non-numeric values for account IDs or amounts).
* Error messages guide the user to provide correct input.

**6. Exit and Customer Support**

case 8:

    System.out.println("FOR FURTHER ENQUIRY CONTACT OUR CUSTOMER CARE TOLL FREE NUMBER \n 1800 1200 1300 or 1800 1200 1400");

    System.out.println("Thank you for using the banking application. Please come again.");

    return;

* Displays customer support information when the user chooses to exit.

**PROJECT – 4**

**QUESTION :**

**Text File Analyzer :**

* Objective: Create a program to analyze a text file.
* Key Features:
  + Count the number of words, lines, and characters.
  + Identify and display the most frequently used word.
  + Handle file reading exceptions gracefully.
* Optional: Add a feature to export the analysis results to another file.

**Objective:**

The primary objective of this project is to develop a program that analyzes a text file and provides insights into its contents. The application aims to count the number of words, lines, and characters in the file, identify the most frequently used word, and handle file reading exceptions gracefully. Additionally, an optional feature allows users to export the analysis results to another file, enhancing the program's functionality and usability.

**Abstract:**

The Text File Analyzer is a console-based application designed to process and analyze text files. It performs essential tasks such as counting the total number of words, lines, and characters in the file, providing users with a comprehensive overview of its structure. A standout feature of the program is its ability to identify the most frequently used word, achieved through string manipulation and frequency calculation. The analysis normalizes inputs by removing punctuation and converting text to lowercase for accuracy.

The program ensures robustness through effective exception handling, allowing it to manage invalid file paths or other errors seamlessly without disrupting the user experience. An optional feature to export the analysis results to a separate file makes it versatile and practical for users who wish to save or share the results.

By utilizing concepts such as file handling, string manipulation, and modular programming, the Text File Analyzer demonstrates the practical application of core Java skills in solving real-world problems. This project not only emphasizes technical competence but also promotes clean, maintainable, and user-friendly code.

**Code:**

package project\_4;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class project\_4 {

    public static void main(String[] args) {

        @SuppressWarnings("resource")

        Scanner scanner = new Scanner(System.in);

        boolean continueAnalysis = true;

        while (continueAnalysis) {

            try {

                System.out.print("Enter the path of the text file to analyze: ");

                System.out.println("Enter path as \"C:\\Users\\Admin\\Desktop\\example\_file.txt\"");

                String filePath = scanner.nextLine();

                filePath = filePath.strip().replaceAll("^\"|\"$", "");

                File file = new File(filePath);

                if (!file.exists() || !file.isFile()) {

                    System.out.println("Invalid file path. Please provide a valid text file.");

                    continue;

                }

                analyzeFile(file);

            } catch (Exception e) {

                System.out.println("An error occurred: " + e.getMessage());

                System.out.println("Please try again.\n");

            }

            System.out.print("\nWould you like to analyze another file? (yes/no): ");

            String response = scanner.nextLine().trim().toLowerCase();

            if (!response.equals("yes")) {

                continueAnalysis = false;

                System.out.println("Exiting the program. Goodbye!");

            }

        }

    }

    private static void analyzeFile(File file) throws IOException {

        int lineCount = 0, wordCount = 0, charCount = 0;

        Map<String, Integer> wordFrequency = new HashMap<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

            String line;

            while ((line = reader.readLine()) != null) {

                lineCount++;

                charCount += line.length();

                String[] words = line.split("\\s+");

                for (String word : words) {

                    wordCount++;

                    word = word.toLowerCase().replaceAll("[^a-zA-Z0-9]", ""); // Normalize the word

                    wordFrequency.put(word, wordFrequency.getOrDefault(word, 0) + 1);

                }

            }

        }

        String mostFrequentWord = wordFrequency.entrySet()

                .stream()

                .max(Map.Entry.comparingByValue())

                .map(Map.Entry::getKey)

                .orElse("None");

        System.out.println("\n--- File Analysis ---");

        System.out.println("Number of lines: " + lineCount);

        System.out.println("Number of words: " + wordCount);

        System.out.println("Number of characters: " + charCount);

        System.out.println("Most frequently used word: " + mostFrequentWord + " ("

                + wordFrequency.getOrDefault(mostFrequentWord, 0) + " times)");

        exportResults(file.getName(), lineCount, wordCount, charCount, mostFrequentWord, wordFrequency.getOrDefault(mostFrequentWord, 0));

    }

    private static void exportResults(String fileName, int lines, int words, int chars, String mostFrequentWord, int frequency) {

        String outputFileName = "Analysis\_" + fileName + ".txt";

        try (PrintWriter writer = new PrintWriter(new FileWriter(outputFileName))) {

            writer.println("--- File Analysis ---");

            writer.println("Number of lines: " + lines);

            writer.println("Number of words: " + words);

            writer.println("Number of characters: " + chars);

            writer.println("Most frequently used word: " + mostFrequentWord + " (" + frequency + " times)");

            System.out.println("\nAnalysis results exported to: " + outputFileName);

        } catch (IOException e) {

            System.out.println("Error exporting results: " + e.getMessage());

        }

    }

}

**Code Explanation :**

**1. Main Class and Entry Point**

public class project\_4 {

    public static void main(String[] args) {

* **Purpose**: This is the main entry point for the program.
* **Scanner**: A Scanner object is initialized to take user input from the console.

**2. Loop for Repeated Analysis**

boolean continueAnalysis = true;

while (continueAnalysis) {

    try {

* **Loop**: The program continuously allows users to analyze multiple files until they choose to exit.
* **Error Handling**: A try-catch block is used to gracefully handle errors such as invalid file paths or runtime exceptions.

**3. File Path Input**

System.out.print("Enter the path of the text file to analyze: ");

System.out.println("Enter path as \"C:\\Users\\Admin\\Desktop\\example\_file.txt\"");

String filePath = scanner.nextLine();

filePath = filePath.strip().replaceAll("^\"|\"$", "");

* **User Prompt**: Asks the user for the file path.
* **Path Sanitization**: The input path is stripped of leading and trailing spaces and double quotes.

**4. File Validation**

File = new File(filePath);

if (!file.exists() || !file.isFile()) {

    System.out.println("Invalid file path. Please provide a valid text file.");

    continue;

}

* **Validation**: The program checks whether the specified path points to a valid file.
* **Error Message**: Notifies the user if the file is invalid.

**5. File Analysis Method**

analyzeFile(file);

* The analyzeFile method is called to perform the actual analysis on the specified file.

**6. Analyze File**

private static void analyzeFile(File file) throws IOException {

* **Method Signature**: This method processes the file, counts lines, words, characters, and determines the most frequent word.

**File Reading and Analysis**

try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

    String line;

    while ((line = reader.readLine()) != null) {

        lineCount++;

        charCount += line.length();

        String[] words = line.split("\\s+");

        for (String word : words) {

            wordCount++;

            word = word.toLowerCase().replaceAll("[^a-zA-Z0-9]", "");

            wordFrequency.put(word, wordFrequency.getOrDefault(word, 0) + 1);

        }

    }

}

* **BufferedReader**: Reads the file line by line.
* **Counters**:
  + lineCount: Tracks the number of lines.
  + charCount: Adds up the length of each line for the total character count.
  + wordCount: Splits lines into words and increments the counter.
* **Word Normalization**:
  + Converts words to lowercase.
  + Removes special characters and punctuation.
* **Word Frequency**: Uses a HashMap to count occurrences of each word.

**Determine Most Frequent Word**

String mostFrequentWord = wordFrequency.entrySet()

    .stream()

    .max(Map.Entry.comparingByValue())

    .map(Map.Entry::getKey)

    .orElse("None");

* **Stream API**: Finds the word with the highest frequency using max on the map entries.

**Display Results**

System.out.println("\n--- File Analysis ---");

System.out.println("Number of lines: " + lineCount);

System.out.println("Number of words: " + wordCount);

System.out.println("Number of characters: " + charCount);

System.out.println("Most frequently used word: " + mostFrequentWord + " (" + wordFrequency.getOrDefault(mostFrequentWord, 0) + " times)");

* **Output**: Prints the results of the analysis to the console.

**7. Export Results**

exportResults(file.getName(), lineCount, wordCount, charCount, mostFrequentWord, wordFrequency.getOrDefault(mostFrequentWord, 0));

* The exportResults method is called to save the analysis results to a file.

**8. Export Results to File**

private static void exportResults(String fileName, int lines, int words, int chars, String mostFrequentWord, int frequency) {

* **Purpose**: Writes the analysis results to a new file named Analysis\_<OriginalFileName>.txt.

**Write to File**

try (PrintWriter writer = new PrintWriter(new FileWriter(outputFileName))) {

    writer.println("--- File Analysis ---");

    writer.println("Number of lines: " + lines);

    writer.println("Number of words: " + words);

    writer.println("Number of characters: " + chars);

    writer.println("Most frequently used word: " + mostFrequentWord + " (" + frequency + " times)");

    System.out.println("\nAnalysis results exported to: " + outputFileName);

} catch (IOException e) {

    System.out.println("Error exporting results: " + e.getMessage());

}

* **FileWriter/PrintWriter**: Writes the analysis results to the specified output file.
* **Error Handling**: Catches IOException to handle any issues during file writing.

**9. Exit Option**

System.out.print("\nWould you like to analyze another file? (yes/no): ");

String response = scanner.nextLine().trim().toLowerCase();

if (!response.equals("yes")) {

    continueAnalysis = false;

    System.out.println("Exiting the program. Goodbye!");

}

* **Loop Exit**: If the user inputs "no," the program terminates.

**Key Features**

1. **Line, Word, and Character Count**: Basic file analysis statistics.
2. **Most Frequently Used Word**: Highlights the most common word and its frequency.
3. **File Validation**: Ensures only valid files are processed.
4. **Result Export**: Saves analysis results for further reference.
5. **Error Handling**: Manages invalid paths, empty files, and I/O issues gracefully.

**PROJECT – 6**

**QUESTION :**

**To-Do List Application :**

* Objective: Build a simple to-do list manager.
* Key Features:
  + Allow users to add, edit, delete, and mark tasks as complete.
  + Store tasks in memory using an ArrayList.
  + Display the list of pending and completed tasks.
* Optional: Save and load tasks from a text file.

**Abstract**

The **To-Do List Application** is a straightforward yet powerful tool designed to help users manage their daily tasks efficiently. Its primary features include the ability to add, edit, delete, and mark tasks as complete, ensuring users can organize and prioritize their responsibilities effectively. Tasks are stored dynamically in memory using an ArrayList, which allows real-time manipulation of the task list. Additionally, the application categorizes tasks into pending and completed lists, providing a clear view of progress. An optional feature enables users to save tasks to a file and reload them for future use, ensuring data persistence across sessions. Robust exception handling is integrated to handle invalid inputs and file operation errors, enhancing reliability and user satisfaction.

**Overview**

The **To-Do List Application** aims to simplify task management by providing an intuitive and interactive platform for users. It allows users to create new tasks, edit existing ones, delete unwanted tasks, and mark tasks as completed when finished. Tasks are stored in an ArrayList, which ensures efficient memory usage and fast operations. The program displays tasks in two categories: pending and completed, making it easy for users to track their progress.

One of the standout features is the optional functionality to save tasks to a text file and reload them in future sessions, adding long-term utility to the application. File operations are carefully managed with exception handling to avoid crashes or data loss in case of errors. This project demonstrates key Java programming concepts, including collections, file I/O, and error handling, in a practical and user-friendly way. Overall, the application is a versatile tool for organizing daily tasks, with an emphasis on simplicity, reliability, and usability.

**Code :**

package project\_6;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Scanner;

public class project\_6 {

    static class Task {

        String description;

        boolean isCompleted;

        Task(String description, boolean isCompleted) {

            this.description = description;

            this.isCompleted = isCompleted;

        }

        public String toString() {

            return (isCompleted ? "[✔]" : "[  ]") + " " + description;

        }

    }

    public static void main(String[] args) {

        ArrayList<Task> tasks = new ArrayList<>();

        String fileName = "Tasks.txt";

        loadTasksFromFile(tasks, fileName);

        Scanner ip = new Scanner(System.in);

        int choice = -1;

        do {

         System.out.println("\n Task loaded");

            System.out.println("\n\n--- To-Do List Manager ---");

            System.out.println("1. Add Task");

            System.out.println("2. View Tasks");

            System.out.println("3. Mark Task as Completed");

            System.out.println("4. Delete Task");

            System.out.println("5. Exit");

            System.out.print("Enter your choice: ");

            try {

                choice = ip.nextInt();

                ip.nextLine();

            } catch (Exception e) {

                System.out.println("\nInvalid input! Please enter an integer.");

                ip.next();

                continue;

            }

            switch (choice) {

                case 1:

                    System.out.println("\nEnter the task description: ");

                    String desc = ip.nextLine();

                    if (desc.isEmpty()) {

                        System.out.println("Task description cannot be empty. Please try again.");

                    } else {

                        tasks.add(new Task(desc, false));

                        System.out.println("Task added!");

                        saveTasksToFile(tasks, fileName);

                    }

                    break;

                case 2:

                    System.out.println("\nYour Tasks:");

                    if (tasks.isEmpty()) {

                        System.out.println("No tasks recorded.");

                    } else {

                        for (int i = 0; i < tasks.size(); i++) {

                            System.out.println((i + 1) + ". " + tasks.get(i));

                        }

                    }

                    break;

                case 3:

                    System.out.print("\nEnter task number to mark as completed: ");

                    try {

                        int completeIndex = ip.nextInt() - 1;

                        if (completeIndex >= 0 && completeIndex < tasks.size()) {

                            tasks.get(completeIndex).isCompleted = true;

                            System.out.println("Task marked as completed!");

                            saveTasksToFile(tasks, fileName);

                        } else {

                            System.out.println("Invalid task number.");

                        }

                    } catch (Exception e) {

                        System.out.println("Invalid input! Please enter a valid task number.");

                        ip.next();

                    }

                    break;

                case 4:

                    System.out.print("\nEnter task number to delete: ");

                    try {

                        int deleteIndex = ip.nextInt() - 1;

                        if (deleteIndex >= 0 && deleteIndex < tasks.size()) {

                            tasks.remove(deleteIndex);

                            System.out.println("Task deleted!");

                            saveTasksToFile(tasks, fileName);

                        } else {

                            System.out.println("Invalid task number.");

                        }

                    } catch (Exception e) {

                        System.out.println("Invalid input! Please enter a valid task number.");

                        ip.next();

                    }

                    break;

                case 5:

                    System.out.println("\nExiting To-Do List Manager. Goodbye!");

                    break;

                default:

                    System.out.println("\nInvalid choice. Please try again.");

            }

        } while (choice != 5);

        ip.close();

    }

    private static void loadTasksFromFile(ArrayList<Task> tasks, String fileName) {

        File file = new File(fileName);

        if (!file.exists()) {

            return;

        }

        try (BufferedReader br = new BufferedReader(new FileReader(file))) {

            String line;

            while ((line = br.readLine()) != null) {

                String[] parts = line.split(";", 2);

                if (parts.length == 2) {

                    boolean isCompleted = Boolean.parseBoolean(parts[0]);

                    String description = parts[1];

                    tasks.add(new Task(description, isCompleted));

                }

            }

        } catch (IOException e) {

            System.out.println("Error reading tasks from file: " + e.getMessage());

        }

    }

    private static void saveTasksToFile(ArrayList<Task> tasks, String fileName) {

        try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {

            for (Task task : tasks) {

                bw.write(task.isCompleted + ";" + task.description);

                bw.newLine();

            }

        } catch (IOException e) {

            System.out.println("Error saving tasks to file: " + e.getMessage());

        }

    }

}

**Code Explanation:**’

**1. Task Class**

This class represents a single task in the to-do list.

static class Task {

    String description;

    boolean isCompleted;

    Task(String description, boolean isCompleted) {

        this.description = description;

        this.isCompleted = isCompleted;

    }

    public String toString() {

        return (isCompleted ? "[✔]" : "[  ]") + " " + description;

    }

}

* **Fields**:
  + description: A String holding the task description.
  + isCompleted: A boolean indicating if the task is completed.
* **Constructor**: Initializes the description and isCompleted values.
* **toString()**: Overrides the default toString method to display the task with a checkbox [✔] (for completed tasks) or [ ] (for pending tasks).

**2. Main Method**

The main method contains the core logic of the program.

public static void main(String[] args) {

    ArrayList<Task> tasks = new ArrayList<>();

    String fileName = "Tasks.txt";

    loadTasksFromFile(tasks, fileName);

* **ArrayList<Task>**: Stores the list of tasks in memory.
* **fileName**: Specifies the file name used to save and load tasks.
* **loadTasksFromFile()**: Reads existing tasks from the file Tasks.txt into the tasks list.

**3. Menu Loop**

The program repeatedly displays a menu and performs the selected action until the user chooses to exit.

do {

    System.out.println("\n--- To-Do List Manager ---");

    System.out.println("1. Add Task");

    System.out.println("2. View Tasks");

    System.out.println("3. Mark Task as Completed");

    System.out.println("4. Delete Task");

    System.out.println("5. Exit");

    System.out.print("Enter your choice: ");

* A **do-while loop** is used to display the menu until the user enters 5 (Exit).
* **Input Handling**: Scanner is used to get user input, and exceptions are caught to handle invalid input gracefully.

**4. Add Task**

case 1:

    System.out.println("\nEnter the task description: ");

    String desc = ip.nextLine();

    if (desc.isEmpty()) {

        System.out.println("Task description cannot be empty.");

    } else {

        tasks.add(new Task(desc, false));

        System.out.println("Task added!");

        saveTasksToFile(tasks, fileName);

    }

    break;

* Prompts the user for a task description.
* Validates that the input is not empty.
* Adds a new task (with isCompleted set to false) to the tasks list.
* **saveTasksToFile()**: Saves the updated task list to the file for persistence.

**5. View Tasks**

case 2:

    System.out.println("\nYour Tasks:");

    if (tasks.isEmpty()) {

        System.out.println("No tasks recorded.");

    } else {

        for (int i = 0; i < tasks.size(); i++) {

            System.out.println((i + 1) + ". " + tasks.get(i));

        }

    }

    break;

* Displays the current tasks.
* If the tasks list is empty, it notifies the user. Otherwise, it iterates over the list and displays all tasks using the overridden toString() method.

**6. Mark Task as Completed**

case 3:

    System.out.print("\nEnter task number to mark as completed: ");

    try {

        int completeIndex = ip.nextInt() - 1;

        if (completeIndex >= 0 && completeIndex < tasks.size()) {

            tasks.get(completeIndex).isCompleted = true;

            System.out.println("Task marked as completed!");

            saveTasksToFile(tasks, fileName);

        } else {

            System.out.println("Invalid task number.");

        }

    } catch (Exception e) {

        System.out.println("Invalid input! Please enter a valid task number.");

        ip.next();

    }

    break;

* Prompts the user for the task number.
* Validates the input and checks if the entered index is within the valid range.
* Marks the specified task as completed and saves the updated list to the file.

**7. Delete Task**

case 4:

    System.out.print("\nEnter task number to delete: ");

    try {

        int deleteIndex = ip.nextInt() - 1;

        if (deleteIndex >= 0 && deleteIndex < tasks.size()) {

            tasks.remove(deleteIndex);

            System.out.println("Task deleted!");

            saveTasksToFile(tasks, fileName);

        } else {

            System.out.println("Invalid task number.");

        }

    } catch (Exception e) {

        System.out.println("Invalid input! Please enter a valid task number.");

        ip.next();

    }

    break;

* Prompts the user for the task number to delete.
* Validates the input and removes the specified task from the list.
* Updates the file with the remaining tasks.

**8. Exit**

case 5:

    System.out.println("\nExiting To-Do List Manager. Goodbye!");

    break;

* Terminates the menu loop and exits the program.

**9. Load Tasks from File**

private static void loadTasksFromFile(ArrayList<Task> tasks, String fileName) {

    File file = new File(fileName);

    if (!file.exists()) {

        return;

    }

    try (BufferedReader br = new BufferedReader(new FileReader(file))) {

        String line;

        while ((line = br.readLine()) != null) {

            String[] parts = line.split(";", 2);

            if (parts.length == 2) {

                boolean isCompleted = Boolean.parseBoolean(parts[0]);

                String description = parts[1];

                tasks.add(new Task(description, isCompleted));

            }

        }

    } catch (IOException e) {

        System.out.println("Error reading tasks from file: " + e.getMessage());

    }

}

* Reads tasks from the specified file line by line.
* Each line contains task data in the format isCompleted;description.
* Parses the data and adds tasks to the list.

**10. Save Tasks to File**

private static void saveTasksToFile(ArrayList<Task> tasks, String fileName) {

    try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName))) {

        for (Task task : tasks) {

            bw.write(task.isCompleted + ";" + task.description);

            bw.newLine();

        }

    } catch (IOException e) {

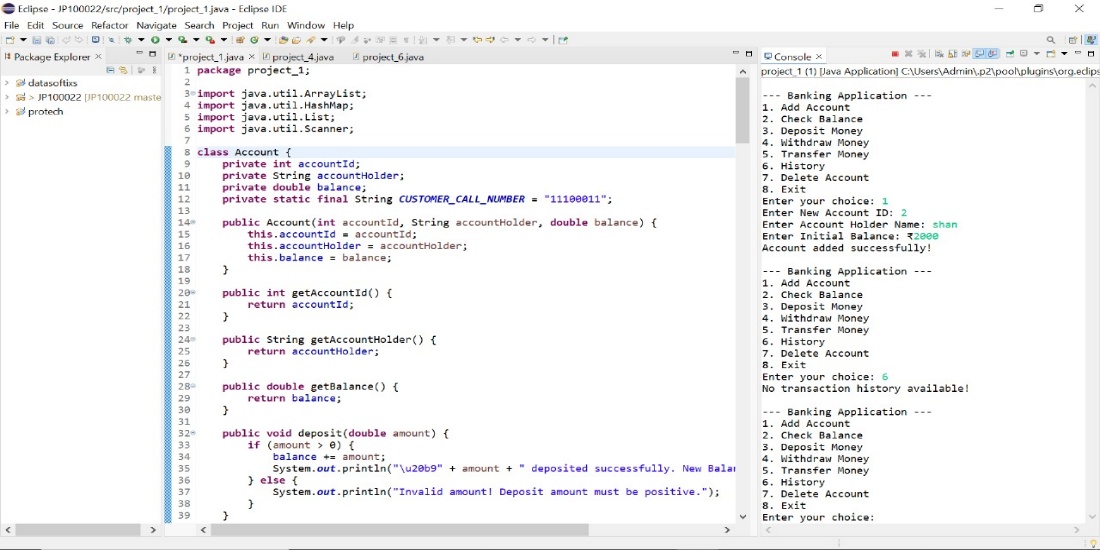
        System.out.println("Error saving tasks to file: " + e.getMessage());

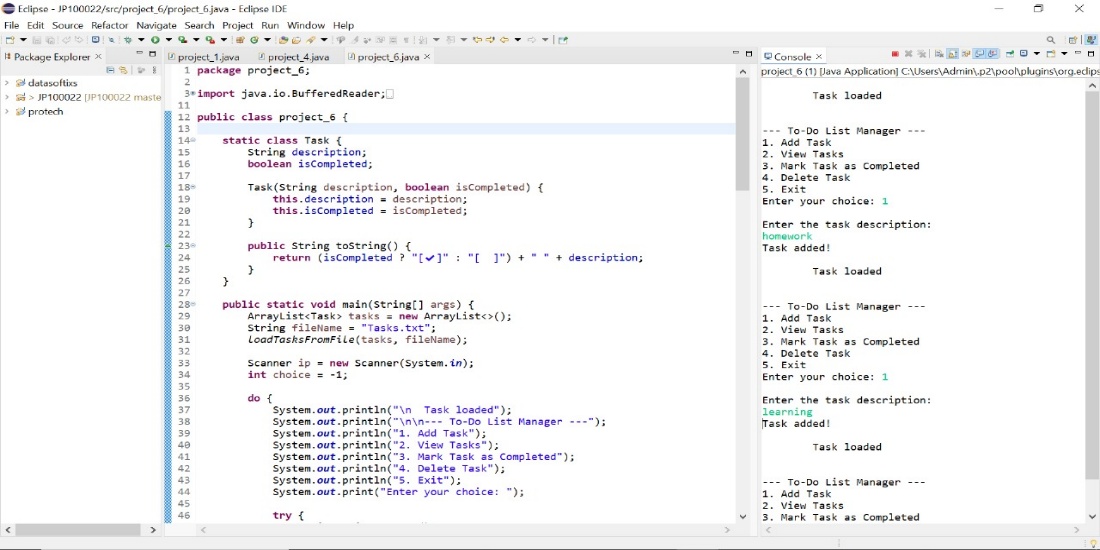
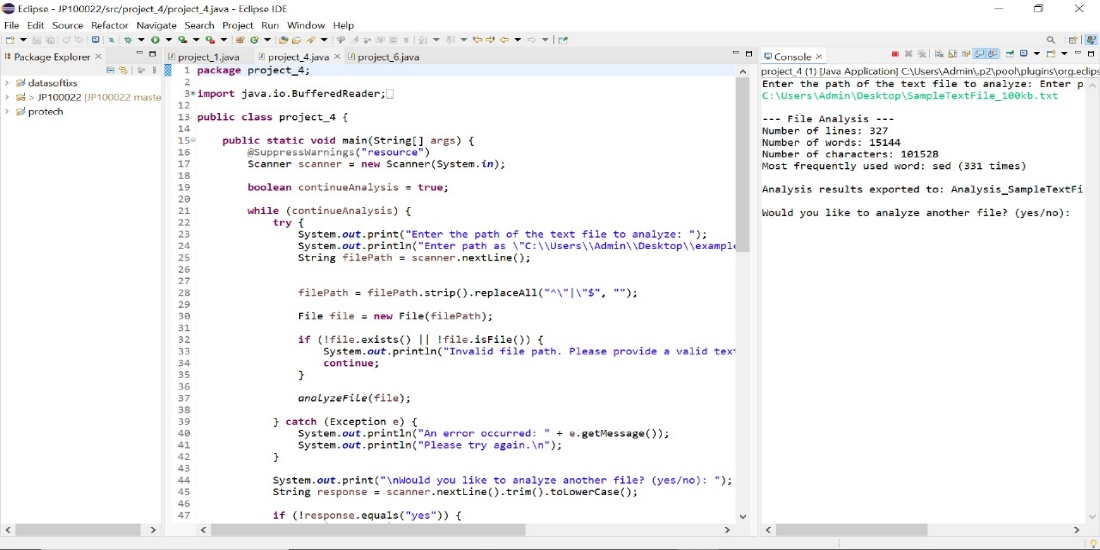
    }

}

* Writes tasks to the file in the format isCompleted;description.
* Ensures data persistence by saving the latest state of the task list.

**Attachments:**

****

****