An Internship Report On:

***JAVA PROGRAMMING***

Prepared By**:**

*BALA MURUGESH J  
INTERN ID: JP100042*

Submitted To:

***DATASOFTIXS***

On The Topic of:

***BASIC UNIT CONVERSION TOOL***

***SIMPLE STUDENT MANAGEMENT SYSTEM***

***TO-DO LIST APPLICATION***

****

**PROJECT – 2**

**QUESTION:**

**Basic Unit Conversion Tool :**

* Objective: Develop a tool to convert between units like length,weight, and temperature.
* Key Features:
  + Provide options for different types of conversions.
  + Use switch-case for efficient decision-making.
  + Include error handling for invalid inputs.
* Optional: Add a history feature to store recent conversions.

**Objective:**

The primary objective of the **Unit Conversion Tool** is to provide a reliable and efficient application for converting units across various categories such as length, weight, and temperature. The tool ensures accurate conversions, employs robust error-handling mechanisms to validate user inputs, and prevents interruptions in the workflow. An optional history feature is included to enhance user experience by allowing quick access to recently performed conversions.

**Abstract:**

The **Unit Conversion Tool** is a versatile application designed to simplify unit conversions across multiple domains, such as length, weight, and temperature. It integrates error-handling mechanisms to validate input data, ensuring seamless operation without disruptions. The tool uses a modular approach with efficient switch-case logic for decision-making. Additionally, the optional history feature records recent conversions, offering users greater convenience and efficiency. This project serves as a practical solution for both educational and professional scenarios, combining functionality with user-friendly features.

**Code:**

package week\_1\_project;

import java.util.ArrayList;

import java.util.Scanner;

public class Project\_2 {

public static Scanner ip = new Scanner(System.in);

static ArrayList<String> History = new ArrayList<>();

public static void main(String[] args) {

int choice = 0;

while (true) {

System.out.println("\n\tEnter your choice");

System.out.println("1. Length");

System.out.println("2. Temperature");

System.out.println("3. Weight");

System.out.println("4. History");

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

try {

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

} catch (NumberFormatException e) {

System.out.println("Invalid input! Please enter a number between 1 and 5.");

continue;

}

switch (choice) {

case 1:

length();

continue;

case 2:

weight();

continue;

case 3:

weight();

continue;

case 4:

System.out.println("\tHistory");

showHistory();

continue;

case 5:

return;

}

System.out.println("Choice should between 1 to 5");

}

}

public static void length() {

int choice = 0;

double d = 0.0D;

double val = 0.0D;

while (true) {

System.out.println("\n\tEnter your choice");

System.out.println("1. Inch to Meter");

System.out.println("2. Inch to Yard");

System.out.println("3. Meter to Inch");

System.out.println("4. Meter to Yard");

System.out.println("5. Yard to Inch");

System.out.println("6. Yard to Meter");

System.out.println("7. Other type of Conversion\n");

try {

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

} catch (NumberFormatException e) {

System.out.println("Invalid input! Please enter a number between 1 and 7.");

continue;

}

if (choice > 0 && choice < 7) {

System.out.print("Enter the value ");

try {

d = ip.nextDouble();

ip.nextLine();

} catch (Exception e) {

System.out.println("Invalid value! Please enter a numeric value.");

ip.nextLine();

continue;

}

}

switch (choice) {

case 1:

val = d / 39.3701D;

System.out.println("The value of " + d + " in Inches = " + val + " Meters");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Inches = " + val + " Meters");

continue;

case 2:

val = d / 36.0D;

System.out.println("The value of " + d + " in Inches = " + val + " Yard");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Inches = " + val + " Yard");

continue;

case 3:

val = d \* 39.3701D;

System.out.println("The value of " + d + " in Meters = " + val + " Inches");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Meters = " + val + " Inches");

continue;

case 4:

val = d \* 1.094D;

System.out.println("The value of " + d + " in Meters = " + val + " Yard");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Meters = " + val + " Yard");

continue;

case 5:

val = d \* 39.0D;

System.out.println("The value of " + d + " in Meters = " + val + " Yard");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Meters = " + val + " Yard");

continue;

case 6:

val = d \* 36.0D;

System.out.println("The value of " + d + " in Yards = " + val + " Inches");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Yards = " + val + " Inches");

continue;

case 7:

return;

}

System.out.println("Choice should between 1 to 7");

}

}

public static void weight() {

int choice = 0;

double d = 0.0D;

double val = 0.0D;

while (true) {

System.out.println("\n\tEnter your choice");

System.out.println("1. Kilo Gram to Pound");

System.out.println("2. Pound to Kilo Gram");

System.out.println("3. Other type of Conversion\n");

try {

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

} catch (NumberFormatException e) {

System.out.println("Invalid input! Please enter a number between 1 and 3.");

continue;

}

if (choice > 0 && choice < 3) {

System.out.print("Enter the value ");

try {

d = ip.nextDouble();

ip.nextLine();

} catch (Exception e) {

System.out.println("Invalid value! Please enter a numeric value.");

ip.nextLine();

continue;

}

}

switch (choice) {

case 1:

val = d \* 2.205D;

System.out.println("The value of " + d + " in Kilo Grams = " + val + " Pounds");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Kilo Grams = " + val + " Pounds");

continue;

case 2:

val = d / 2.205D;

System.out.println("The value of " + d + " in Kilo Grams = " + val + " Pounds");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Kilo Grams = " + val + " Pounds");

continue;

case 3:

return;

}

System.out.println("Choice should between 1 to 3");

}

}

public static void temperature() {

int choice = 0;

double d = 0.0D;

double val = 0.0D;

while (true) {

System.out.println("\n\tEnter your choice");

System.out.println("1. Celsius to Fahrenheit");

System.out.println("2. Fahrenheit to Celsius");

System.out.println("3. Other type of Conversion\n");

try {

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

} catch (NumberFormatException e) {

System.out.println("Invalid input! Please enter a number between 1 and 3.");

continue;

}

if (choice > 0 && choice < 3) {

System.out.print("Enter the value ");

try {

d = ip.nextDouble();

ip.nextLine();

} catch (Exception e) {

System.out.println("Invalid value! Please enter a numeric value.");

ip.nextLine();

continue;

}

}

switch (choice) {

case 1:

val = d \* 1.8D + 32.0D;

System.out.println("The value of " + d + " in Celsius = " + val + " Fahrenheit");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Celsius = " + val + " Fahrenheit");

continue;

case 2:

val = (d - 32.0D) \* 5.0D / 9.0D;

System.out.println("The value of " + d + " in Fahrenheit = " + val + " Celsius");

System.out.println("+============================================+");

addToHistory("The value of " + d + " in Fahrenheit = " + val + " Celsius");

continue;

case 3:

return;

}

System.out.println("Choice should between 1 to 3");

}

}

private static void addToHistory(String conversion) {

History.add(conversion);

}

private static void showHistory() {

if (History.isEmpty()) {

System.out.println("No conversion history available.");

} else {

System.out.println("\nConversion History:");

for (String record : History)

System.out.println(record);

}

}

}

**Code Explanation:**

**1. Package Declaration**

package week\_1\_project;

* The program belongs to the week\_1\_project package. It groups related classes for better organization.

**2. Imports**

import java.util.ArrayList;

import java.util.Scanner;

* ArrayList: Used to maintain a dynamic list of conversion history.
* Scanner: Allows user input from the console.

**3. Global Declarations**

public static Scanner ip = new Scanner(System.in);

static ArrayList<String> History = new ArrayList<>();

* ip: A static Scanner object to take inputs from the user throughout the program.
* History: A static ArrayList that stores the records of previous conversions for the history feature.

**4. Main Method**

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

**Functionality:**

1. **Infinite Loop:** Ensures the program runs until the user chooses to exit.
2. **Menu Display:** Prompts the user to select a type of conversion or view history.
3. **Error Handling:** Validates that input is an integer within the expected range.
4. **Switch-Case:** Calls respective methods (length(), temperature(), weight()) or displays history based on user input.

Key Points:

* Invalid input is caught using a try-catch block to prevent program termination.
* A graceful exit (case 5) stops the loop and ends the program.

**5. Length Conversion**

public static void length() { ... }

**Functionality:**

1. Displays a **sub-menu** for length conversions:
   * Inch ↔ Meter
   * Inch ↔ Yard
   * Meter ↔ Yard
2. Validates input for both menu choices and numeric values.
3. Performs conversions using formulas (e.g., val = d / 39.3701).
4. Adds conversion results to the **History** list using addToHistory().

Key Points:

* Handles invalid numeric inputs using try-catch.
* Provides an option to exit (case 7) back to the main menu.

**6. Weight Conversion**

public static void weight() { ... }

**Functionality:**

1. Offers a **sub-menu** for weight conversions:
   * Kilogram ↔ Pound.
2. Performs conversions using standard formulas:
   * Kilogram to Pound: val = d \* 2.205
   * Pound to Kilogram: val = d / 2.205
3. Adds results to history for future reference.

Key Points:

* Similar structure to length(), ensuring consistent error handling and navigation.

**7. Temperature Conversion**

public static void temperature() { ... }

**Functionality:**

1. Provides a **sub-menu** for temperature conversions:
   * Celsius ↔ Fahrenheit.
2. Uses conversion formulas:
   * Celsius to Fahrenheit: val = d \* 1.8 + 32
   * Fahrenheit to Celsius: val = (d - 32) \* 5 / 9
3. Adds the results to history.

Key Points:

* Ensures valid numeric input for smooth operation.
* Allows easy exit back to the main menu (case 3).

**8. History Management**

private static void addToHistory(String conversion) { ... }

private static void showHistory() { ... }

**addToHistory(String conversion)**

* Adds a conversion record to the History list.
* Tracks all operations performed by the user.

**showHistory()**

* Displays all saved conversions.
* Checks if the history is empty before printing to avoid confusion.

**9. Error Handling**

1. **Input Validation:**
   * Menu inputs are validated with Integer.parseInt inside a try-catch block.
   * Numeric values (e.g., distances, weights) are also checked to ensure valid entries.
2. **Graceful Handling:**
   * Error messages guide the user to correct mistakes.
   * Program flow continues smoothly without crashing.

**10. Overall Structure**

* **Modular Design:** Each type of conversion has its own method for better readability and maintenance.
* **Dynamic Features:**
  + History tracking for user convenience.
  + Error handling to ensure robustness.
* **Scalable:** Easy to add new conversion types or categories.

This tool is user-friendly, robust, and designed for practical use cases.

**PROJECT – 3**

**QUESTION :**

**Simple Student Management System:**

* Objective: Build a program to manage student records.
* Key Features:
  + Allow users to add, view, update, and delete student details (name, ID, marks).
  + Use ArrayList or HashMap for data storage.
  + Provide a menu-driven interface.
* Optional: Implement sorting of students by marks or name.

**Objective**

The objective of this project is to design and implement a simple Student Management System that allows users to efficiently manage student records. The system will enable operations such as adding, viewing, updating, and deleting student details, ensuring an organized approach to handling data. This project aims to simplify student record management through a menu-driven interface, providing users with an intuitive and efficient experience.

**Abstract:**

The Simple Student Management System is a user-friendly Java application designed to streamline the process of managing student records. This project addresses the need for an efficient and organized method to handle operations such as adding, viewing, updating, and deleting student information, including name, ID, and marks. Utilizing data structures like ArrayList or HashMap, the application ensures fast and reliable storage and retrieval of student data.

The system offers a menu-driven interface that guides users through various functionalities, ensuring ease of use even for those with minimal technical expertise. To enhance its capabilities, the application includes an optional feature that allows sorting of student records by marks or name, providing better organization and aiding in decision-making processes, such as identifying top-performing students or generating ordered lists.

Robust error handling mechanisms are integrated into the application, ensuring that invalid inputs or operations do not interrupt the workflow. By maintaining data integrity and offering flexibility in data management, the system is suitable for small educational institutions, coaching centers, or individuals seeking an efficient way to manage student data. This project demonstrates a practical application of core Java concepts and showcases the use of fundamental programming constructs like loops, conditional statements, and data structures.

**Code:**

package week\_2\_project;

import java.io.File;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Scanner;

class student {

int id;

String name;

int mark;

public student(int id, String name, int mark) {

this.id = id;

this.name = name;

this.mark = mark;

}

*@Override*

public String toString() {

return "Student{ID=" + id + ", Name='" + name + "', Mark=" + mark + "}";

}

}

public class Project\_3 {

public static ArrayList<student> *al* = new ArrayList<>();

public static Scanner *ip* = new Scanner(System.***in***);

public static void main(String[] args) {

*loadTasksFromFile*();

int choice;

while (true) {

System.***out***.println("\n\tEnter your choice");

System.***out***.println("1. Add New Entry");

System.***out***.println("2. View the Entry");

System.***out***.println("3. Sort By Mark");

System.***out***.println("4. Sort By Name");

System.***out***.println("5. Search By ID");

System.***out***.println("6. Remove By ID");

System.***out***.println("7. Exit\n");

try {

choice = Integer.*parseInt*(*ip*.nextLine());

} catch (NumberFormatException e) {

System.***out***.println("Invalid input! Please enter a number between 1 and 7.");

continue;

}

switch (choice) {

case 1:

*addEntry*();

break;

case 2:

*show*(*al*);

break;

case 3:

*sortByMark*();

break;

case 4:

*sortByName*();

break;

case 5:

*searchById*();

break;

case 6:

*removeById*();

break;

case 7:

*saveTasksToFile*();

System.***out***.println("Exiting. Goodbye!");

return;

default:

System.***out***.println("Choice should be between 1 to 7.");

}

}

}

public static void addEntry() {

int id;

String name;

int mark;

while (true) {

System.***out***.print("Enter ID for student: ");

try {

id = Integer.*parseInt*(*ip*.nextLine());

break;

} catch (NumberFormatException e) {

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

}

}

System.***out***.print("Enter Name for student: ");

name = *ip*.nextLine();

while (true) {

System.***out***.print("Enter Mark for student: ");

try {

mark = Integer.*parseInt*(*ip*.nextLine());

break;

} catch (NumberFormatException e) {

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

}

}

*al*.add(new student(id, name, mark));

*saveTasksToFile*();

System.***out***.println("Student entry added successfully.");

}

public static <T> void show(ArrayList<T> list) {

if (list.isEmpty()) {

System.***out***.println("No entries found!");

} else {

for (T item : list) {

System.***out***.println(item);

}

}

}

public static void sortByMark() {

if (*al*.isEmpty()) {

System.***out***.println("No entries found!");

} else {

*al*.sort((s1, s2) -> Integer.*compare*(s2.mark, s1.mark));

System.***out***.println("Sorted By Marks (High to Low):");

*show*(*al*);

}

}

public static void sortByName() {

if (*al*.isEmpty()) {

System.***out***.println("No entries found!");

} else {

*al*.sort((s1, s2) -> s1.name.compareToIgnoreCase(s2.name));

System.***out***.println("Sorted By Name:");

*show*(*al*);

}

}

public static void searchById() {

if (*al*.isEmpty()) {

System.***out***.println("No entries found!");

} else {

System.***out***.print("Enter the ID to search: ");

try {

int searchId = Integer.*parseInt*(*ip*.nextLine());

for (student s : *al*) {

if (s.id == searchId) {

System.***out***.println("Student found: " + s);

return;

}

}

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

} catch (NumberFormatException e) {

System.***out***.println("Invalid input!");

}

}

}

public static void removeById() {

if (*al*.isEmpty()) {

System.***out***.println("No entries found!");

} else {

System.***out***.print("Enter the ID to remove: ");

try {

int searchId = Integer.*parseInt*(*ip*.nextLine());

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

if (*al*.get(i).id == searchId) {

System.***out***.println("Student removed: " + *al*.get(i));

*al*.remove(i);

*saveTasksToFile*();

return;

}

}

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

} catch (NumberFormatException e) {

System.***out***.println("Invalid input!");

}

}

}

private static void saveTasksToFile() {

File file = new File("save.txt");

try (java.io.PrintWriter writer = new java.io.PrintWriter(file)) {

for (student s : *al*) {

writer.println(s.id + ", " + s.name + ", " + s.mark);

}

System.***out***.println("Data saved successfully.");

} catch (IOException e) {

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

}

}

private static void loadTasksFromFile() {

File file = new File("save.txt");

if (!file.exists()) {

System.***out***.println("No saved tasks found. Starting fresh.");

return;

}

try (Scanner scanner = new Scanner(file)) {

while (scanner.hasNextLine()) {

String line = scanner.nextLine();

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

if (parts.length == 3) {

int id = Integer.*parseInt*(parts[0]);

String name = parts[1];

int mark = Integer.*parseInt*(parts[2]);

*al*.add(new student(id, name, mark));

}

}

System.***out***.println("Data loaded successfully.");

} catch (IOException | NumberFormatException e) {

System.***out***.println("Error loading tasks: " + e.getMessage());

}

}

**Code Explanation :**

**1. Package and Imports**

package week\_2\_project;

import java.io.File;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Scanner;

* **package week\_2\_project**: Declares the package name, indicating the folder in which this code resides.
* **Imports**:
  + File and IOException: Used for file handling (reading/writing data to/from a file).
  + ArrayList: Dynamic data structure to store the list of students.
  + Scanner: Used for reading user input and reading data from files.

**2. Student Class**

class student {

int id;

String name;

int mark;

public student(int id, String name, int mark) {

this.id = id;

this.name = name;

this.mark = mark;

}

@Override

public String toString() {

return "Student{ID=" + id + ", Name='" + name + "', Mark=" + mark + "}";

}

}

* Represents a **student record** with three fields:
  + id: Student ID (integer).
  + name: Student name (string).
  + mark: Student marks (integer).
* **Constructor**: Used to initialize the fields when creating a new student object.
* **toString() method**: Customizes the string representation of a student object for easy display.

**3. Main Class and Variables**

public class Project\_3 {

public static ArrayList<student> al = new ArrayList<>();

public static Scanner ip = new Scanner(System.in);

* al: A static ArrayList to store multiple student objects.
* ip: A static Scanner object for user input throughout the program.

**4. Main Method**

public static void main(String[] args) {

loadTasksFromFile();

int choice;

while (true) {

System.out.println("\n\tEnter your choice");

System.out.println("1. Add New Entry");

System.out.println("2. View the Entry");

System.out.println("3. Sort By Mark");

System.out.println("4. Sort By Name");

System.out.println("5. Search By ID");

System.out.println("6. Remove By ID");

System.out.println("7. Exit\n");

try {

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

} catch (NumberFormatException e) {

System.out.println("Invalid input! Please enter a number between 1 and 7.");

continue;

}

switch (choice) {

case 1: addEntry(); break;

case 2: show(al); break;

case 3: sortByMark(); break;

case 4: sortByName(); break;

case 5: searchById(); break;

case 6: removeById(); break;

case 7: saveTasksToFile(); System.out.println("Exiting. Goodbye!"); return;

default: System.out.println("Choice should be between 1 to 7.");

}

}

}

* **loadTasksFromFile()**: Loads saved student data from a file into the ArrayList when the program starts.
* **Menu System**: Displays a menu of options and takes user input for the desired operation.
* **Choice Handling**:
  + Handles invalid inputs using a try-catch block.
  + Executes the corresponding method based on the user’s choice (addEntry(), show(), etc.).
  + Exits the program gracefully when choice 7 is selected after saving data.

**5. Add New Entry**

public static void addEntry() {

int id;

String name;

int mark;

while (true) {

System.out.print("Enter ID for student: ");

try {

id = Integer.parseInt(ip.nextLine());

break;

} catch (NumberFormatException e) {

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

}

}

System.out.print("Enter Name for student: ");

name = ip.nextLine();

while (true) {

System.out.print("Enter Mark for student: ");

try {

mark = Integer.parseInt(ip.nextLine());

break;

} catch (NumberFormatException e) {

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

}

}

al.add(new student(id, name, mark));

saveTasksToFile();

System.out.println("Student entry added successfully.");

}

* Adds a new student record to the ArrayList by:
  + Taking user input for ID, name, and marks with proper validation.
  + Storing the new record in the ArrayList.
  + Saving the updated list to a file.

**6. Display Entries**

public static <T> void show(ArrayList<T> list) {

if (list.isEmpty()) {

System.out.println("No entries found!");

} else {

for (T item : list) {

System.out.println(item);

}

}

}

* Displays all student records stored in the ArrayList.
* Prints a message if the list is empty.

**7. Sorting**

**Sort By Marks**

public static void sortByMark() {

if (al.isEmpty()) {

System.out.println("No entries found!");

} else {

al.sort((s1, s2) -> Integer.compare(s2.mark, s1.mark));

System.out.println("Sorted By Marks (High to Low):");

show(al);

}

}

* Sorts students in descending order of marks using a lambda expression.

**Sort By Name**

public static void sortByName() {

if (al.isEmpty()) {

System.out.println("No entries found!");

} else {

al.sort((s1, s2) -> s1.name.compareToIgnoreCase(s2.name));

System.out.println("Sorted By Name:");

show(al);

}

}

* Sorts students alphabetically by name.

**8. Search By ID**

public static void searchById() {

if (al.isEmpty()) {

System.out.println("No entries found!");

} else {

System.out.print("Enter the ID to search: ");

try {

int searchId = Integer.parseInt(ip.nextLine());

for (student s : al) {

if (s.id == searchId) {

System.out.println("Student found: " + s);

return;

}

}

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

} catch (NumberFormatException e) {

System.out.println("Invalid input!");

}

}

}

* Searches for a student by their ID and displays the details if found.

**9. Remove By ID**

public static void removeById() {

if (al.isEmpty()) {

System.out.println("No entries found!");

} else {

System.out.print("Enter the ID to remove: ");

try {

int searchId = Integer.parseInt(ip.nextLine());

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

if (al.get(i).id == searchId) {

System.out.println("Student removed: " + al.get(i));

al.remove(i);

saveTasksToFile();

return;

}

}

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

} catch (NumberFormatException e) {

System.out.println("Invalid input!");

}

}

}

* Removes a student from the ArrayList based on the provided ID.

**10. File Handling**

**Save Data to File**

private static void saveTasksToFile() {

File file = new File("save.txt");

try (java.io.PrintWriter writer = new java.io.PrintWriter(file)) {

for (student s : al) {

writer.println(s.id + ", " + s.name + ", " + s.mark);

}

System.out.println("Data saved successfully.");

} catch (IOException e) {

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

}

}

* Writes the ArrayList data to a file (save.txt), each record as a line.

**Load Data from File**

private static void loadTasksFromFile() {

File file = new File("save.txt");

if (!file.exists()) {

System.out.println("No saved tasks found. Starting fresh.");

return;

}

try (Scanner scanner = new Scanner(file)) {

while (scanner.hasNextLine()) {

String line = scanner.nextLine();

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

if (parts.length == 3) {

int id = Integer.parseInt(parts[0]);

String name = parts[1];

int mark = Integer.parseInt(parts[2]);

al.add(new student(id, name, mark));

}

}

System.out.println("Data loaded successfully.");

} catch (IOException | NumberFormatException e) {

System.out.println("Error loading tasks: " + e.getMessage());

}

}

* Reads student data from the save.txt file at startup and populates the ArrayList.

**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.

**Objective**

The **To-Do List Application** is designed to help users efficiently manage and organize their tasks, offering a simple and intuitive interface that supports the addition, deletion, editing, and completion tracking of tasks. By storing tasks in memory using an ArrayList, the application provides an efficient and responsive way to manage tasks. The core functionality allows users to easily add new tasks, mark them as completed, or delete them when no longer needed. Additionally, the application offers the ability to save tasks to a text file, ensuring data persistence even when the application is closed, and allowing users to load previously saved tasks when they return. This feature enhances the practicality of the tool by ensuring that tasks are not lost, offering a convenient way to keep track of important activities over time. By combining essential task management features with a focus on user convenience and data persistence, the To-Do List Application aims to streamline personal productivity, helping users stay organized and on top of their responsibilities.

**Abstract**

The **To-Do List Application** is a Java-based desktop application designed to help users manage and organize their tasks effectively. This application provides a user-friendly interface using the Swing GUI framework, which enables users to add new tasks, edit existing ones, mark tasks as completed, and delete tasks from their to-do list. Each task is represented by a description and a completion status, with tasks that are marked as completed shown with a checkmark symbol.

The application stores the tasks in memory using an ArrayList, which ensures efficient handling and management of the list. Additionally, the application supports data persistence by saving the list of tasks to a text file and loading it back upon application startup. This ensures that the tasks remain intact even after the program is closed and reopened.

To further enhance the user experience, the program allows for toggling the completion status of tasks, providing a simple way to track progress. With its minimal yet effective features, the **To-Do List Application** is an essential tool for anyone looking to stay organized, manage their daily responsibilities, and keep track of their tasks. By combining ease of use with functionality, this application addresses the common need for task management and contributes to improved productivity and organization.

**Code :**

package week\_3\_project;

import java.awt.BorderLayout;

import java.awt.GridLayout;

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 javax.swing.DefaultListModel;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JList;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTextField;

public class Project\_6 {

static class Task {

String description;

boolean isCompleted;

Task(String description) {

this.description = description;

this.isCompleted = false;

}

@Override

public String toString() {

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

}

}

private static final String FILE\_NAME = "tasks.txt";

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

private static DefaultListModel<String> listModel;

private static JList<String> taskList;

public static void main(String[] args) {

loadTasksFromFile();

JFrame frame = new JFrame("To-Do List Application");

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setSize(720, 480);

frame.setLayout(new BorderLayout());

listModel = new DefaultListModel<>();

updateTaskList();

taskList = new JList<>(listModel);

JScrollPane scrollPane = new JScrollPane(taskList);

JTextField taskInput = new JTextField();

JButton addButton = new JButton("Add Task");

JButton completeButton = new JButton("Mark Completed");

JButton removeMarkButton = new JButton("Remove Marking");

JButton deleteButton = new JButton("Delete Task");

JButton saveButton = new JButton("Save and Exit");

JPanel inputPanel = new JPanel(new BorderLayout());

inputPanel.add(taskInput, BorderLayout.CENTER);

inputPanel.add(addButton, BorderLayout.EAST);

JPanel buttonPanel = new JPanel(new GridLayout(1, 4));

buttonPanel.add(completeButton);

buttonPanel.add(removeMarkButton);

buttonPanel.add(deleteButton);

buttonPanel.add(saveButton);

frame.add(inputPanel, BorderLayout.NORTH);

frame.add(scrollPane, BorderLayout.CENTER);

frame.add(buttonPanel, BorderLayout.SOUTH);

addButton.addActionListener(e -> {

String desc = taskInput.getText().trim();

if (!desc.isEmpty()) {

tasks.add(new Task(desc));

taskInput.setText("");

updateTaskList();

}

});

completeButton.addActionListener(e -> {

int index = taskList.getSelectedIndex();

if (index != -1) {

tasks.get(index).isCompleted = true;

updateTaskList();

} else {

JOptionPane.showMessageDialog(frame, "Please select a task to mark as completed.");

}

});

removeMarkButton.addActionListener(e -> {

int index = taskList.getSelectedIndex();

if (index != -1) {

tasks.get(index).isCompleted = false;

updateTaskList();

} else {

JOptionPane.showMessageDialog(frame, "Please select a task to remove the completion marking.");

}

});

deleteButton.addActionListener(e -> {

int index = taskList.getSelectedIndex();

if (index != -1) {

tasks.remove(index);

updateTaskList();

} else {

JOptionPane.showMessageDialog(frame, "Please select a task to delete.");

}

});

saveButton.addActionListener(e -> {

saveTasksToFile();

JOptionPane.showMessageDialog(frame, "Tasks saved. Exiting...");

System.exit(0);

});

frame.setVisible(true);

}

private static void updateTaskList() {

listModel.clear();

for (Task task : tasks) {

listModel.addElement(task.toString());

}

}

private static void saveTasksToFile() {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE\_NAME))) {

for (Task task : tasks) {

writer.write(task.description + "," + task.isCompleted + "\n");

}

} catch (IOException e) {

JOptionPane.showMessageDialog(null, "Error saving tasks: " + e.getMessage());

}

}

private static void loadTasksFromFile() {

File file = new File(FILE\_NAME);

if (!file.exists()) return;

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

String line;

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

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

Task task = new Task(taskData[0]);

task.isCompleted = Boolean.parseBoolean(taskData[1]);

tasks.add(task);

}

} catch (IOException e) {

JOptionPane.showMessageDialog(null, "Error loading tasks: " + e.getMessage());

}

}

}

**Code Explanation:**’

This code is a To-Do List application built using Java, leveraging the **Swing** library for the graphical user interface (GUI). The application enables users to manage tasks by adding, completing, removing completion marks, deleting tasks, and saving tasks to a text file. Let's break down the main components of the code, with a focus on the GUI part to help you gain a good understanding for your report.

**1. Task Class:**

The Task class is used to represent individual tasks. It contains:

* **description**: A string that holds the task's description.
* **isCompleted**: A boolean flag that tracks whether the task is marked as completed.

The Task class also overrides the toString() method to display tasks in a user-friendly format, either marked with a check ("[✔]") or left unchecked ("[ ]") depending on whether the task is completed or not.

static class Task {

String description;

boolean isCompleted;

Task(String description) {

this.description = description;

this.isCompleted = false;

}

@Override

public String toString() {

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

}

}

**2. GUI Setup with JFrame:**

The GUI is built using the **Swing** library, which provides various components such as frames, buttons, text fields, and panels. The GUI is managed inside a JFrame, which is the main window of the application.

JFrame frame = new JFrame("To-Do List Application");

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setSize(720, 480);

frame.setLayout(new BorderLayout());

Here, the JFrame is initialized with the title "To-Do List Application", and its layout is set to **BorderLayout** for organizing components.

**3. Task List Display (JList):**

The tasks are displayed in a **JList**, which is a Swing component for displaying a list of items. The list is backed by a DefaultListModel, which allows dynamic updates (e.g., adding or removing tasks).

listModel = new DefaultListModel<>();

updateTaskList();

taskList = new JList<>(listModel);

JScrollPane scrollPane = new JScrollPane(taskList);

The listModel stores the list of task descriptions, and the taskList is the actual component that displays these tasks. The JScrollPane is used to allow scrolling when there are many tasks.

**4. Task Input Field:**

A **JTextField** component allows users to enter a description for new tasks.

JTextField taskInput = new JTextField();

The entered text can be added to the list as a new task.

**5. Buttons for Task Operations:**

The GUI contains five buttons to interact with the tasks:

* **Add Task** (addButton)
* **Mark Completed** (completeButton)
* **Remove Marking** (removeMarkButton)
* **Delete Task** (deleteButton)
* **Save and Exit** (saveButton)

Each button has an ActionListener to define the behavior when clicked:

addButton.addActionListener(e -> {

String desc = taskInput.getText().trim();

if (!desc.isEmpty()) {

tasks.add(new Task(desc));

taskInput.setText("");

updateTaskList();

}

});

For example, when the "Add Task" button is clicked, the application adds a new task with the description entered in the text field. The task list is then updated to reflect the changes.

**6. Task Operations (Mark as Completed, Delete, Remove Marking):**

Each task can be marked as completed or have its completion status removed:

* **Mark Completed**: This button marks the selected task as completed by updating its isCompleted field to true.
* **Remove Marking**: This button removes the completion status from a task by setting isCompleted to false.
* **Delete Task**: This button deletes the selected task from the list.

These operations are performed based on the selected task in the JList (using taskList.getSelectedIndex()), ensuring that the right task is modified or removed.

**7. Save and Load Tasks:**

* **Save Tasks to File**: The application saves tasks to a text file (tasks.txt) when the "Save and Exit" button is clicked. Each task is written in the format <description>,<isCompleted>, where isCompleted is a boolean.

private static void saveTasksToFile() {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE\_NAME))) {

for (Task task : tasks) {

writer.write(task.description + "," + task.isCompleted + "\n");

}

} catch (IOException e) {

JOptionPane.showMessageDialog(null, "Error saving tasks: " + e.getMessage());

}

}

* **Load Tasks from File**: When the application starts, it attempts to load tasks from the file and populate the task list. This allows users to persist their tasks between sessions.

private static void loadTasksFromFile() {

File file = new File(FILE\_NAME);

if (!file.exists()) return;

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

String line;

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

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

Task task = new Task(taskData[0]);

task.isCompleted = Boolean.parseBoolean(taskData[1]);

tasks.add(task);

}

} catch (IOException e) {

JOptionPane.showMessageDialog(null, "Error loading tasks: " + e.getMessage());

}

}

**8. Final Layout and Display:**

The final layout consists of:

* **Input Panel**: The taskInput field and "Add Task" button.
* **Button Panel**: The buttons for completing, removing completion, deleting, and saving tasks.
* **Scroll Pane**: For displaying the list of tasks with the option to scroll when needed.

frame.add(inputPanel, BorderLayout.NORTH);

frame.add(scrollPane, BorderLayout.CENTER);

frame.add(buttonPanel, BorderLayout.SOUTH);

**9. Event Handling and User Feedback:**

The GUI includes **JOptionPane** to display error messages if a task is not selected or if there is an issue with saving/loading tasks. This ensures a smooth user experience with clear feedback on their actions.

**Attachments:**

 

