JAVA PROGRAMMING INTERNSHIP

PROJECT-1

Project Title: Daily Expenses Tracker

Objective: When creating a Daily Expenses Tracker program in Java, the primary objective is to design a software tool that helps users record, manage, and analyze their daily expenses.

Project Features:

1.Add Expenses:

Users can log expenses by specifying the amount, category and short description.

```
import java .util.  Array List;
import java. util. Scanner;
// Class to represent an expense
class Expense {
   private String category;
   private double amount;
   private String description;
   public Expense(String category, double amount, String description) {
      this. category = category;
      this. amount = amount;
      this. description = description;
   }

   public String get Category() {
      return category;
   }


   public double get Amount() {
```

```
        return amount;
    }

    public String get Description() {
        return description;
    }

    @Override
    public String to String() {
        return "Category: " + category + ", Amount: $" + amount + ", Description: " + description;
    }
}

public class Expense Tracker {
    priv…
```

2.View Summary:

The applications can display

- Total expenses for the day
- Total expenses for the week
- Total expenses for the month

```
import java. time. Local Date;
import java. util. Array List;
import java. util. Scanner;
// Class to represent an expense
class Expense {
    private String category;
    private double amount;
    private String description;
    private Local Date date;
    public Expense(String category, double amount, String description, Local Date date) {
```

```
        this. Category = category;
        this. Amount = amount;
        this. Description = description;
        this. Date = date;
    }

    public String get Category() {
        return category;
    }

    public double get Amount() {
        return amount;
    }

    public String get Description() {
        return description;
    }

    public Local Date get Date() {
        return date;
    }

    @Override
    public String …
```

3.Save Data :

All expenses records are stored in a text file for feature reference.

```
import java.io.*;
import java. time. Local Date;
import java. util. Array List;
import java. util. Scanner;
```

```
// Class to represent an expense
class Expense {
    private String category;
    private double amount;
    private String description;
    private Local Date date;

    public Expense(String category, double amount, String description, LocalDate date) {
        this. category = category;
        this. amount = amount;
        this. description = description;
        this .date = date;
    }

    public String get Category() {
        return category;
    }

    public double get Amount() {
        return amount;
    }

    public String get Description() {
        return description;
    }

    public Local Date get Date() {
        return date;
    }
```

@Override

…

4.Concept Covered:

- **Object -oriented programming [OOP]:**Design classes like expenses and expenses manager to encapsulate data and logic.

```
import java.io.*;
import java. time. Local Date;
import java. util. Array List;
import java. util. List;
// Class to represent an individual expense
class Expense implements Serializable {
    private String category;
    private double amount;
    private String description;
    private Local Date date;
    public Expense(String category, double amount, String description, Local Date date) {
        this. category = category;
        this. amount = amount;
        this. description = description;
        this. date = date;
    }

    public String get Category() {
        return category;
    }

    public double get Amount() {
        return amount;
    }
```

```
    public String get Description() {
        return description;
    }

    public Local Date get Date() {
        re…
```

File Handling: Use java's file I/O to store and retrieve expense data.

```
import java.io.*;
import java. time. Local Date;
import java. util. Array List;
import java. util. List;
import java. util. Scanner;
// Class to represent an individual expense
class Expense {
    private String category;
    private double amount;
    private String description;
    private Local Date date;
    public Expense(String category, double amount, String description, Local Date date) {
        this. category = category;
        this. amount = amount;
        this. description = description;
        this .date = date;
    }

    public String get Category() {
        return category;
    }
```

```
    public double get Amount() {
        return amount;
    }

    public String get Description() {
        return description;
    }

    public Local Date get Date() {
        …
```

Basic Loops And User Input: Implement loops for iterative operations and handle user input seamlessly.

```
import java.io.*;
import java .time. Local Date;
import java. util. Array List;
import java. util. List;
import java. util. Scanner;
// Class to represent an individual expense
class Expense {
    private String category;
    private double amount;
    private String description;
    private Local Date date;
    public Expense(String category, double amount, String description, Local Date date) {
        this. category = category;
        this. amount = amount;
        this. description = description;
        this. date = date;
    }
```

```
public String get Category() {
    return category;
}

public double get Amount() {
    return amount;
}

public String get Description() {
    return description;
}

public Local Date get Date() {
    …
```

Tips For Success:

1.Plan Your Code Structure: Identify the necessary classes, methods , and their interactions before starting the implementations. Suggested classes include:

Expense for storing details of individual expenses.

Code Structure:

```
class Expense:
    def _in it_(self, name, amount, category, date):
        self.name = name          # Name or description of the expense
        self. Amount = amount      # Amount of the expense
        self. Category = category  # Category of the expense (e.g., Food, Transport, etc.)
        self. Date = date          # Date of the expense (can be a string or datetime)
    def _report_(self):
        return f"Expense({self.name}, {self. Amount}, {self. Category}, {self. Date})"
class Expense Tracker:
    def _init_(self):
        self.expenses = []  # List to store multiple expenses
```

```
    def add_expense(self, expense: Expense):
        """Add a new expense to the tracker."""
        self.expenses.append(expense)
    def get_total_expenses(self):
        """Calculate the total amount spent across all expenses."""
        return sum(expense.amount for expense in self.expenses)
    def get_expenses_by_category(self, category):
        """Filter expenses by a given category."""
        return [expense for expense in self.expenses if expense.category == category]
    def get_expenses_on_date(self, date):
        """Filter expenses by a specific date."""
        return [expense for expense in self.expenses if expense.date == date]
    def display_expenses(self):
        """Display all recorded expenses."""
        for expense in self.expenses:
            print(f"{expense.name} | {expense.amount} | {expense.category} | {expense.date}")
# Example usage
if _name_ == "_main_":
    # Create an expense tracker
    tracker = ExpenseTracker()
    # Adding expenses
    tracker.add_expense(Expense("Lunch", 15.00, "Food", "2025-01-21"))
    tracker.add_expense(Expense("Bus ticket", 2.50, "Transport", "2025-01-21"))
    tracker.add_expense(Expense("Coffee", 3.00, "Food", "2025-01-20"))
    # Display all expenses
    tracker.display_expenses()
    # Get total expenses
    print("\nTotal Expenses:", tracker.get_total_expenses())

    # Filter expenses by category
```

```
food_expenses = tracker.get_expenses_by_category("Food")
print("\nFood Expenses:", food_expenses)
# Filter expenses by date
date_expenses = tracker.get_expenses_on_date("2025-01-21")
print("\nExpenses on 2025-01-21:", date_expenses)
```

Explanation:

Expense Class: Represents an individual expense. Each expense has a name, amount, category, and date.

ExpenseTracker Class: Manages a list of expenses. It allows you to:

Add expenses (add_expense)

Get the total of all expenses (get_total_expenses)

Filter expenses by category (get_expenses_by_category)

Filter expenses by date (get_expenses_on_date)

Display all expenses (display_expenses)

Example Usage:

The program records a few expenses and then calculates the total, filters by category, and lists expenses by date.

You can easily extend this structure to include more complex features, such as handling multiple currencies, saving the expenses to a file or database, and more detailed reporting.

2.Start sample : Begin by implementing basic functionalities like adding and displaying all expenses before moving to advanced features like summaries and file handling.

Step-by-Step Approach

1. Define the Structure

Use a class Expense to represent an expense, including a description and amount.

Use an ArrayList to store the expenses.

2. Create a Simple Menu

Use a console-based menu for interaction with options to:

Add an expense.

Display all expenses.

Exit the program.

3. Implement the Logic

Add logic to:

Accept user input for expense details.

Store the expenses in a list.

Display the stored expenses when requested.

Code Example

java

Copy code

```
import java.util.ArrayList;
import java.util.Scanner;
class Expense {
    private String description;
    private double amount;
    public Expense(String description, double amount) {
        this.description = description;
        this.amount = amount;
    }

    @Override
    public String toString() {
        return "Description: " + description + ", Amount: $" + amount;
    }
}

public class ExpenseTracker {
    public static void main(String[] args) {
        ArrayList<Expense> expenses = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);
        int choice;
        do {
            System.out.println("\n--- Expense Tracker ---");
```

```
System.out.println("1. Add an Expense");
System.out.println("2. Display All Expenses");
System.out.println("3. Exit");
System.out.print("Enter your choice: ");
choice = scanner.nextInt();
scanner.nextLine(); // Consume the newline character
switch (choice) {
    case 1:
        // Add an expense
        System.out.print("Enter expense description: ");
        String description = scanner.nextLine();
        System.out.print("Enter expense amount: ");
        double amount = scanner.nextDouble();
        expenses.add(new Expense(description, amount));
        System.out.println("Expense added successfully!");
        break;
    case 2:
        // Display all expenses
        if (expenses.isEmpty()) {
            System.out.println("No expenses to display.");
        } else {
            System.out.println("\n--- List of Expenses ---");
            for (Expense expense : expenses) {
                System.out.println(expense);
            }
        }
        break;

    case 3:
        // Exit the program
```

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

            default:
                System.out.println("Invalid choice. Please try again.");
        }
    } while (choice != 3);

    scanner.close();
  }
}
```

How It Works

Adding an Expense:

Prompts the user to enter a description and amount.

Creates a new Expense object and adds it to the ArrayList.

Displaying All Expenses:

Iterates through the ArrayList and prints each expense.

Handles the case where no expenses are recorded.

Exit Option:

Allows the user to quit the program cleanly.

[12:18 PM, 1/22/2025] Malle_hema: Example Output

mathematica

Copy code

```
--- Expense Tracker ---
1. Add an Expense
2. Display All Expenses
3. Exit
Enter your choice: 1
Enter expense description: Groceries
Enter expense amount: 50.0
```

Expense added successfully!

--- Expense Tracker ---

1. Add an Expense

2. Display All Expenses

3. Exit

Enter your choice: 2

--- List of Expenses ---

Description: Groceries, Amount: $50.0

--- Expense Tracker ---

1. Add an Expense

2. Display All Expenses

3. Exit

Enter your choice: 3

Exiting the program. Goodbye!

Next Steps

Once you've implemented and tested these basic functionalities:

Add a total expense summary.

Allow categorization of expenses.

Implement file handling to save and retrieve data.

Let me know if you'd like guidance on any of these next steps!

3.Test incrementally: Test each feature as you implementing it to ensure smooth functionality and identify bugs early.

Step 1: Test Basic Setup

Before writing the full code, test the setup:

Create a basic main method with a simple print statement.

Run the program to ensure your development environment is working.

Example:

java

Copy code

public class ExpenseTracker {

```
    public static void main(String[] args) {
        System.out.println("Expense Tracker is running!");
    }
}
```

Step 2: Test Adding an Expense

After implementing the "Add Expense" feature:

Write the code to accept user input for a single expense.

Print the entered details to verify the input.

Test Code:

java

Copy code

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter expense description: ");
String description = scanner.nextLine();
System.out.print("Enter expense amount: ");
double amount = scanner.nextDouble();
System.out.println("Expense Added - Description: " + description + ", Amount: " + amount);
```

What to Check:

Does the program accept valid input?

What happens if invalid input (e.g., text instead of a number) is entered?

Step 3: Test Storing Expenses

After creating the Expense class and ArrayList for storage:

Add a single expense to the list and print the list to verify it's stored correctly.

Test Code:

java

Copy code

```
ArrayList<Expense> expenses = new ArrayList<>();
expenses.add(new Expense("Groceries", 50.0));
for (Expense expense : expenses) {
    System.out.println(expense);
```

}

What to Check:

Is the expense stored in the list correctly?

Does the toString() method of the Expense class display details as expected?

Step 4: Test Displaying All Expenses

Once the "Display All Expenses" functionality is implemented:

Add multiple expenses to the list.

Display them to verify the output format and correctness.

Test Code:

java

Copy code

```
expenses.add(new Expense("Groceries", 50.0));
expenses.add(new Expense("Coffee", 3.5));
for (Expense expense : expenses) {
    System.out.println(expense);
}
```

What to Check:

Does the program handle multiple expenses?

What happens if the list is empty?

Step 5: Test the Menu

After adding the main menu and user input handling:

Test each option individually to ensure it works as intended.

Try invalid inputs (e.g., entering letters instead of numbers for choices).

Test Case: Run the menu and test these scenarios:

Add multiple expenses.

Display expenses when the list is empty.

Display expenses after adding some.

Exit the program.

Step 6: Test Edge Cases

Empty Inputs:

What happens if the user presses "Enter" without typing anything?

Negative Values:

Does the program accept negative expense amounts?

Invalid Data Types:

What happens if the user enters text instead of a number for the amount?

Step 7: Test Incremental Features

Once the basics are working, test each new feature incrementally:

Summaries:

Add a feature to calculate total expenses and test it with different scenarios (e.g., no expenses, multiple expenses).

File Handling:

Save expenses to a file and verify the file contents.

Load expenses from a file and check if they’re displayed correctly.

Tips for Incremental Testing

Test After Each Change:

Write a small piece of functionality, test it, and then move on to the next feature.

Use Debug Statements:

Print values at critical points to verify they’re as expected.

Handle Errors Gracefully:

Add error handling (e.g., for invalid inputs) and test those scenarios.

Example: Incremental Testing Workflow

Step 1: Write and test adding one expense.

Step 2: Add and test multiple expenses.

Step 3: Test displaying expenses (empty and non-empty).

Step 4: Integrate the menu and test all options sequentially.

Step 5: Add summaries and test with varying data.

Step 6: Add file handling and test saving/loading.

By testing incrementally, you’ll catch issues early, avoid introducing bugs into already working code, and make debugging much easier. Let me know if you need help testing a specific part!

4.Organize your code: Use clear, meaningful variable and method names for readability and maintainability.

Key Principles for Organizing Code

Use Classes for Separation of Concerns:

Separate logic into meaningful classes (e.g., Expense, Expense Manager, and Main).

Avoid putting all logic in the main method.

Follow Single Responsibility Principle:

Each class or method should handle only one task.

Example: The Expense Manager class handles expense storage and operations, while the Main class handles user interaction.

Keep Methods Short and Focused:

Break long methods into smaller ones for clarity and reusability.

Use Constants:

Define reusable constants for menu options or messages.

Organized Code Example

java

Copy code

```
import java.util. Array List;
import java.util. Scanner;
// Class to represent an expense
class Expense {
    private String description;
    priva…
```

Name: M.Hemalatha

Roll No: 223N5A0407

Branch: ECE

College: PVKK IT Anatapuram