Even Semester (2022)

BINUS UNIVERSITY

BINUS INTERNATIONAL

Object-Oriented Programming Final Project

(Individual Work)

Student Information:

Surname: Munthe **Given Name:** Priscilla Abigail **Student ID:** 2602109883

Course Code : COMP6699001 **Course Name** : Algorithm and Programming

Class : L2AC **Lecturer** : Jude Joseph Lamug Martinez, MCS

Type of Assignment: Final Project Report

Submission Pattern

Due Date : 16 June 2023 **Submission Date** : 16 June 2023

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BINUS International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept, and consent to BINUS International's terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: Priscilla Abigail Munthe

BINUS UNIVERSITY **1**
BINUS INTERNATIONAL **1**
Object-Oriented Programming Final Project 1
A. Project Specification **4**
1. Program Name and Logo 4
2. Program Description 4
3. Program Flow Summary 4
4. Technical Descriptions 5
5. Program Dependencies 6
B. Solution Design **7**
1. UML Class Diagram 7
2. Screenshots of the Progam 8
C. Code Design and Explanation **11**
1. Project Structure 11
2. Credentials 11
3. GUI 15
4. GUIhelper 38
5. Model 39
6. Saved 52
D. Lessons Learned **52**
References **52**
Relevant Links **52**
GitHub Repository: 52
Demo Video: 52

A. Project Specification

1. Program Name and Logo

The name of the program I made this semester is BudgetBuddy. The name itself encapsulates the program's purpose, which is to be a reliable companion in the quest for financial stability.

The image to the right of the program name is the program logo, which has finance-related elements incorporated into it.

2. Program Description

BudgetBuddy is an expense tracker desktop application that simplifies financial management. It was made using several libraries including Java Swing, JfreeChart, and other Java utility libraries. The program manages the data stored by writing into and reading from their corresponding txt files.

3. Program Flow Summary

As a user first opens the program, the user will be prompted to enter their credentials, i.e., their username and password. The user can also make a new account in case they do not have a preexisting account.

Once logged in, the program opens the dashboard page, which serves as the central hub. From here, users can easily browse through their expenses and access various program features using buttons that lead to different pages.

The "Add Income" button opens a page where users can add new income transactions. They can select the income transaction category, enter the amount, description, and transaction date, and click "Add" to save it. Users can also create new categories by selecting the "Add New Category" button.

Similarly, the "Add Expense" button allows users to add new expense transactions by filling in the required fields. However, each expense category has its own transaction limit. When adding a new category, users must specify the limit amount after the category name. Moreover, users cannot add a transaction with an amount exceeding the category's limit.

Clicking on the "View Transaction Details" button takes the user to a dedicated page displaying a table with all their stored transactions. By selecting a specific transaction, detailed information about it appears on the upper right side of the page. Additionally, a visual representation of the user's expenses is presented through a pie chart located in the upper left corner of the page.

4. Technical Descriptions

Java Swing is a Java library that helps developers create graphical user interfaces (GUIs) for desktop applications. It provides various components like buttons, labels, and menus, which can be customized and arranged to build the interface. Swing is platform-independent and supports features like event handling and layout management. It allows developers to design interactive and visually appealing applications with ease.

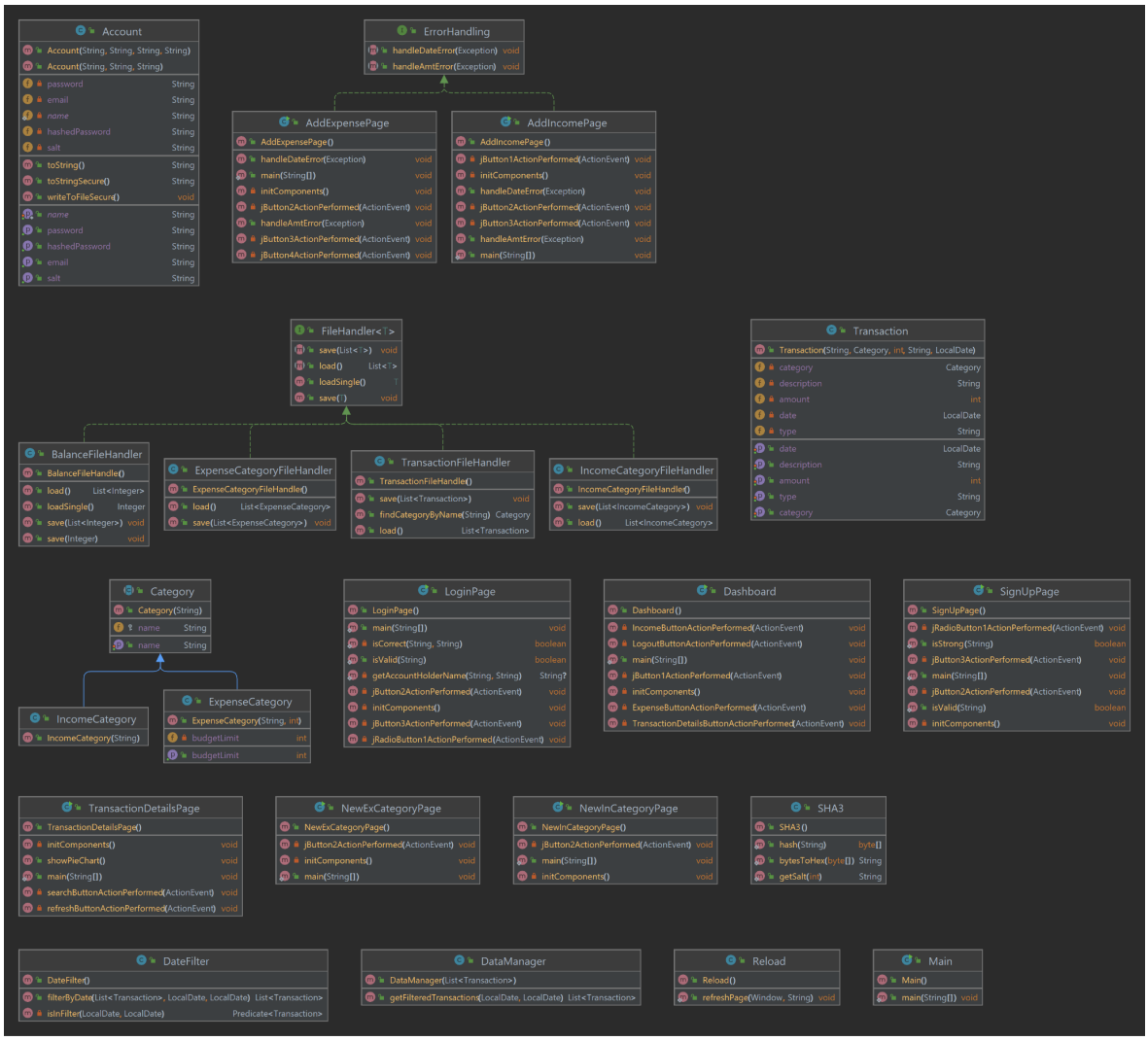
JFreeChart is a Java library that enables developers to create professional-quality charts and graphs in their applications. It provides a wide range of chart types, including bar charts, pie charts, line charts, and more. With JFreeChart, developers can easily visualize data, customize chart appearance, and add interactive features. It simplifies the process of data visualization and allows for the creation of visually appealing and informative charts in Java applications.

5. Program Dependencies

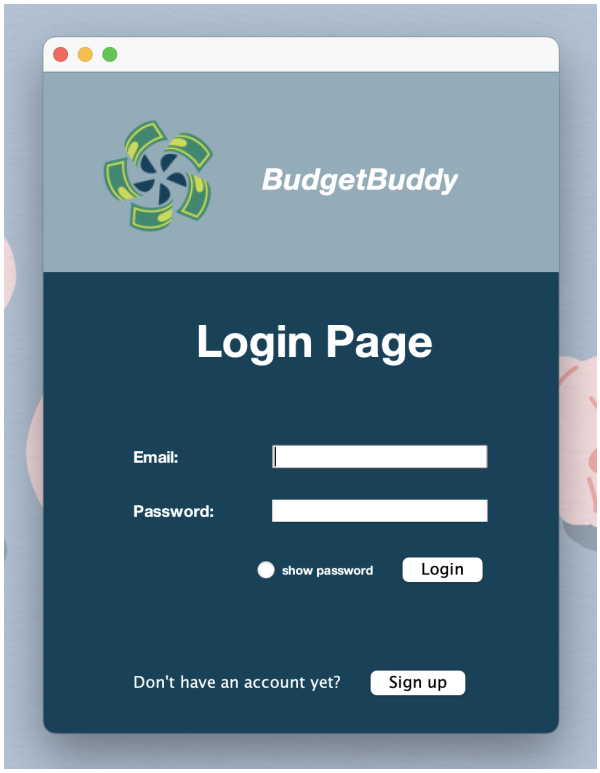
```
javax.swing.*;

javax.swing.table.DefaultTableModel;

javax.swing.event.ListSelectionEvent;

javax.swing.event.ListSelectionListener;

java.awt.*;

java.io.*;

java.nio.file.Files;

java.nio.file.Paths;

java.time.LocalDate;

java.time.format.DateTimeFormatter;

java.time.format.DateTimeParseException;

java.util.*;

java.util.function.Predicate;

java.util.regex.Pattern;

org.jfree.chart.*;

org.jfree.chart.plot.PiePlot;

org.jfree.data.general.DefaultPieDataset;
```

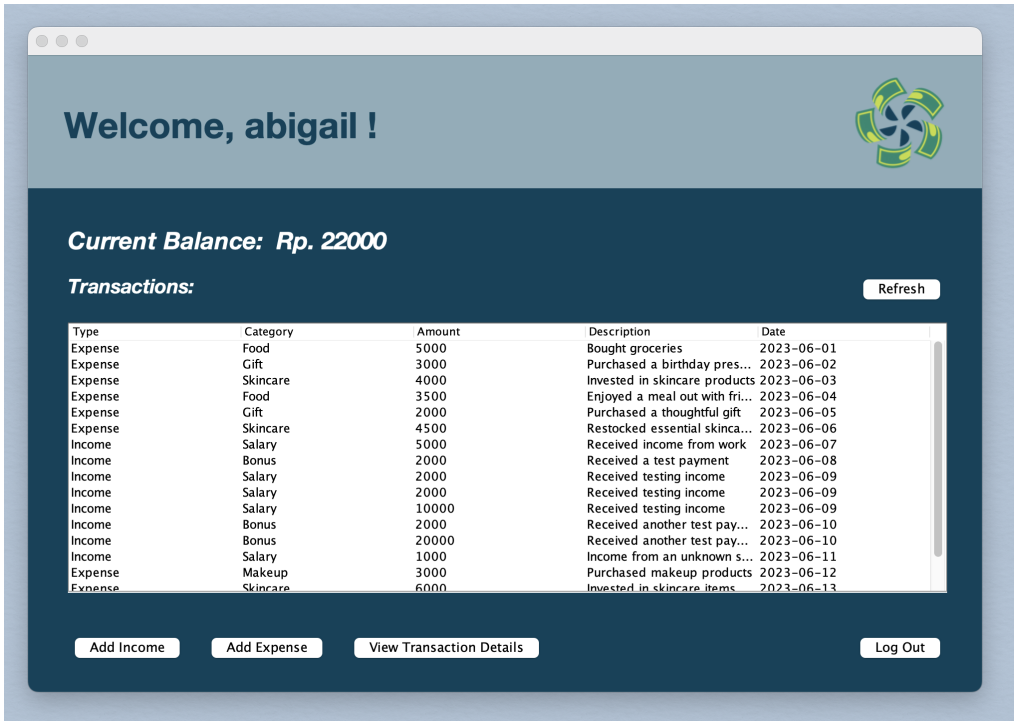
B. Solution Design

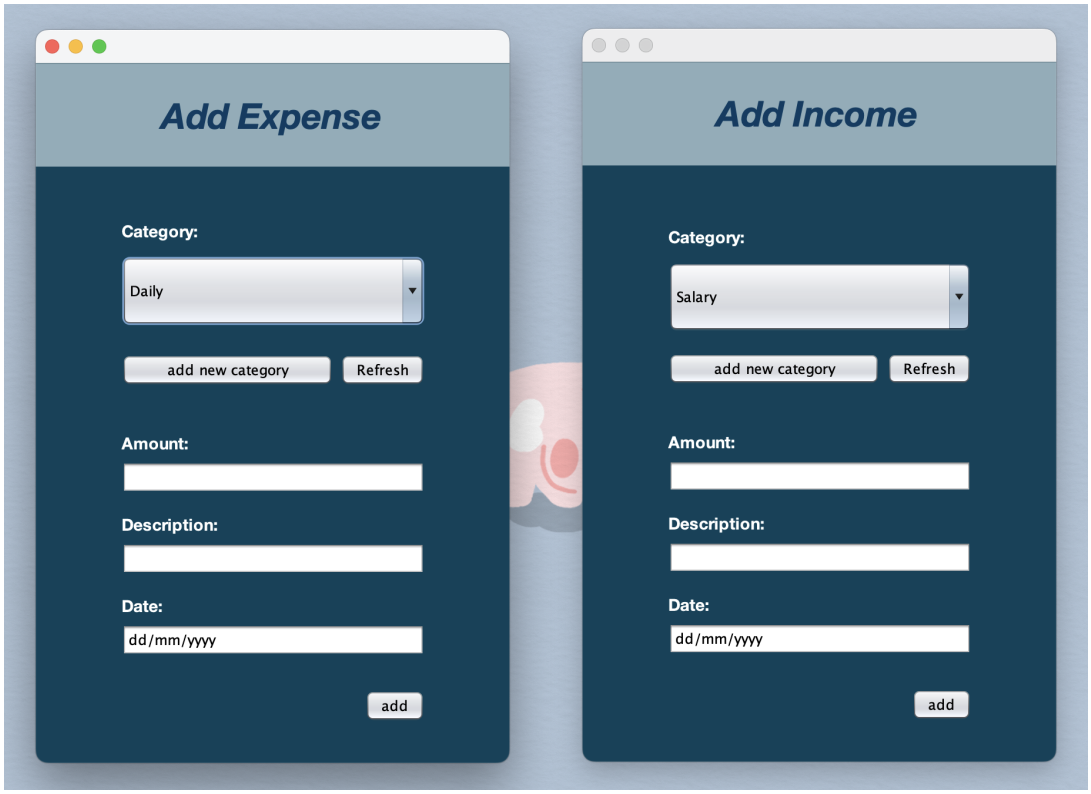
1. UML Class Diagram

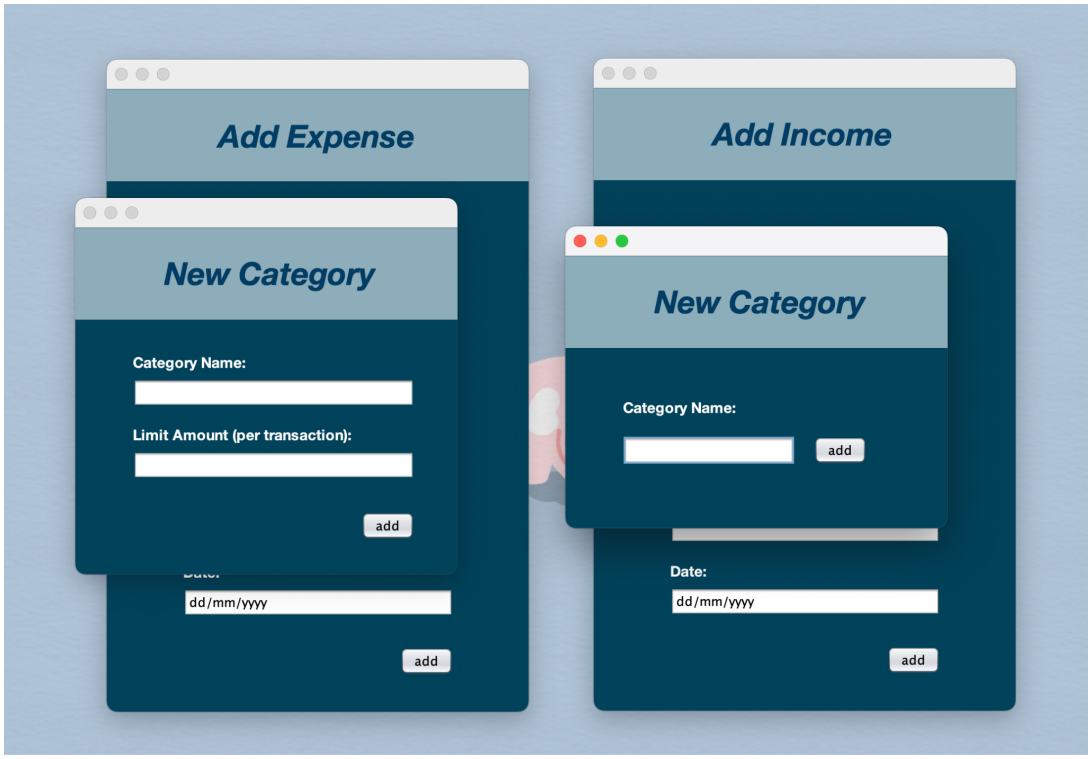


2. Screenshots of the Progam

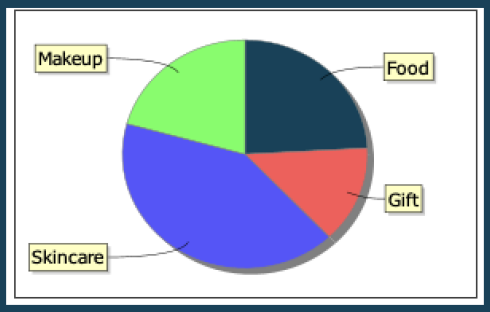





Add Expense
Category:
Daily
add new category
Refresh
Amount:
Description:
Date:
dd/mm/yyyy
add
Add Income
Category:
Salary
add new category
Refresh
Amount:
Description:
Date:
dd/mm/yyyy
add


Add Expense
New Category
Category Name:
Limit Amount (per transaction):
add
dd/mm/yyyy
add
Add Income
New Category
Category Name:
add
Date:
dd/mm/yyyy
add

Transaction Details



Detailed Information

Type: Expense
Category: Gift
Amount: 2000
Description:
Purchased a thoughtful gift

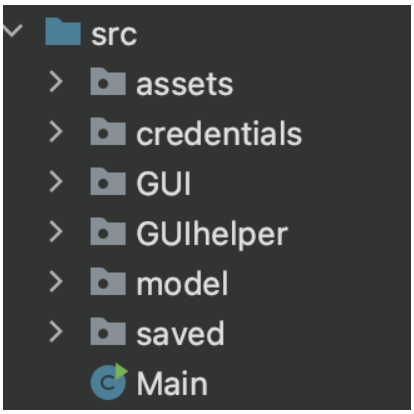
From: mm/dd/yyyy **To:** mm/dd/yyyy Search Refresh

Type	Category	Amount	Description	Date
Expense	Food	5000	Bought groceries	2023-06-01
Expense	Gift	3000	Purchased a birthda...	2023-06-02
Expense	Skincare	4000	Invested in skincare...	2023-06-03
Expense	Food	3500	Enjoyed a meal out ...	2023-06-04
Expense	Gift	2000	Purchased a though...	2023-06-05
Expense	Skincare	4500	Restocked essential...	2023-06-06
Income	Salary	5000	Received income fro...	2023-06-07
Income	Bonus	2000	Received a test pay...	2023-06-08
Income	Salary	2000	Received testing inc...	2023-06-09
Income	Salary	2000	Received testing inc...	2023-06-09
Income	Salary	10000	Received testing inc...	2023-06-09
Income	Bonus	2000	Received another te...	2023-06-10
Income	Bonus	20000	Received another te...	2023-06-10
Income	Salary	1000	Income from an unk...	2023-06-11
Expense	Makeup	3000	Purchased makeup ...	2023-06-12
Expense	Skincare	6000	Invested in skincare...	2023-06-13
Income	Salary	8000	Received income fro...	2023-06-14
Income	Gift	1000	Received a gift	2023-07-10

C. Code Design and Explanation

1. Project Structure

The main src folder has 6 packages and a “Main” class which acts as the main file to run in order to run the whole program. The first package “assets” contains the png file of the logo.



2. Credentials

The first class in the “credentials” package is the “Account” class.

```
package credentials;

import java.io.*;

public class Account implements Serializable {
  private static String name;
  private String email;
  private String password;
  private String hashedPassword;
  private String salt;

  public Account(String name, String email, String password) {
     this.name = name;
     this.email = email;
     this.password = password;
  }

  public Account(String name, String email, String hashedPassword,
String salt) {
     this.name = name;
     this.email = email;
```

```
        this.hashedPassword = hashedPassword;
        this.salt = salt;
    }

    public static String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public String getPassword() {
        return password;
    }
    public String getHashedPassword() {
        return hashedPassword;
    }
    public String getSalt() {
        return salt;
    }
```

This class represents an account and contains instance variables to store the account's name, email, password, hashed password, and salt. It provides getter methods to retrieve the values of these variables and a setter method to update the name. The class also implements the Serializable interface, allowing instances of the class to be serialized and deserialized.

```
@Override
    public String toString() {
        return getName() + " " + getEmail() + " " + getPassword() +
"\n";
    }
    public String toStringSecure() {
        return getName() + " " + getEmail() + " " +
getHashedPassword() + " " + getSalt() + "\n";
    }

    public void writeToFileSecure() {
        try {
            Writer output = null;
            String text = this.toStringSecure();
            File file = new File("src/saved/account.txt");
            output = new PrintWriter(new FileWriter(file, true));
            output.append(text);
```

```
            output.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The `toString()` method returns a string with the account's name, email, and password. The `toStringSecure()` method includes additional security-related information such as the hashed password and salt. The `writeToFileSecure()` method writes the secure string representation to a file named "account.txt" in the "src/saved" directory.

The second class in the package is the “SHA3” class.

```
package credentials;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

public class SHA3 {
```

This class includes methods for generating secure hashes using the SHA-3 cryptographic hash function and working with random salts.

```
// Hashes the plaintext using the SHA3-256 algorithm and returns
the hash as a byte array.
public static byte[] hash(String plaintext) {
    // Specify the hashing algorithm as SHA3-256.
    String algorithm = "SHA3-256";

    MessageDigest digest = null;

    try {
        // Create a MessageDigest object with the specified algorithm.
        digest = MessageDigest.getInstance(algorithm);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    // Convert the plaintext to bytes and hash it using the digest.
    final byte[] hashBytes =
digest.digest(plaintext.getBytes(StandardCharsets.UTF_8));

    return hashBytes;
```

```
}
```

The first method is named “hash” which takes a plaintext string as input. It uses the SHA3-256 algorithm to hash the plaintext and returns the hash as a byte array. The method creates a MessageDigest object with the specified algorithm and converts the plaintext to bytes. It then hashes the byte representation of the plaintext using the digest method of the MessageDigest object. Finally, it returns the resulting hash as a byte array.

```
// Generates a random salt (nonce) of the specified size and
returns it as a hexadecimal string.
public static String getSalt(int nonceSize) {
  // Create a byte array to store the salt.
  byte[] nonce = new byte[nonceSize];

  // Generate random bytes and store them in the nonce array.
  new SecureRandom().nextBytes(nonce);

  // Convert the nonce to a hexadecimal string representation.
  String nonceString = bytesToHex(nonce);

  return nonceString.toString();
}
```

This next method “getSalt” generates a random salt (nonce) of a specified size and returns it as a hexadecimal string. It uses a secure random number generator to generate random bytes, which are then converted into a hexadecimal string representation.

```
// Converts a byte array to a hexadecimal string representation.
public static String bytesToHex(byte[] bytes) {
  StringBuilder hexString = new StringBuilder(2 * bytes.length);

  // Iterate over each byte in the array and convert it to a
two-digit hexadecimal representation.
  for (int i = 0; i < bytes.length; i++) {
     String hex = Integer.toHexString(0xff & bytes[i]);

     // If the hexadecimal representation has only one digit, add a
leading zero.
     if (hex.length() == 1) {
        hexString.append('0');
     }

     // Append the hexadecimal representation to the hexString.
     hexString.append(hex);
```

```
    }

    return hexString.toString();
}
```

The last method “bytesToHex” converts a byte array to a hexadecimal string representation by iterating over each byte, converting it to a two-digit hexadecimal representation, and appending it to a StringBuilder. The resulting hexadecimal string is then returned.

3. GUI

```
package GUI;

import GUIhelper.*;
import javax.swing.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.*;
import model.*;
import model.category.*;
import model.filehandling.*;

public class AddExpensePage extends JFrame implements
ErrorHandling{

    public AddExpensePage() {
        initComponents();
    }
    @SuppressWarnings("unchecked")

    // Components stored here
    private void initComponents()
```

The AddExpensePage class is a graphical user interface (GUI) page for adding expenses. It extends the JFrame class and includes methods for initializing and managing GUI components. It imports various packages related to GUI operations, date and time handling, utility classes, and the application model. The class provides a constructor and a method for initializing GUI components, but the specific details of the components and their configurations are not provided in the code snippet. The AddExpensePage class serves as an interface for users to input and manage expense-related information in the application.

```
@Override
public void handleAmtError(Exception e) {
```

```
    JOptionPane.showMessageDialog(this, "Invalid amount input! Do
not use periods.", "Error", JOptionPane.ERROR_MESSAGE);
}

@Override
public void handleDateError(Exception e) {
    JOptionPane.showMessageDialog(this, "Invalid date", "Error",
JOptionPane.ERROR_MESSAGE);
}
```

The code snippet demonstrates two methods, handleAmtError and handleDateError, which are used for handling errors related to amount input and date input, respectively, in the AddExpensePage class. These methods display error messages using JOptionPane.showMessageDialog to inform the user about the specific error encountered. The handleAmtError method informs the user about an invalid amount input that should not contain periods, while the handleDateError method notifies the user about an invalid date input. Both methods display error message dialogs with appropriate error icons to provide visual cues for the user.

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent
evt) {
   // Add button
   try {
       // Retrieve the values from the text fields
       String type = "Expense";
                                        String     categoryName     =
jComboBox1.getSelectedItem().toString();
       int amount = Integer.parseInt(jTextField1.getText());
       String description = jTextField2.getText();
           String dateString = jTextField3.getText(); // Assuming
jTextField3 contains the date in the format dd/mm/yyyy
                                    DateTimeFormatter    formatter    =
DateTimeFormatter.ofPattern("dd/MM/yyyy");
       LocalDate date = LocalDate.parse(dateString, formatter);

       // find category by name
                       TransactionFileHandler   fileHandler   =   new
TransactionFileHandler();
                                           Category     category     =
fileHandler.findCategoryByName(categoryName);

       if (category instanceof ExpenseCategory) {
               ExpenseCategory expenseCategory = (ExpenseCategory)
category;

           // Retrieve the budget limit from the ExpenseCategory
```

```
            int limit = expenseCategory.getBudgetLimit();

            // Check if the amount exceeds the limit
            if (amount > limit) {
                 JOptionPane.showMessageDialog(this, "Amount exceeds
the limit.", "Error", JOptionPane.ERROR_MESSAGE);
                      return; // Exit the method to prevent further
execution
            }
        }

          Transaction transaction = new Transaction(type, category,
amount, description, date);

        // Load existing transactions
        List<Transaction> transactions = fileHandler.load();

        // Add the new transaction
        transactions.add(transaction);

        // Save the updated transactions
        fileHandler.save(transactions);

        // Clear input fields
        jTextField1.setText("");
        jTextField2.setText("");

                    JOptionPane.showMessageDialog(this,   "Transaction
successfully added!", "Success", JOptionPane.INFORMATION_MESSAGE);
    } catch (NumberFormatException e) {
        // Call the handleError method
        handleAmtError(e);
    } catch (DateTimeParseException e) {
        handleDateError(e);
    }
}
```

The code snippet handles the event when the "Add" button is clicked on the "AddExpensePage" GUI. It retrieves the input values from text fields, such as type, category name, amount, description, and date. It then performs various operations, such as finding the corresponding category, checking if the amount exceeds the budget limit (for expense categories), creating a new transaction object, loading existing transactions, adding the new transaction, and finally saving the updated transactions to a file. It also handles exceptions for invalid amount and date inputs by displaying error messages using JOptionPane.

```
private    void    jButton3ActionPerformed(java.awt.event.ActionEvent
evt) {
    // Add category
    new NewExCategoryPage().setVisible(true);
}

private    void    jButton4ActionPerformed(java.awt.event.ActionEvent
evt) {
    // Refresh
    Reload.refreshPage(this, AddExpensePage.class.getName());
}
```

This next code snippet shows the action performed when the "Add category" button (jButton3) or "Refresh" button (jButton4) is clicked on the GUI. For jButton3ActionPerformed, it creates a new instance of the "NewExCategoryPage" and sets it as visible, allowing the user to add a new expense category. For jButton4ActionPerformed, it calls the "refreshPage" method from the "Reload" class to refresh the current page (AddExpensePage) by reloading its contents.

```
package GUI;

import GUIhelper.ErrorHandling;
import GUIhelper.Reload;
import javax.swing.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.ArrayList;
import java.util.List;
import model.category.Category;
import model.category.IncomeCategory;
import model.filehandling.IncomeCategoryFileHandler;
import model.Transaction;
import model.filehandling.TransactionFileHandler;

public    class    AddIncomePage    extends    JFrame    implements
ErrorHandling{

    public AddIncomePage() {
        initComponents();
    }
```

The next class is called "AddIncomePage". This class extends the JFrame class and implements the ErrorHandling interface. The constructor "AddIncomePage()" initializes the class and calls the "initComponents()" method to set up the GUI components. The class includes import statements for the necessary dependencies,

such as ErrorHandling, Reload, Swing components (JFrame, JOptionPane), LocalDate, DateTimeFormatter, DateTimeParseException, List, and various model classes and file handlers. Overall, this class represents a GUI page for adding income transactions.

```
@Override
public void handleAmtError(Exception e) {
    JOptionPane.showMessageDialog(this, "Invalid amount input! Do
not use periods.", "Error", JOptionPane.ERROR_MESSAGE);
}

@Override
public void handleDateError(Exception e) {
     JOptionPane.showMessageDialog(this, "Invalid date", "Error",
JOptionPane.ERROR_MESSAGE);
}
```

This code overrides two methods from the ErrorHandling interface: "handleAmtError" and "handleDateError". The "handleAmtError" method is responsible for handling an exception related to invalid amount input. It displays an error message dialog using JOptionPane, indicating that the amount input is invalid and that periods should not be used. The "handleDateError" method handles an exception related to an invalid date. It displays an error message dialog using JOptionPane, indicating that the date input is invalid. Both methods provide visual feedback to the user when an error occurs during amount or date input in the GUI.

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent
evt) {//GEN-FIRST:event_jButton2ActionPerformed
   // Sign Up button
   try {
       // Retrieve the values from the text fields
       String type = "Income";
                                        String     categoryName     =
jComboBox1.getSelectedItem().toString();
       int amount = Integer.parseInt(jTextField1.getText());
       String description = jTextField2.getText();
           String dateString = jTextField3.getText(); // Assuming
jTextField3 contains the date in the format dd/mm/yyyy
                                    DateTimeFormatter    formatter    =
DateTimeFormatter.ofPattern("dd/MM/yyyy");
       LocalDate date = LocalDate.parse(dateString, formatter);

       // Create a new Transaction object
                        TransactionFileHandler   fileHandler   =   new
TransactionFileHandler();
```

```
                                        Category      category      =
fileHandler.findCategoryByName(categoryName);

         Transaction transaction = new Transaction(type, category,
amount, description, date);

   // Load existing transactions
       List<Transaction> transactions = fileHandler.load();

       // Add the new transaction
       transactions.add(transaction);

       // Save the updated transactions
       fileHandler.save(transactions);

       // Clear input fields
       jTextField1.setText("");
       jTextField2.setText("");

                JOptionPane.showMessageDialog(this,   "Transaction
successfully added!", "Success", JOptionPane.INFORMATION_MESSAGE);
   } catch (NumberFormatException e) {
       // Call the handleError method
       handleAmtError(e);
   } catch (DateTimeParseException e) {
       handleDateError(e);
   }
}
```

The jButton2ActionPerformed method is triggered when the "Sign Up" button is clicked in the AddIncomePage GUI. It performs the task of retrieving the input values from the text fields, such as the transaction type, category name, amount, description, and date. It then creates a new Transaction object with the retrieved values. The existing transactions are loaded from the file handler, and the new transaction is added to the list. The updated list of transactions is saved back to the file. If any exceptions occur during the process, such as invalid amount input or date format, the appropriate error message is displayed using JOptionPane. Finally, the input fields are cleared, and a success message is displayed to indicate that the transaction was successfully added.

```
private    void    jButton3ActionPerformed(java.awt.event.ActionEvent
evt) {
   // Add category
   new NewInCategoryPage().setVisible(true);
}
```

```
private    void    jButton1ActionPerformed(java.awt.event.ActionEvent
evt) {
    // Refresh
    Reload.refreshPage(this, AddIncomePage.class.getName());
}
```

The jButton3ActionPerformed method is triggered when the "Add category" button is clicked in the AddIncomePage GUI. It opens a new NewInCategoryPage window, allowing the user to add a new income category. The jButton1ActionPerformed method is triggered when the "Refresh" button is clicked in the AddIncomePage GUI. It reloads the current page (AddIncomePage) using the Reload class, refreshing the content and resetting any changes made.

```
package GUI;

import GUIhelper.Reload;
import credentials.Account;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.util.List;
import model.Transaction;
import model.filehandling.BalanceFileHandler;
import model.filehandling.TransactionFileHandler;

public class Dashboard extends JFrame {

    public Dashboard() {
        initComponents();
    }
```

The Dashboard class represents the main graphical user interface (GUI) of the application, where users can view their account information and transaction history. It utilizes various components such as buttons, labels, and tables to display and interact with the data. The class is responsible for loading and saving transaction data and balance details using the TransactionFileHandler and BalanceFileHandler classes. It also includes methods to update and refresh the displayed information. Overall, the Dashboard class serves as the central interface for users to manage and monitor their financial activities.

```
private void LogoutButtonActionPerformed(java.awt.event.ActionEvent
evt) {
    // Display a confirmation dialog
     int choice = JOptionPane.showConfirmDialog(this, "Are you sure
you want to logout?", "Confirm Logout", JOptionPane.YES_NO_OPTION);
```

```
    // Check the user's choice
    if (choice == JOptionPane.YES_OPTION) {
        // User confirmed logout
        new LoginPage().setVisible(true);
        dispose();
    } else {
        // User chose to cancel, do nothing
    }
}
```

The LogoutButtonActionPerformed method is an event handler for the logout button in the Dashboard class. When the button is clicked, a confirmation dialog is displayed asking the user to confirm if they want to logout. If the user selects "Yes," indicating confirmation, the method creates a new instance of the LoginPage class to display the login page and disposes of the current Dashboard frame, effectively logging out the user. If the user selects "No" or cancels the dialog, the method simply returns without performing any further actions. This method ensures that the user is prompted to confirm their logout decision before proceeding.

```
private void IncomeButtonActionPerformed(java.awt.event.ActionEvent
evt) {
    new AddIncomePage().setVisible(true);
}

private                                                    void
ExpenseButtonActionPerformed(java.awt.event.ActionEvent evt) {
    new AddExpensePage().setVisible(true);
}

private                                                    void
TransactionDetailsButtonActionPerformed(java.awt.event.ActionEvent
evt) {
    new TransactionDetailsPage().setVisible(true);
}

private    void    jButton1ActionPerformed(java.awt.event.ActionEvent
evt) {
    // Refresh
    Reload.refreshPage(this, Dashboard.class.getName());
}
```

The IncomeButtonActionPerformed method opens a new instance of the AddIncomePage GUI form when the income button is clicked. Similarly, the

ExpenseButtonActionPerformed method opens a new instance of the AddExpensePage GUI form when the expense button is clicked. The TransactionDetailsButtonActionPerformed method opens a new instance of the TransactionDetailsPage GUI form when the transaction details button is clicked. Lastly, the jButton1ActionPerformed method is responsible for refreshing the current page by calling the Refresh.refreshPage method, passing the current instance of the Dashboard class and its name.

```
package GUI;

import credentials.Account;
import credentials.SHA3;
import javax.swing.*;
import java.io.*;
import java.util.regex.Pattern;

public class LoginPage extends JFrame {

   public LoginPage() {
       initComponents();
   }
```

The LoginPage class represents a graphical user interface (GUI) for a login page. It provides a user interface for entering login credentials and authenticating the user. The class handles the visual layout and components of the login page, and it likely includes functionality for user account management, such as creating new accounts and verifying login credentials.

```
private    void    jButton2ActionPerformed(java.awt.event.ActionEvent
evt) {
   // Sign up button
   // Check if the "account.txt" file contains data
   File file = new File("account.txt");
   if (file.exists() && file.length() > 0) {
           JOptionPane.showMessageDialog(this, "An account already
exists.", "Error", JOptionPane.ERROR_MESSAGE);
   } else {
       // If the file doesn't contain data, open the SignUpPage
       new SignUpPage().setVisible(true);
       dispose();
   }
}
```

The jButton2ActionPerformed method is the event handler for the "Sign up" button in the login page GUI. When the button is clicked, the method checks if the "account.txt" file exists and if it contains any data. If the file exists and has data, it displays an error message stating that an account already exists. If the file doesn't exist or is empty, it opens the SignUpPage GUI, allowing the user to create a new account, and disposes of the current login page GUI.

```
private    void    jButton3ActionPerformed(java.awt.event.ActionEvent
evt) {
   // Login button
   String emailText;
   String passText;
   emailText = jTextField1.getText();
   passText = jPasswordField1.getText();

      // Checking if a field has been left blank and acting
accordingly

   if (emailText.isBlank() || passText.isBlank()) {
            jTextField1.setText("");
            jPasswordField1.setText("");
                JOptionPane.showMessageDialog(this, "All fields are
required", "Error!", JOptionPane.ERROR_MESSAGE);

             // Checking if the email address entered is valid and
acting accordingly

   } else if (!isValid(emailText)) {
            jTextField1.setText("");
            jTextField1.setText("");
            jPasswordField1.setText("");
              JOptionPane.showMessageDialog(this, "The email address
you have entered is invalid", "Error!",
                              JOptionPane.ERROR_MESSAGE);

                  // Checking if the email address and password
combination is correct and acting
            // accordingly

   } else if (isCorrect(emailText, passText)) {

             // If all of the above conditions check out, the user
will be taken to the main
            // application screen

             JOptionPane.showMessageDialog(this, "Login Successful",
"Success!", JOptionPane.INFORMATION_MESSAGE);
```

```
                                        Account acc = new
Account(getAccountHolderName(emailText, passText), emailText,
passText);

            new Dashboard().setVisible(true);
            dispose();
    } else {

            // Otherwise, an error message will be displayed

            jPasswordField1.setText("");
            JOptionPane.showMessageDialog(this, "Invalid username or
password", "Error!",
                                JOptionPane.ERROR_MESSAGE);
    }
}
```

The jButton3ActionPerformed method handles the action when the "Login" button is clicked in the login page GUI. It retrieves the email and password entered by the user and performs several checks. If any of the fields are left blank, it clears the fields and displays an error message indicating that all fields are required. If the email address entered is invalid, it clears the email and password fields and displays an error message stating that the email address is invalid. If the email and password combination is correct, it displays a success message, creates an Account object with the account holder's name, email, and password, and opens the dashboard page. If the email and password combination is incorrect, it clears the password field and displays an error message indicating that the username or password is invalid.

```
private                                                    void
jRadioButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    if (jRadioButton1.isSelected()) {
  jPasswordField1.setEchoChar((char) 0);
} else {
  jPasswordField1.setEchoChar('*');
}
}
```

The jRadioButton1ActionPerformed method handles the action when the radio button is selected or deselected in the GUI. If the radio button is selected, it sets the echo character of the jPasswordField1 component to 0, effectively making the password visible. This allows the user to see the entered password. On the other hand, if the radio button is deselected, it sets the echo character of the

jPasswordField1 component back to '*', which is the default behavior for password fields. This hides the password characters to ensure security.

```
public static boolean isValid(String email) {
   // Regular expression to validate email format
          String    emailRegex    =    "^[a-zA-Z0-9_+&*-]+(?:\\."    +
"[a-zA-Z0-9_+&*-]+)*@" + "(?:[a-zA-Z0-9-]+\\.)+[a-z"
           + "A-Z]{2,7}$";

   Pattern pat = Pattern.compile(emailRegex);
   if (email == null)
       return false;
   return pat.matcher(email).matches();
}
```

The isValid method is a utility method that checks if an email address is valid. It takes an email address as input and uses a regular expression to validate its format. The regular expression pattern emailRegex follows the standard email format, allowing for alphanumeric characters, dots, underscores, and certain special characters in the local part of the email address. It also checks the domain part of the email address to ensure it has at least two characters and follows the standard domain format. The method returns true if the email address is valid and false otherwise. It also handles the case where the input email is null and returns false in that case.

```
private static boolean isCorrect(String email, String password) {
   BufferedReader read;
   try {
       // Open the file for reading
                              read    =    new    BufferedReader(new
FileReader("src/saved/account.txt"));
       String line;
       while ((line = read.readLine()) != null) {
           // Split the line into separate values
           String[] var = line.split(" ");
             // Check if the email and password match the values in
the file
                                    if    (var[1].equals(email)    &&
var[2].equals(SHA3.bytesToHex(SHA3.hash(var[3] + password)))) {
               // Create an Account object and close the file
                  Account acc = new Account(var[0], var[1], var[2],
var[3]);
               read.close();
               return true;
           }
       }
```

```
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return false;
}
```

The isCorrect method checks if the provided email and password match the values stored in a file. It reads the file line by line, compares the email and password with the stored values, and returns true if there is a match. Otherwise, it returns false.

```
private static String getAccountHolderName(String email, String
password) {
    BufferedReader read;
    try {
        // Open the file for reading
                                read    =    new    BufferedReader(new
FileReader("src/saved/account.txt"));
        String line;
        while ((line = read.readLine()) != null) {
            // Split the line into separate values
            String[] var = line.split(" ");
              // Check if the email and password match the values in
the file
                                        if    (var[1].equals(email)    &&
var[2].equals(SHA3.bytesToHex(SHA3.hash(var[3]+password)))) {
                    // Close the file and return the account holder's
name
                read.close();
                return var[0];
            }
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

The getAccountHolderName method retrieves the account holder's name based on the provided email and password. It reads the contents of a file line by line, splits each line into separate values, and compares the email and password with the stored values. If a match is found, it returns the account holder's name. Otherwise, it returns null.

```
package GUI;

import javax.swing.*;
import java.util.List;
import model.category.ExpenseCategory;
import model.filehandling.ExpenseCategoryFileHandler;

public class NewExCategoryPage extends JFrame {

   public NewExCategoryPage() {
       initComponents();
   }
```

The NewExCategoryPage class is a GUI page that allows users to create a new expense category. It extends the JFrame class and provides a graphical interface for users to input information about the category. The class handles user actions such as saving or canceling the creation of the category. It may interact with other components or files, such as the ExpenseCategoryFileHandler, to handle the storage and retrieval of expense categories. Overall, it facilitates the creation of new expense categories through a user-friendly interface.

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent
evt) {
   //Retrieve the category name from the JTextField
   String categoryName = CategoryName.getText();
   String budgetLimitText = BudgetLimit.getText();
   double budgetLimitDouble = Double.parseDouble(budgetLimitText);
   int budgetLimit = (int) budgetLimitDouble;

   // Create a new Category object
     ExpenseCategory category = new ExpenseCategory(categoryName,
budgetLimit);
             ExpenseCategoryFileHandler fileHandler = new
ExpenseCategoryFileHandler();

   // Load existing categories from a file if needed
   List<ExpenseCategory> categories = fileHandler.load();

   // Add the new category to the list
   categories.add(category);

   // Save the updated categories to a file using the save method
   fileHandler.save(categories);

   // Clear the JTextField after saving
   CategoryName.setText("");
   BudgetLimit.setText("");
```

```
    // Show a JOptionPane message dialog indicating successful
addition
          JOptionPane.showMessageDialog(null,    "Category    added
successfully.");
}
```

The jButton2ActionPerformed method in the NewExCategoryPage class handles the addition of a new expense category. It retrieves the category name and budget limit from the input fields, creates a new ExpenseCategory object, saves it to a file using a file handler, and provides a success message to the user.

```
package GUI;

import javax.swing.*;
import java.util.List;
import model.category.IncomeCategory;
import model.filehandling.IncomeCategoryFileHandler;

public class NewInCategoryPage extends JFrame {

   public NewInCategoryPage() {
       initComponents();
   }
```

The NewInCategoryPage class is a graphical user interface (GUI) window that allows users to create new income categories. It extends the JFrame class and contains methods for initializing the components of the window.

```
private    void    jButton2ActionPerformed(java.awt.event.ActionEvent
evt) {
   // Add button
   // Retrieve the category name from the JTextField
   String categoryName = jTextField1.getText();

   // Create a new Category object
   IncomeCategory category = new IncomeCategory(categoryName);
            IncomeCategoryFileHandler    fileHandler    =    new
IncomeCategoryFileHandler();

   // Load existing categories from a file if needed
   List<IncomeCategory> categories = fileHandler.load();

   // Add the new category to the list
   categories.add(category);

   // Save the updated categories to a file using the save method
```

```
    fileHandler.save(categories);

    // Clear the JTextField after saving
    jTextField1.setText("");

      // Show a JOptionPane message dialog indicating successful
addition
            JOptionPane.showMessageDialog(null,    "Category    added
successfully.");

}
```

The jButton2ActionPerformed method in the NewInCategoryPage class handles the event when the user clicks the "Add" button. It retrieves the category name from the jTextField1 input field and creates a new IncomeCategory object with the retrieved name. It then uses an instance of the IncomeCategoryFileHandler class to handle file operations. Existing income categories are loaded from a file, if any, into a list. The new category is added to the list, and the updated list is saved back to the file. The jTextField1 input field is cleared, and a JOptionPane message dialog is displayed to indicate that the category was added successfully.

```
package GUI;

import credentials.Account;
import credentials.SHA3;
import javax.swing.*;
import java.util.*;
import java.util.regex.*;

public class SignUpPage extends javax.swing.JFrame {

    public SignUpPage() {
        initComponents();
    }
```

The SignUpPage class represents the graphical user interface for the sign-up page in a user registration system. It extends the javax.swing.JFrame class to create a window for user interaction. The class includes methods for initializing and managing the components of the sign-up page.

```
private    void    jButton2ActionPerformed(java.awt.event.ActionEvent
evt) {
    // Login button
    new LoginPage().setVisible(true);
    dispose();
}
```

The jButton2ActionPerformed method is an event handler for the login button in the sign-up page. When the button is clicked, it creates an instance of the LoginPage class and makes it visible by calling the setVisible(true) method. It also disposes of the current sign-up page window by calling the dispose() method, effectively closing it. This action allows the user to navigate back to the login page from the sign-up page.

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent
evt) {
    // Sign Up button
       // Checking if a field has been left blank and acting
accordingly

if (nameField.getText().isBlank() || emailField.getText().isBlank()
                    ||    jPasswordField1.getText().isBlank()    ||
jPasswordField2.getText().isBlank()) {
   JOptionPane.showMessageDialog(this, "All fields are required",
"Error!", JOptionPane.ERROR_MESSAGE);
}

// Checking if the user's name contains any numbers and acting
accordingly

else if (nameField.getText().matches(".*\\d.*")) {
    JOptionPane.showMessageDialog(this, "The name field must not
contain any numbers", "Error!",
        JOptionPane.ERROR_MESSAGE);
}

// Checking if the email address entered is valid and acting
accordingly

else if (!isValid(emailField.getText())) {
   JOptionPane.showMessageDialog(this, "The email address you have
entered is invalid", "Error!",
        JOptionPane.ERROR_MESSAGE);
}

// Checking if the password entered is strong enough and acting
accordingly

else if (!isStrong(jPasswordField1.getText())) {

  JOptionPane.showMessageDialog(this,
           "Your password must contain at least 8 characters, an
uppercase letter, a lowercase letter, a number and a special
character",
```

```
        "Error!", JOptionPane.ERROR_MESSAGE);
}

// Checking if the password entered in the password field is the
same as the
// password entered in the confirm password field and acting
accordingly.

else                                                    if
(!jPasswordField2.getText().equals(jPasswordField1.getText())) {
  jPasswordField1.setText("");
  jPasswordField2.setText("");
   JOptionPane.showMessageDialog(this, "The passwords don't match",
"Error!", JOptionPane.ERROR_MESSAGE);
} else {

   // If all of the above validations check out, a new account is
created and saved
  // in a text file

  String passSalt = SHA3.getSalt(16);
     String   hashedPass   =   SHA3.bytesToHex(SHA3.hash(passSalt   +
jPasswordField1.getText()));
         Account     acc     =     new     Account(nameField.getText(),
emailField.getText(), hashedPass, passSalt);
  acc.writeToFileSecure();
    JOptionPane.showMessageDialog(this,  "Your  account  has  been
created successfully", "Account Creation",
        JOptionPane.INFORMATION_MESSAGE);
  new LoginPage().setVisible(true);
  dispose();
}
}
```

The "Sign Up" button's action performs a series of validations to ensure that all required fields are filled correctly before creating a new account. It checks if any of the fields (name, email, password, confirm password) are left blank and displays an error message if so. It also verifies that the name field does not contain any numbers and that the email address is valid using a regular expression pattern. Additionally, it checks if the password meets the criteria of having at least 8 characters, an uppercase letter, a lowercase letter, a number, and a special character. If any of these validations fail, the user is prompted with an appropriate error message. If all the validations pass, a new account is created with the provided information, the account details are securely saved in a text file, and a success message is displayed. The user is then redirected to the login page, and the current sign-up page is closed.

```
private void jRadioButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
    // Show password
    if (jRadioButton1.isSelected()) {
  jPasswordField1.setEchoChar((char) 0);
  jPasswordField2.setEchoChar((char) 0);
} else {
  jPasswordField1.setEchoChar('*');
  jPasswordField2.setEchoChar('*');
}
}
```

The "Show password" action associated with the radio button toggles the visibility of the password fields. When the radio button is selected, it sets the echo character of both password fields to 0, which makes the entered characters visible. This allows the user to see the actual content of the password fields. Conversely, when the radio button is deselected, the echo character is set back to '*' for both password fields, which hides the entered characters by displaying them as asterisks.

```
public static boolean isValid(String email) {
      // Regular expression to validate email format
  String emailRegex = "^[a-zA-Z0-9_+&*-]+(?:\\." + "[a-zA-Z0-9_+&*-]+)*@"
+ "(?:[a-zA-Z0-9-]+\\.)+[a-z"
        + "A-Z]{2,7}$";

  Pattern pat = Pattern.compile(emailRegex);
  if (email == null)
     return false;
  return pat.matcher(email).matches();
}
```

The method isValid is used to validate the format of an email address. It takes an email string as input and uses a regular expression to check if the email matches the specified pattern. The regular expression ensures that the email follows a valid format, including the presence of alphanumeric characters, special characters, and a correct domain name format. The method returns true if the email is valid and false otherwise.

```
public static boolean isStrong(String input) {
   int n = input.length();
     boolean hasLower = false, hasUpper = false, hasDigit = false,
specialChar = false;
   // Set of special characters to check for
   Set<Character> set = new HashSet<Character>(
            Arrays.asList('!', '@', '#', '$', '%', '^', '&', '*', '(',
')', '-', '+'));
```

```
    // Iterate over each character in the input string
    for (char i : input.toCharArray()) {
        if (Character.isLowerCase(i))
            hasLower = true;
        if (Character.isUpperCase(i))
            hasUpper = true;
        if (Character.isDigit(i))
            hasDigit = true;
        if (set.contains(i))
            specialChar = true;
    }

    // Check if the input meets the strong password criteria
    if (hasDigit && hasLower && hasUpper && specialChar && (n >= 8)) {
        return true;
    }
    return false;
}
```

The method isStrong is used to check if a given input string represents a strong password. It examines the input string character by character and checks for the presence of specific criteria: at least one lowercase letter, at least one uppercase letter, at least one digit, and at least one special character from a predefined set. Additionally, it checks if the length of the input string is at least 8 characters. If all of these criteria are met, the method returns true, indicating that the password is strong. Otherwise, it returns false.

```
package GUI;

import GUIhelper.Reload;
import java.awt.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.List;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.table.DefaultTableModel;
import model.filehandling.TransactionFileHandler;
import model.Transaction;
import model.data.DataManager;
import org.jfree.chart.*;
import org.jfree.chart.plot.PiePlot;
import org.jfree.data.general.DefaultPieDataset;

public class TransactionDetailsPage extends javax.swing.JFrame {

    public TransactionDetailsPage() {
```

```
        initComponents();
        showPieChart();
    }
```

The TransactionDetailsPage class is a GUI window that displays transaction details. It extends javax.swing.JFrame and includes components for handling transaction data. The constructor initializes the GUI components and displays a pie chart using the showPieChart() method. It also implements a ListSelectionListener to respond to item selection changes. Overall, it provides a convenient way to view and analyze transactions.

```
private void searchButtonActionPerformed(java.awt.event.ActionEvent evt)
{
    // Search button
                              DateTimeFormatter          formatter          =
DateTimeFormatter.ofPattern("MM/dd/yyyy");
    String dateStartString = fromText.getText();
    LocalDate startTime = LocalDate.parse(dateStartString, formatter);
    String dateEndString = toText.getText();
    LocalDate endTime = LocalDate.parse(dateEndString, formatter);


    TransactionFileHandler fileHandler = new TransactionFileHandler();
    List<Transaction> defaultTransactions = fileHandler.load();

    DataManager dataManager = new DataManager(defaultTransactions);
                        List<Transaction>       filteredTransactions       =
dataManager.getFilteredTransactions(startTime, endTime);


    // Clear the current data in the table
    DefaultTableModel tableModel = (DefaultTableModel) jTable1.getModel();
    tableModel.setRowCount(0);

    // Add the filtered transactions to the table model
    for (Transaction transaction : filteredTransactions) {
        String[] dataRow = new String[5];
        dataRow[0] = transaction.getType();
        dataRow[1] = transaction.getCategory().getName();
        dataRow[2] = String.valueOf(transaction.getAmount());
        dataRow[3] = transaction.getDescription();
        dataRow[4] = transaction.getDate().toString();
        tableModel.addRow(dataRow);
    }

    // Refresh the table to display the updated data
```

```
    jTable1.repaint();
}
```

The searchButtonActionPerformed method handles the action when the search button is clicked. It retrieves the start and end dates from the corresponding text fields and parses them into LocalDate objects using a DateTimeFormatter. It then loads the default transactions from a file using a TransactionFileHandler and creates a DataManager object to manage the transaction data. The getFilteredTransactions method is called to obtain a list of transactions within the specified date range.

Next, the current data in the table is cleared by setting the row count of the table model to zero. The filtered transactions are then added to the table model row by row, populating the transaction details such as type, category, amount, description, and date. Finally, the table is refreshed to display the updated data.

```
private void refreshButtonActionPerformed(java.awt.event.ActionEvent evt)
{
    // Refresh
    Reload.refreshPage(this, TransactionDetailsPage.class.getName());
}
```

The refreshButtonActionPerformed method handles the action when the refresh button is clicked. It utilizes the Reload utility class to refresh the current page (TransactionDetailsPage) by reloading its contents. This ensures that any updates or changes made to the data or interface are reflected immediately.

```
public void showPieChart() {
// Create dataset
DefaultPieDataset dataset = new DefaultPieDataset();

// Load transactions from the file using TransactionFileHandler
TransactionFileHandler fileHandler = new TransactionFileHandler();
List<Transaction> transactions = fileHandler.load();

// Iterate over the transactions and add the ones with type "Expense" to
the dataset
for (Transaction transaction : transactions) {
    if (transaction.getType().equals("Expense")) {
        String category = transaction.getCategory().getName();
        double amount = transaction.getAmount();
        dataset.setValue(category, amount);
    }
}
```

The showPieChart method is responsible for displaying a pie chart representing the expenses in the TransactionDetailsPage. It starts by creating a DefaultPieDataset object to hold the data for the chart. Then, it loads the transactions from a file using the TransactionFileHandler. It iterates over the transactions and adds the ones with the type "Expense" to the dataset, using the category name as the key and the transaction amount as the value. This dataset will be used to create and display the pie chart.

```
// Create chart
JFreeChart pieChart = ChartFactory.createPieChart("", dataset, false,
true, false);

PiePlot piePlot = (PiePlot) pieChart.getPlot();

piePlot.setBackgroundPaint(Color.white); // Set the background color of
PiePlot

// Set custom colors for the pie slices
piePlot.setSectionPaint("Food", new Color(0, 66, 90)); // Set custom
color for Category 1

// Create chartPanel to display chart
ChartPanel pieChartPanel = new ChartPanel(pieChart);
pieChartPanel.setPreferredSize(new Dimension(300, 200)); // Set the
preferred size of the ChartPanel
jPanel3.removeAll();
jPanel3.add(pieChartPanel, BorderLayout.CENTER);
jPanel3.revalidate();
```

In this piece of code, a pie chart is created using JFreeChart library. The createPieChart method is called with an empty string for the title, the dataset containing expense categories and amounts, and some customization options. The resulting pie chart is then customized further by setting the background color of the plot to white and assigning custom colors to specific pie slices, such as the "Food" category.

A ChartPanel is created to display the pie chart. Its preferred size is set to 300x200 pixels. The jPanel3 component is cleared, and the pie chart panel is added to it using the BorderLayout.CENTER position. Finally, the jPanel3 is refreshed to reflect the changes.

Overall, this code segment generates a customized pie chart representing expense categories and their corresponding amounts, and displays it in the jPanel3 component.

4. GUIhelper

The package GUIhelper contains two files. The first one is the “ErrorHandling” Interface.

```
package GUIhelper;

public interface ErrorHandling {

   void handleAmtError(Exception e);

   void handleDateError(Exception e);
}
```

This interface has two methods. “handleAmtError” and “handleDateError”. It serves as a blueprint for classes that need to handle errors related to amounts and dates in a graphical user interface (GUI). Implementing classes must provide their own logic for handling these specific types of errors.

The second class is the “Reload” class.

```
package GUIhelper;

import java.awt.Window;

public class Reload {
   public static void refreshPage(Window window, String
windowClassName) {
       window.dispose(); // Close the current window

       try {
           Class<?> cls = Class.forName(windowClassName);
           Window newWindow = (Window)
cls.getDeclaredConstructor().newInstance();
           newWindow.setVisible(true);
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

This code defines a class called Reload with a method “refreshPage” that refreshes a GUI page by closing the current window and creating a new instance of a specified window class.

5. Model

```java
package model.category;

public abstract class Category {

    protected String name;

    public Category(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

The first class defines an abstract class called Category that represents a category. It has a name variable, a constructor to set the name, and getter/setter methods for the name. It serves as a base class for specific category implementations.

```java
package model.category;

public class ExpenseCategory extends Category {

    private int budgetLimit;

    public ExpenseCategory(String name, int budgetLimit) {
        super(name);
        this.budgetLimit = budgetLimit;
    }

    public int getBudgetLimit() {
        return budgetLimit;
    }

}
```

The next code is a class called “ExpenseCategory” that extends the “Category” class. It adds a “budgetLimit” variable and a constructor to set its value. The class represents an expense category with a budget limit.

```java
package model.category;

public class IncomeCategory extends Category {

    public IncomeCategory(String name) {
        super(name);
    }
}
```

This next class is a class called “IncomeCategory” that extends the “Category” class. It has a constructor to set the name of the income category. It represents a specific type of category related to income.

```java
package model.data;

import model.Transaction;
import java.time.LocalDate;
import java.util.List;

public class DataManager {

    // Constructor that initializes the DataManager with a list of
loaded transactions
    public DataManager(List<Transaction> loadedTransactions) {
        transactions = loadedTransactions; // Assign the loaded
transactions to the transactions variable
        dateFilter = new DateFilter();
    }

    private List<Transaction> transactions; // List to store the
transactions
    private DateFilter dateFilter; // DateFilter object to perform
filtering

    // Returns a list of transactions filtered by the specified
start and end dates
    public List<Transaction> getFilteredTransactions(LocalDate
startDate, LocalDate endDate) {
        // Call the filterByDate method of the dateFilter object to
filter the transactions
        List<Transaction> filteredTransactions =
dateFilter.filterByDate(transactions, startDate, endDate);
        return filteredTransactions;
    }
}
```

This next class “DataManager” is responsible for managing and filtering transactions. It has a constructor to initialize the class with a list of transactions

and provides a method to retrieve a filtered list of transactions based on specified start and end dates using a DateFilter object.

```
package model.data;

import model.Transaction;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class DateFilter {
```

This next class “DateFilter” is responsible for filtering transactions based on dates.

```
// Filters the list of transactions based on the specified start
and end dates
public List<Transaction> filterByDate(List<Transaction>
transactions, LocalDate startDate, LocalDate endDate) {
   LocalDate fromDate = startDate;
   LocalDate toDate = endDate;

   Predicate<Transaction> dateFilter = isInFilter(fromDate,
toDate);

   List<Transaction> filteredTransactions = new ArrayList<>();

   // Iterate through each transaction and check if it satisfies
the date filter
   for (Transaction transaction : transactions) {
       if (dateFilter.test(transaction)) {
           filteredTransactions.add(transaction);
       }
   }

   return filteredTransactions;
}
```

The first method is called “filterByDate” which filters a list of transactions based on a specified start and end date. It creates a Predicate to check if each transaction falls within the date range and adds the matching transactions to a new list that is returned as the result.

```
// Returns a predicate that checks if a transaction's date falls
within the specified range
```

```
    private Predicate<Transaction> isInFilter(LocalDate fromDate,
LocalDate toDate) {
        return transaction -> {
            // Assuming transaction.getDate() returns a String
representation of a date
            String dateString =
String.valueOf(transaction.getDate());
            LocalDate transactionDate = LocalDate.parse(dateString);

            // Check if the transaction date is equal to or after
the start date
            // and if it is equal to or before the end date
            return (transactionDate.isEqual(fromDate) ||
transactionDate.isAfter(fromDate))
                    && (transactionDate.isEqual(toDate) ||
transactionDate.isBefore(toDate));
        };
    }
}
```

This next method is a private method called “isInFilter” that returns a Predicate object used to check if a transaction's date falls within a specified date range. It converts the transaction's date to a LocalDate object and then compares it with the start and end dates to determine if it falls within the range.

```
package model.filehandling;

import java.io.*;
import java.util.List;

public class BalanceFileHandler implements FileHandler<Integer> {
    private static final String BALANCE_FILE_PATH =
"src/saved/balance.txt";
```

This next class called “BalanceFileHandler” handles file operations related to balance data. It implements the FileHandler interface with a type parameter of Integer and specifies the file path for the balance file.

```
// Saves the balance to the balance file
@Override
public void save(Integer balance) {
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter(BALANCE_FILE_PATH))) {
        writer.write(String.valueOf(balance));
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
}
```

This next code implements the save method in the “BalanceFileHandler” class. It writes the provided balance value to the balance file specified by the BALANCE_FILE_PATH using a BufferedWriter.

```
// Loads a single balance from the balance file
@Override
public Integer loadSingle() {
        try (BufferedReader reader = new BufferedReader(new
FileReader(BALANCE_FILE_PATH))) {
        String line = reader.readLine();
        return Integer.parseInt(line.trim());
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

The next method is the “loadSingle” method in the “BalanceFileHandler” class. It reads a single balance from the balance file specified by the BALANCE_FILE_PATH using a BufferedReader. The balance is then parsed from the file and returned as an Integer.

```
// Not implemented, as saving a list of balances is not supported
    @Override
    public void save(List<Integer> items) {
            throw new UnsupportedOperationException("Not supported
yet.");
    }

     // Not implemented, as loading a list of balances is not
supported
    @Override
    public List<Integer> load() {
            throw new UnsupportedOperationException("Not supported
yet.");
    }
}
```

And the last piece of code for this class includes two methods, save and load, which are not implemented in the “BalanceFileHandler” class. These methods are meant for saving and loading a list of balances, but they currently throw an

UnsupportedOperationException with an error message indicating that the functionality is not supported.

```
package model.filehandling;

import model.category.ExpenseCategory;
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class ExpenseCategoryFileHandler implements
FileHandler<ExpenseCategory> {
   private static final String EXPENSE_CATEGORIES_FILE_PATH =
"src/saved/expenseCategories.txt";

   @Override
   public void save(List<ExpenseCategory> categories) {
       try (BufferedWriter writer = new BufferedWriter(new
FileWriter(EXPENSE_CATEGORIES_FILE_PATH))) {
           for (ExpenseCategory category : categories) {
               String line = category.getName() + "/" +
category.getBudgetLimit();
               writer.write(line);
               writer.newLine();
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
```

The next class is called “ExpenseCategoryFileHandler”. It handles file operations specific to expense categories. It implements the FileHandler interface for saving a list of ExpenseCategory objects to a file. The save method writes each category's name and budget limit to the file, separated by a forward slash ("/"), with each category on a new line.

```
@Override
   public List<ExpenseCategory> load() {
       List<ExpenseCategory> categories = new ArrayList<>();
            try (BufferedReader reader = new BufferedReader(new
FileReader(EXPENSE_CATEGORIES_FILE_PATH))) {
           String line;
           while ((line = reader.readLine()) != null) {
               String[] parts = line.split("/");
               String categoryName = parts[0].trim();
               int budget = Integer.parseInt(parts[1].trim());
```

```
                                    ExpenseCategory category = new
ExpenseCategory(categoryName, budget);
                categories.add(category);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return categories;
    }
}
```

This nex code implements the load method in the “ExpenseCategoryFileHandler” class. It reads the expense categories from the expense categories file specified by the EXPENSE_CATEGORIES_FILE_PATH. The file is read line by line, and each line is split into parts using the "/" separator. The category name and budget limit are extracted from the parts, and a new ExpenseCategory object is created with these values. The created categories are added to a list and returned as the result.

```
package model.filehandling;

import java.util.ArrayList;
import java.util.List;

public interface FileHandler<T> {
    void save(List<T> items);
    List<T> load();
```

This next piece of code defines an interface called FileHandler that specifies two methods: save and load. The save method is responsible for saving a list of items to a file, while the load method is responsible for loading a list of items from a file. The interface is generic, allowing different types of items to be handled by implementing classes.

```
// Overloaded methods for handling individual items
    default void save(T item) {
        List<T> items = new ArrayList<>();
        items.add(item);
        save(items);
    }

    default T loadSingle() {
        List<T> items = load();
        if (items != null && !items.isEmpty()) {
            return items.get(0);
        }
```

```
        return null;
    }
}
```

And these last lines of code of the interface introduce overloaded methods in the FileHandler interface. The save method allows individual items of type T to be saved, and the loadSingle method retrieves a single item of type T from a file. These methods provide a simpler way to handle individual items without explicitly dealing with lists.

```
package model.filehandling;

import model.category.IncomeCategory;
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class IncomeCategoryFileHandler implements
FileHandler<IncomeCategory> {
    private static final String INCOME_CATEGORIES_FILE_PATH =
"src/saved/incomeCategories.txt";
```

The IncomeCategoryFileHandler class handles file operations for income categories. It implements the FileHandler interface for saving and loading IncomeCategory objects. The save method writes the income category names to a file, and the load method reads the file and creates IncomeCategory objects based on the stored data.

```
// Saves the list of income categories to the income categories
file
@Override
public void save(List<IncomeCategory> categories) {
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter(INCOME_CATEGORIES_FILE_PATH))) {
        for (IncomeCategory category : categories) {
            String line = category.getName();
            writer.write(line);
            writer.newLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

This next code implements the save method in the IncomeCategoryFileHandler class. It takes a list of IncomeCategory objects as input and writes the names of each category to the income categories file specified by the INCOME_CATEGORIES_FILE_PATH. The code uses a BufferedWriter wrapped around a FileWriter to write the category names to the file.

```
// Loads the list of income categories from the income categories
file
   @Override
   public List<IncomeCategory> load() {
       List<IncomeCategory> categories = new ArrayList<>();
            try (BufferedReader reader = new BufferedReader(new
FileReader(INCOME_CATEGORIES_FILE_PATH))) {
           String line;
           while ((line = reader.readLine()) != null) {
               String categoryName = line.trim();
                                    IncomeCategory category = new
IncomeCategory(categoryName);
               categories.add(category);
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
       return categories;
   }
}
```

This code implements the load method in the IncomeCategoryFileHandler class. It reads the income categories from the income categories file specified by the INCOME_CATEGORIES_FILE_PATH and creates a list of IncomeCategory objects based on the stored data.

The code uses a BufferedReader wrapped around a FileReader to read the lines of the file. It iterates over each line, trims any leading or trailing whitespace, and creates a new IncomeCategory object using the trimmed line as the category name. The newly created category is added to the categories list. Finally, the list of categories is returned. If any exception occurs during the file reading process, it will be caught, and the stack trace will be printed. The method returns an empty list if an error occurs.

```
package model.filehandling;

import model.category.Category;
import model.category.IncomeCategory;
import model.category.ExpenseCategory;
```

```
import model.Transaction;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

public class TransactionFileHandler implements
FileHandler<Transaction> {
   private static final String TRANSACTIONS_FILE_PATH =
"src/saved/transactions.txt";
```

This next class “TransactionFileHandler” handles file operations for transactions. It implements the FileHandler interface for saving and loading Transaction objects. The save method writes transaction data to a file, including transaction type, category name, amount, and date. The load method reads the file and creates Transaction objects based on the stored data. The class also includes utility methods for converting dates to and from string representations.

```
// Saves the list of transactions to the transactions file
@Override
public void save(List<Transaction> transactions) {
   try (BufferedWriter writer = new BufferedWriter(new
FileWriter(TRANSACTIONS_FILE_PATH))) {
       for (Transaction transaction : transactions) {
           String line = transaction.getType() + "," +
                   transaction.getCategory().getName() + "," +
                   transaction.getAmount() + "," +
                   transaction.getDescription() + "," +
                   transaction.getDate();
           writer.write(line);
           writer.newLine();
       }
   } catch (IOException e) {
       e.printStackTrace();
   }
}
```

This save method in the TransactionFileHandler class writes a list of Transaction objects to a file by converting each transaction's data into a line of comma-separated values.

```
// Loads the list of transactions from the transactions file
@Override
public List<Transaction> load() {
```

```
    List<Transaction> transactions = new ArrayList<>();
    try {
        List<String> lines =
Files.readAllLines(Paths.get(TRANSACTIONS_FILE_PATH));
        for (String line : lines) {
            String[] parts = line.split(",");
            if (parts.length == 5) {
                String type = parts[0];
                String categoryName = parts[1];
                int amount = Integer.parseInt(parts[2]);
                String description = parts[3];
                String dateString = parts[4];
                DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd");
                LocalDate date = LocalDate.parse(dateString,
formatter);
                Category category =
findCategoryByName(categoryName);
                if (category != null) {
                    Transaction transaction = new Transaction(type,
category, amount, description, date);
                    transactions.add(transaction);
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return transactions;
}
```

The load method in the TransactionFileHandler class is responsible for reading and loading a list of transactions from the transactions file specified by the constant TRANSACTIONS_FILE_PATH.

First, it creates an empty list to store the loaded transactions. Then, using the Files.readAllLines method, it reads all the lines from the transactions file and stores them in a list of strings. For each line in the list, the method splits the line into separate parts using the comma as the delimiter. It checks if the line has the expected number of parts (5) to ensure valid transaction data.

If the line is valid, it extracts the relevant information such as the transaction type, category name, amount, description, and date. The date is parsed using a DateTimeFormatter to convert the string representation into a LocalDate object. Next, it calls the findCategoryByName method (which is not shown in the code snippet) to find the appropriate category object based on the category name

obtained from the line. If a valid category object is found, a new Transaction object is created using the

```
// Finds a category by its name
   public Category findCategoryByName(String categoryName) {
       List<IncomeCategory> incomeCategories = new
IncomeCategoryFileHandler().load();
       List<ExpenseCategory> expenseCategories = new
ExpenseCategoryFileHandler().load();

       // Combine the income and expense categories into a single
list
       List<Category> allCategories = new ArrayList<>();
       allCategories.addAll(incomeCategories);
       allCategories.addAll(expenseCategories);

       for (Category category : allCategories) {
           if (category.getName().equalsIgnoreCase(categoryName)) {
               return category;
           }
       }
       return null;
   }
}
```

The findCategoryByName method in the TransactionFileHandler class is used to locate a category object based on its name. It first loads the lists of income and expense categories from their respective files. Then, it combines these categories into a single list. The method iterates through each category in the list and compares the name of the category (ignoring case) with the provided categoryName. If a match is found, the corresponding category object is returned. This method enables the TransactionFileHandler to retrieve the appropriate category when loading transactions from the transactions file.

```
package model;

import model.category.Category;
import java.time.LocalDate;

public class Transaction {
   private String type;
   private Category category;
   private int amount;
   private String description;
   private LocalDate date;
```

```java
    public Transaction(String type, Category category, int amount,
String description, LocalDate date) {
        this.type = type;
        this.category = category;
        this.amount = amount;
        this.description = description;
        this.date = date;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public Category getCategory() {
        return category;
    }

    public void setCategory(Category category) {
        this.category = category;
    }

    public int getAmount() {
        return amount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public LocalDate getDate() {
        return date;
    }

    public void setDate(LocalDate date) {
        this.date = date;
    }
}
```

This Transaction class represents a financial transaction. It has private instance variables including type (the type of transaction), category (the category of the transaction), amount (the monetary amount of the transaction), description (a description of the transaction), and date (the date of the transaction). The class provides a constructor to initialize these variables, as well as getter and setter methods to access and modify the values of the variables. By encapsulating the transaction data within this class, it becomes easier to manage and manipulate transaction objects within the application.

6. **Saved**

This last package contains the txt files that store the necessary data for each category.

D. Lessons Learned

Through my first experience with GUI development, I acquired an understanding of how GUIs generally operate. I also learned to utilize JFreeChart effectively to create the desired visualizations for my program. YouTube videos and related websites were invaluable resources that helped me grasp the necessary concepts for this project.

References

https://www.geeksforgeeks.org

http://stackoverflow.com

Relevant Links

GitHub Repository:

https://github.com/priscillabigaill/budget_buddy

Demo Video:

https://www.youtube.com/watch?v=ibhQrORU7iU