

**VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
INTERNATIONAL UNIVERSITY**



**PERSONAL EXPENDITURE
MANAGEMENT APPLICATION**

BY

NGUYEN TUAN KIET – ITITWE20032
LE NGOC DANG KHOA – ITITIU20230

ADVISOR

Dr. LE DUY TAN
MSc. NGUYEN TRUNG NGHIA

HOCHIMINH CITY, VIETNAM
2024

INTRODUCTION

Effective money management for individuals has become more important than ever in the modern world. Keeping track of personal money manually is laborious and prone to inaccuracy in the age of digital transactions and varied sources of income and expenses. Personal expenditure applications have arisen as useful solutions to fulfill this demand, allowing people to easily track and manage their financial actions. The development of a Java-based personal spending application is the main topic of this paper, which also discusses its features, architecture, and the decision to utilize Java as the development platform.

About a Personal Expenditure Java App

A personal expenditure app is a powerful tool designed to help individuals manage their finances with ease and efficiency. This app allows users to track their daily spending, categorize expenses, set budgets, and monitor their financial health over time. By providing a clear overview of where money is being spent, it enables users to identify patterns and make informed decisions to improve their financial habits. The app often includes features such as expense tracking, budget planning, and financial reporting, with some advanced versions offering integration with bank accounts for automatic transaction updates. With the ability to set financial goals and receive alerts for upcoming bills or overspending, a personal expenditure app becomes an indispensable companion for anyone looking to achieve financial stability and discipline.

References

- Icons: iconfinder.com
- Tutorial: [Viết chương trình Quản lý thu chi cá nhân bằng Java Swing \(youtube.com\)](https://www.youtube.com/watch?v=...)

Contributions:

- Nguyen Tuan Kiet: UX/UI design, Database design, Expenditure Types, Database connection.
- Le Ngoc Dang Khoa: Dialog, Validator, Expenditure, Database connection.

Software Requirement

IDE: NetBeans

- **IDE (Integrated Development Environment):** NetBeans is a powerful, open-source IDE for Java development, although it supports other languages like PHP and C/C++ as well. It offers features such as code editing, debugging, version control, and GUI building tools.
- **Platform Support:** NetBeans is cross-platform, meaning it runs on multiple operating systems including Windows, macOS, and Linux. This flexibility makes it accessible to a wide range of developers.
- **Java Support:** NetBeans has robust support for Java development, providing tools for Java SE, Java EE, and Java ME development. It includes features like code templates, refactoring tools, and project management capabilities.
- **GUI Builder:** NetBeans includes a drag-and-drop GUI builder tool called Matisse, which simplifies the creation of Java Swing-based user interfaces. This tool allows developers to design UI components visually and generates the corresponding code.
- **Community and Ecosystem:** NetBeans has a vibrant community of developers and contributors who provide support, plugins, and extensions. It also integrates with popular version control systems like Git and supports Maven and Ant for project management.

Java Framework: JavaSwing

- **GUI Toolkit:** Java Swing is a graphical user interface (GUI) toolkit included in the Java Development Kit (JDK). It allows developers to create rich, platform-independent GUI applications for desktop environments.
- **Components:** Swing provides a comprehensive set of components for building GUIs, including buttons, text fields, labels, panels, dialogs, and more. These components are lightweight and customizable, allowing developers to create complex UIs.
- **Event-Driven Programming:** Swing follows an event-driven programming model, where user interactions (such as mouse clicks and key presses) trigger events that are handled by event listeners. This model enables developers to create responsive and interactive applications.
- **Customization:** Swing components can be customized extensively using properties, styles, and listeners. Developers can tailor the appearance and behavior of components to suit their application's requirements.
- **Platform Independence:** Swing applications are written in Java and can run on any platform that supports the Java Virtual Machine (JVM). This platform independence makes Swing ideal for developing cross-platform desktop applications.
- **Integration with NetBeans:** NetBeans provides comprehensive support for Java Swing development, including visual GUI design tools, code generation, and

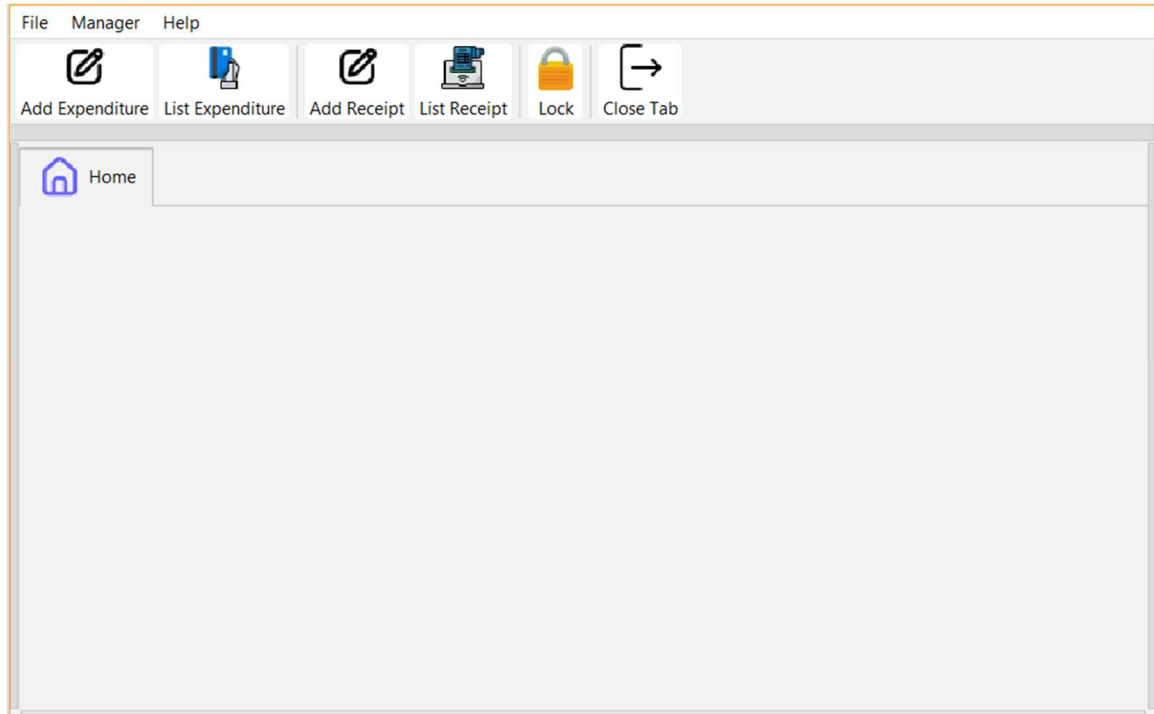
debugging capabilities. This integration streamlines the development process and enhances productivity.

Database: SQL Server (JDBC 12)

- **Database Agnostic:** JDBC allows Java applications to interact with different database management systems (DBMS) such as MySQL, PostgreSQL, Oracle, SQL Server, and more. This agnostic nature makes it possible to write database code that is independent of the underlying database being used.
- **Core Components:**
 - **DriverManager:** Facilitates the establishment of a connection to the database. It manages a set of database drivers, each of which is capable of connecting to a specific type of database.
 - **Connection:** Represents a connection to a database. It provides methods for creating statements, committing transactions, and managing connection properties.
 - **Statement:** Executes SQL queries or updates against the database. There are two types of statements: `Statement` for executing static SQL statements and `PreparedStatement` for executing precompiled SQL statements with parameters.
 - **ResultSet:** Represents the result of a query execution. It provides methods for navigating through the result set and retrieving data.
- **CRUD Operations:** JDBC allows developers to perform CRUD (Create, Read, Update, Delete) operations on databases using SQL queries. This includes inserting new records, retrieving data based on specific criteria, updating existing records, and deleting records from tables.
- **Transaction Management:** JDBC supports transaction management, allowing developers to group database operations into transactions. Transactions can be committed to make the changes permanent or rolled back to undo them in case of errors.
- **Exception Handling:** JDBC methods throw checked exceptions, such as `SQLException`, which need to be handled by the application code. Proper exception handling ensures graceful error recovery and enhances the robustness of database operations.
- **Batch Processing:** JDBC supports batch processing, where multiple SQL statements are grouped together and executed as a batch. This improves performance by reducing the number of round-trips between the application and the database.
- **Connection Pooling:** For improved scalability and performance, JDBC applications often use connection pooling libraries like Apache Commons DBCP or HikariCP. Connection pooling minimizes the overhead of creating and closing database connections by reusing existing connections from a pool.

IMPLEMENT AND DESIGN

MainFrame



1. Imports and Package Declaration:

```
package kk.pe;

import java.awt.Component;
import java.awt.event.KeyEvent;
import kk.pe.dialog.AboutUsDialog;
import kk.pe.dialog.LoginDialog;
import kk.pe.dialog.SettingDialog;
import kk.pe.tabs.*;
```

- The package `kk.pe` contains the classes for the application.
- Various Swing components and custom dialog/tab classes are imported.

2. MainFrame Class Definition:

```
public class MainFrame extends javax.swing.JFrame {
```

- o MainFrame extends javax.swing.JFrame, making it the main window of the application.

3. **Constructor:**

```
public MainFrame() {
    initComponents();
    setLocationRelativeTo(null);
}
```

- o The constructor initializes the components and centers the frame on the screen.

4. **initComponents Method:** This method initializes and sets up the GUI components, including the toolbar, menu bar, and tabs.

```
private void initComponents() {
    // Toolbar setup
    jToolBar1 = new javax.swing.JToolBar();
    tbrAddExpenditure = new javax.swing.JButton();
    tbrListExpenditure = new javax.swing.JButton();
    jSeparator5 = new
javax.swing.JToolBar.Separator();
    tbrAddReceipt = new javax.swing.JButton();
    tbrListReceipt = new javax.swing.JButton();
    jSeparator6 = new
javax.swing.JToolBar.Separator();
    tbrLock = new javax.swing.JButton();
    jSeparator7 = new
javax.swing.JToolBar.Separator();
    tbrCloseTab = new javax.swing.JButton();

    // TabbedPane setup
    tpnBoard = new javax.swing.JTabbedPane();
    jPanel2 = new javax.swing.JPanel();

    // MenuBar setup
    jMenuBar1 = new javax.swing.JMenuBar();
    jMenu1 = new javax.swing.JMenu();
    menuLogin = new javax.swing.JMenuItem();
    menuLogout = new javax.swing.JMenuItem();
    menuLock = new javax.swing.JMenuItem();
    jSeparator3 = new
javax.swing.JPopupMenu.Separator();
    menuSettings = new javax.swing.JMenuItem();
    jSeparator4 = new
javax.swing.JPopupMenu.Separator();
    menuExit = new javax.swing.JMenuItem();
}
```

```

        // ... (other menu items and menus)
    }

```

- `jToolBar1` and its buttons for adding/listing expenditures and receipts, locking, and closing tabs.
- `tpnBoard` is a `JTabbedPane` for managing different panes like home, add expenditure, etc.
- `jMenuBar1` contains `jMenu1`, `jMenu2`, and `jMenu3` for "File", "Manager", and "Help" menus respectively, each with several menu items.

5. **Action Listeners:** Action listeners are added to buttons and menu items to handle user interactions:

```

tbrAddExpenditure.addActionListener(new
java.awt.event.ActionListener() {
    public void
actionPerformed(java.awt.event.ActionEvent evt) {
        tbrAddExpenditureActionPerformed(evt);
    }
});

```

- When `tbrAddExpenditure` is clicked, it triggers `tbrAddExpenditureActionPerformed` which adds a new tab for adding an expenditure.

6. **Menu and Toolbar Actions:** Methods that are invoked by the action listeners to perform specific actions, such as displaying dialogs, adding tabs, etc.

```

private void
tbrAddExpenditureActionPerformed(java.awt.event.Action
Event evt) {
    AddExpenditurePane pane = new
AddExpenditurePane(this);
    tpnBoard.addTab("Add Expenditure", pane);
    tpnBoard.setSelectedComponent(pane);
}

```

- This method adds an "Add Expenditure" tab to the tabbed pane.

7. **Main Method:** The entry point of the application:

```

public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new MainFrame().setVisible(true);

            LoginDialog dialog = new LoginDialog(null,
true);
            dialog.setVisible(true);
        }
    });
}

```



```

    }
    });
}

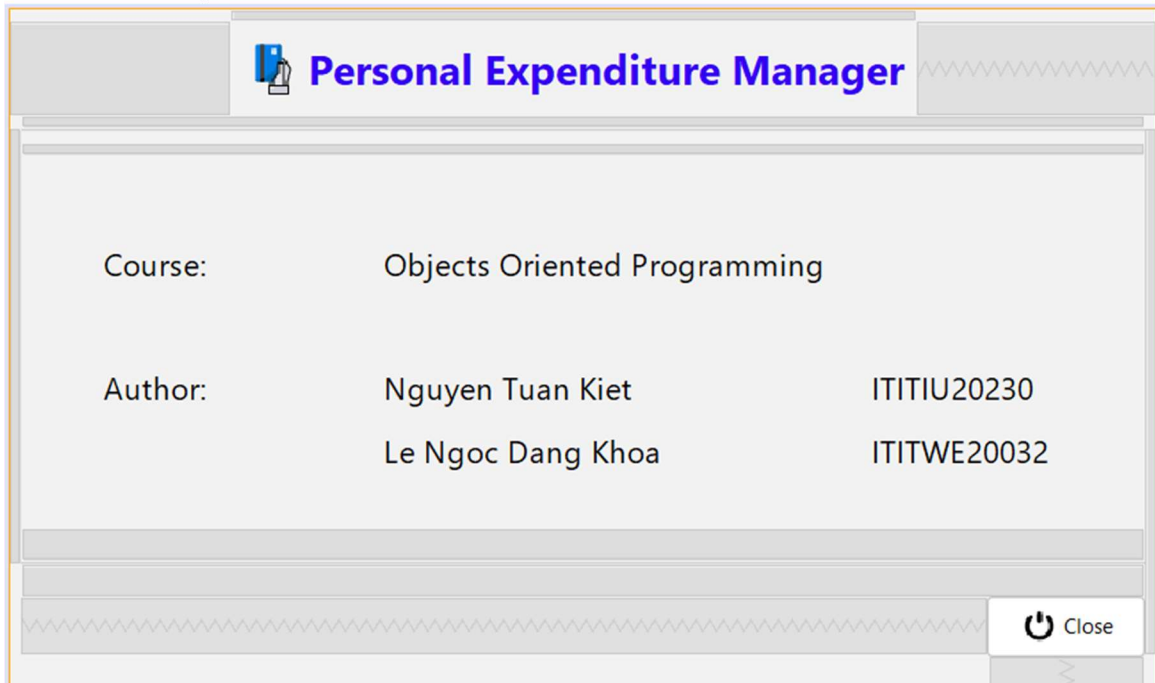
```

- The main method uses the `EventQueue` to create and display the `MainFrame`.
- It also displays the `LoginDialog` upon startup.

Dialog

About us:

- Showing the information of authors



Login:

- Requiring users to log in to be able to operate on the app

Login

Username:

Password:

☐ Remember me

```
private void btnLoginActionPerformed(java.awt.event.ActionEvent evt) {
    String username = txtUsername.getText();
    String password = txtPassword.getText();

    if(username.equals("admin") && password.equals("123456")){
        dispose();
    }else{
        JOptionPane.showMessageDialog(parentComponent, this, message: "Invalid Username or Password", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
    }
}
```

- This code declare username and password values to access the application. If entered incorrectly, an error message will appear.

Settings:

Settings

☒ General
 ☐ Database

Languages:

- Allows users to select language and set database-related values.

Expenditure

Data Access Object (ExpenditureDao.java):

- **insert(Expenditure entity):** Inserts a new expenditure record into the database and returns the inserted entity with its ID set.
- **update(Expenditure entity):** Updates an existing expenditure record in the database based on its ID.
- **delete(int id):** Deletes an expenditure record from the database based on its ID.
- **findAll():** Retrieves all expenditure records from the database.
- **findById(int id):** Retrieves an expenditure record from the database based on its ID.

Plain Old Java Object (Expenditure.java):

- **id:** The unique identifier for the expenditure.
- **name:** The name or description of the expenditure.
- **amount:** The amount of money spent.
- **note:** Any additional notes or comments about the expenditure.
- **type:** An integer representing the type of expenditure. This could be mapped to a specific expenditure category or purpose.
- **expenditureDate:** The date when the expenditure occurred.

Add Expenditure:

Components and Initialization

1. **Instance Variables:**
 - **mainFrame:** Reference to the main frame of the application.
 - Various Swing components like `JLabel`, `JTextField`, `JFormattedTextField`, `JComboBox`, `JTextArea`, and `JBButton`.
2. **Constructor:**
 - `public AddExpenditurePane(MainFrame mainFrame):` Initializes the panel, sets the main frame, calls `initComponents ()` to setup the GUI, and `loadTypes ()` to load expenditure types.
 - `public AddExpenditurePane(MainFrame mainFrame, int id):` Similar to the previous constructor but also calls `loadById (id)` to load a specific expenditure by its ID.
3. **initComponents ():**

- Sets up the GUI components and their layout using GroupLayout.
- Adds action listeners to buttons for handling user interactions.

Methods

1. **loadTypes():**
 - Loads expenditure types from the database using `ExpenditureTypeDao` and adds them to the `cbxType` combo box.
2. **loadById(int id):**
 - Loads an expenditure record by its ID using `ExpenditureDao`.
 - Populates the form fields with the retrieved expenditure data.
 - Adjusts the state of fields and buttons accordingly.
3. **changeButtonStates(boolean edit, boolean save, boolean update, boolean delete):**
 - Enables or disables buttons based on the provided boolean values.
4. **changeFieldStates(boolean isEditable):**
 - Sets the editability of form fields.

Action Listeners

1. **btnNewActionPerformed(evt):**
 - Clears the form fields and enables them for new entry.
 - Adjusts button states for the new entry mode.
2. **btnListActionPerformed(evt):**
 - Switches the view to show the list of expenditures.
3. **btnEditActionPerformed(evt):**
 - Enables the form fields and adjusts button states for edit mode.
4. **btnSaveActionPerformed(evt):**
 - Validates form data using `ExpenditureValidator`.
 - Creates a new `Expenditure` object with the form data.
 - Saves the new expenditure to the database using `ExpenditureDao`.
 - Updates the form and button states post-save.
5. **btnUpdateActionPerformed(evt):**
 - Confirms the update action with the user.
 - Validates the form data.
 - Updates the existing expenditure record in the database using `ExpenditureDao`.
 - Adjusts the form and button states post-update.
6. **btnDeleteActionPerformed(evt):**
 - Confirms the delete action with the user.
 - Deletes the expenditure record from the database using `ExpenditureDao`.
 - Clears the form fields post-delete.

Helper Classes and Utilities

- **DAO Classes:**
 - ExpenditureDao: Handles database operations related to expenditure records.
 - ExpenditureTypeDao: Handles database operations related to expenditure types.
- **Entity Classes:**
 - Expenditure: Represents an expenditure record.
 - ExpenditureType: Represents an expenditure type.
- **Utility Classes:**
 - DateUtil: Handles date formatting and parsing.
 - MessageBox: Displays various types of message dialogs.
 - ExpenditureValidator: Validates the form data for expenditures.

List Expenditure:

- Initializing the Table:

```
private void initTable() {  
    model = new DefaultTableModel();  
    model.setColumnIdentifiers(new String[]{"ID", "Name", "Amount", "Date", "Type"});  
  
    tblList.setModel(dataModel:model);  
}
```

initTable() - Sets up the table model with column identifiers.

- Loading Data:

```

private void loadAll() {
    try{
        ExpenditureDao dao = new ExpenditureDao();
        var list = dao.findAll();

        model.setRowCount(rowCount: 0);
        // Hiển thị thông tin
        for(Expenditure item : list){
            Object[] row = new Object[]{
                item.getId(),
                item.getName(),
                item.getAmount(),
                item.getExpenditureDate(),
                item.getType(),
            };
            model.addRow(rowData: row);
        }
        model.fireTableDataChanged();

    }catch(Exception e){
        MessageBox.showMessageDialog(parent:this, message: e.getMessage());
        e.printStackTrace();
    }
}

```

loadAll() - Loads all expenditures from the database using ExpenditureDao, clears the table model, and adds rows for each expenditure.

- Event Handlers:

o Delete:

```

private void ppmDeleteActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        if(MessageBox.showConfirmMessage(parent:this, message: "Do you want to delete?")
            == JOptionPane.NO_OPTION){
            return;
        }

        ExpenditureDao dao = new ExpenditureDao();

        int selectedRow = tblList.getSelectedRow();

        Object idObj = tblList.getValueAt(row: selectedRow, column: 0);
        if (idObj != null){
            int id = Integer.parseInt(s: idObj.toString());

            if(dao.delete(id)){
                MessageBox.showInformationMessage(parent:this, title: "Information", message: "Expenditure is deleted");
                loadAll();
            }else{
                MessageBox.showMessageDialog(parent:this, title: "Error", message: "Expenditure can not be deleted");
            }
        }

    } catch (Exception e){
        e.printStackTrace();//Thông báo lỗi xuất hiện ở dòng code nào
        MessageBox.showMessageDialog(parent:this, title: "Error", message: e.getMessage());
    }
}

```

ppmDeleteActionPerformed() - Handles deletion of a selected expenditure. It shows a confirmation dialog, then deletes the expenditure if confirmed.

- Edit Action:

```
private void ppmEditActionPerformed(java.awt.event.ActionEvent evt) {  
    int selectedRow = tblList.getSelectedRow();  
  
    Object idObj = tblList.getValueAt(row:selectedRow,column:0);  
    if (idObj !=null){  
        int id = Integer.parseInt(s: idObj.toString());  
  
        mainFrame.showEditExpenditure(id);  
    }  
}
```

ppmEditActionPerformed() - Handles editing of a selected expenditure. It retrieves the ID of the selected expenditure and calls a method in mainFrame to show the edit form.

Expenditure Type:

Data Access Object (ExpenditureDao.java):

Insert Method:

```
public ExpenditureType insert (ExpenditureType entity) throws Exception {
// Chỉ thao tác vào trường dữ liệu Name vì ID là trường tự tăng
String sql = "insert into expenditureType(name) values(?) ";
//-----//

// Để có thể lấy được giá trị của trường tự tăng thì phải có tham số thứ 2 truyền vào PreparedStatement
try(Connection con = DatabaseUtil.getConnection();
    PreparedStatement pstmt = con.prepareStatement(sql,
        autoGeneratedKeys: PreparedStatement.RETURN_GENERATED_KEYS);){
//Thiết lập giá trị cho tham số ? trong values của hàm trên
pstmt.setString(parameterIndex: 1,x: entity.getName());
// Sau khi thực hiện lệnh mới có giá trị của trường ID
pstmt.executeUpdate();
// Lấy các KEYS được sinh ra và thiết lập cho setId và trả về giá trị entity
ResultSet rs = pstmt.getGeneratedKeys();

    if(rs.next()){
        entity.setId(id: rs.getInt(columnIndex: 1));
    }

    return entity;
}
}
```

insert(ExpenditureType entity) - Inserts a new ExpenditureType into the database.

- **SQL Statement** - The SQL command to insert a new record.
- **PreparedStatement** - Prepared with the SQL command and configured to return generated keys.
- **pstmt.setString(1, entity.getName())** - Sets the name parameter in the SQL command.
- **pstmt.executeUpdate()** - Executes the insert operation.
- **rs.getGeneratedKeys()** - Retrieves the generated ID and sets it in the entity.

Update method:

```
public ExpenditureType update (ExpenditureType entity) throws Exception {
// Chỉ thao tác vào trường dữ liệu Name vì ID là trường tự tăng
String sql = "update expenditureType set name= ? where id = ? ";
//-----//

// Để có thể lấy được giá trị của trường tự tăng thì phải có tham số thứ 2 truyền vào PreparedStatement
try(Connection con = DatabaseUtil.getConnection();
    PreparedStatement pstmt = con.prepareStatement(sql);){
//Thiết lập giá trị cho tham số ? trong values của hàm trên
pstmt.setString(parameterIndex: 1,x: entity.getName());
pstmt.setInt(parameterIndex: 2,x: entity.getId());
// Sau khi thực hiện lệnh mới có giá trị của trường ID
pstmt.executeUpdate();
    return entity;
}
}
```

update(ExpenditureType entity) - Updates an existing ExpenditureType in the database.

- **SQL Statement** - The SQL command to update a record.
- **PreparedStatement** - Prepared with the SQL command.
- **pstmt.setString(1, entity.getName())** - Sets the name parameter in the SQL command.
- **pstmt.setInt(2, entity.getId())** - Sets the ID parameter in the SQL command.
- **pstmt.executeUpdate()** - Executes the update operation.

Delete method:

```
public boolean delete (int id) throws Exception {
// Chỉ thao tác vào trường dữ liệu Name vì ID là trường tự tăng
    String sql = "delete from ExpenditureType where id = ? ";
//-----//

// Để có thể lấy được giá trị của trường tự tăng thì phải có tham số thứ 2 truyền vào PreparedStatement
    try(Connection con = DatabaseUtil.getConnection();
        PreparedStatement pstmt = con.prepareStatement(sql)){
//Thiết lập giá trị cho tham số ? trong values của hàm trên
        pstmt.setInt(parameterIndex: 1,x: id);

        return pstmt.executeUpdate()>0;
    }
}
```

delete(int id) - Deletes an ExpenditureType from the database by ID.

- **SQL Statement** - The SQL command to delete a record.
- **PreparedStatement** - Prepared with the SQL command.
- **pstmt.setInt(1, id)** - Sets the ID parameter in the SQL command.
- **pstmt.executeUpdate()** - Executes the delete operation and returns true if a record was deleted.

Find All method:

```
public List<ExpenditureType> findAll () throws Exception {
// Cho phép đọc tất cả dữ liệu từ ExpenditureType
    String sql = "select * from ExpenditureType ";
//-----//

// Để có thể lấy được giá trị của trường tự tăng thì phải có tham số thứ 2 truyền vào PreparedStatement
    try(Connection con = DatabaseUtil.getConnection();
        PreparedStatement pstmt = con.prepareStatement(sql)){

        List<ExpenditureType> list = new ArrayList<>();

        try(ResultSet rs = pstmt.executeQuery()){
// Đọc giá trị từng hàng
            while (rs.next()){
                ExpenditureType entity = new ExpenditureType();
                entity.setId(id: rs.getInt(columnLabel: "id"));
                entity.setName(name: rs.getString(columnLabel: "name"));

                list.add(e: entity);
            }
        }
        return list;
    }
}
```

findAll() - Retrieves all ExpenditureType records from the database.

- **SQL Statement** - The SQL command to select all records.

- **PreparedStatement** - Prepared with the SQL command.
- **ResultSet** - Used to iterate through the results.
- **While Loop** - Iterates through the results, creates ExpenditureType objects, and adds them to the list.

Find By ID method:

```

public ExpenditureType findById (int id) throws Exception {
// Cho phép đọc tất cả dữ liệu từ ExpenditureType
    String sql = "select * from ExpenditureType where id = ?";
//-----//

// Để có thể lấy được giá trị của trường tự tăng thì phải có tham số thứ 2 truyền vào PreparedStatement
    try(Connection con = DatabaseUtil.getConnection();
        PreparedStatement pstmt = con.prepareStatement(sql)){

        pstmt.setInt(parameterIndex: 1, x: id);

        try(ResultSet rs = pstmt.executeQuery()){
//            Đọc giá trị từng hàng
            if(rs.next()){
                ExpenditureType entity = new ExpenditureType();
                entity.setId(id: rs.getInt(columnLabel: "id"));
                entity.setName(name: rs.getString(columnLabel: "name"));

                return entity;
            }
        }
        return null;
    }
}

```

findById(int id) - Retrieves an ExpenditureType record from the database by ID.

- **SQL Statement** - The SQL command to select a record by ID.
- **PreparedStatement** - Prepared with the SQL command.
- **pstmt.setInt(1, id)** - Sets the ID parameter in the SQL command.
- **ResultSet** - Used to get the result.
- **If Statement** - Checks if a record was found and creates an ExpenditureType object.

Add Expenditure Type:

Utility Methods:

```

private void loadById(int id){
    try{
        ExpenditureTypeDao dao = new ExpenditureTypeDao();

        ExpenditureType entity = dao.findById(id);

        txtID.setText("" +entity.getId());
        txtName.setText(t: entity.getName());

        changeButtonStates(edit: true, save: false, update: true, delete: true);
    }catch(Exception e){
        MessageBox.showErrorMessage(parent: this, title: "Error", message: e.getMessage());
    }
}

```

loadById(int id): Loads an expenditure type by its ID for editing.

```

private void changeButtonStates(boolean edit, boolean save, boolean update, boolean delete){
    btnEdit.setEnabled(b: edit);
    btnSave.setEnabled(b: save);
    btnUpdate.setEnabled(b: update);
    btnDelete.setEnabled(b: delete);
}

```

changeButtonStates(boolean edit, boolean save, boolean update, boolean delete): Enables or disables buttons based on the provided boolean values.

Event Listeners:

- Action listeners are defined for various buttons like New, Save, Edit, Update, Delete, and List. These listeners perform actions like saving, updating, deleting, etc.

Action Methods:

- New:

```

private void btnNewActionPerformed(java.awt.event.ActionEvent evt) {
    txtID.setText(t: "");
    txtName.setText(t: "");

    changeButtonStates(edit: false, save: true, update: false, delete: false);
}

```

btnNewActionPerformed: Resets the input fields and button states for adding a new expenditure type.

- Save:

```

private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        Kiểm tra tính hợp lệ của dữ liệu trước khi lưu vào cơ sở dữ liệu
        Tránh trường hợp người dùng không đưa vào giá trị Name và giá trị Name vẫn được lưu vào csdl
        String valid = ExpenditureTypeValidator.validate(txtName);
        if(valid !=null){
            MessageBox.showErrorMessage(parent:this, title: "Error", message: valid);
            return;
        }

        ExpenditureType entity = new ExpenditureType();
        entity.setName(name: txtName.getText());

        ExpenditureTypeDao dao = new ExpenditureTypeDao();
        entity = dao.insert(entity);

        txtID.setText(""+entity.getId());

        JOptionPane.showMessageDialog(this,"Type is save", "Information", JOptionPane.INFORMATION_MESSAGE);
        MessageBox.showInformationMessage(parent:this, title: "Information", message: "Type is saved");
        txtID.setEditable(b: false);
        txtName.setEditable(b: false);
        //thể độ làm việc của các nút khi truy cập vào chức năng Save
        changeButtonStates(edit: true, save: false, update:true, delete:true);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this,"Error: "+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        e.printStackTrace();//Thông báo lỗi xuất hiện ở dòng code nào
        MessageBox.showErrorMessage(parent:this, title: "Error", message: e.getMessage());
    }
}

```

btnSaveActionPerformed: Validates input, saves a new expenditure type, and displays a message.

- Edit:

```

private void btnEditActionPerformed(java.awt.event.ActionEvent evt) {
    txtName.setEditable(b: true);

    changeButtonStates(edit: true, save: false, update:true, delete:true);
}

```

btnEditActionPerformed: Enables editing of an existing expenditure type.

- Update:

```

private void btnUpdateActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        if (MessageBox.showConfirmMessage(parent: this, message: "Do you want to update?")
            == JOptionPane.NO_OPTION) {
            return;
        }

        Kiểm tra tính hợp lệ của dữ liệu trước khi lưu vào cơ sở dữ liệu
        Tránh trường hợp người dùng không đưa vào giá trị Name và giá trị Name vẫn được lưu vào csdl
        String valid = ExpenditureTypeValidator.validate(txtName);
        if (valid != null) {
            MessageBox.showErrorMessage(parent: this, title: "Error", message: valid);
            return;
        }

        ExpenditureType entity = new ExpenditureType();
        entity.setId(id: Integer.parseInt(s: txtID.getText()));
        entity.setName(name: txtName.getText());

        ExpenditureTypeDao dao = new ExpenditureTypeDao();

        entity = dao.update(entity);

        MessageBox.showInformationMessage(parent: this, title: "Information", message: "Type is updated");
        txtID.setEditable(b: false);
        txtName.setEditable(b: false);

        Chờ đợi làm việc của các nút khi truy cập vào chức năng Update
        changeButtonStates(edit: true, save: false, update: false, delete: true);
    } catch (Exception e) {
        e.printStackTrace(); // Thông báo lỗi xuất hiện ở dòng code nào
        MessageBox.showErrorMessage(parent: this, title: "Error", message: e.getMessage());
    }
}

```

btnUpdateActionPerformed: Updates an existing expenditure type.

- Delete:

```

private void btnDeleteActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        if (MessageBox.showConfirmMessage(parent: this, message: "Do you want to delete?")
            == JOptionPane.NO_OPTION) {
            return;
        }

        ExpenditureTypeDao dao = new ExpenditureTypeDao();
        int id = Integer.parseInt(s: txtID.getText());
        if (dao.delete(id)) {
            MessageBox.showInformationMessage(parent: this, title: "Information", message: "Type is deleted");
        } else {
            MessageBox.showErrorMessage(parent: this, title: "Error", message: "Type can not be deleted");
        }

        btnNewActionPerformed(evt);
    } catch (Exception e) {
        e.printStackTrace(); // Thông báo lỗi xuất hiện ở dòng code nào
        MessageBox.showErrorMessage(parent: this, title: "Error", message: e.getMessage());
    }
}

```

btnDeleteActionPerformed: Deletes an existing expenditure type.

List Expenditure Types:

Initialization:

```
private void initTable() {  
    model = new DefaultTableModel();  
  
    model.setColumnIdentifiers(new String [] {"ID", "Name"});  
  
    tblList.setModel(dataModel:model);  
}
```

initTable(): This method initializes the table structure. It creates a DefaultTableModel with columns "ID" and "Name" and sets it as the model for the JTable(tblList).

```
private void loadData() {  
    try{  
        ExpenditureTypeDao dao = new ExpenditureTypeDao();  
        List<ExpenditureType> list = dao.findAll();  
  
        model.setRowCount(rowCount: 0);  
        for(ExpenditureType item : list){  
            Object [] row = new Object[]{item.getId(), item.getName()};  
  
            model.addRow(rowData: row);  
        }  
  
        model.fireTableDataChanged();  
    }catch (Exception e){  
        MessageBox.showErrorMessage(parent:this, title: "Error", message: e.getMessage());  
    }  
}
```

loadData(): This method fetches data from the database using ExpenditureTypeDao and populates the table with the fetched data. It clears the existing data in the table model (model) and adds new rows based on the fetched data.

Event Handling:

- Delete:


```

private void ppmDeleteActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        if (MessageBox.showConfirmMessage(parent: this, message: "Do you want to delete?")
            == JOptionPane.NO_OPTION) {
            return;
        }

        ExpenditureTypeDao dao = new ExpenditureTypeDao();

        int selectedRow = tblList.getSelectedRow();

        Object idObj = tblList.getValueAt(row: selectedRow, column: 0);
        if (idObj != null) {
            int id = Integer.parseInt(s: idObj.toString());

            if (dao.delete(id)) {
                MessageBox.showInformationMessage(parent: this, title: "Information", message: "Type is deleted");
                loadData();
            } else {
                MessageBox.showErrorMessage(parent: this, title: "Error", message: "Type can not be deleted");
            }
        }

    } catch (Exception e) {
        e.printStackTrace(); // Thông báo lỗi xuất hiện ở dòng code nào
        MessageBox.showErrorMessage(parent: this, title: "Error", message: e.getMessage());
    }
}

```

ppmDeleteActionPerformed(ActionEvent evt): This method is invoked when the user selects the "Delete" option from the popup menu (ppmList). It retrieves the selected row from the table, extracts the ID of the selected expenditure type, deletes it using ExpenditureTypeDao, and updates the table data.

- Edit:

```

private void ppmEditActionPerformed(java.awt.event.ActionEvent evt) {
    int selectedRow = tblList.getSelectedRow();

    Object idObj = tblList.getValueAt(row: selectedRow, column: 0);
    mainFrame.showEditExpenditureType(id: Integer.parseInt(s: idObj.toString()));
}

```

ppmEditActionPerformed(ActionEvent evt): This method is invoked when the user selects the "Edit" option from the popup menu. It retrieves the selected row from the table, extracts the ID of the selected expenditure type, and likely navigates to the edit view of that expenditure type using the main frame (mainFrame).

Util:

Database:

Constants:

```
public static final String DB_NAME = "ExpenditureDB";  
public static final String USERNAME = "KK";  
public static final String PASSWORD = "123456";
```

- DB_NAME: Represents the name of the database.
- USERNAME: Represents the username for connecting to the database.
- PASSWORD: Represents the password for connecting to the database.

Methods:

```
public static Connection getConnection() throws ClassNotFoundException, SQLException {  
    return getConnection(dbName: DB_NAME, username: USERNAME, password: PASSWORD);  
}
```

getConnection(): This static method retrieves a database connection using the default database name (DB_NAME), username (USERNAME), and password (PASSWORD). It delegates to the overloaded getConnection(String, String, String) method.

```
public static Connection getConnection(String dbName, String username, String password) throws ClassNotFoundException, SQLException {  
    Class.forName(className: "com.microsoft.sqlserver.jdbc.SQLServerDriver");  
  
    String dbURL = "jdbc:sqlserver://MSI;databaseName=ExpenditureDB"  
        + ";encrypt=true;trustServerCertificate=true;";  
  
    Connection conn = DriverManager.getConnection(url: String.format(format: dbURL, args: dbName), user: username, password);  
  
    return conn;  
}
```

getConnection(String dbName, String username, String password): This static method takes parameters for the database name, username, and password, and retrieves a database connection using these parameters. It first loads the SQL Server JDBC driver (com.microsoft.sqlserver.jdbc.SQLServerDriver). Then, it constructs the database URL (dbURL) using the provided database name, and establishes a connection using DriverManager.getConnection(). The connection is then returned.

Date

Instance Variables:

```
public class DateUtil {  
    public String pattern;  
    private final SimpleDateFormat sdf;
```

- pattern: A string representing the date pattern. It specifies the format in which dates will be formatted and parsed.
- sdf: An instance of SimpleDateFormat, which is a class provided by Java for formatting and parsing dates. It's initialized with the specified pattern.

Methods:

```
public String toString(Date date) {  
    return sdf.format(date);  
}  
  
public Date toDate(String stDate) throws ParseException {  
    return sdf.parse(stDate);  
}
```

- toString(Date date): This method takes a Date object as input and returns a string representation of the date in the specified format. It uses the format() method of SimpleDateFormat to achieve this.
- toDate(String stDate): This method takes a string representing a date (stDate) and parses it into a Date object using the specified format. It uses the parse() method of SimpleDateFormat to achieve this. It throws a ParseException if the input string cannot be parsed according to the specified format.

Message Box:

Methods:

```
public static void showErrorMessage(Component parent, String title, String message) {  
    JOptionPane.showMessageDialog(parentComponent: parent, message, title, messageType: JOptionPane.ERROR_MESSAGE);  
}  
  
public static void showErrorMessage(Component parent, String message) {  
    showErrorMessage(parent, title: "Error", message);  
}  
  
public static void showInformationMessage(Component parent, String title, String message) {  
    JOptionPane.showMessageDialog(parentComponent: parent, message, title, messageType: JOptionPane.INFORMATION_MESSAGE);  
}  
  
public static void showWarningMessage(Component parent, String title, String message) {  
    JOptionPane.showMessageDialog(parentComponent: parent, message, title, messageType: JOptionPane.WARNING_MESSAGE);  
}  
  
public static int showConfirmMessage(Component parent, String title, String message) {  
    return JOptionPane.showConfirmDialog(parentComponent: parent, message, title, optionType: JOptionPane.YES_NO_OPTION, messageType: JOptionPane.QUESTION_MESSAGE);  
}  
  
public static int showConfirmMessage(Component parent, String message) {  
    return showConfirmMessage(parent, title: "Confirmation", message);  
}
```

- `showErrorMessage(Component parent, String title, String message)`: This method displays an error message dialog with the specified message and title, using the parent Component as the dialog's parent. It uses `JOptionPane.showMessageDialog()` with `JOptionPane.ERROR_MESSAGE`.
- `showErrorMessage(Component parent, String message)`: Overloaded method that calls `showErrorMessage(Component parent, String title, String message)` with a default title of "Error".
- `showInformationMessage(Component parent, String title, String message)`: This method displays an information message dialog with the specified message and title, using the parent Component as the dialog's parent. It uses `JOptionPane.showMessageDialog()` with `JOptionPane.INFORMATION_MESSAGE`.
- `showWarningMessage(Component parent, String title, String message)`: This method displays a warning message dialog with the specified message and title, using the parent Component as the dialog's parent. It uses `JOptionPane.showMessageDialog()` with `JOptionPane.WARNING_MESSAGE`.
- `showConfirmMessage(Component parent, String title, String message)`: This method displays a confirmation message dialog with the specified message and title, using the parent Component as the dialog's parent. It presents options for Yes and No and returns the user's choice. It uses `JOptionPane.showConfirmDialog()` with `JOptionPane.YES_NO_OPTION` and `JOptionPane.QUESTION_MESSAGE`.
- `showConfirmMessage(Component parent, String message)`: Overloaded method that calls `showConfirmMessage(Component parent, String title, String message)` with a default title of "Confirmation".

Validator:

Expenditure Validator:

```
public static String validate(JTextField txtName, JTextField txtAmount,
    JTextField txtDate, JComboBox<ExpenditureType> cbxType){
    StringBuilder sb = new StringBuilder();

    if(Validator.isEmpty(component:txtName)){
        sb.append(str: "Name must be entered\n");
    }

    if(Validator.isEmpty(component:txtAmount)){
        sb.append(str: "Amount must be entered\n");
    }

    if(!Validator.isMin(component:txtAmount,min: 0)){
        sb.append(str: "Amount must be greater than 0 or invalid number\n");
    }

    if(Validator.isEmpty(component:txtDate)){
        sb.append(str: "Date must be entered\n");
    }

    if(!Validator.isDate(component:txtDate)){
        sb.append(str: "Invalid date\n");
    }

    return sb.isEmpty()? null : sb.toString();
}
```

Method:

- validate(JTextField txtName, JTextField txtAmount, JTextField txtDate, JComboBox<ExpenditureType> cbxType): This method takes several Swing components as input parameters: text fields for name, amount, and date, and a combo box for expenditure type. It validates the input values and returns a string containing any validation errors found. If no errors are found, it returns null.

Validation Rules:

- Name Validation: It checks if the name field is empty.
- Amount Validation: It checks if the amount field is empty and if the entered amount is greater than 0.
- Date Validation: It checks if the date field is empty and if the entered date is in a valid format.

Validator Class:

- The validation methods (isEmpty, isMin, isDate) are called from the Validator class, which is assumed to contain static methods for common validation tasks. This suggests a modular approach to validation, where generic

validation rules are encapsulated in a separate utility class (Validator), promoting code reuse and maintainability.

Return Value:

- The method returns `null` if no validation errors are found. If there are validation errors, it returns a string containing all the error messages concatenated together.

Expenditure Type Validator:

Method:

```
public static String validate(JTextField txtName) {  
    StringBuilder sb = new StringBuilder();  
  
    if (Validator.isEmpty(component:txtName)) {  
        sb.append(str: "Name must be entered");  
    }  
  
    return sb.isEmpty()? null : sb.toString();  
}
```

- `validate(JTextField txtName)`: This method takes a single parameter, a `JTextField` representing the input field for the expenditure type name. It validates the input value and returns a string containing any validation error found. If no errors are found, it returns `null`.

Validation Rule:

- Name Validation: It checks if the name field is empty.

Validator Class:

- The validation method (`isEmpty`) is called from the `Validator` class, which is assumed to contain static methods for common validation tasks. This suggests a modular approach to validation, where generic validation rules are encapsulated in a separate utility class (`Validator`), promoting code reuse and maintainability.

StringBuilder Usage:

- Inside the `validate` method, a `StringBuilder` is used to accumulate validation error messages. If any validation errors are encountered, they are appended to this `StringBuilder`.

Return Value:

- The method returns `null` if no validation errors are found. If there are validation errors, it returns a string containing the error message.

MessageBox Usage:

- In case of validation errors, the `MessageBox` class from `kk.pe.util` package seems to be utilized to display error messages. However, it's not directly evident

from this class how `MessageBox` is used, as it's called inside other parts of the application.

Validator:

`isEmpty` Method:

```
public static boolean isEmpty(JComponent component){  
    if(component instanceof JTextField){  
        JTextField txt = (JTextField) component;  
        if(txt.getText().equals("")){  
            txt.setBackground(bg: Color.yellow);  
            return true;  
        }else{  
            txt.setBackground(bg: Color.white);  
        }  
    }  
    return false;  
}
```

- `public static boolean isEmpty(JComponent component)`: This method checks if a text field is empty. It takes a `JComponent` as a parameter, which is typically a `JTextField`. If the text field is empty, it sets its background color to yellow and returns `true`; otherwise, it sets the background color to white and returns `false`.

isMin Method:

```
public static boolean isMin(JComponent component, double min){
    if(component instanceof JTextField){
        JTextField txt = (JTextField) component;

        try{
            double value = Double.parseDouble(s: txt.getText());

            if(value >= min){
                txt.setBackground(bg: Color.white);
                return true;
            }
            txt.setBackground(bg: Color.yellow);
        } catch (Exception e){
            txt.setBackground(bg: Color.yellow);
        }
    }
    return false;
}
```

- public static boolean isMin(JComponent component, double min): This method checks if the value in a text field is greater than or equal to a specified minimum value. It takes a JComponent as a parameter, which is usually a JTextField, and the minimum value. If the value in the text field is greater than or equal to the minimum value, it sets the background color to white and returns true; otherwise, it sets the background color to yellow and returns false.


isDate Method:

```
public static boolean isDate(JComponent component) {  
    if (component instanceof JTextField) {  
        JTextField txt = (JTextField) component;  
  
        try {  
            DateUtil dateUtil = new DateUtil();  
            Date date = dateUtil.toDate(stDate: txt.getText());  
  
            txt.setBackground(bg: Color.white);  
            return true;  
  
        } catch (Exception e) {  
            txt.setBackground(bg: Color.yellow);  
        }  
    }  
    return false;  
}
```

- public static boolean isDate(JComponent component): This method checks if the value in a text field represents a valid date. It takes a JComponent as a parameter, typically a JTextField. Inside the method, it attempts to parse the text field's content into a Date object using the DateUtil class. If the parsing is successful, indicating a valid date, it sets the background color to white and returns true; otherwise, it sets the background color to yellow and returns false.

Demo

Login: Click login to be able to perform the application's features.


×


Login

Username:

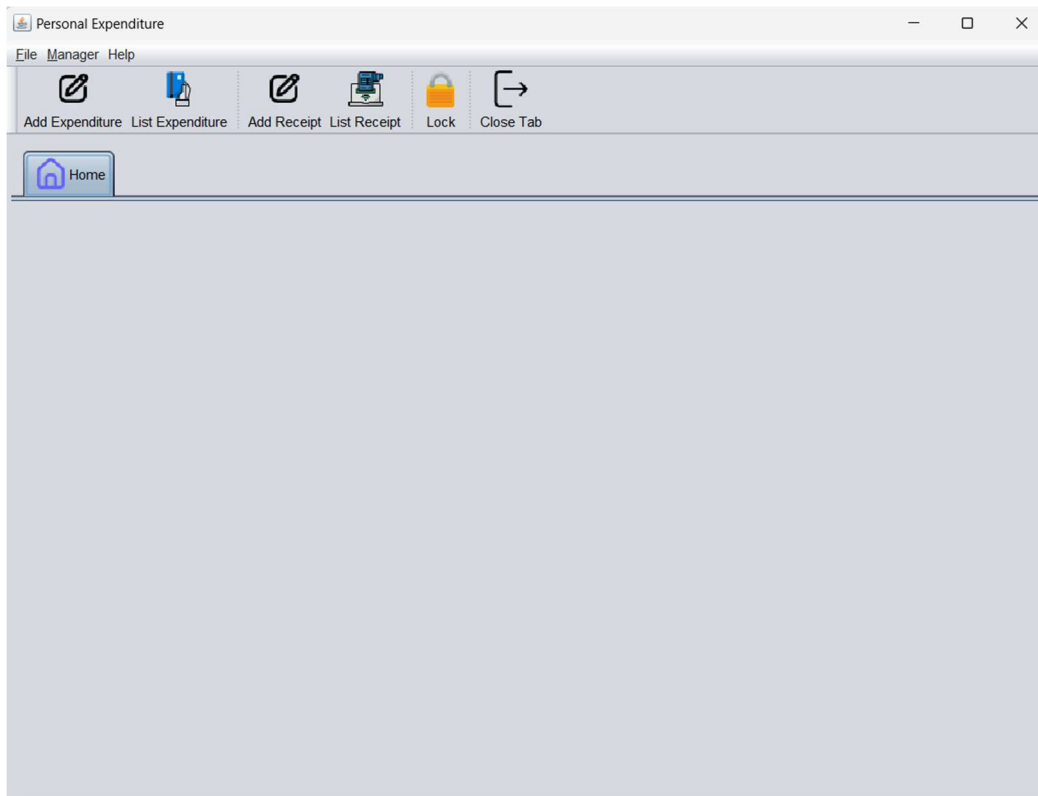
Password:

☐ Remember me

 Login

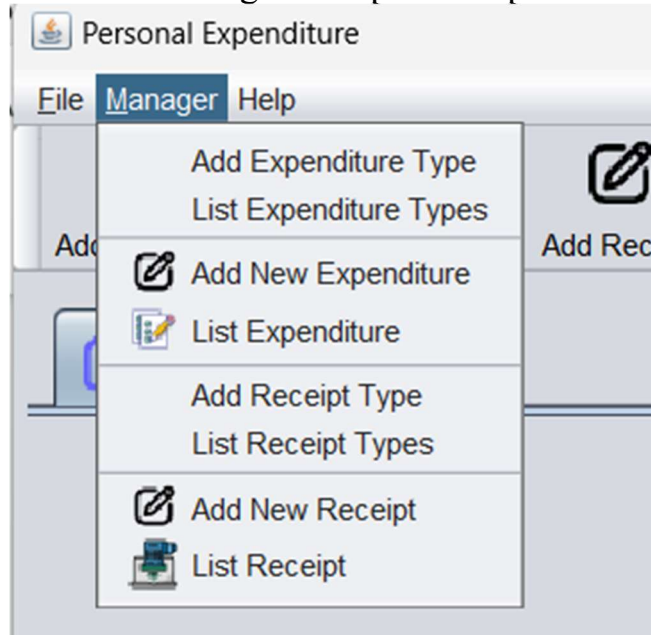
 Close

Main menu:



Add expenditure type:

- Click to “Manager” to open the options window



- Click “Add Expenditure Type”

Add New Expenditure Type

ID:

Name:

Buttons: + New, Edit, Save, Update, Delete, List

- Input Name and click Save. Then you can go to List Expenditure Types to see the data just entered.

List Expenditure Types

ID	Name
4	Food
5	Food 2
7	Food 18
8	Food 10
10	Food 12
12	food 14
13	Study Fees

- At the same time, the system will also update the Type section in Add Expenditure

Add New Expenditure

ID:

Name:

Amount:

Type:

Food - 4

Food - 4

Food 2 - 5

Food 18 - 7

Food 10 - 8

Food 12 - 10

food 14 - 12

Study Fees - 13

Date:

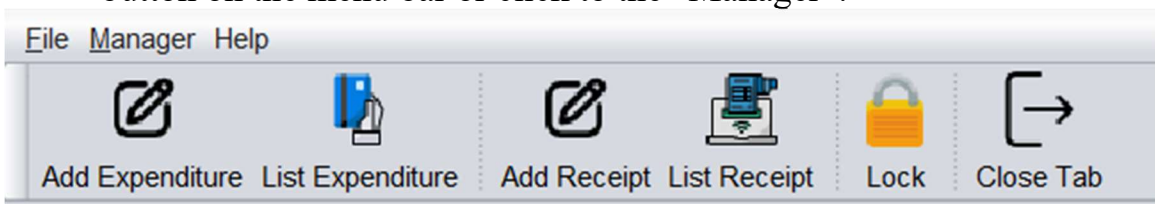
Note:

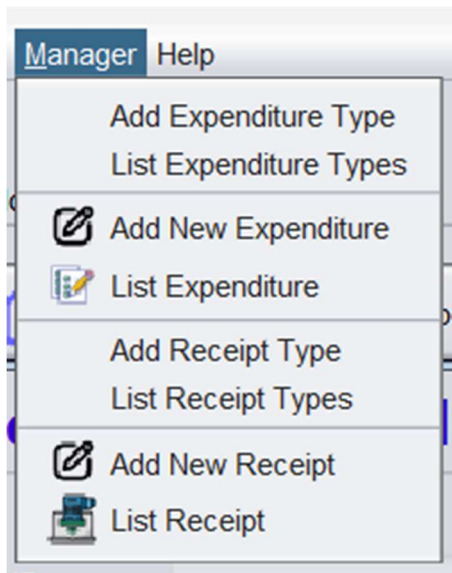
- You can also Edit or Delete the Type in List Expenditure Type by right clicking on the Type data:

List Expenditure Types

ID	Name
4	Food - 4
5	Food 2 - 5
7	Food 18 - 7
8	Food 10 - 8
10	Food 12 - 10
12	food 14 - 12
13	Study Fees - 13

- To add new Expenditure, you can access this window through the button on the menu bar or click to the “Manager”:





- You can also see, edit, or delete the expenditure data on List expenditure

ID	Name	Amount	Date	Type
4	Test	100.0	2022-12-12	4
5	Test3	100.0	2023-01-01	5
7	Test	1.0	2023-12-12	4