INTRODUCTION TO JAVA GUI – PART 2

Swing Package

Overview

- Using the Bank System example, we will enhance the first GUI version of Bank System to make it more user friendly. Doing such will demonstrate the following:
  - `JTable`
  - `JOptionPane`
  - `JLabel and Images`

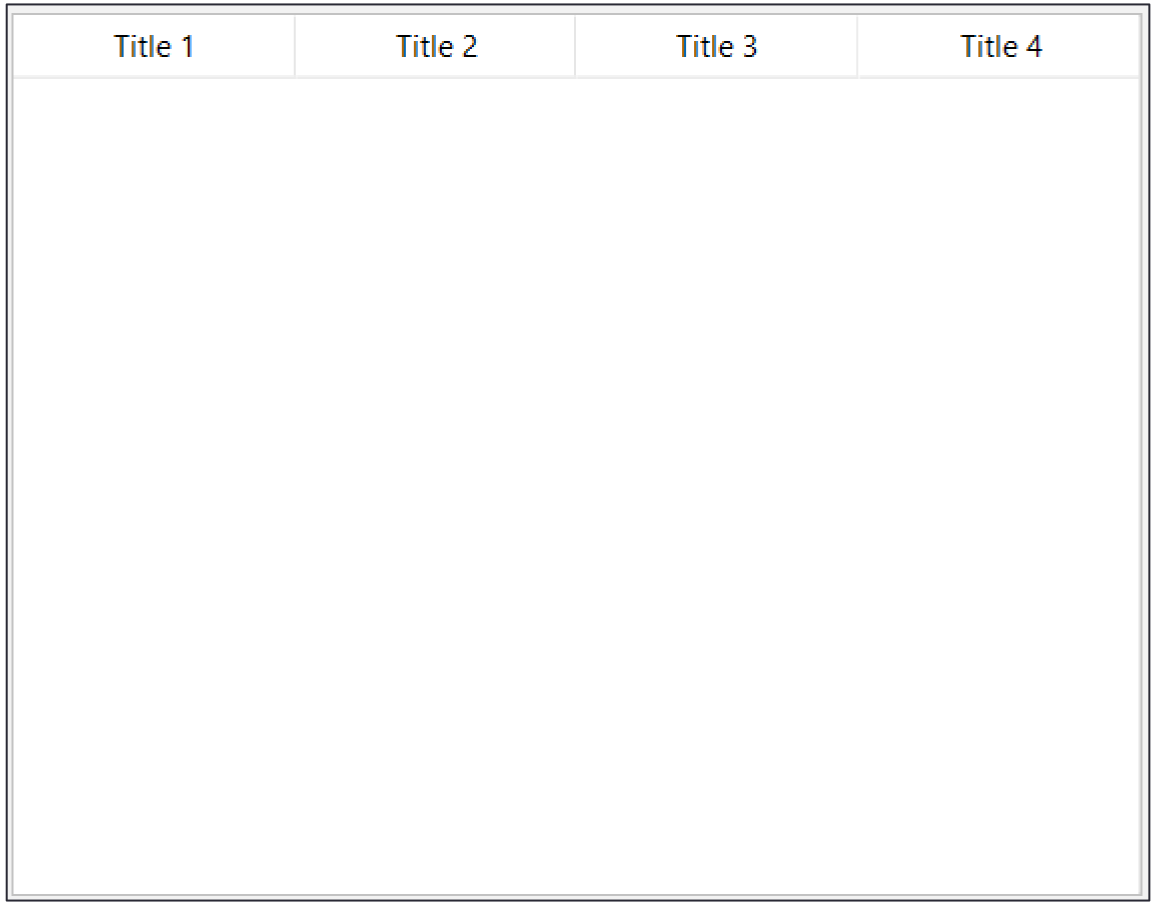
JTABLE

Required library: `javax.swing.table.TableColumn`

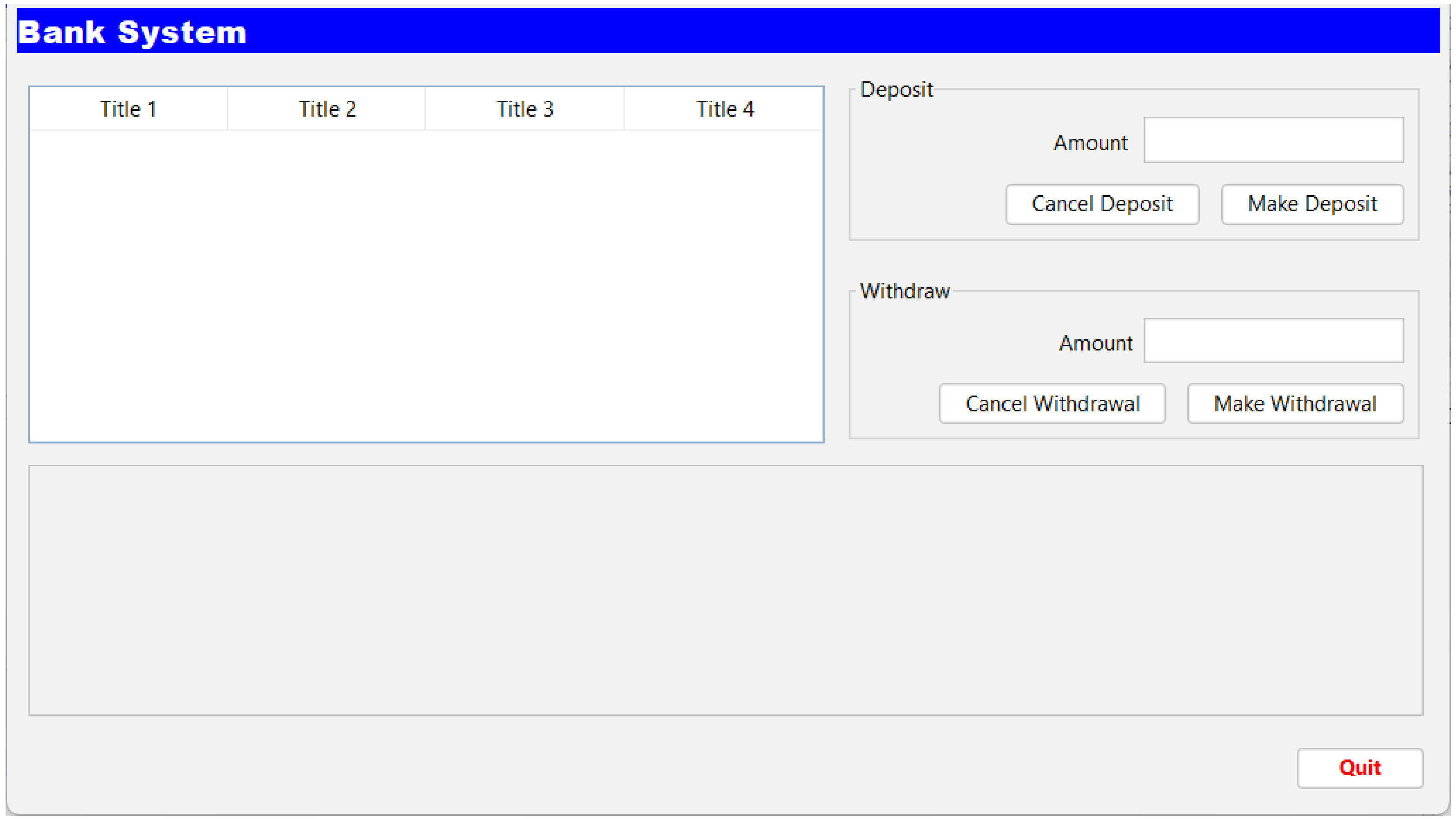
Example project: `BankSystem`

JTable

- Useful for tabular data
  - For example, listing data of all accounts
- Display `JFrame` in Design view and use Pallete to select and position the table
  - When created a `JTable` is embedded into a `JScrollPane`
  - Which will display scroll bars if needed
- By default, the table will have four columns
  - With general headings
- Change the name of `JTable` to `accountTable`



Modified GUI to include JTable



Bit of programming required (1)

Bank Account Data

- We need to specify how our Bank Account data is to be used with the JTable
- This includes the number of columns required, the column headings and the number of rows, and of course the content of each row and column

Abstract Table Model

- We need to create a customised version of an **AbstractTableModel**
- Which will essentially provide the rules by which the table will be populated
- And at the same time will allow the application to update the table
- Updating the table does not automatically update the underlying collection, i.e., ArrayList
- Updating the underlying collection, i.e., ArrayList, does not automatically update the table

Bank Account Table Model

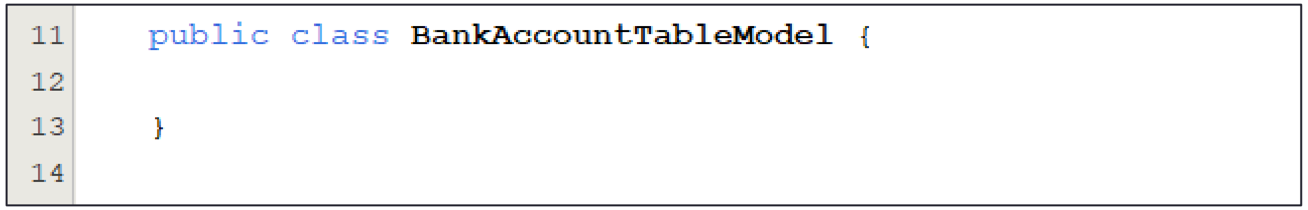
- Our version of the **AbstractTableModel** will be called `BankAccountTableModel`
- It will accept our collection of account and use that to populate the table
- We must provide implementations for a few inherited abstract methods

Bit of programming required (2)

- The number of items in the collection will determine the number of rows in the table
  - I.e., `accountList.size()`
- The number of columns will be based on which fields of the Data Class we wish to display
  - For **BankSystem** we will display all three fields: `name`, `number` and `balance`
  - We will create an array of column titles, using the fields names, in sentence case
  - The length of the array will be used to set the number of columns
- The table will then request the information from our collection
  - It will accept our collection of **BankAccount** objects and use that to populate the table
  - Meaning each row of the table will be a specific **BankAccount** object
  - And each column will be a specific field
- We also want to override the method used to set the column titles: `getColumnName()`

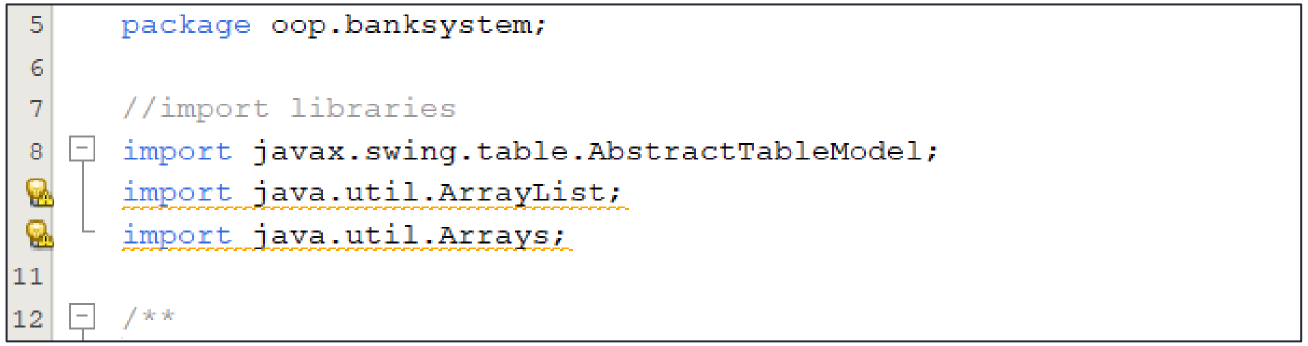
Customised Abstract Table Model

- Add as a new Java class named:
  - BankAccountTableModel



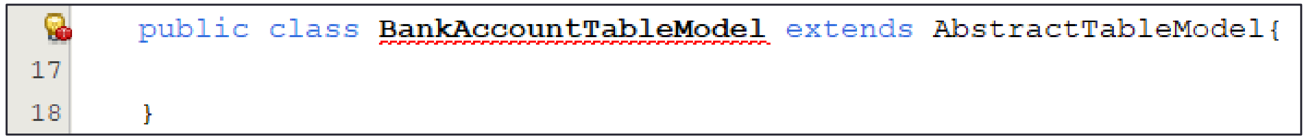
```
public class BankAccountTableModel {

}

```

- After package statement import:
  - java.util.ArrayList
  - java.util.Arrays
  - javax.swing.table.AbstractTableModel



```
package oop.banksystem;

//import libraries
import javax.swing.table.AbstractTableModel;
import java.util.ArrayList;
import java.util.Arrays;

/**
```

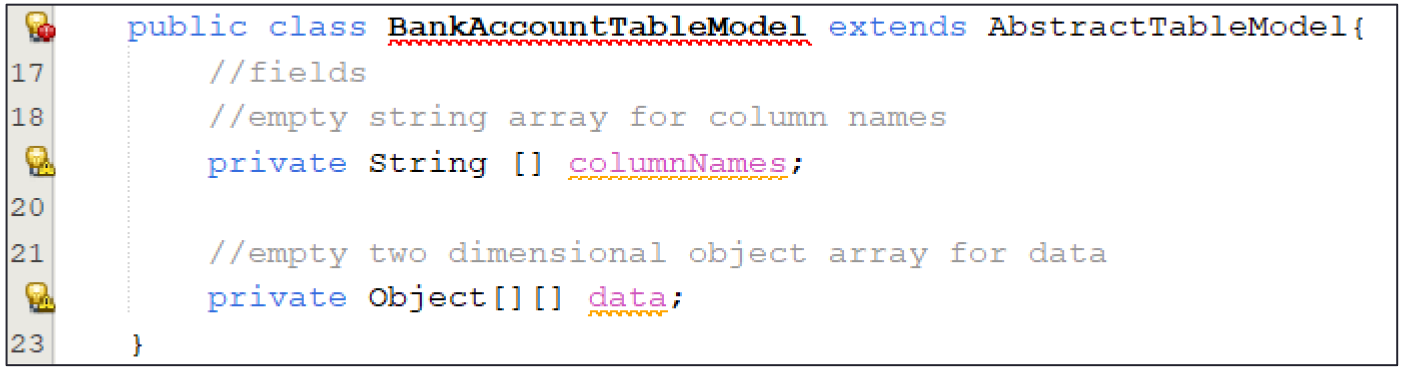
- Modify class header to include:
  - extends AbstractTableModel



```
public class BankAccountTableModel extends AbstractTableModel{

}
```

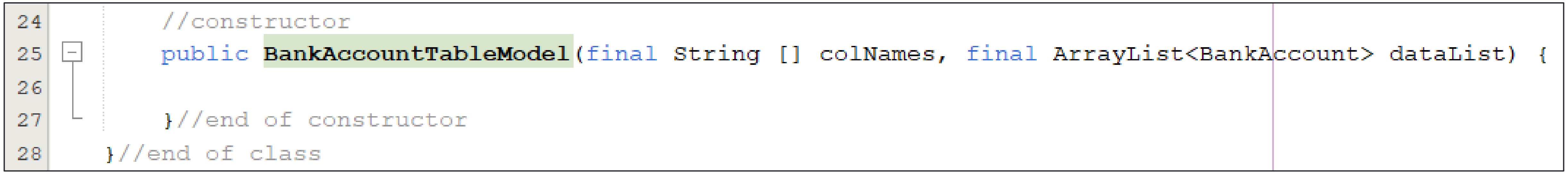
BankAccountTableModel Fields

- The class will have two fields:
  - A String array
  - A two-dimensional Object array
  - Unfortunately, JTable cannot work directly with an ArrayList



```java
    public class BankAccountTableModel extends AbstractTableModel{
        //fields
        //empty string array for column names
        private String [] columnNames;

        //empty two dimensional object array for data
        private Object[][] data;
    }
```

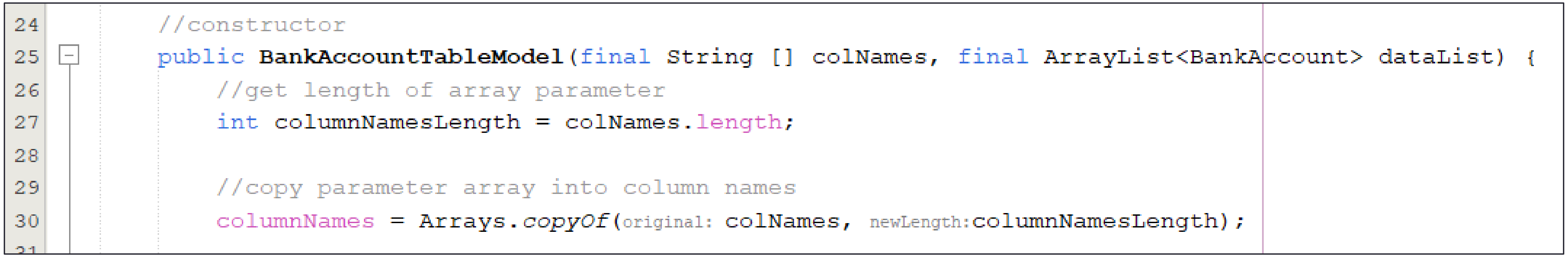
- Code the header and footer of the class constructor method, with two parameters
  - A String array
  - A generic ArrayList



```java
    //constructor
    public BankAccountTableModel(final String [] colNames, final ArrayList<BankAccount> dataList) {

    }//end of constructor
}//end of class
```

Constructor Method – Column Names

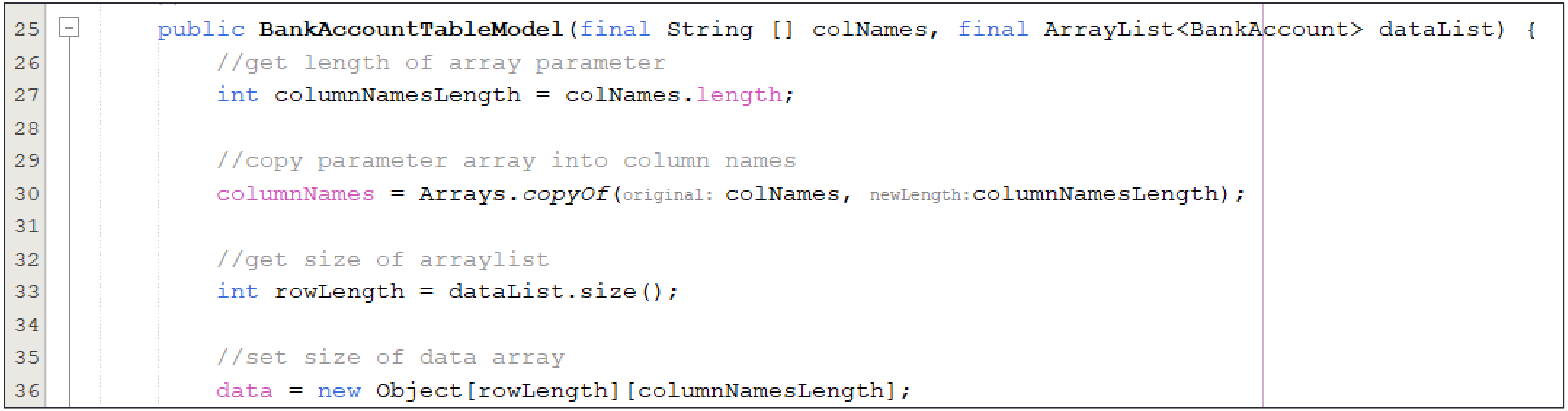
- To populate the `columnNames` array:
  - Get length of the colNames array parameter
  - Use the **Arrays.copyOf** method to copy value of each element of `colNames` into corresponding element of `columnNames`



```
    //constructor
    public BankAccountTableModel(final String [] colNames, final ArrayList<BankAccount> dataList) {
        //get length of array parameter
        int columnNamesLength = colNames.length;

        //copy parameter array into column names
        columnNames = Arrays.copyOf(original: colNames, newLength:columnNamesLength);
```

Constructor Method – 2D Object array size

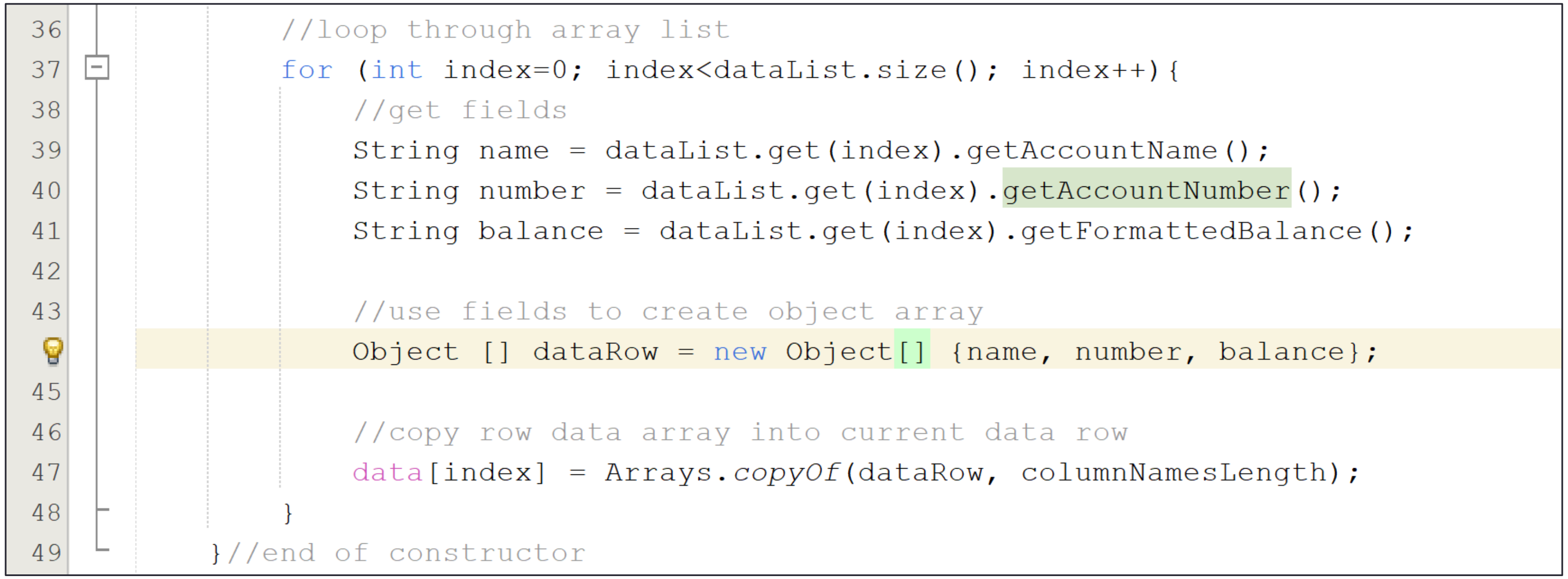
- To populate the `object` two-dimensional array:
  - Get length of the ArrayList parameter
  - Instantiate the size of both dimensions of the 2D array



```
    public BankAccountTableModel(final String [] colNames, final ArrayList<BankAccount> dataList) {
        //get length of array parameter
        int columnNamesLength = colNames.length;

        //copy parameter array into column names
        columnNames = Arrays.copyOf(original: colNames, newLength:columnNamesLength);

        //get size of arraylist
        int rowLength = dataList.size();

        //set size of data array
        data = new Object[rowLength][columnNamesLength];
```

Constructor Method – Populate 2D Array

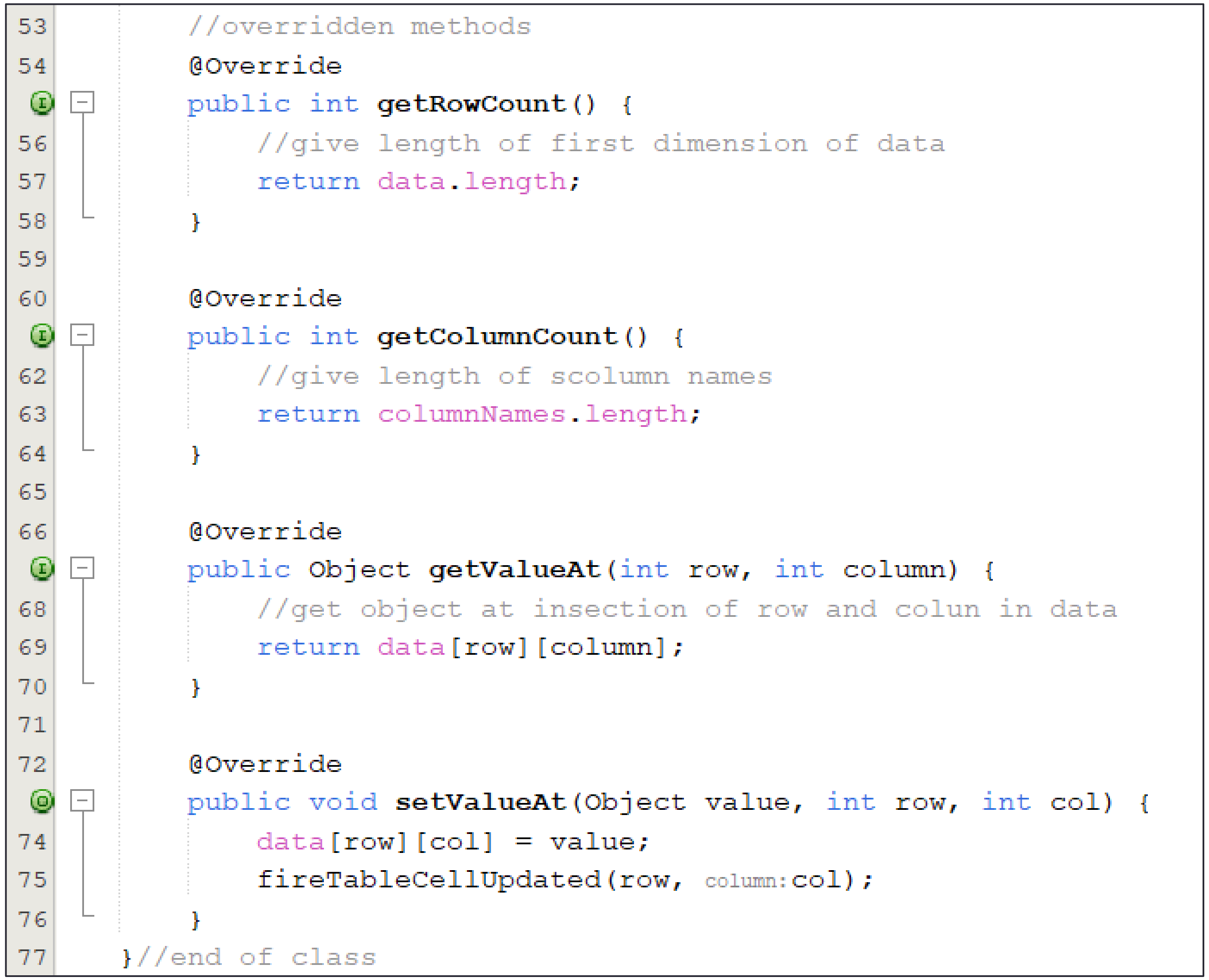
- Use a for loop to iterate through ArrayList parameter and in each pass of the loop
  - Get value of each field of current ArrayList item
  - Use fields to create an object array
  - Set current row of 2D array to be a copy of the object array



```
        //loop through array list
        for (int index=0; index<dataList.size(); index++){
            //get fields
            String name = dataList.get(index).getAccountName();
            String number = dataList.get(index).getAccountNumber();
            String balance = dataList.get(index).getFormattedBalance();

            //use fields to create object array
            Object [] dataRow = new Object[] {name, number, balance};

            //copy row data array into current data row
            data[index] = Arrays.copyOf(dataRow, columnNamesLength);
        }
    }//end of constructor
```

Implementing inherited methods

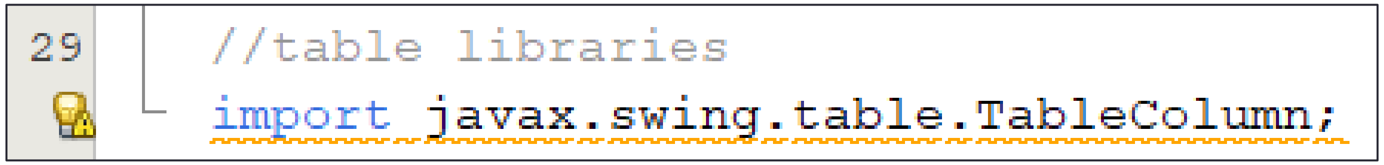
- In extending an Abstract class, we must ensure that each inherited abstract method is coded.
  - `getRowCount()` returns number of rows in the table, i.e. the length of the 2D array
  - `getColumnCount()` returns number of columns in the table, i.e. the length of columnNames
  - `getValueAt()` returns the value of the table cell at a specific of row and column index
  - `setValueAt()` changes the value of the table cell at a specific of row and column index



```
    //overridden methods
    @Override
    public int getRowCount() {
        //give length of first dimension of data
        return data.length;
    }

    @Override
    public int getColumnCount() {
        //give length of scolumn names
        return columnNames.length;
    }

    @Override
    public Object getValueAt(int row, int column) {
        //get object at insection of row and colun in data
        return data[row][column];
    }

    @Override
    public void setValueAt(Object value, int row, int col) {
        data[row][col] = value;
        fireTableCellUpdated(row, column:col);
    }
}//end of class
```

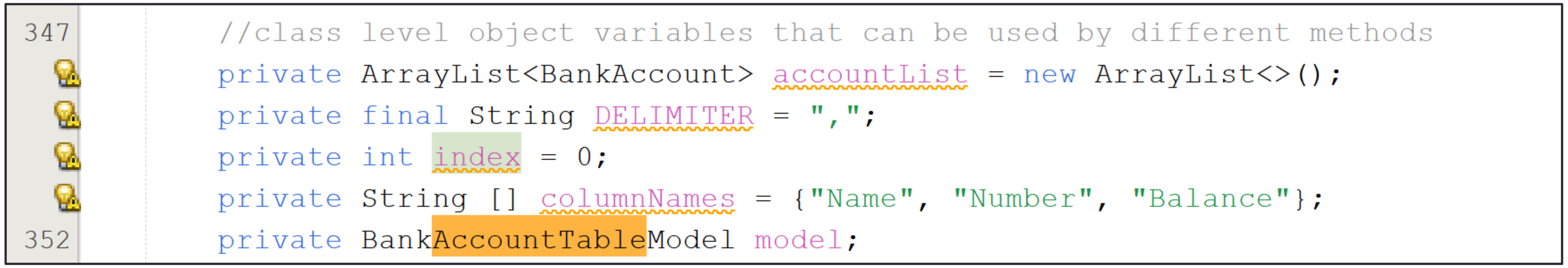
Main Class Changes

- Import an additional library: `javax.swing.table.TableColumn`



```
//table libraries
import javax.swing.table.TableColumn;
```

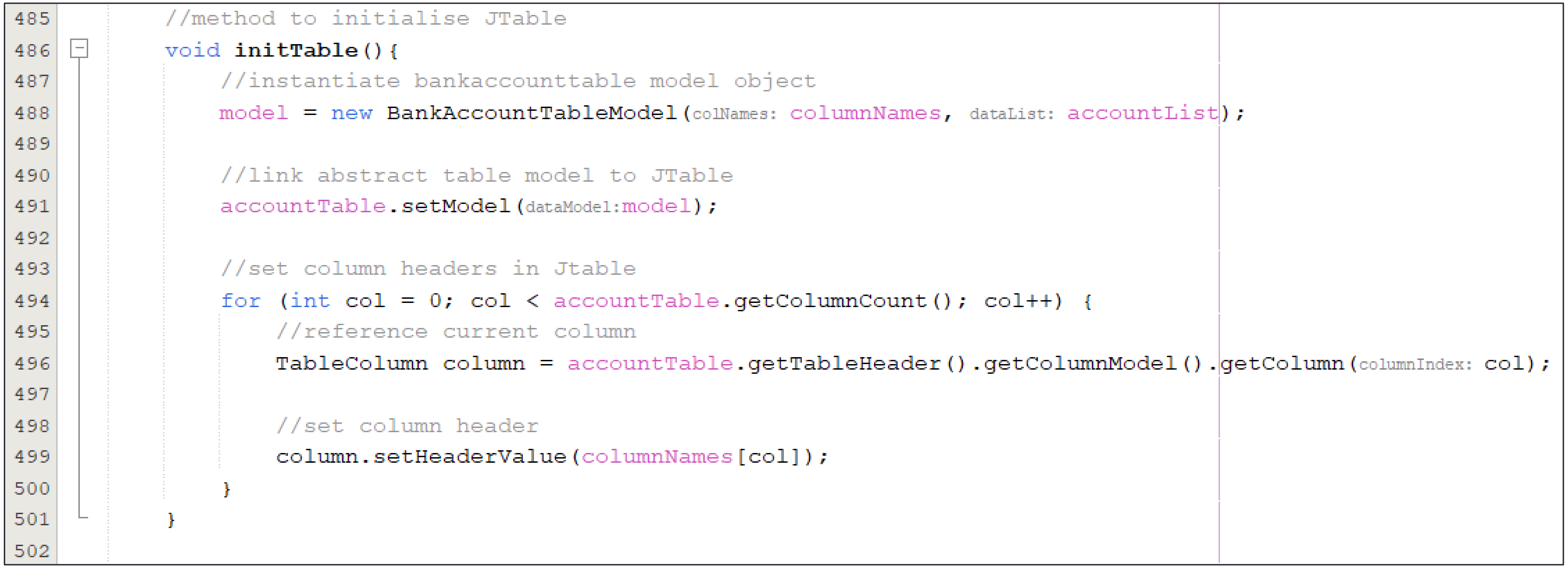
- Class level variables
  - Modify `index` to be `0`
  - Declare and populate `columnNames` String array
  - Create a reference of the type `BankAccountTableModel`



```
//class level object variables that can be used by different methods
private ArrayList<BankAccount> accountList = new ArrayList<>();
private final String DELIMITER = ",";
private int index = 0;
private String [] columnNames = {"Name", "Number", "Balance"};
private BankAccountTableModel model;
```

User defined method

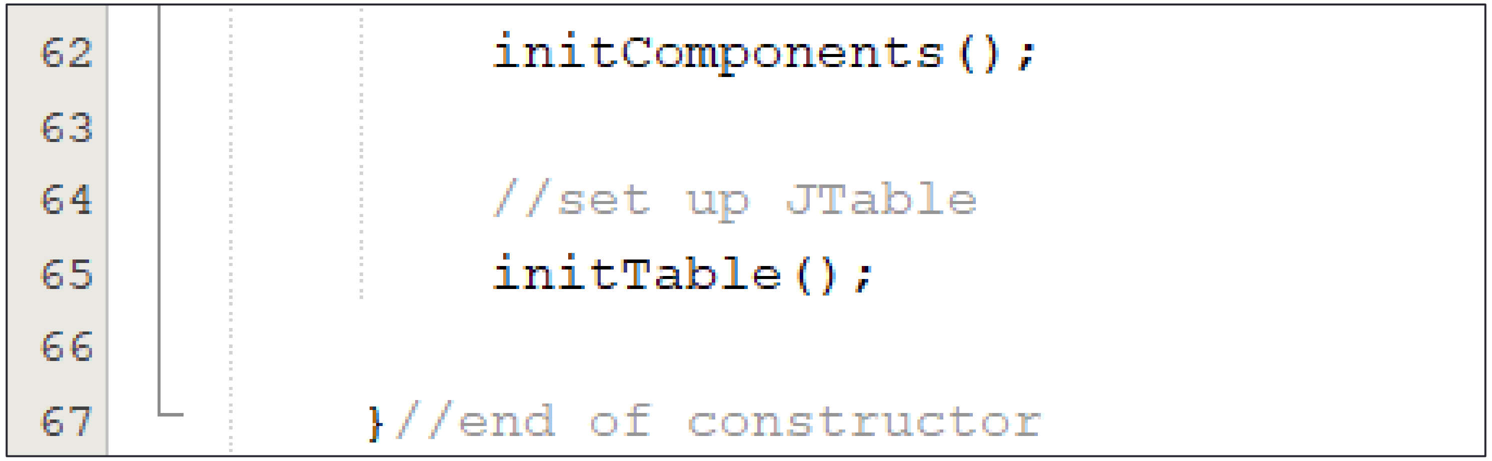
- Using a user defined method `initTable()`
  - We link the JTable to the abstract model, which populates table rows with data
  - Use abstract model to set header for each column



```java
    //method to initialise JTable
    void initTable(){
        //instantiate bankaccounttable model object
        model = new BankAccountTableModel(colNames: columnNames, dataList: accountList);

        //link abstract table model to JTable
        accountTable.setModel(dataModel:model);

        //set column headers in Jtable
        for (int col = 0; col < accountTable.getColumnCount(); col++) {
            //reference current column
            TableColumn column = accountTable.getTableHeader().getColumnModel().getColumn(columnIndex: col);

            //set column header
            column.setHeaderValue(columnNames[col]);
        }
    }
```

Modify Main Class Constructor

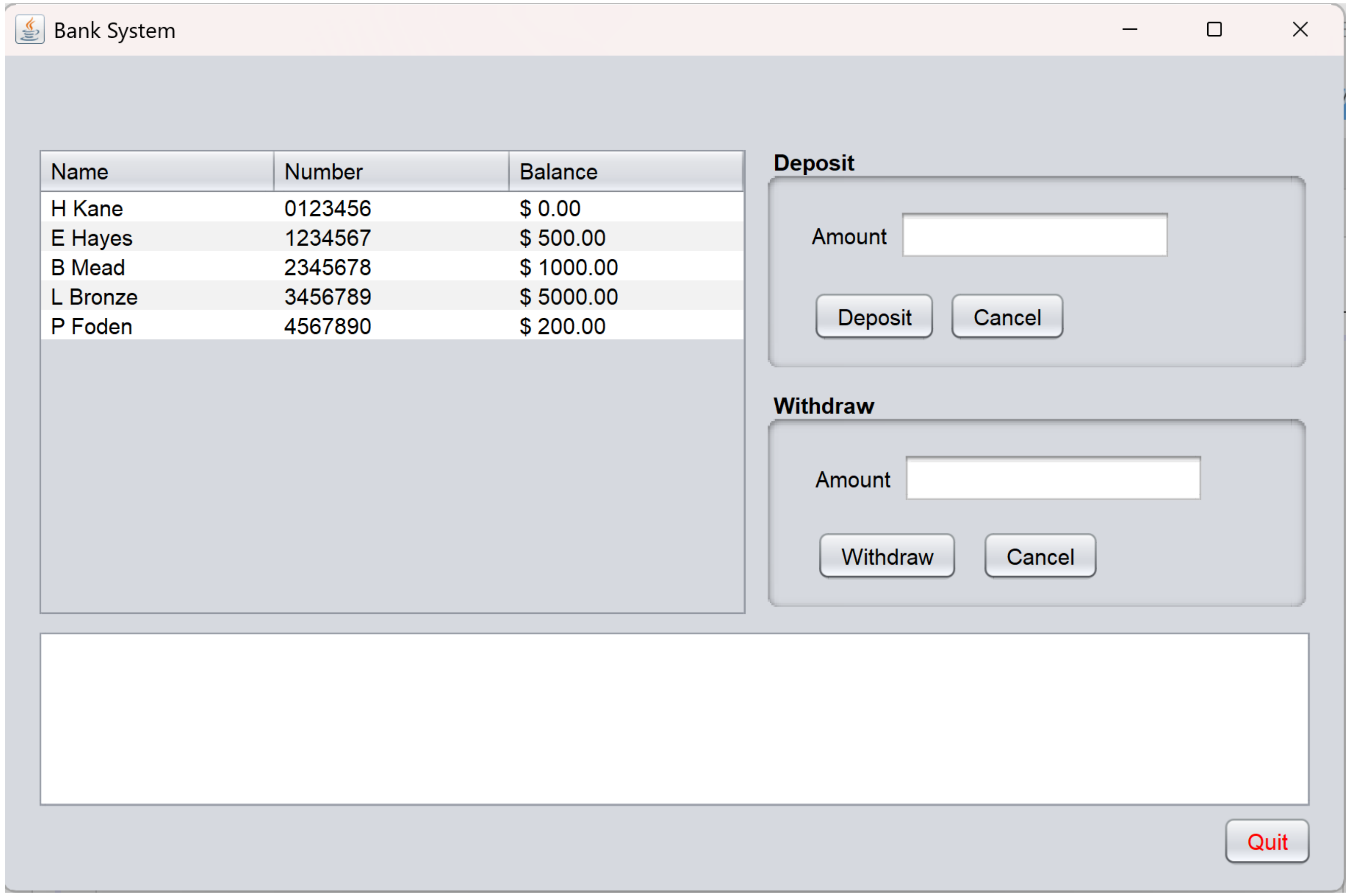
- The `initTable()` method is invoked from the Main class constructor
  - So that the table is populated before the GUI is displayed



```
        initComponents();

        //set up JTable
        initTable();

    }//end of constructor
```

Runtime

Add more data to your ArrayList and re-run your program



Name	Number	Balance
H Kane	0123456	$ 0.00
E Hayes	1234567	$ 500.00
B Mead	2345678	$ 1000.00
L Bronze	3456789	$ 5000.00
P Foden	4567890	$ 200.00

JTABLE EVENT HANDLING

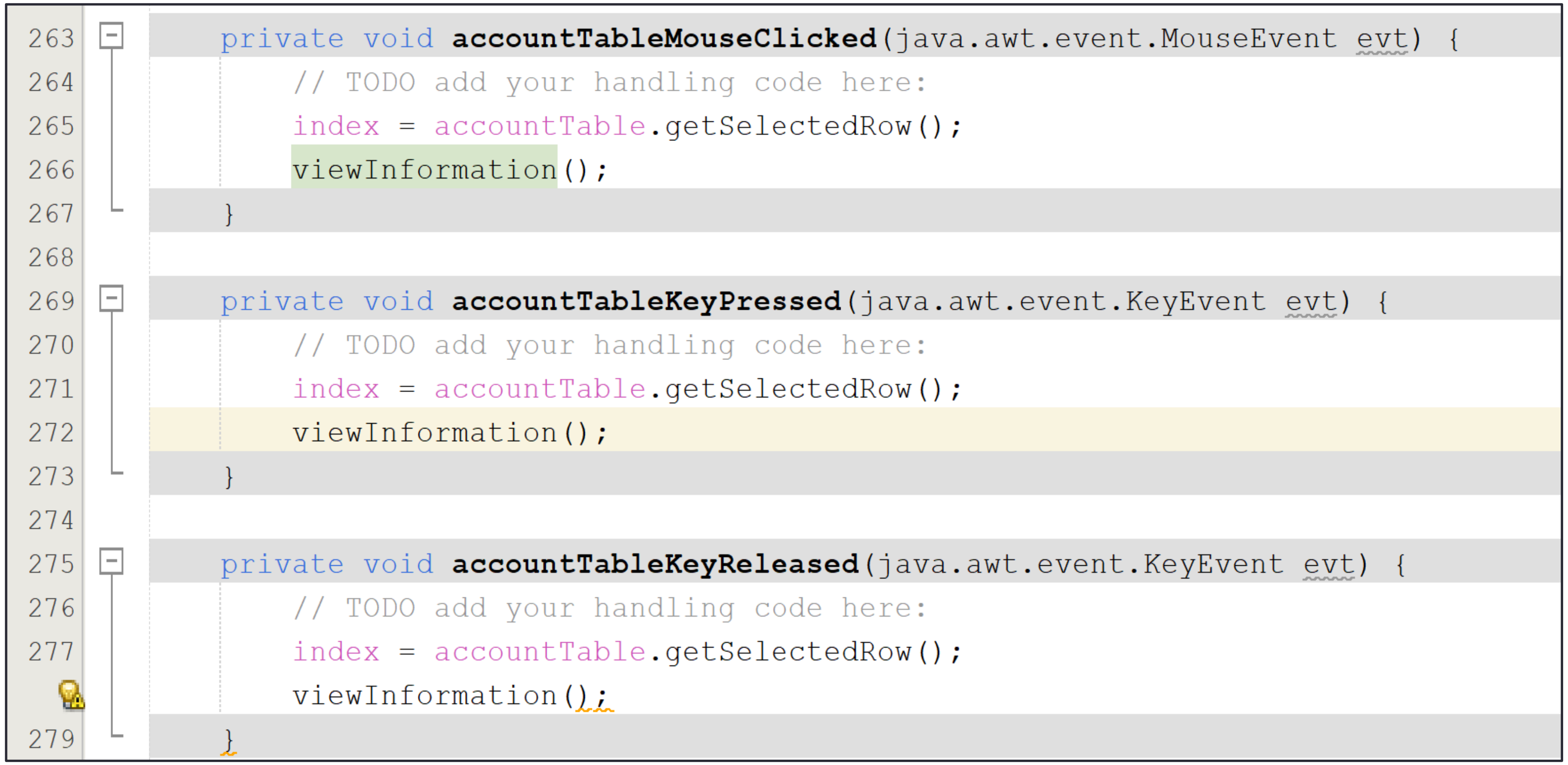
Table Row Selection

- The user initially selects a bank account by clicking on a row in the table
  - After initial selection user can either click, or use cursor keys to select previous or next row

- Once a row is selected the following should occur
  - Class level **index** variable is set to the index of the selected row
    - `index = bankAccountTable.getSelectedRow();`
  - The **viewInformation()** method is invoked
    - `viewInformation();`

JTable Event Handlers

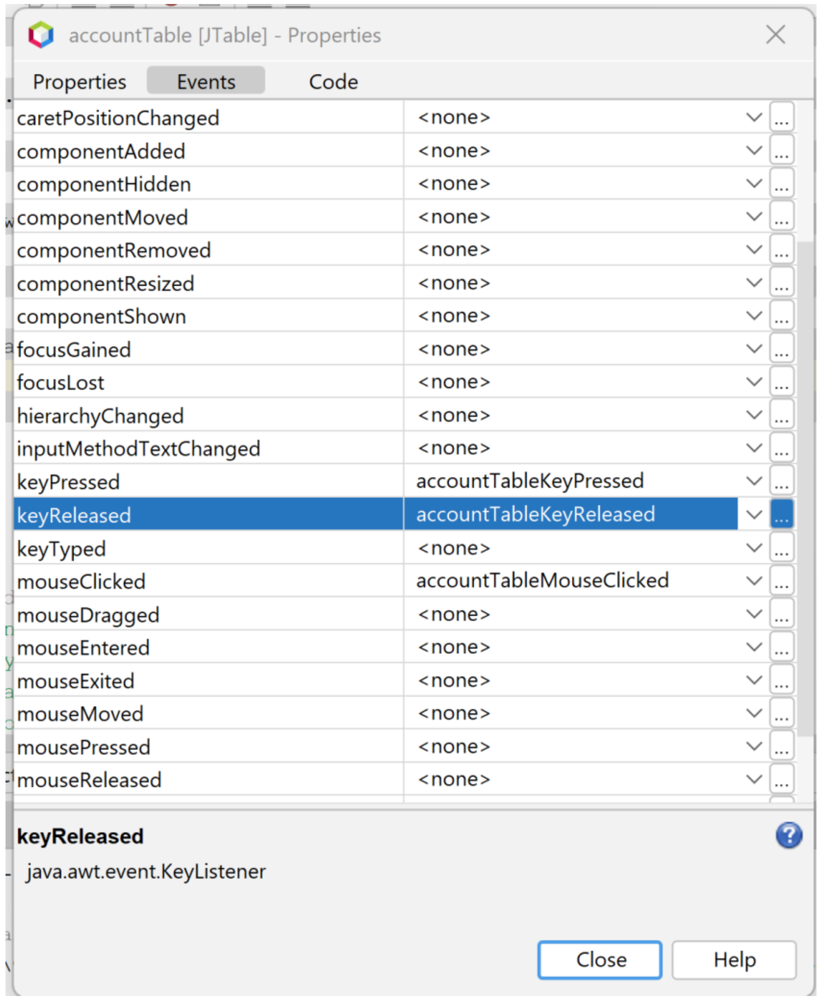
- Three events need to be created for the JTable:
  - Mouse Clicked
  - Key Pressed
  - Key Released



```java
    private void accountTableMouseClicked(java.awt.event.MouseEvent evt) {
        // TODO add your handling code here:
        index = accountTable.getSelectedRow();
        viewInformation();
    }

    private void accountTableKeyPressed(java.awt.event.KeyEvent evt) {
        // TODO add your handling code here:
        index = accountTable.getSelectedRow();
        viewInformation();
    }

    private void accountTableKeyReleased(java.awt.event.KeyEvent evt) {
        // TODO add your handling code here:
        index = accountTable.getSelectedRow();
        viewInformation();
    }
```

JTable Event Handlers

- You can add them by right clicking on JTable and select Properties --> Events:
  - Mouse Clicked
  - Key Pressed
  - Key Released

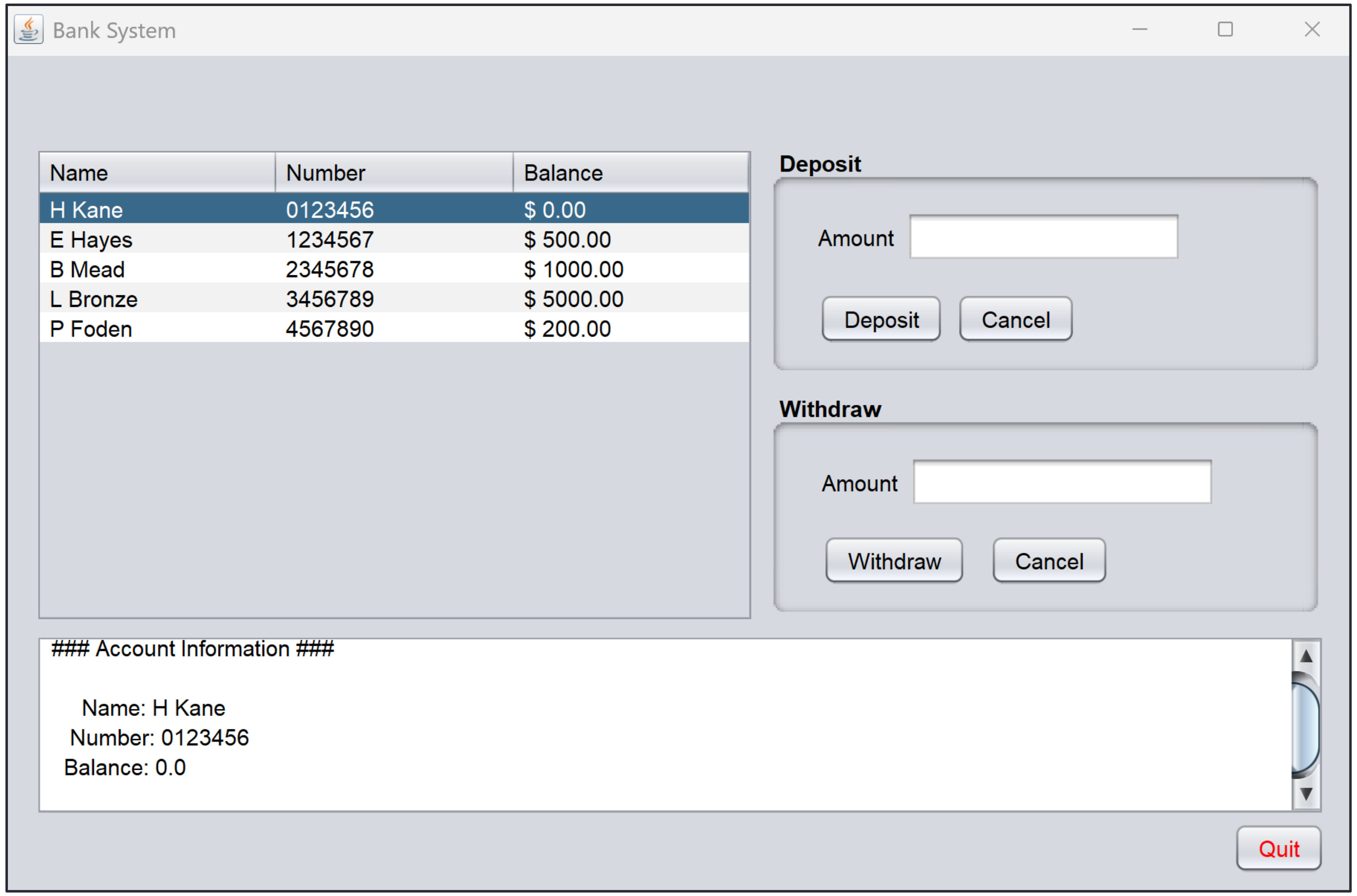


caretPositionChanged	<none>
componentAdded	<none>
componentHidden	<none>
componentMoved	<none>
componentRemoved	<none>
componentResized	<none>
componentShown	<none>
focusGained	<none>
focusLost	<none>
hierarchyChanged	<none>
inputMethodTextChanged	<none>
keyPressed	accountTableKeyPressed
keyReleased	accountTableKeyReleased
keyTyped	<none>
mouseClicked	accountTableMouseClicked
mouseDragged	<none>
mouseEntered	<none>
mouseExited	<none>
mouseMoved	<none>
mousePressed	<none>
mouseReleased	<none>

keyReleased

java.awt.event.KeyListener

Runtime



Name	Number	Balance
H Kane	0123456	$ 0.00
E Hayes	1234567	$ 500.00
B Mead	2345678	$ 1000.00
L Bronze	3456789	$ 5000.00
P Foden	4567890	$ 200.00

```
### Account Information ###

   Name: H Kane
  Number: 0123456
 Balance: 0.0
```

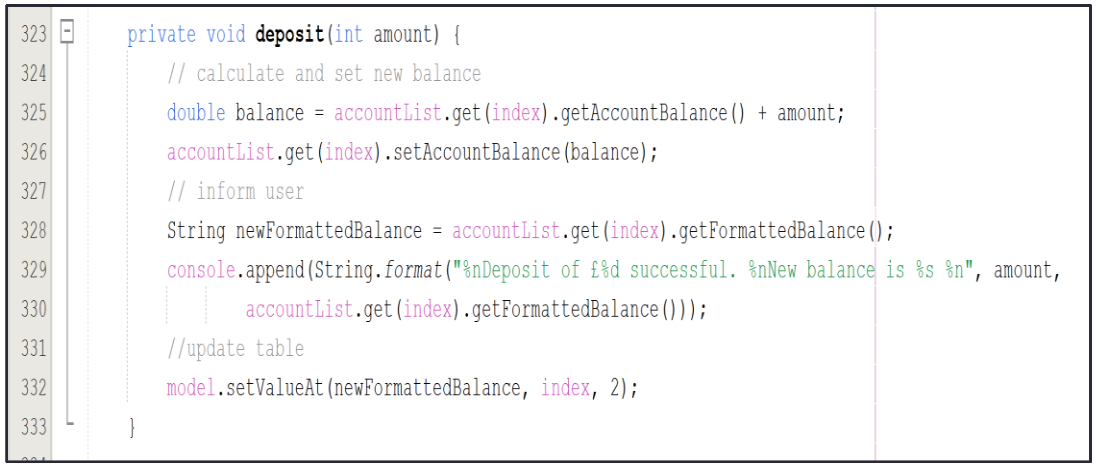
Deposit and Withdraw

- If user makes successful Deposit or Withdrawal
  - We need to update the table model to update displayed information in JTable

- In both methods
  - Get new formatted balance
    - `String newBalance = accountList.get(index).getFormattedBalance();`
  - Use the overridden abstract table model method of **setValueAt**
    - `model.setValueAt(newBalance, index, 2);`

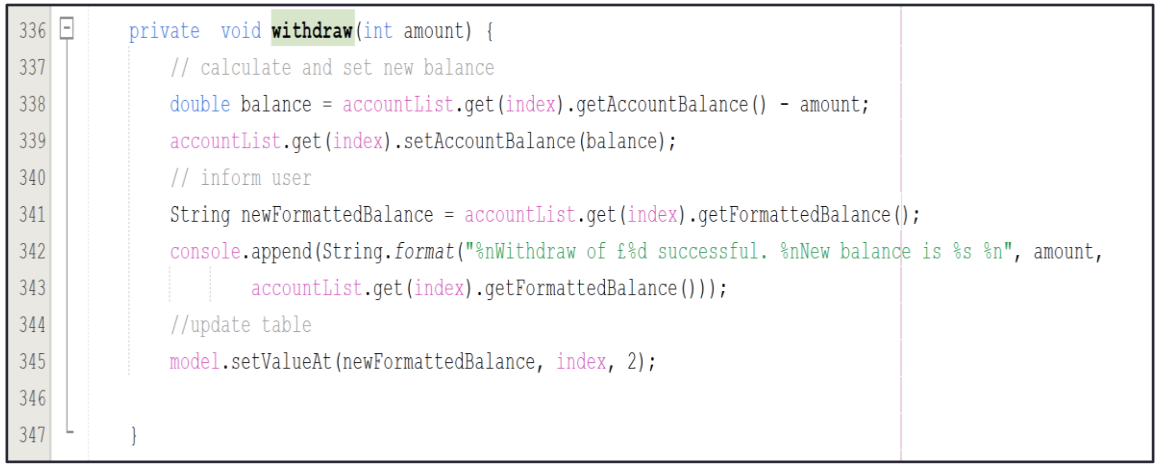
Modified Deposit and Withdraw

Deposit method



```java
private void deposit(int amount) {
    // calculate and set new balance
    double balance = accountList.get(index).getAccountBalance() + amount;
    accountList.get(index).setAccountBalance(balance);
    // inform user
    String newFormattedBalance = accountList.get(index).getFormattedBalance();
    console.append(String.format("%nDeposit of £%d successful. %nNew balance is %s %n", amount,
            accountList.get(index).getFormattedBalance()));
    //update table
    model.setValueAt(newFormattedBalance, index, 2);
}
```

Withdraw method



```java
private  void withdraw(int amount) {
    // calculate and set new balance
    double balance = accountList.get(index).getAccountBalance() - amount;
    accountList.get(index).setAccountBalance(balance);
    // inform user
    String newFormattedBalance = accountList.get(index).getFormattedBalance();
    console.append(String.format("%nWithdraw of £%d successful. %nNew balance is %s %n", amount,
            accountList.get(index).getFormattedBalance()));
    //update table
    model.setValueAt(newFormattedBalance, index, 2);

}
```

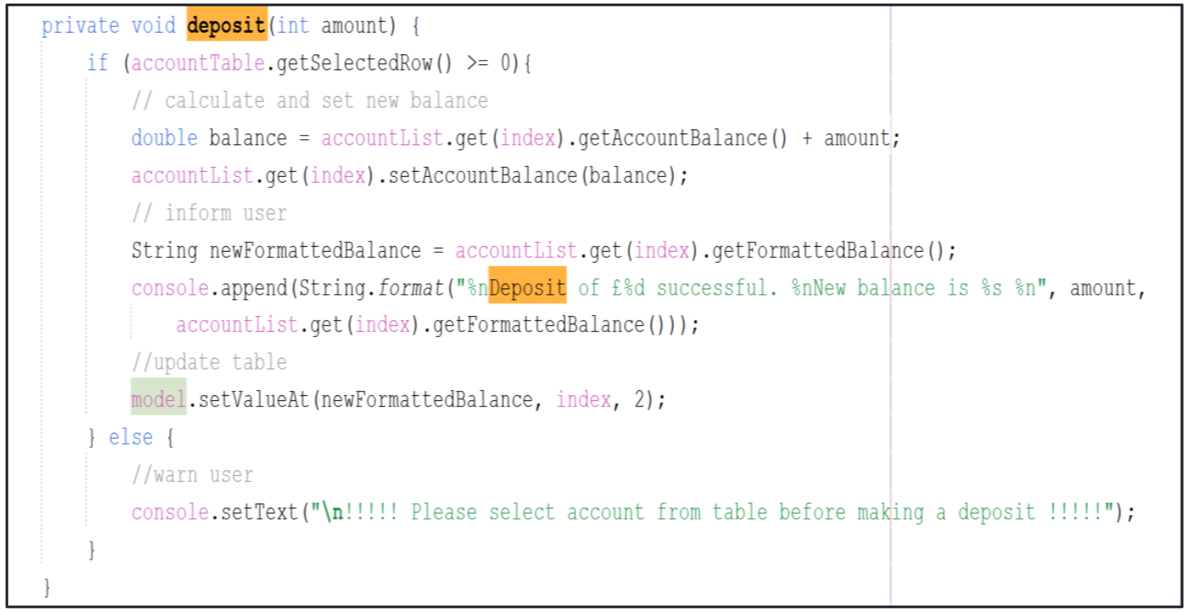
Deposit and Withdraw Event Handlers

- Possibility of user, attempting a deposit or withdrawal before selecting an account through the JTable
  - This can be prevented through a table validation check
- Check if a row has been selected
  - If so, allow the operation to proceed
  - If not, warn the user and do not proceed

```
if (accountTable.getSelectedRow()>= 0){
   //continue
} else {
   //warn user
}
```

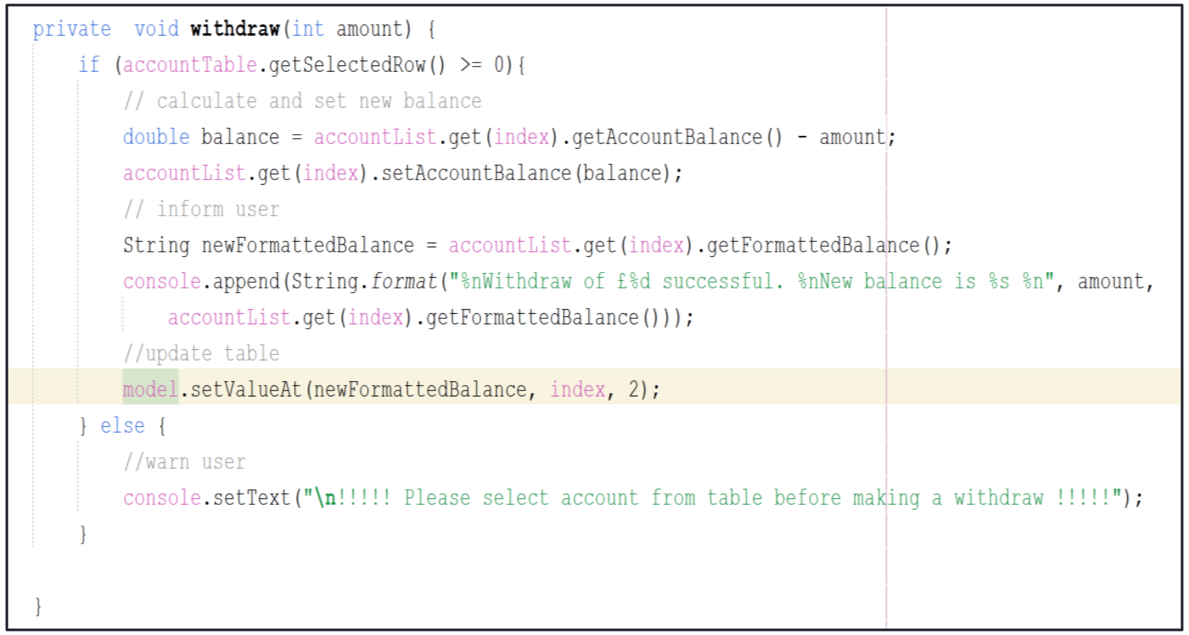
Modified Deposit and Withdraw Event Handlers

Deposit Event Handler



```
private void deposit(int amount) {
    if (accountTable.getSelectedRow() >= 0){
        // calculate and set new balance
        double balance = accountList.get(index).getAccountBalance() + amount;
        accountList.get(index).setAccountBalance(balance);
        // inform user
        String newFormattedBalance = accountList.get(index).getFormattedBalance();
        console.append(String.format("%nDeposit of £%d successful. %nNew balance is %s %n", amount,
            accountList.get(index).getFormattedBalance()));
        //update table
        model.setValueAt(newFormattedBalance, index, 2);
    } else {
        //warn user
        console.setText("\n!!!!! Please select account from table before making a deposit !!!!!");
    }
}
```

Withdraw Event Handler



```
private  void withdraw(int amount) {
    if (accountTable.getSelectedRow() >= 0){
        // calculate and set new balance
        double balance = accountList.get(index).getAccountBalance() - amount;
        accountList.get(index).setAccountBalance(balance);
        // inform user
        String newFormattedBalance = accountList.get(index).getFormattedBalance();
        console.append(String.format("%nWithdraw of £%d successful. %nNew balance is %s %n", amount,
            accountList.get(index).getFormattedBalance()));
        //update table
        model.setValueAt(newFormattedBalance, index, 2);
    } else {
        //warn user
        console.setText("\n!!!!! Please select account from table before making a withdraw !!!!!");
    }

}
```

POP UP DIALOGS

Required library: `javax.swing.JOptionPane`

Example Project: `BankSystem`

Four types of dialogs

message dialog	confirmation dialog	input dialog	option dialog
• Displays a message to the user, along with an OK button.	• Ask the user a question along with buttons Yes, No, and Cancel.	• Prompts the user type input into a text field	• A general dialog, which can be customised

- We are going to use an input dialog for both deposit and withdraw operations
  - For deposit we will use standard input dialog, as we need to allow any integer deposit
  - For withdraw we will use a customised input dialog, which will only allow withdrawal amounts in range 10 to 200, in multiples of ten. Assuming account has sufficient funds

JOptionPane

- The `JOptionPane` class is static, meaning that once the class is imported
  - We can use the class without creating an instance object of the class, like class wrappers such as `Integer`

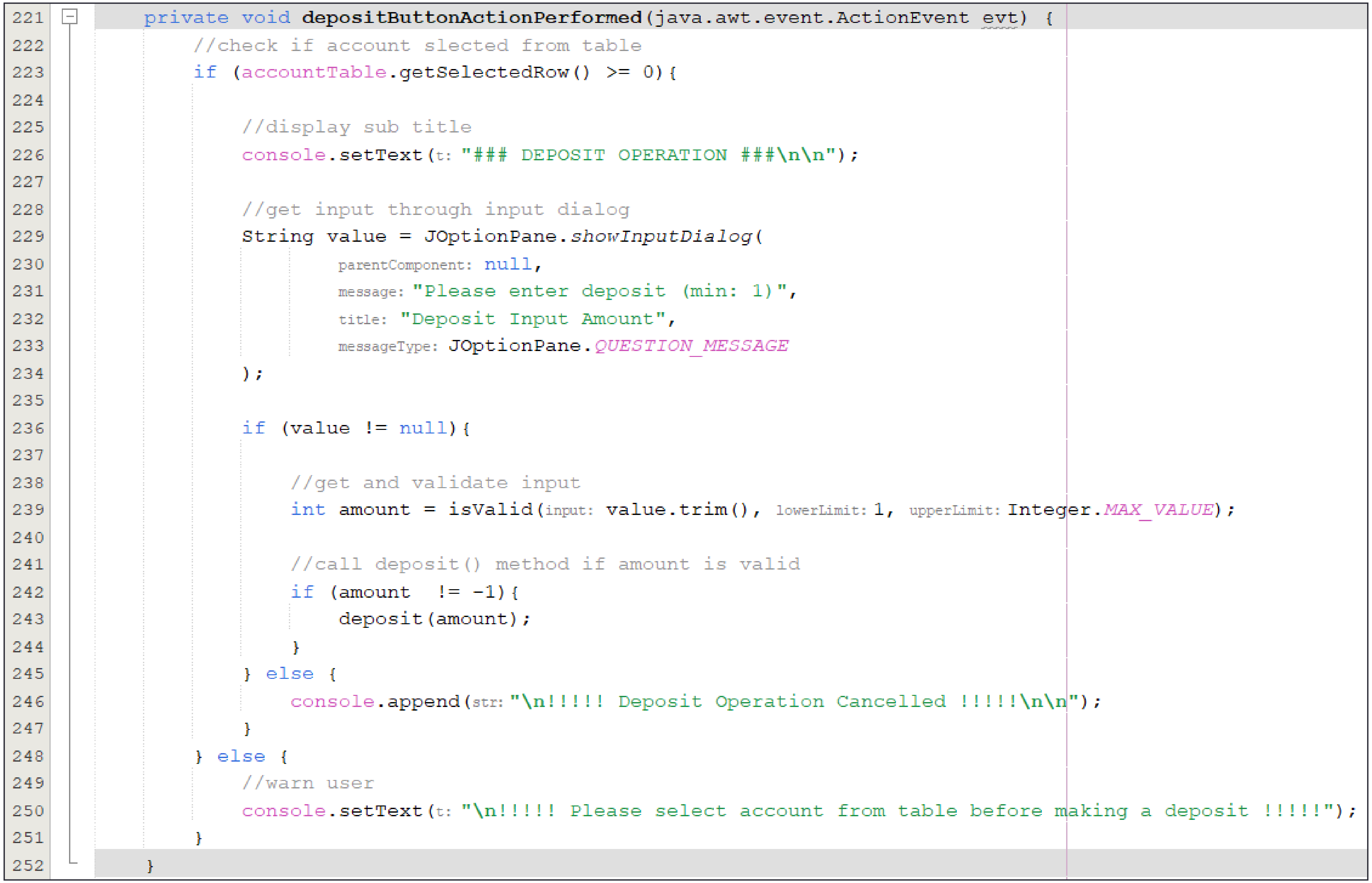
- We are going to use an input dialog for both deposit and withdraw operations
  - For deposit we will use standard input dialog, as we need to allow any integer deposit
  - For withdraw we will use a customised input dialog, which will only allow withdrawal amounts in range 10 to 100, in multiples of ten. Assuming account has sufficient funds

Deposit

- Design change involves only keeping deposit button and moving it near to the quit button
  - With two buttons not in a panel, a new panel (buttonPanel) is created to hold these buttons

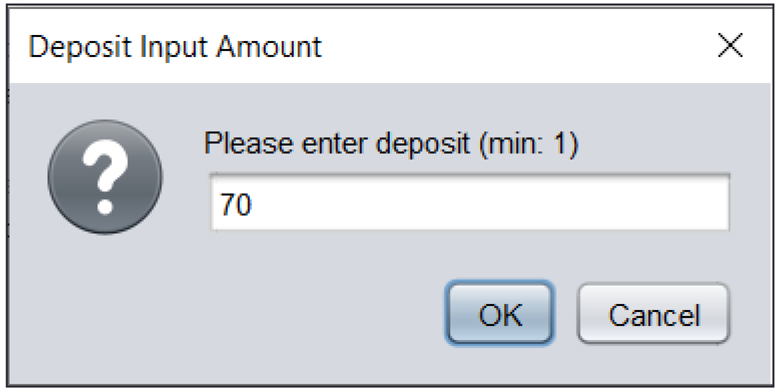
- Coding change only involves **event handler** for deposit button
  - Remove statements involving deleted components e.g. depositTextfield
  - Use Input Dialog to get input: `JOptionPane.showInputDialog`
  - Check if input is null, if so, warn user and stop
  - Process input

Deposit Event Handler Code



```
private void depositButtonActionPerformed(java.awt.event.ActionEvent evt) {
    //check if account slected from table
    if (accountTable.getSelectedRow() >= 0){

        //display sub title
        console.setText(t: "### DEPOSIT OPERATION ###\n\n");

        //get input through input dialog
        String value = JOptionPane.showInputDialog(
                parentComponent: null,
                message: "Please enter deposit (min: 1)",
                title: "Deposit Input Amount",
                messageType: JOptionPane.QUESTION_MESSAGE
        );

        if (value != null){

            //get and validate input
            int amount = isValid(input: value.trim(), lowerLimit: 1, upperLimit: Integer.MAX_VALUE);

            //call deposit() method if amount is valid
            if (amount  != -1){
                deposit(amount);
            }
        } else {
            console.append(str: "\n!!!!! Deposit Operation Cancelled !!!!!\n\n");
        }
    } else {
        //warn user
        console.setText(t: "\n!!!!! Please select account from table before making a deposit !!!!!");
    }
}
```

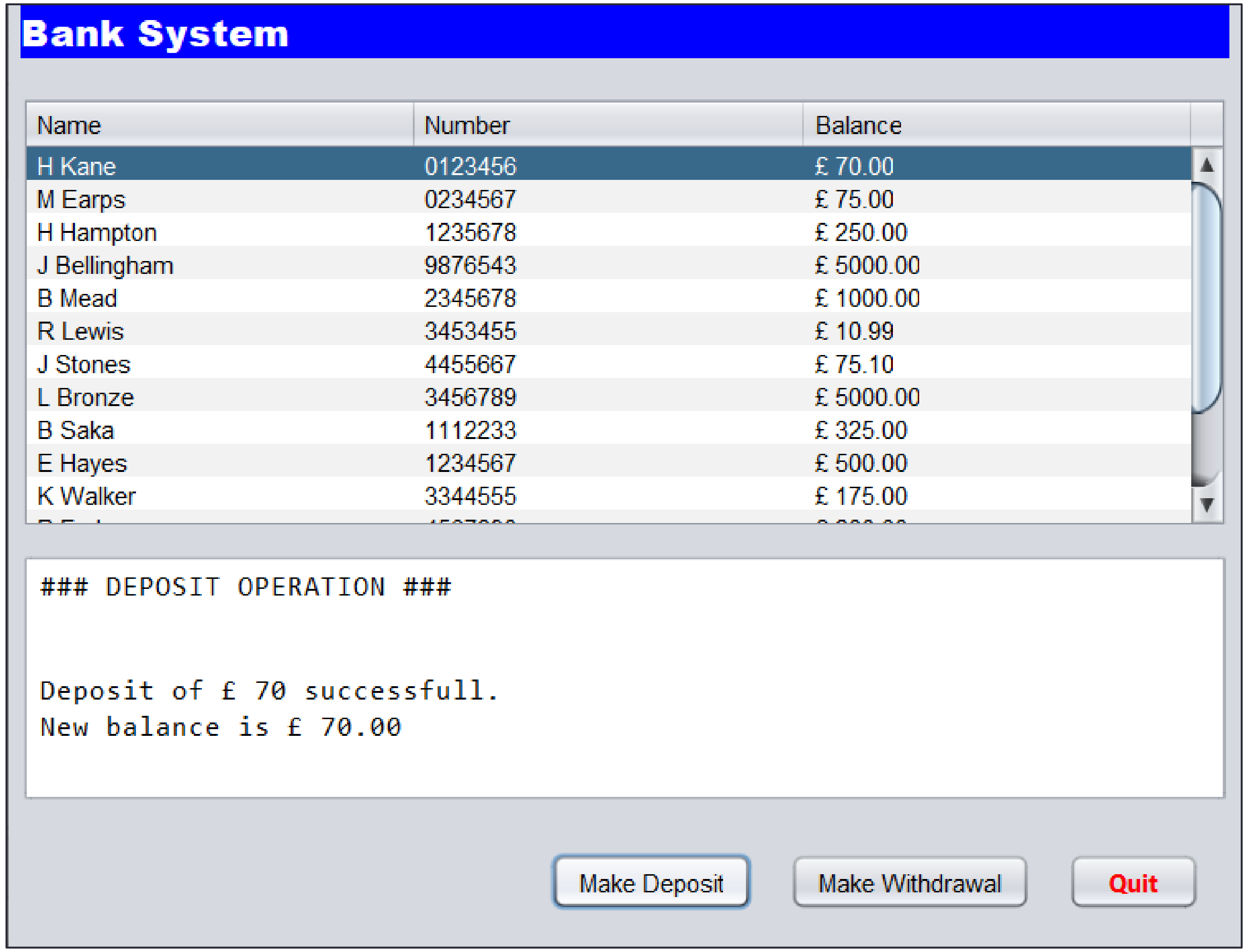
Deposit Runtime



Deposit Input Amount

Please enter deposit (min: 1)

70



Bank System

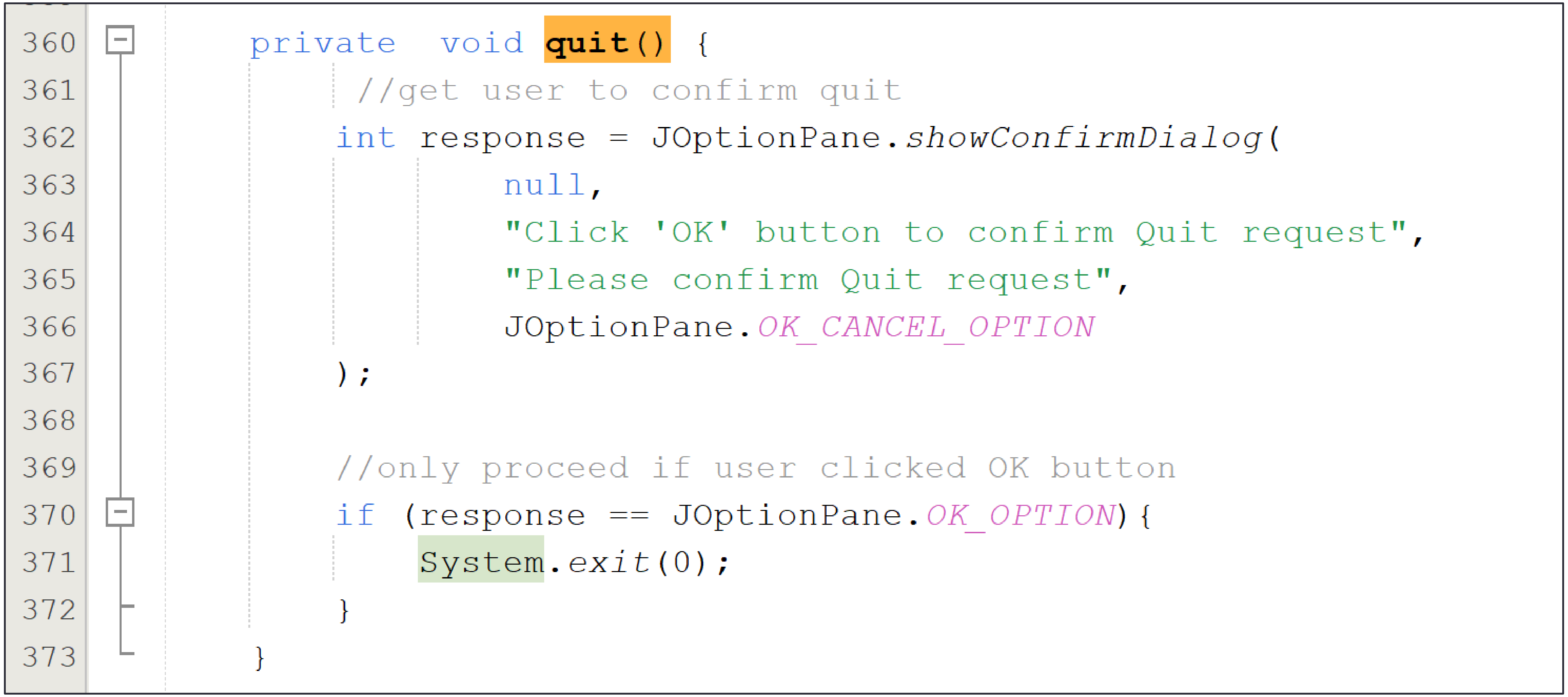
Name	Number	Balance
H Kane	0123456	£ 70.00
M Earps	0234567	£ 75.00
H Hampton	1235678	£ 250.00
J Bellingham	9876543	£ 5000.00
B Mead	2345678	£ 1000.00
R Lewis	3453455	£ 10.99
J Stones	4455667	£ 75.10
L Bronze	3456789	£ 5000.00
B Saka	1112233	£ 325.00
E Hayes	1234567	£ 500.00
K Walker	3344555	£ 175.00

```
### DEPOSIT OPERATION ###


Deposit of £ 70 successfull.
New balance is £ 70.00
```

Confirm Dialog for Quit Operation

- A confirm dialog will be displayed by the quit method
  - To get user confirmation if they wish to quit
- If user confirms quit request
  - Application will save data and exit
- If user decides to cancel quit
  - Do not save and exit, but do return to GUI
- Unlike Input Dialog, the Confirm Dialog returns an integer value
  - Even if user cancels the dialog
- We only need to check if the user has confirmed the quit request
  - By checking if the response was same as positive choice in the buttons, e.g. Yes, OK, Confirm, etc.

Modified Quit Method



```java
    private  void quit() {
        //get user to confirm quit
        int response = JOptionPane.showConfirmDialog(
                null,
                "Click 'OK' button to confirm Quit request",
                "Please confirm Quit request",
                JOptionPane.OK_CANCEL_OPTION
        );

        //only proceed if user clicked OK button
        if (response == JOptionPane.OK_OPTION){
            System.exit(0);
        }
    }
```

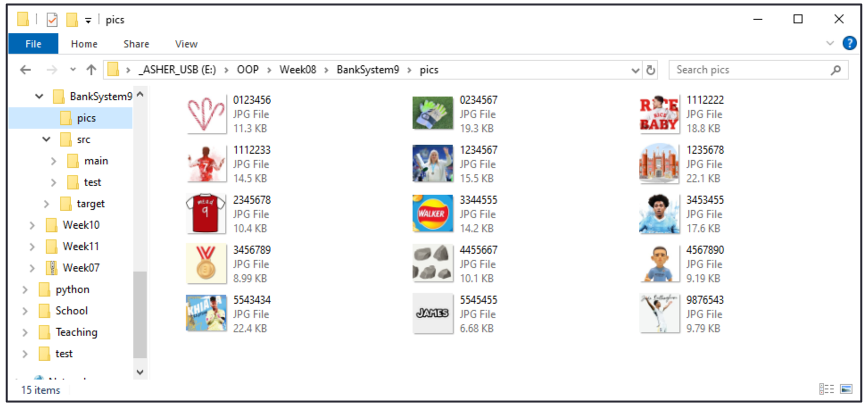
IMAGES

Libraries:

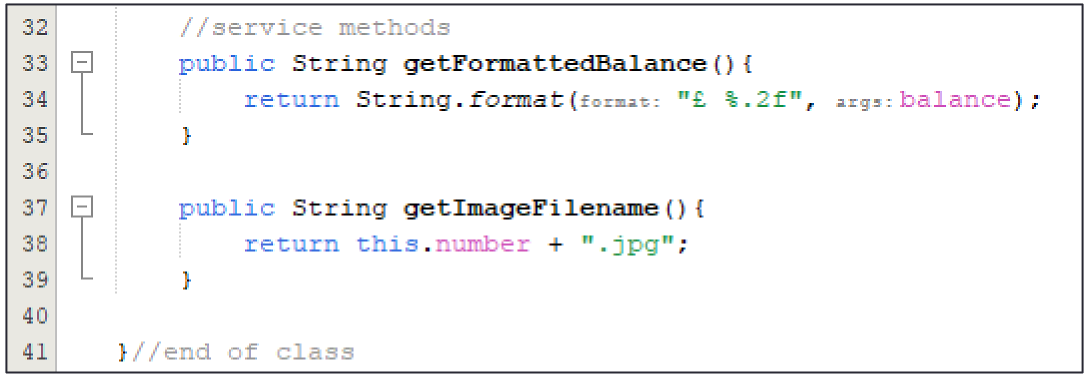
`javax.swing.ImageIcon`, `java.awt.image.BufferedImage`, `java.io.File`, `javax.imageio.ImageIO`, `java.io.IOException`, `java.util.ArrayList`, `java.util.logging.Level` and `java.util.logging.Logger`

Image files

- Image files are best located in a sub folder within the project folder
  - Create sub folder, e.g., pics
  - Copy images to pics sub folder
- If there is a unique name for image file, that is not related to any field of corresponding object
  - Create a new field in data class to save it, along with accessor method
- If image file name is based on an existing field
  - Create a service method to return file name, but not file path



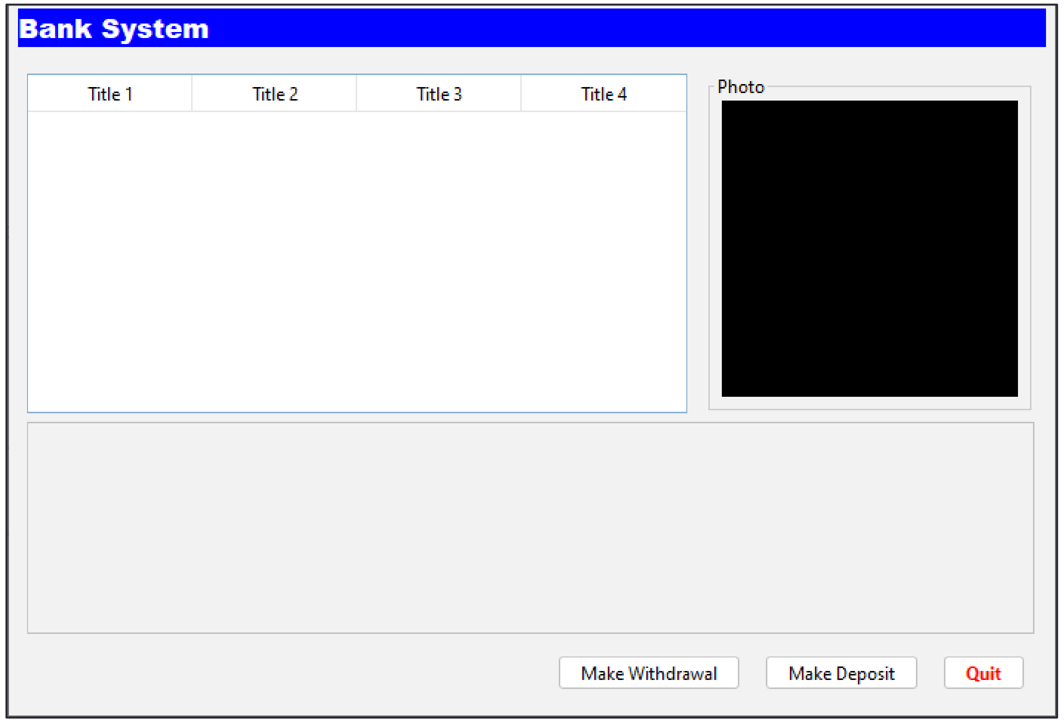
```
0123456   JPG File  11.3 KB     0234567   JPG File  19.3 KB     1112222   JPG File  18.8 KB
1112233   JPG File  14.5 KB     1234567   JPG File  15.5 KB     1235678   JPG File  22.1 KB
2345678   JPG File  10.4 KB     3344555   JPG File  14.2 KB     3453455   JPG File  17.6 KB
3456789   JPG File  8.99 KB     4455667   JPG File  10.1 KB     4567890   JPG File  9.19 KB
5543434   JPG File  22.4 KB     5545455   JPG File  6.68 KB     9876543   JPG File  9.79 KB
```



```java
    //service methods
    public String getFormattedBalance(){
        return String.format( format: "£ %.2f",  args: balance);
    }

    public String getImageFilename(){
        return this.number + ".jpg";
    }

}//end of class
```

GUI Modification

- Create a panel with a label to display image
  - A second optional label could be used to display a caption if required
- Name and set properties for the components
  - For example, the label that is to display images should be big enough to show widest and highest images



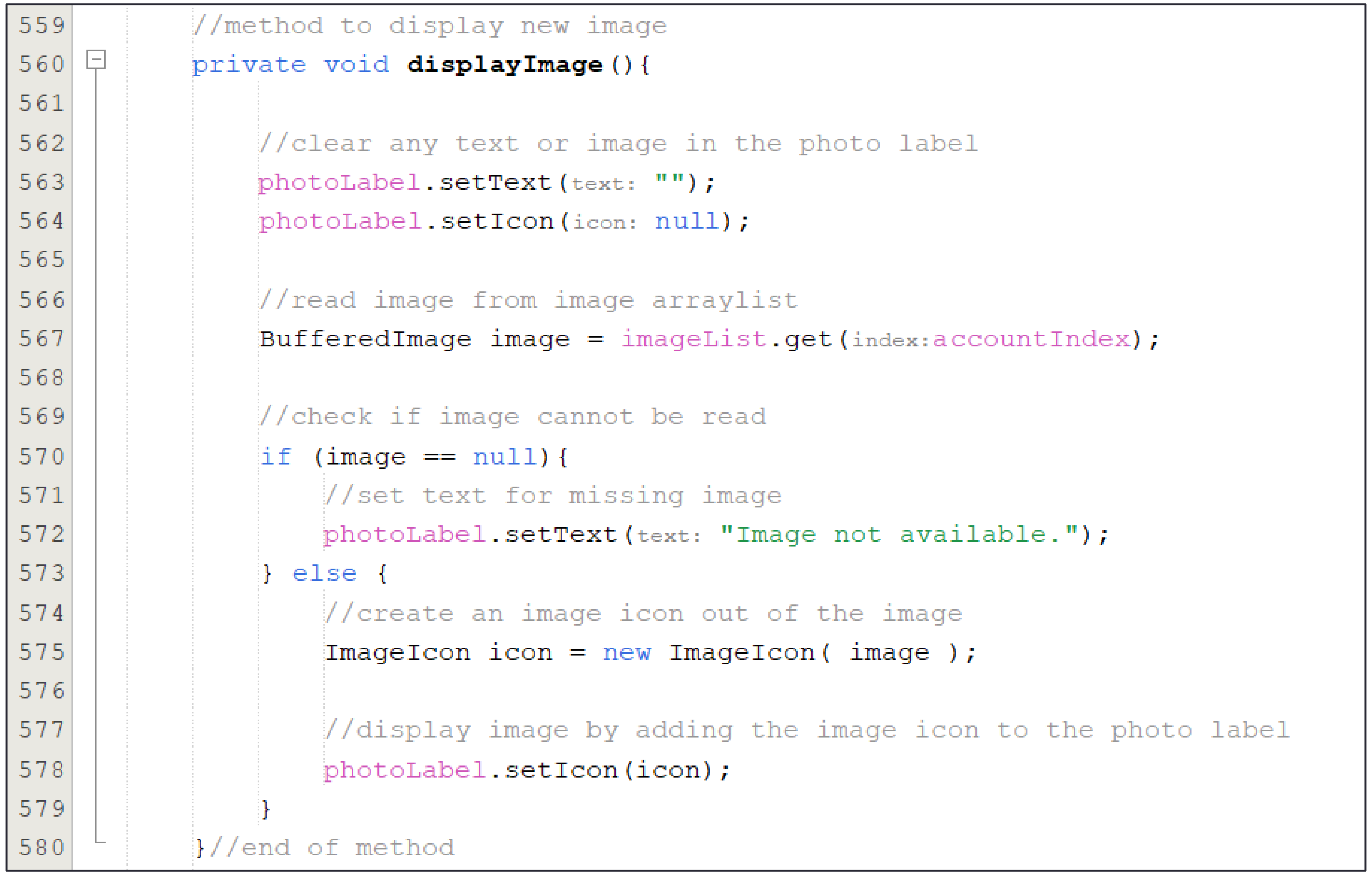
Preloading Images

- We preload images into a ArrayList
  - Create a class level ArrayList `imageList`
  - Create a loadImages() method
  - Invoke method after loadData() within main class constructor

```
private void loadImages(){

  for (BankAccount account: accountList){

     String filepath = "pics/" + account.getImageFilename();

     BufferedImage image = null;

     try {
       image = ImageIO.read(new File(filepath));

     } catch (IOException e) {
       System.err.println("\n\n!!!!! Image Loading Error: !!!!!\n");
     } finally {
       imageList.add(image);
     }
  }
}
```

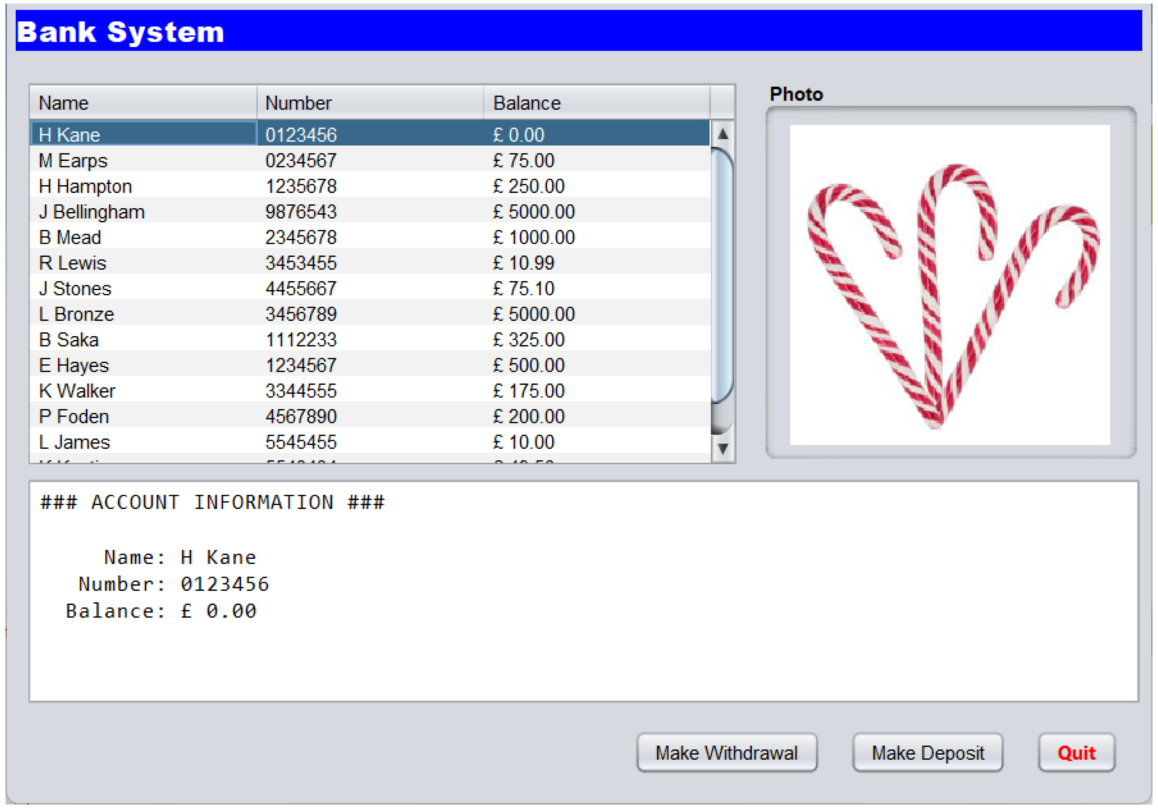
Displaying Images

- Write a `displayImage()` method to display images
- Invoke the method from the view() method



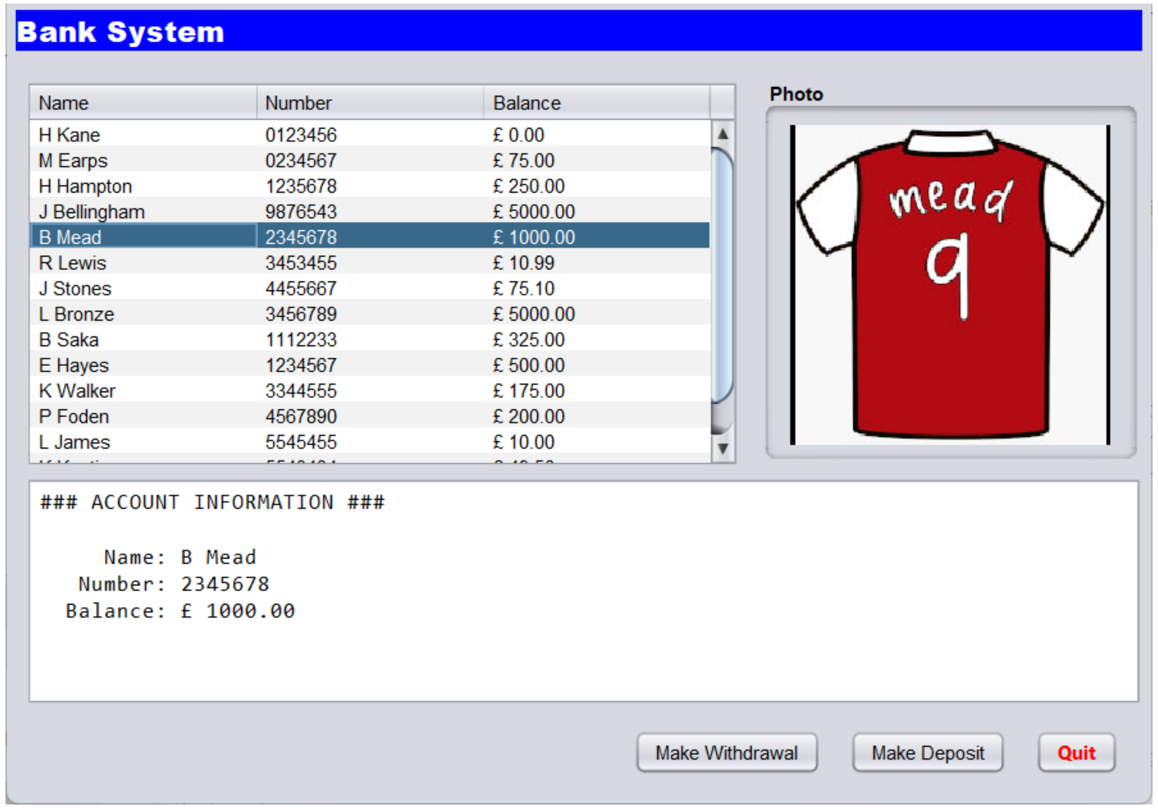
```
//method to display new image
private void displayImage(){

    //clear any text or image in the photo label
    photoLabel.setText(text: "");
    photoLabel.setIcon(icon: null);

    //read image from image arraylist
    BufferedImage image = imageList.get(index:accountIndex);

    //check if image cannot be read
    if (image == null){
        //set text for missing image
        photoLabel.setText(text: "Image not available.");
    } else {
        //create an image icon out of the image
        ImageIcon icon = new ImageIcon( image );

        //display image by adding the image icon to the photo label
        photoLabel.setIcon(icon);
    }
}//end of method
```

Runtime



Name	Number	Balance
H Kane	0123456	£ 0.00
M Earps	0234567	£ 75.00
H Hampton	1235678	£ 250.00
J Bellingham	9876543	£ 5000.00
B Mead	2345678	£ 1000.00
R Lewis	3453455	£ 10.99
J Stones	4455667	£ 75.10
L Bronze	3456789	£ 5000.00
B Saka	1112233	£ 325.00
E Hayes	1234567	£ 500.00
K Walker	3344555	£ 175.00
P Foden	4567890	£ 200.00
L James	5545455	£ 10.00

```
### ACCOUNT INFORMATION ###

     Name:  H Kane
   Number:  0123456
  Balance:  £ 0.00
```



Name	Number	Balance
H Kane	0123456	£ 0.00
M Earps	0234567	£ 75.00
H Hampton	1235678	£ 250.00
J Bellingham	9876543	£ 5000.00
B Mead	2345678	£ 1000.00
R Lewis	3453455	£ 10.99
J Stones	4455667	£ 75.10
L Bronze	3456789	£ 5000.00
B Saka	1112233	£ 325.00
E Hayes	1234567	£ 500.00
K Walker	3344555	£ 175.00
P Foden	4567890	£ 200.00
L James	5545455	£ 10.00

```
### ACCOUNT INFORMATION ###

     Name:  B Mead
   Number:  2345678
  Balance:  £ 1000.00
```