OBJECT ORIENTED PROGRAMMING (OOP)

Java Data Structures and Collections

CASE STUDY: BANK SYSTEM CONSOLE APPLICATION

Purpose and Variables
Main Method
Operations and Methods
Data Class

Purpose and Variables

- Aim of program is to model a simple banking process
- Account data
  - Account name: class level `String` variable
  - Account number: class level `String` variable
  - Account balance: class level `double` variable
- Additional variables
  - Scanner object sc at class level
  - Choice int local variable in main method
  - Amount int local variable in deposit and withdraw methods

```
import java.util.Scanner;

public class SimpleBanking {

    // Class level variables for account information
    static String accountName;
    static String accountNumber;
    static double accountBalance;

    // Class level Scanner object
    static Scanner sc = new Scanner(System.in);

    // Constructor to initialize account details
    public SimpleBanking(String name, String number,
double balance) {
        this.accountName = name;
        this.accountNumber = number;
        this.accountBalance = balance;
    }
```

Main method

```
public static void main(String[] args) {
        // Create a SimpleBanking object with
initial account details
        SimpleBanking account = new
SimpleBanking("John Doe", "123456789",
1000.00);
        int choice;
        do {
            // Display menu
            System.out.println("\nWelcome, " +
accountName + "\n"
            + "\t1. View account information \n"
            + "\t2. Deposit \n"
            + "\t3. Withdraw \n"
            + "\t4. Exit \n"
            + "Enter your choice: ");
            choice = sc.nextInt();
```

```
switch (choice) {
                case 1 -> viewInformation();
                case 2 -> deposit();
                case 3 -> withdraw();
                case 4 -> exit();
                default -> System.out.println("Invalid choice.
Please try again.");
                }
        } while (choice != 4);
    }
```

View Operation

```
public static void viewInformation() {
        // display information
        System.out.printf("%-20s %-10s %s %n", "Name", "Number", "Balance");
        System.out.println("-".repeat(40));
        System.out.printf("%-20s %-10s $%.2f %n", accountName, accountNumber, accountBalance);
    }
```

- Note use of `System.out.printf` to display formatted output
- Within round brackets are two components: Message format and *args* list
- Message format specifies the literal message, using formatters where values should be in the message
- First printf displays column headers and following printfs display data under column headers
- `%-20s` formats the first item in the *args* list as a string value, to be displayed left aligned in a **field width** of 20 characters (`%-10s` formats display of second item in the *args* list)
- `%.2f` formats third item (a double value) to be displayed to two decimal places. (Use `%d` for integers)
- Fields widths used in first printf, should be used in following printfs and values be in same sequence

Deposit

```
public static void deposit() {
        System.out.print("Enter deposit amount: ");
        int amount = sc.nextInt();
        if (amount > 0) {
            accountBalance += amount;
            System.out.println("Successfully deposited $" + amount);
            System.out.println("Updated Balance: $" + accountBalance);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }
```

Withdraw

```
public static void withdraw() {
        System.out.print("Enter withdrawal amount: ");
        int amount = sc.nextInt();
        if (amount > 0 && amount <= accountBalance) {
            accountBalance -= amount;
            System.out.println("Successfully withdrew $" + amount);
            System.out.println("Updated Balance: $" + accountBalance);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient funds.");
        }
    }
```

Exit

```
public static void exit() {
        System.out.println("Thank you for using our service!");
    }
```

Runtime

Menu and View Operation

```
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice: 1
Name                Number     Balance
-------------------------------------
John Doe            123456789  $1000.00
```

Deposit and View Operation

```
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice: 2
Enter deposit amount: 100
Successfully deposited $100
Updated Balance: $1100.0
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice: 1
Name                Number     Balance
-------------------------------------
John Doe            123456789  $1100.00
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice:
```

Runtime Continued

Withdraw and View Operation

```
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice: 3
Enter withdrawal amount: 200
Successfully withdrew $200
Updated Balance: $900.0
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice: 1
Name                    Number      Balance
-----------------------------------------
John Doe                123456789   $900.00
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice:
```

Exit Operation

```
Welcome, John Doe
        1. View account information
        2. Deposit
        3. Withdraw
        4. Exit
Enter your choice: 4
Thank you for using our service!
```

Run your code

- Create a new project and call it SimpleBanking
- Develop all code in one class (use the SimpleBanking.java)
- Run the code and reflect on the OOP principles
- What could be improved?

Reflection

Bank System (1st version)

- Works well for a single Bank account

If more bank accounts need to be created

- The number of variables and operational methods needed to be increased
- We will need additional code to work out which account we are dealing with so to select correct variables and methods to use
- Both the above demonstrate inefficiency in terms of coding and runtime

Solution

- Adopt Object Oriented Programming principles
- Where we separate different features of the application using classes
- But also aim to be efficient

OOP Class Types

Controller

Represents a controlling class, i.e., the main class

Boundary

Represents a user interface class, i.e., the command line menu or GUI

Entity

Represents a data class, i.e., objects which store data

For our console application

Hybrid Approach

- Data Class
- Main Class

Data Class

- Specifies fields to hold data attributes for an entity
- Specifies methods to perform operations on the data
- May have one or more different data classes

Main Class

- Manages application (i.e. controller)
- Provides user interface (i.e. boundary) including input validation
- Includes main method
- Interacts with data classes, by creating object variables of the Data Class

Recap

- To better solve real world problems through a computer program
  - We model real world objects in the program, i.e., Object Oriented Programming (OOP)
  - Applying OOP concepts of **Abstraction** and **Encapsulation**

Abstraction

- Model entities though the **minimum** of properties and behaviours to solve the problem
- If a property can be calculated, provide a behaviour to do so

Encapsulation

- Design and build the Data class
- Use private variables, i.e. fields, to implement attributes
- Provide public method to initialise fields, i.e. constructor methods
- Use public methods to access fields and to implement behaviours

Recap Continued

- **Constructor method(s)**
  - Initialise some or all the fields
  - Has same name as class and no return type
  - Can be overloaded
  - If none is defined Java provides empty constructor at compile time by default
- **Accessor methods**
  - Provide read only access to fields
  - Should have one accessor method for each field
  - Name of accessor methods usually begins with ‘get’
- **Mutator methods**
  - Provide write only access to fields
  - Only provide mutator method for attributes that abstract design identifies can change
  - Name of mutator methods usually begins with ‘set’
- **Service methods**
  - Can calculate a value based on fields
  - Can provide a formatted version of a field value

BANKACCOUNT DATA CLASS

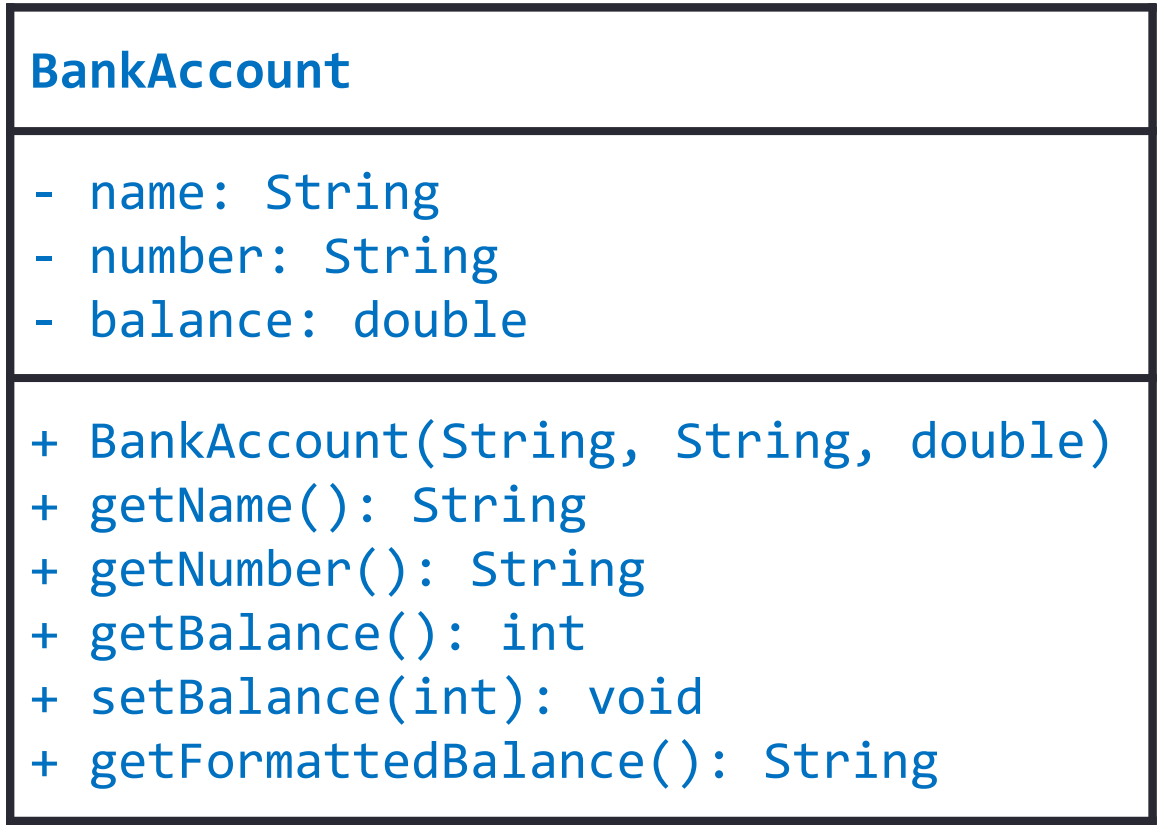
Abstract Design
Encapsulation design
Class Diagram
Visibility
Data Class in NetBeans
Data Class Code

Bank Account Abstract Design

- In terms of abstraction be general, for example
  - We are not concerned with a specific bank, so no need for sort code
  - We are not concerned in modifying account name or number
- Attributes
  - Name | Number | Balance
- Behaviours
  - Full Constructor (i.e. to initialise all attributes)
  - Accessors (getters)
  - Mutator for balance (setters)
  - Formatted output for balance

Encapsulation design

- Design is through a combination of class diagram and pseudo code
- Class Diagram uses a three-row table
- In top row goes the name of the class
- In middle row goes the names and types of fields
  - **-** indicates method will be private
  - Data type goes after a colon
- In the third row goes the method signatures
  - **+** indicates method will be public
  - Arguments specified by type
  - Return type goes after a colon



Visibility: Accessibility of fields, methods and Classes

Level	Symbol	Modifier	Description
Public	+	public	Accessible by any java code (in the world)
Private	-	private	Only accessible by java code in same class.
Protected	#	protected	Only accessible by code in classes in same package and subclasses inside or outside the package
Default (Package)	*No symbol*	*No modifier*	Only accessible by java code in classes and sub classes in same package

Coding the data class - Fields

- First declare the fields, using visibility, type and name from class diagram

```
public class BankAccount {

    // Class level variables for account information
    private String accountName;
    private String accountNumber;
    private double accountBalance;
```

Coding the data class - Constructor

- Secondly code the constructor method
  - Remember it has same name as class and no return type
  - Method parameters hold initial values for fields
- Inside the method body
  - Fields are referenced through use of the `this` keyword
  - And are assigned appropriate parameter

```
public class BankAccount {

    // Class level variables for account information
    private String accountName;
    private String accountNumber;
    private double accountBalance;

    // Constructor to initialize account details
    public BankAccount(String name, String number,
double balance) {
        this.accountName = name;
        this.accountNumber = number;
        this.accountBalance = balance;
    }
```

Coding the data class - Accessors

- Next enter the accessors
  - Which can be one liner statements (i.e., one liners)
- As the methods have no parameters
  - There is no need to use the `this` keyword

```
public class BankAccount {
    // Class level variables for account information
    private String accountName;
    private String accountNumber;
    private double accountBalance;

    // Constructor to initialize account details
    public BankAccount(String name, String number, double
balance) {
        this.accountName = name;
        this.accountNumber = number;
        this.accountBalance = balance;
    }

    //accessors
    public String getAccountName() {return accountName;}
    public String getAccountNumber() {return accountNumber;}
    public double getAccountBalance() {return accountBalance;}
}
```

Coding the data class - Mutator

- Mutator methods can also be one liners
- Acceptable to use initial or abbreviation for parameter
  - In which case there is no need to use the `this` keyword

```
    //accessors
    public String getAccountName() {return accountName;}
    public String getAccountNumber() {return accountNumber;}
    public double getaccountBalance() {return accountBalance;}

    //mutator
    public void setAccountBalance(double balance) {
accountBalance = balance;}
```

Coding the data class – Service methods

- Service methods tend to be in standard method style

```
//mutator
public void setAccountBalance(double balance) { accountBalance = balance;}

//service method
public String getFormattedBalance() { return String.format("$ %.2f", accountBalance);}
```

- If no parameters
  - Then there is no need to use the `this` keyword

BankAccount Complete Data Class Code

```
public class BankAccount {

    // Class level variables for account information
    private String accountName;
    private String accountNumber;
    private double accountBalance;

// Constructor to initialize account details
    public BankAccount(String name, String number, double balance) {
        this.accountName = name;
        this.accountNumber = number;
        this.accountBalance = balance;
    }

    //accessors
    public String getAccountName() {return accountName;}
    public String getAccountNumber() {return accountNumber;}
    public double getaccountBalance() {return accountBalance;}

    //mutator
    public void setAccountBalance(double balance) { accountBalance = balance;}

    //service method
    public String getFormattedBalance() { return String.format("$ %.2f", accountBalance);}
}
```

USING THE DATA CLASS

Creating BankAccount Object instance
Using data class methods via object instance
Runtime
Changes in main class code

Creating a BankAccount Object variable

- Instance variables of the data class are created in the main class
  - No need to import data class, if it is in same package
  - Instantiation makes use of the new keyword followed by the data class constructor method
- Declaration example

```
BankAccount account;
```

- Instantiation (requires instance variable to be already declared)

```
account = new BankAccount(“H Kane”, “0123456”, 0);
```

- Declaration and Instantiation at same time

```
BankAccount account = new BankAccount(“H Kane”, “0123456”, 0);
```

Using data class methods

- Data class methods are used by code in the main class, via the object variable

- Accessor usage example

```
System.out.println("Account Name: " + account.getAccountName() );
```

- Mutator usage example

```
int amount = sc.nextInt();
double balance = account.getAccountBalance + amount;
account.setAccountBalance(balance);
```

- Service method usage example

```
System.out.println("Balance: " + account.getFormattedBalance() );
```

Notable main Class Changes

- Class level BankAccount object created instead of primitive variables
- No need to format balance to two decimal places in output statements.
  - Instead use `%s` as formatter and call objects `getFormattedBalance()` method as value

```
System.out.printf("%-20s %-10s %s  %n",
account.getName(),account.getNumber(),account.getFormattedBalance());
```

- When making a deposit, the new balance is calculated by main class deposit method and then set as objects new balance using the mutator method
  - We could make a deposit service method within data class
  - But this goes against abstraction principle, i.e. minimal code
- Same as above applies to the withdraw operation

Using ArrayList for Data Class objects

Steps to use ArrayList

- Import ArrayList Library
- Create and Code a data class
- Create Class-Level ArrayList, generic to the data class
- Define a `loadData()` method
  - Use provided data to create anonymous data class objects
  - Add anonymous objects to the ArrayList
- Invoke `loadData()` method in `main()` before displaying command line menu
- Modify viewInformation, deposit and withdraw methods to make use of ArrayList
- Above steps will be deployed in SimpleBanking 3rd version, using BankAccount data class

Account Data

Account name	Account number	Account balance
H Kane	0123456	0
E Hayes	1234567	500
B Mead	2345678	1000
L Bronze	3456789	5000
P Foden	4567890	200

ArrayList

- Import Library: `import java.util.ArrayList;`
- Class Level ArrayList: `static ArrayList<BankAccount> accountList = new ArrayList<>();`

```
import java.util.ArrayList;
import java.util.Scanner;

public class SimpleBanking3 {

    // Class level Scanner object
    static Scanner sc = new Scanner(System.in);
    // ArrayList to hold multiple BankAccount objects
    private static ArrayList<BankAccount> accountList = new ArrayList<>();
```

loadData() method – anonymous objects

- Using supplied data
  - Create an anonymous data class object
  - Add anonymous object to arraylist

- To achieve the above, create anonymous object as the argument for the arraylist add method

```
accountList.add( new BankAccount("H Kane", "0123456", 0) );
accountList.add( new BankAccount("E Hayes", "1234567", 500) );
accountList.add( new BankAccount("B Mead", "2345678", 1000) );
accountList.add( new BankAccount("L Bronze", "3456789", 5000) );
accountList.add( new BankAccount("P Foden", "4567890", 200) );
```

loadData() method – code and invocation

In this project, loadData() manually populates the ArrayList

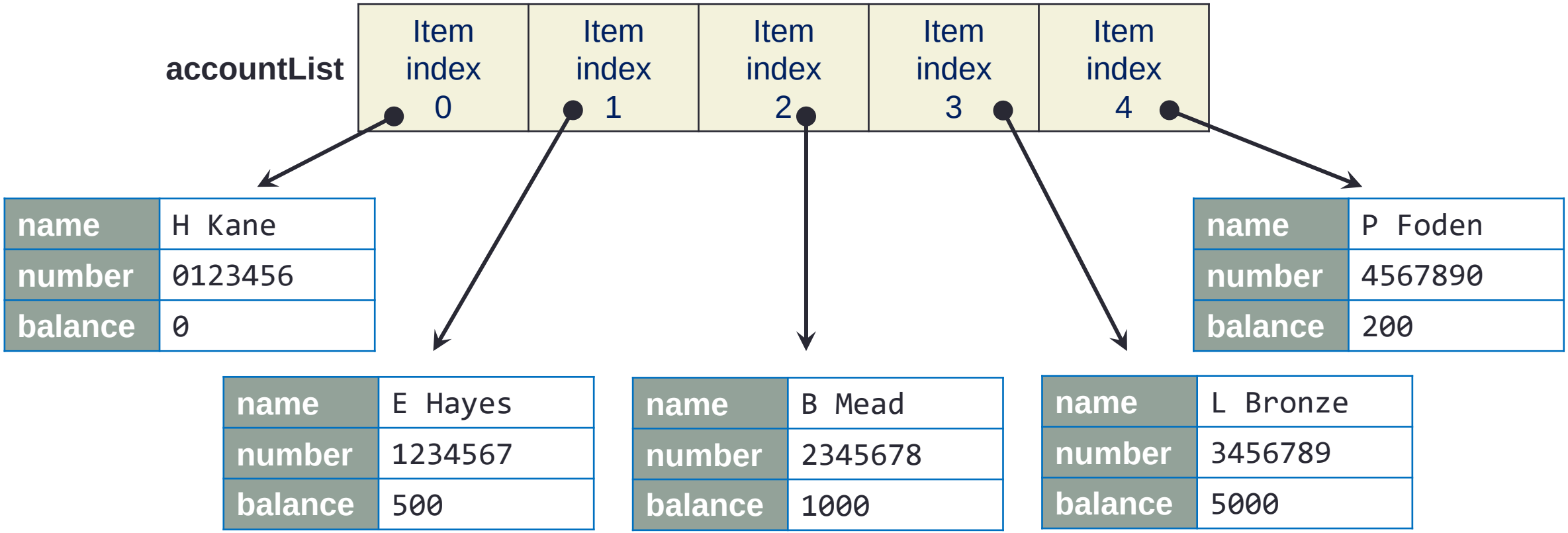
```
public static void loadData() {
        // Create some initial accounts and add them to the list
        accountList.add( new BankAccount("H Kane", "0123456", 0) );
        accountList.add( new BankAccount("E Hayes", "1234567", 500) );
        accountList.add( new BankAccount("B Mead", "2345678", 1000) );
        accountList.add( new BankAccount("L Bronze", "3456789", 5000) );
        accountList.add( new BankAccount("P Foden", "4567890", 200) );
    }
```

The loadData() method is invoked from within the main method.

```
public static void main(String[] args) {
    loadData();
    System.out.println("Bank System Version 3");
```

Visual Representation

- Each item of the ArrayList references an instantiated data class object



ViewInformation Method

- Display data of each object to the screen
- This involves referencing the object, e.g., to reference the first object:

```
accountList.get(0)
```

- Then applying get method via dot notation, e.g., to get `name` value of first object:

```
accountList.get(0).getName();
```

- No need to manually reference each object
  - Instead, a for-loop can be used to process each item in the ArrayList

ViewInformation method code

```
// Method to view information
    public static void viewInformation() {
        // display information
        System.out.printf("%-20s %-10s %s %n", "Name", "Number", "Balance");
        System.out.println("-".repeat(40));
        for (BankAccount account: accountList){
                System.out.printf("%-20s %-10s $%.2f %n", account.getAccountName(),
account.getAccountNumber(), account.getAccountBalance());

        }
        System.out.println("-".repeat(40));

    }
```