**实 验 报 告**

|  |  |
| --- | --- |
| **课程名称：** | **Java语言程序设计** |
| **学生姓名：** |  |
| **学生学号：** |  |
| **学生专业：** | **软件工程** |
| **开课学期：** | **2024-2025学年第一学期** |

**软件学院**

**2023年11月**

**实验二 面向对象的编程技术**

**一、实验目的**

1. 掌握面向对象的继承与多态以及类型转换；
2. 掌握ArrayList类创建动态数组；
3. 掌握Singleton模式的使用

**二、实验内容及结果**

**1.step5**

步骤一 ：创建 banking 目录并复制项目文件

1. 在 /step5 目录下创建 banking 目录。
2. 将前面的 Banking 项目文件复制到此 banking 目录中。

步骤二 ：修改 Account 类

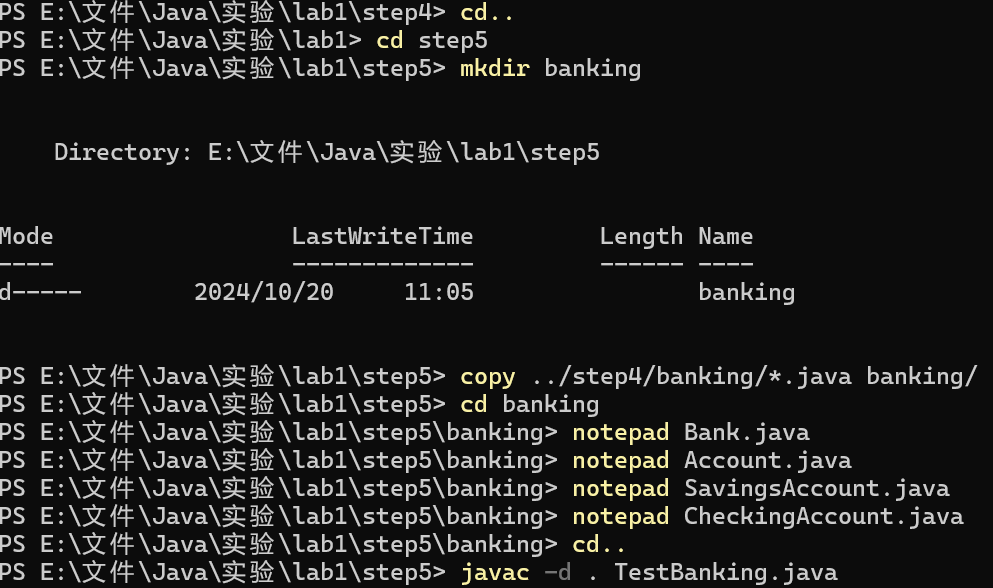
1. 将 Account 类中的 balance 属性修改为 protected 访问级别，这样子类可以访问它。
2. package banking;
4. public class Account {
5. protected double balance;
7. // 构造函数初始化余额
8. public Account(double init\_balance) {
9. this.balance = init\_balance;
10. }
12. // 获取当前余额
13. public double getBalance() {
14. return balance;
15. }
17. // 存款方法，返回 true
18. public boolean deposit(double amount) {
19. if (amount > 0) {
20. balance += amount;
21. return true;
22. }
23. return false;
24. }
26. // 取款方法，条件化取款，返回 true 或 false
27. public boolean withdraw(double amount) {
28. if (amount > 0 && amount <= balance) {
29. balance -= amount;
30. return true;
31. }
32. return false;
33. }
34. }

步骤三 ：实现 SavingsAccount 子类

1. 创建 SavingsAccount 类，继承 Account 类。
2. 在 SavingsAccount 类中添加一个 interestRate 属性（类型为 double）。
3. 实现一个公共构造函数，该构造函数接受两个参数：balance 和 interest\_rate，并通过调用 super(balance) 传递 balance 给父类构造函数。
4. package banking;
6. public class SavingsAccount extends Account {
7. private double interestRate;
9. public SavingsAccount(double balance, double interestRate) {
10. super(balance);
11. this.interestRate = interestRate;
12. }
13. }

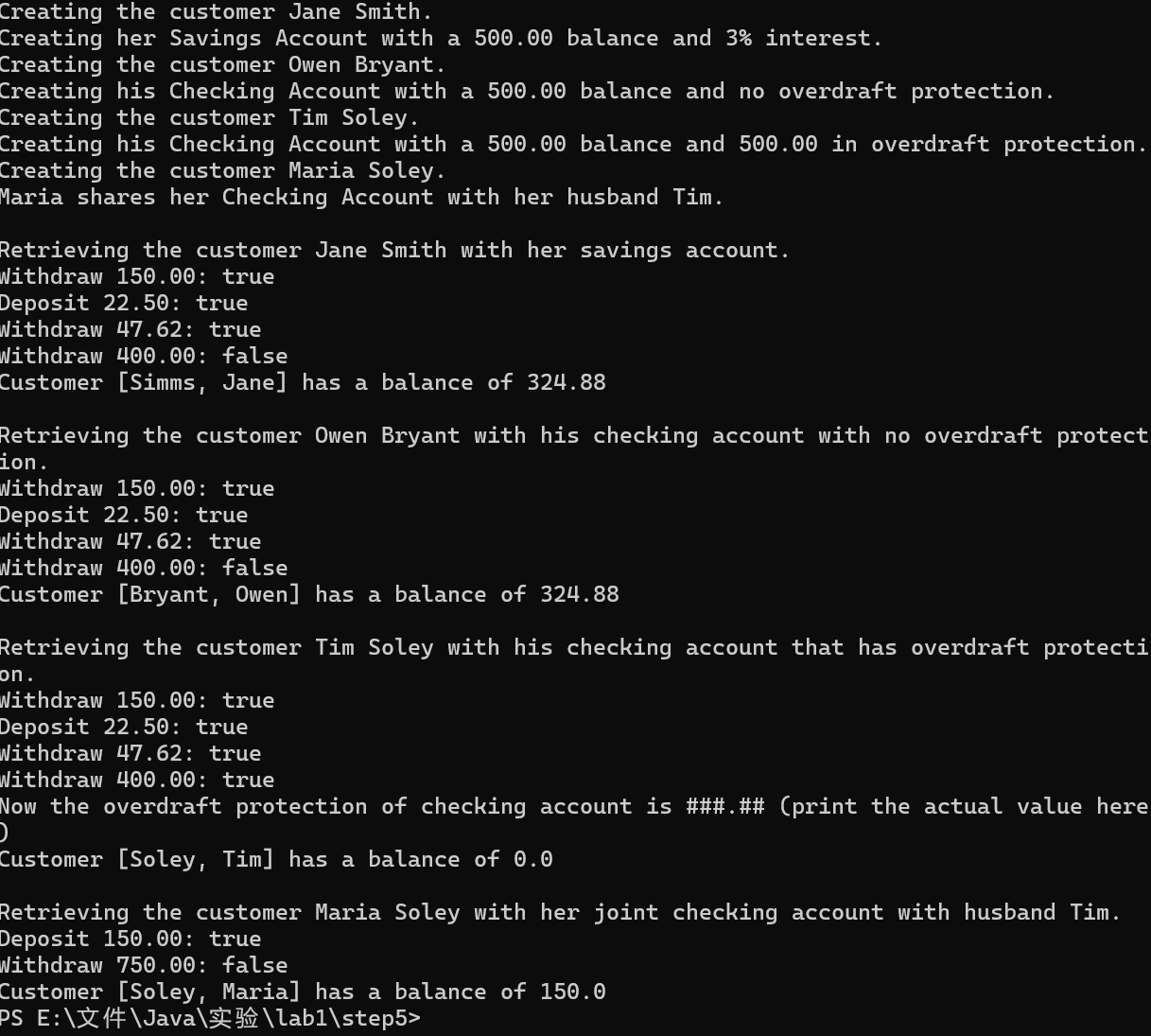
步骤四：实现 CheckingAccount 子类

1. 创建 CheckingAccount 类，继承 Account 类。
2. 添加 overdraftProtection 属性（类型为 double）。
3. 实现一个只接受 balance 参数的构造函数，并通过 super(balance) 调用父类构造函数。
4. 实现另一个构造函数，接受 balance 和 overdraftProtection 参数，并调用 super(balance) 来设置余额，初始化 overdraftProtection。
5. 重写 withdraw 方法，检查余额是否足够，若余额不足则尝试使用透支保护。如果保护不足以覆盖差额，则交易失败，余额不变。
6. 添加 getOverdraftProtection 方法，返回当前的透支保护金额。
7. package banking;
9. public class CheckingAccount extends Account {
10. private double overdraftProtection;
12. public CheckingAccount(double balance) {
13. super(balance);
14. this.overdraftProtection = 0;
15. }
17. public CheckingAccount(double balance, double overdraftProtection) {
18. super(balance);
19. this.overdraftProtection = overdraftProtection;
20. }
22. @Override
23. public boolean withdraw(double amount) {
24. if (balance >= amount) {
25. balance -= amount;
26. return true;
27. } else if (overdraftProtection >= (amount - balance)) {
28. overdraftProtection -= (amount - balance);
29. balance = 0;
30. return true;
31. }
32. return false;
33. }
35. public double getOverdraftProtection() {
36. return overdraftProtection;
37. }
38. }



步骤五 ：测试

1. 在主目录 step5 下编译并运行 TestBanking.java，确保输出结果符合预期：



结果分析

* SavingsAccount 类与普通的账户没有太大差别，只是添加了利率。
* CheckingAccount 支持透支保护，在余额不足时使用透支金额来完成交易。

1. **Step 6\_pre1**

步骤一 ：创建 banking 目录并复制项目文件

1. 在 /step6-pre1 目录下创建 banking 目录。
2. 将前面的 Banking 项目文件复制到此 banking 目录中。

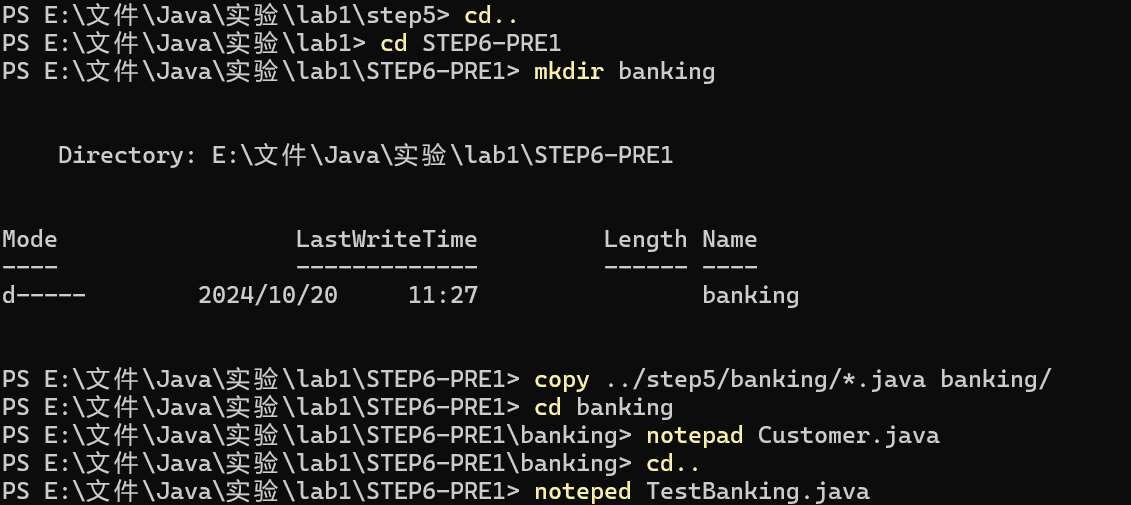
步骤二 ：修改 Customer 类

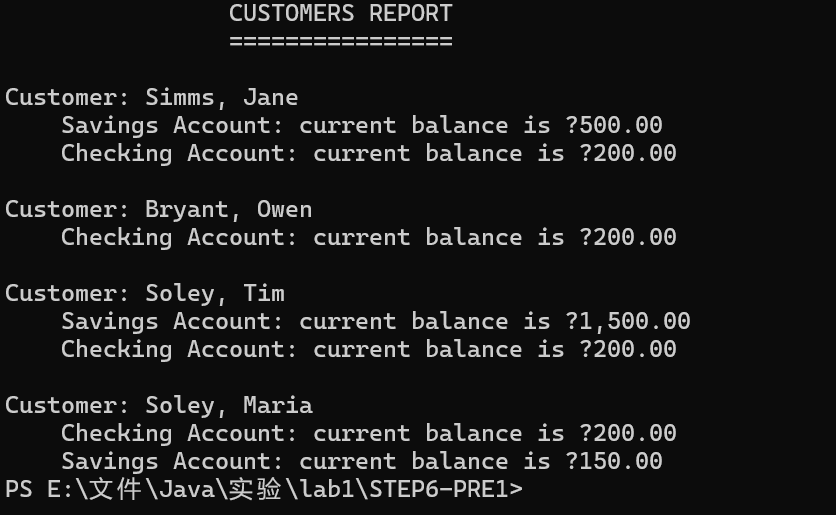
在 Customer 类中添加如下修改，使其能够处理多个账户的关联：

1. 使用 ArrayList 来存储 Account 对象的数组。
2. 添加以下公共方法：
   * addAccount(Account account)：用于将账户添加到客户的账户列表中。
   * getAccount(int index)：用于根据索引获取指定的账户。
   * getNumOfAccounts()：返回客户的账户数量。
3. package banking;
5. import java.util.ArrayList;
7. public class Customer {
8. private String firstName;
9. private String lastName;
10. private ArrayList<Account> accounts;
12. public Customer(String f, String l) {
13. firstName = f;
14. lastName = l;
15. accounts = new ArrayList<>();
16. }
18. public String getFirstName() {
19. return firstName;
20. }
22. public String getLastName() {
23. return lastName;
24. }
26. public void addAccount(Account account) {
27. accounts.add(account);
28. }
30. public Account getAccount(int index) {
31. return accounts.get(index);
32. }
34. public int getNumOfAccounts() {
35. return accounts.size();
36. }
37. }

步骤 3 ：完成 TestBanking 程序

1. 在 TestBanking.java 中找到标记为 /\*\*\* ... \*\*\*/ 的注释块，并按照如下步骤补全代码：
   * Step 1: 使用 instanceof 运算符来判断账户的类型，并将 account\_type 设置为 "Savings Account" 或 "Checking Account"。
   * Step 2: 打印账户的类型和余额，使用 currency\_format 来格式化余额。
2. for ( int acct\_idx = 0; acct\_idx < customer.getNumOfAccounts(); acct\_idx++ ) {
3. Account account = customer.getAccount(acct\_idx);
4. String  account\_type = "";
6. // Step 1: 确定账户类型
7. if (account instanceof SavingsAccount) {
8. account\_type = "Savings Account";
9. } else if (account instanceof CheckingAccount) {
10. account\_type = "Checking Account";
11. }
13. // Step 2: 打印账户类型和余额
14. System.out.println("    " + account\_type + ": current balance is "
15. + currency\_format.format(account.getBalance()));



步骤 4：编译并运行

结果分析

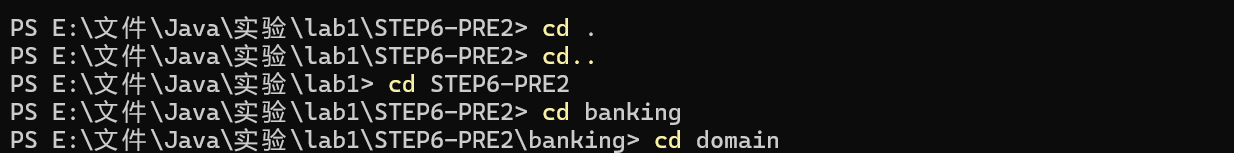
通过该实验，我们实现了让每个客户能够拥有多个账户的功能，并通过 ArrayList 轻松管理这些账户。程序能够判断账户类型，并格式化输出账户的余额。

1. **Step6\_pre2**

步骤 1 ：设置项目目录结构

1. 将 /step6-pre1 目录中的所有 Java 文件复制到 /step6-pre2/banking/domain 目录下。

2. 为报告类创建一个新的目录 /step6-pre2/banking/reports，稍后在该目录下创建 CustomerReport 类。



步骤 2 ：修改 Bank 类以实现单例模式

修改 Bank 类，使其遵循单例模式，确保系统中只有一个 Bank 实例。

1. 在 Bank 类中添加一个静态的私有 Bank 实例作为类的属性。

2. 构造函数 Bank 设为私有，以防止外部创建实例。

3. 添加一个公共的静态方法 getBank，返回唯一的 Bank 实例。如果实例不存在，则创建它。

1. package banking.domain;
3. import java.util.ArrayList;
5. public class Bank {
6. private static Bank instance = null; // 单例实例
7. private ArrayList<Customer> customers;
8. private int numberOfCustomers;
10. // 构造函数设为私有
11. private Bank() {
12. customers = new ArrayList<>();
13. numberOfCustomers = 0;
14. }
16. // 获取 Bank 单例实例的方法
17. public static Bank getBank() {
18. if (instance == null) {
19. instance = new Bank();
20. }
21. return instance;
22. }
24. public void addCustomer(String f, String l) {
25. customers.add(new Customer(f, l));
26. numberOfCustomers++;
27. }
29. public int getNumOfCustomers() {
30. return numberOfCustomers;
31. }
33. public Customer getCustomer(int index) {
34. return customers.get(index);
35. }
36. }

步骤 3：创建并修改 CustomerReport 类

在 banking/reports 目录下创建 CustomerReport 类，并修改其代码以使用 Bank 类的单例对象。

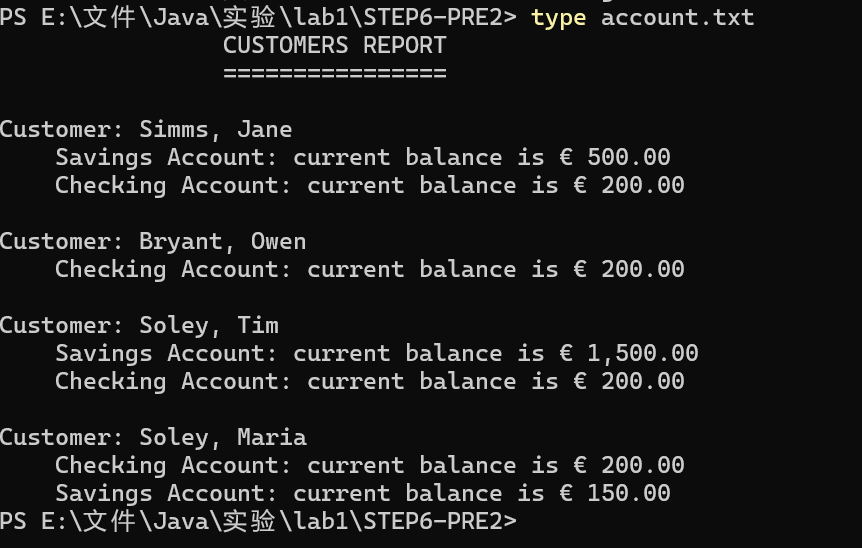
1. package banking.reports;
3. import banking.domain.\*;
5. public class CustomerReport {
7. public void generateReport() {
8. Bank bank = Bank.getBank();  // 使用单例 Bank 对象
10. System.out.println("\t\t\tCUSTOMERS REPORT");
11. System.out.println("\t\t\t================");
13. for (int cust\_idx = 0; cust\_idx < bank.getNumOfCustomers(); cust\_idx++) {
14. Customer customer = bank.getCustomer(cust\_idx);
16. System.out.println();
17. System.out.println("Customer: "
18. + customer.getLastName() + ", "
19. + customer.getFirstName());
21. for (int acct\_idx = 0; acct\_idx < customer.getNumOfAccounts(); acct\_idx++) {
22. Account account = customer.getAccount(acct\_idx);
23. String account\_type = "";
25. if (account instanceof SavingsAccount) {
26. account\_type = "Savings Account";
27. } else if (account instanceof CheckingAccount) {
28. account\_type = "Checking Account";
29. }
31. System.out.println("    " + account\_type + ": current balance is € "
32. + String.format("%.2f", account.getBalance()));
33. }
34. }
35. }
36. }

步骤 4：修改 TestBanking 类

在 TestBanking.java 中，确保 Bank 类通过 Bank.getBank() 获取实例，并调用 CustomerReport 生成报告。

1. import banking.domain.\*;
2. import banking.reports.CustomerReport;
4. public class TestBanking {
6. public static void main(String[] args) {
7. Bank bank = Bank.getBank();  // 使用单例 Bank 对象
8. Customer customer;
9. CustomerReport report = new CustomerReport();
11. // 创建多个客户及其账户
12. bank.addCustomer("Jane", "Simms");
13. customer = bank.getCustomer(0);
14. customer.addAccount(new SavingsAccount(500.00, 0.05));
15. customer.addAccount(new CheckingAccount(200.00, 400.00));
17. bank.addCustomer("Owen", "Bryant");
18. customer = bank.getCustomer(1);
19. customer.addAccount(new CheckingAccount(200.00));
21. bank.addCustomer("Tim", "Soley");
22. customer = bank.getCustomer(2);
23. customer.addAccount(new SavingsAccount(1500.00, 0.05));
24. customer.addAccount(new CheckingAccount(200.00));
26. bank.addCustomer("Maria", "Soley");
27. customer = bank.getCustomer(3);
28. // Maria 和 Tim 共享一个账户
29. customer.addAccount(bank.getCustomer(2).getAccount(1));
30. customer.addAccount(new SavingsAccount(150.00, 0.05));
32. // 生成报告
33. report.generateReport();
34. }
35. }

步骤 5：编译并运行程序

在主目录 /step6-pre2 中编译并运行 TestBanking 程序，输出应如下所示：

结论：

通过实现单例设计模式，我们确保了系统中只有一个 Bank 实例。CustomerReport 类能够生成客户及其账户余额的详细报告，这展示了如何通过单例模式在多个类中共享同一个 Bank 实例。

**三．实验问题回答**

（1）对 ArrayList 中的 Account 集合排序：

要对 ArrayList 中的 Account 对象按照余额进行排序，可以使用 Collections.sort() 方法结合自定义的比较器。实现的关键思想是实现 Comparator 接口，重写 compare 方法，以便根据余额进行比较。下面是实现的步骤和关键代码：

实现思想

1. 实现 Comparator 接口：定义一个比较器类或使用 Lambda 表达式。

2. 重写 compare 方法：在该方法中比较两个 Account 对象的余额。

3. 使用 Collections.sort() 方法：调用该方法并传入要排序的 ArrayList 和比较器。

（2）单例模式的设计原理

单例模式确保一个类只有一个实例，并提供一个全局访问点。其设计原理包括以下几点：

1. 私有构造函数：将构造函数设置为私有，防止外部代码通过 new 关键字创建多个实例。

2. 静态属性：创建一个静态变量来存储该类的唯一实例。因为这是类级别的属性，与对象无关，因此必须是 static。

3. 静态方法：提供一个静态方法用于获取唯一实例。该方法通常会检查实例是否已创建，如果没有，则创建并返回该实例。

属性和方法为 static 的原因：

- 静态属性：由于只有一个实例，因此使用 static 可以确保所有访问者共享同一个实例。

- 静态方法：静态方法可以直接通过类名调用，而不需要创建对象。这使得获取实例的过程更加方便。

构造函数为 private 的原因

- 防止外部创建多个实例：将构造函数设为私有，确保其他类无法直接实例化该类，从而实现单例模式的目标。