Implementing a Commercial/Business Banking System Cover page

By: Deanna Nguyen

Boston University

MET CS 665

Credits to: https://www.inc.com/aj-agrawal/how-to-open-a-business-bank-account-for-your-startup.html

Abstract

This final project will explain the design steps to creating a Commercial Banking Application. The Goal of the application is to process transactions for Commercial Banking Customers. Credits to Professor Kia Teymourian, Facilitator Michael Huang, and River City Bank.

Table of Contents

TABLE OF CONTENTS

Code Contributions/ Changes

Note: The original code I tested based on Professor Vidal's example contained many bugs and was not able to run correctly. This code was revised unique to my project and is intended to be runnable.

1. Class Names: Bank
Lines of Code: 108-145
Explanation: The class is modified to check for credit score and account type. Previously, only account type was checked. Afterwards, the account will be added to the database as soon as creation occurs. In contrast, his code adds the account at the "while loop". The code was rearranged to ensure approved bank accounts would be added to the database (and not any ineligible accounts).

Previous code: Credits to Professor Jose M. Vidal (see references, chapter 12 slides)

Case Study: Reading from a File (2/3)

```
public class Bank
{
   private void readAccountFile(Scanner in)
   { while (in.hasNextLine())
     { this.addAccount(Account.read(in));
       in.nextLine();
     }
   }
}

public class Account
{ ...

   /** A factory method to construct an appropriate account object for the data in the file. */
   public static Account read(Scanner in)
   { String type = in.next();
     if (type.equals("mb"))        { return new MinBalAccount(in); }
     else if (type.equals("pu"))   { return new PerUseAccount(in); }
     else  { throw new Error("Unrecognized Account type: " + type);       }
   }
}
```

Input File:
```
mb 1000 1500.00 false
mb 1001 300.00 true
pu 1005 40.00 0
pu 1007 100.00 3
```

Modified code: Credits to Professor Jose M. Vidal (see references, chapter 12 slides)

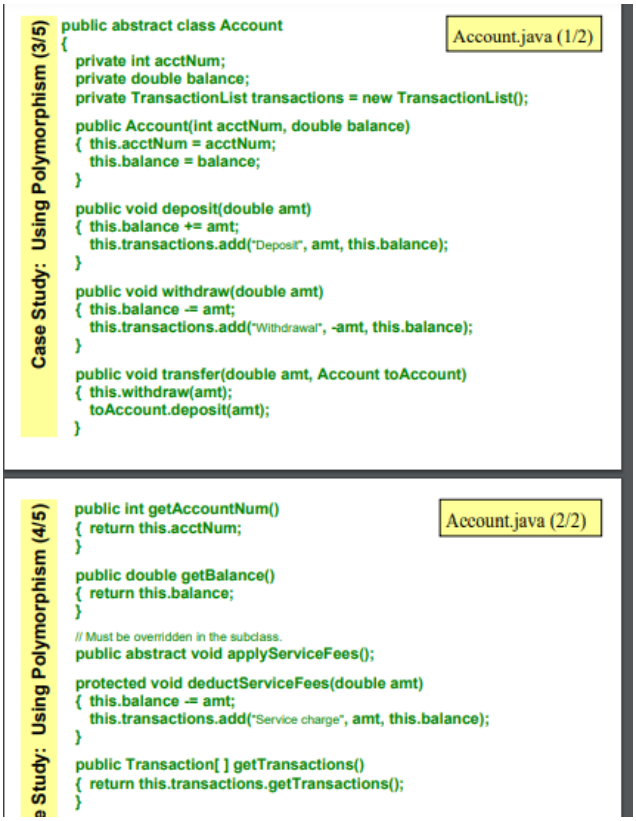
```java
private void createAccount(Scanner in) {
  in.nextLine(); // Skip the first column
  while (in.hasNextLine()) { // read each file line by line
    // Scan for the account information in each line.
    String accountType = in.next(); // checking or savings type
    int creditScore = in.nextInt(); // credit score
    int accountNum = in.nextInt(); // account num
    double balance = in.nextDouble(); // initial balance in file
    BankAccount account = null; // container to store in database

    /*
     * Create a bank account based on the type provided and if it meets the credit
     * score criteria. The minimum credit score must be 680
     * (https://www.fundera.com/business-loans
     * /guides/credit-score-for-business-loan) Per conversation with my bank on
     * 7/16/2021, the minimum is 680.
     */
    if (accountType.contains("BusinessChecking") // If business checking type
        && (creditScore >= 680)) { // if credit score is 680 or above
      // create the business checking account and add to accounts database
      addAccount(new BusinessChecking(accountNum, balance));
    } else if (accountType.contains("BusinessSavings") && (creditScore >= 680)) {
      // If business savings type and credit score is 680 or higher,
      // create the business savings account and add to database
      addAccount(new BusinessSavings(accountNum, balance));
    } else {
      // Print out a statement stating which account number
      // is not eligible and cannot be created.
      System.out.println("Account number: " + accountNum + " is ineligible to be created.");
    }
    try {
      // Check if there is information at the end of the file.
      in.nextLine();
    } catch (NoSuchElementException exception) {
      break; // if no lines, finish the code.
    }
  }
}
```

2. Class Names: BankAccount
Lines of Code: 15-19, 54-99, 151-153
Explanation: The abstract class contains a new attribute called "checkBalBelow". It is used to verify if the balance is below 0. Previously, the attribute was in the subclass.

Second, the withdraw and transfer methods were modified to fit my current bank's standards. Typically, a transaction fee is charged if below balance or if a deposit is large.

Third, the transactionfees and "subtract" methods were modified. Previously, they were "deduct and applied service fees".

Previous code: Credits to Professor Jose M. Vidal (see references, chapter 12 slides)



Case Study: Using Polymorphism (3/5)

Account.java (1/2)

```
public abstract class Account
{
  private int acctNum;
  private double balance;
  private TransactionList transactions = new TransactionList();

  public Account(int acctNum, double balance)
  { this.acctNum = acctNum;
    this.balance = balance;
  }

  public void deposit(double amt)
  { this.balance += amt;
    this.transactions.add("Deposit", amt, this.balance);
  }

  public void withdraw(double amt)
  { this.balance -= amt;
    this.transactions.add("Withdrawal", -amt, this.balance);
  }

  public void transfer(double amt, Account toAccount)
  { this.withdraw(amt);
    toAccount.deposit(amt);
  }
```

Case Study: Using Polymorphism (4/5)

Account.java (2/2)

```
public int getAccountNum()
{ return this.acctNum;
}

public double getBalance()
{ return this.balance;
}

// Must be overridden in the subclass.
public abstract void applyServiceFees();

protected void deductServiceFees(double amt)
{ this.balance -= amt;
  this.transactions.add("Service charge", amt, this.balance);
}

public Transaction[ ] getTransactions()
{ return this.transactions.getTransactions();
}
```

Modified code: Credits to Professor Jose M. Vidal (see references, chapter 12 slides)
Lines: 14-19
Refactoring Method : Pull up

```
14 public abstract class BankAccount {
15   private int accountNumber; // account number
16   private int numWithdrawals = 0; // number of withdrawals
17   private double balance; // balance funds
18   // Used to check if balance is below 0.
19   protected boolean checkBalBelow = false;
```

Lines: 54-69

```
54   public void withdraw(double funds) {
55     // If the funds is less than balance and the user did not input 0,
56     // or a negative number, withdraw funds.
57     if ((funds < balance) && (funds >= 1)) {
58       balance -= funds;
59       if (balance < 0) { // Verify the balance is not 0.
60         checkBalBelow = true; // if 0, change the bool to true
61       }
62       numWithdrawals++; // increase number of withdrawals
63     } else if ((new BigDecimal(funds).setScale(2,RoundingMode.UP))
64         == (new BigDecimal(balance).setScale(2,RoundingMode.UP))) {
65       // If user withdraws exact balance amount, notify the user
66       // they will have 0 balance. Big decimal rounds up to 2.
67       // https://java2blog.com/format-double-to-2-decimal-places-java/
68       System.out.println("The user will have 0 balance.");
69       balance = 0; // set balance to 0
```

3. Class Names: BusinessChecking
Lines of Code: 33-42
Explanation: The transaction fees method is based on the "checkBalBelow" attribute moved to the parent class. In addition, the statement is reduced in coding and charges a different transaction fee of 30.00.

Previous code: Credits to Professor Jose M. Vidal (see references, chapter 12 slides)

Case Study: Using Polymorphism (5/5)

MinBalAccount.java (1/1)

```
public class MinBalAccount extends Accour
{
  public static final double MIN_BAL = 1000.00;
  public static final double SERVICE_FEE = 7.50;
  private boolean balBelow = false;

  public MinBalAccount(int theAcctNum, double balance)
  { super(theAcctNum, balance);
    this.balBelow = this.getBalance() < MinBalAccount.MIN_BAL;
  }

  public void withdraw(double amt)
  { super.withdraw(amt);
    if (this.getBalance() < MinBalAccount.MIN_BAL)
    { this.balBelow = true;
    }
  }

  public void applyServiceFees()
  { if (this.balBelow)
    { super.deductServiceFees(MinBalAccount.SERVICE_FEE);
      this.balBelow = this.getBalance() < MinBalAccount.MIN_BAL;
    }
  }
}
```

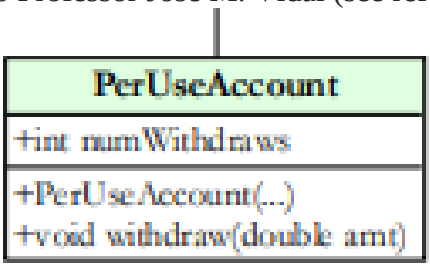
Modified code:

```
31  // this method will deduct transaction fees
32  @Override
33  public void transactionFees() {
34    // If the boolean balance below is true subtract transaction fees
35    // Sources:https://rivercitybank.com/resources/fee-schedule-bus/
36    // https://jmvidal.cse.sc.edu/csce145/fall06/Ch12/Ch12%20Slides.pdf
37    if ((checkBalBelow) || (getnumofWithdrawals() > 6)) {
38      busCheckTransactionFee = 30.00; // change transaction fees to 30 dollars.
39      // subtract transaction fees.
40      super.subtract(busCheckTransactionFee);
41      // verify balance is equal to checkbalance
42      checkBalBelow = checkBalance() < busCheckMinimumBalance;
43    }
44  }
```

3. Class Names: BusinessSavings
Lines of Code: 34-43

Explanation: The transaction fees method is based on the "checkBalBelow" attribute moved to the parent class. In addition, the statement is reduced in coding and charges a different transaction fee of 30.00, checking for the balance.

Previous code: Credits to Professor Jose M. Vidal (see references, chapter 12 slides)



Modified code:

```
32    // this method will deduct transaction fees
33    @Override
34    public void transactionFees() {
35      // If the boolean balance below is true subtract transaction fees
36      // Sources:https://rivercitybank.com/resources/fee-schedule-bus/
37      // https://jmvidal.cse.sc.edu/csce145/fall06/Ch12/Ch12%20Slides.pdf
38      if ((checkBalBelow) || (getnumofWithdrawals() > 6)) {
39        busSavingsTransactionFee = 30.00; // change transaction fees to 30 dollars.
40        // subtract transaction fees.
41        super.subtract(busSavingsTransactionFee);
42        // verify balance is equal to checkbalance
43        checkBalBelow = checkBalance() < busSavingsMinimumBalance;
44      }
45    }
46  }
```

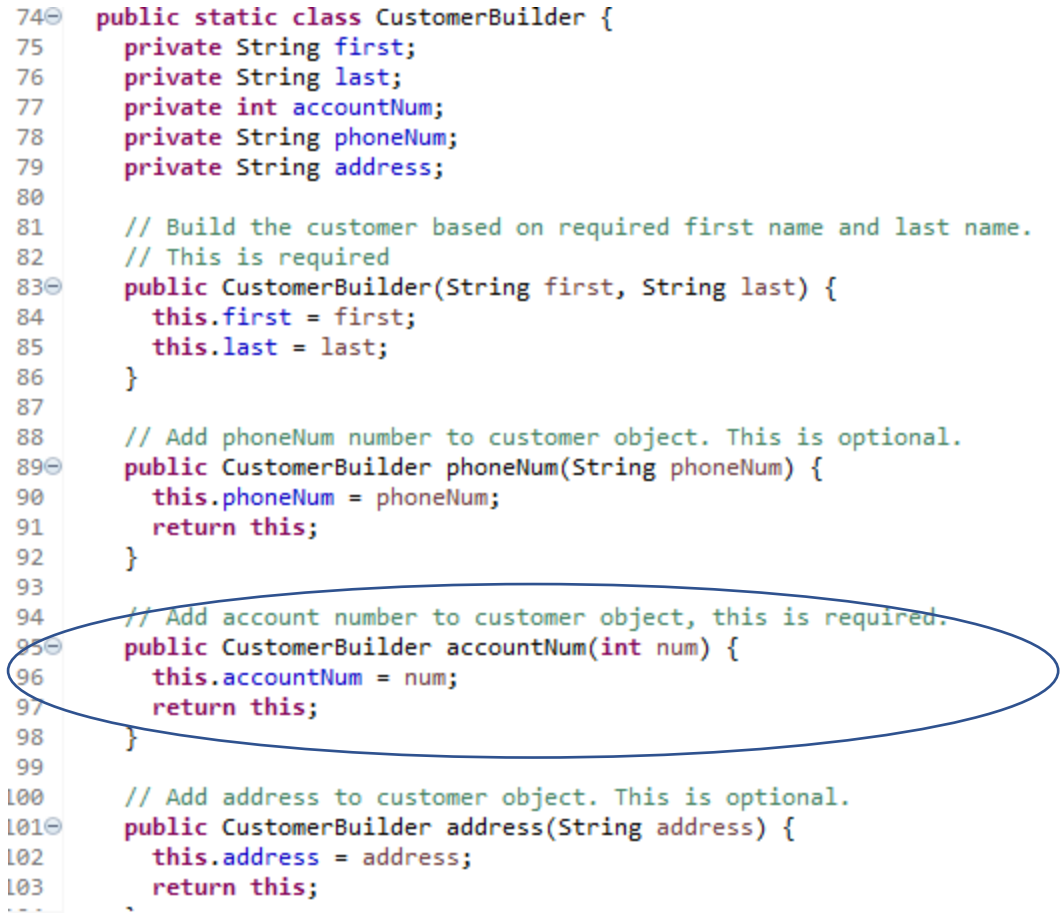
4. Class Names: Customer
Lines of Code: 9-14, 74-79
Explanation: The attributes for the customer builder were modified to be private only. This is because first and last names, phone numbers, and address can be changed anytime. However, an account number (unless deactivated) will remain the same. Therefore, the account number was made immutable.

Previous code: Credits to "how to do in java" (see references).

```java
public class User
{
    //All final attributes
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }
```

Modified code:

```java
/**
 * The responsibility of this class is to store customer objects. Source:
 * https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/
 * https://refactoring.guru/design-patterns/builder/java/example
 */
public class Customer {
  // These attributes are for the customer.
  private String first; // First name
  private String last; // Last name
  private final int accountNum; // Account Number
  private String phoneNum; // phoneNum
  private String address; // Address
```



```
74  public static class CustomerBuilder {
75    private String first;
76    private String last;
77    private int accountNum;
78    private String phoneNum;
79    private String address;
80
81    // Build the customer based on required first name and last name.
82    // This is required
83    public CustomerBuilder(String first, String last) {
84      this.first = first;
85      this.last = last;
86    }
87
88    // Add phoneNum number to customer object. This is optional.
89    public CustomerBuilder phoneNum(String phoneNum) {
90      this.phoneNum = phoneNum;
91      return this;
92    }
93
94    // Add account number to customer object, this is required.
95    public CustomerBuilder accountNum(int num) {
96      this.accountNum = num;
97      return this;
98    }
99
100   // Add address to customer object. This is optional.
101   public CustomerBuilder address(String address) {
102     this.address = address;
103     return this;
```

Task 1 : Design pattern and its Use Case Scenario Description. (4 points)

Use Case Scenario: A commercial/business bank has decided to fully transition to mobile banking. However, one of the main concerns is appealing to new potential customers. In addition, the bank is concerned with lending to customers who may not have the ability to pay the interest rates. Therefore, the bank is hoping to implement a mobile banking application that can do the following:

1. Run a credit check to see if it meets the Bank's requirements for lending.
2. Check that they can deposit the minimum amount of five-hundred dollars for checking accounts and one-hundred dollars for savings accounts.
3. Check for overdraft fees.

Once designed, the Bank hopes to implement the system immediately to attract new customers. Here, is an example use case provided below to make a withdrawal.

Use Case Example: Existing Customer makes a withdrawal

Use Case	**Description**
Name	Withdrawing money.
Description	An existing customer would like to withdraw an amount of twenty dollars.
Actors	Customer
Pre-conditions	1. The customer has an existing account. 2. The customer has a mobile application. 3. The bank system has existing information regarding the customer.
Primary Scenario	1. The customer will access the mobile application. 2. The customer will be prompted to enter the account number. 3. Once entered, the user will be prompted to enter "w", "t", "b", or "d" in terms of the type of transaction. 3. The customer enters "w". 4. The customer is prompted to enter the amount of money to withdraw. 5. The customer enters "20.00". 6. The customer makes the withdrawal successfully and a message is displayed saying it's successful.
Alternate Scenario	1. If user has insufficient funds, they will be notified that the withdrawal will proceed. However, they will have a negative balance due to the transaction fees. 2. If the customer withdraws all their balance, a message will appear that they have a balance of zero dollars.

Design Goals for this system:

The design goals for this system are the following:

1. The system shall be able to create a Business Checking and Savings account.
2. The system shall be able to check for balance.
3. The system shall be able to make withdrawals.
4. The system shall be able to make deposits.
5. The system shall be able to charge a service fee if over 6withdrawals is made, or the user is about to have a negative balance.
6. The system shall be able to check for statements that contain the words "checking" and "savings".

Flexibility of System

How the system is flexible is that there is an abstract BankAccount class. An abstract class "optionally provides implementation code for each declared operation", meaning that it defers implementation to subclasses (Rogers, 2001). This allows the ability to add different types of BankAccount classes.

Second, there is a customer build class. Having a builder class allows for flexibility because it allows adding optional attributes. For example, the customer build attributes such as first name, last name, and phone are private (but not final). Per "how to do in java", this makes it a mutable attribute.

Simplicity and Understandability of System

How the system is simplified and understandable is the following:

1) Each class has a singular responsibility. For example, the responsibility of the CustomerBuilder static class is to "build" the customer based on the information provided in the customer file.

2) Each class name and method have a clear purpose. For example, the "deposit" method is used to deposit funds.

3) The system implements Builder Patterns, which assists with constructing objects. This pattern "separates object creation" and "object representation" (StackAbuse). This makes it simpler for the user to view.

How duplicate code is avoided

Duplicate code is avoided through inheritance and refactoring.

For inheritance, the "BankAccount" abstract class has withdraw, deposit, transfer and check balance methods. These methods can be inherited by the "BusinessChecking" and "BusinessSavings" classes. This way, these methods do not have to be re-written in subclasses.

Second, the "checkBalBelow" attribute was "pulled up" to the BankAccount Abstract class. This helped reduce the line of code and reduce duplicates.

Design Patterns used

1. Factory Pattern
Class Name: Bank
Lines of Code: 34-52
Explanation: The reason for selecting this design pattern is because multiple Business Checking and Business Savings accounts may be created. Incorporating a creational pattern will fit this criteria. Second, the right new objects need to be created at runtime (Teymourian, module 2 lecture notes). Having a method ensures this.

When the "Bank" is initialized, the files will be read using "readFile" and objects are created. The method is private because from a business standpoint, the bank wants to ensure the Customer information is private and secure. Making this method public will make it easier for unauthorized users to obtain the customer information.

2. Builder Pattern
Class Name: Customer
Lines of Code: 91-93
Explanation: The reason for selecting this design pattern is because it can establish a "class hierarchy" for the Customer class. A customer may have multiple attributes that may be optional or mandatory. To effectively create customer objects, a "Builder Pattern" can assist with that.

Design Trade - offs

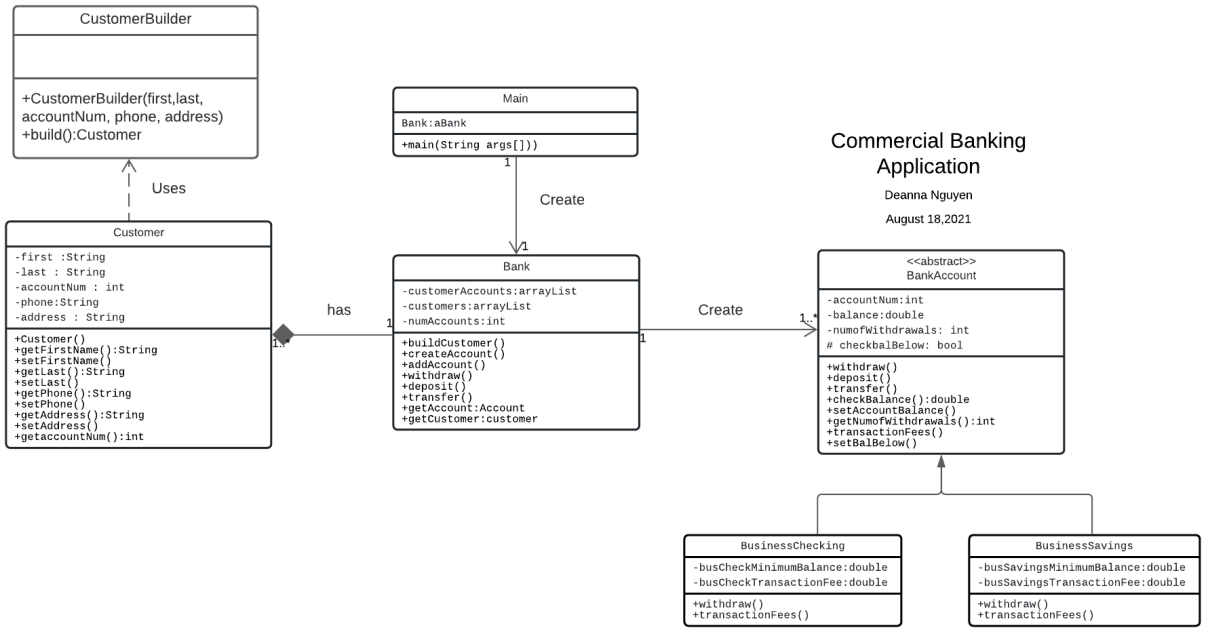
1. High Coupling
Class Names: BankAccount
Lines of Code: 19
Explanation: The checkBalBelow attribute is protected. This is because it needs to be used to determine if a transaction fee is charged. This creates a dependency between the subclasses and parent class. This is not preferred as the attributes should not be seen between other classes in the package. However, this attribute needs to be protected so that the "transaction fees" can be deducted for each banking account type.

2. Modularity
Class Names: Bank
Lines of Code: 58-96, 108-145
Explanation: The class is responsible for both, processing the customer information and bank account information. Although a customer class and builder was created, this program could improve on modularity. However, it was designed this way to ensure there customer information could be retrieved and match with an account. The customer information was necessary to create an account and having a class like this would link them ("factory").

3. Increased lines of Code
Class Names: Customer
Lines of Code: 74-113
Explanation: Per StackAbuse, the lines of code increase due to the implementation of the static builder class and methods. This is not a favorable practice because the goal is to reduce the lines of code.

Task II: UML Diagram

The diagram is designed based on the design pattern descriptions stated above.



Task III: Implement Project

How to compile the project

To compile the project, type "mvn clean compile". The results should appear below:

```
--- maven-compiler-plugin:3.1:compile (default-compile) @ JavaProjectTemplate
Changes detected - recompiling the module!
Compiling 6 source files to C:\Users\dnguy\Documents\met-cs-665-project-deang
-------------------------------------------------------------------------
BUILD SUCCESS
-------------------------------------------------------------------------
Total time:  1.964 s
Finished at: 2021-08-15T03:08:10-07:00
```

How to run Main

To run main, type

mvn -q clean compile exec:java -Dexec.executable="edu.bu.met.cs665.Main" -Dlog4j.configuration="file:log4j.properties".

As a user, input the account number (qualifying account numbers are 1000,1001 and 1005).

```
dnguy@LAPTOP-AR9L3V5R MINGW64 ~/Desktop/met-cs665-assignment-project-deanguyen (master)
$ mvn -q clean compile exec:java -Dexec.executable="edu.bu.met.cs665.Main" -Dlog4j.configuration="file:log4j
roperties"
Reading the files and initiating customer bank accounts.
Account number: 1007 is ineligible to be created.
The file has been processed.
Enter your integer account number here:
1000
```

Input "w", "t", "b" or "d". Case does not matter.

```
What Transaction would you like to do today?Type in 'w' for withdraw, 'd' for deposit,
't' for transfer or 'b' for check balance. Case will be ignored.
```

If "w", "t", or "d", enter the dollar amount in double as shown below.

```
Enter the amount of money to withdraw,deposit, or transfer.
20.00
```

The output should appear as shown below

```
For deposits over 20.00, 1 0cents charge will apply. Otherwise,
the balance will be subtracted from deposit amount applied.
The balance for account holder: 1000 is 1520.0 dollars.
```

If select "b", it should show like this:

```
Enter your integer account number here:
1000
What Transaction would you like to do today?Type in 'w' for withdraw, 'd' for deposit,
't' for transfer or 'b' for check balance. Case will be ignored.
b
Your current balance is: 1500.0
The balance for account holder: 1000 is 1500.0 dollars.
```

If bad user input, this will appear below:

```
Enter your integer account number here:
1001
What Transaction would you like to do today?Type in 'w' for withdraw, 'd' for deposit,
't' for transfer or 'b' for check balance. Case will be ignored.
o
Cannot process transaction
The balance for account holder: 1001 is 300.0 dollars.
```

How to run the Junit tests

To run jUnit tests, type "mvn clean compile test". The results should appear below:

```
Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.808 s
[INFO] Finished at: 2021-08-15T03:09:41-07:00
[INFO] ------------------------------------------------------------------------
'cmd' is not recognized as an internal or external command,
operable program or batch file.
```

Summary of Junit tests

1. Test for retrieving the balance for account numbers 1000 and 1001. The balances should be 1500 for account holder 1 and 300 for account holder 1001.
2. Test to deposit 0, 21.00 and 1500 for both account holders. It is expected a 10 cent charge will be applied to both accounts for deposits over 20.00 dollars.
3. Test to withdraw 0, 21.00 and 1500 for both accounts. It is expected that a 30 dollar transaction fees are applied when account balance is negative.
4. Test to transfer 0, 21.00 and 1500.00 between the two accounts.
5. Test to retrieve first and last name for both account holders based on the customer accounts.

How to check for bugs

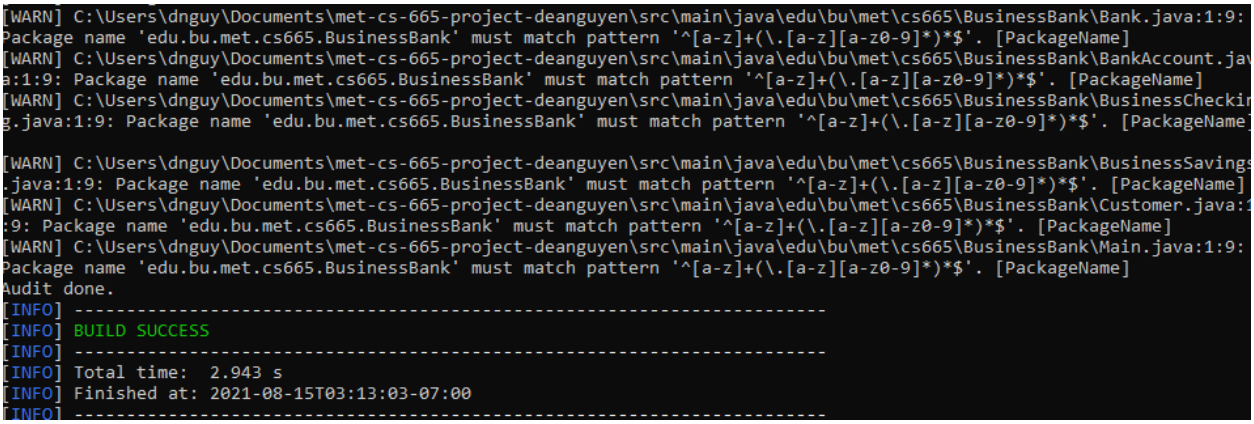
To run bug tests, type " mvn spotbugs:check". The results should appear below:

```
[INFO] --- spotbugs-maven-plugin:4.1.3:check (default-cli) @ JavaProjectTemplate ---
[INFO] BugInstance size is 0
[INFO] Error size is 0
[INFO] No errors/warnings found
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  7.344 s
[INFO] Finished at: 2021-08-15T03:11:38-07:00
```

How to check for style

To run checkstyle tests, type " mvn spotbugs:check". The results should appear below:

Note: The errors are from the package name, which was preassigned by professor Teymourian.



Credits:

https://www.nerdwallet.com/best/small-business/business-checking-accounts

https://jmvidal.cse.sc.edu/csce145/fall06/Ch12/Ch12%20Slides.pdf

https://www.infoworld.com/article/2075627/maximize-flexibility-with-interfaces-and-abstract-classes.html

Facilitator Michael Huang: Providing recommendation for Builder Pattern at https://java-design-patterns.com/patterns/builder/

Professor Kia Teymourian For Project Template, Explanation of Design Patterns at: https://learn.bu.edu/bbcswebdav/pid-9107738-dt-content-rid-56186957_1/courses/21sum2metcs665so2/course/module1/allpages.htm

River City Bank: https://rivercitybank.com/resources/fee-schedule-bus/

GeekForGeeks Builder Design Pattern UML: https://www.google.com/search?q=uml+diagram+builder+pattern&rlz=1C1CHBF_enUS855US855&tbm=isch&sxsrf=ALeKk000aa4dKl4h3pqnN25tayH5FIpZSA:1629018837297&source=lnms&sa=X&ved=0ahUKEwjZj4SJ2LLyAhUlFFkFHf8IAwAQ_AUIyQUoAQ&biw=1366&bih=625#imgrc=IrvDn_XcQ6tWUM

How to Do it In Java: Builder Design Pattern: https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/

Tutorials point for UML Diagram: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Business Insider: https://www.businessinsider.com/mobile-banking-market-trends

Caroline GoldStein Credit Score: https://www.fundera.com/business-loans/guides/credit-score-for-business-loan
https://stackabuse.com/the-builder-design-pattern-in-java/