HW16: Bank Accounts: Modified to Include Sorting and Searching Methods

You have been hired as a programmer by a major bank. Your first project is a small banking transaction system. The user of the program (the teller) can create a new account, as well as perform deposits, withdrawals, balance inquiries, close accounts, etc.

For this assignment, you must have at least the following classes:

1. A **Bank** class which consists of an unsorted **ArrayList of Accounts** currently active or closed.
   In addition, the Bank class has several static member variables and methods:
   totalAmountInSavingsAccts - sum total of balances in all Savings accounts
   totalAmountInCheckingAccts - sum total of balances in all Checking accounts
   totalAmountInCDAccts - total - sum total of balances in all CD accounts
   totalAmountInAllAccts - total - sum total of balances in all accounts
   Make sure to **provide appropriate methods** so as to allow for the **addition to, subtraction from, and reading of**, the current values of each of these static variables.
   Be sure to **print the values of all of these static variables when you print the database of accounts**.
   The Bank Class also has an **ArrayList of Integers** (e.g., **ArrayList<Integers> acctNumSortKey)** to act as a sorting key to sort the database of Accounts by account numbers.
   The Bank Class also has an **ArrayList of Integers** (e.g., **ArrayList<Integers> ssnSortKey)** to act as a sorting key to sort the database of Accounts by SSNs.
   The Bank Class also has an **ArrayList of Integers** (e.g., **ArrayList<Integers> nameSortKey)** to act as a sorting key to sort the database of Accounts alphabetically by Name objects.
   The **Bank** class does not override either the toString() or the equals() method.
2. An **Account** class which consists of a **Depositor, an account number, an account type, account status (open or closed), account balance, an ArrayList of TransactionReceipts performed on the account (Note: creating an account is considered a transaction.).**
   The **Account** class must have a copy constructor and override both the toString() and the equals() methods.
   The Account class has several subclasses:

2a. The **SavingsAccount** class is a subclass of the Account class.
   For Saving Accounts, deposits and withdrawals are allowed at any time.
   The **SavingsAccount** class must have a copy constructor and override both the toString() and the equals() methods.

2b. The **CheckingAccount** class is a subclass of the Account class.
   For Checking Accounts, deposits, withdrawals, and check clearing are allowed at any time. Remember, you may only clear a check if the date on the check is no more than six months ago. No post-dated checks (checks with a future date) may be cleared. Use the **Calendar class** to implement this. In addition, a check will clear only if there is sufficient funds in the account. If the account lacks sufficient funds, the check will not clear and the account will be charged a $2.50 Service Fee for "bouncing" a check. In addition to the previous rules, if the current balance of the account is below $2500, each withdrawal or cleared check is charged a fee of $1.50.
   The **CheckingAccount** class must have a copy constructor and override both the toString() and the equals() methods.

2c. The **CDAccount** class is a subclass of the Account class.
   The class has a data member: **a maturityDate** which is a **Calendar** class object.
   As before, deposits and withdrawals will be allowed only on or after the maturity date. When a deposit or withdrawal is made, have the user select a new maturity date for the CD. The choices are either 6, 12, 18, or 24 months from the date of the deposit or withdrawal. Again, use the **Calendar** class to implement this.
   The **CDAccount** class must have a copy constructor and override both the toString() and the equals() methods.

3. A **Depositor** class which has a **Name and a social security number**.
   The **Depositor** class must have a copy constructor and override both the toString() and the equals() methods.
4. A **Name** class which consists of **first and last names**.
   The **Name** class must have a copy constructor and override the **toString()**, the **equals()**, and the **compareTo()** methods.
5. A **Check** class with data fields consisting of **an account number, the check amount, and a dateOfCheck**
   The **Check** class must have a copy constructor and override the toString() method.
6. A **TransactionTicket** class with data fields consisting of **a dateOfTransaction, typeOfTransaction (deposit, withdrawal, balance inquiry, new account, delete account, etc.), account number, amountOfTransaction (for deposits and withdrawals), termOfCD (6, 12, 18, or 24 months - see below)**.
   The **TransactionTicket** class must have a copy constructor and override the toString() method.
7. A **TransactionReceipt** class with data fields consisting of **a TransactionTicket, successIndicatorFlag, reasonForFailure String, preTransactionBalance, postTransactionBalance, postTransactionMaturityDate (for CDs).**
   The **TransactionReceipt** class must have a copy constructor and override the toString() method.

You must implement appropriate methods in each class so as to implement the functionality required.

The **data members of each class must be private (or protected when providing subclass access)**.
Provide accessor and mutator methods as necessary.

An Account object should access subclass methods using polymorphism.

Remember, all I/O should be done only in the methods of the class that contains the main() method.

Initially, the account information of existing customers is to be read into the database.use method readAccts() described below. **(Note: the ArrayList of Accounts will grow dynamically as each account is created.)** The program keeps track of the actual number of currently active accounts.
After initialization, **a neatly formatted table of the initial unsorted database of active accounts should be printed.** Use method printAccts() described below

Now sort The ArrayList of Accounts into ascending numerical order by account number using the Account Number sort key (e.g., acctNumSortKey) and the QuickSort Algorithm. After sorting, print a neatly formatted table of the sorted database of active accounts. Use method printAcctsByAcctNumSortKey() to do this.

Now sort The ArrayList of Accounts into ascending order by SSN using the SSN sort key (e.g., ssnSortKey) and the Bubble Sort Algorithm. After sorting, print a neatly formatted table of the sorted database of active accounts by SSN. Create and use method printAcctsBySSNSortKey() to do this.

Now sort The ArrayList of Accounts into alphabetical order by Name using the Name sort key (e.g., nameSortKey) and the Insertion Sort Algorithm. After sorting, print a neatly formatted table of the sorted database of active accounts by in alphabetical order. Create and use method printAcctsByNameSortKey() to do this.

The program then allows the user to select from the following menu of transactions:

Select one of the following:
- W - Withdrawal
- D - Deposit
- C - Clear Check
- N - New account **(NOTE: the ArrayList of Accounts will grow when you create a new Account)**
- B - Balance
- I - Account Info (without transaction history) **(NOTE: include at least one depositor who has multiple accounts)**
- H - Account Info plus Account Transaction History
- S - Close Account (close (shut), but do not delete the account) **(Note: no transactions are allowed on a closed account)**
- R - Reopen a closed account
- X - Delete Account (close and delete the account from the database)) **(NOTE: must have zero balance to delete)**
- Q - Quit

Note 1: The Clear Check transaction is only valid for checking accounts. It is like a withdrawal; except, **you must also check the date of the check.** You may only clear a check if the date on the check is no more than six months ago. No post-dated checks (checks with a future date) may be cleared. Use the **Calendar class** to implement this. In addition, a check will clear only if there is sufficient funds in the account. If the account lacks sufficient funds, the check will not clear and the account will be charged a $2.50 Service Fee for "bouncing" a check.

Note 2: **CD accounts will now contain a maturityDate.** Deposits and Withdrawals will be allowed only on or after the maturity date. When a deposit or withdrawal is made, have the user select a new maturity date fro the CD. the choices are either 6, 12, 18, or 24 months from the date of the deposit or withdrawal. Again, use the **Calendar** class to implement this.

The **main method** then prompts the user for a selection. You should verify that the user has typed in a valid selection (otherwise print out an error message and repeat the prompt). Once the user has entered a valid selection, using a **switch** statement, appropriate methods (in the class that contains the main() method) should be called to perform the specific transaction. These methods will call the class implemented methods as necessary.

At the end, before the user quits, the program prints the contents of the final database four ways:
- **a) unsorted**
- **b) sorted by account number (using the QuickSort Algorithm)**
- **c) sorted by SSN (using the Bubble Sort Algorithm)**
- **d) sorted alphabetically (using the Insertion Sort Algorithm)**

Invalid operations are to be handled by exceptions. You should minimally implement the following exceptions:
- InvalidAccountException
- InvalidAmountException
- AccountClosedException
- InsufficientFundsException
- InvalidMenuSelectionException
- PostDatedCheckException
- CheckTooOldException
- AccountNotCheckingAccountException
- CDMaturityDateException

All exceptions are to be handled within the Bank Class except for the InvalidMenuSelectionException which is to be handled within main().

Note: Make sure that when a method in an aggregate class returns a reference to a field object, it returns a reference to a copy of the field object.

Make sure to use enough test cases so as to completely test program functionality.
Make sure that there is at least one depositor that has multiple accounts at the bank.
Make sure that there is at least two depositors with the same last name and different first names.

Notes:
1. All output must be file directed (i.e., sent to an output file)
2. Only output must go to the file - not interactive prompts and menus.
3. No global variables are allowed
4. The program and all methods must be properly commented.
5. All data members of classes are to be private (or protected when providing subclass access)
6. All I/O should be done within the methods of the class that contains the main() method.

Extra Credit #1:
The initial database of Accounts should be printed as follows:
- **a) unsorted**
- **b.1) sorted by account number (using the QuickSort Algorithm - using sort key acctNumQuickSortKey)**
- **b.2) sorted by account number (using the Bubble Sort Algorithm - using sort key acctNumBubbleSortKey)**
- **b.3) sorted by account number (using the Insertion Sort Algorithm - using sort key acctNumInsertionSortKey)**
- **c.1) sorted by SSN (using the QuickSort Algorithm - using sort key ssnQuickSortKey)**
- **c.2) sorted by SSN (using the Bubble Sort Algorithm - using sort key ssnBubbleSortKey)**
- **c.3) sorted by SSN (using the Insertion Sort Algorithm - using sort key ssnInsertionSortKey)**
- **d.1) sorted by Name (using the QuickSort Algorithm - using sort key nameQuickSortKey)**
- **d.2) sorted by Name (using the Bubble Sort Algorithm - using sort key nameBubbleSortKey)**
- **d.3) sorted by Name (using the Insertion Sort Algorithm - using sort key nameInsertionSortKey)**

At the end, before the user quits, the program prints the contents of the final database as follows:
- **a) unsorted**
- **b.1) sorted by account number (using the QuickSort Algorithm - using sort key acctNumQuickSortKey)**
- **b.2) sorted by account number (using the Bubble Sort Algorithm - using sort key acctNumBubbleSortKey)**
- **b.3) sorted by account number (using the Insertion Sort Algorithm - using sort key acctNumInsertionSortKey)**
- **c.1) sorted by SSN (using the QuickSort Algorithm - using sort key ssnQuickSortKey)**
- **c.2) sorted by SSN (using the Bubble Sort Algorithm - using sort key ssnBubbleSortKey)**
- **c.3) sorted by SSN (using the Insertion Sort Algorithm - using sort key ssnInsertionSortKey)**
- **d.1) sorted by Name (using the QuickSort Algorithm - using sort key nameQuickSortKey)**
- **d.2) sorted by Name (using the Bubble Sort Algorithm - using sort key nameBubbleSortKey)**
- **d.3) sorted by Name (using the Insertion Sort Algorithm - using sort key nameInsertionSortKey)**

Minimal Class Requirements:

1a. The Bank class should **at least** have a default (No-Arg) constructor that would allow statements of the form:

```
Bank bank = new Bank();                    //implements a No-Arg Constructor
```

1b. The Bank class should have **at least** have the following methods:

```
public TransactionReceipt openNewAcct(TransactionTicket...)   Must maintain the sorted ArrayList of Accounts
public TransactionReceipt deleteAcct(TransactionTicket...)    Must maintain the sorted ArrayList of Accounts
public int findAcct(...) - must use binary search (return value indicates index of found Account or reason for failure)
public Account getAcct(...)                    //returns a reference to a copy of the requested Account.
public Account getAcctByAcctNumSortKey(...)    //returns a reference to a copy of the requested Account.
public Account getAcctBySSNSortKey(...)        //returns a reference to a copy of the requested Account.
public Account getAcctByNameSortKey(...)       //returns a reference to a copy of the requested Account.
public int getNumAccts(...)
private void sortAccountsByAcctNum(...)        Note: the sort methods are all to be private to the Bank class
private void sortAccountsBySSN(...)
private void sortAccountsByName(...)
```

2a. The Account class should **at least** have a constructor that would allow statements of the form:

```
Account account = new Account(...);        //implement both a No-Arg and a Parametized Constructor
```

2b. The Account class should have **at least** the following methods:

```
public TransactionReceipt getBalance(TransactionTicket...)
public TransactionReceipt makeDeposit(TransactionTicket...)          //implemented via the appropriate subclass
public TransactionReceipt makeWithdrawal(TransactionTicket...)       //implemented via the appropriate subclass
public TransactionReceipt clearCheck(Check...)                       //implemented via the appropriate subclass
public TransactionReceipt closeAcct(TransactionTicket...)
public TransactionReceipt reopenAcct(TransactionTicket...)
public ArrayList<TransactionReceipt> getTransactionHistory(TransactionTicket...)
public Depositor getDepositor(...)
public int getAcctNumber(...)
public String getAcctType(...)
public Calendar getMaturityDate(...)
public void addTransaction(TransactionReceipt...)
```

3a. The Depositor class should **at least** have a constructor that would allow statements of the form:

```
Depositor depositor = new Depositor(...);  //implement both a No-Arg and a Parametized Constructor
```

3b. The Depositor class should have **at least** the following methods:

```
public Name getName(...)
public String getSSN(...)
public void setName(...)
public void setSSN(...)
```

4a. The Name class should **at least** have a constructor that would allow statements of the form:

```
Name name = new Name(...);                 //implement both a No-Arg and a Parametized Constructor
```

4b. The Name class should have **at least** the following methods:

```
public String getFirstName(...)
public String getLastName(...)
public void setFirstName(...)
public void setLastName(...)
```

5a. The Check class should **at least** have a constructor that would allow statements of the form:

```
Check check = new Check(...);
```

6a. The TransactionTicket class should **at least** have a constructor that would allow statements of the form:

```
TransactionTicket transactionTicket = new TransactionTicket(...);
```

6b. The TransactionTicket class should have **at least** the following methods:

```
public Calendar getDateOfTransaction(...)
public String getTransactionType(...)
public int getAcctNumber(...)
public double getTransactionAmount(...)
public int gettermOfCD(...)
```

7a. The TransactionReceipt class should **at least** have a constructor that would allow statements of the form:

```
TransactionReceipt transactionReceipt = new TransactionReceipt(...);
```

7b. The TransactionReceipt class should have **at least** the following methods:

```
public TransactionTicket getTransactionTicket(...)
public boolean getTransactionSuccessIndicatorFlag(...)
public String getTransactionFailureReason(...)
public double getPreTransactionBalance(...)
public double getPostTransactionBalance(...)
public Calendar getPostTransactionMaturityDate(...)
```

8. **Add additional constructors and methods to each class as necessary.**

9. The class containing the main() method should have at least the following methods (see below for additional information):

```
public static void main(String[] args)
public static void readAccts(...)
public static void printAccts(...);          (make sure to include the printing of the static variables)
public static void menu(...)
public static void balance(...);
public static void deposit(...);
public static void withdrawal(...);
public static void clearCheck(...);
public static void acctInfo(...);
public static void acctInfoHistory(...);
public static void newAcct(...);
public static void closeAcct(...);
public static void reopenAcct(...);
public static void deleteAcct(...);
```

The transaction methods in main() should "fill out" a TransactionTicket object, and then call the appropriate method within the Bank class to carry out the requested transaction. The method should then print an appropriate transaction receipt (see samples below). These methods must make use of the overridden toString() methods as appropriate:

Methods in the class containing the main() method:

public static void readAccts(Bank bank)

This method reads in and loads the initial database of accounts from an input file. The data format contained in the input file is:

```
fisrtName lastName SSN acctNum acctType balance maturityDate(for CD accounts)
```

Sample initial database info:

```
John  Doe    123445678  123456  Savings    200.55
Jim   Beam   345556789  567890  Checking  1234.56
Jane  Eyre   345667890  987654  Savings      2.33
Tom   Sawyer 456778901  234567  CD         500.00   7/22/2021
Huck  Finn   567889012  345678  Checking   123.98
John  Doe    123445678  222222  CD        5000.00  12/12/2021
John  Doe    123445678  333333  Checking   999.99
Huck  Finn   567889012  654321  Savings    543.66
Jack  Spratt 678990123  785609  Savings    333.33
Jane  Doe    456789012  389765  Checking   888.56
Jane  Doe    456789012  123123  Savings   8765.43
```

public static void menu()

This method only displays the menu. The **main program** then prompts the user for a selection.

public static void balance(Bank bank, PrintWriter outFile, Scanner inFile)

This method prompts the user for an account number. If the account does not exist, it prints an error message. Otherwise, it prints the account balance.

public static void deposit(Bank bank, PrintWriter outFile, Scanner inFile)

This method prompts the user for the account number. If the account does not exist, it prints an error message. Otherwise, it asks the user for the amount of the deposit.

public static void withdrawal(Bank bank, PrintWriter outFile, Scanner inFile)

This method prompts the user for the account number. If the account does not exist, it prints an error message. Otherwise, it asks the user for the amount of the withdrawal. If the account does not contain sufficient funds, it prints an error message and does not perform the transaction.

public static void clearCheck(Bank bank, PrintWriter outFile, Scanner inFile)

This method prompts the user for the account number. If the account does not exist, it prints an error message. Otherwise, it asks the user for the date and the amount of the check. If the check is invalid it prints an error message and does not perform the transaction.

```
public static void newAcct(Bank bank, PrintWriter outFile, Scanner inFile)
```

This method prompts the user for a new account number. If the account already exists, it prints an error message. Otherwise, it adds the account to the database. **The method then prompts the user to enter the new depositor's first name, last name, social security number, the account type (Checking, Savings, or CD), and the initial opening deposit. Make sure that the database stays sorted (by account number, by SSN, and by Name) each time you add a new account.**

```
public static void deleteAcct(Bank bank, PrintWriter outFile, Scanner inFile)
```

This method prompts the user for an account number. If the account does not exist, or if the account exists but has a non-zero balance, it prints an error message. Otherwise, it closes and deletes the account. It returns the new number of accounts. **Make sure that the database stays sorted (by account number, by SSN, and by Name) each time you delete a n account.**

```
public static void closeAcct(Bank bank, PrintWriter outFile, Scanner inFile)
```

This method prompts the user for an account number. If the account does not exist, it prints an error message. Otherwise, it closes the account. No transactions are allowed on a closed account.

```
public static void reopenAcct(Bank bank, PrintWriter outFile, Scanner inFile)
```

This method prompts the user for an account number. If the account does not exist, it prints an error message. Otherwise, it reopens (or leaves open) the account. Transactions are once again allowed on a reopened account.

```
public static void accountInfo(Bank bank, PrintWriter outFile, Scanner inFile)
```

This method prompts the user for a **social security number (SSN)**. If no account exists for this SSN, it prints an error message. Otherwise, it prints the complete account information **for all of the accounts with this SSN**.

```
public static void accountInfoWithTransactionHistory (Bank bank,
                                                       PrintWriter outFile, Scanner inFile)
```

This method prompts the user for a **social security number (SSN)**. If no account exists for this SSN, it prints an error message. Otherwise, it prints the complete account information and transaction history **for all of the accounts with this SSN**.

```
public static void printAccts(Bank bank, PrintWriter outFile)
```

This method prints an **unsorted, neatly formatted table** (with column headings) of the complete account information for every active account. The column headings should include: **Last Name, First Name, SSN, Account Number, Account Type, Account Status (open or closed), Balance (with a precision of 2), Maturity Date (used for CD accounts)**.

```
public static void printAcctsByAcctNumSortKey(Bank bank, PrintWriter outFile)
```

This method prints a **neatly formatted table** (with column headings) of the complete account information for every active account **sorted by account number**. The column headings should include: **Last Name, First Name, SSN, Account Number, Account Type, Account Status (open or closed), Balance (with a precision of 2), Maturity Date (used for CD accounts)**.

```
public static void printAcctsBySSNSortKey(Bank bank, PrintWriter outFile)
```

This method prints a **neatly formatted table** (with column headings) of the complete account information for every active account **sorted by SSN**. The column headings should include: **Last Name, First Name, SSN, Account Number, Account Type, Account Status (open or closed), Balance (with a precision of 2), Maturity Date (used for CD accounts)**.

```
public static void printAcctsByNameSortKey(Bank bank, PrintWriter outFile)
```

This method prints a **neatly formatted table** (with column headings) of the complete account information for every active account **sorted by Name**. The column headings should include: **Last Name, First Name, SSN, Account Number, Account Type, Account Status (open or closed), Balance (with a precision of 2), Maturity Date (used for CD accounts)**. The method must use the overridden toString() method of the Account class to print the attributes of each Account.

Notes:

1. **All output must be file directed**
2. **Only output must go to the file - not interactive prompts and menus.**
3. **No global variables are allowed**
4. **The program and all methods must be properly commented.**
5. **All data members of classes are to be private or protected (as appropriate)**
6. **The program must be properly tested.**
    - **a. The initial database should consist of at least 10 accounts**
    - **b. The initial database should be read from an input file**
    - **c. Account numbers are integers of 6 digits in the range 100000 - 999999**
    - **d. Balances are real numbers in dollars and cents**
    - **e. At least two depositors have multiple accounts**
    - **e. At least one depositors has all three account types**
    - **f. The initial database should be printed as a neat, formatted table to the output file**
    - **g. Test at least 2 balance inquiries:**
        - **i. valid account**
        - **ii. invalid account**
    - **h. Test at least 5 deposits:**
        - **i. valid account - valid deposit amount**
        - **ii. valid account - invalid deposit amount (i.e., negative amount)**
        - **iiii. valid CD account - valid day to deposit - CD renewal**
        - **iv. valid CD account - invalid day to deposit (CD maturity day not reached)**
        - **V. invalid account**
    - **i. Test at least 6 withdrawals:**
        - **i. valid account - valid withdrawal amount**
        - **ii. valid account - invalid withdrawal amount (i.e., negative amount)**
        - **iii. valid account - insufficient funds**
        - **iiii. valid CD account - valid day to withdraw - CD renewal or closure**
        - **v. valid CD account - invalid day to withdraw (CD maturity day not reached)**
        - **vi. invalid account**
    - **j. Test at least 6 clear check transactions:**
        - **i. valid account - valid amount**
        - **ii. valid account - invalid amount (i.e., negative amount)**
        - **iii. valid account - insufficient funds**
        - **iv. valid account - post dated check**
        - **v. valid account - check too old**
        - **vi. invalid account**
    - **k. Test close account transactions**
        - **i. valid account - close the account - attempt transactions on closed account**
        - **ii. invalid account**
    - **l. Test reopen account transactions**
        - **i. valid account - account is closed - reopen the account - attempt transactions on reopened account**
        - **ii. invalid account**
    - **m. Test at least 4 acctInfo transactions**
        - **i. valid depositor - depositor has a single account**
        - **ii. valid depositor - depositor has multiple accounts (can have multiple accounts of the same type)**
        - **iii. valid depositor - depositor has all three account types**
        - **iv. invalid account**
    - **n. Test at least 4 acctInfoWithTransactionHistory transactions**
        - **i. valid depositor - depositor has a single account**
        - **ii. valid depositor - depositor has multiple accounts (can have multiple accounts of the same type)**
        - **iii. valid depositor - depositor has all three account types**
        - **iv. invalid account**
    - **o. Create at least 6 new accounts with an initial non-zero balance (two of each type: Savings, Checking, CD)**
    - **p. Test the creation of at least one invalid new account**
    - **q. Test several transactions on the new accounts (deposits, withdrawals, clear check, etc.)**
    - **r. Test the deletion of at least three accounts (accounts must be old accounts that had no transactions until now)**
        - **i. valid account - valid deletion: (i.e., account exists and has a zero balance)**
        - **ii. valid account - invalid deletion: (account has a non-zero balance, withdraw the balance, delete again)**
        - **iii. invalid account: (account does not exist)**
    - **s. Test at least 2 invalid menu selections**
    - **t. Quit and print the final database**
7. **All I/O should be done only within the methods of the class that contains the main() method.**

Sample Transaction Output:

Transaction Type: Balance Inquiry
Account Number: 987654
Current Balance: $300.50

Transaction Type: Balance Inquiry
Account Number: 999888
Error: Account 999888 does not exist

Transaction Type: Deposit
Account Number: 987654
Current Balance: $300.50
Amount to Deposit: $123.45
New Balance: $423.95

Transaction Type: Deposit
Account Number: 987654
Current Balance: $423.95
Amount to Deposit: $-100.00
Error: Invalid Deposit Amount - Transaction voided

Transaction Type: Withdrawal
Account Number: 786543
Current Balance: $975.25
Amount to Withdraw: $5000.00
Error: Insufficient Funds - Transaction voided

Transaction Type: Withdrawal
Account Number: 786543
Current Balance: $975.25
Amount to Withdraw: $100.00
New Balance: $875.25

Transaction Type: Clear Check
Account Number: 786543
Current Balance: $875.25
Amount to Withdraw: $5000.00
New Balance: $872.75
Error: Insufficient Funds Available - Bounce Fee Charged

Error: 'Z' is not a valid menu selection

Extra Credit #2:
Revise Extra Credit #1 as follows:
Implement the Bank Class sorting methods using Comparators (see textbook pages 1156-1157)

```
private void bubbleSortByComparator(ArrayList<Integer> sortKey,
                                                    Comparator<Account> comp)
private void insertionSortByComparator(ArrayList<Integer> sortKey,
                                                    Comparator<Account> comp)
private void quickSortByComparator(ArrayList<Integer> sortKey, int start, int end,
                                                    Comparator<Account> comp)
```

Submission Requirements:
Create a folder on Google Drive that will contain the following:
1. The source files (i.e., *.java files) for each of the implemented Classes:
- **pgmHW16.java**
- **Bank.java; Account.java; Depositor.java, Name.java**
- **SavingsAccount.java; CheckingAccount.java; CDAccount.java**
- **Check.java; TransactionTicket.java; TransactionReceipt.java;**
- **implemented Comparator Classes (if done)**
- **implemented Exception Classes**
- **etc.**

2. The text file containing the initial database of accounts (e.g., initAccounts.txt)
3. The test cases text file (e.g., myTestCases.txt)
4. The output text file which contains all of the required program output (e.g., pgmOutput.txt)
Then, make the folder shareable and send me a link to the folder.