|  |  |
|---|---|
| **Course:** | **INFO3134 W2020** |
| **Professor:** | **Bill Pulling / Madhavi Mohan** |
| **Assignment:** | **Project # 1: Polymorphic Banking** |
| **Service Packs:** | None yet! |
| **Due Date:** | **Files: Friday, March 6, 2020 by 8:00 p.m. (2000 hours)** |

**BankAccount Class Hierarchy**

In this project, you will design a class hierarchy to model a real-world bank application that will create bank account objects and record transactions made in the account.

A customer can have two types of bank accounts: a personal chequing account is an account on which the user can write cheques to pay for purchases. This type of account will pay a low interest rate.

A savings account is intended to encourage customers to leave their money in the bank by not allowing cheques to be written on the account, and it also pays a higher interest rate.

This application is intended to be executed over the period of one month. The classes that will be used in this application include:

- BankAccount, an abstract superclass which holds features that are common to it subclasses

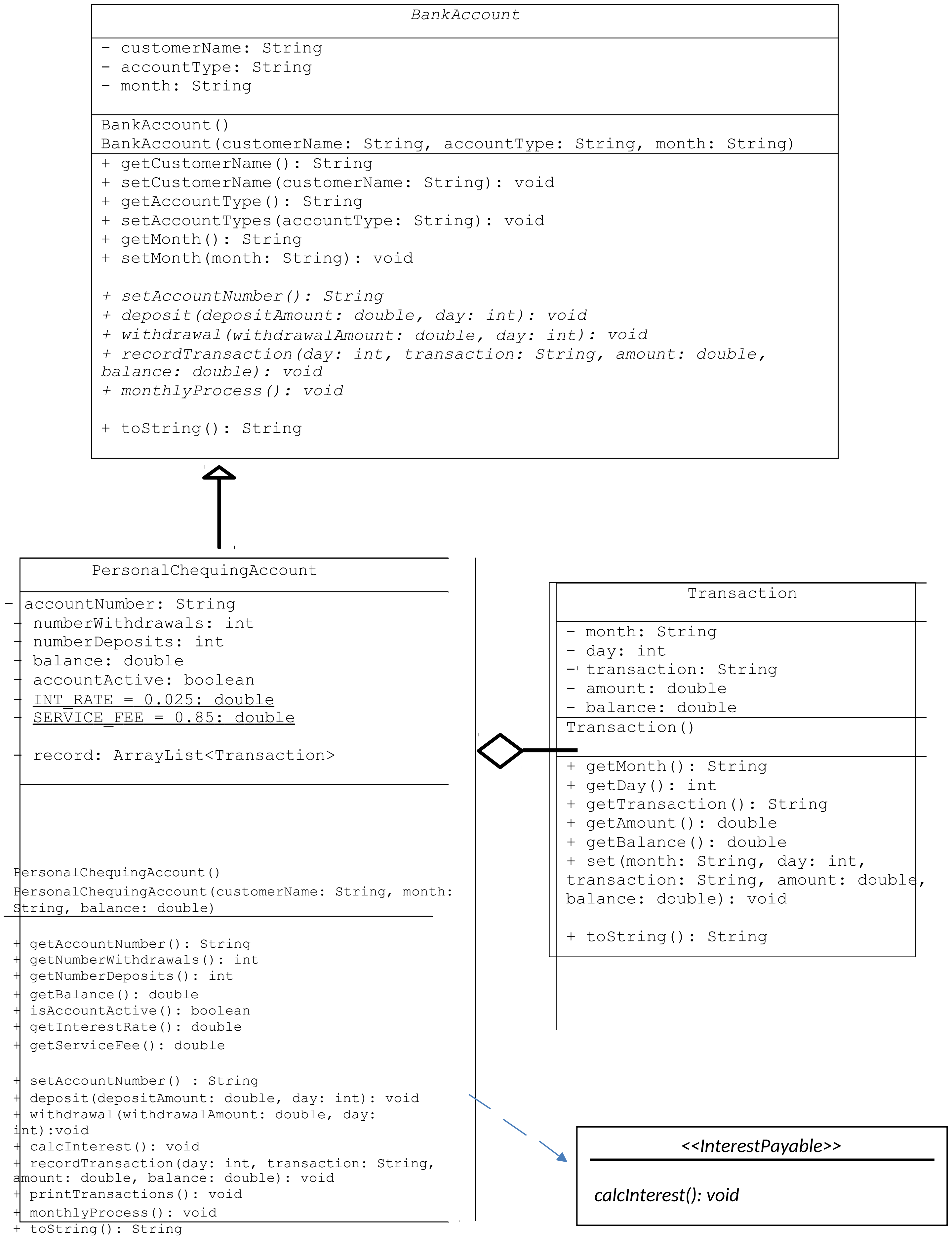
- PersonalChequingAccount, a concrete subclass of BankAccount

- SavingAccount, another concrete subclass of BankAccount.

- Transaction, an aggregate class whose objects will be instantiated in the two subclasses and stored in an ArrayList structure in order to keep track of the monthly transactions against each account object.

There is also an interface called InterestPayable, which will hold one abstract method called calcInterest(), and it will be implemented differently by each of the two subclasses.

Use the UML diagram and the class specifications that follow on the next pages to code your BankAccount application.



```
BankAccount
- customerName: String
- accountType: String
- month: String

BankAccount()
BankAccount(customerName: String, accountType: String, month: String)

+ getCustomerName(): String
+ setCustomerName(customerName: String): void
+ getAccountType(): String
+ setAccountTypes(accountType: String): void
+ getMonth(): String
+ setMonth(month: String): void

+ setAccountNumber(): String
+ deposit(depositAmount: double, day: int): void
+ withdrawal(withdrawalAmount: double, day: int): void
+ recordTransaction(day: int, transaction: String, amount: double,
balance: double): void
+ monthlyProcess(): void

+ toString(): String
```

```
PersonalChequingAccount
- accountNumber: String
- numberWithdrawals: int
- numberDeposits: int
- balance: double
- accountActive: boolean
- INT_RATE = 0.025: double
- SERVICE_FEE = 0.85: double

- record: ArrayList<Transaction>

PersonalChequingAccount()
PersonalChequingAccount(customerName: String, month:
String, balance: double)

+ getAccountNumber(): String
+ getNumberWithdrawals(): int
+ getNumberDeposits(): int
+ getBalance(): double
+ isAccountActive(): boolean
+ getInterestRate(): double
+ getServiceFee(): double

+ setAccountNumber() : String
+ deposit(depositAmount: double, day: int): void
+ withdrawal(withdrawalAmount: double, day:
int):void
+ calcInterest(): void
+ recordTransaction(day: int, transaction: String,
amount: double, balance: double): void
+ printTransactions(): void
+ monthlyProcess(): void
+ toString(): String
```

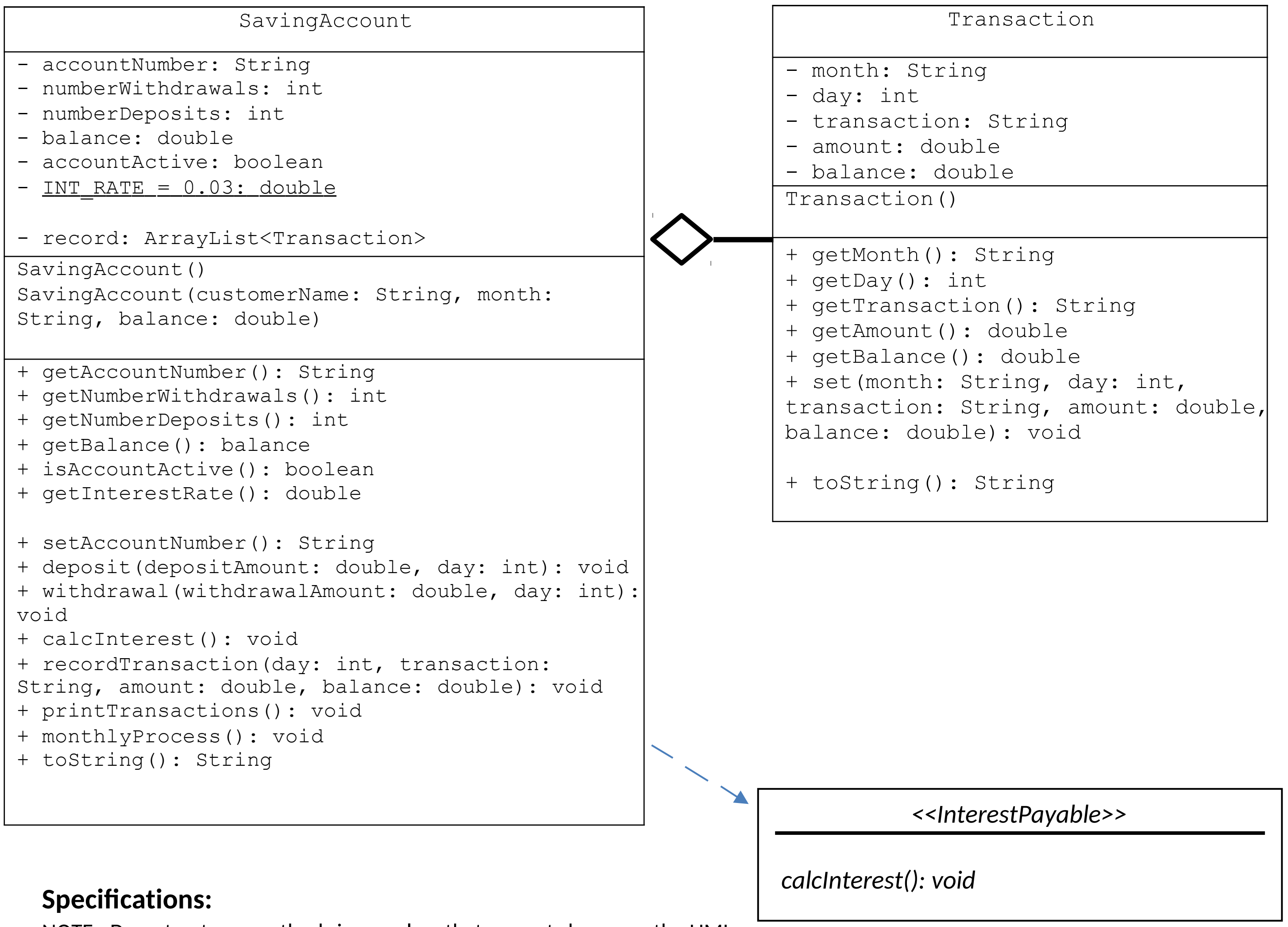
```
Transaction
- month: String
- day: int
- transaction: String
- amount: double
- balance: double
Transaction()

+ getMonth(): String
+ getDay(): int
+ getTransaction(): String
+ getAmount(): double
+ getBalance(): double
+ set(month: String, day: int,
transaction: String, amount: double,
balance: double): void

+ toString(): String
```

```
<<InterestPayable>>

calcInterest(): void
```


SavingAccount
- accountNumber: String
- numberWithdrawals: int
- numberDeposits: int
- balance: double
- accountActive: boolean
- INT_RATE = 0.03: double
- record: ArrayList<Transaction>
SavingAccount()
SavingAccount(customerName: String, month: String, balance: double)
+ getAccountNumber(): String
+ getNumberWithdrawals(): int
+ getNumberDeposits(): int
+ getBalance(): balance
+ isAccountActive(): boolean
+ getInterestRate(): double
+ setAccountNumber(): String
+ deposit(depositAmount: double, day: int): void
+ withdrawal(withdrawalAmount: double, day: int): void
+ calcInterest(): void
+ recordTransaction(day: int, transaction: String, amount: double, balance: double): void
+ printTransactions(): void
+ monthlyProcess(): void
+ toString(): String
Transaction
- month: String
- day: int
- transaction: String
- amount: double
- balance: double
Transaction()
+ getMonth(): String
+ getDay(): int
+ getTransaction(): String
+ getAmount(): double
+ getBalance(): double
+ set(month: String, day: int, transaction: String, amount: double, balance: double): void
+ toString(): String
<<InterestPayable>>
calcInterest(): void

## Specifications:

NOTE: Do not put any methods in any class that are not shown on the UML diagram! Especially pay attention to the use of getters and setters.

## BankAccount is an abstract class

- for the no-arg constructor, set the field values to null
- for the multi-arg constructor, set the field values to the arguments
- there are five abstract methods that will be overridden and implemented in the subclasses
    - setAccountNumber()
    - deposit(double depositAmount, int day)
    - withdrawal(double withdrawalAmount, int day)
    - recordTransaction(int day, String transaction, double amount, double balance)
    - monthlyProcess()
- for the toString() method have the method return a String that reads:

| | |
|---|---|
| Customer | *customerName* |
| Account Type: | *accountType* |
| Current Month: | *month* |

## Transaction is an aggregate class

- An aggregate class is just a class whose object can be a data member in another class
- Transaction objects will be stored in an ArrayList data structure in PersonalChequingAccount and SavingAccount to save daily transactions as they occur throughout the month. So, the record data member in each type of account is as ArrayList.
    - private ArrayList<Transaction> record = new ArrayList<Transaction>();
- For the no-arg constructor, set the fields to null or 0
- For the toString() method have the method return a String that reads as shown. Include use of correct number formats and take steps to ensure alignment of fields in the printed output.
    - *month day transaction*":" *amount* "Balance: " *balance*

## The Interface *InterestPayable*

This interface has just one abstract method, *calcInterest()*. It will be implemented by the both classes. The reason we are putting the *calcInterest*() method in an interface is that not all types of bank accounts pay interest. For example, a straight chequing account for a small business will not usually pay any interest on the balance. Our particular PersonalChequingAccout *does* pay interest, so both it and SavingsAccount will implement this interface. The details on how to implement the method will be provided in the SavingAccount class description.

## PersonalChequingAccount extends BankAccount implements *InterestPayable*

- for the no-arg constructor, set the customer name and account type to null, set account number to null, set accountActive to false, number of withdrawals and deposits to 0 and balance to 0.0
- for the multi-arg constructor, set the field values to the arguments, set the accountType to "Chequing", set the accountNumber by calling the setAccountNumber() method, set the accountActive by calling isAccountActive() method
- for the isAccountActive() method, if the balance falls below $25.00 the account becomes inactive
    - set accountActive to false
- for the setAccountNumber() method, the number is a string that:
    - is made up of the bank's number, the transit number, and a randomly generated account number, followed by account type number (a chequing account is 550)
    - takes the format "002-623490-XXXXXX-550" where each X is a digit within the range of 0-9
- for the deposit() method:
    - update the account balance, update the number of deposits, and update accountActive by calling the isAccountActive() method
    - record the transaction by calling the recordTransaction() method
- for the withdrawal() method:
    - declare a local variable called transactionMessage, used to store information about the transaction
    - if the account balance will not fall below 0.0 and the account is active update the account balance and set the transaction message to "Withdrawal"
    - or if the account balance will fall below 0.0, set the transaction message to "Transaction cancelled. Insufficient funds"
    - or if the account is inactive, set the transaction message to "Transaction cancelled. Account is inactive."
    - update the number of withdrawals
    - update accountActive by calling the isAccountActive() method
    - record the transaction by calling the recordTransaction() method
- for the calcInterest() method: For this void method, it should:
    - determine the last day of the current month, since interest is added on the last day
    - calculate the monthly interest amount (INT_RATE / 12) as long as current balance is $1,000 or greater
    - update the account balance with any interest earned on the account
    - record the transaction by calling the recordTransaction() method
- for the recordTransaction() method:
    - create a new Transaction object
    - set the five Transaction object fields (month, day, transaction, amount, balance):
    - add it to the ArrayList called record: **record.add(trans);**
- for the printTransaction() method:
    - print a title
    - print the ArrayList contents in a nicely formatted manner.
- for the monthlyProcess() method:
    - determine the last day of the current month, since the service fee is charged on the last day
    - calculate the service fee if number of withdrawals for the month is greater than 4 (numberWithdrawals * SERVICE_FEE)
    - update balance with any service fee
    - call the calcInterest() method
    - record the transaction by calling the recordTransaction() method
    - print the transactions for the month by calling the printTransaction() method
    - reset number of withdrawals and deposits and clear the ArrayList elements to get ready for the next month
    - update accountActive by calling the isAccountActive() method
- for the toString() method have the method return a string as shown, being sure to include use of correct number formats and alignment of fields

| | |
|---|---|
| Account Number: | *accountNumber* |
| Number of Withdrawals: | *numberWithdrawals* |
| Number of Deposits: | *numberDeposits* |
| Balance: | *balance* |
| Account Active: | *accountActive* |
| Annual Interest Rate: | *INT_RATE* |

  - Monthly Service Fee Rate: *SERVICE_FEE* (applied to no. of withdrawals if withdrawals are over 4)

**SavingAccount extends BankAccount implemements *InterestPayable***

- for the no-arg constructor, set the customer name and account type to null, set the account number to null, set accountActive to false, number of withdrawals and deposits to 0 and balance to 0.0
- for the multi-arg constructor, set the field values to the arguments, set the accountType to "Saving", set the accountNumber by calling the setAccountNumber() method, set the accountActive by calling isAccountActive() method
- for the isAccountActive() method if the balance falls below $25.00 the account becomes inactive
  - set accountActive to false
- for the setAccountNumber() method, the number is a string that:
  - is made up of the bank's number, the transit number, and a randomly generated account number, followed by account type number (a saving account is 575)
  - takes the format "002-623490-XXXXXX-575" where each X is a digit within the range of 0-9
- for the deposit() method:
  - update the account balance, update the number of deposits, and update accountActive by calling the isAccountActive() method
  - record the transaction by calling the recordTransaction() method
- for the withdrawal() method:
  - declare a local variable called transactionMessage, used to store information about the transaction
  - if the account balance will not fall below 0.0 and the account is active, update the account balance and set the transaction message to "Withdrawal"
  - or if the account balance will fall below 0.0, set the transaction message to "Transaction cancelled. Insufficient funds"
  - or if the account is inactive, set the transaction message to "Transaction cancelled.  Account is inactive."
  - update the number of withdrawals
  - update accountActive by calling the isAccountActive() method
  - record the transaction by calling the recordTransaction() method
- for the implementation of the *abstract calcInterest()* method, which has a void return type:
  - determine the last day of the current month, since interest is added on the last day
  - calculate the monthly interest amount (INT_RATE / 12) as long as balance is $25 or greater (balance * monthly interest rate)
  - **as an incentive to maintain a high balance in the savings account, if  a minimum balance of $2,000 was maintained throughout the month, the interest rate is increased by  0.75%  (i.e. + 0.0075)**
    - **Hint:  you will have to traverse the ArrayList checking  balance after each transaction to see if the bonus rate is applicable.**
  - update the balance with any interest earned on the account
  - record the transaction by calling the recordTransaction() method
- for the recordTransaction() method:
  - create a new Transaction object
  - set the five Transaction object fields (month, day, transaction, amount, balance)
  - add it to the ArrayList called record:   **record.add(trans);**
- for the printTransaction() method:
  - print a title
  - print the ArrayList contents in a nicely formatted manner.
- for the monthlyProcess() method:
  - call the calcInterest() method
  - record the transaction by calling the recordTransaction() method
  - print the transactions for the month by calling the printTransaction() method
  - reset number of withdrawals and deposits and clear the ArrayList elements to get ready for the next month
  - update accountActive by calling the isAccountActive() method

- For the toString() method have the method return a string that reads as shown. Include use of correct number formats and alignment of fields.

| | |
|---|---|
| Account Number: | *accountNumber* |
| Number of Withdrawals: | *numberWithdrawals* |
| Number of Deposits: | *numberDeposits* |
| Balance: | *balance* |
| Account Active: | *accountActive* |
| Annual Interest Rate: | *INT_RATE* |

**To test your program, write a class called BankAccountTester.java**

1. Assume that any subclass of BankAccount object is created *at the beginning of a month*. Using polymorphism, create a PersonalChequingAccount object. Use the multi-arg constructor and the following data:

| | |
|---|---|
| Customer Name: | Janice Joplin |
| Balance: | $2,345 |
| Month: | March |

2. Print the object at the start of the month. Sample output (note that the component of the account number 730012 was generated randomly). You should see something like this:

```
************************************
**** Customer Name:Janice Joplin
Account Type:      PersonalChequing
Current Month:     March
****************************************
Account Number:          002-623490-730012-550
Number of Withdrawals:   0
Number of Deposits:      0
Balance:                 $2,345.00
Account Active:          true
Annual Interest Rate:    2.5%
Monthly Service Fee:        $0.85 (applied to no. of withdrawals if withdrawals over 4)
```

3. Apply the following transactions to Janice's chequing account for the month of March:

| | | |
|---|---|---|
| March 5 | Deposit: | $25.98 |
| March 6 | Withdrawal: | 1,300 |
| March 10 | Withdrawal: | 1,700 |
| March 11 | Deposit: | 1,050 |
| March 11 | Withdrawal: | 1,700 |
| March 25 | Deposit: | 25.98 |
| March 26 | Withdrawal: | 400.00 |
| March 28 | Withdrawal: | 27.00 |
| March 28 | Withdrawal: | 7.50 |

4. Prepare the monthly process, using the monthlyProcess() method, which also prints the month's transaction record. Sample output:

```
*****************************************
**** Transaction Record for the Month of
March
*********************************************
March  5    Deposit:    $25.98      Balance:    $2,370.98
March  6    Withdrawal: $1,300.00   Balance:    $1,070.98
March  10   Transaction cancelled.                    Insufficient funds:     $1,700.00
      Balance:                   $1,070.98
March  11   Deposit:    $1,050.00   Balance:    $2,120.98
March  11   Withdrawal: $1,700.00   Balance:    $420.98
March  25   Deposit:    $25.98      Balance:    $446.96
March  26   Withdrawal: $400.00     Balance:    $46.96
March  28   Withdrawal: $27.00      Balance:    $19.96
March  28   Transaction cancelled.                    Account is inactive:    $7.50Balance:
      $19.96
March  31   Interest:   $0.00       Balance:    $19.96
March  31   Service Fee: $0.00      Balance:    $19.96
```

5. Then re-print the object at the end of the month. You should see something like this.

```
**************************************
** Customer Name:      Janice Joplin
Account Type:

PersonalChequing
Current Month:          March
****************************************
Account Number:         002-623490-730012-550
Number of Withdrawals:  4
Number of Deposits:     3
Balance:               $19.96
Account Active:         false
Annual Interest Rate:   2.5%
Monthly Service Fee:    $0.85 (applied to no. of withdrawals if withdrawals over 4)
```

6. Assume that any subclass of BankAccount object is created at the beginning of a month. Using polymorphism, create a SavingAccount object. Use the multi-arg constructor and the following data:

| | |
|---|---|
| Customer Name: | Elvis Presley |
| Balance: | $6,100 |
| Month: | March |

7. Print the object at the beginning of the month. Sample output:

```
************************************
**** Customer Name: Elvis Presley
Account Type:     Saving
Current Month:    March
****************************************
Account Number:         002-623490-386661-575
Number of Withdrawals:  0
Number of Deposits:     0
Balance:               $6,100.00
Account Active:         true
Annual Interest Rate:   3%
```

8. Apply the following transactions to Elvis's saving account:

| | | |
|---|---|---|
| March 3 | Deposit: | 500 |
| March 6 | Withdrawal: | 1,000 |
| March 15 | Deposit: | 250 |
| March 21 | Withdrawal: | 3,000 |
| March 28 | Withdrawal: | 825 |
| March 29 | Deposit: | 250 |

9. Prepare the monthly process, using the monthlyProcess() method, which also prints the month's transaction record. Sample output:

```
    *****************************************
    **** Transaction Record for the Month of
    March
    *********************************************
    March 3  Deposit:    $500.00     Balance:    $6,600.00
    March 6              $1,000.00   Balance:    $5,600.00
    March 15 Deposit:    $250.00     Balance:    $5,850.00
    March 21             $3,000.00   Balance:    $2,850.00
    March 28             $825.00     Balance:    $2,025.00
    March 29 Deposit:    $250.00     Balance:    $2,275.00
    March 31 Interest:  $7.11       Balance:    $2,282.111
```

10. Then re-print the object after the transactions have been processed.

```
************************************
**** Customer Name: Elvis Presley
Account Type:     Saving
Current Month:    March
****************************************
Account Number:        002-623490-386661-575
Number of Withdrawals:  3
Number of Deposits:     3
Balance:                $2,282.11
Account Active:         true
Annual Interest Rate:   3%
```

## Marking Scheme

| Description | Marks Available | Marks Awarded |
|---|---|---|
| • Each class has appropriate documentation<br>• Utility methods have appropriate documentation<br>• Adequate commenting within methods has been done in all classes<br>• All variables have proper access modifiers assigned to them<br>• All files compile and run without any error messages | 2 | |
| **BankAccount class** is complete according to specifications<br>• Fields and two constructors<br>• Getters/setters<br>• five abstract methods<br>• toString() method | 6 | |
| **PersonalChequingAccount class** is complete according to specifications<br>• Fields, static variables, ArrayList object of type Transaction<br>• Two constructors<br>• Getters/setters<br>• Six implemented methods<br>• printTransactions()<br>• toString() overridden from superclass | 6 | |
| **SavingAccount class** is complete according to specs, differs from PersonalChequingAccount in the following ways:<br>• No SERVICE_FEE field or getter<br>• Different implementation of calcInterest()<br>• Different implementation of monthlyProcess()<br>• toString() overridden from superclass | 6 | |
| **Transaction Class**<br>• Fields and constructor<br>• Getters/setter<br>• toString() is overridden | 3 | |
| **Interface *InterestPayable* is complete and properly implemented by both classes**<br>• Abstract method is properly declared | 2 | |
| • **BankAccountTester class** is coded and all specified actions are carried out.<br>• Two different types of account objects are created polymorphically<br>• All specified transactions for each account object are executed.<br>• All calculations and output are correct. | 5 | |
| Total Marks | 30 | |