420-202-RE Assignment 2 (Revision 1) Due Date: March 21, 2017

Object-Oriented Programming (OOP)

Encapsulation + Inheritance + Polymorphism

OOP Terms

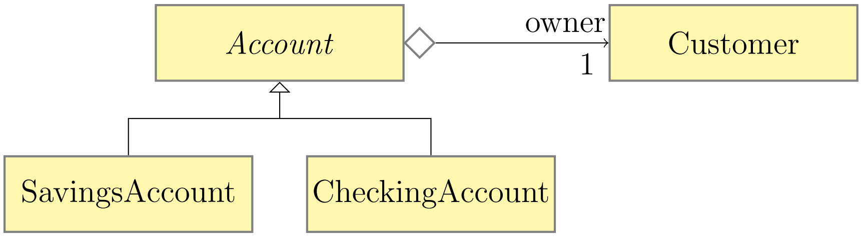
1. A *class* represents objects with *common characteristics*.
2. The *common characteristics* of objects of a class are expressed in terms of their attributes (*data*) and operations (*methods*).
3. *Information hiding* refers to the hiding of implementation details of data and operations while allowing access to public operations.
4. *Encapsulation* refers to the binding of data and operations into a “capsule” or “object”. The class construct in Java implements this concept.
5. Through *inheritance* we can *derive* a new class from an existing *base* class.
6. Through *polymorphism*, we can use a *base* class reference variable to call the overridden methods in the *derived* classes,
7. A *base* class is also called *superclass* or *parent* class.
8. A *derived* class is also called *subclass* or *child* class.
9. A method’s *signature* comprises that method’s name and parameter types.
10. To *override* a method of a superclass , a subclass provides a new version of that method with the same signature, which effectively hides the superclass version from the client.
11. To *overload* a method, a class provides a new version of that method with a different signature. A client of the class can call any of the public versions of overloaded methods.

Objectives

- Reuse code, creating new classes from existing ones
- Create an inheritance hierarchy
- Learn about *protected* access control
- Apply polymorphism in program design
- Override inherited methods, including superclass’s **toString** and **equals** methods.

Your Task

Implement the following inheritance hierarchy that models bank accounts that a bank might offer to customers. Bank customers can deposit money into their accounts and withdraw money from their accounts. Savings and Checking accounts are specific types of bank accounts. Savings accounts earn interest on the money they hold. Checking accounts charge a fee per transaction.



The UML[1] diagram above models an inheritance hierarchy of four classes of which ***Account*** (written in *italics*) represents an *abstract* class and the other three represent *concrete* classes.

You will recall that you cannot instantiate (create) objects of an *abstract* class, but you can always instantiate (create) objects of a *concrete* classes.

The class diagram suggests that:

- the ***Account*** knows about the **Customer** class, and the **Customer** class plays the role of ”owner” of the bank account. However, the **Customer** class has no idea that it is associated with the ***Account***.
- an ***Account*** will always store a reference to a **Customer** instance.
- a **Customer** object is *part-of* ***Account***. This is an aggregation relationship. **Customer** can exist without ***Account***.
- both **SavingsAccount** and **CheckingAccount** inherit from ***Account***, specializing it into concerete classes.

Here are the UML diagrams of the classes involved in our simple banking system:

[1] Unified Modeling Language. See page 324, Chapter 9 (Objects and Classes) of the course textbook, or visit here.

Class Account

Account	Abstract class (since class name is in italic)
− `accountNumber : int`	instance field, stores account number
− `balance : double`	instance field to stores account balance
− `customer : Customer`	instance field, stores customer
− `nextAccountNumber : int`	class field, stores the next unique 4-digit account number, starting at 1001
+ `Account(cust : Customer, bal : double ) :`	Initializes `customer` to a reference to a clone copy of `cust`. Uses `setBalance(bal)` to initialize `ballance` to `bal`. Sets account number to the next unused integer.
+ `getAccountNumber() : int`	Returns account number
+ `getBalance() : double`	Returns balance
+ `getCustomer() : Customer`	Returns customer
# `setBalance(bal : double) : void`	Changes balance. Prints an appropriate error message and sets `balance` to zero if the supplies `bal` is negative
+ `setCustomer(customer : Customer) : void`	Changes customer
+ `deposit(amount : double) : boolean`	Adds `amount` to `balance` and returns `true`, if `amount` is non-negative; otherwise, it displays a warning message and then returns `false`, leaving the account unchanged.
+ `withdraw(amount : double) : boolean`	Deducts `amount` from `balance` and returns `true`, if `amount` is non-negative and is ≤ `balance`; otherwise, it displays a warning message and then returns `false`, leaving the account unchanged.
+ `transferTo(amount : double, other : Account) : boolean`	Transfers specified `amount` from `this` account to the `other` account. Specifically, attempts to withdraw the `amount` from `this` account. If withdrawal transaction succeeds, it then deposits the amount into `other` account and returns `true`; otherwise, displays a warning message and then returns `false`, leaving the accounts unchanged.
+ `transferFrom(amount : double, other : Account) : boolean`	Transfers specified `amount` from the `other` account to `this` account. Specifically, attempts to withdraw the `amount` from the `other` account. If withdrawal transaction succeeds, it then deposits the amount into `this` account and returns `true`; otherwise, displays a warning message and then returns `false`, leaving the accounts unchanged.
+ `performMonthEndProcessing() : void`	Abstract method. Definition deferred to concrete subclasses. Used for polymorphic calls.
+ `toString() : String`	Returns a string representation of this account.

Class CheckingAccount

The important features of a Checking account are:

- **CheckingAccount** provides all of the functionality of **Account**.
- Checking accounts, which have no interest, charge a fee for each transaction (deposits and withdrawals).
- **CheckingAccount** has an instance data field for storing the number of transaction made on the account.
- **CheckingAccount** has a method for computing transaction fees and charging the account. **CheckingAccount** resets the transaction count to zero after deducting the transaction fees from the account.
- The **toString()** method adds the transaction count and transaction fee for the **CheckingAccount** to the information returned from **Account**'s **toString()** method.

A UML diagram for the `CheckingAccount` class is shown bellow:

	CheckingAccount	Concrete class
−	`transactionCount: int`	field to store number of deposits + withdrawals
−	`transactionFee: double`	field to store fee charged per transaction
+	`CheckingAccount(cust : Customer, bal : double, fee : double ):`	Calls *super*'s constructor passing `cust` and `bal` as arguments. Sets `transactionCount` to zero, and `transactionFee` fo `fee`.
+	`deposit(amount:double):boolean`	Calls *super*'s `deposit` to deposit `amount` into this checking account and, if successful, increments the transaction count by 1 and returns true; otherwise returns `false` leaving the account unchanged.
+	`withdraw(amount:double):boolean`	Calls *super*'s `withdraw` to withdraw `amount` from this checking account and, if successful, increments the transaction count by 1, and returns `true`; otherwise returns `false` leaving the account unchanged.
+	`getCount() : int`	Returns transaction count
+	`getFee() : double`	Returns transaction fee
#	`setCount(count :int) : void`	Changes transaction count
#	`setFee(fee :double) : void`	Changes transaction fee
+	`performMonthEndProcessing():void`	Deducts transaction fees from the account and then resets `transactionCount` to zero.
+	`toString() : String`	Returns a string representation of this checking account.

Class SavingsAccount

The important features of a savings account are:

- **SavingsAccount** provides all of the functionality of **Account**.
- A savings account collects interest. **SavingsAccount** has an instance data field for the interest rate.
- **SavingsAccount** has a method for computing interest and adding the interest to the account.
- Interest is paid on the minimum balance during a period. Thus, **SavingsAccount** updates the minimum balance when money is withdrawn from the account.
- The **toString()** method adds the interest rate and minimum balance for the **SavingsAccount** to the information returned from **Account**'s **toString()** method.

A UML diagram for the `SavingsAccount` class is shown bellow:

	SavingsAccount	Concrete class
−	`interest : double`	field to store annual interest rate
−	`minimumBalance: double`	field to store minimum balance value
+	`SavingsAccount(customer:Customer, bal:double, air:double):`	Calls *super*'s constructor passing `cust` and `bal` as arguments. Initializes `interest` to annual interest rate `air`, and `minimumBalance` to zero.
+	`deposit(amount:double):boolean`	Calls *super*'s `deposit` to deposit `amount` into this checking account and return accordingly.
+	`withdraw(amount:double):boolean`	Calls *super*'s `withdraw` to withdraw `amount` from this checking account and, if successful, updates `minimumBalance`, and returns `true`; otherwise returns `false` leaving the account unchanged.
+	`getInterest() : double`	Returns annual interest rate
+	`getMinBalance() : double`	Returns minimum balance
#	`setInterest(air :double) : void`	Changes annual interest rate
#	`setMinBalance(bal :double) : void`	Changes minimum balance
+	`performMonthEndProcessing():void`	Computes interest earned on the minimum balance, deposits the interest into the account, and then resets `minimumBalance` to current balance.
+	`toString() : String`	Returns a string representation of this savings account.

Class Customer

A UML diagram for the `Customer` class is shown bellow:

	Customer	Concrete class
−	`name:String`	field to store name
−	`phone:String`	field to store phone number
−	`email:String`	field to store email address
+	`Customer(name:String, phone:String, email:String):`	normal constructor
+	`Customer(anotherCustomer:Customer):`	copy constructor
+	`getName():String`	Returns name
+	`getPhone():String`	Returns phone number
+	`getEmail():String`	Returns email address
+	`setName(name:String):void`	Changes name
+	`setPhone(phone:String):void`	Changes phone number
+	`setEmail(email:String):void`	Changes email address
+	`equals(owner:Customer):boolean`	Compares this and another customer objects.
+	`toString():String`	Returns string representation

Sample Program Runs

1. Create two customer objects, one for Mary and one for Mark. Use your own values for email addresses and phone numbers:

```
Customer mary = new Customer("Mary", "123456789", "mary@herhotmail.com");
System.out.println(mary + "\n");

Customer mark = new Customer("Mark", "987654321", "mark@hishotmail.com");
System.out.println(mark + "\n");
```

output

```
Name:  Mary
Telephone: 123456789
Email: mary@herhotmail.com

Name:  Mark
Telephone: 987654321
Email: mark@hishotmail.com
```

2. Create a savings account for Mary with initial balance of $100.00 and 5.0% annual interest rate. Print the account.

```java
SavingsAccount herSavings = new SavingsAccount(mary, 100.0, 0.5);
System.out.println("Her initial account profile:\n" + herSavings);
System.out.println();
```

output

```
Her initial account profile:
Name:  Mary
Telephone: 123456789
Email: mary@herhotmail.com
Account No.: 1001
Balance     : 100.00
Account type: Savings
Annual interest rate: 0.50%
Minimum balance: 100.00
```

3. Create a checking account for Mark with initial balance of $150.00 and 0.75 cents fee per transaction. Print the account.

```java
CheckingAccount hisChecking = new CheckingAccount(mark, 150, 0.75);
System.out.println("His initial account profile:\n" + hisChecking);
System.out.println();
```

output

```
His initial account profile:
Name:  Mark
Telephone: 987654321
Email: mark@hishotmail.com
Account No.: 1002
Balance     : 150.00
Account type: Checking
Transactions: 0
Transaction fee: 0.75
```

4. Deposit $10.00 to Mark's account and print the account:

```java
String hisName = hisChecking.getCustomer().getName();
double depositAmt = 10.0;
hisChecking.deposit(depositAmt);
System.out.println("Deposited $" + depositAmt + " to " + hisName + "'s account");
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println();
```

output

```

Deposited $10.0 to Mark's account
Mark's balance : 160.0
```

5. Deposit $10000.00 to Mary's account and print the account:

```java

String herName = herSavings.getCustomer().getName();
depositAmt = 10000.0;
herSavings.deposit(depositAmt);
System.out.println("Deposited $" + depositAmt + " to " + herName + "'s account");
System.out.println(herName + "'s balance : " + herSavings.getBalance());
System.out.println();
```

output

```

Deposited $10000.0 to Mary's account
Mary's balance : 10100.0
```

6. Transfer $90.00 from Mary's account to Mark's and print both accounts:

```java

double transferAmt = 90.0;
herSavings.transferTo(transferAmt, hisChecking);
System.out.println("Transfered $" + transferAmt + " from " + herName
      + " to " + hisName);
```

output

```

Transfered $90.0 from Mary to Mark
Mark's balance : 250.0
Mary's balance : 10010.0

System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println(herName + "'s balance : " + herSavings.getBalance());
System.out.println();
```

7. Withdraw $20 from Mark's account 5 separate times (transactions). Print the account:

```java
double withdrawAmt = 20;
hisChecking.withdraw(withdrawAmt);
hisChecking.withdraw(withdrawAmt);
hisChecking.withdraw(withdrawAmt);
hisChecking.withdraw(withdrawAmt);
hisChecking.withdraw(withdrawAmt);
System.out.println("Withdrew 5 times $" + withdrawAmt + " from " + hisName + "'s account");
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println();
```

output

```
Withdrew 5 times $20.0 from Mark's account
Mark's balance : 150.0
```

8. Apply monthly charge fees to Mark's account. Print the account:

```java
System.out.println("About to apply monthly charge fees to " + hisName + "'s account");
System.out.println("Number of transactions: " + hisChecking.getCount());
hisChecking.PerformMonthEndProcessing();
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println();
```

output

```
About to apply monthly charge fees to Mark's account
Number of transactions: 5
Mark's balance : 146.25
```

9. Repeat Step 8 above.

```java
System.out.println("About to apply monthly charge fees to " + hisName + "'s account");
System.out.println("Number of transactions: " + hisChecking.getCount());
hisChecking.PerformMonthEndProcessing();
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println();
```

output

```
About to apply monthly charge fees to Mark's account
Number of transactions: 0
Mark's balance : 146.25
```

10. Withdraw $20 from Mark's account. Print the account:

```

System.out.println("About to withdraw $" + withdrawAmt + " from " + hisName);
hisChecking.withdraw(withdrawAmt);
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println("His current account profile\n" + hisChecking);
System.out.println();
```

output

```

About to withdraw $20.0 from Mark
Mark's balance : 126.25
His current account profile
Name:  Mark
Telephone: 987654321
Email: mark@hishotmail.com
Account No.: 1002
Balance    : 126.25
Account type: Checking
Transactions: 1
Transaction fee: 0.75
```

11. Transfer $10:00 from Mark's account to Mary's account. Print only balance amounts of both accounts.

```

transferAmt = 10.0;
System.out.println("About to transfer $" + transferAmt + " from " + hisName
      + " to " + herName);
herSavings.transferFrom(transferAmt, hisChecking);
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println(herName + "'s balance : " + herSavings.getBalance());
System.out.println();
```

output

```

About to transfer $10.0 from Mark to Mary
Mark's balance : 116.25
Mary's balance : 10020.0
```

12. Transfer $1000:00 from Mary's account to Mark's account. Print only balance amounts of both accounts.

```

transferAmt = 1000.0;
herSavings.transferTo(transferAmt, hisChecking);
System.out.println("Transfered $" + transferAmt + " from " + herName
      + " to " + hisName);
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println(herName + "'s balance : " + herSavings.getBalance());
System.out.println();
```

output

```

Transfered $1000.0 from Mary to Mark
Mark's balance : 1116.25
Mary's balance : 9020.0
```

13. Make a polymorphic call: apply charges to Mary's account.

```

// make a polymorphic call
// base reference = a subclass object
Account herBankAccount = herSavings;
herBankAccount.PerformMonthEndProcessing();
System.out.println("Polymorphic month end proccesing on " + herName + "'s account");
System.out.println(herName + "'s balance : " + herSavings.getBalance());
System.out.println();
```

output

```

Polymorphic month end proccesing on Mary's account
Mary's balance : 9020.5
```

14. Make a polymorphic call: apply charges to Mark's account.

```

// make a polymorphic call
// base reference = a subclass object
Account hisBankAccount = hisChecking;
hisBankAccount.PerformMonthEndProcessing();
System.out.println("Polymorphic month end proccesing on " + hisName + "'s account");
System.out.println(hisName + "'s balance : " + hisChecking.getBalance());
System.out.println();
```

output

```

Polymorphic month end proccesing on Mark's account
Mark's balance : 1114.75
```

15. Print the two accounts.

```java

System.out.println("Her final account profile\n" + herSavings);
System.out.println();
System.out.println("His final account profile\n" + hisChecking);
System.out.println();
```

output

```

Her final account profile
Name:  Mary
Telephone: 123456789
Email: mary@herhotmail.com
Account No.: 1001
Balance     : 9020.50
Account type: Savings
Annual interest rate: 0.50%
Minimum balance: 9020.50

His final account profile
Name:  Mark
Telephone: 987654321
Email: mark@hishotmail.com
Account No.: 1002
Balance     : 1114.75
Account type: Checking
Transactions: 0
Transaction fee: 0.75
```

Evaluation Criteria	
Correctness of execution of your program	60%
Proper use of required Java concepts	20%
Java API documentation style	10%
Comments on nontrivial steps in code, Choice of meaningful variable names, Indentation and readability of program	10%