Project 9: Cardholders – Part 1

Deliverables:

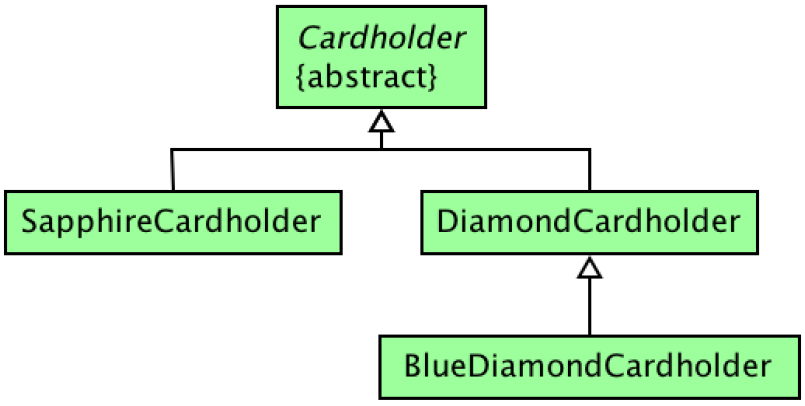
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. In order to avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The grades for the Completed Code submission will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

- Cardholder.java
- SapphireCardholder.java, SapphireCardholderTest.java
- DiamondCardholder.java, DiamondCardholderTest.java
- BlueDiamondCardholder.java, BlueDiamondCardholderTest.java

Specifications – Use arrays in this project; ArrayLists are not allowed!

Overview: This week you will complete the first part of a three-part software project to process the monthly purchases made by credit card company's cardholders. Part 1 of the project focuses on the implementation of the classes of cardholders. The completed class hierarchy is shown in the UML class diagram below.



Cardholder.java

Requirements: The Cardholder class is an *abstract* class from which other cardholder classes are derived. It contains fields and methods that will be inherited by the classes derived from Cardholder.

Design: The Cardholder class has fields, a constructor, and methods as outlined below.

(1) **Fields:** category, acctNumber, name (all String objects), prevBalance of type double, payment of type double, an array of type double representing new purchases; and a public constant INTEREST_RATE set to 0.01 (a double). *The instance variables should be protected rather than private.*

(2) **Constructor:** The constructor accepts String objects for the card holder's acctNumber and name as and assigns the fields accordingly. The constructor should also create the purchases array of type double with initial length of 0 and assign the field.

(3) **Methods:** Usually a class provides methods to access and modify each of its instance variables (i.e., getters and setters) along with any other required methods. The methods for Cardholder are described below.

- `getAcctNumber` and `setAcctNumber` are the getter and setter methods for the acctNumber fields. The getter accepts no parameters and returns a String. The set method accepts a String and does not return a value.
- `getName` and `setName` are the getter and setter methods for the name field. The getter accepts no parameters and returns a String. The set method accepts a String and does not return a value.
- `getPrevBalance` and `setPrevBalance` are the getter and setter methods for the previous balance field. The set method accepts a double and does not return a value. The getter accepts no parameters and returns a double.
- `getPayment` and `setPayment` are the getter and setter methods for the payment field. The set method accepts a double and does not return a value. The getter accepts no parameters and returns a double.
- `getPurchases` and `setPurchases` are the getter and setter methods for the purchases array field. The set method accepts a double[] and does not return a value. The getter accepts no parameters and returns a double[].
- `addPurchases` has no return value and accepts a variable length parameter list of type double and adds the double values in the list to the purchases array (see end notes). Before values are added, the purchases array should be copied and replaced with a new array of appropriate length so that the length of the array reflects the number of values in the array. Note that java.util.Arrays contains a static method called Arrays.copyOf which takes two parameters (double[] orginal, int newLength) and returns the copy of the array with increased or decreased length. For example, the following statement makes a copy of an array called purchases and with the length increased by 1, and then replaces the previous array with the new one.

```
purchases = Arrays.copyOf(purchases, purchases.length + 1);
```

- `deletePurchases` has no return value and accepts a variable length parameter list of type double and deletes them from the purchases array. A value is deleted from the purchases array by locating the value to be removed and then copying each element to the right one

place to the left, thus, overwriting the element to be removed. The purchases array should be copied and replaced with a new array which has a length one less than the previous array so that the length of the array reflects the number of values in the array. If a value to be deleted is not found in the purchases array, then the array is left unchanged. You may want to create a private method that deletes a single purchase and then call it from the deletePurchases method described here as it loops through the array of purchases to be deleted.

- `interest` returns double calculated as (previous balance - payment) * INTEREST_RATE.
- `totalPurchases` returns a double based on the total of the items in the purchase array.
- `balance` returns a double representing previous balance with interest plus the total purchases.
- `currentBalance` returns a double calculated as follows:
  previous balance - payment + interest + total of new purchases;
- `minPayment` returns double calculated as 3% of current balance.
- `toString` returns a String describing a Cardholder (see example output below). This method should use a DecimalFormat object with the pattern ("$#,##0.00") to format the values representing dollars and a DecimalFormat object with the pattern ("#,##0") to format the values representing purchase points as shown in the example return value for a SapphireCardholder.

```
Sapphire Cardholder
AcctNo/Name: 10001 Smith, Sam
Previous Balance: $1,200.00
Payment: ($200.00)
Interest: $10.00
New Purchases: $548.00
Current Balance: $1,558.00
Minimum Payment: $46.74
Purchase Points: 548
```

- `purchasePoints` is an *abstract* method that returns an int representing the purchase points earned on the total purchases in the purchases array.

Code and Test: Since the Cardholder class is abstract you cannot create instances of Cardholder upon which to call the methods. However, these methods will be inherited by the subclasses of Cardholder. You should consider first writing skeleton code for the methods in order to compile Cardholder so that you can create the first subclass described below. At this point you can begin completing the methods in Cardholder and writing the JUnit test methods for your subclass that tests the methods in Cardholder.

SapphireCardholder.java

Requirements: The SapphireCardholder class is derived from the Cardholder class. Objects of this class will receive one purchase point per dollar spent.

Design: The Preferred class has fields, constructor, and methods as outlined below.

(1) **Fields:** No fields are added. Note that category, acctNumber, name, prevBalance, payment, purchases array, and INTEREST_RATE are inherited from Cardholder.
(2) **Constructor:** The constructor accepts String objects for the card holder's acctNumber and name and passes them to the super class's constructor. It should also set the category field to "Sapphire Cardholder".
(3) **Methods:** The methods in Cardholder are inherited; the following methods need to be added:
   - `purchasePoints` overrides the abstract method inherited from the Cardholder and returns an int calculated at one point per whole dollar of the total purchases in the purchases array.

Code and Test: Since the parent of this class was abstract, the inherited methods, as well as the method created in this class, should be tested in the JUnit test file for this class. You will need to complete any inherited methods that have not been completed. It is common to complete the methods in the source files as you develop the associated test methods in the JUnit test file. You also may want create a driver class with a main method that creates instances of the class, sets fields as appropriate, and invokes the toString method and perhaps others to get you started.

DiamondCardholder.java

Requirements: The DiamondCardholder class is derived from the Cardholder class. An Object of this class will receive three purchase points per dollar spent as well as a 5% discount on the subtotal.

Design: The DiamondCardholder class has fields, constructor, and methods as outlined below.
(1) **Fields:** Adds a field for discountRate set to .05, a double. Other fields are inherited from Cardholder. *This instance variable should be protected rather than private.*
(2) **Constructor:** The constructor accepts String objects for the customer's acctNumber and name and passes them to the super class's constructor. It should also set the category field to "Diamond Cardholder".
(3) **Methods:** The methods in Cardholder are inherited; the following methods need to be added:
   - `getDiscountRate` and `setDiscountRate` are the getter and setter methods for the discountRate field. The getter accepts no parameters and returns a double. The set method accepts a double and does not return a value.
   - `purchasePoints` overrides the abstract method inherited from the Cardholder and returns an int calculated at 3 points per dollar of the total purchases in the purchases array.
   - `totalPurchases` overrides the inherited `totalPurchases` method to include the discount indicated by the discountRate field.
   - `toString` overrides the inherited toString to return a String that includes a statement that a discount was applied to subtotal (see example return value below). This method should use a DecimalFormat object with the pattern ("0.0%") to format the discount rate.

```
Diamond Cardholder
AcctNo/Name: 10002 Jones, Pat
Previous Balance: $1,200.00
Payment: ($0.00)
Interest: $12.00
New Purchases: $473.10
```

```
Current Balance: $1,685.10
Minimum Payment: $50.55
Purchase Points: 1,419
(includes 5.0% discount rate applied to New Purchases)
```

Code and Test: As this class and its methods are implemented, the methods should be tested in the JUnit test file for this class. You also may want create a driver class with a main method that creates instances of the class, sets fields as appropriate, and invokes the toString method and perhaps others to get you started.

BlueDiamondCardholder.java

Requirements: The BlueDiamondCardholder class is derived from the DiamondCardholder class. An Object of this class will receive five purchase points per dollar spent as well as a 10% discount on the subtotal. Also, if the new purchases exceed $2,500.00, a bonus of 2,500 purchase points is earned.

Design: The BlueDiamondCardholder class has fields, constructor, and methods as outlined below.

(1) **Fields:** Adds a field for bonusPurchasePoints set to 2500, an int. Other fields are inherited from Cardholder and DiamondCardholder. *This instance variable should be private.*

(2) **Constructor:** The constructor accepts String objects for the customer's acctNumber and name and passes them to the super class's constructor. The category field should be set to "Blue Diamond Cardholder" and the discountRate should be set to 0.10 in constructor.

(3) **Methods:** The methods in Cardholder are inherited; the following methods need to be added:

- `getBonusPurchasePoints` and `setBonusPurchasePoints` are the getter and setter methods for the bonusPurchasePoints field. The getter accepts no parameters and returns an int. The set method accepts an int and does not return a value.
- `purchasePoints` overrides the inherited method and returns an int calculated at 5 points per dollar of the total purchases in the purchases array. If the total is greater than $2,500.00, the value of bonusPurchasePoints is also added.
- `toString` overrides the inherited toString to returns a String that includes the toString of the superclass and if the subtotal is greater than $2,500, a statement indicating that bonus points were added to the purchase points (see examples of two return values below). This method should use a DecimalFormat object with the pattern ("#,##0") to format the bonus purchase points.

```
Blue Diamond Cardholder
AcctNo/Name: 10003 King, Kelly
Previous Balance: $1,200.00
Payment: ($0.00)
Interest: $12.00
New Purchases: $538.20
Current Balance: $1,750.20
Minimum Payment: $52.51
Purchase Points: 2,690
```

```
(includes 10.0% discount rate applied to New Purchases)

Blue Diamond Cardholder
AcctNo/Name: 10004 Jenkins, Jordan
Previous Balance: $1,200.00
Payment: ($0.00)
Interest: $12.00
New Purchases: $9,000.00
Current Balance: $10,212.00
Minimum Payment: $306.36
Purchase Points: 47,500
(includes 10.0% discount rate applied to New Purchases)
(includes 2,500 bonus points added to Purchase Points)
```

Code and Test: As this class and its methods are implemented, the methods should be tested in the JUnit test file for this class. You also may want create a driver class with a main method that creates instances of the class, sets fields as appropriate, and invokes the toString method and perhaps others to get you started.

Example Cardholder Objects and Output (generated by the toString methods)

```
SapphireCardholder sc = new SapphireCardholder("10001", "Smith, Sam");
sc.addPurchases(34.5, 100.0, 63.50, 350.0);
sc.setPrevBalance(1200.0);
sc.setPayment(200);
System.out.println(sc + "\n");

DiamondCardholder dc = new DiamondCardholder("10002", "Jones, Pat");
dc.addPurchases(34.5, 100.0, 63.50, 300.0);
dc.setPrevBalance(1200.0);
System.out.println(dc + "\n");

BlueDiamondCardholder bdc = new BlueDiamondCardholder("10003", "King, Kelly");
bdc.addPurchases(34.5, 100.0, 63.50, 300.0, 100.0);
bdc.setPrevBalance(1200.0);
System.out.println(bdc + "\n");

BlueDiamondCardholder bdc2 = new BlueDiamondCardholder("10004", "Jenkins, Jordan");
bdc2.addPurchases(5000.0, 1000.0, 4000.0);
bdc2.setPrevBalance(1200.0);
System.out.println(bdc2 + "\n");
```

Output:

```
Sapphire Cardholder
AcctNo/Name: 10001 Smith, Sam
Previous Balance: $1,200.00
Payment: ($200.00)
Interest: $10.00
New Purchases: $548.00
Current Balance: $1,558.00
Minimum Payment: $46.74
Purchase Points: 548

Diamond Cardholder
AcctNo/Name: 10002 Jones, Pat
Previous Balance: $1,200.00
Payment: ($0.00)
Interest: $12.00
New Purchases: $473.10
Current Balance: $1,685.10
Minimum Payment: $50.55
Purchase Points: 1,419
(includes 5.0% discount rate applied to New Purchases)

Blue Diamond Cardholder
AcctNo/Name: 10003 King, Kelly
Previous Balance: $1,200.00
Payment: ($0.00)
Interest: $12.00
New Purchases: $538.20
Current Balance: $1,750.20
Minimum Payment: $52.51
Purchase Points: 2,690
(includes 10.0% discount rate applied to New Purchases)

Blue Diamond Cardholder
AcctNo/Name: 10004 Jenkins, Jordan
Previous Balance: $1,200.00
```

```
Payment: ($0.00)
Interest: $12.00
New Purchases: $9,000.00
Current Balance: $10,212.00
Minimum Payment: $306.36
Purchase Points: 47,500
(includes 10.0% discount rate applied to New Purchases)
(includes 2,500 bonus points added to Purchase Points)
```

Notes

Below is an example of a variable length parameter list of type double ter declared in a method:

```
myMethod(double... myList)
```

where `myMethod` is the method name and `myList` is the name of the array where the double values will be collected. Here are several examples of how `myMethod` can be called. Variables of type double could be used in place of the double literals.

```
myMethod(12.25);
myMethod(12.25, 23.43);
myMethod(12.25, 23.43, 14.19, 45.67);
double dArr = {12.25, 23.43, 14.19, 45.67};
myMethod(dArr)
```