Lab 9 – Aggregation Relationship (12 pts)

Lab Objectives

- Be able to write a copy constructor
- Be able to write `equals` and `toString` methods
- Be able to use objects made up of other objects (aggregation)
- Be able to write methods that pass and return objects

Deliverables

This lab has three tasks. When you have all tasks done, run the report in Blackboard. The report is a Blackboard test with short-answer, file-response, multiple-answer, and other types of questions.

In the report, you may be asked to provide code segments, Java source code files (must have extension *.java*), screenshot of program execution, files in PDF format, and your analysis of the results.

If a short answer question requests a code segment, please ensure that your input is readable: all **new lines and indents** are in place.

Screenshots in your report **must show a full screen**, so your computer can be identified. Please resize your IDE panels the way that the required dialog or output is visible along with the source code. Show as much source code as possible.

NOTE:

- Use **Blackboard only** to submit your work; **no email** submission unless your instructor directs it.
- If Blackboard gives you multiple submission attempts (usually three), the **last one** will be evaluated and graded.
- **No late submissions**, **no changes** in your submission after the due date.

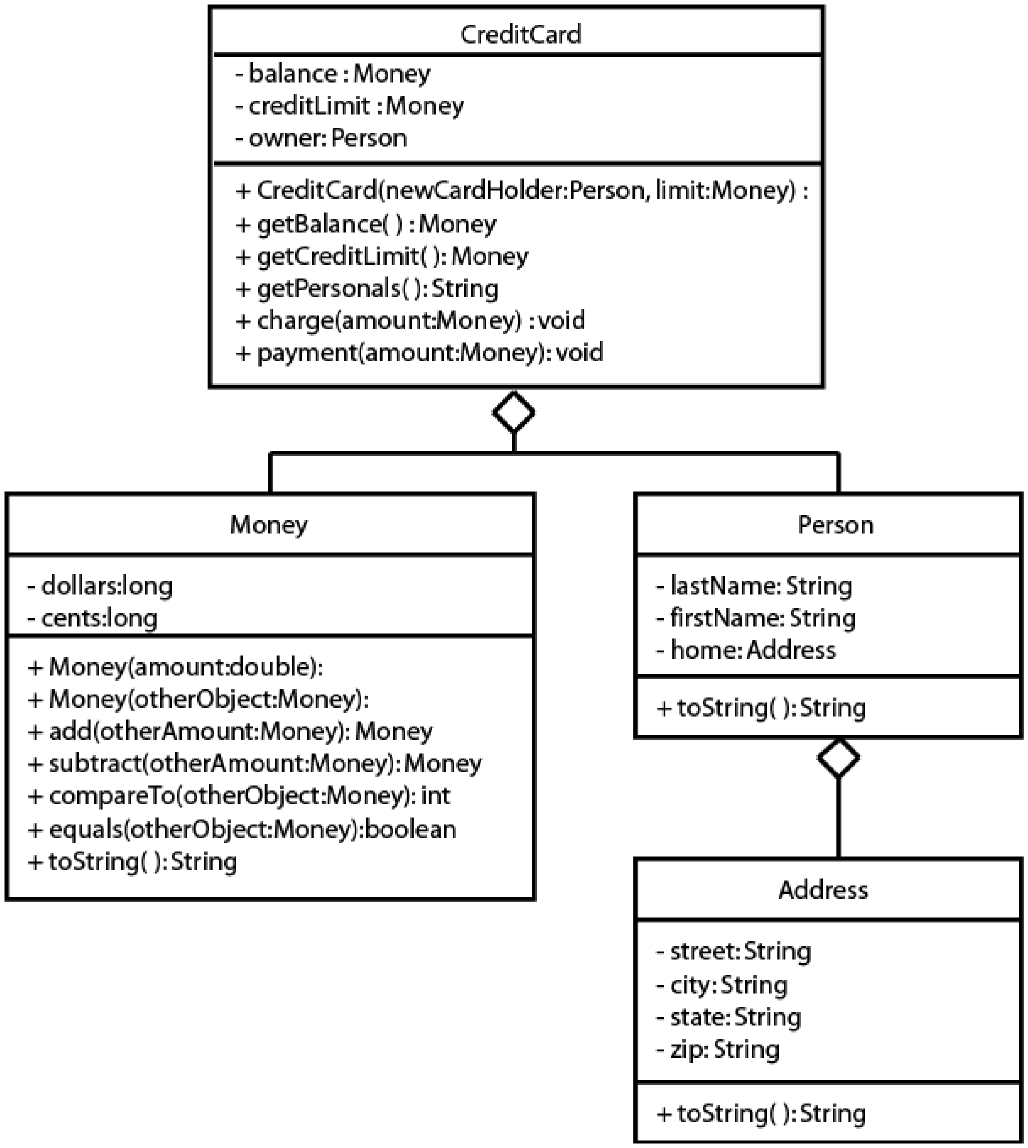
Introduction

Before doing this lab please read textbook Ch. 8.7 and review Lectures 9a and 10a.

In this lab the object we are going to create is more complicated than we worked on before. It is made up of other objects. This is called aggregation. A credit card is an object that is very common, but not as simple as you can think first.

Attributes of the credit card include information about the owner, as well as a balance and credit limit. These things would be our instance fields. A credit card allows you to make payments and charges. These would be methods. As we have seen before, there would also be other methods associated with this object in order to construct the object and access its fields.

Examine the UML diagram that follows. Notice that the instance fields in the `CreditCard` class are other types of objects: a `Person` object and a `Money` object. We can say that the `CreditCard` object *"has a"* `Person` object, which means aggregation, and the `Person` object *"has a"* `Address` object as one of its instance fields. This aggregation structure can create a very complicated object. We will try to keep this lab reasonably simple.



To start with, we will be editing a partially written class, `Money`. The constructor that you will be writing is a copy constructor. This means it should create a new object, but with the same values in the instance variables as the object that is being copied.

Next, we will write the `equals` and `toString` methods. These are very common methods that are needed when you write a class to model an object. You will also see a `compareTo` method that is also a common method for objects.

After we have finished the `Money` class, we will write a `CreditCard` class. This class contains `Money` objects, so you will use the methods that you have written to complete the `Money` class. The `CreditCard` class will explore passing objects and the possible security problems associated with it. We will use the copy constructor we wrote for the `Money` class to create new objects with the same information to return to the user through the accessor methods.

Task #1 Writing a Copy Constructor (2 pts)

1. Copy the files *Address.java*, *Person.java*, *Money.java*, *MoneyDemo.java*, and *CreditCardDemo.java* as directed by your instructor.
   Address.java, *Person.java*, *MoneyDemo.java*, and *CreditCardDemo.java* are complete and will not need to be modified.
   We will start by modifying *Money.java*.
2. Overload the constructor. The constructor that you will write will be a copy constructor. It should use the parameter `Money` object to make a duplicate `Money` object, by copying the value of each instance variable from the parameter object to the instance variable of the new object.

Task #2 Writing the `equals` and `toString` methods (4 pts)

1. Write and document an `equals` method. The method compares the instance variables of the calling object with instance variables of the parameter object for equality and returns `true` if the `dollars` and the `cents` of the calling object are the same as the `dollars` and the `cents` of the parameter object. Otherwise, it returns `false`.
2. Write and document a `toString` method. This method will return a `String` that looks like currency, including the dollar sign. Remember that if you have less than 10 cents, you will need to put a 0 before printing the `cents` so that it appears correctly with 2 decimal places.
3. Compile, debug, and test by running the `MoneyDemo` program. You should get the following output:
   ```
   The current amount is $500.00
   Adding $10.02 gives $510.02
   Subtracting $10.88 gives $499.14
   $10.02 equals $10.02
   $10.88 does not equal $10.02
   ```

Task #3 Passing and Returning Objects (6 pts)

1. Create the `CreditCard` class according to the UML diagram above. It should have data fields that include an `owner` of type `Person`, a `balance` of type `Money`, and a `creditLimit` of type `Money`.
2. It should have a **constructor** that has two parameters, a reference to a `Person` object to initialize the `owner` and a reference to a `Money` object to initialize the `creditLimit`. The `balance` can be initialized to a `Money` object with a value of zero. Remember you are passing in objects (passed by reference), so you are passing the memory address of an object. If you want your `CreditCard` to have its own `creditLimit` and `balance`, you should create a new object of each using the copy constructor in the `Money` class.
3. It should have accessor methods to get the `balance` and the `creditLimit`. Since these are `Money` objects (passed by reference), we don't want to create a security issue by passing out addresses to components in our `CreditCard` class, so we must return a new object with the same values. Again, use the copy constructor to create a new object of type `Money` that can be returned.
4. It should have an accessor method to get the information about the `owner`, but in the form of a `String` that can be printed out. This can be done by calling the `toString` method for the `owner` (an instance of the `Person` class).
5. It should have a method that will charge to the `CreditCard` by adding the `amount` passed in the parameter to the `balance`, but only if it will not exceed the `creditLimit`. If the `creditLimit` will be exceeded, the `amount` should not be added, and an error message can be printed to the console.
6. It should have a method that will make a payment on the `CreditCard` by subtracting the `amount` passed in the parameter from the `balance`.
7. Compile, debug, and test it out completely by running the `CreditCardDemo` program.
8. You should get the output:

```
Diane Christie, 237J Harvey Hall, Menomonie, WI 54751
Balance: $0.00
Credit Limit: $1000.00

Attempting to charge $200.00 Charge: $200.00
Balance: $200.00

Attempting to charge $10.02 Charge: $10.02
Balance: $210.02

Attempting to pay $25.00 Payment: $25.00
Balance: $185.02

Attempting to charge $990.00 Exceeds credit limit
Balance: $185.02
```