CS6311
Lab 05

Accruing Interest

Objectives:

- Write Java statements to implement the counting looping structure
- Write Java statements to implement the conditional looping structure
- Practice introductory Defensive Programming by enforcing preconditions

Overview:

This week the text introduces counting and conditional looping structures. Basically put, we'll use a counting loop (for) when we know *at design time* the exact number of iterations the loop body should run. By contrast if the number of iterations of the loop body is determined *at run time*, then we'll use a conditional loop (either while or do/while). Please know that part of your grade will be based on which loop structure you choose to implement – be sure you know the difference and when each is appropriate.

We're going to practice writing both types of loops in our exercise as we attempt to build a table of monthly balances for a savings account that accrues interest. We'll conditional loops to force our user to enter a number within a specified range before moving on to use the counted loop to build the table.

To make things go a little quicker, I have provided a Starter File on Moodle, which includes the *complete* Account class (you won't need to write any code inside this one). This will allow you to focus on the loops that we're learning about this week.

The Problem Statement - `BalanceBuilder`
The class you will develop is named `BalanceBuilder`, which will keep track of the `Account` object under consideration, number of years calculated in the results table, and the actual table itself. In order to allow this object to build the table, we'll define methods to:

- Set the number of years
- Set the Account to be used
- Get the number of years
- Build the results table
- Get the actual table

To better organize our code, we'll provide a set of private helper methods to:

- Build the table's heading
- Build a row for a particular year
- Add the monthly values within a single year

Be sure to read through all items and the existing code before you begin. If you 'jump right in' on this exercise, you will likely miss the overall goal of the code and get lost quickly.

1. Download the Zipped started file from Moodle and extract its contents to a location where you'll be able to find it later.
2. Use Eclipse to open this Java project (if you've forgotten how to do this, refer to the earlier homework that describes this process). Rename the Project `*YourLastName*Lab05`
3. Inside the Starter, you'll find the shell of the `BalanceBuilder` class. Go through each of the class members present and read the existing code and comments. There are a number of // TODO comments throughout the different classes giving you tips on what needs to be done. Go through the code, **remove** the existing //TODO comments and **replace** each one with one or more Java statements that will perform the task. Pay close attention to existing code and comments as they may give you a hint as to how to model your code.
4. Create a package named `edu.westga.cs6311.interest.view`. Add a new class named `InterestView` to this package. This class should declare *only* the following instance variables:
   - A `BalanceBuilder` to be used to build the results table
   - A `Scanner` object to be used for user input
5. Define a constructor that accepts no parameters and instantiates the `BalanceBuilder` and `Scanner` objects.
6. Define a method named `run`, which is a void method that accepts no parameters. This method will serve to direct the flow of the application by calling on the individual helper methods (the way a conductor directs an orchestra). As you add these new methods, be sure to include code in run to call the method, as appropriate.
7. Define a private helper method named `inputInitialBalance`, which is a method that accepts no parameters and returns a double value. After displaying an appropriate prompt to the user, use the `Scanner` object's `nextLine` method to read in the value and convert it to the appropriate data type. Be sure to include code so that if the user's input is outside the acceptable range (at least 0 for the balance), then continue to prompt them for another value until it is within this range. Only when the user finally enters an acceptable value should this method return the value.

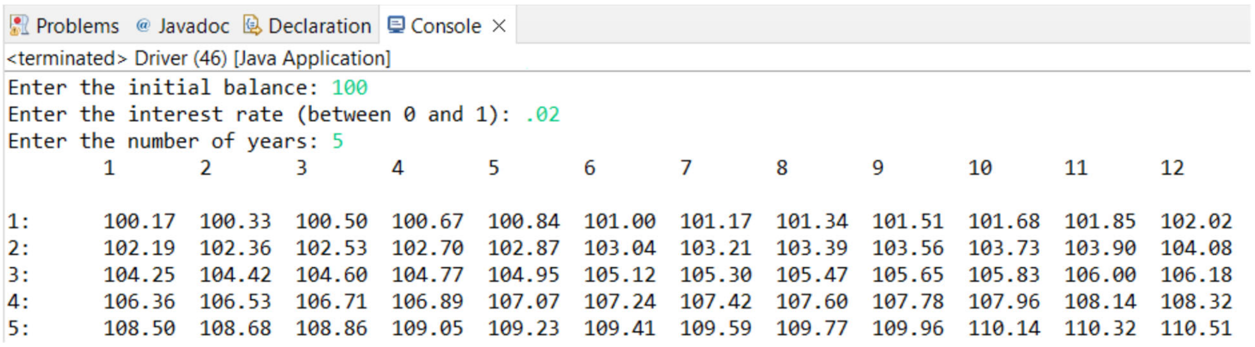
   NOTE: We are still assuming that the user will enter the appropriate data type – we just don't know what actual value they might enter. In the FAQ's on Moodle, this is what we called a 'semi-intelligent user'.
8. Repeat the same process for methods `inputRate` and `inputNumberOfYears`. Note that a rate must be between 0 and 1, inclusive in order for it to be valid. The number of years must be at least 1.

9. (HINT: You should *already* have code inside of run that's calling the other helper methods to get the input – that's what we mean by 'directing the flow of the application')
   Include code inside of `run` to:
   a. Use the user's input values to create an `Account` object and add that object to the `BalanceBuilder`.
   b. Use the user's number of years to set the `BalanceBuilder`'s number of years.
   c. Use the `BalanceBuilder` object to have the results table built.
   d. Write code to display to the console the value returned from the `BalanceBuilder`'s `getResults` method.

Submitting:

When you are finished, be sure that there are no Checkstyle or any other issues listed in the Problems tab. Make sure all of the class files are saved, close Eclipse, and Zip the folder holding your Eclipse project using 7-Zip with a .zip file extension. Give your file the name 6311*YourLastName*Lab05.zip Upload the Zip file to the appropriate link in Moodle.

Sample Output:



```
<terminated> Driver (46) [Java Application]
Enter the initial balance: 100
Enter the interest rate (between 0 and 1): .02
Enter the number of years: 5
        1       2       3       4       5       6       7       8       9       10      11      12

1:      100.17  100.33  100.50  100.67  100.84  101.00  101.17  101.34  101.51  101.68  101.85  102.02
2:      102.19  102.36  102.53  102.70  102.87  103.04  103.21  103.39  103.56  103.73  103.90  104.08
3:      104.25  104.42  104.60  104.77  104.95  105.12  105.30  105.47  105.65  105.83  106.00  106.18
4:      106.36  106.53  106.71  106.89  107.07  107.24  107.42  107.60  107.78  107.96  108.14  108.32
5:      108.50  108.68  108.86  109.05  109.23  109.41  109.59  109.77  109.96  110.14  110.32  110.51
```

Please know that we will only be testing your application using input that will always generate a balance of less than $10,000 (otherwise it would mess up the formatting of this table)