HW02 Problem Set

CS-3020

Assignment
Exercises 5.0, 6.0 and Deitel and Deitel Exercise 7.36

HW02-1: (Exercise 5.0 - Instructor provided)

Implement a class named **`RevolvingLoan`** that represents a somewhat simplified revolving loan account such as a credit card. The end goal of this assignment is to be able to determine how long it would take to pay a purchase off, and how much the final cost would be, if only the minimum payment were made each month. To do this, our loan object has to be able to process purchases and payments and close each statement period by adding interest charges and telling us what the minimum payment is.

In addition to the constructor, in which we establish the terms for the account, we need a public interface that includes the ability to make purchases, close a statement period, get the minimum payment, accept payments, and provide the balance.

The public methods are thus:

`RevolvingLoan(decimal rate, decimal pmtPct, decimal pmtDlr)`

> **`rate`**: Annualized interest rate as a percentage (19.8% is enterest as 19.8).
> **`pmtPct`**: The fraction of the balance, as a percentage, this is due each month.
> **`pmtDlr`**: The floor of the minimum payment unless the balance is below this.

A common method of calculating the minimum payment involves three steps. First, the minimum payment is a specified fraction of the statement balance. This is typically between 1% and 2%, so an account with a $247 balance and a 1% minimum would have a minimum payment of $2.47. If this minimum is less than the minimum dollar payment, then the minimum payment is increased to the minimum dollar amount. Finally, if the minimum payment is greater than the account balance, the minimum payment is set equal to the account balance.

`void Purchase(decimal amount)`

> **`amount`**: Dollar amount of a purchase.

The amount of the purchase is added to the account balance.

`void Payment(decimal amount)`

> **`amount`**: Dollar amount of a payment.

The amount of the payment is subtracted from the account balance.

`decimal Close()`

This method closes a statement period by first calculating the interest charge for the period (defined here as 1/12 of a year), adding that to the balance, and then calculating and returning the minimum payment that is due. For simplicity, all purchases and payments are assumed to have occurred at the very start of the period being closed.

`decimal Balance()`

This method returns the current balance of the account.

`decimal MinimumPayment()`

This method returns the minimum payment for the most recently closed statement period.

You may (and should) have additional private methods to perform other tasks, such as calculating the minimum payment or calculating the interest charges. Some of these may benefit from further, more narrowly defined, helper methods.

You should use the decimal data type for nearly all of your variables.

Your dollar computations should be rounded to the nearest cent, rounding 0.5 cent upward. There is a math library method (**`Math.Floor()`**) that takes a **`double`** and returns an **`int`** that is the integer part of the value passed, truncating any fractional part no matter how big or small. To use this with **`decimal`** variables you will need to typecast the decimal variable to a **`double`**.

The test class, RevolvingLoanTest, is provided for you with two tests built in.

You are to write a third test that determines how long it would take, display as years and months, for someone to pay off a $5000 purchase made on a card having an interest rate of 29.8%, a monthly minimum payment of 3% of the balance with a floor of $15. Instead of returning a pass/fail result to the main program, this test should print out a line similar to the following:

```
Paying off $1000 will take 24 years and 0 months and cost $3717.14.
```

This should be the output if you use the scenario in Test2()

Write a fourth test in which the card holder makes payments equal to twice the minimum payment (except for the last one which should bring the balance exactly to zero).

HW02-2: (Exercise 6.0 – Instructor provided)

Implement a class named **`Mortgage`** which is very similar to the **`RevolvingLoan`** class from the previous exercise, in fact the public interface is nearly identical, though the details of some of the methods are slightly different.

The public methods are thus:

`Mortgage(decimal principal, decimal rate, decimal term)`

`rate`: Annualized interest rate as a percentage (19.8% is enterest as 19.8).
`principal`: The initial loan amount.
`term`: The number of years the loan is amortized over.

`void Payment(decimal amount)`

`amount`: Dollar amount of a payment.

The amount of the payment is subtracted from the account balance.

`decimal Close()`

This method closes a statement period by first calculating the interest charge for the period (defined here as 1/12 of a year) and adding that to the balance. For simplicity, all payments are assumed to have occurred at the very start of the period being closed. The function returns the amount of interest that was charged.

`decimal Balance()`

This method returns the current principle balance of the account.

`decimal MinimumPayment()`

This method returns the minimum payment associated with this mortgage.

The minimum payment (for principal and interest) on a mortgage is the amount that will come closest to paying of the loan at the end of the term (assuming all payments are made on time and for the minimum amount due). This is almost never an exact outcome and the final payment, called the residual, is either slightly more or slightly less than the calculated minimum payment. In general, the minimum payment is calculated at the beginning of the loan and does not change even if additional principal payments are made. That is the case that will be assumed here.

`decimal ResidualPayment()`

This method returns the amount of the residual (last) payment associated with this mortgage.

Write a test class (**`MortgageTest`**) that uses the following tests (each test should print results to the console).

Test1(): For a $200,000 30-year mortgage at 4.5%, what are the:

a) Minimum and residual payments?
b) Total cost of all payments if paid on schedule?

Test2(): For a $200,000 15-year mortgage at 4.5%, what are the:

a) Minimum and residual payments?
b) Total cost of all payments if paid on schedule?

Test3(): For the loan in Test1(), if each month an extra amount is added to the payment equal to the prior month's interest (the first month just the minimum payment is made), what will be

a) The fractional increase from the first payment to the second (as a percentage)?
b) The length of time required to pay off the loan (years and months)?
c) The total cost of all payments made?
d) The net savings compared to paying the loan off on schedule?

Test4(): For the loan in Test1(), if each month an extra amount is added to the payment equal to the prior month's principle reduction, what will be

a) The fractional increase from the first payment to the second (as a percentage)?
b) The length of time required to pay off the loan (years and months)?
c) The total cost of all payments made?
d) The net savings compared to paying the loan off on schedule?

Test5(): For the loan in Test1(), if each month the payment is twice the minimum payment, what will be

a) The length of time required to pay off the loan (years and months)?
b) The total cost of all payments made?
c) The net savings compared to paying the loan off on schedule?

<u>**HW02-3**: (Deitel & Deitel Exercise 7.36)</u>

7.36 *(Towers of Hanoi)* Every budding computer scientist must grapple with certain classic problems, and the *Towers of Hanoi* (see Fig. 7.17) is one of the most famous. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there's little incentive for us to facilitate their efforts.

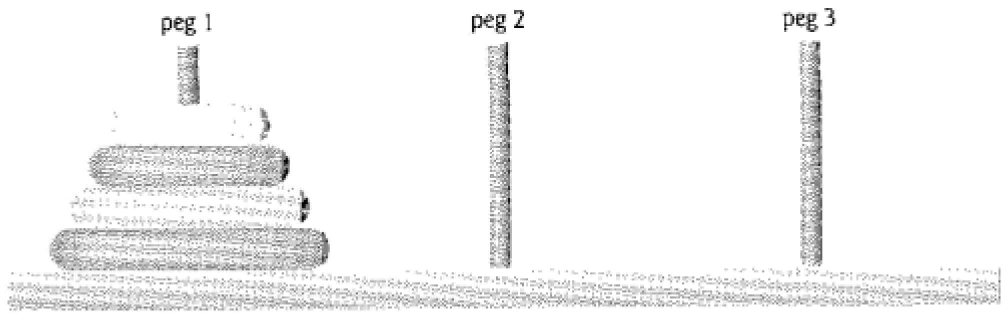

Fig. 7.17 | The Towers of Hanoi for the case with four disks.

Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will display the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we'd rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving *n* disks can be viewed in terms of moving only *n* – 1 disks (hence the recursion) as follows:

a) Move *n* – 1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
b) Move the last disk (the largest) from peg 1 to peg 3.
c) Move the *n* – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving *n* = 1 disk (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area.

Write an app to solve the Towers of Hanoi problem. Allow the user to enter the number of disks. Use a recursive Tower method with four parameters:

a) the number of disks to be moved,
b) the peg on which these disks are initially threaded,
c) the peg to which this stack of disks is to be moved, and
d) the peg to be used as a temporary holding area.

Your app should display the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your app should display the following series of moves:

```
1 --> 3 (This notation means "Move one disk from peg 1 to peg 3.")
1 --> 2
```

The scan for the second part of the assignment is very poor, but should be readable with a bit of effort. However, the bottom line of the first part has been cut off. It reads, starting and ending with the adjacent words on the adjacent lines, "ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind."

Grading Rubric

Each problem is worth 10 pts (score will be recorded as a percentage of that amount)

10% Properly submitted
10% Properly named
20% Adequate comments
10% Runs
20% Produces correct output
30% Effort evidenced by the submitted work