Scott Scherzer

10/26/2020

Computing Future Investment Value

**Problem Description:**

Write a method that computes future investment value at a given interest rate for a specified number of years. We can also assume this model will not have to handle zero or negative numbers since we are not expecting a zero percent return or a zero number of years. Also, the model requires capital to operate so we can also assume we will not have to test zero or negative numbers for principal.

**Analysis**

**Describe the problem:**

The goal of this project is to create a method to calculate the compounded returns of an investment made or simulated by the user. The formula used to calculate this is:

**P \* (1 + R / 12) years\*12**

**Part I: Program and test a functioning method**

To begin programming my method I am going to take my variables from the method header. I plugged in the proper formulas with print statements to see what would be returned before I proceeded. All was going accordingly after executing

//return = investmentAmount × (1 + monthlyInterestRate) ^ numberOfYears×12

public static double futureInvestmentValue(double investmentAmount,

double monthlyInterestRate,

int years){

double future\_Investment\_Value = investmentAmount \*

Math.pow((1 + monthlyInterestRate), years\*12);

return future\_Investment\_Value;

}

And calling this method by:

public static void main(String[] args) {

double i = futureInvestmentValue(10000,

0.05/12, 5);

System.out.println("i: " + i);

}

My output after calling this method was 12833.586785035119 so I knew my method functioned properly. My steps for designing this method were to copy the formula given, which was

Return = investmentAmount × (1 + monthlyInterestRate) ^ numberOfYears×12

To raise to the power of numberOfYears\*12 I used the method *Math.pow*

All was running fine at this point.

**Part II: Clean up method and add variables to prepare it for a loop**

If we are going to simulate 30 years of this investment compounding it would make much more sense to set up a loop rather than copy and paste many times. Also a loop adds robustness to the program because a different output can be received if the user wants to switch how many years they want to let their investment perform.

I first ran into trouble when I created variables for the input

public static void main(String[] args) {

double principal = 10000;

double interest = 0.05;

int t = 5; //t = time

double i = futureInvestmentValue(principal,

interest, t);

System.out.println("i: " + i);

}

My output here was i: 2.9753582081680648E16

I then realized my mistake when I read the original formula and saw it was monthly interest, so with a quick tweak to monthlyinterestRate in my method all was well once again.

//return = investmentAmount × (1 + monthlyInterestRate) ^ numberOfYears×12

public static double futureInvestmentValue(double investmentAmount,

double monthlyInterestRate,

int years){

double future\_Investment\_Value = investmentAmount \*

Math.pow((1 + (monthlyInterestRate/12)), years\*12);

return future\_Investment\_Value;

}

**Part III: Create loop**

Now I had to begin my loop to match the example output. Now the only change that had to be made was the amount invested, since we are compounding returns. Now since we are showing the yearly returns we are going to have to set years to 1 while we are executing our loop.

The logic I am going to use for my loop is

i = futureInvestmentValue( principal, interest, t);

principal = i

This would count as one repetition. We want to repeat this for as many years we would like to compound the capital for.

Setting up the loop there is an immediate issue.

double principal = 1000;

double interest = 0.09;

**int t = 30; //t = time**

for(int years = 1; t <= years ; years++){

double i = futureInvestmentValue(principal,

interest, t);

System.out.println(years + " " + i);

principal = i;

}

If we let t = 30 for this loop, the loop would assume a 30 year investment period for each loop. The output for this would be

21 3.408900988966207E27

22 5.0215075513874544E28

23 7.39696992381353E29

24 1.0896162854257583E31

25 1.605067563736869E32

26 2.3643569930249012E33

27 3.4828340667796297E34

28 5.1304152344615844E35

29 7.557397215404273E36

30 1.113248149735665E38

11 7.086259627890763E15

12 1.04384686876273104E17

13 1.53764657611068134E18

14 2.2650419939730887E19

15 3.336537351410389E20

16 4.914911744231846E21

17 7.239948158643267E22

18 1.0664860747776111E24

19 1.5709954308674198E25

20 2.3141667783341243E26

1 14730.576123040439

2 216989.87291668908

3 3196385.8409281597

4 4.7084604948400885E7

5 6.935833574157058E8

6 1.021688244408602E10

7 1.5050056458276456E11

8 2.216960023146977E12

9 3.265709838270404E13

10 4.810578736840427E14

To fix this, we simply fix our method to 1 rather than t, basically allowing t = 1. The resulting output for this fix would give us the correct output.

1 1093.8068976709837

2 1196.413529392622

3 1308.6453709165362

4 1431.4053333137103

5 1565.68102694157

6 1712.552706821279

7 1873.201963346229

8 2048.9212282389344

9 2241.1241722322507

10 2451.3570781248095

21 6572.851386618246

22 7189.430184049327

23 7863.848325637125

24 8601.531540820304

25 9408.414529883774

26 10290.988708934778

27 11256.354433687073

28 12312.278122196281

29 13467.25473610184

30 14730.576123040419

11 2681.3112807075054

12 2932.836773640889

13 3207.9570927515183

14 3508.885595484167

15 3838.0432674789395

16 4198.078199528145

17 4591.88689160607

18 5022.637555363693

19 5493.795602558135

20 6009.151524472607

To further clean our data, we need to round our output to dollar amounts which would be two decimal places. The formula we will use for this is:

//round to dollars

i = Math.round(i \* 100.0)/100.0;

Our output is now:

|  |  |  |
| --- | --- | --- |
| 1 1093.81 | 11 2681.31 | 21 6572.87 |
| 2 1196.42 | 12 2932.84 | 22 7189.45 |
| 3 1308.65 | 13 3207.96 | 23 7863.87 |
| 4 1431.41 | 14 3508.89 | 24 8601.56 |
| 5 1565.69 | 15 3838.05 | 25 9408.45 |
| 6 1712.56 | 16 4198.09 | 26 10291.03 |
| 7 1873.21 | 17 4591.9 | 27 11256.4 |
| 8 2048.93 | 18 5022.65 | 28 12312.33 |
| 9 2241.13 | 19 5493.81 | 29 13467.31 |
| 10 2451.36 | 20 6009.17 | 30 14730.64 |

However, the issue here is this does not match the example output. To fix this I will only print the rounded number, but I will not use it in any of the model calculations.

The code correction I made is:

//round to dollars

double answer = Math.round(i \* 100.0)/100.0;

System.out.println(years + " " + answer);

Now there is no rounding in *i* and we have allocated memory to a variable only for printing the result for that year, which will get replaced every “year”. Here is the updated output:

|  |  |  |
| --- | --- | --- |
| 1 1093.81 | 11 2681.31 | 21 6572.85 |
| 2 1196.41 | 12 2932.84 | 22 7189.43 |
| 3 1308.65 | 13 3207.96 | 23 7863.85 |
| 4 1431.41 | 14 3508.89 | 24 8601.53 |
| 5 1565.68 | 15 3838.04 | 25 9408.41 |
| 6 1712.55 | 16 4198.08 | 26 10290.99 |
| 7 1873.2 | 17 4591.89 | 27 11256.35 |
| 8 2048.92 | 18 5022.64 | 28 12312.28 |
| 9 2241.12 | 19 5493.8 | 29 13467.25 |
| 10 2451.36 | 20 6009.15 | 30 14730.58 |

**Testing: Functionality**

Add print statements to confirm there are no arithmetic errors.

If we calculate by hand the initial example given, with the parameters

**principal = 1000**

**interest = 0.09**

**t = 30**

P \* (1 + interest/12)t\*12

**Step 1: Inside of parenthesis**

1 + interest/12 1 + 0.09/12

1 + 0.09/12 =

1 + 0.0075 =

1.0075

**Step 2: Raise to power**

1.0075t\*12  1.007530\*12

1.007530\*12 =

1.0075360 =

14.73057612304044

**Step 3: Multiply by principal**

P \* 14.73057612304044 1000 \* 14.73057612304044

1000 \* 14.73057612304044 =

**14730.576123040439**

Since this is the same output we received in the first example, we now know our method works arithmetically with the numbers provided. Since we can also assume this model will not have to handle zero or negative integers, we will not have to test any special variables as we would with other model testing under different circumstances.