

Homework 2

Note: In all the homework programs, unless we say otherwise, you don't have to write code that checks the inputs. The programs that you write can assume that the inputs are valid, as described in the program descriptions.

1. Time Calculations (20 Points)

The TimeCalc program gets a *time* input which is given using the 24-hour *hh:mm* format, and a number of *minutesToAdd* input. The program computes *time + minutesToAdd*, and prints the resulting time using the 24-hour *hh:mm* format. For example, the time 10:30 + 4055 minutes happens to be 06:05. Here is *one way* to calculate this new time value:

Total minutes = (10 hours * 60 minutes) + 30 minutes + 4055 minutes = 4685 minutes;

Total hours = 4685 / 60 = 78;

New hours = 78 % 24 = 6; // The *remainder operator* % is also known as *modulo*

New minutes = 4685 - (78 * 60) = 5;

Therefore the answer is 06:05.

Explanation:

- If *x* and *y* are both int values, the expression *x / y* evaluates to the *integer division* of *x* by *y*
- If *x* and *y* are both int values, the expression *x % y* evaluates to the *remainder* of the integer division *x / y*

The TimeCalc program gets the *hh:mm* and *minutesToAdd* values as two command-line arguments. Here are some program run examples:

```
% java TimeCalc1 10:20 30
10:50

% java TimeCalc1 10:20 40
11:00

% java TimeCalc1 10:20 500
18:40

% java TimeCalc1 11:30 350
17:20

% java TimeCalc1 09:25 700
21:05

% java TimeCalc1 10:30 4055
06:05

% java TimeCalc1 22:07 12345
11:52

% java TimeCalc 23:59 10
00:09
```

Input rules:

The *hh* and *mm* inputs are supplied as a single command-line argument: A string consisting of two-digit characters, followed by the character ':', followed by two-digit characters;

Both *hh* and *mm* use a leading '0', as needed. For example, the time "5 hours and 8 minutes" is represented as the string "05:08";

Midnight is represented as "00:00".

Once converted to int values, the *hh* and *mm* values must be, respectively, between 0 and 23 and between 0 and 59, inclusive.

Output rules: Exactly the same as the input rules. (Note that the output rules of this program are different than the output rules of the TimeFormat program from Homework 1).

Implementation tip: Start by writing a program that gets the inputs and prints them – that's all. For example:

```
% java TimeCalc1 09:30 1475
Hours: 9
Minutes: 30
Minutes to add: 1475
```

Next, calculate the new hours and minutes, and print them also, as int values:

```
% java TimeCalc1 09:30 1475
Hours: 9
Minutes: 30
Minutes to add: 1475
New hours: 10
New minutes: 5
```

Once you get the calculation right, eliminate the above outputs by commenting out the Java statements that generate them. Finally, write the final version that prints the result using the output rules:

```
% java TimeCalc1 09:30 1475
10:05
```

2. How random is Math.random? (15 Points)



(Dilbert Comic, by Scott Adams, Dilbert © 2018, Andrews McMeel Syndication)

How “good” is Java’s random number generator function `Math.random`? One possible test is as follows: Call `Math.random` N times, where N is a non-negative integer, and compare the number of times the function returned a number greater than 0.5 to the number of times it returned a number less than or equal to 0.5. If `Math.random` generates values between 0 and 1 randomly, we would expect the ratio between these two numbers to converge to 1 as N becomes larger.

The `TestRandom` program performs this test. Here are some program run examples:

```
% java TestRandom 100
> 0.5: 47 times
<= 0.5: 53 times
Ratio: 0.8867924528301887

% java TestRandom 10000
> 0.5: 4986 times
<= 0.5: 5014 times
Ratio: 0.994415636218588

% java TestRandom 3
> 0.5: 3 times
<= 0.5: 0 times
```

The last example illustrates an important *edge case*: It may be that `Math.random()` will always generate values greater than 0.5 (or less than or equal to 0.5), just like shown in the Dilbert cartoon... In that case, we don’t want to calculate the ratio, since this calculation will result with a run-time error (do you see why?). Your program must take care of this special case.

Complete the program’s implementation, and test it on various N values. Make sure that the output that your program prints looks exactly like the examples shown above (with different values, of course).

3. Cheers (15 Points)

The `Cheers` program gets two inputs: a string and a non-negative integer value, and prints a crowd cheering output. Here is an example of the program’s execution:

```
% java Cheers RUNI 5
Give me an R: R!
Give me a U: U!
Give me an N: N!
Give me an I: I!
What does that spell?
RUNI!!!
RUNI!!!
RUNI!!!
RUNI!!!
RUNI!!!
```

If the inputted letter is one of the letters `{'A', 'E', 'F', 'H', 'I', 'L', 'M', 'N', 'O', 'R', 'S', 'X'}`, the program prints "an" instead of "a". Tip: This condition can be tested using a function call of the form `string.indexOf(ch)`. You have to figure out what you need to place instead of `string` and `ch`.

Implementation plan: We suggest writing the program in stages. Start by writing a program that gets the two command-line arguments and uses a loop to print only the "Give me..." lines, without the "a" / "an" touch. Then add the "a" / "an" touch. Finally, use another loop to generate the remaining lines. Assume that the input string contains only letters, and no spaces. The inputted letters can be either lowercase, or uppercase. The outputted letters must be uppercase. Tip: Convert the inputted text to upper case just after inputting it.

4. Calculating π (20 Points)

Consider the following sum:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \dots$$

The mathematicians Gregory and Leibnitz discovered that as the number of terms in this sum increases, the sum converges to $\pi / 4$. The `CalcPi` program uses this approximation algorithm to calculate π . Here are two examples of the program's execution:

```
% java CalcPi 10
pi according to Java: 3.141592653589793
pi, approximated:    3.0418396189294032

% java CalcPi 1000
pi according to Java: 3.141592653589793
pi, approximated:    3.140592653839794
```

The program has one non-negative integer input: the number of terms that should be used in the calculation. The benchmark value is obtained by printing the Java library constant `Math.PI`.

5. Collatz Conjecture (30 Points)

A *hailstone sequence* is created as follows: Start with some non-negative integer, let's call it *seed*, and obtain a sequence of numbers as follows: (1) If the current number is even, divide it by 2; otherwise, multiply it by 3 and add 1; (2) Repeat.

For example, here are the first 8 hailstone sequences (the first number in each sequence is the seed):

```
1, 4, 2, 1, 4, 2, 1, ...
2, 1, 4, 2, 1, 4, 2, 1, ...
3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
4, 2, 1, 4, 2, 1, ...
5, 16, 8, 4, 2, 1, 4, 2, 1, ...
6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...
```

It appears, from these examples, that hailstone sequences tend to reach the number 1. Indeed, in 1937 the mathematician Lothar Collatz argued that for any seed N , a hailstone sequence will eventually reach 1. The Collatz conjecture seems to be true, but up until today no-one succeeded proving it (and that's why it's called a "conjecture" rather than a "theorem"). It is, quite simply, one of the most tantalizing unsolved problems in mathematics

The Collatz program is designed to test the Collatz conjecture for all seeds between 1 and N . The program takes two inputs: the highest seed N , and a string which we call *mode*. This string can be either “v” (*verbose*) or “c” (*concise*). In *verbose* mode, the program prints all the sequences from $seed = 1$ to $seed = N$. For each sequence, the program prints all the values until the sequence reaches 1. Next, the program prints the number of steps the series took to reach 1. Finally, the program prints a summary line. In *concise* mode, the program prints only the summary line.

If the program terminates and prints the summary line, it verifies the Collatz conjecture up to N . Here are two example program runs:

```
% java Collatz 7 v
1 4 2 1 (4)
2 1 (2)
3 10 5 16 8 4 2 1 (8)
4 2 1 (3)
5 16 8 4 2 1 (6)
6 3 10 5 16 8 4 2 1 (9)
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 (17)

Every one of the first 7 hailstone sequences reached 1.

% java Collatz 100 c

Every one of the first 100 hailstone sequences reached 1.
```

Proposed implementation: Start by writing a loop that prints one hailstone sequence only, for a given seed value (N). Test the program for several such seed values. Next, nest this loop inside an outer loop that varies the seed from 1 to N . Finally, handle the verbose/concise requirement.

String comparison: Strings should be compared for equality using Java’s boolean method `str1.equals(str2)`. For example, if `mood` is a variable of type `String`, and you wish to check if it equals “happy”, use the method call `mood.equals("happy")`.

Submission guidelines

Before submitting your work for grading, make sure that your code is written according to our [Java Coding Style Guidelines](#).

Submit the following files only on GitHub:

- TimeCalc.java
- TestRandom.java
- Cheers.java
- CalcPi.java
- Collatz.java

Submission deadline: 13.11.2024, 23:55

A Note on Using ChatGPT

When writing a program like TimeCalc, it makes sense to consult with ChatGPT about time arithmetic. What follows is an example of one such a session, which resulted in an important general lesson about using AI agents. The prompts are shown in grey boxes. Everything else was generated by ChatGPT:

Calculate the time 22:07 + 12345 minutes



To calculate 22 : 07 + 12345 minutes:

1. **Convert 22:07 to minutes:**

$$22 \times 60 + 7 = 1327 \text{ minutes.}$$

2. **Add 12345 minutes:**

$$1327 + 12345 = 13672 \text{ total minutes.}$$

3. **Convert 13672 minutes back to hours and minutes:**

- $13672 \div 60 = 227$ hours with a remainder of 32 minutes.
- 227 hours is equivalent to $227 \bmod 24 = 11$ (since every 24 hours cycles back to 0).

This gives us **11:32** in 24-hour format.

So, the result of 22 : 07 + 12345 minutes is **11:32**.

Notice how the AI answer is broken neatly into well-structured and well-explained steps (in the context of AI chatbots, this output is sometimes referred to as “chain of thought”). The only problem with this neat and clear chain of thought is that it is wrong. In particular, the first calculation in step 3 is incorrect. How did we figure it out? Well, we actually did the calculation ourselves, using paper and pencil, and got the time 11:52, and not 11:32. Here is the rest of the dialog:

Dont you have an error here:

$$13672 \div 60 = 227 \text{ hours with a remainder of } 32 \text{ minutes.}$$



You're absolutely correct; I made an error in the division. Let's correct it.

To calculate $22 : 07 + 12345$ minutes accurately:

1. Convert 22:07 to minutes:

$$22 \times 60 + 7 = 1327 \text{ minutes}$$

2. Add 12345 minutes:

$$1327 + 12345 = 13672 \text{ total minutes}$$

3. Convert 13672 minutes back to hours and minutes:

- $13672 \div 60 = 227$ hours with a remainder of 52 minutes, not 32.
- 227 hours is equivalent to $227 \bmod 24 = 11$ (since every 24 hours cycles back to 0).

So, the correct result is 11:52 in 24-hour format.

Thank you for catching that mistake! The result of $22 : 07 + 12345$ minutes is indeed 11:52.

The lesson is clear: Never take for granted what an AI (or human) consultant says. Be skeptical, and check the results on your own. Here is the rest of the dialog:

I understand that you sometime make errors. Does this mean that I have to double-check every calculation that you make?



I strive for accuracy in every response, but you're right to be cautious, especially when it comes to critical or detailed calculations. While I aim to provide reliable answers, occasional errors can occur, especially with complex or lengthy steps.

If an answer seems off or if the calculation is particularly important, double-checking it is a good practice. Additionally, you can always ask me to review my steps, as I'm here to clarify and correct anything if needed. In cases where accuracy is essential, running the calculation independently can offer extra peace of mind.

We could not have stated it better. So, *caveat emptor*!

For those of you who don't speak Latin: *caveat emptor* is a common legal term, implying that the buyer must assume the risk in a transaction, meaning that the buyer is responsible for checking the quality and suitability of the product before a purchase is made. One reason why Latin is considered a great language is due to the fact that it can express such a notion in two words.

By the way, the session shown above is not necessarily replicable. The fact that ChatGPT erred in this particular calculation does not mean that it will err on it again. Or, it may err in a different way.