Overview of the Prettifier Test

Approach

The approach I used is to first transform the integer part of the input number into a string version with "," as separator for thousands. In this way its easy to check how many thousands (or powers of 1000) are in any given number (to figure out the unit symbol to append at the end of the pretty number), just count the number of commas.

I was about to use this formula for counting the thousands in the input number:

```
Long thousandCnt = (long)floor(floor(log10(abs(intNumber)))/3);
```

which works! but not in java because of its long lack of precision. And BigDecimal doesn't have a log function builtin so I just simplified the computation by using this less efficient but elegant java 8 code:

```
Long thousandCnt = numAstr.chars().filter(c -> c == ',').count();
```

Once I have the string version of the integer part of the input number (I didn't want to use a builtin api for this, no cheating) I need only extract the stuff until the first comma. For example, for *12,999,000,000* I extract *12*. Then following that example, I need to append the *9* on the next thousand to get a truncated version: *12.9*.

And last, add the unit: *12.9B* in this case (billion) Somewhere in the code I handle numbers that have fraction part **only if the number in question is less than 1M**. For numbers greater than 1M its irrelevant as I never show that info in the pretty version. For this I extract the fraction and truncate it to 1 digit.

The code also handles negative numbers with no problem. There is a special case when the number is negative and greater than -1. For example *-0.5*. In that cases I check that and append a - to the pretty version. For all other negative numbers the negative sign is handled smoothly.

Assumptions

The code checks for limit cases like *NaN* (Not a number) and *infinite* values. In those cases I return null to indicate an abnormal scenario. I assume that the code only needs to handle up to +/- trillions, so I put an upper and lower bound of *999,999,999,999,999.9* and *-999,999,999,999,999.9* respectively and check that the input is not outside that range.

I assume that when a thousand is followed by a *0*, then I don't have to add a *.x* to the pretty version. For example, in *1,500,000* the pretty version is *1.5M*, but in *1,050,000* the pretty version is *1M* because the truncation only includes 1 digit and to include *.0* is pointless and not pretty.

When the problem says: *The prettified version should include one number after the decimal when the truncated number is not an integer.* That seems to be redundant, but I'm not 100% sure. This is my point: if the number is not an integer (has a fraction part) or not, anyway I include a *.x* only if *x* is not *0* and I never add fraction information to the pretty format barring the number is less than a million, in which case I print the whole number with the fraction truncated to 1 digit ceiled if the number is positive and floored otherwise. That's what this code does:

```
fraction = fraction.compareTo(BigDecimal.ZERO) == 1 ?
            fraction.setScale(2, BigDecimal.ROUND_CEILING) :
            fraction.setScale(2, BigDecimal.ROUND_FLOOR);
```

I also assume a short scale (US, Canada) for the nomenclature of numbers. For example, a billion in this nomenclature is 10^9 (but in the long scale is 10^{12})

Design Decisions

The code is not large or complex, so I just decided to separate the conversion to string of the input number from the further processing to generate the final pretty version.

I don't like splitting code into methods more than its helpful. In this case, there was no need.

The first part is done in *longToDecString()* method, and the second one in *prettifyNumber()* which has some overloaded versions to accept different types of input. I could had done less processing by not having a separate *longToDecString()* method but I just wanted that when the input number is less than *1M* I could be able to print a pretty version of the number separating the thousands with commas (i.e. *1000* as *"1,000"*).

Also having that *longToDecString()* as a separate functionality becomes handy for other prettify routines.