# Design Document

## Assumptions

* Based on provided “NumberService”, we will not be worried about decimal input
  + Extension: Division should return whole number + remainder, not decimal answer.
* Based on provided “NumberService”, we will not be worried about negative input
* There is not unnecessary left-padding of 0’s in input.
* If provided “NumberService#normalize” throws an exception, then the entire operation is invalid
* “CalculatorService” will only ever be called with normalized input
* Given numbers, or the resulting numbers, may overflow regular Java primitives, which is why input and returns are Strings.
* Given numbers, when normalized, are not larger than the MAX\_INT value in length
* The expected results of operations are not larger than MAX\_INT value in length.
* It is desired to not access Java built-in String-to-int functionality, and thus operations will need to be done digit-by-digit rather than converting blocks of digits at a time.
* Presence of Eclipse settings files and .vscode folder indicates only an additional help to get the project started, not a preference or mandate to use a particular IDE, or both IDEs
* Based on being the “meat” of the issue, the TODOs present, and the only parts originally outlined in this document, the desire is to see an implementation of the basic mathematical operations and to think about scalability, not to create a fully-fleshed out GUI application/build upon the given HTML or Spring design.
* The order of the operands (a then b) is the order of the operation. This is important for subtraction and division. So all operations of interest are a+b, a-b, a\*b, and a/b.

## Algorithms

### Potential Runtime Enhancements

* Detect common right-padding with 0’s in **both** operands, and skip some iterations based on operation
* Each loop iteration of Multiply could be its own thread, which may help practical runtime.

### Potential Additional Functionality

* NumberService#normalize could also call #removeLeftPad for further normalization/sanitizing of input
* Handling of negative inputs, in all operations
* Division to create decimal answers out to some precision level
* Handling of decimal inputs, in all operations

### Add

If one String is “0”, return the other string.

Starting from the very right of both Strings (the ones column), add the numbers, keeping track of potential carry to bring to the next digits (the tens column), and so on, until one Strings has been fully exhausted and no carry remains, or both Strings are exhausted.

If both Strings are exhausted, add any remaining carry at the end.

If only one String is exhausted, prepend the rest of the other String.

Each iteration **prepends** the resulting add to the current String. For better processing, use StringBuilder rather than concatenating Strings continuously.

### Subtract

If a is “0”, and b is not, return negative b. If b is “0”, return a.

We start from the right-most digits of both numbers.

Determine the “true” a and b, where a > b. This logic must examine potentially the entirety of a and b, but the alternative is potentially equally expensive logic for running into special logic for fixing a – b where b > a (another call to subtract near completion of original operation).

From Assumptions, we know neither number has left-padding with 0’s. If one number is longer than the other, that is the larger.

Otherwise, they are the same length, so start from the left-most digit of each, iterating until one number is determined to be the largest.

If they are equal, we can return “0” right away.

From this point forward, “a” refers to the larger of the two numbers, and “b” to the smaller.

When both b (the smaller number) still has a digit to give:

If there is a “subtraction carry”:

If a’s digit is not 0, subtract b’s digit and 1 from a’s digit. Unset “subtraction carry”.

If a’s digit is 0, we know from Assumptions a has another digit to give, subtract b’s digit from 9. Keep “subtraction carry” set.

If there is not a “subtraction carry”, subtract b’s digit from a’s.

If the result is non-negative, prepend to the return.

If the result is negative:

We know a is larger than b, so we know a must have at least one more digit after this one. Set “subtraction carry”. Add 10 to the current result, and prepend to the return.

If a still has digits to give (but b does not):

While there is a “subtraction carry”:

If the current digit of a is 0, prepend 9 to return. Keep “subtraction carry” set.

If this is the final digit of a and it is 1, do not prepend anything. Unset “subtraction carry”.

Otherwise, prepend the digit of a minus 1 to return. Unset “subtraction carry”.

Prepend the remainder of a.

We may have added in some left-padded zeroes, for instance if subtracting 888533 – 888531 = 000002. Remove this left-padding.

If we switched a and b at the beginning, prepend “-“ to the result.

### Multiply

If either a or b is 0, return 0.

If a is 1, return b. If b is 1, return a.

Keep a result to be continuously added to.

For each digit in b, right-pad a temporary version of a with 0’s equal to b’s digits place (ones is 0, tens is 1,…). Now add that temp a b-digit times to the result.

Return the result.

### Divide

If b is 0, return “Division by 0!”

If b is 1, return a.

If a is 0, return “0”.

Track a whole number quotient and a remainder value, both starting at 0.

Determine how many more digits a has than b.

If the digit difference is >1:

add 10 \* (difference – 1) to quotient.

Recursively call divide, with a new b right-padded with (difference – 1) 0’s

Parse the sub-quotient and remainder from this result. Store the remainder as the new a, and add the sub-quotient right-padded with (difference – 1) 0’s to the current quotient. If a remainder does not exist, set the new a to “0”.

Repeat this until remainder is 0, or digit difference is <= 1.

Keep subtracting b from a. Whenever the result is positive or 0, add 1 to the quotient. When it is negative, stop subtracting and use the last positive value of a as the remainder. When it is 0, stop subtracting and use 0 as the remainder.

If 0 is the remainder, return just the quotient.

If 0 is not the remainder, return in the format “quotientRremainder”

## Scalability

I think it would be best to have multiple instances of CalculatorService running, and actually potentially even better to have add, subtract, multiply, and divide as their own microservices. Based upon anticipated usage of each, and knowing that multiply/divide call other services, it could be roughly estimated how many instances of each service should be spun up on a nominal basis. Ideally if traffic spikes more instances should be automatically brought online to handle the load.

NumberService and its methods would likely also get split up into microservices, or as a utility class its functionality could be packaged into each of add, subtract, multiply, and divide. Multiply depends on Add, and Divide depends on Add and Subtract. These dependencies could be packaged in to Multiply/Divide. Such packaging makes heavier services rather than completely microservice architecture, which would mean less messaging overhead but potentially inconsistent implementations of Add/Subtract comparing their own services vs Multiply’s version of the service vs Divide’s version of the services. The same could be said of NumberService utilities, though that seems to be a much more easily argued candidate for packaging within services rather than separating as its own set of services due to runtime being so low and anticipated infrequency of changes compared to the main features.

Some operations could be multithreaded. For instance, each loop iteration in the multiply method could be its own thread, and as threads complete they add their results to the current tally (with appropriate synchronization/locking).

