



## CodeCrunch

[Home](#) | [My Courses](#) | [Browse Tutorials](#) | [Browse Tasks](#) | [Search](#) | [My Submissions](#) | [Logout](#) | Logged in as: **e0959123**

## CS2030 (2310) Exercise #5: Logging

## Tags &amp; Categories

Tags:

Categories:

## Related Tutorials

## Task Content

## Logging

## Topic Coverage

- Generics
- Function
- Lambda expression
- map and flatMap

## Problem Description

In this lab, we are going to write a Log class to handle the context of logging changes to values while they are operated upon.

Let's start with an example. The following recursive function (or method) returns the sum of non-negative integers from 0 to n.

```
int sum(int n) {  
    if (n == 0) {  
        System.out.println("base case returns 0");  
        return 0;  
    } else {  
        int ans = sum(n - 1) + n;  
        System.out.println("adding " + n);  
        return ans;  
    }  
}
```

Notice that debugging statements are added to show how the computation progresses. For instance,

```
jshell> sum(5)  
base case returns 0  
adding 1  
adding 2  
adding 3  
adding 4  
adding 5  
$.. ==> 15
```

Since debugging output is a side-effect, you are to define a generic Log<T> class that encapsulates a value of type T, as well as a String.

A sample logging session is shown below:

```
jshell> sum(5)
$.. ==> Log[15]
hit base case!
adding 1
adding 2
adding 3
adding 4
adding 5
```

## Task

Your task is to write a `Log` class that provides the operations of, equals, map and flatMap. You will also be using `Log` within solutions to classic computation problems. This would allow us to look at the values changes when solving each problem.

## Level 1

Define a `Log<T>` class with two factory methods `of(T t)` and `of(T t, String log)`.

```
jshell> Log.<Integer>of(5)
$.. ==> Log[5]

jshell> Log.<Integer>of(5, "5")
$.. ==> Log[5]
5

jshell> Log.<String> hello = Log.<String>of("Hello")
hello ==> Log[Hello]
```

To ensure that a valid `Log` is created, an `IllegalArgumentException` would be thrown from the `of` method when the argument is null or another `Log`.

```
jshell> try { Log.<Object>of(hello); }
...> catch (Exception e) { System.out.println(e); }
java.lang.IllegalArgumentException: Invalid arguments

jshell> try { Log.<Integer>of(null); }
...> catch (Exception e) { System.out.println(e); }
java.lang.IllegalArgumentException: Invalid arguments

jshell> try { Log.<Integer>of(5, null); }
...> catch (Exception e) { System.out.println(e); }
java.lang.IllegalArgumentException: Invalid arguments
```

*Tip: define the `of(T, String)` method with*

```
static <T> Log<T> of(T value, String log) {
    return Optional.ofNullable...;
}
```

In this exercise, we are not overly concerned that the `of` method would be invoked after the creation of a `Log`.

## Level 2

Include the method `map` that takes in an appropriate `Function` and maps the value in the `Log`. Note that `map` does not add further logging.

```
jshell> Log.<Integer>of(5, "5, add 1").map(x -> x + 1)
$.. ==> Log[6]
5, add 1

jshell> Log.<Integer>of(5, "5, add 1, mul 2").map(x -> x + 1).map(x -> x * 2)
$.. ==> Log[12]
5, add 1, mul 2

jshell> Log.<String>of("five", "five, get length, mul 2").map(x -> x.length()).map(x -> x * 2)
$.. ==> Log[8]
five, get length, mul 2
```

## Level 3

Now define the method flatMap that takes in an appropriate Function so as to add further logging.

```
jshell> Function<Integer, Log<Integer>> f = x -> Log.<Integer>of(x + 1, "add 1")
f ==> $Lambda$...

jshell> Function<Integer, Log<Integer>> g = x -> Log.<Integer>of(x, "mul 2").
...> map(y -> y * 2)
g ==> $Lambda$...

jshell> Log.<Integer>of(5, "5").flatMap(f)
$.. ==> Log[6]
5
add 1

jshell> Log.<Integer>of(5, "5").flatMap(f).flatMap(g)
$.. ==> Log[12]
5
add 1
mul 2
```

## Level 4

Include the overriding equals method that returns true if the argument Log object is the same as this Log, or false otherwise. Two loggers are equal, if and only if, both the wrapped value as well as the logs are the same.

```
jshell> Log<Integer> five = Log.<Integer>of(5)
five ==> Log[5]

jshell> five.equals(five)
$.. ==> true

jshell> Log.<Integer>of(5).equals(five)
$.. ==> true

jshell> five.equals(5)
$.. ==> false

jshell> Log.<Integer>of(5, "five").equals(five)
$.. ==> false

jshell> Log.<Integer>of(5, "").equals(five)
$.. ==> true

jshell> Function<Log<Integer>, Boolean> idf = x -> x.map(y -> y).equals(x) // functor identity
idf ==> $Lambda$...

jshell> idf.apply(Log.<Integer>of(5))
$.. ==> true

jshell> Function<Integer, Integer> f = x -> x + 1
f ==> $Lambda$...

jshell> Function<Integer, Integer> g = x -> x * 2
g ==> $Lambda$...

jshell> Log.<Integer>of(5).map(f).map(g).
...> equals(Log.<Integer>of(5).map(g.compose(f))) // functor associativity
$.. ==> true

jshell> Log.<Integer>of(5).flatMap(x -> Log.<Integer>of(x)).equals(Log.<Integer>of(5)) // monad r
$.. ==> true

jshell> Function<Integer, Log<Integer>> f = x -> Log.<Integer>of(x + 1, "add 1")
f ==> $Lambda$...

jshell> Function<Integer, Boolean> idleft = x -> Log.<Integer>of(x).flatMap(f).equals(f.apply(x))
idleft ==> $Lambda$...

jshell> idleft.apply(5)
$.. ==> true

jshell> Function<Integer, Log<Integer>> g = x -> Log.<Integer>of(x, "mul 2").map(y -> y * 2)
g ==> $Lambda$...
```

```
jshell> Function<Integer,Boolean> assoc = x -> Log.<Integer>of(x).flatMap(f).flatMap(g).
...> equals(f.apply(x).flatMap(g)) // monad associativity
assoc ==> $Lambda$...

jshell> assoc.apply(5)
$.. ==> true
```

## Level 5

Let's write some applications using JShell that makes use of our Log so as to observe how the values changes over the course of the computation. Save your methods in the file level5.jsh.

Start by modifying the sum method so that logging is now handled in a Log context.

```
Log<Integer> sum(int n) {
    ...
}
```

The following shows the output of sum(5).

```
jshell> sum(5)
$.. ==> Log[15]
hit base case!
adding 1
adding 2
adding 3
adding 4
adding 5
```

The *Collatz conjecture* (or the  $3n+1$  Conjecture) is a process of generating a sequence of numbers starting with a positive integer that *seems to always* end with 1.

```
int f(int n) {
    if (n == 1) {
        return 1;
    } else if (n % 2 == 0) {
        return f(n / 2);
    } else {
        return f(3 * n + 1);
    }
}
```

Modify the method f such that the following log is obtained using the Log context.

```
jshell> f(11)
$.. ==> Log[1]
3(11) + 1
34 / 2
3(17) + 1
52 / 2
26 / 2
3(13) + 1
40 / 2
20 / 2
10 / 2
3(5) + 1
16 / 2
8 / 2
4 / 2
2 / 2
1
```

Note that you should keep to the recursive implementations of methods sum and f. Do not convert them to iterative implementations using internal or external iteration constructs.

## Submission (Course)

Select course: CS2030 (2023/2024 Sem 1) - Programming Methodology II ✓

Your Files:

BROWSE

SUBMIT (only .java, .c, .cpp, .h, .jsh, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files

© Copyright 2009-2023 National University of Singapore. All Rights Reserved.  
Terms of Use | Privacy | Non-discrimination

MySoC | Computing Facilities | Search | Campus Map  
School of Computing, National University of Singapore