

# Concurrent programming

Kristoffer Klokke

2022

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Anonymous,- and lambdafunctions</b>	<b>3</b>
<b>3</b>	<b>Streams</b>	<b>5</b>
3.1	Parrallel streams . . . . .	6
<b>4</b>	<b>Bugs and solutions of parallelizing</b>	<b>6</b>
4.1	Race conditions . . . . .	7
4.2	Unmutable objects . . . . .	7
<b>5</b>	<b>Threading in Java</b>	<b>7</b>
<b>6</b>	<b>Safe shared objects</b>	<b>9</b>
6.1	Constructing objects . . . . .	10
6.2	Thread safe classes . . . . .	10

# 1 Introduction

Concurrency is the act of having multiple execution done simultaneously which interact with each other.

This is done to utilise multiple CPU cores rather than rely on CPU speed. Not only this but instead of having single powerful computers, bigger networks of computers can be used.

The benefits comes at a cost of complexity, due to the all possible outcomes of different timed execution.

The risk includes race-conditions where variables are changed while used in another thread or liveness where threads are stuck forever waiting for a non released variable nad such.

## 2 Anonymous,- and lambdafunctions

For at simple class which is given in an argument, instead of creating a class and then parsing it, the class can be created in the argument field.

For instance a class which implements comparable, it can be programmed as such:

```
1 public interface StringExecute {
2     public void run(String content);
3 }
4
5 public static void doAndMeasure( StringExecutable
6     runnable ) {
7     long t1 = System.currentTimeMillis();
8     runnable.run();
9     System.out.println( "Elapsed time: " + (System.
10         currentTimeMillis() - t1) + "ms" );
11 }
12
13 public static void anonFunc() {
14     doAndMeasure(new StringExecute() {
15         public static void run(String content) {
16             System.out.println(content + " Wow!");
17         }
18     });
19 }
20
21 public static void lambdaFunc() {
22     doAndMeasure( (content) -> System.out.println(
```

```

21         content + " Wow!"));));
22     }
23     public static void lambdaFuncOpt() {
24         doAndMeasure( (content) -> System.out::println); //
25         Only prints content and not + " Wow!"
    }

```

Here lambda function is only possible due to the compiler knowing what type of object is created due to restrains from the function and the runnable interface only contains a single run function. Another use of lambda expression is when working with maps.

```

26     public static void main() {
27         String text = "Hello world hope your having a good
28         day!";
29         Map<Charachter , Interger> occurrences = new HashMap
30         <>();
31         for(int i = 0; i < text.length(); i++){
32             final char c = text.charAt(i);
33             if(occurrences.containsKey(c))
34                 occurrences.put(c,occurrences.get(c)+1);
35             else
36                 occurrences.put(c,1);
37         }
38         for(int i = 0; i < text.length(); i++){
39             final char c = text.charAt(i);
40             occurrences.merge(c,1,(currValue,value) ->
41                 currValue+value);
42         }
43     }

```

Both for loops do the same, the second uses merge which takes a position (c) and a default value (1) and a bifunction which is a function with two inputs. The arguments will automaticly be assigned such currValue is the current hash value and value is the same as the default value 1.

Some of the most usefull built-in functional interfaces includes:

- Predicates - 1 argument returns boolean
- Functions - 1 argument returns result
- Suppliers - Like Functions but no arguments
- Consumers - 1 argument no return
- Comaprators - implements compareTo

### 3 Streams

Streams are like an foreach and like the name it is a stream of data. In more fine words it is a monad which is datastructure of a sequence of steps of operations.

There are different types of streams, the *Stream* is an object stream, whereas primitives stream also exists *IntStream*, *LongStream*, *DoubleStream* with possibilities like *IntStream.range*(1, 4) which has a stream of 1,2,3 and *IntStream.range*(1, 4).*sum*()

To tell the compiler if an object stream is transformed into a primitive type *.mapTo*(*Int*, *Double*, ...) with a parser as argument.

The most common functions used by streams are:

```
40 List<String> words = Arrays.asList("watercan", "digital"  
    , "citizen");  
41 //Performs function with each element  
42 words.stream().forEach(word -> System.out::println);  
43 //Modifies the element  
44 words.stream().map(word -> String::length);  
45 //Only elements which fulfill the function will 'parse'  
46 words.stream().filter(word -> word.startsWith("d"));  
47 //Streams the sorted stream  
48 words.stream().sorted((s1, s2) -> s1.compareTo(s2));  
49 //Counts the number of elements  
50 words.stream().count();  
51 //replace element by stream of given function  
52 words.stream().flatMap(word -> Stream.of(word.split("a"))  
    );  
53 //Gather elements in stream, for analysis or into an  
    object ex list  
54 words.stream().collect(Collectors.toList());  
55 //Reduce takes a stream and makes a single element Ex.  
    this takes the longest string  
56 words.stream().reduce((s1,s2) -> s1.length() > s2.length  
    () ? s1 : s2);
```

The functions can then be chained together, so methods which returns a stream can be worked upon.

When working with file stream it is done as following:

```
57 try( Stream< String > lines = Files.lines(Paths.get("text.txt"))) {  
58     lines.forEach(System.out::println);  
59 }
```

```

60 | catch ( IOException e) {
61 |     e.printStackTrace();
62 | }

```

By using the try and putting it into the parenthesis it will cause the stream to close when the stream is finished.

When a terminal operation (returns result rather than intermediate which returns stream) is done on the stream, it will close the stream. To counteract this a function which returns the stream may be used as such:

```

63 | Supplier<Stream<String>> streamSupplier = () -> Stream.
    | of("d2", "a2").filter(s -> s.startsWith("a"));
64 | //Returns true due to there is elements in the stream
65 | streamSupplier.get().anyMatch(s -> true); //
66 | //Returns false due to elements existing in stream
67 | streamSupplier.get().noneMatch(s -> true);

```

### 3.1 Parallel streams

When working with parallel stream the stream is outsourced to threads using a common ForkJoinPool.

This makes stream elements being handled side by side.

Some function needs modifications such as reduce:

```

68 | words.stream().parallelStream().reduce(0,(sum,s) -> sum
    | + s.length(), (sum1, sum2) -> sum1+sum2);

```

Reduce is now broken up onto three arguments,

- identifier - initial value
- accumulator - takes current result and element and operates
- Combiner - takes two partial accumulators and combines them

Like so the *sort()* function will either wait for all parallel streams or use parallel sorting in large amounts.

## 4 Bugs and solutions of parallelizing

When working with parallel threads it is important to encapsulate as much as possible. Threads may access data outside its thread, but this can lead to some of the following bugs.

## 4.1 Race conditions

This is a bug which occurs when two threads are accessing the same global variable.

This is a bug based upon atomicity is not kept and an operation is not done before another access data from the operation.

The most simple form is two threads which are incrementing a variable.

Say that the first threads read the variable, at the same the second thread reads the variable.

This will conclude in both threads having the same value for the variable and when incrementing the variable will only increment once.

To counteract this lock can be used. This is where code snippets are ran synchronised and threads are not able to execute twice.

## 4.2 Unmutable objects

Most often objects are a problem of race conditions.

To counteract this an object is made unmutable.

This means the object:

- is not including setters
- is not including references to itself
- has only final methods and variables
- is private such no method is overwritten

## 5 Threading in Java

Threading is done through the *Thread* object which takes a runnable object as an argument.

The thread is then executed by *start()* and *join()* is used as a wait till a thread is done.

If join is not used the main thread will stop its execution before child threads are done.

```
69 Thread t1 = new Thread( () -> {for(int i = 0; i < 1000;  
    {i++; System.out.println(i);}}});  
70 t1.start();  
71 try{  
72     t1.join();
```

```

73 }
74 catch(InterruptedException e) {e.printStackTrace();}

```

The try is due to threads may be interrupted, which otherwise would case the program to halt, due to the thread never joining.

For when working with locks in java the following can be done:

```

75 private static class Counter {
76     private int c = 0;
77 }
78
79 public static void main() {
80     Thread t1 = new Thread( () -> {for(int i = 0; i <
81         1000; i++) {
82             synchronized( counter ) {
83                 counter.c++;
84             }
85         });
86     Thread t2 = new Thread( () -> {for(int i = 0; i <
87         1000; i++) {
88             synchronized( counter ) {
89                 counter.c++;
90             }
91         });
92     t1.start();
93     t2.start();
94     try{
95         t1.join();
96         t2.join();
97     }
98     catch(InterruptedException e) {e.printStackTrace();}
99 }

```

The synchronized will here lock the counter object before access from which the code is executed. This will ensure that c is equal 2000.

Another way is to making a whole method synchronised by:

```

97 private synchronized void increment() {
98     i++;
99 }

```



## 6 Safe shared objects

Shared objects between threads can introduce many dangers, mainly that thread can be sure to read the same value it just wrote.

Stale data is term for none up to date values from shared objects which can lead to non expected outcomes.

A solution for this is synchronizing the shared object. By this only one thread can access the shared object at a time, therefore ensuring no stale data.

Another solution is all variables being local to each threat

In Java there is a safety gurantee to variables, that once updated they are fully dated. Unless the type of variable is long or double, due to the 64 bit required data, it is updated by two times of 32 bit each, therefore half up-dated values may accour.

Synchronization is done by locking the given object, such no other thread can access it.

Volatile variables are a weak synchronised variable, due to not ensuring atom-icity. It is not as safe, but can be used for veryfying states rather correctness. Therefore volatile variables should only be used when its value does not depend on current value or only one threads updates aswell as the variable not participating in invariants.

Ad-hoc thread confinement is a way of describing how an object may only be in certain safe ways. This is ofcourse not a stable way of implementing objects.

A more stable alternative is stack confinement, where reference an variables are all local to a thread.

ThreadLocal solves helps to ensure no leaks could accour in stack confiment, by making an object with variables which can not be accessed outside a thread.

Another way is an immutable object, which is a final object which only contains final fields, can not be modified and construction has not been escaped. It is not enough to just make the object reference final, due to the object itself being able to change its variables.

The immutable object may also be good in case of outside methods, this ensures that the methods may not change anything or save a reference.

A object may be treated as ummutable such as the date object, but it requires clear documentation.

## 6.1 Constructing objects

When constructing objects it is important, that a reference to the object not escape before initialization.

This can result in half initialized objects being treated as done.

To prevent this a static initializer may be used, which creates the object, parses it to a method which makes it public and returns the reference.

## 6.2 Thread safe classes

First the invariant variables are identified and variables for object states. Such they can get policy according to use.

State-depend operations are preconditions on operations, which also acts as a policy for the class.

It will always be easier to encapsulate methods for objects such synchronised always has the correct lock.

Remember to always use thread safe classes, to make life easier but still remember to handle them safely.

When access multiple variables wich relates send them as tuples, such they always are synchronized.

A thread safe class may include public variables, which has to be handled safe but in some cases it makes sense to make them public.

Some class may have to be extended with a synchronised method, such as a list extension with a method for adding of absent.

A custom lock may also be but in a method Ex. `public boolean m() { synchronized(list) {...}}`, use the list variable as a custom lock.