Execises 11

Some of these exercises includes timing. As we have seen earlier, timing results are much more stable if you can repeat many times. The online kotlin playground is not well suited for accurate timing, and I recommend to either use the command line compiler or an IDE.

Compiling from command prompt

Kotlin, like Java, need its class path to be set up. For these smaller programs I find it easiest to just pass the coroutine jar file as classpath parameter.

The kotlinx-coroutine-core jar file is here, this link should download it.

Compiling the program 'MyProg.kt' can be done as:

```
kotlinc MyProg.kt -cp kotlinx-coroutines-core-1.4.0.jar
```

this produces a large number of class files. To run the program:

'java -cp *:. MyProgKt'

Notice, the missing extension on the filename, and that the class file for `MyProg.kt` is called `MyProgKt`.

But you are welcome to use an IDE - or the Playground when possible.

1. Airline tweets

This exercise will use some twitter data from the Kaggle site, Twitter US Airline Sentiment. The file to use in this exercise is in the code directory as file `AirlineTweets2015.csv`.

The file is comma separated, and has 15 columns.

Green

1. How many lines are there?

The code in file "ReadTweets.kt" contains functions to read lines from the file, and convert them to `Tweet` objects. Count the number of lines in the file using the `readLines()` function and a sequence operator.

2. Form a stream of Tweets

Some records are split over more than one line, and some seems incomplete. A wellformed tweetline is one which when split using comma, will have 15 elements.

Write a function which returns a Stream of tweet objects from the file, and use it to count how many well formed tweets there are.

Hint: There is a function in "ReadTweets.kt" which produce a tweet from a line if the line is wellformed, or null if it is not.

3. From streams to channels

To simulate that the tweets come in live, you should create a channel `tweetChannel`, and have one coroutine (a producer) write the wellformed tweets to tweetChannel, while an other coroutine (a consumer) reads them from the channel and print the tweet text of every 200 it receives.

The producer should have a delay of 5ms between each write to the channel.

Yellow

4. Channel fan out

The airline field of the tweets can be one of: airline (presumably a generic name), Virgin America, United, Southwest, Delta, US Airways, American.

We are going to take a look at the data for Southwest and American.

Add two more channels to the program, one for Southwest and one for American. Rewrite the consumer from before to inspect the airline field of each tweet it receives. If it is for Southwest it sends the tweet to the southwest channel, and similarly for American. Other tweets are ignored.

Create two new coroutines which listen to the Southwest and American channels. This time they should print the tweet text of every 25 received tweet.

5. Filterning channel input

It turns out that Southwest want to have all tweets printed where the negativereason field is non-empty, while American only wants the negative reason printed if the negativereason_confidence is above 0.75.

Change the consumer coroutines for the two airlines to account for this.

Red

6. Fan-in

The two airlines Southwest and American from before desires to hire a twitter the observation office ‘Birdsong’. Change the two consumers for Southwest and American so they still filter as before, but rather

than printing anything, they both all tweets which pass the filter to a birdsong channel.

The birdsong office (a coroutine consumer) will read tweets, and aggregate the field negativereason. Every time the birdsong office have received 200 tweets with a negativereason, it will print an aggregate map of reason and count of that reason (*Hint: a groupBy operation*).

2. Coroutine creation time

We saw it took quite some time to start a new Thread. This java program prints the number of nanoseconds it takes to create a thread, start it, and until it executes its first statement:

```
class StartThreadTiming {
  public static void main(String[] args){
    final long start = System.nanoTime();
    new Thread( ()->{
      long time = System.nanoTime() - start;
      System.out.printf("Time to start %,d\n", time);
    }).start();
  }
}
```

Green

1. Getting consistent results

Try change the program above to include a loop which repeats the measurement a number of times. (Try with 10,20,50,100, 500, 1000 times)

Pick a repeat number where you get a reasonable consistent result.

How do the number of cores influence the java code?

Are you sure the printing time does not influence the results?

2. Timing coroutine start

The corresponding kotlin coroutine program is this one:

```
import kotlinx.coroutines.*
fun main() {
  runBlocking {
    val start = System.nanoTime()
    var time : Long
    launch {
      time = System.nanoTime() - start
      println("Time to start %,d".format(time))
    }
  }
}
```

Again, try with starting more than one coroutine, try with the same number of repeats to get consistent (or strange) results.

Is it true that coroutines starts faster? How big is the difference?

Yellow

3. Fixing a mistake?

Compare this measurement which repeats 100 times.

```
import kotlinx.coroutines.*
fun main() {
  runBlocking {
    for(i in 0..99){
      val start = System.nanoTime()
      launch {
        val time = System.nanoTime() - start
        println("Starttime: %,d".format(time))
      }
    }
  }
}
```

to this one, which also repeats a 100 times.

```
import kotlinx.coroutines.*
fun main() {
  runBlocking {
    val list:MutableList<Long> = MutableList<Long>(100, {0})
    for(i in 0..99){
      val start = System.nanoTime()
      launch {
        list[i] = System.nanoTime() - start
      }
    }
    delay(1000) // wait for all coroutines to have stored result
    list.forEach{ println("Starttime: %,d".format(it)) }
  }
}
```

On my machine there is a difference in reported start-up times of about a factor 4. Do you get similar results, and why? Does that influence your trust in the timing done for the Java program above.

4. Different dispatchers

In section on dispatchers and threads in the kotlin documentation, they mention the Default, and the Unconfined dispatchers.

Try to launch the coroutines in these two dispatchers and see if and how that influence startup time. Use the last of the above two programs (or your own version) for comparison.

3. Handling timing info

As we saw in the previous exercise, one has to be carefull in how one collects data. One idea for collecting data is to have a dedicated coroutine to collect timing data. In this exercise we will explore that idea.

The idea is that we will have a channel `measurement` on which other coroutines (in our case below, just a single one) can send measurements of type Long (which is the type returned by `System.nanoTime()`). There is a consumer coroutine `measurementCollector` which reads from thechannel, and keeps sum, sum of squares, and number of measurements. It is given a parameter `n`, which is the number of measurements to expect.

The exercise takes it outset in this program:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
fun main() {
  runBlocking {
    val measurement = Channel<Long>()
    val reps = 100
    launch({ measurementCollector(reps, measurement) }
    for(i in 0..reps-1){
      val start = System.nanoTime()
      launch {
        val time = System.nanoTime() - start
        // TODO - send the measurement
      }
    }
  }
}

suspend fun measurementCollector(n: Int, measurements: Channel<Long>): Unit {
    var count = 0L
    var sum = 0L
    var ssm = 0L
    // *** TO DO ***
    val mean = (sum/count).toDouble()
    val sdev = Math.sqrt( (ssm - mean*mean*count).toDouble() )/(count-1)
    println("%,6.1f ns +/- %,8.2f %,d%n".format( mean, sdev, count) )
}
```

Yellow

1. Finish the program

Write the missing pieces and verify with the results from exercise #2 that they are "reasonable".

2. What is a good number of repetitions

Try with a different number of repetitions to see if the mean or the derivation get lower. Is it a problem to start say 1000000 coroutines?

3. Warm up

It might be that the JIT compiler need to efficiently compile this code. Change the code such that it skips the first many (say 10%) measurements.

Red

4. How long does it take to send

One question you might have about this approach is how long time does it take from a message (in our case a Long) is sent until it is received. Design a measurement which answers that issue. *Hint: you can send a nanosecond measurement over the channel.*

If you use the measurementCollector to collect your data, will that in itself influence the validity of the measurements?