Exercises 10 - coroutines week 1

The exercises for this week will be in kotlin. There are three meaningful ways in which to write and run kotlin programs:

- Online using kotlin playground
- Instaling the kotlin command line compiler
- Using an IDE - intellij IDEA is a good choice for this. You can use the community version, or sign up for an educational license.

The exercises can be solved using the kotlin playground.

A. Kotlin intro exercises, no coroutines here

This first set of exercises are meant to get you know and try enough of kotlin to read and use the code in the other section of the exercises.

1. Var/val final (and perhaps some closures)

Green

Consider this kotlin program:

```
fun main() {
    val a = listOf(1,2,3,4,5,6,7,8,9,10)
    var sum = 0
    a.forEach {x -> sum += x}
    println("Sum: " + sum)
}
```

and it similar java program (not working)

```java
import java.util.Arrays;
import java.util.List;
class Summa {
  public static void main(String[] args){
    final List<Integer> a = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
    Integer sum = 0;
    a.forEach( x -> sum += x );
    System.out.println("Sum: " + sum);
  }
}
```

Why will the java program not compile? Is there any way to use a lambda expression and forEach to sum the elements in Java?

Yellow

If you know the concept of closures, what does this example tell about support of closures in kotlin and Java? (If you do not know what a closure, this class is not the one for learning about it).

2. Null check at compile time

Green

Kotlin, like some other modern languages, has support for compile time check of not-null. This is in my oppinion one of the best things since sliced bread. It helps against the dreaded null pointer exception.

```kotlin
fun main() {
    val list = listOf("aba","x", "xyz", "wasitacaroracatisaw","palindrome")
    list.filter(::palindrome).forEach(::println)
}

fun palindrome(s:String):Boolean {
    var first=0; var last=s.length-1
    while (first < last){
        if (s[first++] != s[last--]) return false
    }
    return true
}
```

Try to add `null` to the list of words. Then fix the palindrome function to allow it to accept null strings.

- It must return `false` in case of null
- It must test for null using the *elvis operator* (google kotlin elvis operator).

3. Extension functions and trailing lambdas

Like C#, Kotlin supports *extension functions*. In java terms, it is a static method which is given a syntax which looks like it is calling a method on an object. It is described in Extensions.

Like Java and C#, kotlin has support for lambda expressions. However, they have a special syntax for a special case, and they use this all over the place. It is called parsing trailing lambda. In essence, it allow a function which takes a lambda expression as its last parameter to be called like this: `foo { x -> x*x }` rather than `foo ( { x -> x* x } )`.

It is in particular often used for lambda's with no parameters.

Green

1. Define a function `ifNotEmpty`, which takes a String? and a lambda expression of the type String->Unit. It should evaluate the lambda expression only if the string argument is not-null and not the empty string.

   That is, a call like `ifNotEmpty("Joe", { s-> println(s) })` should print "Joe".

2. Verify (by making a call to the function) that it can also be called like `ifNotEmpty("Joe") { s-> println(s) }` due to the trailing lambda rule

Yellow

1. Change the definition of `ifNotEmpty` to be an extension function on String?. Verify again that you are now able to call the function as: "Joe".ifNotEmpty { s-> println(s) }

B. Sequences and yield

The following exercises are all about using the sequence builder and yield to create sequences.

4. All leap years between 1940 and 2040

Green

Write a generator of leap years between those two years (both inclusive). Your solution must use `yield()` (as this is what the exercise is about).

5. Every second element

Green

- Write an extension method for Sequence which skips every second element.

```kotlin
fun <T>Sequence<T>.dropEverySecond(): Sequence<T> = sequence {
    var iter = this@dropEverySecond.iterator()
    // TO DO
}
```

The use of the strange notion `this@methodName` is used to get hold of the object this extension method extends.
You often need to use the iterator of a sequence if you want to read elements one at a time as sequences as such are immutable in kotlin.

Try your solution in a main function such as:

```kotlin
fun main() {
    val seq = (1..40).asSequence()
    seq.dropEverySecond().filter { v -> v%3 == 0 }.forEach(::println)
}
```

6. Rewrite to statemachines

Yellow

Rewrite the leap-year and dropEverySecond to a statemachine as outlined in the slides. Pay attention that your states are closely related to the code in the two prior solutions.

7. Merging two sequences

Green

1. Write a kotlin function to merge to sorted integer sequences into a new sorted integer sequence.

```kotlin
fun merge(s1: Sequence<Int>, s2: Sequence<Int>): Sequence<Int>
```

Hints: You will most likely need a **sequence builder** and **yield/yieldAll**. Also, it might be easier to work with iterators internally (`s1.iterator()` gives you a kotlin iterator) as in the first two exercises.

8. Person lists

Yellow

In the file ‘persons.kt’ is some kotlin code which can produce a number of persons with random names, telephone numbers and addresses (`PersonGenerator.persons()`). Assume you use it to generate a sequence of 10000 random persons.

- Using the kotlin versions of map, filter, reduce etc. (See Sequence), solve the following:
    - print the first 10 persons generated (not the same on each run as they are random)
    - print out the number of persons whose first name has less than 5 letters, and who live in a city with an odd zip code.
    - group persons by the length of their name, and print out the number of persons in each group
- There is a very simple binary tree implementation in the code (class PersonTree). It is intended to be used as in this main:

  ```
  fun main() {
      val tree = PersonTree()
      PersonGenerator.persons(10000).
          filter { it.name.length > 22 }.forEach { tree.insert(it) }
      for( p in tree.sorted())
      println( p.nice() )
  }
  ```

  Finish the sorted method. It should return a sequence of all persons sorted by their name.
- Using the merge function above (including solving the issue of comparing persons) create two short sequences of persons, and ensure that they are merged correctly.

9. Red exercises

- I am not sure there is an "elegant" solution to this, if you find one please mail me. Try to write the dropEverySecond for java’s stream. Java does not have extension methods, so you are restricted to using the existing methods (ie. map, filter, etc). *Hint: I have the feeling there might be a solution in using the iterate, pairs, and a filter, but I have not yet tried it.*
- I am not sure there is an elegant solution to this one either. Try to write a iterator/sequence for the search tree of persons, which *do not* use yield.