Java Workbook

09. Concurrency

~by Andrzej "Andret2344" Chmiel

Exercise 1: Filling in with squares

Given the following list:

```
class mainSquares {
    public static void main(final String[] args) {
        // Fix the following line
        final List<Integer> integers = null;
        integers.add(2);
        integers.add(3);
        integers.add(4);
        integers.add(5);
        for (final int i : integers) {
            integers.add(i * i);
        }
        for (final int i : integers) {
            System.out.println(i);
        }
    }
}
```

Initialize the `integers` variable correctly so the program compiles and runs without any exceptions and prints out this:

```
2
3
4
5
4
9
16
25
```

Exercise 2: Parallel stream concatenation

Given the following code:

```
class MainParallelStreamConcatenation {
    public static void main(final String[] args) {
        final List<String> strings = Arrays.asList(
                " Kenobi!    ",
                "       General   ",
                "  there!      ",
                " Hello ");
        System.out.println(concatenate(strings.parallelStream()));
    }

    private static String concatenate(final Stream<String> stream) {
        return null;
    }
}
```

Fill in the `concatenate` method, so it concatenates all the strings from the parallel stream, skipping all whitespace characters and returns the following result:

`Hellothere!GeneralKenobi!`

Exercise 3: Recursive Trapezoidal Rule

Given the following code:

```
class MainRecursiveTrapezoidalRule {
    public static void main(final String[] args) {
        final ForkJoinTask<Double> integral = new Integral(
                Math::sin,
                0,
                Math.PI,
                Math.PI / 1000.0);
        final ForkJoinPool pool = new ForkJoinPool();
        final double sum = pool.invoke(integral);
        System.out.println(sum);
    }
}
```

Create a class that inherits from the `RecursiveTask<Double>` class, so the code compiles and uses the recursive trapezoidal rule for approximating the definite integral of the given function over the given range ($f(x) = sin(x)$ over $x \in [a, b]$). For the code above, the result should be near the number of 2.

Exercise 4: Linear Trapezoidal Rule

Given the following code:

```
record Interval(double begin, double end) {
}

class MainLinearTrapezoidalRule {
    public static void main(final String[] args) {
        System.out.println(integral(
                Math::sin,
                divideBigInterval(0, Math.PI, 1000)));
    }

    private static Stream<Interval> divideBigInterval(final double a,
                                                      final double b,
                                                      final int k) {
        return Stream.empty();
    }

    private static double integral(final DoubleUnaryOperator function,
                                   final Stream<Interval> intervals) {
        return 0.0
    }
}
```

Fill in the `divideBigInterval` method, so it splits the range $[a, b]$ into k equal `Interval` objects and returns a stream of them, and the `integral` method, so it calculates the approximate area of each the `Interval` object and sums them up and returns a result. Running the code above prints out a result near to the number of 2.

Exercise 5: Parallel printout

Given the following code:

```
class MainParallelPrintout {
    public static void main(String[] args) {
        Stream.iterate(1, i -> i + 1)
                .limit(100)
                .parallel()
                .map(i -> i * i)
                .filter(i -> i % 2 == 0)
                .forEach(System.out::println);
    }
}
```

Change the code, so the stream is still parallel and prints out all the numbers in ascending order.

Exercise 6: Wait for me!

Given the following code:

```
class MainWaitForMe {
    public static void main(final String[] args) {
        runAfter(
                MainWaitForMe::supplyMethod,
                MainWaitForMe::operateMethod,
                MainWaitForMe::consumeMethod);
    }

    private static <T> void runAfter(final Supplier<T> supplier,
                                     final UnaryOperator<T> operator,
                                     final Consumer<T> consumer) {
        consumer.accept(operator.apply(supplier.get()));
    }

    private static int supplyMethod() {
        System.out.println(Thread.currentThread().getName());
        return 1;
    }

    private static int operateMethod(final int value) {
        System.out.println(Thread.currentThread().getName());
        return value + 2;
    }

    private static void consumeMethod(final int value) {
        System.out.println(Thread.currentThread().getName());
        System.out.println(value);
    }
}
```

The code works fine, but all the operations happen in the main thread. Change only the `runAfter` method so that each call happens in a thread from an executor pool, so the code prints out this:

```
pool-1-thread-1
pool-1-thread-2
pool-1-thread-3
3
```

Exercise 7: Are threads good?

Using the previously created `Timer` class, count the time of the following operations: Create a stream of two billion random integers and then filter them so that only numbers $x \in [1000, 2000]$ stay in the stream. Use:

1. A single-threaded stream.
2. A multithreaded stream.

Compare times and conclude.

Exercise 8: Alignment

Given the following code:

```
class MainAlignment {
    public static void main(final String[] args) {
        final ExecutorService taskExecutor = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 10; i++) {
            taskExecutor.execute(new MyTask(i));
        }
        System.out.println("All done");
        taskExecutor.shutdown();
    }
}

class MyTask implements Runnable {
    private final Random random = new Random();
    private final int i;

    MyTask(final int i) {
        this.i = i;
    }

    @Override
    public void run() {
        try {
            System.out.println("Starting " + i);
            Thread.sleep(random.nextInt(5000));
            System.out.println("Finishing " + i);
        } catch (final InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Fix the code so the `"All done"` text is printed out after all ten tasks finish. Don't remove any code, just add own code (whole lines or changes to the existing ones).

Exercise 9: The order does matter

Given the following code:

```
class MainTheOrderDoesMatter {
    public static void main(final String[] args) {
        new Responder().run();
        System.out.println("Can I be before, please?");
    }
}

class Responder implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println("Duh, be my guest!");
        } catch (final InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Not changing the `Responder` class and not using the `Thread` class, fix the code so running it prints out this:

```
Can I be before, please?
Duh, be my guest!
```

Exercise 10: Stand in the queue

Given the following code:

```
class MainStandInTheQueue {
    public static void main(final String[] args) {
        final MyListResource myListResource = new MyListResource();
        final Thread listThread1 = new Thread(() -> {
        for (int i = 1; i < 11; i++) {
                try {
                    myListResource.add(i);
                } catch (final InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

        final Thread listThread2 = new Thread(() -> {
            for (int i = 1; i < 11; i++) {
                try {
                    System.out.println(myListResource.getList());
                } catch (final InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

        listThread1.start();
        listThread2.start();
    }
}

class MyListResource {
    private final List<Integer> list = new ArrayList<>();

    public void add(final int x) throws InterruptedException {
        Thread.sleep(100);
        list.add(x);
    }

    public List<Integer> getList() throws InterruptedException {
        Thread.sleep(100);
        return list;
    }
}
```

Changing only the `MyListResource` class, fix the code, so it compiles and doesn't print out this:

```
[]
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```