Hands-On Exercises

STREAMS & LAMBDA EXPRESSIONS BY EXAMPLES

Lambda Expression have been released with Java 8 (March 2014). So, they are not completely new, but I know many experienced developers which are not familiar with them and/or use them where reasonable. The following collection of exercises about Streams & Lambda Expressions give a brief overview of the most important aspects in a simple form.

LAMBDAS & STREAMS BASICS

Streams & Lambdas support parallel operations on collections based on the fork-and-join-framework. The new concept of Streams gives the possibility to operate on a collection with Lambda Expressions.

A stream can be executed in serial or in parallel way. When you execute a stream in parallel, Java partitions the stream into multiple sub-streams. The results are then combined again to a single result. All this happens behind the scenes and is managed by the Java run time.

Syntax of Lambda Expressions

Lambda Expressions have three parts, i.e. ArgList, Arrow and Body.

Lambda = ArgList Arrow Body

ArgList = Identifier

| "(" Identifier [ "," Identifier ]\* ")"

| "(" Type Identifier [ "," Type Identifier ]\* ")"

Body = Expression

| "{" [ Statement ";" ]+ "}"

Examples:

x -> x + 1

(a, b) -> a \* b

(int a, int b) -> a \* b

() -> System.out.print("Hello World! I'm a Runnable")

(Point p) -> { System.out.print("p="); System.out.println(p);}

Functional Interfaces

You may find in the Java documentation the following description: "Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. [...]" (see [Package java.util.function](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html&sa=D&sntz=1&usg=AFQjCNFBJ_ZUOxeSeKnhS08ZP2DCMkDKgA" \t "https://www.sw-engineering-candies.com/blog-1/_blank))

Many useful Functional Interfaces are already implemented (see [Uses of Class java.lang.FunctionalInterface](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/lang/class-use/FunctionalInterface.html&sa=D&sntz=1&usg=AFQjCNH_JjkrqsvXxQhREx2w8HY9kXPbPA" \t "https://www.sw-engineering-candies.com/blog-1/_blank)). A functional interface is always annotated with ****@FunctionalInterface**** and has just one abstract method. The annotation is just informal, but it is a good practice to use it your own functional interfaces.

For example the ****java.lang.Runnable**** interface:

package java.lang;

@FunctionalInterface

public interface Runnable {

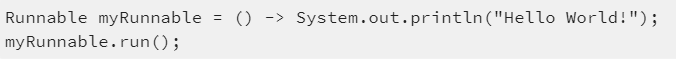
public abstract void run();

}

So, it is possible to assign and execute a Lambda Expression:

Lab Exercise 1:

Create a class and execute the following within the main method.



Expected output:

Hello World!

Streams of Elements

The package ****java.util.stream**** contains classes and interfaces to support functional-style operations on streams of elements, such as map-reduce transformations on collections. With [Interface Stream<T>](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html&sa=D&sntz=1&usg=AFQjCNGAluA7Q5H9ZBs57uZQ6BxHKNNLrQ" \t "https://www.sw-engineering-candies.com/blog-1/_blank) as the central interface of this package and some primitive specializations for [IntStream](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html&sa=D&sntz=1&usg=AFQjCNHEV2uaeFlRTkfZQ52UneiuBNDU1g" \t "https://www.sw-engineering-candies.com/blog-1/_blank), [LongStream](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/LongStream.html&sa=D&sntz=1&usg=AFQjCNFgpOkAQ_W8tG1zD5Zpa_ecSTee5g" \t "https://www.sw-engineering-candies.com/blog-1/_blank), and [DoubleStream](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/DoubleStream.html&sa=D&sntz=1&usg=AFQjCNHBWuT1QChZXHqKBVGUWJO9cwvtTQ" \t "https://www.sw-engineering-candies.com/blog-1/_blank). This interface is used in many other packages like [java.io](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/class-use/Stream.html%23java.io&sa=D&sntz=1&usg=AFQjCNFdtPGFH6vgkRWyfX-3KNU9otHZWg" \t "https://www.sw-engineering-candies.com/blog-1/_blank), [java.nio.file](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/class-use/Stream.html%23java.nio.file&sa=D&sntz=1&usg=AFQjCNFFuRbeGcSbodRb9qZ0NV50c9bWBQ" \t "https://www.sw-engineering-candies.com/blog-1/_blank), [java.util.jar](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/class-use/Stream.html%23java.util.jar&sa=D&sntz=1&usg=AFQjCNE7QSDkILwsDmY6MJQ2i311uMxoDQ" \t "https://www.sw-engineering-candies.com/blog-1/_blank), [java.util.regex](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/class-use/Stream.html%23java.util.regex&sa=D&sntz=1&usg=AFQjCNH2DOPlE3yzJh6nsGhUzDZHF-Xubw" \t "https://www.sw-engineering-candies.com/blog-1/_blank), [java.util.stream](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/class-use/Stream.html%23java.util.stream&sa=D&sntz=1&usg=AFQjCNH8mjrFYP8aGdDd9EgO9R6J1R2ItQ" \t "https://www.sw-engineering-candies.com/blog-1/_blank) and [java.util.zip](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/class-use/Stream.html%23java.util.zip&sa=D&sntz=1&usg=AFQjCNGmg9_xEcfn-fPR8xHywq_7vQbUyg" \t "https://www.sw-engineering-candies.com/blog-1/_blank).

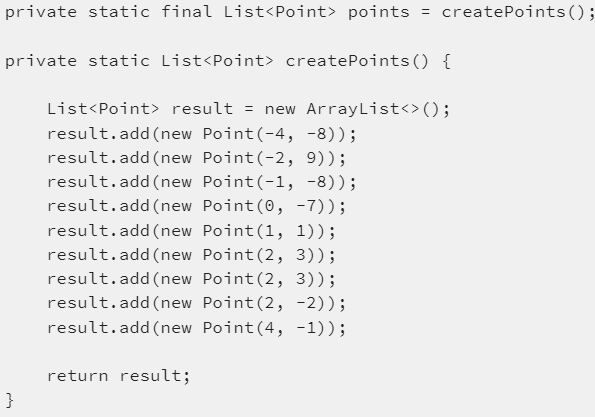
"Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce." (see [Package java.util.stream](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html&sa=D&sntz=1&usg=AFQjCNFtIoFSzuMIjdX9USv6hi9GjqiE1g" \t "https://www.sw-engineering-candies.com/blog-1/_blank)).

OUTPUT WITH LAMBDA EXPRESSIONS AND METHOD REFERENCES

For the following collection of lab exercises, we need a filled list to operate on existing data sets.

Lab Exercise 2:

Create a class “LambdaBasics” and add the following method.



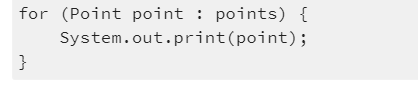
The class ****java.awt.Point**** is just a very simple class that holds two number and this is convenient for calculations. With some minor changes all the examples can work with different classes.

Standard in Java to Print all Elements of a List

Before we start with lambdas we have a look at the standard way in Java to print all elements of the list:

Lab Exercise 3:

Execute the following within the main method of the class defined in exercise 2.



Here the Iterator of the list is used to get each Point and do the print operation. We use an External Iterator (aka Active Iterator) - this gives full control over the order of execution, exception handling, etc. This is in principal good but the code is serial and can just be executed in a parallel way with serious effort.

ForEach Lambda Expression or Method Reference to Print all Elements of a List

The Iterable.forEach() method can take a lambda expression:

Lab Exercise 4:

Execute the following within the main method of the class defined in exercise 2.



or an instance method reference (System.out is an instance of PrintStream):



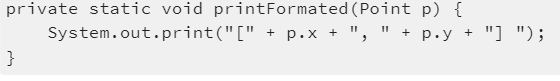
ere the forEach() of the stream is used to call the print operation. We call this an Internal Iterator (aka Passive Iterator) - which delegates the control to the Java run time. This code can be executed in a serial and parallel way. The disadvantage is that the order is not deterministic in the case it is executed in a parallel way.

ForEach to Print all Elements of a List with Special Formatting

With the following helper method the output can be printed in a more convenient format:

Lab Exercise 5:

Execute the following within the main method of the class defined in exercise 2.

Then we can use a simple lambda expression:



or we can use a static method reference:



**Kinds of Method References**

There are four kinds of method references that can be used (see also [Method References](http://www.google.com/url?q=http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html&sa=D&sntz=1&usg=AFQjCNExuQAXMWW-2yrEtzbCxDbU9jRYlw" \t "https://www.sw-engineering-candies.com/blog-1/_blank)):

* static methods (****MyClass::staticMethodName****),
* instance methods of a particular object (****myObject::instanceMethodName****),
* instance methods of an arbitrary object of a particular type (****MyType::methodName****) and
* constructors (****MyClass::new****).

**CALCULATIONS WITH MAPTOINT(), REDUCE(), IFPRESENT() AND SUM()**

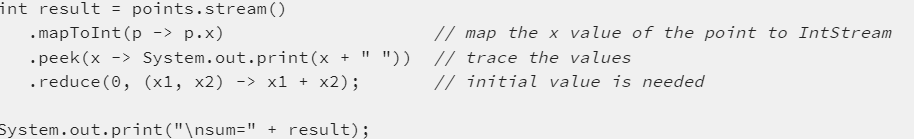
Examples that demonstrate the use of some intermediate and terminal operations.

Calculate Sum of all X-Coordinates with mapToInt () and reduce()

The operation ****mapToInt()**** creates a new stream of the type integer and fills it with the x-coordinate. Then the operation ****reduce()**** calculates the sum of all x-coordinates:

Lab Exercise 6:

Execute the following within the main method of LambdaBasics.



Expected output:

-4 -2 -1 0 1 2 2 2 4

sum=4

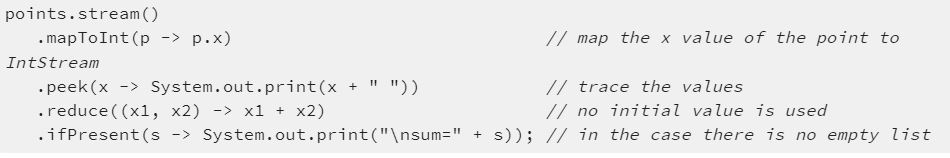
The intermediate operation ****peek()**** exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline (see [Interface Stream<T>](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html%23peek-java.util.function.Consumer-&sa=D&sntz=1&usg=AFQjCNHJ2dPP9R6APhFz_lU4A2jNih1Qzg" \t "https://www.sw-engineering-candies.com/blog-1/_blank)).

Calculate Sum of all X-Coordinates with mapToInt () and reduce() and ifPresent()

The initial value in the ****reduce()**** method can be empty, but in this case you should be sure that the stream has elements. If the stream could be empty the terminal operator ****ifPresent()**** should be.

Lab Exercise 7:

Execute the following within the main method of LambdaBasics.



Expected output:

-4 -2 -1 0 1 2 2 2 4

sum=4

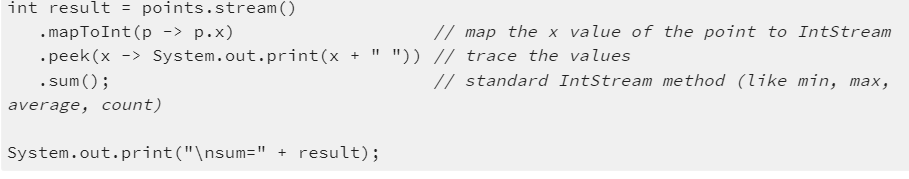
You may notice that the terminal operation reduce is not the last called method. In this case the return value of reduce() is the container object [OptionalInt](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/OptionalInt.html&sa=D&sntz=1&usg=AFQjCNH9sbW1RI8l0A47iRszEgSlYx4Bxw" \t "https://www.sw-engineering-candies.com/blog-1/_blank). This class has the method ****ifPresent()**** which is called. Other useful methods of this container are ****isPresent()****, ****orElse()**** and ****getAsInt()****.

Calculate Sum of all X-Coordinates with sum()

Another way to calculate the sum are the statistics functions of stream class.

Lab Exercise 8:

Execute the following within the main method of LambdaBasics.



Expected output:

-4 -2 -1 0 1 2 2 2 4

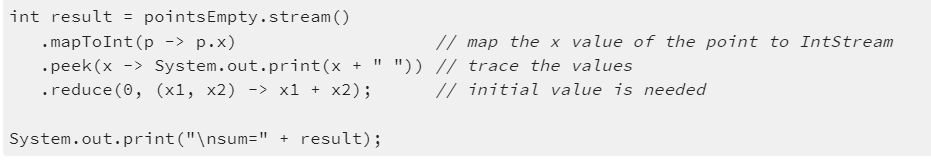
sum=4

Calculate Sum of all X-Coordinates with reduce() on Empty List

Notice that ****pointsEmpty**** is an empty list For empty lists the behavior may be different. As long the reduce() method has an initial value, the result is correct:

Lab Exercise 9:

Execute the following within the main method of LambdaBasics.



Expected output:

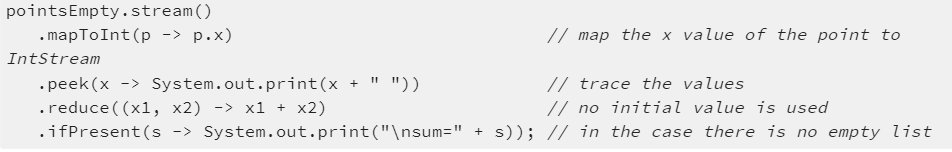
sum=0

Calculate Sum of all X-Coordinates with reduce() and ifPresent() on Empty List

Notice that ****ifPresent()**** method doesn't execute the lambda for an empty list.

Lab Exercise 10:

Execute the following within the main method of LambdaBasics.



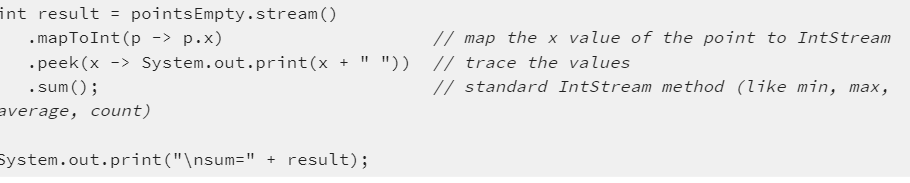
Expected output (is empty):

Calculate Sum of all X-Coordinates with sum() on Empty List

The ****sum()**** method works properly also with an empty list.

Lab Exercise 11:

Execute the following within the main method of LambdaBasics.



Expected output:

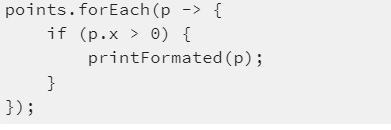
sum=0

Use of filter() and distinct()

Filter all Points which are Positive in X with Lambda Expression

Lab Exercise 12:

Execute the following within the main method of LambdaBasics.



Expected output:

[1, 1] [2, 3] [2, 3] [2, -2] [4, -1]

Filter all Points which are Positive in X with filter()



Expected output:

[1, 1] [2, 3] [2, 3] [2, -2] [4, -1]

Filter Distinct Points which are Positive in X with filter()



Expected output:

[1, 1] [2, 3] [2, -2] [4, -1]

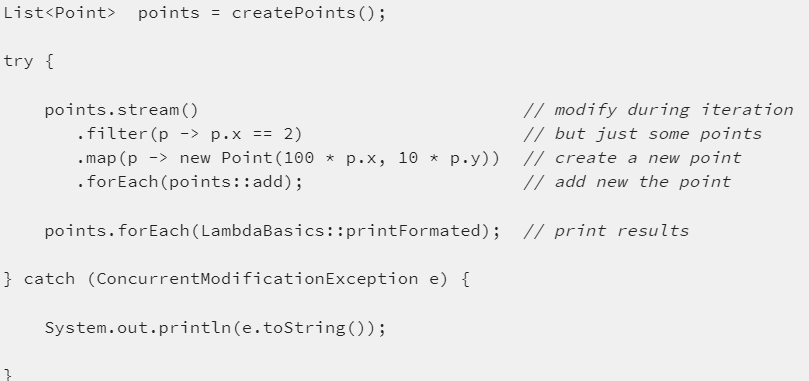
WRONG/CORRECT WAY TO ADD POINTS TO AN EXISTING LIST

Add Point to Original List in the Case the X-Value is Equal Two

This implementation is wrong!

Lab Exercise 13:

Execute the following within the main method of LambdaBasics.

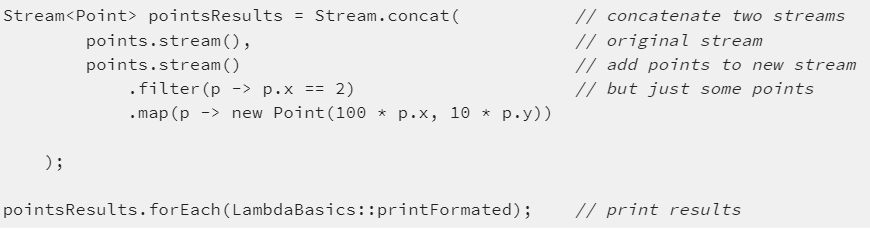


Expected output:

java.util.ConcurrentModificationException

Add Point to New List in the Case the X-Value is Equal Two

This implementation is correct!



Expected output:

[-4, -8] [-2, 9] [-1, -8] [0, -7] [1, 1] [2, 3] [2, 3] [2, -2] [4, -1] [200, 30] [200, 30] [200, -20]

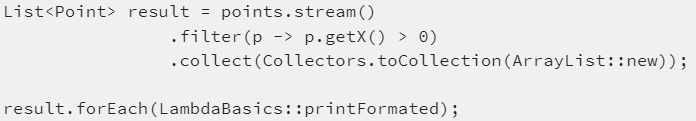
USE COLLECT() AND COLLECTORS

The class ****Collectors**** is used for accumulating elements into collections and has a large number of useful reduction operations like grouping, partitioning, joining, counting, statistic, mapping (see [Class Collectors](https://www.google.com/url?q=https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html&sa=D&sntz=1&usg=AFQjCNGOUWRH0WuZXHdxjTJDncw29gkA8A" \t "https://www.sw-engineering-candies.com/blog-1/_blank)).

Use Collectors to Store all Points with Positive X into a New List

Lab Exercise 14:

Execute the following within the main method of LambdaBasics.

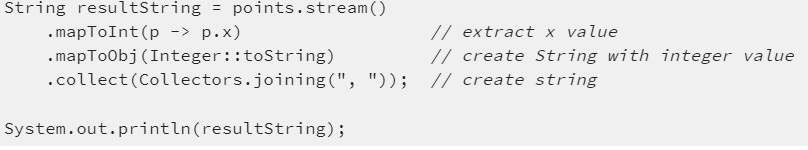


Expected output:

[1, 1] [2, 3] [2, 3] [2, -2] [4, -1]

The interesting thing here it the ****Collectors.toCollection(ArrayList::new)**** which creates a new ****ArrayList**** and fills it.

Use Collectors to Create Comma Separated String



Expected output:

-4, -2, -1, 0, 1, 2, 2, 2, 4

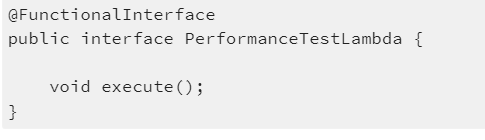
The performance of the method Collectors.joining() is better than a direct String concatenation in the lambda expression.

PERFORMANCE OF SEQUENTIAL AND PARALLEL EXECUTION

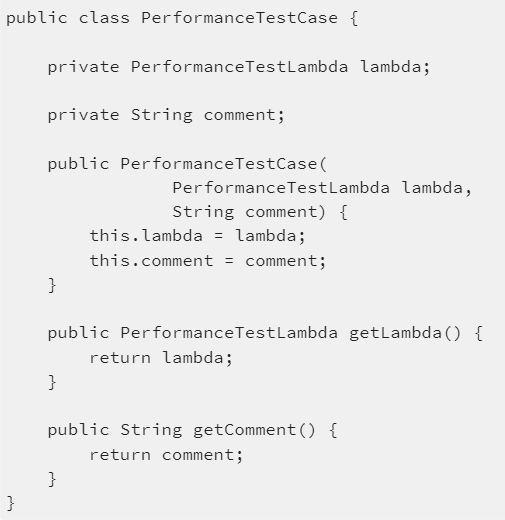
The following tests are implemented with the execute-around-pattern using Lambdas. For this we need a Functional interface:

Lab Exercise 15:

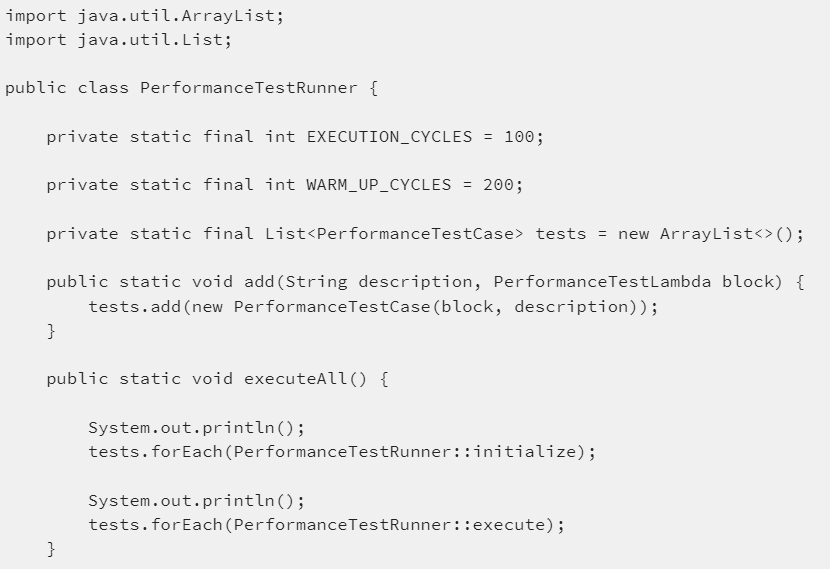
Create the following interface and classes.

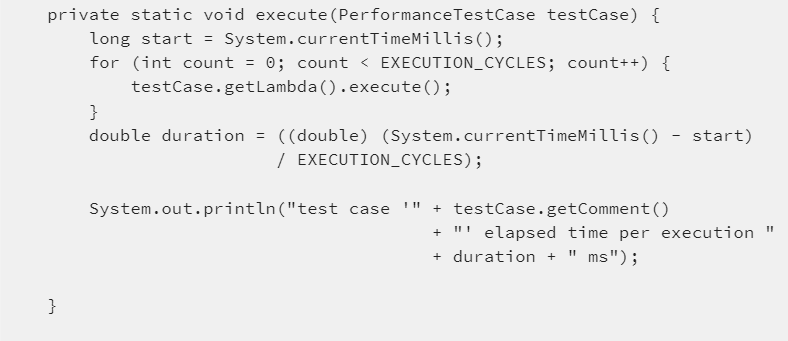


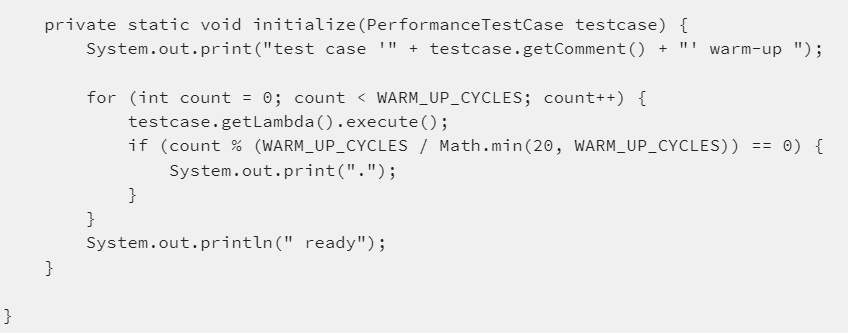
The helper class for the test cases:



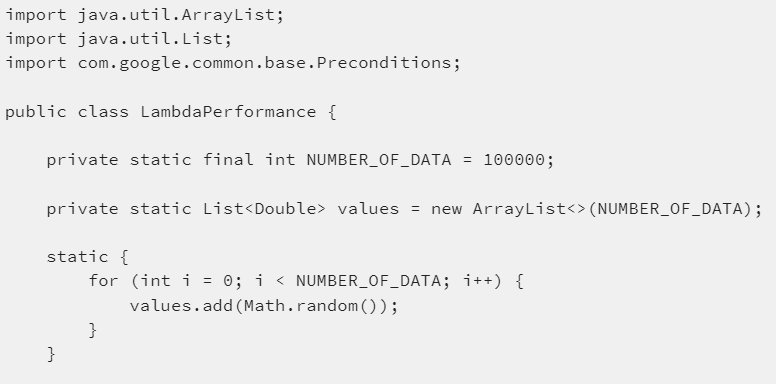
And the test runner:

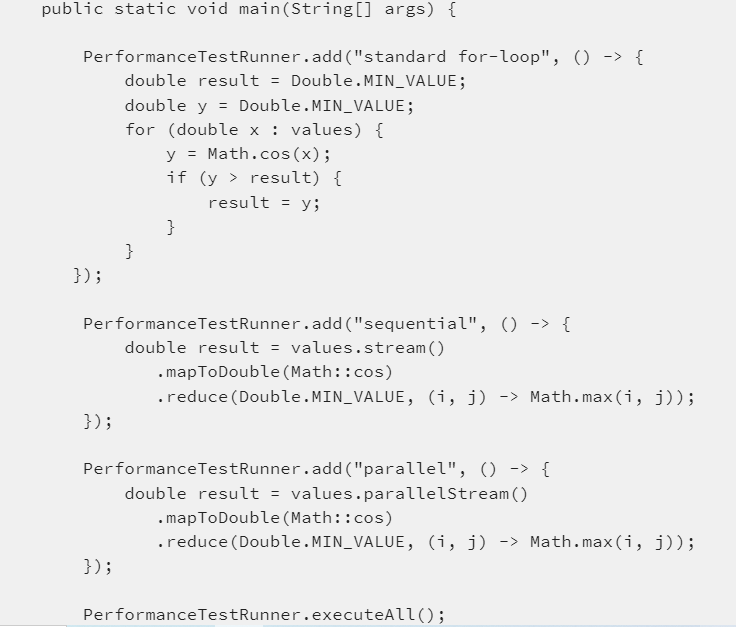






Then the following performance tests can be executed:







Expected output:

test case 'standard for-loop' warm-up .................... ready

test case 'sequential' warm-up .................... ready

test case 'parallel' warm-up .................... ready

test case 'standard for-loop' elapsed time per execution 5.68 ms

test case 'sequential' elapsed time per execution 6.1 ms

test case 'parallel' elapsed time per execution 1.79 ms

These results are not the same for all types of Collections, but in general two things are quite common for all collections:

* the sequential execution is slightly slower than a standard for-loop and
* the parallel execution may result in a significant performance improvement.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*