Functional Programming in Java

COMP40009 – Computing Practical 1

24th – 28th January 2022

Aims

- To practice writing immutable classes in Java.
- To gain experience with constructor *overloading*.
- To practice working with Java collections.
- To practice working with *streams*, *method references* and *lambdas* in Java, which enable functional programming.
- To gain further experience with the unit testing framework *JUnit*.

Problem

- Your first task is to implement two immutable classes, one to represent a *point*, the other a *rectangle*, and to implement various algorithms that work across collections of points and rectangles. You will implement the algorithms in an imperative style, using collections and loops, and in a functional style, using streams.
- You are then invited to write a number of methods, unrelated to points and rectangles, that utilize additional features of Java's `Stream` interface.
- You are given a suite of tests that you should run incrementally to help check the correctness of your solution.
- You will also write additional tests as you proceed through the various tasks below.

What to Do

Clone your exercise repository (remembering to replace *`login`* with your login) with:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2122_spring/javafunctionalprogramming_login.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[1]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -p51567BAF
```

[1] `https://www.7-zip.org/download.html`

(*n.b., 51567BAF is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.
If you list all the files in the newly checked out repository, you should see the following:

```
% ls -a javafunctionalprogramming
.  ..  .git  .gitignore  src  test
```

As in previous labs, these are:

- `.`, `..` – these refer to the current directory and its parent directory.
- `src` – the directory in which the Java source files for the exercise reside. The `src` directory contains two further directories, `rectangles` and `advancedstreams`, corresponding to Java packages.
- `test` – this directory contains a Java source file corresponding to a JUnit 4 test suite for the exercise. This is discussed further under **Testing**, below.
- `.git`, `.gitignore` – these contain the git repository information, and a list of filename patterns that should be ignored by git, respectively.

Testing

`TestSuite.java`, under `test`, contains a commented-out JUnit test suite. The code is separated into several sections, each initially commented out. As instructed below, you should un-comment these sections successively as you proceed through the various tasks of the lab, debugging your solution so that the tests pass. You should ensure that changes you make in subsequent tasks do not introduce *regressions* (cases where previously-passing tests now fail). This is called *regression testing*.

As part of the lab exercise, you will add additional tests to the test suite.

A note on coordinates

The exercise will involve writing classes to represent points and rectangles. As is traditional in computer graphics, we regard a point as being further to the right the larger its x coordinate is, and (differing from Euclidean geometry) further *down* the larger its y coordinate is.

Implementing a `Point` class

In the `rectangles` package you will find an empty class, `Point`. Fill out this class so that it models a point, represented by two integer coordinates that must be non-negative. Your `Point` class should be **immutable** and **fully encapsulated**.

The class should provide the following constructors and methods:

Public constructors

- A constructor that accepts two `int` arguments, representing the x and y coordinates of the point.
- A constructor that accepts no arguments, creating a point at $(0, 0)$.

- A constructor that accepts one `int` argument, x say, creating a point with coordinates $(x, 0)$.

A constructor can invoke another constructor by making a `this(...)` call. You should use this feature to minimise the amount of code associated with your three constructors.

Public methods

- `int getX()` and `int getY()`, which return the x and y components of the point.
- `Point setX(int newX)`, which returns a new point with the given value as its x coordinate, and the same y coordinate as the point on which `setX` is invoked. The parameter must be non-negative.
- `Point setY(int newY)`; analogous to `setX`.
- `boolean isLeftOf(Point other)`; returns *true* if and only if the point on which the method is invoked lies to the left of the given point.
- Analogous methods `isRightOf`, `isAbove` and `isBelow`.
- `Point add(Point vector)`, which returns a new point obtained by adding the coordinates of `vector` to the coordinates of the point on which the method is invoked. For example, if $(2, 7)$ is added to $(10, 12)$, the result should be $(2 + 10, 7 + 12) = (12, 19)$.

Testing, and adding more tests Un-comment the first block of tests, associated with the `Point` class. Debug your solution until they all pass.

At the end of `TestSuite.java` add **at least four** additional tests for the `Point` class. These tests should complement the existing test cases for `Point`, for example by testing edge cases or relationships between methods that are not already tested by the current tests for `Point`. Above each new test, write a brief source code comment explaining what it brings to the suite.

Implementing a `Rectangle` class

Your next task is to implement a class, `Rectangle`, modelling a rectangle.

- A rectangle has a top-left corner (x_1, y_1); a top-right corner (x_2, y_1), where $x_2 \geq x_1$; a bottom-left corner, (x_1, y_2), where $y_2 \geq y_1$; and a bottom-right corner (x_2, y_2).
- Each of x_1, x_2, y_1 and y_2 is a non-negative integer.
- A rectangle *contains* a point (a, b) if $x_1 \leq a \leq x_2$ and $y_1 \leq b \leq y_2$.
- The width of a rectangle is $x_2 - x_1$.
- The height of a rectangle is $y_2 - y_1$.
- It is permissible for either or both of $x_1 = x_2$ and $y_1 = y_2$ to hold.

Fill out the empty `Rectangle` class in the `rectangles` package to model a rectangle. It is up to you how this class is represented internally, but your design should avoid *redundancy*: you should not include a field if the value of that field could be computed from the values of other fields. Your class should be **immutable** and **fully encapsulated**.

Your `Rectangle` class should support the following constructors and methods:

Public constructors

- A constructor that takes three arguments: a `Point`, representing the top-left corner of the rectangle, and two `int`s representing, in order, the width and height of the rectangle.
- A constructor that takes two `Point` arguments. These represent two diagonally opposing corners of the rectangle, but which corners they represent is dependent on their values. For example:
  - Passing the points $(1, 5)$, $(3, 6)$ constructs a rectangle such that $x_1 = 1, x_2 = 3, y_1 = 5, y_2 = 6$.
  - Passing the points $(10, 1)$, $(10, 1)$ constructs a (single point) rectangle such that $x_1 = x_2 = 10$ and $y_1 = y_2 = 1$.
  - Passing the points $(10, 1)$, $(0, 10)$ constructs a rectangle such that $x_1 = 0, x_2 = 10$ and $y_1 = 1, y_2 = 10$.
- A constructor that takes two `int` arguments, `width` and `height`. If invoked with parameter values w and h, this constructs a rectangle such that $x_1 = y_1 = 0$, $x_2 = w$ and $y_2 = h$.

Challenge: see whether you can design your constructors so that only one of them actually assigns to the fields of a rectangle, and the other two each consist of a single invocation of `this(...)` to delegate construction to one of the other constructors.

Public methods

- `int getWidth()` and `int getHeight()`, which return the width and height of the rectangle.
- `Rectangle setWidth(int newWidth)`, which returns a new rectangle with the same top- and bottom-left corners and height as the original rectangle, but adjusted so that its width is equal to the given given parameter (with the top- and bottom-right corners correspondingly adjusted). The parameter must be non-negative.
- `Rectangle setHeight(int newHeight)`; analogous to `setWidth`.
- `Point getTopLeft()`, which returns a point representing the top-left corner of the rectangle.
- Analogous methods `getTopRight`, `getBottomLeft`, `getBottomRight`.
- `int area()`, which returns the area of the rectangle.
- `boolean intersects(Rectangle other)`, which returns true if and only if the given rectangle *intersects* with the rectangle on which the method is invoked. Two rectangles intersect if they contain at least one common point.

Testing, and adding more tests Un-comment the second block of tests, associated with the `Rectangle` class. Debug your solution until they all pass.

As you did with `Point`, at the end of `TestSuite.java` add **at least four** additional tests for the `Rectangle` class. Similar to your `Point` tests, these tests should complement the existing test cases for `Rectangle`. Above each new test, write a brief source code comment explaining what it brings to the suite.

Algorithms Using Collections

Your next task is to implement a number of methods that operate on *lists* of rectangles, and one method that creates a *map* from rectangles to integers. You will implement these in an imperative style by looping over lists.

Class `ListAlgorithms` contains declarations for each of the methods you should implement. Each is method is *static*. That is, it is not invoked on any particular object, operating only on the parameters that it is passed.

In the skeleton class, each method returns a dummy value, and has a "`TODO`" comment. You should remove the comment and dummy return statement, and replace the method body with appropriate code.

The expected behaviour of each method is described briefly with a JavaDoc comment. The JavaDoc comments are deliberately concise, and only provide details of parameters and return values (using the `@param` and `@return` annotations) where there might otherwise be ambiguity. You will also find at least one test case per method in the accompanying test suite; these tests also help to document what is expected of the methods.

Implement each method to provide the functionality described by the method name, JavaDoc and test cases. Do **not** make use of streams when implementing these methods.

You should find that many of the methods are very similar in nature, and that some of the later methods can be implemented with reference to the previous methods.

Testing, and adding more tests Un-comment the third block of tests, associated with the `ListAlgorithms` class. Debug your solution until they all pass.

Once again, at the end of `TestSuite.java` add **at least four** additional tests for the `ListAlgorithms` class to complement the existing test cases. Above each new test, write a brief source code comment explaining what it brings to the suite.

Algorithms Using Streams

Now you should re-implement each of the algorithms from the previous task, but this time so that they operate over *streams* of rectangles. This requires a *functional* style, applying functions to streams, and should involve no explicit loops.

Class `StreamAlgorithms` specifies the methods you should implement, providing a dummy return value and "`TODO`" comment for each. They are identical to the methods specified earlier in `ListAlgorithms`, except that they use `Stream` wherever `List` was used previously.

You will find the stream methods `map`, `filter` and `reduce` useful when implementing the `StreamAlgorithms` methods. In some cases you will find it convenient to pass lambda expressions as function arguments to `map`, `filter` and `reduce`; in other cases you may find it more appropriate to pass method references.

In the implementation of `getAreaMap`, you will need to use the method `Collectors.toMap` to create a map from `Rectangle` to `Integer`. `Collectors.toMap` requires two function arguments: a function from `Rectangle` to `Rectangle` to specify how a map key is computed for a given rectangle (**hint:** this is the simplest possible function), and a function from `Rectangle` to `Integer` to specify how an associated value is computed for a given rectangle.

Testing, and adding more tests Un-comment the fourth block of tests, associated with the `StreamAlgorithms` class. Debug your solution until they all pass.

Then un-comment the fifth block of tests, which check equivalence of behaviour between the `ListAlgorithms` and `StreamAlgorithms` classes. If you are curious, have a look at the

way these tests use the `forEach` method, which applies a given method to each element of a collection. Debug your solution until all these tests pass.

At the end of `TestSuite.java` add **at least four** additional tests for the `StreamAlgorithms` class to complement the existing test cases. If you wish, these may simply be variants of the new tests you already wrote to complement the `ListAlgorithms` tests, re-targeted for streams.

Intersecting rectangles

Recall that the `Optional<T>` class can be used to handle the scenario where an object of type `T` may be unavailable as the result of an operation.

Add to `Rectangle` a public method:

- `Optional<Rectangle> intersection(Rectangle other)`

This method should determine whether `other` intersects with the rectangle on which `intersection` is invoked. If they do indeed intersect, and `Optional` containing a rectangle representing the intersection should be returned. Otherwise, an empty `Optional` should be returned.

Add to `StreamAlgorithms` a public static method:

- `Optional<Rectangle> intersectAll(Stream<Rectangle> rectangles)`

This method should call `reduce` on the stream of rectangles to produce an `Optional` that contains the common intersection of all rectangles in the stream, if this exists, or an empty `Optional` otherwise.

Testing, and adding more tests Un-comment the sixth block of tests, associated intersecting rectangles. Debug your solution until they all pass. Add at least one more test for each of the `intersection` and `intersectAll` methods at the bottom of the test file.

Submission

As usual, use `git add`, `git commit` and `git push` to add and send your created **.java** files to the lab server.

Then, log into the gitlab server `https://gitlab.doc.ic.ac.uk`, and click through to your `javafunctionalprogramming_<login>` repository. If you view the `Commits` tab, you will see a list of the different versions of your work that you have pushed. Select the right version of your code and ensure it is working before submitting to CATe.

Extensions: exploring the Stream interface

The Java `Stream` interface contains a host of additional method for creating and manipulating streams.

As an extension to the lab exercise, here are a number of tasks that can be solved elegantly with reference to the additional methods that are provided by `Stream`.

Read the interface carefully:

`https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`

and think about how to use the various provided methods in implementing the following functionality.

Streaming cubes

In the `advancedstreams` package you will find an empty class, `CubeSupplier`. Change this class so that it implements the `Supplier<Integer>` interface. This interface requires there to be single method, `get()`, which takes no arguments and returns an `Integer`.

Implement `get` so that an instance of `CubeSupplier` returns the sequence of positive integer cubes—1, 8, 27, etc.—on successive calls to `get`. If so many cubes are requested that the next cube cannot be represented as a positive integer, `get` should throw a `NoSuchElementException`, which you can do via "`throw new NoSuchElementException();`".

Now add a *static* method, `cubeStream()`, to `CubeSupplier` that returns a `Stream<Integer>`, such that the elements of the stream are the positive integer cubes, in order. Look at the `Stream` interface to see how you can do this with minimal code, given what you have already implemented.

The trouble with `cubeStream()` is that it returns an endless stream of cubes, which will lead to a `NoSuchElementException` being thrown when an excessively large cube is tried. To overcome this, write another static method in `CubeSupplier` called `boundedCubeStream`, which takes two integer arguments, `start` and `end`. The `boundedCubeStream` should return a stream of positive cubes of length `end-start`, starting with the (`start`+1)th positive cube. For example, `boundedCubeStream(3, 7)` should return the stream of cubes containing 64, 125, 216, 343. You should find some helper methods in the `Stream` interface that make this easy.

In the `IsPalindrome` class, implement the `isPalindrome` method, which should return true if and only if the given string argument is a palindrome. (This has nothing to do with streams.)

Finally, in `CubeSupplier`, write a method, `palindromicCubes`, that takes the same arguments as `boundedCubeStream` and behaves identically to `boundedCubeStream` except that it returns only those cubes that are palindromes when represented as decimal strings.

Testing, and adding more tests The seventh block of tests provides tests for the above functionality, which you can use to help debug your solution.

String prefixes

Your task here is to implement the three methods specified in the `advancedstreams.StringPrefixes` class, according to their JavaDoc descriptions. Try to write your code as compactly as possible by exploiting the functionality provided by the `Streams` interface. (Note that this is an exercise in using the `Streams` interface; it is not necessarily a good idea to always go for the most compact form of code in practice.)

Testing, and adding more tests The eighth block of tests provides tests for these methods, which you can use to help debug your solution.

Assessment

F - E: Very little to no attempt made.
Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
solutions may not be correct, or may not have a good style.

B: Implementations of all functions attempted, and solutions are mostly correct. Code style is generally good.

A: There are no obvious deficiencies in the solution or the student's coding style. In addition, there is evidence of productive testing.

A*: As for an A -- plus the student has done additional work beyond the basic spec, e.g. by considering (and clearly commenting) interesting variations or extensions to the given functions; e.g. based on their own research.