**CS2030 Programming Methodology**
Semester 1 2020/2021

7 October 2020
Problem Set #6 Functional Interfaces and Lambda Expressions

1. What is the difference between
   (a) `System.out.println(‘‘CS2030’’)`, and
   (b) `()->System.out.println(‘‘CS2030’’)` ?

   Explain in terms of the data type and the execution of the printing.

> **Solution:** (a) is a statement, and has no return value (ie. void). It is executed immediately.
>
> (b) is a lambda expression that immediately evaluates to a closure. This lambda (an anonymous function) has no parameters, and its body is the `println` statement. It is equivalent to the following function:
>
> ```
> void foo() {
>     System.out.println("CS2030");
> }
> ```
>
> When Java encounters the above, the function `foo` is immediately defined. Its body is executed only when you invoke the function with `foo()`. Likewise, the body of the lambda is executed only when the lambda is invoked. In other words, the lambda *delays* the execution of its body until a future time.

2. Consider the `Point` class.
   (a) Complete the code below to provide the `add` and `scale` methods.

```
public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }

    public Point add(Point q) {
        // insert code to return a new point that is the
        // sum of this point and point q. Just sum the
        // corresponding x and y coordinates.
    }
```

```
    public Point scale(double k) {
        // insert code to return a new point whose x,y
        // coordinates are k times those of this point.
    }
}
```

**Solution:**

```
public class Point {
    // ... previous code

    public Point add(Point q) {
        return new Point(this.x + q.x, this.y + q.y);
    }

    public Point scale(double k) {
        return new Point(this.x * k, this.y * k);
    }
}
```

(b) Recall the generic `average` function:

```
import java.util.function.BinaryOperator;
import java.util.function.BiFunction;

<T> T average(List<T> list, BinaryOperator<T> add, BiFunction<T,Double,T> scale) {
     T sum = list.get(0);    // assume list is non-empty
    int size = list.size();
    for (T item : list.subList(1, size))
        sum = add.apply(sum, item);
    return scale.apply(sum, new Double(1.0/size));
}
```

Provide appropriate lambdas in the code below to compute the average of the three points.

```
List<Point> list = new ArrayList<>();
list.add(new Point(1.0, 1.0));
list.add(new Point(1.0, 2.0));
list.add(new Point(-1.0, 1.0));
average(list, ??, ??);
```

**Solution:**

```
average(list, (x,y) -> x.add(y), (x,k) -> x.scale(k));

// OR, using method references

average(list, Point::add, Point::scale);
```

(c) Now, in the `Circle` class, write an `add(other)` method that returns a new circle whose centre is the sum of the centres of this and `other` circle, and whose radius is the sum of their corresponding radii.

Likewise, write a `scale(k)` method that returns a new circle whose radius and centre are scaled by `k`. Finally, create a list of three circles with different centres and radii,

and find the circle whose centre lies at the average of the centres of the three, and whose radius is their average radii. Don't forget to override the `toString` method in `Circle` for displaying it.

```
public class Circle {
    private final Point centre;
    private final double radius;

    public Circle(Point c, double r) {
        this.centre = c;
        this.radius = r;
    }
    //Don't change the above, but insert code here
    //for add(other) and scale(k) methods
}

List<Circle> list = new ArrayList<>();
list.add(new Circle(new Point(1.0, 1.0), 1.0));
list.add(new Circle(new Point(1.0, 2.0), 4.0));
list.add(new Circle(new Point(-1.0, 1.0), 2.0));
average(list, ??, ??);
```

**Solution:**

```
public class Circle {
    // ... previous code
    public Circle add(Circle other) {
        return new Circle(this.centre.add(other.centre),
                          this.radius + other.radius);
    }

    public Circle scale(double k) {
        return new Circle(this.centre.scale(k), this.radius * k);
    }
}

// .. previous code
average(list, (x,y) -> x.add(y), (x,k) -> x.scale(k));

// OR, using method references

average(list, Circle::add, Circle::scale);
```

3. Read the Java documentation of the `andThen` method in `java.util.function.Function`.

   (a) It is possible to compose functions without using `andThen`. Complete the code below to do so:

```
import java.util.function.Function;

<T,U,R> Function<T,R> compose(Function<T,U> f, Function<U,R> g) {
    // Insert code here to return a function h, such that
    // h(x) = g(f(x))
}
```

**Solution:**

```
<T,U,R> Function<T,R> compose(Function<T,U> f, Function<U,R> g) {
    return x -> g.apply(f.apply(x));
}
```

(b) Compose the given functions `f` and `g`, and apply it to 5 to get 49 as the result. What if you wanted a result of 729?

```
Function<Integer,Integer> f = x -> x+2;
Function<Integer,Integer> g = x -> x*x;
```

**Solution:**

```
compose(f,g).apply(5); //since 49 = (5+2) * (5+2)

compose(g, compose(f,g)).apply(5); //since 729 = (5*5 + 2) * (5*5 +2)
// ALTERNATIVELY,
compose(compose(g,f), g).apply(5);
```

(c) Suppose these two methods are defined in the `Point` class:

```
public double distanceTo(Point q) {
    return Math.sqrt(sumOfSquares(this.x - q.x, this.y - q.y));
}

public static double sumOfSquares(double a, double b) {
    return a*a + b*b;
}
```

Run the following code in JShell to see what happens. Explain. (Note that the `var` keyword may be used to denote "some suitable type". The Java compiler will infer the type of the variable based on the right hand side of the assignment. Also note the use of *method references*.)

```
double howFarFromHere(Circle c, Point here) {
    return compose(Circle::getCentre, here::distanceTo).apply(c);
}

var c = new Circle(new Point(3.0, 4.0), 1.0);
var p = new Point(0.0, 0.0);
howFarFromHere(c, p);
```

**Solution:** This computes the distance from the centre of circle `c` to the point `p`. The `compose` returns a function that accepts a circle `c`. When invoked (via `apply`), this function calls `c.getCentre()`. The result of this is then passed to the `here.distanceTo` method.

Note that this style of coding obscures the intention, and is only meant to show how `compose` may be used to express the idea of calling one function after another. It is *much clearer* to say:

```
double howFarFromHere(Circle c, Point here) {
    return here.distanceTo(c.getCentre());
}
```

4. *Currying*[1] is the conversion a function of two arguments into two functions, each taking one argument, such that their sequencing computes the same result as that of the original function.

   For example: suppose $f(x, y) = x + y^2$. Define $h_x(y) = x + y^2$, ie. it is a function of one argument $y$, since the value of $x$ is fixed. Also define $g(x) = h_x$. Then clearly, $f(x, y) = h_x(y) = g(x)(y)$. We say that $curry(f) = g$. This idea is readily expressed as:[2]

```
import java.util.function.BiFunction;

<X,Y,Z> Function<Y, Function<X,Z>> curry(BiFunction<X,Y,Z> f) {
    return y -> (x -> f.apply(x,y));
}
```

   (a) What is the result of executing the following code, and how does this compare with slide 19 of the lecture notes on Functional Interfaces?

```
Function<Integer, Function<Integer, Integer>> addN = curry( (n,x) -> n+x );
Function<Integer, Integer> add5 = addN.apply(5);
add5.apply(7);
```

   (b) What is the result of: `addN.apply(10).apply(7)` ?

   (c) What should `???` be in: `??? mystery = curry(Point::sumOfSquares);`
   That is, what's the type of `mystery`? Note that `sumOfSquares` was defined in Q3(c).

   (d) Show how you would use `mystery` to calculate $3^2 + 4^2$.

**Solution:**

(a) The result is 12. The difference is only slight: here, you invoke `addN` as `addN.apply(5).apply(7)`, whereas in slide 19, you use `addN(5).apply(7)`.

(b) 17

(c) The type is: `Function<Double, Function<Double, Double>>`

(d) `mystery.apply(3.0).apply(4.0)`

[1] Named after Haskell Curry. See `https://en.wikipedia.org/wiki/Currying`

[2] Look up `BiFunction` in the Java documentation.