Introduction

Given that COMP 440 often provides Java code to students, and that solutions to some COMP 440 problems may be computationally intensive, this document will overview some useful features of Java that may help students to write concise and efficient code.

1 Anonymous Inner Classes

Java provides Anonymous Inner Classes to allow programmers to instantiate classes that do not merit their own class definition. Imagine a relatively simple class that requires five or seven lines to define, and will be used very infrequently (or maybe only once!). Instead of making up a name befitting the class' specific and limited use (`TwoDimensionalEuclideanDistanceComparatorFactory` — yuck!), and giving it its own source file, the programmer could simply supply an anonymous inner class when needed.

Anonymous inner classes necessarily derive from some other super class or interface — this is necessary not only in the context of Java's type system but also syntactically. When creating a new anonymous inner class, the programmer begins as though he or she were merely instantiating the super type (even if the super type were a non-instantiatable type, such as an `abstract class` or `interface`). The programmer must choose one of the super type's available constructors and provide parameters for it.

The programmer can then provide implementations for abstract methods inherited from the super type, or define their own methods. Note however that no new methods will be visible outside of the anonymous inner class definition. I will supply a brief example, then explain that last point. Refer to Figure 1 on page 2 now.

Note that in Figure 1 on page 2, I could simply have created the anonymous `Comparator` inside the parameter list for the invocation of `Collections.sort(...)`. Instead I put it in a variable `c`, in order to highlight another detail: despite the public method `getDistance(...)` that I added to my anonymous `Comparator`, the commented call to `c.getDistance(...)` would not have compiled if it had been part of my code.

This is caused by the type of `c`: it is a reference to a *`java.util.Comparator`*, and the method `getDistance(...)` is not defined for the type `java.util.Comparator`. Under other circumstances, we might be willing to cast `c` to something closer to its true type to get access to the `getDistance(...)` method. Here, however, that is impossible, as `c`'s type has no name, since it was created anonymously.

2 `PriorityQueue` and `Comparators`

Some algorithms require that sorted order be maintained in their data. One might accomplish this by repeatedly sorting an `ArrayList`, but this is somewhat inefficient.

The `PriorityQueue` class provides a list-like collection of elements that are kept in sorted order at relatively low computational cost.

3 Thread Pools

Although it is possible to perform intricately structured parallel execution using Java threads, a very common practical goal is simply to evenly distribute a large number of roughly similar tasks among a fixed number of threads. That is, given n tasks of relatively equal complexity, and m threads, the goal is to queue the tasks so that each thread need only perform $\frac{n}{m}$ tasks, thereby reducing the time required to complete the entire computation from t to $\frac{t}{m}$ in the ideal case.

Java provides an easy and painless way to accomplish this goal: thread pools. Instead of manually spawning and herding threads into performing different tasks, the programmer can call on Java to abstract many such details. Java will manage the threads and distribute the work among them, if the programmer is able to represent each task as a Java object.

```
public class SortPointsByProximityToAnchorExample {
    public static void main(String[] args) {
        List<Point> points = new ArrayList<Point>();

        // implementation omitted; assume points is populated with data.

        final Point anchor = new Point(10.0f, 10.0f);

        /*
         * In this next line, I invoke points.sort(...), and *in that same
         * line*, I provide the Comparator instance that will be used. Note
         * how I begin by "instantiating" a comparator, then provide methods
         * inside the curly-braces.
         *
         * I am choosing to sort the elements in points by their proximity to
         * the anchor point. This is another useful feature of an anonymous
         * inner class, called a "closure". The programmmer is allowed to use
         * variables in scope at the time of the creation of the anonymous
         * inner class in the class itself. There are a few restrictions, see
         * ***** FIND URL FOR THIS ****** for more details.
         */
        Comparator<Point> c = new Comparator<Point>() {
            @Override
            public int compare(Point one, Point another) {
                Float distanceToOne = getDistance(anchor, one);
                Float distanceToAnother = getDistance(anchor, another);

                return distanceToOne.compareTo(distanceToAnother);
            }

            public Float getDistance(Point a, Point b) {
                float dx = a.x - b.x;
                float dy = a.y - b.y;

                return Math.sqrt((a * a) + (b *  b));
            }
        }

// Note that this won't work:
// c.getDistance(new Point(0, 0), new Point(1, 1));

        Collections.sort(points, c);
    }
}
```

Figure 1: Creation of Anonymous Inner Classes.

```
public class ListSortTask<T extends Comparable<T>> implements Callable<Void> {
    private List<T> list;

    public ListSortTask(List<T> list) {
        this.list = list;
    }

    public Void call() {
        /* This will sort the list in-place. */
        Collections.sort(list);

        return null;
    }
}

public class ManyListsSorter {
    public static void main(String[] args) {
        /* Make 100 lists of random numbers. */
        Random random = new Random();
        List<List<Integer>> lists = new ArrayList<List<Integer>>();
        for (int listnum = 0; listnum < 100; listnum++) {
            List<Integer> l = new ArrayList<Integer>();

            /* Populate l with random data. */
            for (int index = 0; index < 1000; index++) {
                l.add(random.nextInt());
            }
        }

        /* Reperesent the tasks to be done as objects */
        List<ListSortTask<Integer>> sortTasks = new ArrayList<>();
        for (List<Integer> list : lists) {
            sortTasks.add(new ListSortTask<Integer>(list));
        }

        /* Create a thread pool with 5 threads */
        ExecutorService threadPool = Executors.newFixedThreadPool(5);

        /* Give the list of tasks to the thread pool */
        try {
            threadPool.invokeAll(sortTasks);
        } catch (InterruptedException e) {
            // Won't happen unless the user manually interrupts.
        }

        /* All lists are now sorted. */
    }
}
```