CMSC420 Project - Spring 2017
Slushie Version 2.1
Due max(syllabus, submit server)

The BIG 420 Project[*]

Parts 2 will be due on max(syllabus, submit server) at 23:59 hours (plus 48 hour grace

Last Modified February 3, 2017

Contents

[*]Participation in this project may prove HAZARDOUS to your health. Unfortunately, failure to participate early and often will definitely adversely affect your GPA. Take my advice. Start now, because you're already behind. If you don't believe me, ask anyone who took CMSC 420 with Dr. Hugue.

1 Introduction and General Overview

Welcome to the Spring 2017, edition of Dr. Hugue's CMSC 420 project. A substantial portion of your grade in this course will be determined by your performance on several programming assignments (collectively referred to as *The Project*. The time commitment required on your part varies from student to student, but expect to spend a good bit of time planning as well as coding and debugging each part of the projects.

The primary motivation of this project is to give you the experience of building portions of a real world application using the data structures and algorithms that we will study during the semester. This semester's long term goal is to mimic some of the functionality of MapQuest [1]. By the end of this course, if you have worked hard and done your job well, you will have a program that is capable of drawing maps of a general area and supporting

[1]Copyright 1996-2016 MapQuest.com, Inc ("MapQuest"). All rights reserved. See www.mapquest.com for more information.

the following functions: displaying a highlighted route; calculating shortest routes (based on time or distance); generating driving instructions (complete with correct “turn left and then go straight for 2.34 miles” annotations); determining closest points of interest, such as all of the “Internet Cafes” within a 20 mile radius of College Park, MD; and just generally impressing friends and family with its awesome ability to tell you how to get to where you want to be in both plain text and picture form. And, even if things don’t go smoothly, you will have had the opportunity to think about data structures, and programming in general, in new and interesting ways.

The project comprises three parts, which build upon one another to produce the desired features of an on-line mapping system. Part 1 will establish a custom database, including insertion, deletion, and search functions using elements of the JAVA API, and, of course establishing standard notation for inputs and outputs, based on XML. In later parts, you will be replacing elements of the JAVA API with custom code based on structures other than those used in JAVA. MeeshQuest functions will be added with each part, resulting in a very powerful piece of software at the end of the semester.

1.1 Warning and Encouragement

This is a large project that has been carefully designed over several years to provide a challenge to every level of programmer in the class. Each part will take the full time you have been allotted to complete. (In all honesty, many of you will feel that we have not given you enough time–yes, even you hot shots.) But don’t panic! There are plenty of ways to complete each part on time for full credit.

Here’s the secret: start early and start by spending a few days sketching out what objects/classes you will use and how these objects/classes will fit together. You are encouraged to discuss your ideas with classmates (but don’t write any code together!) and to bring your design ideas to the TAs during office hours, or on Piazza, for comments/suggestions. The most successful students have spent time carefully planning their projects. Those students who do not devote any time to design are the one most likely to receive a poor grade.

Many of you have probably been able to sit down and do many of the projects in the lower level classes in an evening or two with little or no planning. Well the honeymoon is over. If you don’t start early and spend time carefully planning each part, you will be very hard pressed to get a fully working solution. Good design and testing, not Mountain-Dew-Code-A-Thons the night before the due date, is the key to surviving 420.

Despite the quantity of work required to implement these projects, most students, when finished with CMSC420, agree that the experience made them better programmers. The rest of your projects this semester will be implemented in the Java programming language. It is possible to go from having almost no Java knowledge to adding ‘professional-level Java developer’ to your resume in a single semester by a single course. Not only will you learn to exploit the Java API data structures you’ll be implementing, we’ll be providing some tips, tricks, and general advice about how to write not only good Java code but how to develop good object-oriented design in general. So, don’t let the programming component of the course scare you off; it will be worth it in the long run–we promise.

Each part will typically involve two major components: coding a functional data structure and actually putting it to use. For example, in the past we have had students implement a Fibonacci heap, and use it as the priority queue component of Dijkstra's algorithm. The first project will require that you write what has come to be known as a *command parser*, which will allow us to pass input to your program as a series of commands (e.g., "add (x,y) to your B+ tree"). Each part will introduce a new set of commands you will need to handle, so updating your command parser will be another component of each of your projects.

1.2 Freezing the Specification

In the real world, as well as in Dr. Purtilo's 435 course and others, the specification can be changed at any time, including the night before the project is due. In CMSC 420, that is not the case. By rule, the specification has to be frozen no earlier than one week before it is due. This is to allow those of you who really don't get it to screw up even more by waiting until the spec is frozen. Just kidding! It is to reassure those of you who freak out because everything that Dr. Hugue does looks totally disorganized that evaluation opportunities in this course are well regulated and rule based. By rule, we have to abide by the letter of the spec in grading your project. If we don't, you get the points back. By Rule.

Note that updates to submission instructions and output syntax are exempt from this freezing rule. However, typically no modification can occur within the last 24 hours before the due date/time without your receiving an extension. Thus, when it comes to grades, you can count on us to be equal opportunity abusers.

1.3 Giving and Seeking Help–OK within Reason

Unlike many other courses at this university, we in CMSC420 believe that you should be free to talk at length about the project with each other. The algorithms you'll be writing to implement data structures are not secrets; in most cases you will have pseudocode or even actual code available to you through many resources. Piazza is mandatory reading in this course and is the perfect place to ask questions and get answers about the project, Java, the data structures, the meaning of life, etc.

Do note, however, that even though our policy is more lenient than other professors in their courses, we do take academic honesty quite seriously and students have received XFs in this course for blatantly copying other students' code. It's one thing to develop an algorithm with another student—it's another to copy and paste your buddy's code. We will still be doing the standard code comparisons not only between your code but also the code of students of semesters past (yes, we keep it). So don't try any funny stuff, and you'll be fine. The general rule of thumb: when in doubt, cite your resource via in-code comments and notify the professor and ask permission.

1.4 I/O Format: XML

The last important matter of business is to mention that all input and output for your command parser will be in XML (eXtensible Markup Language). If you've never heard of XML, ask your friend Google about it, and you'll be in no short supply of information, especially since the newest version of MS office replaces the `.doc` format with XML as the default. This is because XML offers a couple of major advantages to a standard text-based command parser; the first is that it is largely more relevant to a data structures course since the structure of XML is a general tree. The second is that XML is popping up all over the IT industry, and chances are that you will be dealing with XML at your place of employment. It's becoming the new standard for information exchange, so it's a good thing to be learning. Another great thing about XML is that, in the past, students were required to error-check the commands. Dr. Hugue is a dependability expert, and she believes strongly in writing dependable code; a malformed text command should not crash your program. XML is easily validated (both syntactically and structurally); so you can use pre-existing and readily available tools to confirm that both the input and output XML is error-free.

As of Java 5.0, Java contains many interfaces to process and validate XML. You are encouraged to take advantage of them. We will provide you with a Java class called `XmlUtility` which does most of the heavy lifting for you. It is discussed in another section.

1.5 Disclaimer

No programmer is perfect. No guarantee is made that bugs do not exist in the code that is provided. However, most of the code has been tested extensively; so, check your code as well as any code that is supplied. Reporting and, perhaps, correcting bugs in supplied code is just one way to improve the quality of the course and earn brownie points with Meesh. Of course, we have no plan to support validation of outputs except via correctness as tested on the submit server.

2 General notes on Java

By popular request, the notes on JAVA remain in the document, but have been relegated to the Appendix. They were not intended to be confusing to those of you who consider yourselves JAVA programmers, or to indicate required processes. They were merely included to provide a bridge to folks who had their only JAVA experiences in CMSC 330.

3 MeeshQuest Components

There are four major components to this project each of which will upgraded with each part: a dictionary data structure, a spatial data structure, an adjacency list (not used in Part 1), and a mediator.

3.1 Dictionary Data Structure

A dictionary data structure is one which is capable of storing objects in sorted order based on key such as a string or an integer. For instance, you might have several hundred City objects which consist of the name of the city, the latitude and longitude at which it is located, and the radius or expanse of the city limits. One way of storing these cities is to sort them by name; another is to store them in decreasing order by population; yet another is in increasing order by latitude.

To manage a structure based on the names of cities, you would not need a comparator for cities; but, rather, since the keys are city names (strings), your comparator would need to handle `strings`. So to compare cities by name in Java 8.0, your comparator might look like this:

```
public class CityNameComparator implements Comparator<String>
{
            public int compare (String s1, String s2) {

                     (however you want to do string comparison)
                        return

            }

}
```

Specifically in this project we'll use a dictionary to hold and sort cities and attractions by name and a spatial data structure to hold and sort by coordinate.

3.2 Spatial Data Structure

A spatial data structure is one which is capable of storing and sorting objects based on multidimensional keys. The two-dimensional structures, such as the PR quadtree and PM quadtrees, can store objects based on two values, such as the longitude and latitude (x, y) of a city. In Java this is not a big deal, but in other languages, such as C and C++, it is. Note that latitude lines are horizontal, north and south of the equator, and correspond to lines with a fixed `y` coordinate, such as $y = 30N$. Longitude lines are vertical, east and west of Greenwich, England, and correspond to lines with a fixed `x` coordinate, such $x = 40W$. Three-dimension structures, such as a K-d tree or a PM octree, can store objects based on three values, such as the longitude, latitude, and sea level of a city (x, y, z).

3.3 Mediator

A Mediator is a design pattern that is described in the famous book *Design Patterns* by Gamma et Al, also referred to as the Gang of Four (GOF) book.[1]

The intent of a Mediator is to **define** an object that encapsulates how objects within a set interact. Mediator promotes a loose coupling among objects by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently[2]

In other words, the idea is to have one or more objects (hint: more!) which are capable of reading in commands from the input stream and executing them by calling the right functions in the dictionary, spatial, and adjacency list data structures (for Parts 2 and 3) to perform the requested action(s).

Minimally, the Mediator could be a class named CommandParser which would read commands from the standard input stream, parse the data, pass it onto the correct component for further processing, analyze the return values from this component, print the correct success or failure message back to the user, and then loop until all commands have been processed.

It would be wise (hint hint) to break this functionality into several classes which perform one or more of these tasks. These objects, when combined together, form the abstract notion of a Mediator for our MeeshQuest program.

4 Roadmap

This is a roadmap of the major component that we will use in each part of the project. An asterisk (*) indicates that we will be reusing the structure from the previous project with little or no modification. "i" stands for `insert`, "f" stands for `find`, and "d" stands for `delete`. (The command parser will have new commands added with each project, but the overall design need not change after project 1 unless you want to make it more efficient or elegant.)

Part	Dictionary	Spatial	Adj. List	Application
1	Treemaps (i/f)	n/a	n/a	n/a
2	Treemaps (i/f/d)	PR Quad (i/f/d)	n/a	n/a
3	AVL-G (i/f)	PM3 Quadtree (i/f)	Any (i/f/d) in $O(\log n)$	Dijkstra/Shortest
4	AVL-G (i/f/d)PR of	PM1/3 Quadtrees (i/f/d)	*	MST

****NOTE: This roadmap is subject to change as the semester progresses.****

5 Part 1: Comparators, Treemaps, and Cities

For the first part of your project, you will implement a data dictionary that supports both city names and city coordinates as keys. You will also need to write an interpreter that will be able to handle basic XML commands. Your data dictionary can be written by merely playing games with comparators, thereby convincing a good old `TreeMap` or `TreeSet` to act like it's something else altogether.

[2]While this design pattern is presented on page 273 of [1], a Google search using "design patterns mediator" is cheaper than buying the book.

Important: The only commands you must implement for part 1 are createCity, listCities, and clearAll.

6 Part 2: PR Quadtrees and Range Searches

The second part of your project will expand upon the basic functionality implemented in the first part. Commands will require you to insert verified cities into the spatial map, and to delete them from the spatial map. The role of the spatial map is to support range searches where, given a location in 2-d space and a radius, you will find all the cities within that circle, including on the border. These types of operations are allegedly not efficient using the treemap of coordinates; however, I have my doubts and I might even let you prove it to me some day.

6.1 Data dictionaries and notes about Cities

We have alluded to the fact that you are going to have to maintain a dictionary (a collection) of all the cities we create using the `createCity` command. Generally speaking, you are going to want to be able to access cities by their names in logarithmic time. You can use a `TreeMap` for this, where the keys are the cities' names and the values are the city objects themselves. This will allow you to get the cities' (x, y) coordinates based on their names very quickly. You also need to be able to find cities based on their (x, y) coordinates, since aside from two cities bearing the same name being illegal, two cities cannot occupy the same (x, y) coordinate. You'll need a way to sort points in 2D space in such a way that they can be stored in a `TreeMap`. Remember reading about comparators?

You can write a comparator that will sort (x, y) coordinates in a useful way: sort on the y coordinate first; if two cities have the same y coordinate, compare their x coordinates to determine their final ordering. A sample ordering:

```
(0,0)
(1,0)
(100,0)
(0,1)
(1,5)
(100, 100)
(0,1000)
(5,1000)
(50,1000)
```

Keep in mind: cities will always have integer coordinates. In fact, we will only use integer values as inputs to your projects. However, you may find it useful to convert them to floats or doubles in later project parts.

For every city created, you'll need to add it to two dictionaries: the first maps strings to cities (name to city object) and the second would map city objects to strings (names).

However, if you write your City class such that the city's name is one of its fields, then using maps doesn't seem to make any sense because the keys contain all the info you need (but keep reading). You could instead use sets. The first set, which sorts by name, uses a CityNameComparator which compares two cities just based on the default string comparison between their two names, and the second which uses a CityCoordinateComparator which compares two cities based on the (x, y) ordering discussed above. However, there is one minor problem with using sets: once you put something into a set, you can't easily get it out again based on some other data type. In other words, suppose I pass you a string s, which I claim is the name of a city in your set. If you modified your comparator to allow strings to be passed to the compare() method, the best you could do is tell me that the city with that name either exists or doesn't exist—you can't get the actual city object back out of your set in better than linear time. So at least for the names, you definitely want to use a Map despite the fact that a Set could sort them for you. The coordinate dictionary will probably only be used to check containment, so in that case a set is probably sufficient.

Here I go again alluding to this mysterious City object. Yeah, the major data type we'll deal with throughout this project is a colored, named point in 2D space which we will be calling a City. Yes, you will have to write some class to store this information. A subjugate data type which you will probably also need to implement is a line segment in 2D space, which we're calling an Road. You'll have to do geometric computations involving roads and cities (specifically distance). Boo. Fortunately, Java has already done that for you. If you look at the java.awt.geom package, you'll see two useful classes: Point2D and Line2D. Both of these are abstract classes, but each contains two inner classes that are concrete implementations of their enclosing classes: Point2D.Float, Point2D.Double, Line2D.Float, and Line2D.Double. As you've guessed, the .Float and .Double refer to the precision in which their coordinates are stored. There's no Point2D.Integer, but you can just use the Float version. We'll never pass you a point whose coordinates have more than 23 significant digits (thus subjecting you to a precision error due to the limitations of Float).

The best way to implement a City is to have your class extend Point2D.Float, and add data members (strings) for name and color. Note that Point2D defines two public fields, x and y, which store the coordinate data. Note: do *not* redefine an x and y in your City class if you extend Point2D.Float. Most Java compilers will allow you to create fields with the same names as fields in the parent class, but good editors like Eclipse will warn you about it. The problem with redefining your own fields is that the Point2D class has already implemented all the geometric computations you need to worry about, but those methods use the x and y as defined in the Line2D.Float class. If you redefine x and y in City and fail to set the x and y in the parent class (by saying super.x = and super.y =) your geometric computations will never work because all your points will be treated like (0.0f,0.0f). The same goes for Line2D and its x1, y1, x2, y2 fields.

You should avoid the urge to make your Citys implement the Comparable interface since I've already described two obvious ways to sort them and there are probably more. It's better to force your users to provide a Comparator than run the risk of them expecting one ordering and discovering another. `Comparator`s are quick and easy to write and prevent any possible confusion on how they'll turn out when sorted.

6.2 Drawing a spatial map using CanvasPlus

We have developed a Java applet to visualize what you are doing. Mapping commands will interact with the visual map you are building, and two commands will be able to print out what the map looks like. Everything you need to build the map properly is discussed below.

To create a new visual map of dimensions spatialWidth x spatialHeight:

```
CanvasPlus canvas = new CanvasPlus("MeeshQuest", spatialWidth, spatialHeight);
```

To initialize the map and make sure it looks correct, we need to add rectangular bounds to the map since the map gives us a small border. The rectangle should be black and unfilled.

```
canvas.addRectangle(0, 0, spatialWidth, spatialHeight, Color.BLACK, false);
```

To add a named point to the map (despite the fact that cities have color and radius attributes we will always be drawing them as black points):

```
canvas.addPoint("name", x, y, Color.BLACK);
```

To remove a point works the same way:

```
canvas.removePoint("name", x, y, Color.BLACK);
```

To add other shapes (line segments, circles, rectangles, etc) the syntax is similar. See the javadoc for CanvasPlus. For example, to add a blue, unfilled circle:

```
canvas.addCircle(x, y, radius, Color.BLUE, false);
```

You will also be required to add a cross (a horizontal and a vertical line partitioning a rectangular section into 4 equal sections) where your PR Quadtree is partitioned (i.e. at each internal node). The cross should follow the rules of the PR Quadtree correctly. A cross should exist for each internal node of the PR Quadtree. The cross should be black, centered at the internal node's spatial center, and the radius of the cross should not exceed the spatial bounds of the internal node.

Examples will be provided for all visual components of the project.

To save a map to an image file:

```
canvas.save("filename");
```

Important: After calling the save method, CanvasPlus is still running. Remember to call the map's dispose() method when you are finished processing all the input. Otherwise your program will keep running.

The drawing packages are provided for your benefit. You are encouraged to use them when testing your PR Quadtree. In that case, you could simply call the CanvasPlus draw() method to open a window displaying the current map.

6.3 A command interpreter for XML

If you haven't heard of XML yet, now is a good time to read up. XML stands for extensible markup language and is quickly becoming the standard for textual data representation. If you haven't used XML yet at work or for another class, you will probably see it soon. XML is a tag-based hierarchical organization of data (i.e., a data structure). If you want to look at an XML document through a data structures lens, an XML document is a general tree whose nodes are either named tags or blocks of text. A great site with great tutorials that cover the basics of XML is available at www.w3schools.com.

Google will also tell you everything you want to know about XML—it's so widespread at this point that there are probably hundreds of tutorials already written–and this document, like my exams, is already too long.

For your project, your input and output will be in XML format. XML is convenient because it is designed to be validated (in other words, checked for correctness). Every XML document can optionally include a reference to its W3C XML Schema (commonly referred to just as XML Schema) which is an external XML document that contains the rules for what elements an XML document can contain. In other words, a schema defines the rules for the XML structure. Thus, by validating an XML document against a schema, you can quickly determine whether or not the XML contains certain kinds of errors. If you happen to have handy a solid XML parser and a solid schema validator, you can eliminate the majority of possible input/output errors. Fortunately for you, they are included in the Java 7.0 API and the files we provide do a lot of the work for you.

At this point I am going to assume you are familiar with XML. In order to test your project, you are going to provide a Java program (i.e., a class with a `main()` method) that will read a series of XML commands from an input stream, process those commands, and generate an XML document containing the results. For simplicity's sake, the input XML will be simple enough to process without using any XML tools and will in fact differ little from the command structure used in semesters past. The XML will simply be a sequence of empty elements whose names indicate the command to issue and whose attributes will provide the extra information, for example:

```
<createCity name="A" x="5" y="25" color="red" radius="10"/>
<createCity name="B" x="10" y="10" color="blue" radius="10" /> ...
<deleteCity name="A" />
```

and so forth. If you wanted, you could write your own command parser that treats these tags as merely strings with which you can do as you please, as students were required to do in previous semesters. However, because this is XML, it follows all of XML's rules and is free to do whatever it wants when there isn't an XML rule about it. For example, in XML attributes are not required to appear in any order. So for us, the following two commands are identical:

```
<createCity name="X" x="0" y="0" color="yellow" />
<createCity x="0" name="X" color="yellow" y="0" />
```

This will complicate your parsing slightly.

A few things to keep in mind—first, XML is case-sensitive when it comes to element and attribute names, so "createCity" is *not* the same as "cReAtEcity". Another thing to observe is that XML schemas can do type-checking for attributes and elements. So we can, for example, automatically enforce the restriction that the `x` and `y` attributes for the `createCity` command must adhere to the regular expression '`(0|(-?`[-9][0-9]*))' (more specifically, we can simply define them to be of `integer` type). So you do not have to refer to the spec for the regular expressions associated with certain attributes. Note–this makes your project code simpler, because much validation of data occurs at the world/program interface level. Some attributes, like `color`, have only a few legal values. Attributes whose values can be enumerated in a list of legal values can also be enforced by a schema, so you won't have to worry about those. In general:

- Any attribute whose value is obviously a number (such as `x` and `y` in `createCity`) will typically follow the aforementioned regular expression `(0|(-?[1-9][0-9]*))`. In other words, integers. A programmatic description is that you should be able to obtain the integer value passing them through `Integer.parseInt()`. of numerical attributes by obtaining their values as strings and

- Any attribute whose value is defined to be a string (such as `name` in `createCity`) must have a non-numeric first character; however, the remaining part can consist of any combination of numbers, upper case and lower case letters, and the underscore character. Thus, the regular expression for string attributes will be `([_a-zA-Z][_a-zA-Z0-9]*)`.

Again, these type issues are all taken care of by the XML schema; so, as long as you have your validator configured properly you should be able to catch the exceptions thrown by your validator and do what you need to from there.

6.3.1 Using the provided XML processing code to get a working parser (`updated`)

The DOM (Document Object Model) is most relevant to data structures since it organizes XML objects into a general tree. XML objects are lots of things, but the three things we care most about are documents, elements, and attributes. In the DOM, every XML object implements the `org.w3c.dom.Node` interface. `Node`s have a lot of useful methods, but the ones you care most about are `getNodeName()`, `getNodeValue()`, and `getAttribute()`. Suppose you had a `Node` object representing one of our command elements. You could determine which command it was with the following code:

```
Element command = ...;
if (command.getNodeName().equals("createCity"))
// createCity command
```

To get the value of an attribute, you could use this code:

```
String name = command.getAttribute("name");
```

The next logical question is how do we get an input XML file into a collection of `Node` objects? By using the `XmlUtility` file provided there is a static method called `validateNoNamespace` that takes an `InputStream` or a `File` as a parameter and either throws an `SAXException` for an invalid XML document or returns an `org.w3c.dom.Document` object that models the XML document.

For our collection of commands, the XML file will be similar to this:

```
<?xml version="1.0" ?>
<commands>
    <createCity ... />
    ...
</commands>
```

The first line is a processing instruction which you don't have to worry about (if you are writing a parser by hand, you can ignore any tag beginning with "`<?`"). Because XML is a tree, it must have exactly one root, so we have to nest the list of commands as child elements of the single root element. Given an `org.w3c.dom.Document` object, you can acquire the XML's root element by using the `getDocumentElement()` method. To get a list containing all of the child nodes of this element, use the method `getChildNodes()` defined in `Node`. The return type of this method is `NodeList`, which is basically a type-safe subset of the `java.util.List` interface. To iterate through the items in this list, the code would look like this:

```
Document d = XmlUtility.parse(new File("in.xml"));
Element docElement = d.getDocumentElement();
NodeList nl = docElement.getChildNodes();
for(int i = 0; i < nl.getLength(); ++i) {
    Node command = nl.item(i); // process the command here
}
```

The last piece of usefulness is the ability to validate the element against the document's schema to make sure, for example, that the command is one of the valid commands for this part of the project, that all of the required attributes are present in this element, that all values of are correct type, and so forth. I've written a method called `validateNoNamespace` which validates an entire XML Document against an internally referenced schema (you don't need to know about XML namespaces for this project so don't worry about that).

The syntax for binding an XML document to a schema looks looks like this:

```
<?xml version="1.0" ?>
<commands
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="part1in.xsd"
    <createCity ... />
    ...
</commands>
```

Note a few things. The first attribute defines a new namespace based on the W3C's spec for XML Schema. The second attributes specifies the location of the schema. So to make sure your XML validates against the schema, you want something that looks more like this:

```
Document doc = XmlUtility.validateNoNamespace(new File("in.xml"));
Element docElement = d.getDocumentElement();
NodeList nl = docElement.getChildNodes();
for(int i = 0; i < nl.getLength(); ++i) {
    Node command = nl.item(i); // process the command here
}
```

To ignore comment nodes, you can even do an `instanceof` check of the command node to make sure it is of type `Element`.

The last bit of testing you'll need to do is contextual or semantic checking—for instance, attempting to create two cities with the same name should result in the second command issuing an error. The full list of error conditions will be listed separately since as the semester progresses new error conditions will be introduced as new commands are introduced as well. This last type of checking will involve interfacing with your dictionaries.

6.3.2 Outputting XML using DOM

While you certainly could print out all the XML manually, it is much better in the long run to learn to use DOM to print out your XML. Again, the `XmlUtility` file we provide simplifies some of this process. You only need to know a few methods to easily do this project.

To create a Document object:

```
Document results = XmlUtility.getDocumentBuilder().newDocument();
```

To create an Element object:

```
Element elt = results.createElement("elementName");
```

To set an attribute for an Element:

```
elt.setAttribute("attributeName", "attributeValue");
```

To append an Element to another Node:

```
results.appendChild(elt);
```

To print out a `Document` to `System.out`:

```
XmlUtility.print(results);
```

6.3.3 Outputting XML Conventions

For the first project, again for the sake of brevity and ease of grading, your output will be a series of elements in response to commands, each of which is a reaction to an issued command.

General `<success>` Output

This is an example of the form (in the exact output order) of a general `<success>` tag. All tags will be present even if there are no parameters or outputs.

```
<success>
    <command name="name1"/>
    <parameters>
        <param1 value="value1"/>
        <param2 value="value2"/>
    </parameters>
    <output/>
</success>
```

General `<error>` Output

This is the form (in the exact output order) of a general `<error>` tag. The `<parameters>` tag will always be present even if there are no parameters to the command. In each command there may be several errors listed if several errors occur in a command you will only output the error of *highest priority*. For more information see XML Input specification for each command.

```
<error type="error1">
    <command name="name1"/>
    <parameters>
        <param1 value="value1"/>
        <param2 value="value2"/>
    </parameters>
</error>
```

General `<fatalError>` Output

This is the form of a General `<fatalError>` tag. This is used when there is a problem with the entire document (i.e. the input file is invalid XML or does not conform to the schema)

```
<fatalError />
```

General `<undefinedError>` Output

This is the form of a general `<undefinedError>` tag. This is a default error that will be used if there is an error that is not specified in the spec and is discovered post freezing.

```
        <undefinedError />
```

General sorting of `<city>`

Ordering of `<city>` tags that are contained within the `<output>` tag is in increasing asciibetical order of the city's *name* according to the `java.lang.String.compareTo()` method *unless* the method for sorting is specified within the command's specification.

The testing of your messages won't be as stringent as the fact that you reported a success in the first place. More precise information about what the messages are for each command is contained in the **Input** section of this document.

The root element for output will be `<results>`. The final XML, at a minimum, should look like this:

```
<results>
    <success ... />
    <success ... />
    <success ... />
    <error ... />
    <success ... />
    ...
    <success ... />
</results>
```

6.4 XML command specifications–Current for Spring 2017

Herein lies the XML specifications for the input files you will be provided and for the output files you will be expected to generate. Check these pages regularly as they contain the information both most relevant and most likely to change. Each phase of the project will introduce a new set of commands your parsers will be expected to handle.

6.4.1 Commands

Input will be provided via input redirection. In other words, input files will be passed to your program via the standard input stream (`System.in`).

The basic structure of the input XML will be a document whose root tag is `<commands>`. `<commands>` will contain, as its children, any series of the other commands. All commands (for Part 1 at least) will be single empty elements (recall that an empty element is one that has no children) which contain zero or more attributes. Input values that are numeric will always be integers. The schema and sample inputs will probably be sufficient in describing the input format. *If the schema does not match the input a `<fatalError>` tag is printed without a `<results>` tag.* Of most interest is the success and error conditions for each command and the corresponding messages each should provide. Here's the list:

`commands`

This is the root element of the XML tree for the input files. It contains attributes that will affect the behavior of your command interpreter.

The attributes are `spatialWidth` and `spatialHeight`, both **powers of 2**, but in the range of 32 bit 2's complement integers. These two attributes define the rectangular region the spatial data structure can store. Note that the lower left corner of this region is always $(0,0)$, the origin, as we will not be dealing with negative **city coordinates** in this project. Thus, the spatial structure's bounding volume is the rectangle whose **lower left corner** is $(0,0)$ and whose width and height are given by these two parameters. You must setup your spatial structure to be centered accordingly. Note that although all of our coordinates will be given as integers, you may need to center your spatial structure around a coordinate that is not an integer; so, make sure you plan accordingly.

Observe that these two values have an impact on the success of the `<mapCity>` command, butDo Not affect the success of the `<createCity>` command. Cities can be created in the data dictionary with coordinates outside the boundaries of this range, on the boundary as well as inside this range. However, only cities on the left and bottom boundaries of this range can be "mapped" or inserted into the PR quadtree.

`Possible errors:`

If there is any problem with the attributes in the `<command>` tag a `<fatalError>` tag is outputted without a `<results>` tag, and the program exits.

createCity

Creates a city (a vertex) with the specified name, coordinates, radius, and color. A city can be successfully created if its name is unique (i.e., there isn't already another city with the same name) and its coordinates are also unique (i.e., there isn't already another city with the same (x, y) coordinate).

All names are *case-sensitive*.

`Parameters: (In output order)`

name
x
y
radius
color

`Possible` *`<output>`*`:`

(none)

`Possible` *`<error>`* `types (In priority order):`

duplicateCityCoordinates
duplicateCityName

<success> Example:

```
<success>
    <command name="createCity"/>
    <parameters>
        <name value="Annapolis"/>
        <x value="12"/>
        <y value="14"/>
        <radius value="15"/>
        <color value="red"/>
    </parameters>
    <output/>
</success>
```

deleteCity

Removes a city with the specified name from data dictionary. The criteria for success here is simply that the city exists. Note that if the city has been mapped, then it must be removed from the PR quadtree first, and then deleted.

`Parameter:`

name

`Possible` *<output>*:

If the city is in the spatial data structure, a cityUnmapped tag will appear as:

```
<cityUnmapped name="city1" x="coordx" y="coordy" color="color1" radius="
```

`Possible` *<error>* `types (In priority order):`

cityDoesNotExist

<success> Example:

```
<success>
    <command name="deleteCity"/>
    <parameters>
        <name value="Annapolis"/>
    </parameters>
    <output/>
</success>
```

clearAll

Resets all of the structures including the PR Quadtree, clearing them. This has the effect of removing every city. This command cannot fail, so it should unilaterally produce a `<success>` element in the output XML.

`Parameter:`

(none)

Possible *<output>*:

(none)

Possible *<error>* types:

(none)

<success> Example:

```
<success>
    <command name="clearAll"/>
    <parameters/>
    <output/>
</success>
```

listCities

Prints all cities currently present in the dictionary. The order in which the attributes for the `<city>` tags are listed is unimportant. However, the city tags themselves must be listed either by name in descending order or by coordinate in ascending order, as per the `sortBy` attribute in the `listCities` command, whose two legal values are `name` and `coordinate`. The ordering by name is asciibetical according to the `java.lang.String.compareTo()` method, and the ordering by coordinate uses the comparator discussed in Section 6.1. To reiterate, coordinate ordering is done by comparing y values first; for cities with the same y value, one city is less than another city if its x value is less than the other. This command is only successful if there is at least 1 (1 or more) cities in the dictionary.

Parameter:

sortBy

Possible *<output>*:

A `<cityList>` tag will be contained in *output* and will contain 1 or more city tags of the form:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`

Possible *<error>* types:

noCitiesToList

<success> Example:

```
<success>
    <command name="listCities"/>
    <parameters>
        <sortBy value="name"/>
    </parameters>
    <output>
        <cityList>
            <city name="Derwood" x="19" y="20" color="red" radius="40"/>
```

```
                <city name="Annapolis" x="5" y="5" color="blue" radius="90"/>
            </cityList>
        </output>
    </success>
```

mapCity

Inserts the named city into the spatial map.

`Parameter:`

name

`Possible` *`<output>`*`:`

(none)

`Possible` *`<error>`* `types:`

nameNotInDictionary
cityAlreadyMapped
cityOutOfBounds

`<success>` `Example:`

```
<success>
    <command name="mapCity"/>
    <parameters>
        <name value="Annapolis"/>
    </parameters>
    <output/>
</success>
```

unmapCity

Removes the named city from the spatial map.

`Parameter:`

name

`Possible` *`<output>`*`:`

(none)

`Possible` *`<error>`* `types:`

nameNotInDictionary
cityNotMapped

`<success>` `Example:`

```
<success>
    <command name="unmapCity"/>
    <parameters>
        <name value="Annapolis"/>
    </parameters>
    <output/>
</success>
```

printPRQuadtree

Prints the PR quadtree. Since PR quadtrees are deterministic, your XML should match exactly the primary input/output.

`Parameter:`

(none)

`Possible <output>:`

A `<quadtree>` tag will be contained within *output* and will contain several `<gray>` `<white>` and `<black>` nodes.
The *first* node in the *quadtree* will be the root node, this node can be *gray* or *black*, then the rest of the PR Quadtree follows. Remember, the exact structure of the *quadtree* will be represented by the XML output.

`<gray>`

Nodes will contain *4* children nodes with ordering decided by the order of the nodes within the actual *gray* node in your PR Quad Tree.`<gray>` nodes will have the attributes **x** and **y**, these are integers that identify the location of the *grey* node. They will appear as such

```
<gray x="72" y="40">
    ...
</gray>
```

`<black>`

Nodes represent a mapped city in your PR Quadtree, they have the attributes **name**, **x** and **y**. These will identify the city name and location of the *black* node, and will appear as such;
`<black name="Chicago" x="81" y="47"/>`

`<white>`

Nodes represent an empty node in your PR Quadtree and will appear as such;
`<white/>`

`Possible <error> types:`

mapIsEmpty

`<success> Example:`

```
<success>
    <command name="printPRQuadtree"/>
    <parameters/>
    <output>
        <quadtree>
```

```
                    <gray x="64" y="64">
                        <white/>
                        <white/>
                        <white/>
                        <gray x="96" y="32">
                            <gray x="80" y="48">
                                <white/>
                                <white/>
                                <gray x="72" y="40">
                                    <black name="Boston" x="71" y="42"/>
                                    <white/>
                                    <white/>
                                    <black name="Baltimore" x="76" y="39"/>
                                </gray>
                                <gray x="88" y="40">
                                    <black name="Chicago" x="81" y="47"/>
                                    <white/>
                                    <black name="Atlanta" x="84" y="33"/>
                                    <white/>
                                </gray>
                            </gray>
                            <black name="Los_Angeles" x="118" y="33"/>
                            <black name="Miami" x="80" y="25"/>
                            <white/>
                        </gray>
                    </gray>
                </quadtree>
            </output>
```

saveMap

Saves the current map to a file. The image file should be saved with the correct name. It should match our image file: same dimensions, same cities, same colors, same partitions, etc. How to keep track of your graphic map is discussed in the previous section. Printing it out is discussed there too.

`Parameters (In output order):`

name - filename to save the image to

`Possible` *`<output>`*`:`

(none)

`Possible` *`<error>`* `types:`

(none)

`<success>` `Example:`

```
    <success>
        <command name="saveMap"/>
        <parameters>
            <name value="map_1"/>
        </parameters>
        <output/>
    </success>
```

rangeCities

Lists all the cities present *in the spatial map* within a `radius` of a point `x,y` *in the spatial map*. Cities on the boundary of the circle are included, and `x,y` are integer coordinates. That is, only cities that are in the spatial structure, in this case, the PR quadtree, are relevant to this commmand.

`<success>` will result from the existence of at least one `<city>` that satisfies the range check condition. If none does, then an `<error>` tag will be the result. If the radius is 0 and no city occupies the point `x,y`, then `<error>` tag is the result.

It should be noted that the radius attribute for a city does not factor into this calculation; all cities are considered points.

If the `saveMap` attribute is present, the current map will be saved to an image file (see saveMap). The image file should be saved with the correct name. It should match our image file: same dimensions, same cities, etc. How to keep track of your graphic map is discussed in saveMap. Printing it out is discussed there too. The main difference with saveMap is that the image file should have a blue unfilled circle centered at the (x,y) values passed in with the radius passed in. Because CanvasPlus does not behave well when shapes exceed the bounds of the spatial map, the saveMap attribute will only be present when an entire range circle lies inclusively within the bounds of the spatial map.

`Parameters (Listed in output order):`

x
y
radius
saveMap (optional) - image filename

`Possible` *`<output>`*`:`

The *output* will contain one `<cityList>` which will contain the list of cities. This is an example of a city tag:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`
The cities should be printed in descending asciibetical order of the names according to java.lang.String.compareTo().

`Possible` *`<error>`* `types:`

noCitiesExistInRange

`<success>` `Example:`

```
<success>
    <command name="rangeCities"/>
    <parameters>
        <x value="1"/>
        <y value="1"/>
        <radius value="100"/>
    </parameters>
    <output>
        <cityList>
            <city name="Derwood" x="20" y="30" color="red" radius="12"/>
            <city name="Annapolis" x="20" y="40" color="blue" radius="23"/>
        </cityList>
    </output>
</success>
```

nearestCity

Will return the name and location of the closest city *in the spatial map* to the specified point in space. To do this correctly, you may want to use an algorithm using a *PriorityQueue*, such as www.cs.umd.edu/users/meesh/cmsc420/Notes/neighbornotes/incnear.p or

www.cs.umd.edu/users/meesh/cmsc420/Notes/neighbornotes/incnear2.pdf–otherwise, you might not be fast enough. If two or more cities are equidistant from the specified point in space, then output the city with the larger name in asciibetical order. That is, if city `dark1` and city `duck1` are the same distance from the specified point, then the `duck1` would be given as the nearest city.

`Parameters (In output order):`

x
y

`Possible` *`<output>`*`:`

The *output* will contain one *city* tag which is the nearest city. This is an example of a city tag:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`

`Possible` *`<error>`* `types:`

mapIsEmpty

`<success>` `Example:`

```
<success>
    <command name="nearestCity"/>
    <parameters>
        <x value="1"/>
        <y value="2"/>
    </parameters>
    <output>
        <city name="Annapolis" x="20" y="30" color="red" radius="12"/>
    </output>
</success>
```

The the XML specification appears in www.cs.umd.edu/users/meesh/cmsc420/ProjectBook/part1/part
Do note: except for one set of tests per project, we will provide syntactically error-free XML, which means that we will only test your error checking on one set of test inputs. However, we will check the syntactic validity of *every* output file produced from your code.

7 General Policies

We will be using the latest version of the submit server to test your project submit.cs.umd.edu. We provide no further information regarding submission here, unless a paragraph or two is written by someone who has a clue what should go here.

There may be a few points for secret tests or extra credit tests; or, there may not. We have given you basic I/O files and any further testing is your problem. However, students are encouraged to produce test files and help each other (and us) validate tests prior to final submission. That is, should you produce a test input file, we will be glad to generate a correct output file from our canonical solutions. So, please feel free to generate all the test files you'd like, because you will get feedback.

7.1 Grading Your Project

Projects will be graded on a point system, whereby points will be awarded for each test successfully passed, and the test results for a given structure will be scaled according to the point breakdown given in the spec and the relative part weights given in the syllabus. As you will soon see when you begin implementing your first data structure, we will only ask that you implement a fraction of the functionality described by the Java API for the interfaces we are going to ask you to implement.

However, we strongly believe in rewarding students who go above and beyond the requirements of a project, and occasionally a few points may be awarded for exceeding the project specifications by running extra credit tests on your project. Don't get too excited—extra credit will not be massive, and will boost your project grade just a bit. It's meant more as a means to identify students who are making remarkable progress on the project or to provide those who lost points on an earlier project part with an opportunity for redemption. However, in general, not all students will be able to complete all parts of the project on time. So, please, don't be too ashamed to submit code that only provides partial functionality. We recognize that different students have have varying levels of programming and educational experience; so it is not necessary to get 100% of all available project points to do well in the course. But, if you find yourself never passing more than half of the tests, you should visit Dr. Hugue (even by email) or the teaching assistant of your choice and ask for help.

7.1.1 Criteria for Submission

Your file must satisfy the following criteria in order to be accepted:

- It must be a tar or jar file.
- It must have the extension ".tar" or ".jar"
- It must have the main function in a class called "MeeshQuest", which be placed in the package "cmsc420.meeshquest.part1".
- It must contain a README file to document any references, or implementation notes of interest.
- It must NOT contain any files with extension `class` or `jar`.
- It must compile successfully with the command:

  ```
  javac -classpath xml.jar MeeshQuest.java
  ```

  Although you are welcomed to use your own `xml.jar` file when developing your code, your code must compile successfully with `cmsc420util.jar` to be graded. That is, no custom jar files are acceptable, or needed.
- It must pass the primary input (or some subset thereof). Currently, the primary tests correspond to the public tests on the submit server.
- There are no private tests per se–merely public and release tests for part1. This may change without notice. However, for part1, the private tests must be extra credit once the specification is frozen.

7.1.2 README file Contents

Your README file must contain the following information:

- your name
- your login id
- citations of sources you used, including documentation of all portions of the code that were borrowed or adapted or otherwise not written from scratch by you.
- any additional information you think might be relevant–including documentation of non-working parts of the project–this can be useful later when you've forgotten what works or doesn't.

If you leave out the `README` your project will fail to submit!

7.1.3 Project Testing and Analysis

Your program will be tested using the command:

```
java -classpath xml.jar :. MeeshQuest< primary.input > primary.output
```

We will then check that your output is correct. You are responsible for matching the *letter of the specification*, which is why we freeze a week before the project is due. If you find a discrepancy between the supplied Primary I/O and the specification, please report it immediately so that the Primary can be corrected, and any remaining ambiguities can be clarified. Remember, we have to return points if your output matches the specification and we don't. And, helping us keep the primary correct will benefit us all in the long run.

7.2 Grading

There will be several methods used to grade your projects. Your projects will be graded by running them on a number of test files with pre-generated correct (we hope) output. Your output will have all punctuation, blank lines, and non-newline whitespace stripped before diffing similarly cleaned files.

Some of your data structures may be included with the TAs own testing code to test their efficiency and correctness.

Some text output cannot always be graded by simply diffing because there is no guarantee that we will have the same output. In these cases your project's output will be pre-processed. In the case of the B+ tree, for instance, this program will verify that each node has the correct number of keys, that they are correctly ordered, and that all the correct data is at the leaves (and any other rules I may have left out).

Thanks to the miracle of automation you should expect your projects to be run on very very large inputs.

7.2.1 Testing Process Details

Each part of your project will be subjected to a variety of tests with points assigned based on the importance of the structure to the project, and the relative difficulty of producing working code. Note that we will do our best to assure partial credit where possible by decoupling independent tests.

Evaluation of your project may include, but is not limited to, the following testing procedures:

- Basic I/O: when the structure is deterministic, the test outputs are processed using a a special "XML diff" that parses your output into a DOC (just like your program does) and then compares it with a valid output. Thus, you can put whitespace wherever you want it, and order attributes within tags however you like. However, your output MUST be syntactically valid XML. IF WE CANNOT PARSE YOUR OUTPUT, THEN IT WILL FAIL THE TEST IMMEDIATELY.

- Rule-Based: when multiple correct answers, we will use code-based verification that your results satisfy the required properties.
- Interface: when you are required to implement an interface, we will use our test drivers to execute your code. (Not for Part 1)
- Stress and Timing: used to verify that your algorithms meet the required asymptotic complexity bounds.

7.2.2 Small, Yet Costly, Errors

In general, your project will either completely pass each test, or completely fail it. In particular, the following "small" mistakes have been known to fail projects completely in many tests:

- Forgetting to match each "open" XML tag (e.g. ¡`<tree>`¿) with a "close" XML tag (eg. `</tree>`)
- Forgetting to include the `/` at the end of single tags (eg. `<success />`)
- Sloppy capitalization (eg. `<Success/>`)
- Misspellings (eg. `<success/>` or `<success command = ‘‘creatCity’’>` ....

On rare occasions, the tests may be extended to allow minor errors to pass, but only if we find that the cost of that error is significant and unfair. However, in the large, you'd do better to get this stuff right the first time, than expect partial credit.

The tests will try to test mutually exclusive components of your projects independently. However, if you don't have a dictionary which at least correctly stores all points, and some 'get lost', you may still end up failing other tests since they all require a working dictionary. This does not reflect double jeopardy on the part of the test data, since it is always possible to use some other structure (such as the Java `TreeMap`) for the data dictionary, and receive points for portions of the tests that merely reference the data in the dictionary. Do not hesitate to ask for suggestions as to how to work around missing project functionality.

7.3 Standard Disclaimer: Right to Fail (twice for emphasis!)

As with most programming courses, the instructor reserves the right to fail any student who does not make a good faith effort to complete the project.

If you have problems with completing any given part of the project please talk to Dr. Hugue immediately—do not put it off! While some may enjoy failing students, Dr. Hugue does not; so please be kind and do the project, or ask for advice immediately if you find yourself unable to submit the first two parts of the project in a timely manner. A submission that gets only 20 or 30 points is considerably better for you than no submission at all. And, we employ the buddy system. If you are still lost after adopting the Pqrt 1 canonical to start Part 2, please ask for help. Recognizing when you need help and being willing to as for it is often a sign of maturity.

7.4 Integrity Policy

Your work is expected to be your own or to be labeled with its source, whether book or human or web page. Discussion of all parts of the project is permitted and encouraged, including diagrams and flow charts. However, pseudocode writing together is discouraged because it's too close to writing the code together for anyone to be able to tell the difference.

Since the projects are interrelated, and double jeopardy is not our goal, we have a very liberal code use and reuse policy.

- In general, any resources that are accessed in producing your code should be documented within the code and in a `README` file that should be included in each submission of your project.
- First and foremost, use of code produced by anyone who is taking or has ever taken CMSC 420 from Dr. Hugue requires email from provider and user to be sent to Dr. Hugue. That means that any student who wants to share portions of an earlier part of the project with anyone must inform Dr. Hugue and receive approval for code sharing prior to releasing or receiving said code. This also applies to sharing code from another course with a friend. Please, please, ask.
- Second, since we recognize that the ability to modify code written by others is an essential skill for a computer scientist, and that no student should be forced to share code, we will make working versions of critical portions of the project available to all students once grading of each part is completed, or even before, when possible.
- Dr. Hugue is the sole arbiter of code use and reuse and reserves the right to fail any student who does not make a good faith effort on the project. Violators of the policies stated herein will be referred to the Honor Council.

Remember, it is better to ask and feel silly than not to ask and receive a complimentary F or XF.

7.4.1 Code Sharing Policy

During the semester we may provide you with working solutions to complete portions of the project. It is legal to look at these solutions, adopt pieces of them, and replace any part of your project with anything from them so long as you indicate that you *accessed* this code in your `README`.

Furthermore, any portion of your code that contains any portion of the distributed work should contain identifying information in the comments. That is, indicate the source or inspiration of your code in the file where it was used, as well as in your README. It is a good idea to wrap shared code with comments such as "`Start shared code from source XYZ`" and "`End shared code from source XYZ.`" You may also use comments such as "`Parts` `of this function/file were based on code from source XYZ.`" You cannot err by including this information too often. Making it easy for us to find, makes it easy for us to recognize your compliance with the rules.

Failure to properly document use of distributed code in your project could result in a violation of the honor code. Indicate the distribution solution(s) on which your code is based.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, One Jacob Way, Reading, MA 01867, USA, 1995.

A General notes on Java

This is the last semester that we will assume merely a minimal understanding of the Java language, so don't fret that your less-than-expert knowledge of Java will preclude your passing the course, much less excelling in it. We will provide detailed explanations of the Java concepts and constructs to be used throughout the semester, and you should be extremely comfortable in Java by semester's end. The syntax of Java is very similar to C++, but there are a few important distinctions to be aware of before jumping into data structure development with both feet.

A.1 IDEs

We **strongly** encourage you to use an IDE to develop your project code. Although you could develop this project using only Emacs and a Java compiler and virtual machine, it is in your best interest to use an integrated development environment (IDE). An IDE allows you to write, compile, test, debug, and run your program without having to go to the command line (or a shell in Emacs). A good IDE is one that helps you find compilation errors and allows you to debug your program by stepping through it line-by-line while displaying a print out of all local variables.

Many Java IDEs are available. Try out a few, and find out that works for you. Some potential IDEs include but are not limited to:

- Eclipse [www.eclipse.org]
- JCreator [www.jcreator.com]
- Dr. Java [drjava.sourceforge.net]
- jbuilder [www.borland.com/jbuilder] (free but registration required)
- NetBeans [wwws.sun.com/software/sundev/jde]

- SunOne [www.sun.com/sunone] (Community Edition is a free download from Sun)

Do a Google search to find the URLs to download these IDEs or look for them on the WAM machines (I have no idea which of these are installed on the UMD networks, other than Eclipse and Dr. Java). If you find another IDE which you like, post it to the Discussion Board to earn class participation points and allow others to share in your wisdom at the same time. Note that Eclipse is installed on the grace cluster, which is where your code will be submitted.

A.2 Pass by reference, but not really

Every semester a new group of students gets caught up by the same thing in Java. They start out hearing "Java is always pass by reference," and they do silly looking things like the following:

```
void foo(String t) { t = new String("World"); }

String s = new String("Hello"); foo(s);

System.out.print(s); //prints "Hello".  Why didn't it change?
```

In true pass by reference C++ this would have worked. But what is happening is not really pass by reference; it is pass by value, except what is being passed is a pointer. If you were to transfer the above to C++ it would look like

```
void foo(String *t) { t = new String("World"); }

String *s = new String("Hello"); foo(s);

cout<<*s<<endl; //prints "Hello".  Hopefully obvious why
```

You can see in the second example that `t` is only a local copy of `s`. If you alter the value `t` is pointing at then `s` will see the change. However, if you point `t` at something else, `s` will never know. In this example there is actually no way for `foo` to change `s`, since Java `String`s are immutable after creation. An error less obvious than the above is

```
void foo(String t) { t = t+"World"; }
```

This looks like concatenation, not reallocation, but that '+' operator actually allocates a new `String`. The above is actually just a shortcut in Java for

```
void foo(String t) {
    StringBuilder temp = new StringBuilder();
    temp.append(t);
    temp.append("World");
    t = temp.toString();
}
```

It's important to realize what's going on in the background! Of course in the above example, `foo` still doesn't change `t`, but what you could do instead is

```
void foo(StringBuilder t) { t.append("World"); }
```

This time, since `t` always points to the same location, the original value really is modified. In Java, "pass by reference" as C++ programmers tend to think of it always requires some kind of wrapper. In the last example, `StringBuilder` is a wrapper for a dynamically sized character tabular. There is a quick and dirty hack to get a similar effect without building an entire class wrapper; pass a 1 element tabular instead:

```
void foo(String[] t) { t[0] = new String("World"); }

String s[] = new String[1];
s[0] = new String("Hello");
foo(s); //s[0]= "World"
```

This works for a similar reason. `t` points to the same tabular in memory that `s` does. When an element of the tabular is updated by `t`, `s` will see the change as well. This ends my FYI on pass by reference; try not to get caught up by this common error :)

A.3 A few notes about Java's object oriented design

In C++, inheritance and polymorphism could be considered practically an afterthought by comparison to languages like Java and C# which were designed as object oriented from the ground up. In Java, you never write programs; you write classes (objects). It's not possible to write a function that is not encapsulated as part of a class—in other words, all "functions" in Java are more appropriately methods. Writing Java code that isn't written with careful attention to its design as an object is a crime against the language; no more should you write C style code in Java than you should write Lisp style code in C. It's sufficient to remember that when you're writing Java code, you're writing OO code. Don't forget that. Following is a brief outline of a few of Java's OO features that differ from C++.

In C++, to inherit from any other class, the syntax looks like this:

```
class Foo : Parent1, Parent2 { ... };
```

In Java, the colon is subsumed by the keyword **extends**. Also note that every class-level and package-level declaration requires an access modifier, so an entire Java class must be marked as being either `public`, `protected`, or `private`. So in Java, the syntax for inheritance looks like this:

```
public class Foo extends Parent1 { ... }
```

Java does not allow multiple inheritance in this manner. Every class (except `Object`) has exactly one parent. If a class is not defined to explicitly extend some other class, by default it extends `Object`. Every class you write in Java will extend `Object`. Clearly `Object` is the only class in the Java language that does not have a parent class.

However, Java does allow multiple inheritance, but that inheritance is restricted. Java introduces the concept of interfaces. An interface could be defined as being a class which may contain only public abstract methods and static fields of any kind. (In Java, "abstract" means the same thing as "pure virtual" in C++ speak; "static" means the same thing as in C++). A class is free to inherit from as many interfaces as it wants, however a different keyword is used when describing inheritance from an interface:

```
class Foo extends Bar implements Interface1, Interface2 { ... }
class Foo implements Interface1 { ... }
```

Note that in Java all methods (except static methods) are virtual. This has two consequences: the first is that you are not required to qualify your methods as "virtual" and the second is that you cannot deliberately make a method non-virtual. Static methods are not virtual but they also behave differently and have added restrictions (for instance, you cannot refer to the class's fields inside a static method since the magic "this" pointer does not exist in static methods), so don't use static methods expressly to prevent a method from being treated as virtual. (In case you are rusty on your OO slang, a virtual method in C++ is one whereby if you assign a child class to a parent class pointer and invoke an inherited method on that pointer, the child's method will be executed. A more eloquent description, which uses even more OO jargon, would be to say that a virtual method is invoked based on a variable's actual type irrespective of its declared type).

In Java, methods can be explicitly marked as being "pure virtual" (in C++, a pure virtual method is a virtual method that doesn't have a body). However, recall that in Java the term for "pure virtual" is instead "abstract". If you wish to declare a method "abstract", do so like this:

```
public abstract void foo();
```

If a class contains a single abstract method, the class itself must also be marked abstract. Abstract classes are not allowed to be instantiated (i.e., their constructors are not allowed to be invoked). Abstract classes are still allowed to have constructors since their non-abstract child classes might wish to invoke the parent constructor from within their own constructors. To mark a class abstract:

```
public abstract class Foo { ... }
```

Although the presence of a single abstract method in the body of the class would suggest that the entire class is abstract and therefore marking it explicitly is basically unnecessary, no compliant compiler will allow you to compile a class that contains an abstract method but is not itself declared abstract.

A.4 `Comparable`, Comparators, and how Java gets by without overloaded operators

In C++, you had the opportunity to overload operators for whichever types you wanted. In 214 you may have implemented a templatized binary search tree (BST) that worked with the caveat that the provided type had to have at least its 'less than' operator overloaded. This is fine and it works, but there's no elegant way for you to enforce this invariant in code—the BST would simply break if you passed in a class that didn't have its operator properly defined. Furthermore, suppose your application changes and you want to sort your data elements in a different order. With the stock C++ approach you'd need to modify the data element's less than operator to reflect the change. But what happens if your application changes again and you want to have both of the orderings (forward and reverse) available simultaneously? Have two versions of your data element, with different operators? That probably sounds like a terrible solution because it is. Overloaded operators are one feature that many programmers (even those who would call themselves OO competent) claim to miss. But overloaded operators actually cause more problems than they solve, and Java's solution is actually much more elegant and far more modular.

It is true that not all objects can be compared to one another in any meaningful way; some data is simply not sortable. Some data has many different orderings. Some data has one obvious and universally meaningful ordering. Numbers, for instance, typically have a well understood ordering. Strings, too, have an accepted lexicographical ordering that programmers have come to expect. In Java, objects that have one commonly accepted method for comparing themselves against each other are designated to implement the `Comparable` interface.

The `Comparable` interface contains one method with the following signature:

```
public int compareTo(Object obj);
```

Note the return type. The API specifies that the `int` value returned must obey three rules: if the parameter is less than this object, return a number strictly less than zero ($x < 0$). If the parameter is equal to this object, return exactly 0. If the parameter is greater than this object, return a number strictly greater than zero ($x > 0$). Objects that implement this interface can be compared like this:

```
public class Foo implements Comparable { ... }
...
Foo f1 = new Foo();
Foo f2 = new Foo();
int r = f1.compareTo(f2);
if (r < 0) System.out.println("foo1 < foo2");
else if (r == 0) System.out.prinltn("foo1 == foo2");
else System.out.println("foo1 > foo2");
```

Note that in Java, the `==` operator is unilaterally used for pointer comparison. The `==` operator will return true only if the two variables refer to *exactly* the same object in memory (i.e., they point to the same location). The most commonly used class that implements `Comparable` is `String`. Remember that interfaces are types. The following code segments are legal:

```
Comparable c = new String();
if (c instanceof Comparable) System.out.println("tautology");
int r = ((Comparable)c).compareTo("hello");
```

Be warned, however, that the object versions of the primitives (`Integer`, `Float`, `Double`, `Long`, `Short`, `Byte`, `Boolean`) do in fact implement the `Comparable` interface, but they are not `Comparable` to each other—this has to do with precision issues. You can compare a `Float` against a `Float`, but if you try to compare a `Float` against an `Integer`, a `ClassCastException` will be thrown.

Objects that implement the `Comparable` interface can be inserted into sorted data structures in the API such as `TreeMap`.

This is fine, but suppose you are given some class for which you lack the ability or the privileges to modify its source to make it implement the `Comparable` interface, or you are in the situation whereby you wish to impose an alternate ordering to preexisting objects. The `Comparable` interface does not solve the BST problem suggested previously—you would need to modify the body of `String`'s `compareTo()` method to sort strings in reverse asciibetical order. In this case, you can't modify `String` or `TreeMap`, so how could you put `String`s into a `TreeMap` and have them sorted in reverse order? The answer is by introducing a third party object that will do the comparison for you.

A `Comparator` is an independent class which contains one method which takes two parameters—these two parameters are the two objects being compared. Where the `compareTo()` method in `Comparable` always uses the invocation target as one of the two objects being compared (i.e., the object on the left of the `.`), a comparator takes both sides of the comparison as parameters. In Java, comparators are defined by the `Comparator` interface, which contains a `compare` method with this signature:

```
public int compare(Object o1, Object o2);
```

A typical `Comparator` class will look like this:

```
public class MyComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        // code to compare o1 against o2 goes here
    }
}
```

Observe that to be as generic as possible, the parameters that the `compare` method takes are both `Object`s, but typically a comparator is designated for a specific type. Usually the first thing the `compare()` method will do is cast the parameters into the target types. For instance, let's write a comparator for `String`s that imposes reverse ordering:

```
public class StringReverseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        int r = s1.compareTo(s2);
        if (r < 0) return Math.abs(r);
        else if (r == 0) return 0;
        else return -r;
    }
}
```

It's easy to see that this code can be reduced: all we're doing is swapping the sign of the result of the `compareTo()` method for the two `Strings`. A better version would be:

```
public class StringReverseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return -s1.compareTo(s2);
    }
}
```

However, to be even more generic, we can observe that this code would work for any two objects that implement `Comparable`. In fact, at a minimum, it would work for any two objects as long as the first parameter implements `Comparable`. (By work, I mean execute without throwing an exception). Granted, `o1` and `o2` should probably not just be `Comparable` but also be the same type. We can check that programmatically, but for brevity's sake let's trust the user (cardinal rule in software engineering: *never trust the user*). A still better version:

```
public class ReverseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return -((Comparable)o1).compareTo(o2);
    }
}
```

We can still make another improvement. Here, we are reversing the ordering imposed by the `compareTo()` method only for objects that implement `Comparable`. But remember the entire reason for inventing `Comparators` in the first place—not all objects are comparable! We might want to reverse an ordering imposed by another comparator. Thus, what we really want to do is compose our `ReverseComparator` out of another comparator. Consider this implementation:

```
public class ReverseComparator implements Comparator {
    private Comparator comp;
```

```
        public ReverseComparator() { this(null); }
        public ReverseComparator(Comparator comp) { this.comp = comp; }
        public int compare(Object o1, Object o2) {
            if (comp != null) return -comp.compare(o1,o2);
            else return -((Comparable)o1).compareTo(o2);
        }
    }
```

Now we have a comparator that can handle both situations—if you create a `ReverseComparator` based on some other `Comparator`, the result will be the reversal of that comparator's ordering. Otherwise, if you pass in nothing (or `null`), it will assume that the objects you pass it implement `Comparable`. (If they don't, a `ClassCastException` will be thrown).

Now that you see how a `Comparator` is implemented, consider how a `TreeMap` could use one. Taking a look at the docs for `TreeMap`, you will notice that `TreeMap` has a constructor which takes a `Comparator` as a parameter. By passing in an instance of our `ReverseComparator`, `TreeMap` will use that comparator whenever it has to make a comparison between two objects. It will use this comparator to figure out where in the tree each object belongs. `TreeMap` will sort a given object `x` as first in its ordering if, for all other objects `y` in the map, `comp.compare(x,y) < 0` where `comp` is the comparator passed in to the `TreeMap` at creation time. Likewise, when searching the `TreeMap` for a given object `x`, the `TreeMap` will report a successful search if it finds some other object `y` in the map such that `comp.compare(x,y) == 0`. You can make `TreeMap` (or any other sorted structure available in Java) do amazing and varied things by altering the comparator you pass into it. For instance, you could easily create a "black hole" object by passing a `TreeMap` a comparator that never returns 0. You could add objects all day long and it would sort them properly—iterating over the objects in the `Map` would produce the proper ordering. But any object you tried to search for would be reported as not existing in the tree because the comparator is incapable of returning 0. The bulk of your first project will be tricking `TreeMap` into behaving like plenty of other things by playing tricks with the `Comparator`s you pass it.