**­Documentation**

Note: The JAR files contain the original Java source code.

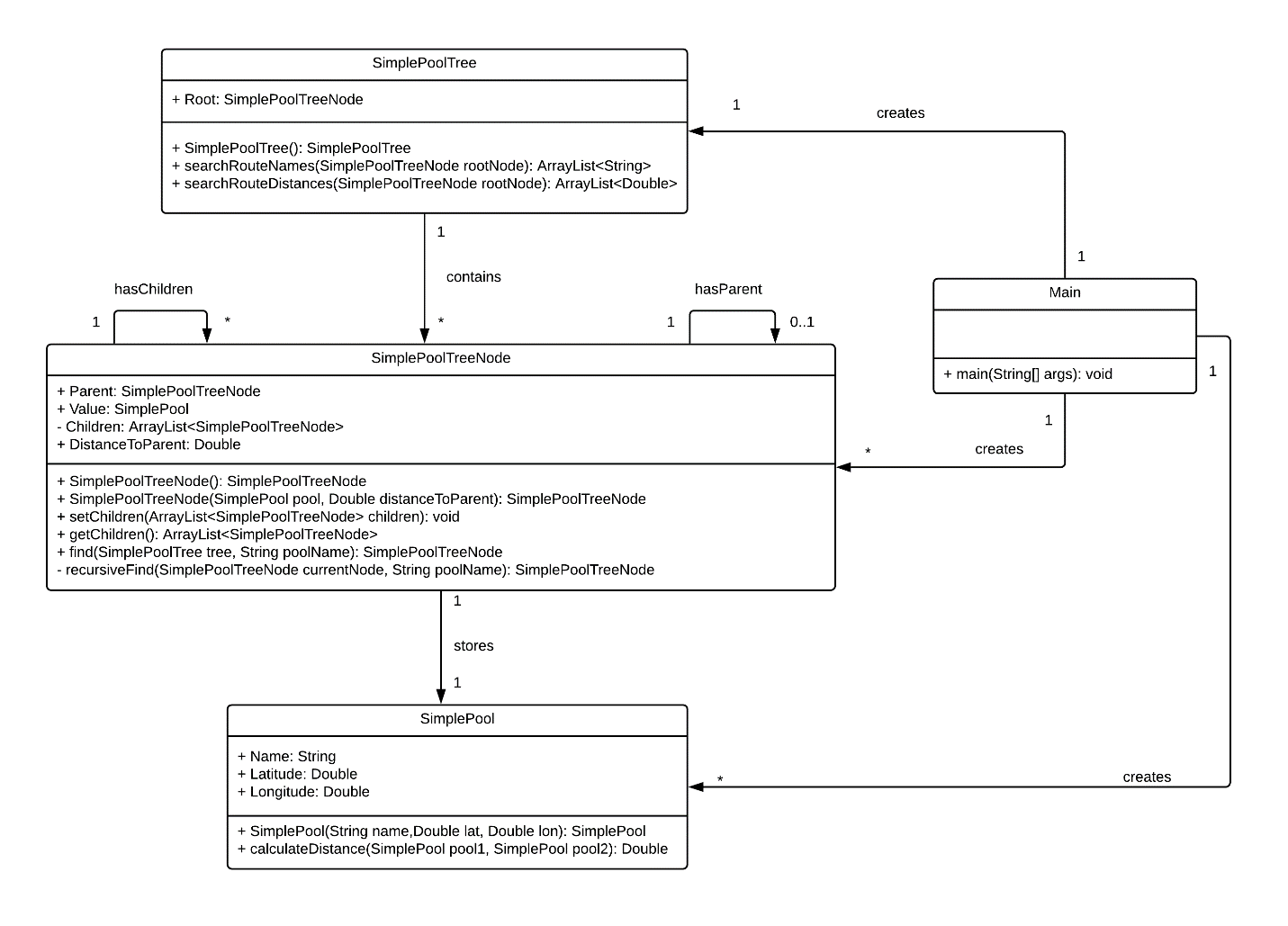
**Question 0 - Preprocessing**

This was the preprocessing phase done in **Java**. The **Jackson** library was used for Json Parsing.

The *wading-pools.json* file was first modified (*wading-pools2.json* was the file used in this phase) manually to eliminate things that I did not understand as I was only familiar with the very basics of Json. Next, I used a Java program that I created as well as the third-party library Jackson to process this modified Json file into both a text file (.txt) and a Prolog database (.pl) file. The professor allowed us to **sort the pools** in this step before creating the files, so that is what I did. To sort the files, I had to override the object’s *compareTo* method and implemented it so that is compares a Pool object’s longitude attribute, so that the pools are sorted from west to east. It is important to note that only 3 attributes were parsed from the Json pools: name, latitude and longitude (printed to files in that order). Double-click the JAR file to create the two files.

**Question 1 - Java**

Due to the nature of the solution, support for nary tree creation, search and traversal was required. The following **nary tree** structure was created in an object-oriented manner:



After running *Preprocess.jar* by double clicking it, both the Prolog database (*FindRoute.pl*) and text file will be exported (*ParsedPools.txt*). Copy *ParsedPools.txt* to the same path as *FindRoute.jar* and open the command prompt. Change to the directory with the previously mentioned JAR file and run the following command:

***java -jar FindRoute.jar ParsedPools.txt***

The solution will be in the newly created *solution.txt* located in the same path.

The steps to implement this object-oriented solution were:

1. Create a list of pools (sorted) from the input text file.
2. Build the root node and then the remaining tree.
3. Pre order traverse the tree to obtain the solution.
4. Print the solution to a new file.

I realized only after submission that I had **used the incorrect formula for angle to rad conversion that the professor posted initially in the assignment description** and I forgot to update it for my Java implementation, therefore the solutions printed by this Java implementation are incorrect because of this section of code from ***SimplePool.java***:

double lat1Rad = 180\*(poolOne.Latitude/Math.PI);

double lon1Rad = 180\*(poolOne.Longitude/Math.PI);

double lat2Rad = 180\*(poolTwo.Latitude/Math.PI);

double lon2Rad = 180\*(poolTwo.Longitude/Math.PI);

**Question 2 - Prolog**

Note: Not enough time was available for me to debug the Prolog code after finishing coding the implementation solution, but most, if not all the logic should be correct.

The first step was to read the Prolog database. This database was a simple rule that defines a variable to the list of sorted pools, where each pool is a list of 3 elements itself. The remaining code was added to the same (.pl) file containing the rule defining the list of sorted pools. The pool structure, node structure and nary tree structure were all based on lists, as there is not really an alternative to this in Prolog. You can view the code and the Scheme description for more information.

The remaining implementation steps (2 to 4) were similar to the Java implementation, however, since this is Prolog, almost everything was done recursively instead of iteratively.

All defined predicates are explained in *Predicate Definitions.txt*. Note that unfortunately, the code does not do anything as there was an error that I did not have time to correct.

**Question 3 - Scheme**

This implementation is similar to the Prolog implementation, in the way that recursion is the main way of processing the lists and also in terms of near identical pool structure, node structure and tree structure.

Here are the following key design decisions (near identical to Prolog implementation):

* Pool Structure: '(PoolName Latitude Longitude)
* Node Structure: '((PoolName DistanceToParent) (ChildrenNodesList))
* Tree Structure: A list of Nodes.
* Solution: List of solution steps.
* Solution Step: ' (PoolName DistanceToParent)

Note the code is well-commented as well as all predicates defined in a separate txt file.

The solution calculation follows the same general steps as all other implementations:

1. Obtain a sorted list of pools from an input file.
2. Build the tree.
3. Traverse the tree.
4. Save the solution to an output file.

**Question 4 – Go**

This implementation was very similar to the Java implementation in the way that the tree structure is based off the use of containers (Go structs / Java Classes). Most of the implementation steps are near identical as well. One major difference is that the node struct has a pointer to a parent Node struct and an array of pointers to Node structs for a node’s children, while in the Java implementation, pointers were not explicitly used and instead of an array (used in Go), a list data structure was used to store the children nodes of a node in Java.

Run the Go file and follow the instructions in the console. Use *ParsedPools.txt* from Question 0.

Another major difference is the use of concurrency to solve the problem which will be described in question 4’s *Strategy Explained.docx.*

**Key observations for all questions**

This problem deals with nary trees and in order the pre-order traversal of an nary tree requires us to visit a node followed by the *k* children from left to right.