# Part 1:

First the user is asked if they would like through the prompt of input being between -1 to 5. 1 being the ability to add to the knowledge base; 2 to clear the current knowledge base; 3 to display the current knowledge base; 4 to enter a query; 5 to run the query that has been entered on the knowledge base and finally, -1 will exit out of part one and return to the main menu.

If 1 is chosen, the user will be prompted to enter the data into the knowledge base. This data is to be structured in [Literal1,Literal2,!Literal3] with the number of literals being able to be any number greater than one. The ‘¬’ symbol was replaced with the ‘!’ due to issues in Java in processing the symbol. The program gets the user input via starting an infinite while loop, that can be exited by inputting -1. If the input is not -1 then the program will create a new instance of the Clause object with the input that the user entered being the only parameter. The Clause object first takes the data and strips the surrounding square brackets from the input. It then splits up the string using the regex of ‘,’ it then loops through these new strings creating new Literal objects. It checks to see if the string contains the ‘!’ symbol if it does then the symbol is removed. When the Literal is made it takes two parameters, the string of the name of the literal and a boolean. The boolean marks if the literal is false. It then adds these new literals to a list called “listOfLiterals” which can be retrieved from the object later. This will continue until the user enters -1 to go back to the menu to choose another action.

If 2 is entered, then the knowledge base stored in the local space is set to null and the clearKnowledgeBase method in the InputManager class is called. This sets the knowledge base that is also stored in this class to null then creates a new empty ArrayList.

If 3 is inputted, then first it checks if the knowledge base is not null. If it is not, then it will start a for loop of the length of the knowledge base and output each element in the knowledge base. If the knowledge base is null, then it will do nothing.

4 prompts the user to input the query in the negative form and stores it in the Input Manager.

5 first makes sure the knowledge base is not null, the calls the method to run the query in the Query class. It then checks if the boolean returned by this function is true or false, if its true it will output ‘SOLVED’ otherwise it will output ‘NOT SOLVED’. The runQuery in the Query class works out if to return to true or false, first by taking 2 parameters the knowledgebase (ArrayList of clauses) and the query(Literal). First a boolean called *change* is defined then a new ArrayList of Literals called currentQuery the query that was passed to this method is then added to this new list. A do while loop is then started with the condition of *change* being true and will loop until change is false after all the subsequent loops. Within the first do while loop change is set to false, the knowledgebase is sorted by the length of the literals inside. Int i,j and k are declared and set to 0. If current query is empty, then the method will return true otherwise 3 nested do while loops are then created. The inner most loop is as follows:

1. Check if currentQuery(K)s data is equal to knowledgebase(i)s sub list literal (j)s data
2. Check to see if the above have opposite polarity if they are equal.
3. Remove k from the current query
4. The remove j from the knowledgebase.
5. Check if i is then empty if it is then remove it from the knowledge base otherwise add everything left in the Clause to the currentQuery
6. Increments k

This first loop will loop while k < the size of the current query and while change is false. The middle do while loop increments j and sets k back to 0. This loop will loop while j < than the size of the list of literals in element I in knowledge and while change is equal to false. The outer most loop increments I and sets j and k to 0 this will loop through until i < the size of the knowledgebase and change is not true. If it ever gets outside the look that checks if a change has been made this means, there is nothing that can cancel out in the knowledge base and query therefore the method returns false.

If anything, else other than the above defined inputs are inputted then the program will inform the user of an invalid inputs.

## Testing:

# Part 2:

Firstly, another loop is started once again using -1 to exit the loop, with two other options 1 and 2.

1 adds the information to the inheritance network first by starting a loop. This will loop until the user enters ‘-1’ otherwise it prompts the user to input the new data in the form “Child IS-A/IS-NOT-A Parent”. If the input does contain the ‘IS-A’ or ‘IS-NOT-A’ then the string is split based on this regex with a space on both sides. In the case that it is a ‘IS-A’ then the polarity is set to true otherwise it is set to false. After this a new Connection object that is made with the input in the form of (‘Parent’, ‘Child’, Polarity). All the nodes are then searched to see if any of them match the name of the ‘child’. If one of them does, then a new connection is added to the list of connections in the node object and a boolean called connectionFound is set to true. After this search is completed. There is a check to see if connectionFound is false then a new node is created with the parameter ‘child’ after this new node is created the new connection that was made previously was added to this node. This new node is added to the list of nodes. Finally, the connectionFound boolean is set back to false.

When the user inputs 2 first it prompts the user to input the start and end points. First it is checked if the string contains IS-A or IS-NOT-A if it is the later then the boolean isNot is set to true. The input is then split with the respective regex. All the nodes are then looped through to see if the start node exists and if it does then the startNodeFound boolean is set to true the startNode is then set to the node that matched the name given in the original input. After this loop the endNodeName is set to the name given in the original input.

If startNodeFound is true the a new ArrayList of type paths, this is set to the returned value of the getPaths method in CreatePaths it has the parameters of: the list of nodes, the start node and the name of the endNode otherwise the program will output that “The start node does not exist in the network”.

Inside getPaths firstly, a new Path is created called path. A path is an object that stores and ArrayList of Strings of all the nodes names aswell as the nextNode, targetNode and the paths length. The startNodes name is added to the first space in this new path. An ArrayList of connections is then made called connections then furthermore an ArrayList of paths called paths is then created within the method. A variable of Node called currentNode is set to the Start Node that was passed to this method. A boolean called currentNodeChanged is then created and set to false. connections are then set to the ArrayList stored in the currentNode that stores all the connections to that Node. If the returned connection list size is 0 then “null” is added to the end of the path and the path is added to paths and the paths list is returned.

If on the other hand it is equal to 1 then, the polarity of the connection is checked if it is equal to true indicating the connection has a positive polarity then, that parent name of that connection is retrieved to see if it is equal to the name of the end node. If it is then the endNodeName is added to the path and then added to the list of paths with the list being returned. If it is not equal to the end nodes name, then there is a search made throughout the list of nodes to see if there is a node that can be found with the name that matches the connections parent name. If one is found, then the name of this node is added to the path the currentNode is set to equal node and currentNodeChanged is set to true the loop is then broken out off. If a node could not be found, then the method will return null. If the polarity is negative though, then the connections parent is checked to see if it is equal to the end nodes name if it is then the end nodes name is added to the path with a prefix of ‘!’ this path is then added to the list of paths that is then returned. If the parent is not equal to the end node name, then “null” is added to the end of the path this is then added and returned as before. The “null” addition is because a negative connection can only be at the end of a path so much connect to the end node.

If the list of connections is any greater than 1 then a loop is started to go through each one of the connections. First the connections polarity is checked if it is positive then a Path called tempPath is created using the parameters of: A new ArrayList, the start node and the endNodeName. All the names of the nodes in path are then added to this tempPath. If the connections parents name is the end node then this temp path is added to the list of paths after having the ends nodes name appended to it. Otherwise, a new ArrayList of Paths called tempList is created. The list of nodes is then looped through to find a node that matches the parent nodes name in the connection. When it is found, the method is recursively called passing: the list of nodes, the node that was found and the end nodes name. The result that is returned from this recursive call is stored in the tempList variable. After this the new temporary list is looped through each time creating a new Path variable adding all the node name from the current path in this temporary list of paths to it. This new path is then added to the list of paths. If the connection is found to be negative, then a tempPath is still created all the names from the current path are then added to the temporary path. Like before when the path was a negative polarity the parent name is checked against the end nodes name and if it matches the end node name prefixed with a ‘!’ is added to the end of the path and this path is added to the list of paths. otherwise null is added to the end of the path and added to the list of paths. After all the connections have been looped through the list of paths is returned.

After the list of paths of the knowledge network has been obtained the isNot boolean is check and if paths is equal to null then the program will inform the users that child is not a child of parent. Otherwise, if the path size is greater than 0 the list of paths is looped through and if the path contains the word null in the list of nodes in that path then the path is removed. After this an Integer called lowestDistance is defined and set to the maxValue and Integer can be set to. An ArrayList of Paths is then once again created called shortestPaths. The remaining valid paths are first printed out and are then looped through again checking to see if the length of the path is less than the lowestDistance if it is then the lowestDistance is set to be equal to the length of the path. The ArrayList of shortestPaths is cleared then this path is added to it. If it is equal in length, then the path is simply added to the list of shortest paths.

After this, the list of paths is checked if any are redundant. This is done by a method in the CreatePaths class called checkIfRedundant it takes one parameter which is the list of paths. This method first starts a for look to look through the list of paths. inside this for loop a new ArrayList of the nodeNames of the current path being checked. A second for loop then is started to loop through all then nodes in this list. Two variables called childName and parentName are made. Then a third for loop is started to loop through all the paths. if this for loops path does not equal the first for loops path them two Integers are created for the childNameIndex and parentNameIndex both are first set to -1. A second list of node names is created to store the nodes from this path from the 3rd loop. This ArrayList is then looped through to see if the index in the array is equal to either the childName or parentName found earlier. If it is then the childNameIndex or parentNameIndex is set appropriately. After this for loop has been completed, there is a check to see if childNameIndex and parentNameIndex are not equal to -1 and they have a difference greater than 1. If they do, then the path is removed for being redundant. Finally, the updated list of paths is returned.

Then there is a check to see if there is any pre-emption in the inheritance network this also only takes one parameter being the list of paths. This method first loops through the list of paths that was passed to it, the then creates an ArrayList of the node names from each path in the list. A second loop is then started to loop through the list of paths again. If the first and second loop are not on the same index, then a loop is started to loop through the second paths nodes. If the name of the node matches the name being looked for then the index is noted, then the loop is broken out of. After this loop there is a check to see if the index being stored is not equal to -1. If this is true, then there is a check t o see if the length of the path – the index of the node before the endNode is greater than 1. If it is then a subPathEndNodeName is created and the endNode of this path is assigned to it. If this is not equal to the endNode that we are looking for then the path is deleted as it is pre-empted. Finally, the list of paths with the paths that were pre-empted removed are returned.

The results of the shortest paths are then outputted and any paths that were not pre-empted or redundant are outputted.

## Testing: