Criterion C: Development

File Handling

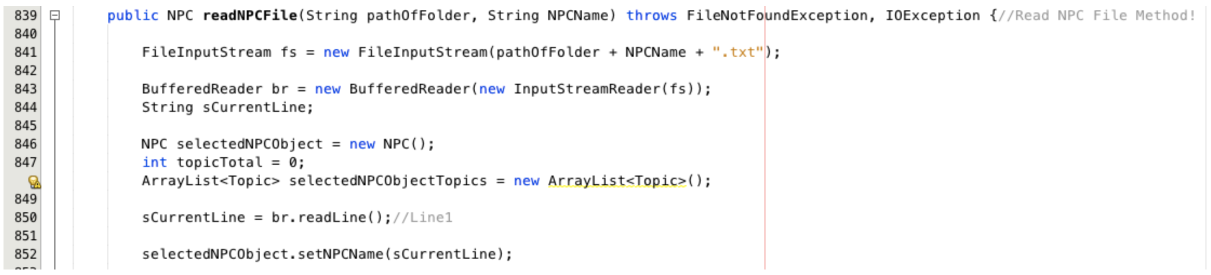
The buffered reader class is used to read data from stored NPC files. This class was chosen for:
- Already implemented in default Java libraries (IO); no need to install external libraries.
- Has a corresponding buffered writer class for ease of writing to files.
- Fast and efficient; does not do special parsing as opposed to the scanner class.

By default, buffered reader tokenizes the input stream with the '\n' delimiter. As such, different sections of data were chosen to be placed in separate lines, with final file storage structure for NPCs being:

NPC Name
NPC Constellation Name
List of Topic Names
List of Topic Interest Levels
List of Topic Favor Levels
NPC Favor Level
NPC Interest Level

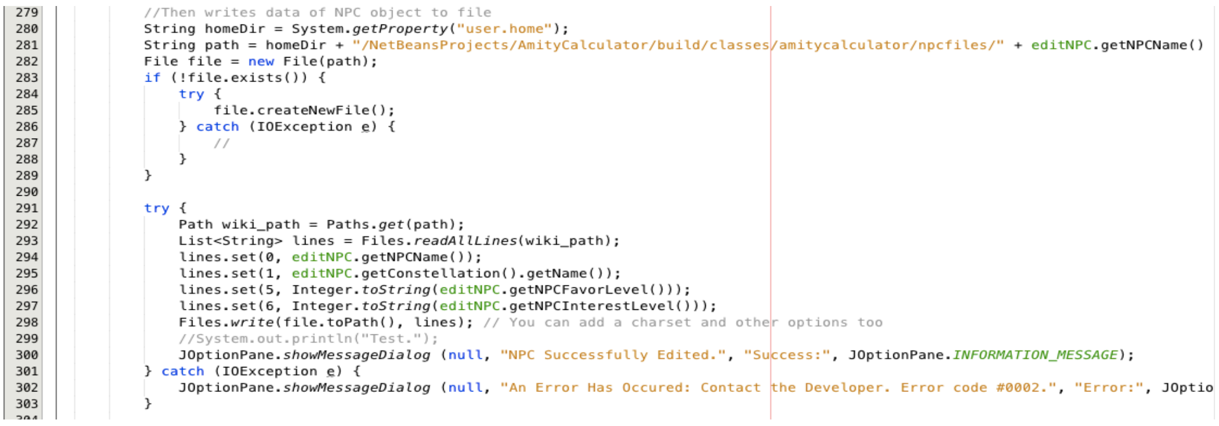
File handling – external storage of data items – was chosen because internal storage of data members in the program during runtime would take up RAM – thus slowing the computer down – as opposed to secondary storage. A method to shift the needed NPC for calculation/editing in and out of memory and secondary storage is instead employed.

Example of file reading and parsing (from readNPCFile method)



```
    public NPC readNPCFile(String pathOfFolder, String NPCName) throws FileNotFoundException, IOException {//Read NPC File Method!

        FileInputStream fs = new FileInputStream(pathOfFolder + NPCName + ".txt");

        BufferedReader br = new BufferedReader(new InputStreamReader(fs));
        String sCurrentLine;

        NPC selectedNPCObject = new NPC();
        int topicTotal = 0;
        ArrayList<Topic> selectedNPCObjectTopics = new ArrayList<Topic>();

        sCurrentLine = br.readLine();//Line1

        selectedNPCObject.setNPCName(sCurrentLine);
```

Example of file writing (from EditNPCFilesUI class main method)

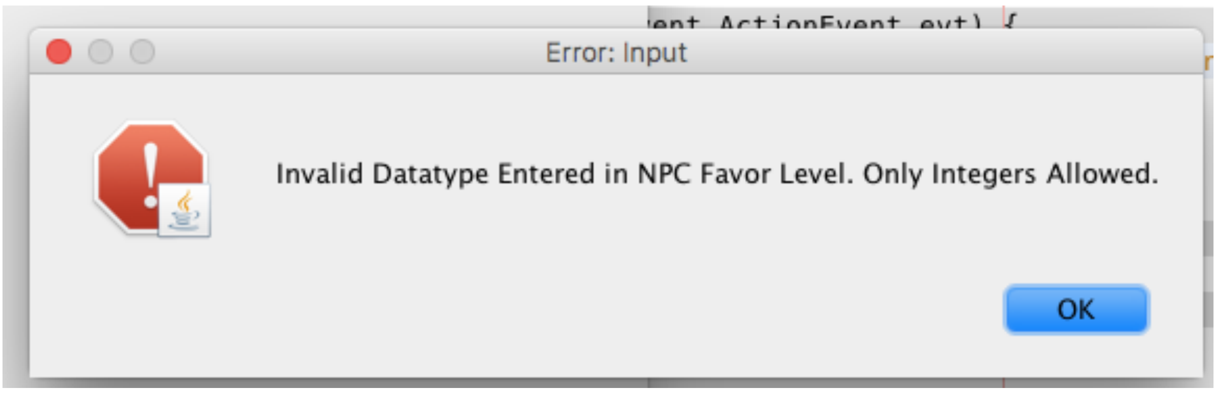


```
        //Then writes data of NPC object to file
        String homeDir = System.getProperty("user.home");
        String path = homeDir + "/NetBeansProjects/AmityCalculator/build/classes/amitycalculator/npcfiles/" + editNPC.getNPCName()
        File file = new File(path);
        if (!file.exists()) {
            try {
                file.createNewFile();
            } catch (IOException e) {
                //
            }
        }

        try {
            Path wiki_path = Paths.get(path);
            List<String> lines = Files.readAllLines(wiki_path);
            lines.set(0, editNPC.getNPCName());
            lines.set(1, editNPC.getConstellation().getName());
            lines.set(5, Integer.toString(editNPC.getNPCFavorLevel()));
            lines.set(6, Integer.toString(editNPC.getNPCInterestLevel()));
            Files.write(file.toPath(), lines); // You can add a charset and other options too
            //System.out.println("Test.");
            JOptionPane.showMessageDialog (null, "NPC Successfully Edited.", "Success:", JOptionPane.INFORMATION_MESSAGE);
        } catch (IOException e) {
            JOptionPane.showMessageDialog (null, "An Error Has Occured: Contact the Developer. Error code #0002.", "Error:", JOptio
        }
```

Exception Handling and Error Notification

Exception notification and handling was employed due to considerations of the consumer base and general user friendliness. The user base is comprised of those who play the game; a recent age survey (http://www.strawpoll.me/6575705/r) places mode player age at 26+. Assuming standard computer literacy distributions among the age group and with consideration of initial client requests towards user friendliness, exception handling and error notifications are employed to ensure that user input does not return program crashes.

Example of a designed user-friendly error notification:

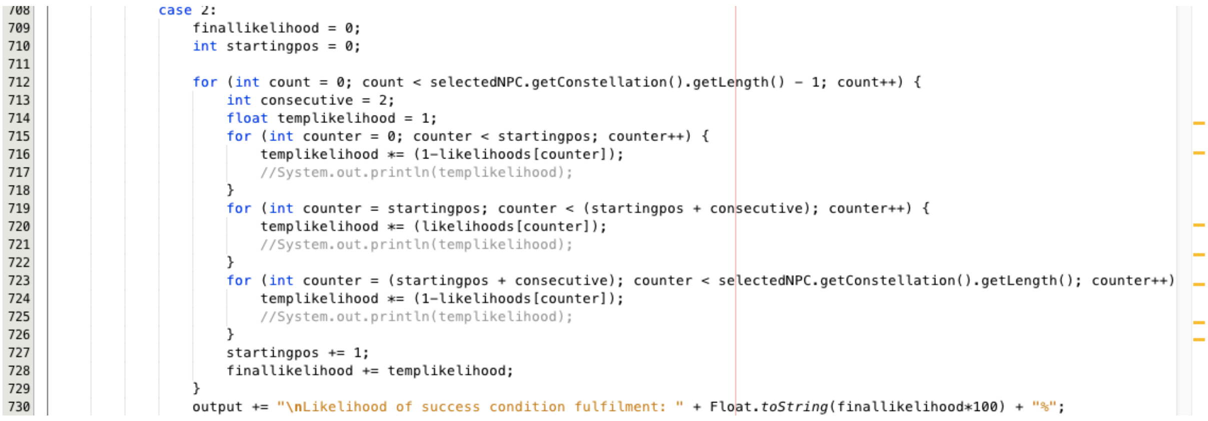

Error: Input

Invalid Datatype Entered in NPC Favor Level. Only Integers Allowed.

Combinational Probability Calculations

The combinational probability method was decided upon because it is easily implemented in a (Java) programming environment using multiple for loops. Methods in JDistlib (an external library) were not employed because doing so would require the initialization of external libraries; because of the mass-distributed and extensible nature of this software, the need to manually install external libraries may serve to increase user frustration. Because of light-weight considerations, bundling the external library together with installation files was deemed unwelcome.

Example of the combinational probability calculation in use:

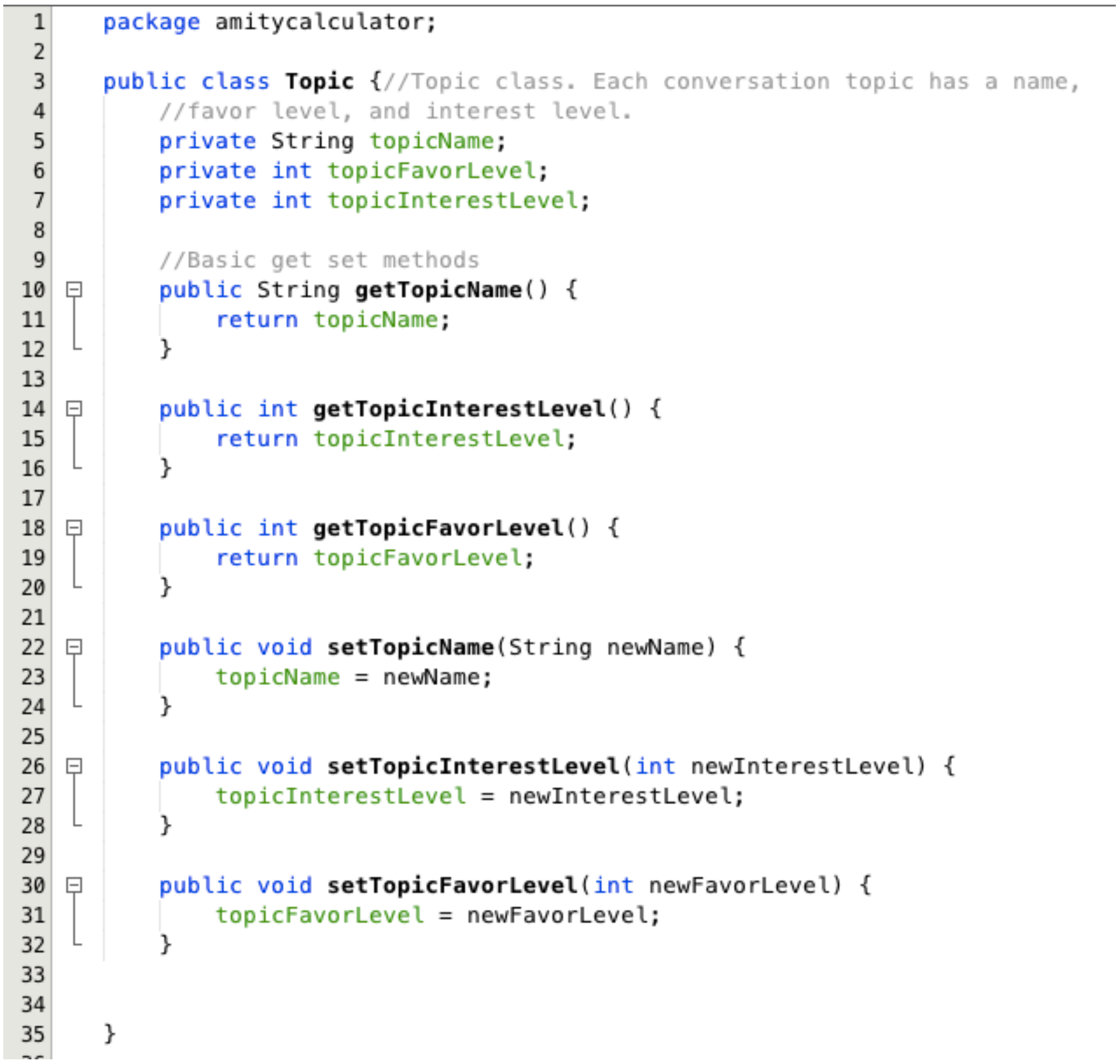

```
            case 2:
                finallikelihood = 0;
                int startingpos = 0;

                for (int count = 0; count < selectedNPC.getConstellation().getLength() - 1; count++) {
                    int consecutive = 2;
                    float templikelihood = 1;
                    for (int counter = 0; counter < startingpos; counter++) {
                        templikelihood *= (1-likelihoods[counter]);
                        //System.out.println(templikelihood);
                    }
                    for (int counter = startingpos; counter < (startingpos + consecutive); counter++) {
                        templikelihood *= (likelihoods[counter]);
                        //System.out.println(templikelihood);
                    }
                    for (int counter = (startingpos + consecutive); counter < selectedNPC.getConstellation().getLength(); counter++)
                        templikelihood *= (1-likelihoods[counter]);
                        //System.out.println(templikelihood);
                    }
                    startingpos += 1;
                    finallikelihood += templikelihood;
                }
                output += "\nLikelihood of success condition fulfilment: " + Float.toString(finallikelihood*100) + "%";
```

Classes and OOP Techniques

Encapsulation was used in many instances to:

- Hide the unnecessary complexities of internal processing to the user
- Easy establishment of permissions and exception handling (to be seen in the following section)
- Define clear parameters for range of user input (e.g. datatype)

Example of Encapsulation (Topic Class)

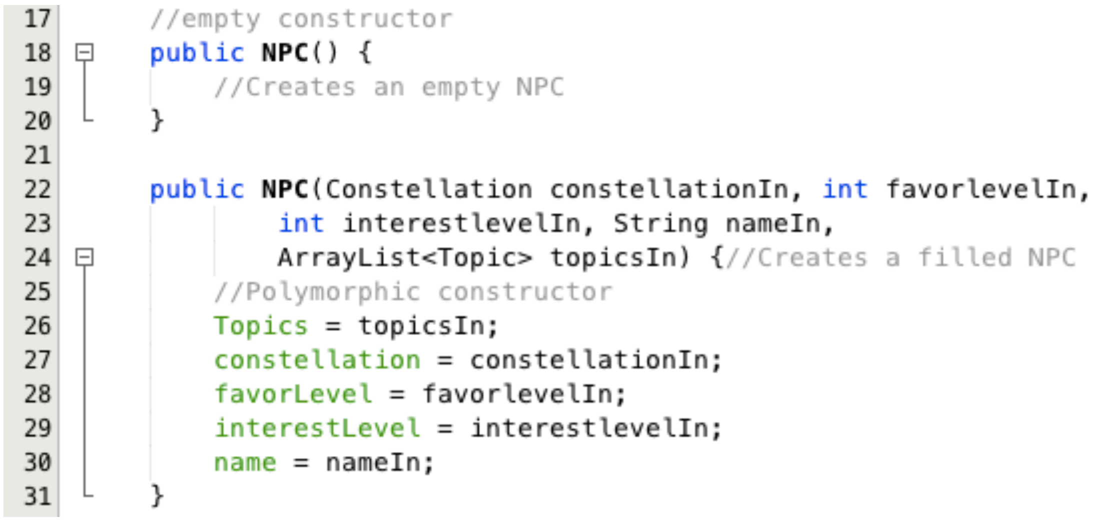


```java
package amitycalculator;

public class Topic {//Topic class. Each conversation topic has a name,
    //favor level, and interest level.
    private String topicName;
    private int topicFavorLevel;
    private int topicInterestLevel;

    //Basic get set methods
    public String getTopicName() {
        return topicName;
    }

    public int getTopicInterestLevel() {
        return topicInterestLevel;
    }

    public int getTopicFavorLevel() {
        return topicFavorLevel;
    }

    public void setTopicName(String newName) {
        topicName = newName;
    }

    public void setTopicInterestLevel(int newInterestLevel) {
        topicInterestLevel = newInterestLevel;
    }

    public void setTopicFavorLevel(int newFavorLevel) {
        topicFavorLevel = newFavorLevel;
    }


}
```

Object oriented programming was chosen to reduce redundancy of information and data; especially important in this software where a large potential of data could be required to be in storage (e.g. a potentially unlimited number of NPCs, as the game is constantly expanding).

Polymorphism was used, particularly, in the constructor of the NPC class. This is to accommodate for different usage (different types of parameters are passed to construct an NPC object in different circumstances) in editing and creation. Creating an NPC has calls to the full constructor method when supplied all the necessary parameters, whereas editing has calls to the empty constructor when reading from the stored file sequentially and relevant set methods to allow reading and editing of the NPC should the data file be corrupted and some information be lost.

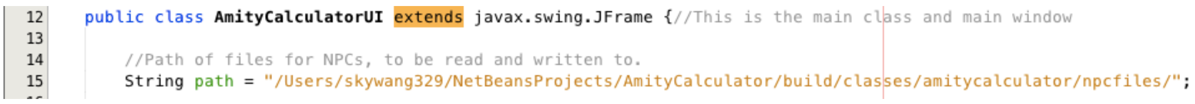
Screenshot of the two constructors:



```
    //empty constructor
    public NPC() {
        //Creates an empty NPC
    }

    public NPC(Constellation constellationIn, int favorlevelIn,
            int interestlevelIn, String nameIn,
            ArrayList<Topic> topicsIn) {//Creates a filled NPC
        //Polymorphic constructor
        Topics = topicsIn;
        constellation = constellationIn;
        favorLevel = favorlevelIn;
        interestLevel = interestlevelIn;
        name = nameIn;
    }
```

Inheritance is used for all UI windows to extend from the internal swing library classes, utilizing pre-built swing library classes to ease manual construction of the UI.

Example screenshot of inheritance of the JFrame class in the Swing library:



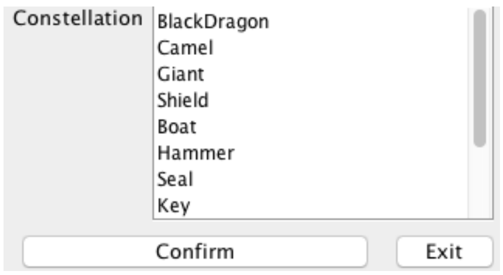
```
public class AmityCalculatorUI extends javax.swing.JFrame {//This is the main class and main window

    //Path of files for NPCs, to be read and written to.
    String path = "/Users/skywang329/NetBeansProjects/AmityCalculator/build/classes/amitycalculator/npcfiles/";
```

UI Features

Error handling and notification is mentioned above.

Swing.JLists are used as to define and simplify range of user input and increase user friendliness in many instances whereby user input is limited to a selection of data items. Such instances include NPC (main window), success condition list (main window), and constellation of NPC (editing NPC + create new NPC window).

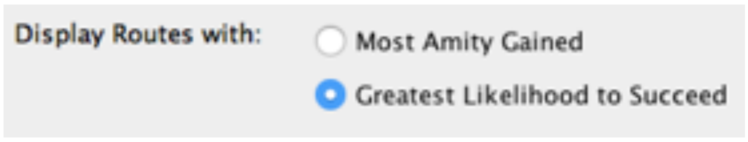
A screenshot of an employed JList is as follows (Constellation selection):



The swing library was chosen for design of the UI in (particularly over AWT) consideration of cross-platform extensibility. Swing is 100% cross-platform portable; whereas AWT is a Java interface in concordance native system GUI code – in other words, what is displayed will not be 100% same across all operating systems.

Radio buttons and button groups are used too to simplify (as well as define and limit) the range of user input and increase user friendliness. In particular, when selecting the desired outcome (most amity gained/most likely to succeed), a binary choice is present: only one is to be selected for final calculations.

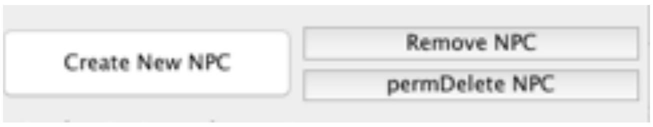
Screenshot:



Display Routes with: Most Amity Gained [✓ Greatest Likelihood to Succeed]

An option to delete/permanent delete was distinguished as per client request. This distinction was employed as to allow the user flexibility in decision: if the user is unsure of whether or not he/she is to use the deleted file in the future, simple delete gives the user the option to recover the file. To employ this, simple delete hides the file with the dot-file convention, whereas permDelete deletes the file – this method was chosen in consideration of cross-platform independence and compatibility.

Screenshot of UI:



(Word Count: 858)