```java
/**
 * GameMap.java
 * Written By: Lauren Luo & Adrianna Valle
 * Written On: Dec 16, 2016
 *
 * Game map creates the conditions for a game using information that a user enters for the
 * key text responses, the door text responses, the images, etc. Note that the rooms and weights do not correspond
 * nor do they need to be in a certain order with regards to the map.
 *
 * This class allows you to create a game in which the weights of the entered .tgf file corresponds
 * to a specified door reference. If the door contains a weight of zero, that mean the door is unlocked
 * and no further action is required.
 *
 * TRAVERSING:Every key value present contains a corresponding door value in which they interact together depending on
 * the decisions the user makes. These values are saved in their own dictionary. Every room contains keys that corresponds
 * to all of the weighted edges[represented as doors]that connect to other rooms. In total, every room contains
 * 10 keys where some keys correspond to a door that connects from the room as some that are used merely to challenge
 * the player.
 *
 * ORIENTATION: The number value assigned to the room name is used to help the user orient themselves around the game.
 * The structure is maze like and therefore, the user can not enter the same room various times in order to find the end.
 *
 * HOW TO WIN: The player does not necessarily need to unlock all the doors but must reach the assigned final room and
 * complete the minigame. What's more, the player needs to not run out of chances or the lose the game.
 *
 * The user of this class MUST:
 * #1Create a .tgf with vertices that correspond to a saved image file name.
 * #2 Not include multiple edges of the same weight unless it is of weight 0.
 * #3 Must create the Needed Text file for the Key and Dictionary in the exact format specified throughout the program.
 *
 * OVERALL GAME EXPANSION:
 * 1--> Control more of the IO input.
 * 2--> Automatically determine best starting and final room ad sets it
 * 3--> Use polymorphism to create diversity in key door interaction.
 * 4--> More game features such as mingames, player character
 */
import java.util.*;
import java.io.*;

public class GameMap {
  //instance variables
  private AdjMatGraph<String> map;
  private Room currentRoom, finalRoom;
  private LinkedList<Room> allRooms;
  private Hashtable<Room, LinkedList<Room>> rooms;
  private static Hashtable<Integer, Key> allKeys; //will contain 25 keys
  private static Hashtable<Integer, Door> allDoors;
  private int chancesLeft;
  private static final int MAX_CHANCES = 3;

  public GameMap() {
    map = new AdjMatGraph<String>();
    try {
      AdjMatGraph.loadTGF("gameMap.tgf", map);   //Check if valid tgf file
    } catch (FileNotFoundException ex) {
      System.out.println("error: file not found");
    }

    chancesLeft = MAX_CHANCES;
    allRooms = new LinkedList<Room>();
    rooms = new Hashtable<Room, LinkedList<Room>>();
    allKeys = new Hashtable<Integer, Key>();
    allDoors = new Hashtable<Integer, Door>();

    buildKeyDict("keyText.txt");
    buildDoorDict("doorText.txt");
    buildRooms();

    setCurrentRoom(findRoom("1room"));
    setFinalRoom(findRoom("10room"));
  }

  /** Builds the key Dictionary out of a text file that contains the key name, a message to give
    * when user first interacts with it and a message it gives if it is not one of the
    * designated keys for the room.
    *
    * EXPANSION CAPABILITIES: Allows for the key dictionary to expand increasing the possible map size.
    * ASSUMPTIONS: Assumes the user enters a vaild format of the necessary text.
```

```
   Note some special characters
79      * are not accepted.
80      * FUTURE EXPANSION:Could allow for a scanner option for input and resolve
   special character issue
81      *
82      * @param: String TextFile @return: none*/
83    private void buildKeyDict(String fileIn) {
84      try {
85        Scanner sc = new Scanner(new File(fileIn));
86        int count = 1;
87
88        while (sc.hasNextLine()) {
89
90          String[] temp = sc.nextLine().trim().split(" \\+ ");
91          allKeys.put(count, new Key(temp[0], temp[1], temp[2]));
92          count++;
93        }
94        sc.close();
95      } catch (IOException ex) {
96        System.out.println("error in reading keys from file");
97      }
98    }
99
100   /** Builds the Door Dictionary out of a text file that contains a messsage when
   the door
101     * is locked and a message it returns when it is sucessfully unlocked by a key.
102     *
103     * EXPANSION CAPABILTIES: Allows for the number of doors to increase therefore
   the number
104     * of connections between rooms.
105     * ASSUMPTIONS: Assumes user formatted the file correctly. Note some special
   characters are not accepted.
106     * FUTURE EXPANSION: Could allow for a scanner option for input and could allow
   for special characters.
107     *
108     * @param: String TextFile @return: none*/
109   private void buildDoorDict(String fileIn) {
110     try {
111       Scanner sc = new Scanner(new File(fileIn));
112       int count = 0;
113       while (sc.hasNextLine()) {
114         String[] temp = sc.nextLine().trim().split(" \\+ ");
115         allDoors.put(count, new Door(count, temp[0], temp[1]));
116         count++;
117       }
118       sc.close();
119     } catch (IOException ex) {
120       System.out.println("error in reading doors from file");
121     }
122   }
123
124   /**Builds the rooms from the tgf vertices where all of the names correspond to a
   saved img. In building
125     * the rooms also finds the keys that must go in the room[determined by the
   weighted edges which are
126     * equivalent to the doors] from allKeys and stores it. Also, creates a
   dictionary of the rooms, and one
127     * for the connecting rooms to refer to.
128     * EXAPNSION CAPABILITIES: Allows you to build lots of rooms
129     * ASSUMPTIONS: User enters correct img filename.
130     * FUTURE EXPANSION: It may be better to organize our rooms in a different
   structure.*/
131   private void buildRooms() {
132     //Loads the img names[without '.jpg'] onto an array
133     String[] vertices = new String[map.n()];
134     for (int h = 0; h < map.n(); h++) {
135       vertices[h] = map.getVertex(h);
136     }
137
138     //Gets the connecting room names for each room
139     for (int i = 0; i < map.n(); i++) {
140       LinkedList<String> successors = map.getSuccessors(vertices[i]);
141
142       //Loads the keys that correspond to any of the locked doors that connect from
   the room.
143       LinkedList<Key> activeKeys = new LinkedList<Key>();
144       for (int j = 0; j < successors.size(); j++) {
145         int keyNum = map.getWeight(vertices[i], successors.get(j));
146         if(keyNum!=0)                                    //no key is loaded for a weight
   of zero.
147           activeKeys.add(allKeys.get(keyNum));
148
149       }
150
151       String fileName = vertices[i] + ".jpg";
152       allRooms.add(new Room(fileName, activeKeys)); //Creates the room with
   coresponding img name and keys & stores it.
153     }
154
155     //Creates a Dictionary of all the rooms with their connecting rooms
156     for (int k = 0; k < map.n(); k++) {
157       LinkedList<String> successors = map.getSuccessors(vertices[k]);
158       LinkedList<Room> successorRooms = new LinkedList<Room>();
159
```

```
160       for (int m = 0; m < successors.size(); m++) {
161         successorRooms.add(findRoom(successors.get(m)));
162       }
163       rooms.put(findRoom(vertices[k]), successorRooms);
164     }
165   }
166
167   /*SETTER: Allows the user to set the starting room. Setting also allows for
168    * a room traversal to be simulated as the player enters different rooms.
169    *
170    * EXPANSION CAPABILITES: Can choose any room to be the first room.
171    * ASSUPTIONS: none
172    * FUTURE EXPANSION: can select any room at random and from this, use this point
   of reference
173    * to create pointer to final room
174    * @param: Room selectedRoom @return: -- */
175   public void setCurrentRoom(Room selectedRoom) {
176     currentRoom = selectedRoom;
177   }
178
179   /**GETTER: Returns the current room
180     * @param: -- @reuturn: Room currentRoom */
181   public Room getCurrentRoom() {
182     return currentRoom;
183   }
184
185   /*SETTER: Allows the user to set the final room.
186    *
187    * EXPANSION CAPABILITES: Can choose any room to be the final room.
188    * ASSUPTIONS: The room isn't close enough to the start where it will end the
   game to quickly
189    * FUTURE EXPANSION: Using a traversal to find a far enoguh room to be the future
   room. User won't need to
190    * manually enter it.
191    * @param: Room selectedRoom @return: -- */
192   public void setFinalRoom(Room selectedRoom) {
193     finalRoom = selectedRoom;
194   }
195
196   /**GETTER: Returns the final room
197     * @param: -- @reuturn: Room finalRoom */
198   public Room getFinalRoom() {
199     return finalRoom;
200   }
201
202   /** Returns the Room with the corresponding room name. If no room is found,
   returns null
203     * NOTE: When using findRoom, check if null before assigning it anywhere.
204     * @param: String roomName @return: Room correspondingRoom */
205   private Room findRoom(String roomName) {
206     for (int i = 0; i < map.n(); i++) {
207       if (allRooms.get(i).getRoomName().equals(roomName)) {
208         return allRooms.get(i);
209       }
210     }
211     return null;
212   }
213
214   /** Returns the room that is in the list of the connecting rooms to the current
   room. If the roomName
215     * is not a corresponding room, returns null.
216     * Note: When using getRoom, check if null before assigning it anywhere.
217     * @param: String roomName @return: Room correspondingRoom*/
218   public Room getRoom(String roomName) {
219     for (int i = 0; i < getConnectingRooms().size(); i++) {
220       if (getConnectingRooms().get(i).getRoomName().equals(roomName+"room")) {
221         return getConnectingRooms().get(i);
222       }
223     }
224     return null;
225   }
226
227   /** Returns a list of the rooms that connect to the current room.
228     * @param: -- @return: LinkedList<Room> connectingRooms */
229   public LinkedList<Room> getConnectingRooms() {
230     return rooms.get(currentRoom);
231
232   }
233
234   /** Returns true if user player reaches the final room.
235     * @param: -- @return: boolean foundFinalRoom */
236   public boolean endOfMap(){
237     return currentRoom.equals(finalRoom);
238   }
239
240
241   /**GETTER: Returns a randomly selected key from the key Dictionary.Used to fill
   the spots of the remaining
242     * keys needed[to make 10] in order to challenge the user.
243     * @param: -- @return: Key randomKey */
244   public static Key getRandomKey() {
245     int ran = (int)((Math.random() * 25) + 1);
246     return allKeys.get(ran);
247   }
```

```
248
249   /**Returns all of the keys that exist. Static in order to be accessed by the room
    class.
250     * @param: -- @return: Hashtable<Integer, Key> allKeys */
251   public static Hashtable<Integer, Key> getAllKeys() {
252     return allKeys;
253   }
254
255   /**Returns the Door of that connects the currentRoom with the selectedRoom by
    getting the weight
256     * and pulling the assigned integer door value from the door dictionary.
257     * @param: Room selectedRoom @return: Door connectingDoor */
258   public Door getDoor(Room selectedRoom) {
259     int doorNum = map.getWeight(currentRoom.getRoomName(), selectedRoom.
    getRoomName());
260     return allDoors.get(doorNum);
261   }
262
263   /** Replaces the door poniter to that of door zero to simulate the door being
    unlocked.
264     * After this, the door can be entered however many times and would still be
    unlocked.
265     * @param: @return: -- */
266   public void unlockDoor(Room lockedRoom) {
267     map.addEdge(currentRoom.getRoomName(),lockedRoom.getRoomName(),0);
268   }
269
270
271   /** Returns whether the player still have chances to guess wrong.
272     * @return: boolean chancesExist @param: -- */
273   public boolean chancesLeft() {
274     return chancesLeft > 0;
275   }
276
277   /**Returns the number of chances left to guess wrong.
278     * @param: -- @return: int numberOfChances */
279   public int getChances() {
280     return chancesLeft;
281   }
282
283   /** Decrements the number of chances when the player guesses incorrectly.
284     * @param: --  @return: -- */
285   public void wrongAnswer() {
286     chancesLeft--;
287   }
288
289   /** Prints a string representation of the possible rooms the player can enter.
    Only used for the Driver
290     * class.*/
291   public String printRooms() {
292     String s = "";
293     LinkedList<Room> neighbors = getConnectingRooms();
294     for (int i = 0; i < neighbors.size(); i++) {
295       String name = neighbors.get(i).getRoomName();
296       s += "[" + name.substring(0,name.indexOf("r"));
297       if (map.getWeight(currentRoom.getRoomName(),neighbors.get(i).getRoomName())
    == 0) {
298         s += "+";
299       } else {
300         s += "-";
301       }
302       s += "]";
303     }
304     return s;
305   }
306
307   /** Prints a string representation of the possible keys a player can enter. Only
    used for the Driver Class. */
308   public String printKeys() {
309     String s = "";
310     Key[] currentKeys = currentRoom.getRoomKeys();
311     for (int i = 0; i < currentKeys.length; i++) {
312       s += "[" + currentKeys[i].getName() + "]";
313     }
314     return s;
315   }
316
317   public static void main(String[] args) {
318     //Testing AdjMatGraph with game map graph, with (test) and without weights
    (test2)
319 //     AdjMatGraph<String> test = new AdjMatGraph<String>();
320 //     try {
321 //       AdjMatGraph.loadTGF("gameMapNoWeights.tgf", test);
322 //     } catch (FileNotFoundException ex) {
323 //       System.out.println("error: file not found");
324 //     }
325 //     System.out.println(test);
326
327 //     AdjMatGraph<String> test2 = new AdjMatGraph<String>();
328 //     try {
329 //       AdjMatGraph.loadTGF("gameMap.tgf", test2);
330 //     } catch (FileNotFoundException ex) {
331 //       System.out.println("error: file not found");
332 //     }
```

```
333 //    System.out.println(test2);
334
335 //  GameMap map = new GameMap();
336
337 //    map.printAllKeys();
338 //    System.out.println();
339 //    map.printAllDoors();
340 //    System.out.println();
341 //    map.printAllRoomNeighbors();
342 //    System.out.println();
343 //    map.printRooms();
344 //    map.printMapGraph();
345 //    System.out.println(map.contains(4));
346   }
347 }
```