

```
1 //Game.java
2
3 import java.awt.Font;
4 import java.awt.Graphics;
5 import java.awt.event.KeyAdapter;
6 import java.awt.event.KeyEvent;
7
8 import javax.swing.JFrame;
9 import javax.swing.JPanel;
10
11 import misc.Util;
12 import cells.Cell;
13 import cells.Dungeon;
14 import entities.Player;
15
16
17 public class Game extends JPanel
18 {
19     private static final int VIEW_DISTANCE = 8;
20     private static final long serialVersionUID = -824413027067294540L;
21
22     //Uses Singleton design pattern
23     private static Game instance = getInstance();
24
25     Dungeon d = new Dungeon();
26     Player p = new Player();
27     Game() {
28         init();
29     }
30
31     //Uses Singleton design pattern
32     public static Game getInstance() {
33         if(instance != null)
34             return instance;
35         instance = new Game();
36         return instance;
37     }
38
39     private void init() {
40         //init Game variables
41         d = new Dungeon();
42         p.x = d.getStartX();
43         p.y = d.getStartY();
44         d.cells[p.x][p.y].setLivingEntity(p);
45
46         //init GUI stuff
47         JFrame container = new JFrame("RogueScape");
48         container.addKeyListener(new GameKeyHandler());
49         container.setContentPane(this);
50         container.setSize(640, 480);
51         container.setVisible(true);
52     }
```

```
53
54     public static void movePlayer(int direction) {
55         //uses helper functions in Util class to determin new location
56         int x = instance.p.x + Util.getDX(direction);
57         int y = instance.p.y + Util.getDY(direction);
58         //checks if cell is in array bounds
59         //    if cell is walkable
60         //        and if cell isn't already occupied
61         if(Util.isValidLocation(x, y, instance.d.SIZE) &&
62 ... instance.d.cells[x][y].isWalkable() && instance.d.cells[x][y].setLivingEntity(instance.p))
63 ...
64     {
65         instance.d.cells[instance.p.x][instance.p.y].clearLivingEntity();
66         instance.p.x = x;
67         instance.p.y = y;
68     }
69
70     //duh.. gets dat der cell dat has duh p'ayer init
71     public Cell getCellContainingPlayer() {
72         return d.cells[p.x][p.y];
73     }
74
75     //Prints from VIEW_DISTANCE cells north-west of the player to
76     //VIEW_DISTANCE cells south-east, accounting for array bounds
77     @Override
78     public void paintComponent(Graphics g) {
79         super.paintComponent(g);
80         Font backup = g.getFont();
81         g.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 24));
82         int x = p.x - VIEW_DISTANCE;
83         int y = p.y - VIEW_DISTANCE;
84         for(int i=1; i<(VIEW_DISTANCE*2 + 2); i++)
85             for(int j=1; j<(VIEW_DISTANCE*2 + 2); j++)
86             {
87                 if((x+i) < 0 || (y+j) < 0 || (x+i) >= d.SIZE || (y+j) >= d.SIZE)
88                     g.drawString("~", i*24, j*24);
89                 else
90                     g.drawString(""+d.cells[x+i][y+j].getRepresentation(), i*24, j*24);
91             }
92         g.setFont(backup);
93         this.repaint();
94     }
95
96     //Handles all keyboard input
97     private class GameKeyHandler extends KeyAdapter {
98         public GameKeyHandler()
99         {
100             super();
101             // TODO Auto-generated constructor stub
102         }
103         @Override
```

```
103     public void keyPressed(KeyEvent e) {
104         if(e.getKeyCode() == KeyEvent.VK_LEFT || e.getKeyCode() == KeyEvent.VK_H)
105             Game.movePlayer(Util.WEST);
106         else if(e.getKeyCode() == KeyEvent.VK_RIGHT || e.getKeyCode() ==
... KeyEvent.VK_L)
107             Game.movePlayer(Util.EAST);
108         else if(e.getKeyCode() == KeyEvent.VK_DOWN || e.getKeyCode() ==
... KeyEvent.VK_J)
109             Game.movePlayer(Util.SOUTH);
110         else if(e.getKeyCode() == KeyEvent.VK_UP || e.getKeyCode() == KeyEvent.VK_K)
111             Game.movePlayer(Util.NORTH);
112         else if(e.getKeyCode() == KeyEvent.VK_PERIOD)
113             Game.descendFloor();
114         else if(e.getKeyCode() == KeyEvent.VK_COMMA)
115             Game.ascendFloor();
116
117     }
118 }
119
120 public static void main(String[] args) {
121     Game instance = Game.getInstance();
122 }
123
124 public static void descendFloor()
125 {
126     if(instance.getCellContainingPlayer().getDefaultRepresentation() == '>')
127     {
128         instance.d = new Dungeon();
129         instance.p.x = instance.d.getStartTime();
130         instance.p.y = instance.d.getStartTime();
131         instance.getCellContainingPlayer().setLivingEntity(instance.p);
132         instance.repaint();
133     }
134 }
135
136 public static void ascendFloor()
137 {
138     if(instance.getCellContainingPlayer().getDefaultRepresentation() == '<')
139     {
140         instance.d = new Dungeon();
141         instance.p.x = instance.d.getEndTime();
142         instance.p.y = instance.d.getEndTime();
143         instance.getCellContainingPlayer().setLivingEntity(instance.p);
144         instance.repaint();
145     }
146 }
147 }
148
149 //cells.Cell.java
150 package cells;
151
152 import misc.Representation;
```

```
153 import entities.Entity;
154 import entities.Inventory;
155 import entities.ItemEntity;
156 import entities.LivingEntity;
157
158
159 public class Cell extends Entity
160 {
161     protected Inventory items = new Inventory();
162     protected LivingEntity livingEntity = null;
163     protected boolean solid = false;
164     protected boolean liquid = false;
165     protected boolean walkable = true;
166
167     Cell() {
168     }
169
170     Cell(char ascii) {
171         this.DEFAULT REPRESENTATION = new Representation(ascii);
172     }
173
174     public LivingEntity getLivingEntity() {
175         return livingEntity;
176     }
177
178     public boolean setLivingEntity(LivingEntity e) {
179         if(livingEntity != null)
180             return false;
181         livingEntity = e;
182         return true;
183     }
184
185     public ItemEntity getItem() {
186         return items.get(0);
187     }
188
189     public boolean isWalkable() {
190         return walkable;
191     }
192
193     @Override
194     public char getRepresentation() {
195         if(livingEntity != null)
196             return livingEntity.getRepresentation();
197         if(items.size() > 0)
198             return items.getFirst().getRepresentation();
199         return this.DEFAULT REPRESENTATION.ASCII;
200     }
201
202     public void setRepresentation(char ASCII) {
203         DEFAULT REPRESENTATION.setASCII(ASCII);
204     }
```

```
205
206     public void addItem(ItemEntity item) {
207         items.add(item);
208     }
209
210     public void clearLivingEntity()
211     {
212         this.livingEntity = null;
213     }
214 }
215
216 //cells.Cells.java
217 package cells;
218
219 import java.util.HashMap;
220
221 public class Cells
222 {
223     private static HashMap<Integer, CellTemplate> cells = new HashMap<Integer, CellTemplate>();
224 ...
225
226     public static final CellTemplate AIR = initAIR();
227     public static final CellTemplate GRANITE_WALL = initGRANITE_WALL();
228
229     public static final CellTemplate STAIRS_UP = initSTAIRS_UP();
230     public static final CellTemplate STAIRS_DOWN = initSTAIRS_DOWN();
231
232     public static final CellTemplate OPEN_DOOR = initOPEN_DOOR();
233     public static final CellTemplate CLOSED_DOOR = initCLOSED_DOOR();
234
235     private static final CellTemplate initAIR() {
236         CellTemplate template = new CellTemplate(' ', (int)' ');
237         template.walkable = true;
238         template.solid = false;
239         template.liquid = false;
240         cells.put(template.TEMPLATE_ID, template);
241         return template;
242     }
243
244     private static final CellTemplate initGRANITE_WALL() {
245         CellTemplate template = new CellTemplate('#', (int)'#');
246         template.walkable = false;
247         template.solid = true;
248         template.liquid = false;
249         cells.put(template.TEMPLATE_ID, template);
250         return template;
251     }
252
253     private static final CellTemplate initSTAIRS_UP() {
254         CellTemplate template = new CellTemplate('<', (int)'<');
255         template.walkable = true;
```

```
256     template.solid = false;
257     template.liquid = false;
258     cells.put(template.TEMPLATE_ID, template);
259     return template;
260 }
261
262 private static final CellTemplate initSTAIRS_DOWN() {
263     CellTemplate template = new CellTemplate('>', (int)'>');
264     template.walkable = true;
265     template.solid = false;
266     template.liquid = false;
267     cells.put(template.TEMPLATE_ID, template);
268     return template;
269 }
270
271 private static final CellTemplate initOPEN_DOOR() {
272     CellTemplate template = new DoorTemplate(',', (int)',');
273     cells.put(template.TEMPLATE_ID, template);
274     return template;
275 }
276
277 private static final CellTemplate initCLOSED_DOOR() {
278     CellTemplate template = new CellTemplate('+', (int)'+' );
279     cells.put(template.TEMPLATE_ID, template);
280     return template;
281 }
282
283 public static CellTemplate getTemplate(int TEMPLATE_ID)
284 {
285     return cells.get(TEMPLATE_ID);
286 }
287
288 public boolean contains(int TEMPLATE_ID) {
289     if(cells.containsKey(TEMPLATE_ID))
290         return true;
291     return false;
292 }
293 }
294
295 //cells.CellTemplate.java
296 package cells;
297
298 import miscRepresentation;
299
300 public class CellTemplate extends Cell
301 {
302     public final int TEMPLATE_ID;
303
304     CellTemplate(char ASCII, int TEMPLATE_ID) {
305         this.DEFAULT_REPRESENTATION = new Representation(ASCII, false);
306         this.TEMPLATE_ID = TEMPLATE_ID;
307     }
```

```
308     }
309
310     public Cell makeInstance() {
311         Cell instance = new Cell(this.DEFAULT REPRESENTATION.ASCII);
312         instance.solid = this.solid;
313         instance.liquid = this.liquid;
314         instance.walkable = this.walkable;
315
316         return instance;
317     }
318 }
319
320
321 //cells.DoorTemplate.java
322 package cells;
323
324 import misc.Openable;
325
326 //An example of a cell type with greater complexity
327 public class DoorTemplate extends CellTemplate
328 {
329
330     DoorTemplate(char ASCII, int TEMPLATE_ID) {
331         super(ASCII, TEMPLATE_ID);
332     }
333
334     @Override
335     public Cell makeInstance() {
336         if(this.DEFAULT REPRESENTATION.ASCII == ',')
337             return new Door(true);
338         return new Door(false);
339     }
340
341     private class Door extends Cell implements Openable {
342         private boolean isOpen = false;
343         Door(boolean isOpen) {
344             this.isOpen = isOpen;
345             this.solid = true;
346             this.liquid = false;
347         }
348
349         @Override
350         public char getRepresentation() {
351             if(this.isOpen())
352                 return ',';
353             return '+';
354         }
355
356         @Override
357         public boolean isWalkable() {
358             return this.isOpen();
359         }

```

```
360
361     @Override
362     public boolean isOpen()
363     {
364         return isOpen;
365     }
366
367     @Override
368     public String open()
369     {
370         this.isOpen = true;
371         return "The door swings open.";
372     }
373
374     @Override
375     public String close()
376     {
377         this.isOpen = false;
378         return "The door slams shut.";
379     }
380 }
381
382
383
384 //cells.Dungeon.java
385 package cells;
386
387 import java.util.ArrayList;
388 import java.util.List;
389
390 import misc.Util;
391
392 public class Dungeon extends Grid
393 {
394     //How many rooms per floor
395     public static final int MAX_ROOMS = 10;
396     public static final int MIN_ROOMS = 6;
397     //How big/small the rooms can be
398     public static final int MAX_SIZE = 10;
399     public static final int MIN_SIZE = 4;
400     //How many other rooms each room should be connected to
401     public static final int MIN_CONNECTIONS = 2;
402     //How far from the edges of the array bounds rooms can be generated
403     private static final int DUNGEON_EDGE_BUFFER = 4;
404
405     //Used to remember where stairs up/down are place
406     int stairsUpX = -1;
407     int stairsUpY = -1;
408     int stairsDownX = -1;
409     int stairsDownY = -1;
410
411     //Holds the rooms contained in the dungeon :P
```

```
412     private ArrayList<Dungeon.Room> rooms = new ArrayList<Dungeon.Room>();  
413     //Total number of rooms there should be, post-generation  
414     private int numRooms;  
415     public Dungeon() {  
416         super();  
417         //Fill in everything with granite  
418         Grid.fillCells(this, Cells.GRANITE_WALL);  
419         //Figure out how many rooms we'll be making  
420         numRooms = Util.randomInt(MIN_ROOMS, MAX_ROOMS);  
421         //Carve out those rooms, assuring that no rooms overlap  
422         makeRooms();  
423         //Carve out corridors connection those rooms  
424         connectRooms();  
425         //Make stairs up and down  
426         makeStairs();  
427     }  
428  
429     private void makeRooms() {  
430         int start_x, start_y, width, height;  
431         //While we haven't generated all of our rooms  
432         while(rooms.size() < numRooms) {  
433             Room newRoom;  
434             //Generate room dimensions  
435             width = Util.randomInt(MIN_SIZE, MAX_SIZE);  
436             height = Util.randomInt(MIN_SIZE, MAX_SIZE);  
437             do {  
438                 //Find a place to put such a room  
439                 start_x = Util.randomInt(DUNGEON_EDGE_BUFFER, cells.length - MAX_SIZE -  
... DUNGEON_EDGE_BUFFER);  
440                 start_y = Util.randomInt(DUNGEON_EDGE_BUFFER, cells.length - MAX_SIZE -  
... DUNGEON_EDGE_BUFFER);  
441                 newRoom = new Room(start_x, start_y, width, height);  
442             } while(newRoom.intersects(rooms));  
443             //Carve out room  
444             newRoom.makeRoom(this);  
445             //Add it to the list of rooms  
446             rooms.add(newRoom);  
447         }  
448     }  
449     //Carves out corridors connecting rooms  
450     private void connectRooms() {  
451         //for each room  
452         for(Room room: rooms) {  
453             //while the room hasn't reached the minimum number of connections  
454             while(room.connections < MIN_CONNECTIONS) {  
455                 //find another room  
456                 Room otherRoom;  
457                 do {  
458                     otherRoom = rooms.get(Util.randomInt(0, rooms.size()-1));  
459                 } while(room == (otherRoom));  
460                 //pick appropriate places to connect a corridor between the two
```

```
462         int start_x = room.start_x;
463         int start_y = room.start_y;
464         int end_x = otherRoom.start_x;
465         int end_y = otherRoom.start_y;
466
467         if(room.isAbove(otherRoom))
468     {
469             start_x += room.width/2;
470             start_y += room.height;
471             end_x += otherRoom.width/2;
472         }else if(room.isBelow(otherRoom)) {
473             start_x += room.width/2;
474             end_x += otherRoom.width/2;
475             end_y += otherRoom.height;
476         }else if(room.isEastof(otherRoom)) {
477             start_y += room.height/2;
478             end_y += otherRoom.height/2;
479             end_x += otherRoom.width;
480         }else {
481             start_y += room.height/2;
482             end_y += otherRoom.height/2;
483             start_x += room.width;
484         }
485
486         //carve corridor between the two
487         cells[start_x][start_y] = Cells.AIR.makeInstance();
488         cells[end_x][end_y] = Cells.AIR.makeInstance();
489
490         while(start_x < end_x)
491             cells[++start_x][start_y] = Cells.AIR.makeInstance();
492         while(start_x > end_x)
493             cells[--start_x][start_y] = Cells.AIR.makeInstance();
494         while(start_y < end_y)
495             cells[start_x][++start_y] = Cells.AIR.makeInstance();
496         while(start_y > end_y)
497             cells[start_x][--start_y] = Cells.AIR.makeInstance();
498         room.connections++;
499         otherRoom.connections++;
500     }
501 }
502 }
503
504 //Place both sets of stairs in an empty cell in a room
505 private void makeStairs() {
506     Room roomWithStairs = rooms.get(Util.randomInt(0, rooms.size()-1));
507     do {
508         stairsUpX = Util.randomInt(roomWithStairs.start_x, roomWithStairs.start_x +
... roomWithStairs.width - 1);
509         stairsUpY = Util.randomInt(roomWithStairs.start_y, roomWithStairs.start_y +
... roomWithStairs.height - 1);
510         }while(cells[stairsUpX][stairsUpY].getRepresentation() != ' ');
511         cells[stairsUpX][stairsUpY] = Cells.STAIRS_UP.makeInstance();
```

```
512
513     roomWithStairs = rooms.get(Util.randomInt(0, rooms.size()-1));
514     do {
515         stairsDownX = Util.randomInt(roomWithStairs.start_x, roomWithStairs.start_x +
... roomWithStairs.width - 1);
516         stairsDownY = Util.randomInt(roomWithStairs.start_y, roomWithStairs.start_y +
... roomWithStairs.height - 1);
517         }while(cells[stairsDownX][stairsDownY].getRepresentation() != ' ');
518         cells[stairsDownX][stairsDownY] = Cells.STAIRS_DOWN.makeInstance();
519     }
520     private class Room{
521         int start_x;
522         int start_y;
523         int width;
524         int height;
525         int connections;
526
527         Room(int start_x, int start_y, int width, int height)
528     {
529         this.start_x = start_x;
530         this.start_y = start_y;
531         this.width = width;
532         this.height = height;
533     }
534
535         boolean intersects(List<Room> rooms)
536     {
537         for(Room otherRoom: rooms) {
538             if(this.isAbove(otherRoom))
539                 continue;
540             if(this.isBelow(otherRoom))
541                 continue;
542             if(this.isWestOf(otherRoom))
543                 continue;
544             if(this.isEastOf(otherRoom))
545                 continue;
546
547                 return true;
548             }
549             return false;
550         }
551
552         void makeRoom(Grid grid) {
553             for(int i=0;i<width;i++)
554                 for(int j=0;j<height;j++)
555                     cells[start_x+i][start_y+j] = Cells.AIR.makeInstance();
556         }
557
558         boolean isAbove(Room otherRoom) {
559             return (this.start_y + this.height) < otherRoom.start_y;
560         }
561     }
```

```
562     boolean isBelow(Room otherRoom) {
563         return otherRoom.isAbove(this);
564     }
565
566     boolean isWestOf(Room otherRoom) {
567         return (this.start_x + this.width) < otherRoom.start_x;
568     }
569
570     boolean isEastOf(Room otherRoom) {
571         return otherRoom.isWestOf(this);
572     }
573 }
574
575 //Where are the stairs up
576 public int getStartX() {
577     return stairsUpX;
578 }
579
580 public int getStartY() {
581     return stairsUpY;
582 }
583
584 //used to test generation
585 public static void main(String[] args) {
586     Dungeon dungeon = new Dungeon();
587     for(int y=0;y<dungeon.SIZE;y++) {
588         for(int x=0;x<dungeon.SIZE;x++)
589             System.out.print(dungeon.cells[x][y].getRepresentation());
590         System.out.println();
591     }
592 }
593
594 //Where are the stairs down
595 public int getEndX()
596 {
597     return stairsDownX;
598 }
599
600 public int getEndY()
601 {
602     return stairsDownY;
603 }
604
605
606 }
607
608
609 //cells.Grid.java
610 package cells;
611
612 import java.io.File;
613 import java.util.Scanner;
```

```
614
615 public class Grid
616 {
617     public int SIZE = 64;
618
619     public Cell[][] cells;
620
621     Grid() {
622         cells = new Cell[SIZE][SIZE];
623     }
624
625     Grid(int size) {
626         this.SIZE = size;
627         cells = new Cell[SIZE][SIZE];
628     }
629
630     public static void clearCells(Grid grid) {
631         for(int i=0;i<grid.SIZE;i++)
632             for(int j=0;j<grid.SIZE;j++)
633                 grid.cells[i][j] = Cells.AIR.makeInstance();
634     }
635
636     public static void fillCells(Grid grid, CellTemplate cellType)
637     {
638         for(int i=0;i<grid.SIZE;i++)
639             for(int j=0;j<grid.SIZE;j++)
640                 grid.cells[i][j] = cellType.makeInstance();
641     }
642
643     public static Grid getGridFromFile(String url)
644     {
645         return getGridFromFile(new File(url));
646     }
647
648     public static Grid getGridFromFile(File file) {
649         Grid grid = new Grid();
650         try {
651             Scanner in = new Scanner(file);
652             grid.SIZE = in.nextInt();
653             for(Cell[] row: grid.cells)
654                 for(Cell cell: row)
655                     cell = Cells.getTemplate(in.nextInt());
656         } catch(Exception e) {
657             System.err.println(e.getMessage());
658             return null;
659         }
660         return grid;
661     }
662 }
663
664
665 //entities.Entity.java
```

```
666 package entities;
667 import misc.Representable;
668 import misc.Representation;
669
670
671 public abstract class Entity implements Representable
672 {
673     protected Representation DEFAULT REPRESENTATION;
674     String name;
675     String description;
676
677     @Override
678     public char getRepresentation()
679     {
680         return DEFAULT REPRESENTATION.ASCII;
681     }
682
683     public char getDefaultRepresentation() {
684         return DEFAULT REPRESENTATION.ASCII;
685     }
686
687     public void setRepresentation(char ASCII)
688     {
689         this.DEFAULT REPRESENTATION.setASCII(ASCII);
690     }
691 }
692
693 //entities.Inventory.java
694 package entities;
695
696 import java.util.ArrayList;
697
698 public class Inventory extends ArrayList<ItemEntity>
699 {
700     public ItemEntity getFirst() {
701         return this.get(0);
702     }
703 }
704
705
706 //entities.ItemEntity.java
707 package entities;
708
709 import misc.Representation;
710
711 public class ItemEntity extends Entity
712 {
713     private Representation representation;
714
715     public char getRepresentation() {
716         return representation.ASCII;
717     }
}
```

```
718 }
719
720
721 //entities.LivingEntity.java
722 package entities;
723
724 public class LivingEntity extends Entity
725 {
726     Inventory inventory = new Inventory();
727     public int x, y;
728 }
729
730
731 //entities.Player.java
732 package entities;
733
734 import misc.Representation;
735
736
737 public class Player extends LivingEntity
738 {
739     public Player() {
740         DEFAULT REPRESENTATION = new Representation('@');
741     }
742 }
743
744
745 //misc.Openable.java
746 package misc;
747
748 public interface Openable
749 {
750     public boolean isOpen();
751     public String open();
752     public String close();
753 }
754
755
756 //misc.Representable.java
757 package misc;
758
759 public interface Representable
760 {
761     public char getRepresentation();
762     public char getDefaultRepresentation();
763 }
764
765 //misc.Representation.java
766 package misc;
767
768 public class Representation
769 {
```

```
770     public char ASCII;
771     protected boolean modifiable = true;
772
773     public Representation(char ASCII)
774     {
775         this.ASCII = ASCII;
776         // TODO Auto-generated constructor stub
777     }
778
779     public Representation(char ASCII, boolean isModifiable)
780     {
781         this.ASCII = ASCII;
782         this.modifiable = isModifiable;
783         // TODO Auto-generated constructor stub
784     }
785
786     public boolean isModifiable() {
787         return modifiable;
788     }
789
790     public boolean setASCII(char ASCII) {
791         if(modifiable)
792         {
793             this.ASCII = ASCII;
794             return true;
795         }
796
797         return false;
798     }
799 }
800
801 //misc.Util.java
802 package misc;
803
804 import java.util.Random;
805
806 public class Util
807 {
808
809     private static long lastSeed = System.currentTimeMillis();
810     private static Random random = new Random(lastSeed);
811     public static final int SOUTH = 2;
812     public static final int WEST = 4;
813     public static final int EAST = 6;
814     public static final int NORTH = 8;
815     public static final int SOUTH_WEST = 1;
816     public static final int SOUTH_EAST = 3;
817     public static final int NORTH_WEST = 7;
818     public static final int NORTH_EAST = 9;
819
820     public static int randomInt(int min, int max) {
821         lastSeed += System.currentTimeMillis();
```

```
822     random.setSeed(lastSeed);
823     random.nextInt();
824
825     int n = random.nextInt(max-min+1);
826     return n+min;
827 }
828
829     public static int getDX(int direction)
830 {
831     if(direction == WEST || direction == SOUTH_WEST || direction == NORTH_WEST)
832         return -1;
833     if(direction == EAST || direction == SOUTH_EAST || direction == NORTH_EAST)
834         return 1;
835     return 0;
836 }
837
838     public static int getDY(int direction)
839 {
840     if(direction == NORTH || direction == NORTH_WEST || direction == NORTH_EAST)
841         return -1;
842     if(direction == SOUTH || direction == SOUTH_WEST || direction == SOUTH_EAST)
843         return 1;
844     return 0;
845 }
846
847     public static boolean isValidLocation(int x, int y, int SIZE)
848 {
849     if(x < 0 || y < 0 || x >= SIZE || y >= SIZE)
850         return false;
851     return true;
852 }
853 }
854
855
856
```