Route

경로를 만들 때 먼저 고민한 부분은 "무작위로 만들어지는 방들을 어떻게 연결할 수 있을까?" 였습니다.

그래서 생각한 방법은 먼저 한 층의 방을 만든 후 만들어진 방을 바탕으로 다음 층의 방을 만들면 되겠다고 생각해서 room이라는 변수와 link라는 변수를 만들었습니다.

이 때 room의 자료형은 RoomKind라는 열거체로 방의 종류를 저장합니다.

link는 왼쪽 방을 가리키면 0, 가운데 방을 가리키면1, 오른쪽 방을 가리키면 2의 인덱스에 방의 위치 인덱스를 저장합니다.

```
1  package route;
2
3  import java.io.Serializable;
4
5  public class Route implements Serializable {
6      final int ROW = 15;
7      final int COL = 7;
8      private RoomKind[][] room = new RoomKind[ROW][COL];
9      private int[] roomCount = new int[ROW];
10     private String[][][] link = new String[ROW - 1][COL][3];        //방과 방 사이의 길 / 각 방마다 가지고 있으며, 최대 3개
11     final int ELITE_ROW = 7;         //중간 보스 8층에서 나옴
12     final int BOSS_ROW = 14;         //보스는 15층에서 나옴
13
```

아래 사진은 방을 만드는 메서드로 link에 다음 방의 인덱스를 저장합니다.

link는 방을 만들 때 필요한 변수는 아니지만, GUI 구성할 때 필요할 것 같아서 만들었습니다.

또한, link의 자료형이 String으로 되어있는데 원래는 '₩', '|', '/'로 저장하려 했으나, 중간 보스와 보스 몬스터의 방의 인덱스가 3으로 고정되어 이를 표현할 수가 없어서 숫자를 String으로 저장했습니다.

```
151     //아래 층을 바탕으로 방을 만듦
152     private void makeRoom(int row, int col) {
153         if(room[row - 1][col] != null) {
154             int direction = decideDirection(row - 1, col);      //direction은 -1, 0, 1 중 하나
155 
156             link[row - 1][col][direction + 1] = Integer.toString(col + direction);      //0에는 왼쪽, 1에는 가운데, 2에는 오른쪽 저장
157 
158             int probability = (int) (Math.random()*50);     //확률 / 0~29 몬스터, 30~38 상인, 39~44 보물, 45~49 휴식
159 
160             if(probability < 30) {
161                 room[row][col + direction] = RoomKind.ENEMY;
162             }
163             else if(probability < 39) {
164                 room[row][col + direction] = RoomKind.MERCHANT;
165             }
166             else if(probability < 45 ){
167                 room[row][col + direction] = RoomKind.TREASURE;
168             }
169             else {
170                 room[row][col + direction] = RoomKind.REST;
171             }
172         }
173     }
```

아래의 코드는 이전 방에서 다음 방으로 향하는 방향을 정하는 메서드입니다.

direction에는 -1, 0, 1의 값이 저장되며, 만약 위치가 유효하지 않다면 while문을 통해 계속 방향을 다시 정하게 됩니다.

잘려 있는 주석의 내용은 경로가 X자로 겹치지 않도록 한다는 내용입니다.

```
175     //방향 정하기
176     private int decideDirection(int row, int col) {
177         int direction = (int) (Math.random()*3) - 1;
178         while (!isValidLocation(col + direction)        //가리키는 곳이 -1이거나 7인 경우
179            || (col > 0 && Integer.toString(col).equals(link[row][col - 1][2]) && direction == -1)   //왼쪽 링크가 자신을 가리키고, direct
180            || (col < 6 && Integer.toString(col).equals(link[row][col + 1][0]) && direction == 1)        //오른쪽 링크가 자신을 가리키고, d
181               ) {
182
183             direction = (int) (Math.random()*3) - 1;
184         }
185
186         return direction;
187     }
```

아래 코드는 생성자에서 1층을 만드는 부분입니다.
9층을 만드는 코드 또한 이와 같습니다.

방은 최소 3개에서 최대 6개까지 만들어집니다.

```
21          //1층
22          for(int i = 0; i < COL; i++) {
23              if(countRowRoom(0) == 6) {       //방이 6개라면 for문 탈출
24                  break;
25              }
26
27              int existence = (int) (Math.random()*2);
28
29              if(existence == 1 && room[0][i] == null) {       //1일 때 방 생성, 첫 번째 방은 ENEMY
30                  room[0][i] = RoomKind.ENEMY;
31              }
32
33              if(i == 6 && countRowRoom(0) < 3) {      //for문을 다 돌았는데 방의 개수가 3보다 작으면 i를 0으로 바꿈
34                  i = 0;
35              }
36          }
```

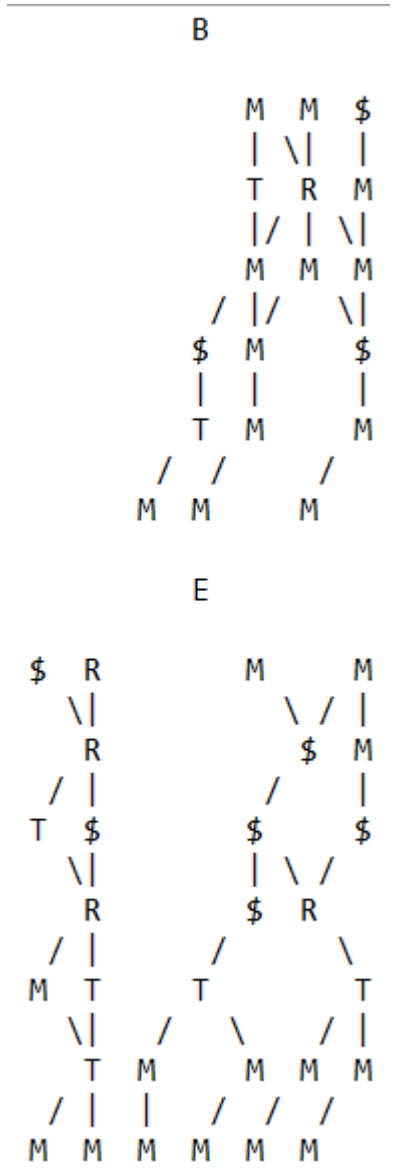
오른쪽의 사진은 2~14층을 만드는 코드입니다.

기본적으로 1층을 만드는 방과 거의 유사하지만, 추가적으로 이전 층의 방이 많을수록 생성 확률이 줄어듭니다.

이는 방이 점점 많아질 수 밖에 없는 점을 고치기 위해서 넣었습니다.

```
59          //2층 ~ 14층 / 8층, 9층은 제외 8층은 중간 보스방, 9층은 1층과 같은 방법으로 생성
60          for(int i = 1; i < BOSS_ROW; i++) {
61              if(i == ELITE_ROW || i == ELITE_ROW + 1) {
62                  continue;
63              }
64
65              for(int j = 0; j < COL; j++) {
66                  makeRoom(i, j);
67              }
68
69              for(int j = 0; j < COL; j++) {
70                  if(room[i - 1][j] != null) {
71                      if(countRowRoom(i) == 6) {      //방이 6개라면 for문 탈출
72                          break;
73                      }
74
75                      int existence = (int) (Math.random()*countRowRoom(i - 1));
76
77                      if(existence == 0) {        //0일 때 방 생성, 16% ~ 33%, 이전 층에 방이 적으면 확률 높음
78                          makeRoom(i, j);
79                      }
80                  }
81              }
82
83              roomCount[i] = countRowRoom(i);
84
85              if(roomCount[i] < 3) {
86                  i--;
87              }
88          }
```

콘솔에서 실행 시 모습입니다.



Card

카드 중 적에게 쓸 수 없는 카드가 존재하기 때문에 canUseEnemy라는 변수를 만들었고,

적 전체에게 사용할 수 있는 카드가 존재하기 때문에 canUseAll이라는 변수를 만들었습니다.

카드의 종류는 공격 카드와 스킬 카드가 있습니다.

아래쪽의 File부분은 카드 사용 시 효과음을 출력하기 위해서 만든 변수들입니다.

```
14 public class Card {
15     protected int cost;
16     protected int damage;
17     protected int additionDamage;
18     protected int additionShield;
19 
20     //additionWeak과 additionWeakening은 지속되는 시간을 저장
21     protected int additionWeak;          //취약
22     protected int additionWeakening;     //약화
23     protected boolean canUseEnemy;       //damage가 있는 경우에만 true
24     protected boolean canUseAll;         //모든 적을 공격하는 지 확인
25     protected boolean hasSkill;          //특수 능력을 가지고 있는 지 확인
26 
27     //ATTACK or SKILL
28     protected CardKind kind;
29 
30     private File attackSound;
31     private AudioInputStream stream;
32     private AudioFormat format;
33     private DataLine.Info info;
34     private Clip clip;
```

카드 사용 시 적에게 쓸 수 있는 카드는 attack메서드가 호출이 됩니다.

skill은 몇몇 카드들만 가지고 있으며, 기본적으로는 아무런 능력도 없기 때문에 Card클래스에서는 오버라이딩용으로 내용이 비어 있습니다.

예를 들어 Anger 카드는 해당 카드를 복사해서 덱에 추가하는 역할을 하기 때문에 skill을 오버라이딩하였습니다.

```
50      public void attack(Enemy monster, Protagonist player) {
51          monster.hit(player, this);
52          playSound("/sounds/PlayerAttack.wav", 10);
53      }
54
55      //오버라이딩용
56      public void skill(Protagonist player) {}
```

```
10  public class Anger extends Card {
11      public Anger() {
12          cost = 0;
13          damage = 6;
14          canUseEnemy = true;
15          hasSkill = true;
16          kind = CardKind.ATTACK;
17      }
18
19      @Override
20      public void skill(Protagonist player) {
21          player.getTrashCan().addCard(new Anger());
22          player.getTmpCardBag().add(new Anger());
23          player.setTmpCardCount(player.getTmpCardCount() + 1);
24      }
25  }
```

CardBag, HandCard, CardTrashCan

플레이어가 전투 중 카드는 총 3곳을 이동합니다.

처음에는 가방에서 나와서 손으로 이동하게 되고, 손에 있던 카드는 쓰레기통으로 이동하게 되고, 쓰레기통에 있던 카드는 가방으로 이동하게 됩니다. 그래서 이를 3가지 클래스로 나눠서 구현하였습니다.

이 3개의 클래스는 카드 추가, 카드 삭제 메서드와 카드 리스트, 카드의 개수 변수가 있습니다.

카드 리스트의 경우 카드가 자주 삽입되고, 삭제되기 때문에 연결리스트를 이용하여 구현하였습니다.

아래의 코드는 Protagonist클래스의 메서드들입니다.

이것을 구현하기 위해서 처음에는 CardBag, HandCard, CardTrashCan에서 작성하려고 했으나, 각각의 인스턴스에 접근하는 데 어려움을 느꼈습니다.

그래서 고민한 결과 카드를 옮기는 것은 원래 사람이 하는 것이기 때문에 Protagonist클래스로 옮겼더니 굉장히 쉽게 코드를 작성할 수 있었습니다.

```
216     //카드를 가방에서 손으로 보냄 - 턴 시작 시 5번 호출 -> 유물 있으면 더 많이 호출
217     public void goHand() {
218         int i = (int) ((Math.random() * cardBag.getBagCount()));
219         hand.addCard(cardBag.deleteCard(i));
220     }
221
222     //카드를 손에서 쓰레기통으로 보냄 - 턴 종료 시 남은 카드 개수만큼 호출
223     public void goTrashCan(int pos) {
224         trashCan.addCard(hand.deleteCard(pos));
225     }
226
227     //카드를 쓰레기통에서 가방으로 보냄
228     public void goBag() {
229         for(int i = trashCan.getCount(); i > 0; i--) {
230             cardBag.addCard(trashCan.deleteCard());
231         }
232     }
```

Protagonist

플레이어는 데미지 버프와 디버프가 있습니다.
버프의 경우는 전투가 종료될 때까지 유지되고, 디버프의 경우는 지속시간이 종료되면 사라지게 됩니다.

또한, CardBag, CardTrashCan, HandCard 클래스의 인스턴스들을 가지고 있습니다.
이 때 tmpCardBag이라는 인스턴스도 있는데, 이 인스턴스의 경우에는 임시카드들을 가지고 있습니다.

예를 들어서 분노 카드는 사용 시 해당 카드를 복사하여 쓰레기통에 추가하는데 이 카드는 전투가 종료된 후에 사라져야 되기 때문에 임시카드 가방에 넣습니다.

임시카드 가방에 있는 카드들은 전투 종료 후에 제거됩니다.

```
16 //플레이어
17 public class Protagonist extends Fighter implements Serializable {
18     private int buffDamage;      //데미지 증가
19     private LinkedList<Integer> deBuffDamage;    //데미지 감소
20     private LinkedList<Integer> deBuffDamageDuration;
21     private int deBuffDamageCount;
22
23     private int shield;      //shield는 지속시간 및 적용 개수 필요없음
24     private int energy;
25
26     private CardBag cardBag;
27     private CardTrashCan trashCan;
28     private HandCard hand;
29     private RelicsBag relicsBag;
30
31     //임시 카드들을 보관하는 가방 -> 전투 종료 후 제거
32     //ex) Anger(분노) 카드는 복사해서 쓰레기통에 넣어 놓음
33     private LinkedList<Card> tmpCardBag;
34     private int tmpCardCount;
35
36     private int gold;
```

카드를 사용하면 에너지가 충분한 지 확인하고, 공격과 각종 버프들을 획득한 후 카드를 쓰레기통으로 보냅니다.

적에게 사용할 수 없는 경우에는

public void useCard(int pos)로 오버로딩 되어있으며,

attack을 호출하지 않습니다.

PPT작성하면서 생각한 것인데 적에게 사용할 수 없는 경우를 오버로딩하는 것보단 useCard에서 적에게 사용할 수 있는 지 판별하는 것이 더 나을 것 같습니다.

```
75      //카드 사용 - 적에게 사용할 수 있는 경우
76      public void useCard(int pos, Enemy target) {
77          if(hand.getCard(pos).getCost() > energy) {
78              System.out.println("Energy shortage");
79              return;
80          }
81          else {
82              hand.getCard(pos).attack(target, this);
83              energy -= hand.getCard(pos).getCost();
84          }
85
86          //카드에 버프가 존재하는 경우
87          if(hand.getCard(pos).getAdditionDamage() != 0) {
88              buffDamage += hand.getCard(pos).getAdditionDamage();
89          }
90          if(hand.getCard(pos).getAdditionShield() != 0) {
91              shield += hand.getCard(pos).getAdditionShield();
92          }
93
94          //카드가 스킬을 가지고 있는 경우
95          if(hand.getCard(pos).isHasSkill()) {
96              hand.getCard(pos).skill(this);
97          }
98
99          goTrashCan(pos);
00      }
```

Enemy

attack은 카드의 skill처럼 본문이 비워져 있으며, 각 몬스터마다 오버라이딩 되어 있습니다.

hit은 몬스터에게 취약이 있는 경우와 없는 경우로 나눠져 있으며 디버프가 존재하는 경우 디버프를 갖게 됩니다.

```
31      //공격 당함
32      public void hit(Protagonist player, Card card) {
33          int damage = card.getDamage() + player.getBuffDamage() - player.getDeBuffDamage();
34          //취약이 없는 경우
35          if(weakDuration <= 0) {
36              if(hp - damage <= 0) {  //몬스터 체력이 0보다 작거나 같은 경우 체력을 0으로 만듬
37                  hp = 0;
38              }
39              else {
40                  hp -= damage;
41              }
42          }
43          //취약이 있는 경우
44          else {
45              if(hp - damage * 3 / 2 <= 0) {
46                  hp = 0;
47              }
48              else {
49                  hp -= damage * 3 / 2;
50              }
51          }
52
53          //카드에 디버프가 존재하는 경우
54          if(card.getAdditionWeak() != 0) {
55              weakDuration += card.getAdditionWeak();
56          }
57          if(card.getAdditionWeakening() != 0) {
58              weakeningDuration += card.getAdditionWeakening();
59          }
60      }
```

attack을 오버라이딩한 이유는 몬스터마다 공격의 개수가 다르기 때문입니다.

```java
    @Override
    public void attack(Protagonist player) {
        //휘두르기 모션

        playSound("/sounds/DevilSlimeAttack.wav", 300);

        int randomDeBuff = (int) (Math.random()*5);

        if(randomDeBuff == 0) {     //쓰레기 카드 추가
            player.getTrashCan().addCard(new Trash());
            player.getTmpCardBag().add(new Trash());
            player.setTmpCardCount(player.getTmpCardCount() + 1);
        }
        else if(randomDeBuff == 1) {        //공격력 디버프 -> 2턴 간 3만큼
            player.setDeBuffDamage(3);
            player.setDeBuffDamageCount(player.getDeBuffDamageCount() + 1);
            player.setDeBuffDamageDuration(2);
        }

        player.hit(this);
    }
}
```

```java
    @Override
    public void attack(Protagonist player) {
        int attackKind = (int) (Math.random()*4);

        if(attackKind < 3) {        //75%확률로 attack1
            attack1(player);
            System.out.println("FlyingEye Attack!");
        }
        else {
            attack2(player);
        }
    }

    //깨물기 or 몸통박치기 공격 -> 특수능력X
    private void attack1(Protagonist player) {
        int motionKind = (int) (Math.random()*2);

        if(motionKind == 0) {       //깨물기 모션

            playSound("/sounds/FlyingEyeAttack1-1.wav", 200);
        }
        else {      //몸통박치기 모션

            playSound("/sounds/FlyingEyeAttack1-2.wav", 200);
        }

        player.hit(this);
    }
```