Praktische Informatik I
Tutor:in: Leander Staack

WiSe 2021/22
Bearbeiter:in: Julius Walczynski [6113829]

Übungsblatt 10

Lösungsvorschlag

Aufgabe 1 Lässig Level laden

```
26      public Level(String filename) {
27          try {
28              FileReader fr = new FileReader(filename);
29              BufferedReader br = new BufferedReader(fr);
30
31              List<String> map = new ArrayList<String>();
32
33              String line = br.readLine();
34              while(line != null){
35                  map.add(line);
36                  line = br.readLine();
37              }
```

Die Feldbeschreibung wird nicht länger direkt hardcoded als *String[ ]* übergeben, sondern in einer Datei gespeichert. Der Name dieser Datei wird als Parameter durch den Konstruktor der Level Klasse übergeben. Diese wird mit Hilfe eines *FileReader*s und *BufferedReader*s Zeile für Zeile ausgelesen und in einer *List<String>*, der Variable *map*, gespeichert.

```
64          } catch(FileNotFoundException e){
65              throw new IllegalArgumentException("Invalid filename.");
66          } catch(IOException e){
67              throw new IllegalArgumentException("something wen wrong while reading file.")
                  ;
68          }
```

Sollten Probleme beim lesen der Datei auftreten werden diese in zwei *catch*-clauses aufgefangen und in verständliche Fehlernachrichten für den User konvertiert.

```
39              String[] tmp = new String[map.size()];
40              map.toArray(tmp);
41
42              for(int i = 0; i < tmp.length; i++) tmp[i] = tmp[i].replaceAll("[\\x21-\\x2c
                  \\x2e-\\x7b\\x7d-\\x7e]", "O");
43              this.field = new Field(tmp);
```

Die Elemente der map (*List<String>*) werden in einen *String[ ]* kopiert. Alle ASCII-chars zwischen 0x21 - 0x7e außer 0x2d ('-') und 0x7c ('|') werden in diesen *String*s durch 'O' ersetzt. Der auf die *Field*-Klasse angepasste *String[ ]* wird benutzt um das richtige Layout als Field zu erstellen. Dieses wird intern im privaten Attribut *field* gespeichert.

```
45              for(int y = 0; y < map.size(); y+=2){
46                  for(int x = 0; x < map.get(y).length(); x+=2){
47                      char info = map.get(y).charAt(x);
48                      if(info == 'O'|| info == ' ') continue;
49
50                      int rotation = info&3;
51                      int actorID = (info>>2)-9;
```

Es wird über jeden Knotenpunkt/char *info* mit den Koordinaten $(2n_1, 2n_2); n_1, n_2 \in \mathbb{N}$ in der map *List<String>* (vorstellbar als *char[ ][ ]*) geloopt. Sollte der entsprechende char *info* kein normaler Knotenpunkt sein ('O', ' '), wird aus ihm die angegebene Rotation und actorID errechnet. Hierfür wird der *char* als *byte* (8-Bit) betrachtet und in 6-Bit + 2-Bit aufgebrochen. Die hinteren 2-Bits (least significant) repräsentieren hierbei die Rotation (0-3), welche vollständig in 2-Bit abgebildet werden kann. Die vorderen 6-Bit bleiben als *actorID* übrig. Um die *rotation*, also nur die hinteren 2-Bits betrachten zu können, werden die vorderen 6-Bits durch eine Bit-Maske (0b00000011 = 3) auf 0 gebracht $\Rightarrow$ *rotation* = *info*&*3*.
Um die vorderen 6-Bits als *actorID* betrachten zu können, müssen diese um die 2-Bit *rotation* nach rechts geschoben/geshiftet werden. Übrig bleibt eine valide *actorID*. Da einfach zuschreibende/nutzbare ASCII-chars jedoch erst bei 0b00100001=0x21='!' starten, wir allerdings um alle möglichen Rotation abspeichern zu können die letzten 2-Bits auf jedenfall auch auf 00 setzen können müssen, beginnen wir mit dem nächst größeren char: 0b00100100=0x24='$'. Um angenehmer mit der gewonnenen und geshifteten *actorID* rechnen zu können, wird ihr Startpunkt auf 0 gesetzt: $actorID_0$=0b00001001-0b00001001=0b00001001-9=0 $\Rightarrow$ 9 muss für einen Start bei $actorID_0 = 0$ subtrahiert werden.

```
53                  if(actorID<0||actorID>NPCNames.length) throw new
                      IllegalArgumentException("invalid character[s] in map file");
```

Sollte die so gewonnene *actorID* weder auf den *Player* noch auf einen einen der spezifizierten *NPC*s verweisen, wird die *actorID* als ungültig angenommen und das Programm durch eine *IllegalArgumentException* beendet. Hierbei gilt:

actorID	actor
0	Player
1	NPC0
2	NPC1
⋮	⋮

```
55                      if(actorID == 0){
56                          if(actors.size()>0 && (actors.get(0) instanceof Player)) throw
                              new IllegalArgumentException("more than one symbolized player
                              in map");
57                          actors.add(0, new Player(x/2, y/2, rotation, this.field));
58                      }else{
59                          actors.add(new NPC(x/2, y/2, rotation, NPCNames[actorID-1], this.
                              field));
60                      }
```

Sollte die *actorID* mit der des *Player*s übereinstimmen, wird dieser als erstes Element (index = 0) in die Liste *actors* hinzugefügt. Sollte jedoch bereits ein Spieler kreiert worden sein, wird eine Fehlernachricht an den User gegeben und das Programm durch eine IllegalArgumentException frühzeitig beendet.
Für alle anderen *actorID*s wird ein entsprechender *NPC* instantiiert und zu *actors* hinzugefügt.

```
63              if(actors.size()<=0||!(actors.get(0) instanceof Player)) throw new
                  IllegalArgumentException("no player on map");
```

Sollte nach dem auslesen der Karte kein Spieler gefunden wurden sein, wird eine Fehlernachricht an den User gegeben und das Programm durch eine IllegalArgumentException beendet.

```
74      public List<Actor> getActors(){
75          return this.actors;
76      }
```

Die *getActors*-Methode gibt das als privat abgespeicherte Attribut *field* (Das Spielfeld des Levels, welches im Konstruktor erstellt wurde) zurück.

```
23          level = new Level("Map.txt");
24
25          Player player = level.getPlayer();
26          while(true){
27              for(Actor actor: level.getActors()){
28                  if(actor instanceof NPC) ((NPC)actor).addPlayerStep(player.getRotation())
                      ;
29                  actor.act();
```

Die Hauptspielklasse muss nun nur noch eine Referenz zu einem *Level* beinhalten. Über die *getActors*-Methode bleibt die Logik zum loopen über die *Actor*s zum ausführen ihrer *act*-Methode weitestgehend bestehen.

Aufgabe 2 Bonusaufgabe: Ich bin dann mal weg

```
90      public void hide(){
91          this.field.hide();
92          for(Actor actor: this.actors){
93              actor.setVisible(false);
94          }
95      }
```

Die *hide*-Methode ruft die *hide*-Methode des *Field*s auf. Nach dem dieses alle Tiles des Spielfeldes unsichtbar gemacht hat, werden alle *Actor*s in *actors* in einer *foreach*-Schleife mit Hilfe der *setVisible*-Method der geerbten GameObject-Klasse unsichtbar gemacht.

```
93      public void hide(){
94          for(GameObject tile: this.tiles){
95              tile.setVisible(false);
96          }
97      }
```

Das selbe verhalten trifft auch auf die *hide*-Methode der *Field*-Klasse zu.