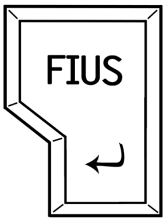
Universität Stuttgart



Übungsblatt 2

Java Vorkurs(WS 2022)

Aufgabe 1: The Player Enters the Game

a) Instanziiere die Simulation wie bekannt und mache dich mit dieser vertraut. Benutze dabei den Task aus der Klasse `Sheet2Task1` und den Verifier aus `Sheet2Task1Verifier`.

b) Lade Neo in die Matrix. Dafür musst du zuerst ein Objekt der Klasse `Neo` erstellen und in einer Variable (z.B. `player`) speichern. Dann kannst du den `player`, wie die Münze in Aufgabenblatt 1 Aufgabe 5, mit dem `PlayfieldModifier` auf dem Spielfeld platzieren.

Dein Code sollte in etwa wie folgt aussehen.

```
14 public void run(Simulation sim) {
15     // Your Implementation here
16
17     PlayfieldModifier pm = new PlayfieldModifier(sim.
           ↪ getPlayfield());
18     Neo player = new Neo();
19     pm.placeEntityAt(player, new Position(1,1));
```

Überprüfe im Task Status Tab ob dein Code korrekt funktioniert.

c) Damit sich Neo bewegt, musst du ihm Kommandos geben. Du kannst Neo die folgenden Kommandos geben.

```
20     player.move();
21     player.moveIfPossible();
22     player.turnClockWise();
```

Teste alle 3 Kommandos. Damit Neo sich bewegen kann, muss die Simulation mit dem Play Button gestartet werden!

d) Neo kann sich nur nach rechts drehen. Wie kannst du Neo trotzdem dazu bringen, sich nach rechts (oder nach hinten) umzudrehen?

Hinweis: Du kannst Kommandos mehrfach hintereinander aufrufen.

e) Gib Neo die folgenden Anweisungen:

i. Bewege dich 2 Schritte geradeaus
ii. Dann drehe dich nach rechts
iii. Gehe einen Schritt vor
iv. Drehe dich nach links
v. Gehe vier Schritte geradeaus
vi. Drehe dich nach rechts
vii. Als letztes gehe einen Schritt geradeaus

Überprüfe im Task Status Tab ob du Neo die richtigen Kommandos gegeben hast.

Aufgabe 2: Exceptions - Mit dem Kopf durch die Wand

Tank und Neo wollen nun in einer neuen Simulationsumgebung trainieren. Dabei soll Neo in einem potenziell gefährlichen Szenario so schnell wie möglich zur nächsten Telefonstation laufen.

a) Instanziiere die Simulation wie bekannt und mache dich mit dieser vertraut. Benutze dabei den Task aus der Klasse `Sheet2Task2` und den Verifier aus `Sheet2Task2Verifier`.

b) Als Operator hat Tank die Möglichkeit Hindernisse zwischen Neo und der Telefonstation einzufügen. Betrachte die Klasse `Sheet2Task2` mit dem Kommando `run` an der markierten Stelle. Überlege dir wie man an Tank's Stelle ein solches Hindernis an einer geeigneten Stelle platzieren kann. Hier kann eine beliebige Position $(x, 0)$ für $1 \leq x \leq 9$ ausgewählt werden. Was passiert, wenn man nun die Simulation wie zuvor ausführt?

Aufbau einer Exception

Wir wollen nun den Aufbau einer Exception genauer verstehen. Dafür betrachten wir zunächst den folgenden Programmausschnitt:

```
1  public class Main{
2      //Programmeinstieg
3      public void main(){
4          //do stuff
5          divide(1, 0);
6      }
7
8      public Integer divide(Integer a, Integer b){
9          return a / b;
10     }
11 }
```

Dabei lässt sich erkennen, dass an irgendeinem Punkt der Programmausführung durch 0 geteilt wird. Dieses ungewollte Verhalten wird bei der Programmausführung durch die folgende Exception ersichtlich:

```
java.lang.ArithmeticException: / by zero
    at de.unistuttgart.informatik.fius.jvk.Main.divide(Main.
       ↪ java:9)
    at de.unistuttgart.informatik.fius.jvk.Main.main(Main.java
       ↪ :5)
```

Im Folgenden wollen wir kurz über ein paar für uns interessante Inhalte der Exception reden:

i) Art der Exception: Diese kann in der ersten Zeile abgelesen werden. In unserem Beispiel handelt es sich hier um eine `ArithmeticException`.

ii) Nachricht der Exception: Diese befindet sich hinter der Exceptionart und soll dem Programmierer ggfs. mehr Informationen über den Fehler selbst geben. In unserem Fall ist die Nachricht `"/ by zero"`.

iii) Fehlerstelle im Programm: Der Stacktrace der Exception beinhaltet weiterhin Informationen, wo im Programm die Exception geflogen ist. Hier interessiert uns fürs Erste nur die zweite Zeile. In unserem Beispiel deutet die zweite Zeile darauf hin, dass die Exception in der `Main` Klasse in der `divide` Operation in Zeile 9 geflogen ist.

c) Im Folgenden betrachten wir beispielhaft den Stacktrace einer Exception:

```
1 de.unistuttgart.informatik.fius.icge.simulation.exception.
     ↪ IllegalMoveException: Solid Entity in the way
```

```
2 at de.unistuttgart.informatik.fius.icge.simulation.entity.
    ↪ MovableEntity.internalMove(MovableEntity.java:linenumber1)
3 at de.unistuttgart.informatik.fius.icge.simulation.entity.
    ↪ MovableEntity.move(MovableEntity.java:linenumber2)
4 at de.unistuttgart.informatik.fius.jvk.tasks.Sheet2Task2.run(
    ↪ Sheet2Task2.java:linenumber3)
5 at de.unistuttgart.informatik.fius.icge.simulation.internal.
    ↪ tasks.StandardTaskRunner.executeTask(StandardTaskRunner.
    ↪ java:linenumber4)
6 at java.base/java.util.concurrent.CompletableFuture$AsyncSupply
    ↪ .run(CompletableFuture.java:linenumber5)
7 at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
    ↪ ThreadPoolExecutor.java:linenumber6)
8 at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run
    ↪ (ThreadPoolExecutor.java:linenumber7)
9 at java.base/java.lang.Thread.run(Thread.java:linenumber8)
```

Dieser soll nun genauer analysiert werden:

i) Welche Exception wurde geworfen und was ist die Nachricht der Exception?

ii) In welcher Operation wurde die Exception geworfen?

iii) In dem obigen Stacktrace sind die dazugehörigen Zeilennummern verloren gegangen. Ersetze diese!

d) Jetzt wollen wir die Exception verhindern. Dazu benutzen wir die Technik des Auskommentierens die ihr in Blatt 1 gelernt habt. Kommentiert in `Sheet2Task2` so lang `.move()` Kommandos aus, bis die Exception nicht mehr auftritt.

Spielt die Reihenfolge in der ihr die Kommandos auskommentiert eine Rolle für das Ergebnis?

e) Kommentiere erstmal wieder alle Zeilen ein. Kannst du Trotzdem verhindern, dass die Exception auftritt?

Hinweis: In Aufgabe 1 hast du noch ein weiteres Kommando kennengelernt mit dem du Neo bewegen kannst. Du kannst Neo auch um die Wand herum gehen lassen.

f) **(Optional)** Welche „linenumber“ aus dem obigen Stacktrace ändert sich, wenn man zwischen den `.move()` Aufrufen jeweils Leerzeilen einfügt?

g) **(Optional)** Versuche möglichst viele unterschiedliche Möglichkeiten zu finden, um die Exception zu verhindern ohne eine der Zeilen vor dem Kommentar zu verändern. Werde Kreativ und versuche auch drastische Lösungen wie eine Wand zu löschen. Du kannst für diese Aufgabe auch Code von anderen Klassen vorübergehend ändern, solltest diese Änderungen aber hinterher wieder rückgängig machen. Wie viele unterschiedliche Möglichkeiten kannst du finden?

Aufgabe 3: What If there is a Wall?!

In dieser Aufgabe wollen wir uns Conditions näher anschauen.

a) Instanziiere die Simulation wie bekannt und mache dich mit dieser vertraut. Was passiert bei der Ausführung?

IF-Condition

Nun betrachten wir ein beispielhaftes IF-Statement:

```
if (player.canMove()) {
    player.move();
} else {
    System.out.println("Morpheus: There is a difference between
        ↪ knowing the path and walking the path")
}
```

Die Abfrage `canMove()` gibt einen Boolean Wert zurück (also `true` oder `false`). Falls sich also vor dem Spieler kein Hindernis befindet gibt die Abfrage `canMove()` `true` zurück, da es wahr ist, dass der Spieler sich frei nach vorne bewegen kann. In diesem Fall ist die Bedingung des IF-Statements erfüllt. Dadurch wird der erste Block ausgeführt, was hier dem Spieler erlaubt sich vorwärts zu bewegen.
In dem anderen Fall, dass sich doch ein Hindernis vor dem Spieler befindet, ist die Bedingung des IF-Statements nicht erfüllt und der zweite Block wird ausgeführt. In diesem Fall bekommen wir hier eine erleuchtende Weisheit von Morpheus.
Die Kommandos in den zwei Code-Blöcken können nach bedarf ausgetauscht werden. In einem Code-Block dürfen sogar beliebig viele Kommandos, Abfragen oder wieder ganze IF-Statements stehen. Auch die genutzte Abfrage kann geändert werde, was in späteren Aufgaben noch erklärt wird.

b) Betrachte die Operation `movement` der Klasse `Sheet2Task3`. Füge das obige IF-Statement an einer geeigneten Stelle ein und führe die Simulation erneut aus. Die vorhandenen `.move()` Kommandos nach dem IF-Statement sollten in den ersten Code-Block vom IF verschoben werden. Der `else` Code-Block sollte erstmal leer bleiben.

c) Starte das Fenster neu. Lösche die Wand in dem Spielfenster. Dazu musst du oben rechts das rote Minus auswählen und dann auf die Wand klicken. Starte dann die Simulation. Neo sollte sich jetzt anders verhalten, obwohl du den Code nicht geändert hast!

d) Jetzt soll Neo die Telefonstation auch erreichen wenn die Wand nicht gelöscht wurde. Behandle also den `else` Fall so, dass Neo in beiden Fällen die Telefonstation erreicht. Gib Neo dafür im `else` Code-Block die notwendigen Kommandos.

Überprüfe deinen Code indem du ihn einmal ausführst ohne die Wand zu löschen und einmal wenn du die Wand davor gelöscht hast.

Aufgabe 4: I have the largest pockets

In dieser Aufgabe wollen wir eine neue Funktion unserer Spieler einführen. Wer die Doku aufmerksam gelesen hat kennt sie vielleicht schon: Unsere Spieler können Goldmünzen aufheben und niederlegen. Dafür verwenden wir `collectCoin()` und `dropCoin()`.

a) Wie immer muss der Task und Verifier gesetzt werden (`Sheet2Task4` und `Sheet2Task4Verifier` ↪)

b) Auf dem Spielfeld siehst du viele Goldmünzen. Wenn dein Spieler auf einem Feld mit Münzen steht kann er mit dem Kommando `collectCoin()` eine Münze aufheben. Räume damit das ganze Feld auf.

c) Mit dem Kommando `dropCoin()` kannst du eine Münze wieder auf dem Spielfeld ablegen. Nun verwende das `dropCoin()` Kommando, um alle Münzen die dein Spieler aufgesammelt hat auf das Feld (3,3) zu legen.

d) Die Kommandos `collectCoin()` und `dropCoin()` funktionieren nicht immer. Unter welchen Bedingungen kannst du die Kommandos korrekt benutzen und wann funktionieren sie nicht? Überprüfe deine Vermutung auch auf dem Spielfeld. Teste zum Beispiel ob du eine Münze ablegen kannst bevor du eine Münze aufgehoben hast. Oder teste ob du Münzen aufheben kannst, die vor dir liegen.

Vor- und Nachbedingungen

Damit ein Kommando richtig funktioniert, müssen oft bestimme Bedingungen gelten, bevor das Kommando aufgerufen wird. Bei dem `move()` Kommando darf sich vor dem Spieler keine Wand befinden. Da diese Bedingungen gelten müssen bevor man die Operation aufruft, nennt man sie Vorbedingungen. Meistens findet man diese Vorbedingungen im Javadoc Kommentar der Operation.
Mit Nachbedingungen kann man beschreiben was nach dem Ausführen der Operation gilt, wenn die Vorbedingungen vor dem Ausführen schon gegolten haben. Nach dem Ausführen von `move()` hat sich der Spieler eine Position nach vorne (in seine Blickrichtung) bewegt. Das gilt natürlich nur, wenn zum Zeitpunkt des `move()` Aufrufs die Vorbedingung, dass sich vor ihm keine Wand befindet, gegolten hat.

Aufgabe 5: While i do stuff on repeat

Ziel dieser Aufgabe ist es, die Syntax und die Funktion einer bestimmten While-Schleife genauer zu verstehen.

a) Instanziiere die Simulation wie bekannt und mache dich mit dieser vertraut.

While Schleife

Wir betrachten eine beispielhafte While Schleife.

```
while(player.canMove()) {
    player.move();
}
```

Ähnlich wie bei dem IF-Statement hat auch die While Schleife nach den runden Klammern einen Code-Block. Dieser wird hier Schleifeninneres genannt. Oft verwendet man auch Schleifenrumpf oder im englischen auch body.
While Schleifen prüfen jeweils vor Ausführung des Schleifeninneren, ob die gegebene Bedingung noch erfüllt ist. In unserem Beispiel wird also zunächst geprüft ob `player.canMove()` `true` ist und der Spieler kein Hindernis vor sich hat. Solange dies erfüllt ist, wird der Code-Block der Schleife ausgeführt und damit auch das Kommando `player.move()`. Jedes mal wenn der Code-Block ausgeführt wurde, wird die Bedingung in den runden Klammern erneut geprüft bevor der Code-Block wieder ausgeführt wird. Wenn die Bedingung von Anfang an `false` ist, dann wird der Code-Block niemals ausgeführt.
Andere für uns relevantere Operationen wären das Aufheben und Ablegen von Münzen. Dies kann mittels einer While Schleife wie folgt realisiert werden:

```
// Aufheben
while(player.canCollectCoin()){
    player.collectCoin();
}

// Ablegen
while(player.canDropCoin()){
    player.dropCoin();
}
```

Dabei prüfen wir mit der Schleifenbedingung jeweils, ob es noch weitere Münzen gibt, die wir aufheben (oder ablegen) können, bevor wir sie aufheben (oder ablegen).

b) Öffne die Klasse *Sheet2Task5*. Hebe alle Münzen auf. Dafür kannst du eine der Schleifen von oben verwenden.

Hinweis: Bei dieser Aufgabe ist das Spielfeld und die Anzahl der Münzen jedes mal anders.

c) Platziere jetzt alle Münzen auf dem rechten mittleren Spielfeld an der Wand. Hier kannst du wieder eine der Schleifen aus den Beispielen oben verwenden.

d) Zeichne mit den aufgehobenen Münzen eine Linie auf der mittleren Spur. Auf jedem Feld in der mittleren Reihe sollte jetzt eine Münze liegen.

Hinweis: Hier kannst du nicht direkt eine Schleife von den angegebenen Beispielen kopieren. Du musst noch den Code-Block der Schleife anpassen, damit mehr Kommandos bei jedem Schleifendurchlauf ausgeführt werden.

e) Platziere nun auf jedem Feld statt einer Münze drei Münzen.

Hinweis: Hier sollte es ausreichen den Code-Block der Schleife aus der Teilaufgabe davor anzupassen.

f) Platziere auf jedem zweiten Feld eine Münze. Das erste Feld solltest du leer lassen.

g) **(Optional)** Lege auf jedem von Neo aus erreichbaren Feld genau eine Münze ab. Beachte hierbei, dass die Wände auf der oberen und unteren Spur zufällig platziert werden.

Hinweis: Hier musst du neben einer While Schleife auch IF-Statements verwenden. Schau dir dafür nochmal Aufgabe 3 an.

Aufgabe 6: Wenn ich mich doch nur nach links drehen könnte

a) Instanziiere den Task und den Verifier (`Sheet2Task6` und `Sheet2Task6Verifier`).

b) Lasse Neo im Uhrzeigersinn an der Wand entlang laufen, bis er wieder an seinem Startpunkt angekommen ist. Implementiere das Verhalten an der markierten Stelle in der Klasse `Sheet2Task6`.

c) Lasse nun Neo gegen den Uhrzeigersinn an der Wand entlang laufen, bis er wieder an seinem Startpunkt angekommen ist. Welche der Varianten ist die schnellere?

Hinweis: Du kannst dir die „Uhrzeit“ der Simulation mit dem folgenden Programmcode auf der Konsole ausgeben lassen.

```
1     Long time = sim.getSimulationClock().getLastTickNumber();
2     System.out.println(time);
```

Neo kann sich nicht einfach nach links drehen. Das wollen wir jetzt ändern. Dazu müssen wir aber erstmal etwas über Operationen lernen.

Operationen von Neo

Wir wollen uns nun die Syntax von Operationen etwas genauer anschauen. Dafür betrachten wir beispielhaft den folgenden Programmcode:

```
public class Neo extends Human {

    // ...

    public void turnAround() {
        this.turnClockWise();
        this.turnClockWise();
    }

    // ...
}
```

In dem Beispiel haben wir die unwichtigen Zeilen durch `// ...` Kommentare ersetzt. Die Operation `turnAround()` existiert noch nicht.
In Blatt 1 habt ihr schonmal die Syntax von Operationen kennen gelernt. Operationen müssen immer in einer Klasse definiert werden. Die Operationen stehen dann auf einem Objekt der Klasse zur Verfügung.
Um Neo eine neue Fähigkeit zu geben, müssen wir die Operation in der Klasse `Neo` definieren. Da `turnAround()` ein Kommando ist, geben wir `void` als Rückgabewert an. Die leeren runden Klammern `()` bedeuten, dass die Operation keinen Parameter hat.
Um andere Operationen des Objekts aufzurufen, nutzen wir in `turnAround()` das Schlüsselwort `this`. Dieses Schlüsselwort kann ähnlich wie eine Variable verwendet werden. Ihr könnt den Wert von `this` aber nicht selber bestimmen. Java bestimmt den Wert von `this` für euch. In der ganzen Klasse `Neo` und damit auch der Operation `turnAround()` könnt ihr mit `this` alle Operationen aufrufen, die ihr auch auf einem Objekt der Klasse `Neo` aufrufen könnt. Das Schlüsselwort wird aber in einer späteren Aufgabe auf Blatt 3 nochmal genauer erklärt.

d) Anstatt Neo *manuell* drei Mal im Uhrzeigersinn drehen zu lassen, wollen wir nun das fehlende Kommando `turnCounterClockwise()` in der Klasse `Neo` implementieren. Finde dafür zunächst die richtige Klasse und die dazugehörige Stelle im Code, in welcher das Kommando eingefügt werden soll (diese Stelle ist auch durch ein Kommentar vorgegeben).

Implementiere anschließend das fehlende Kommando `turnCounterClockwise()`.

Du kannst dafür die Operation aus dem Beispiel oben kopieren und den Namen anpassen. Den Inhalt des Kommandos musst du natürlich auch noch anpassen.

Wenn du möchtest, kannst du auch gleich noch das Kommando `turnAround()` aus dem Beispiel an der gleichen Stelle einfügen.

e) Gehe nun zurück in die `Sheet2Task6` Klasse und löse die Aufgabe c) ein weiteres Mal. Diesmal solltest du aber das neue Kommando `turnCounterClockwise()` benutzen, um Neo nach links zu drehen.

Javadoc

Du hast schon in mehreren Aufgaben Javadoc von Operationen lesen müssen. Vor einer Operation oder einer Klasse kann man einen speziellen Javadoc Kommentar schreiben. Wie andere Kommentare wird er nicht als Programmcode ausgeführt. Eclipse kann diese Kommentare aber benutzen und zeigt sie an, wenn man mit der Maus über einen Aufruf einer Operation fährt.

```
/**
 * Turn Neo around.
 * <p>
 * This operation turns Neo around by calling
 * turnClockWise twice. This operation will fail
 * if Neo is not on a Playfield of a Simulation.
 * </p>
 */
public void turnAround() {
    this.turnClockWise();
    this.turnClockWise();
}
```

Ein Javadoc Kommentar sieht fast genauso aus wie ein mehrzeiliger Kommentar. Der einzige Unterschied ist in der ersten Zeile, hier werden zwei `/**` statt einem `/*` Sterne verwendet. Um die Kommentare einfacher zu unterscheiden, werden sie meist anders eingefärbt.
Der erste Satz ist eine kurze Beschreibung was die Operation macht. Danach kommt ein `<p>` Paragraph um die ausführliche Beschreibung von der kurzen ersten Beschreibung zu trennen. Der Kommentar sollte alles enthalten, was man jemals über die Operation wissen muss. Wenn man zum Beispiel bestimmte Dinge beachten muss, wenn man die Operation benutzen will, dann muss der Javadoc Kommentar darauf hinweisen. Wenn er das nicht tut, dann passiert es sehr schnell, dass die Operation falsch verwendet wird und einen Bug im Programm verursacht. Insbesondere Vorbedingungen sollten in dem Javadoc Kommentar genau beschrieben werden.

f) Nun wollen wir natürlich dafür sorgen, dass unsere Nachfolger auch verstehen was passiert wenn sie das Kommando `turnCounterClockwise()` benutzen. Schreibe Javadoc für diese Operation.

Hinweis: Eclipse kann dir einen Javadoc Kommentar generieren, den du nur noch ausfüllen musst. Dafür musst du in der Zeile direkt über der Operation anfangen `/**` einzugeben. Wenn du dann mit Enter eine neue Zeile anfängst, sollte Eclipse den Rest des Javadoc Kommentars für dich ergänzen. Falls das nicht passiert, solltest du nochmal prüfen, ob du in der richtigen Zeile warst.

g) Die Dokumentation von Operationen soll überraschendes Verhalten bei der Ausführung verhindern. Wenn du Neo um 180° umdrehen willst, kannst du dafür jetzt `turnClockwise()` oder deine neue Operation benutzen. Verhalten sich die beiden Operationen dabei gleich?

Wenn nein: Kannst du das unterschiedliche Verhalten vorhersagen, indem du nur die Dokumentation der beiden Operationen vergleichst? Falls nicht war deine Dokumentation noch nicht genau genug und du solltest sie nochmal verbessern.

Javadoc Nachtrag

Um Parameter, Rückgabewerte und möglicherweise auftretende Exceptions zu dokumentieren gibt es folgende Tags:

i) `@param`: Gefolgt von einem Parameternamen (bspw. `@param n`) erklärt die Bedeutung des Parameters. Die Reihenfolge der `@param` Tags sollte mit der Reihenfolge der Parameter übereinstimmen.

ii) `@return`: Beschreibt den zurückgegebenen Wert unter Annahme, dass die Eingabe korrekt ist und alle Vorbedingungen erfüllt sind.

iii) `@throws`: Gefolgt von der geworfenen Exception beschreibt unter welchen Bedingungen die genannte Exception geworfen wird.

Beispiel für die Anwendung der Parameter:

```
/**
 * Add two positive Intergers together.
 *
 * @param numberA
 *     the first positive Integer to add
 * @param numberB
 *     the second positive Integer to add
 * @return the result of the addition {@code numberA +
    ↪ numberB}
 * @throws IllegalArgumentException
 *     if one or both of the numbers are negative
 */
public Integer addPositiveNumbers(Integer numberA, Integer
   ↪ numberB) {
    if (numberA < 0) {
        throw new IllegalArgumentException("The parameter
           ↪ numberA must be a positive Integer!");
    }
    if (numberB < 0) {
        throw new IllegalArgumentException("The parameter
           ↪ numberB must be a positive Integer!");
    }
    return numberA + numberB;
}
```