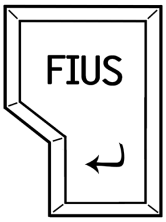

Universität Stuttgart

Übungsblatt 3

Java Vorkurs(WS 2021)

Aufgabe 1: For-Schleifen

a) Passe die Main Operation an die Aufgabe an. Verwende `Sheet3Task1` und `Sheet3Task1Verifier` ↪ .

b) Probiere Neo mithilfe einer While-Schleife 10 Schritte laufen zu lassen.

Hinweis: Wenn du nicht weißt, wie du Neo genau 10 Schritte laufen lässt, kannst du die Simulation mithilfe der Pause Taste stoppen nachdem Neo 10 Schritte gelaufen ist.

For-Schleife

In Java kann man neben der While-Schleife auch eine For-Schleife benutzen. Mit einer For-Schleife kann man genau bestimmen wie oft der Code in der Schleife ausgeführt wird. Anstatt dieselbe Operation n-mal untereinander zu schreiben, um sie n-mal auszuführen, kann man den Code einmal in eine For-Schleife schreiben, die n-mal ausgeführt wird. Dazu verwenden wir eine For-Schleife.

```
//here neo moves five times
neo.move();
neo.move();
neo.move();
neo.move();
neo.move();
```

ist equivalent zu:

```
//this moves neo five times
for (Integer i = 0; i < 5; i++) {
    neo.move();
}
```

Wie die While-Schleife hat auch die For-Schleife einen Schleifenrumpf, der mehrfach ausgeführt wird. Der Inhalt der runden Klammern `()` unterscheidet sich aber deutlich von dem einer While-Schleife.
In den runden Klammern einer For-Schleife stehen insgesamt drei Anweisungen, die jeweils durch ein Semikolon `;` voneinander getrennt sind. Die erste Anweisung `Integer i = 0` setzt die Variable `i` auf den Anfangswert `0`. Der Variablenname `i` wird gerne für Schleifen verwendet und steht häufig für Index. Die erste Anweisung wird nur einmal vor Beginn der Schleife ausgeführt.
Die zweite Anweisung `i < 5` ist vergleichbar mit der Bedingung einer While-Schleife. Sie wird jedes mal überprüft, bevor der Schleifenrumpf ausgeführt wird. Nur wenn sie `true` ist wird der Code-Block der Schleife ausgeführt.
Die dritte Anweisung `i++` wird nach jedem Schleifendurchlauf ausgeführt, bevor die Bedingung erneut überprüft wird. `i++` heißt, dass `i` jeden Schleifendurchlauf um `1` erhöht wird.

> Die obige Schleife wird 5 Mal ausgeführt. Beachte, dass `i` mit dem Wert `0` beginnt und die Schleifenbedingung auf `i < 5` und nicht `i <= 5` prüft. Im letzten Schleifendurchlauf hat `i` den Wert `4`.

c) Laufe nun mithilfe einer For-Schleife 10 Schritte ohne die Simulation zu pausieren.

Hinweis: Dafür musst du die Schleifenbedingung, also die zweite Anweisung, der For-Schleife aus dem Beispiel anpassen.

d) Laufe mit Neo jeweils 3, 7, 14 und 22 Schritte indem du die Schleifenbedingung anpasst. Am Ende sollte Neo auf einem Münzhaufen stehen.

e) Hebe jetzt mit Neo jeweils 2, 5, 16 und 20 Münzen auf. Dazu musst du eine weitere For-Schleife benutzen, in der du auch den Schleifenrumpf anpasst.

f) Verwende für diese Teilaufgabe den Spieler/Neo in der Variable `neoF`.

Hebe alle Münzen auf dem aktuellen Feld auf. Laufe nun 5 Schritte mit einer For-Schleife und lege jeden Schritt so viele Münzen ab, wie du schon Schritte gelaufen bist.

Hinweis: Dafür musst du eine For-Schleife in der For-Schleife verwenden. Dafür musst du in der inneren For-Schleife einen anderen Variablennamen als `i` verwenden. Du kannst die Variable `i` der äußeren For-Schleife in der Schleifenbedingung der inneren For-Schleife benutzen.

g) Verwende für diese Teilaufgabe den Spieler/Neo in der Variable `neoG`.

Laufe 10 Schritte und hebe dabei maximal 5 Münzen pro Feld auf.

Hinweis: In dieser Aufgabe musst du if-Statements verwenden.

```
1 if (neoG.canCollectCoin()) {
2     neoG.collectCoin();
3 }
```

h) Verwende für diese Teilaufgabe den Spieler/Neo in der Variable `neoH`.

Laufe 10 Schritte und hebe dabei alle Münzen auf den Feldern auf.

Hinweis: Hier musst du eine For- und eine While-Schleife verwenden.

Aufgabe 2: If-Conditions

a) Passe die Main Operation an die Aufgabe an.
Verwende `Sheet3Task2` und `Sheet3Task2Verifier`.

b) Mit dem folgenden Codebeispiel kannst du die Anzahl der Münzen in dem Feld unter Neo abfragen und auf der Konsole ausgeben.

```
1 Integer coinsUnderNeo = neo.getCurrentlyCollectableCoins().size
    ↪ ();
2 System.out.println(coinsUnderNeo);
```

Mit der Operation `neo.getCurrentlyCollectableCoins();` bekommt man eine Liste aller Münzen, die auf Neos aktuellem Feld liegen. Eine Liste hat eine Größe die angibt wie viele Elemente sie enthält. Die Größe der Liste aller Münzen entspricht genau der Anzahl der Münzen auf Neos aktuellem Feld. Die Größe kann abgefragt werden mit `list.size()`.

Laufe mit Neo bis zum ersten Feld mit einer Münze und gib die Anzahl der Münzen unter Neo auf der Konsole aus.

If-Statements 2

If-Statements kennt ihr ja schon. Diese können auch dazu verwendet Zahlen (z.B Integer) miteinander zu vergleichen.

```
Integer x = 2;
if ( x < 3 ) {
    System.out.println("x ist kleiner als 3!");
}
```

Das kann man zum Beispiel dazu verwenden, um zu schauen wie viele Münzen Neo hat. In Java kannst du die folgenden Vergleichsoperatoren verwenden:

Text	Math. Zeichen	Java	Bemerkung
größer als	$>$	>	
kleiner als	$<$	<	
gleich	$=$	==	mit dem doppelten == sollte man nur Zahlen und keine Objekte vergleichen! Ein einfaches == ist in Java eine Zuweisung!
ungleich	$\neq$	!=	
größer gleich	$\geq$	>=	
kleiner gleich	$\leq$	<=	

Damit können wir nun prüfen, ob sich auf Neos Feld genau 3 Münzen befinden:

```
if ( neo.getCurrentlyCollectableCoins().size() == 3 ){
    System.out.println("neo hat genau 3 Münzen!");
}
```

== und .equals()

Vorsicht: Mit == sollte man nur Zahlen vergleichen! Zum Vergleichen von zwei Objekten sollte immer die `.equals` Abfrage von einem der beiden Objekte verwendet werden:

```
if (neo1.getPosition().equals(new Position(1, 1))){
    System.out.println("Neo is on Position (1, 1)");
}
```

Mit dem folgenden Codebeispiel kannst du in der Konsole sehen, dass == und .equals nicht zu dem gleichen Ergebnis kommen, wenn man zwei Objekte vergleichen will.

```
System.out.println(new Position(1,1) == new Position(1,1)); //
    ↪ false
System.out.println(new Position(1,1).equals(new Position(1,1)))
    ↪ ; // true
```

c) Neo soll sich ab jetzt immer geradeaus bewegen, solange keine Münze unter ihm liegt. Wenn Neo auf eine Münze trifft, sollst du erst mal anhalten.

Tipp: Verwende hierfür am besten eine Endlosschleife und schreibe den Code für alle Unteraufgaben dieser Aufgabe in den Schleifenkörper. Eine Endlosschleife kann z.B. so aussehen:

```
1 while(true){
2 //your code here
3 }
```

Endlosschleifen und `break`

Eine While-Schleife, in der die Schleifenbedingung `true` ist, ist eine Endlosschleife. Das ist nicht nur ein Name. Die Schleife wird niemals aufhören! Deshalb kann es passieren, dass du den Stopp Knopf in der Eclipse Konsole brauchst, um die Schleife wieder abzubrechen. Endlosschleifen sollte man wann immer möglich vermeiden. Aber man kann auch eine Endlosschleife wieder beenden. Dafür braucht man das Schlüsselwort `break`. Wenn du in einer Schleife `break` aufrufst, dann wird die Schleife sofort unterbrochen. Auch die Schleifenbedingung wird dann nicht mehr überprüft. Das funktioniert auch in einer For-Schleife.

```
while(true){
    // do something
    if (hadEnough) {
        break;
    }
}
```

d) Wenn sich Neo auf einem Feld mit genau einer Münze befindet, dann lass ihn sich nach rechts drehen und diese Münze aufsammeln. Danach soll Neo erst mal einen Schritt geradeaus gehen, um wieder von dem Feld herunter zu kommen.

Wenn er dann auf einem leeren Feld steht soll Neo wieder geradeaus laufen bis er erneut auf einem Feld mit Münzen steht. Wenn er direkt auf einem Feld mit Münzen steht, soll Neo wieder prüfen wie viele Münzen auf dem Feld sind und sich danach entscheiden, was er als nächstes tun muss.

e) Wenn sich mehr als eine Münze unter Neo befindet, dann soll Neo eine Münze aufsammeln und sich nach links drehen bevor er wieder von dem Feld heruntergeht. Die anderen Münzen soll er liegenlassen.

Logische Operationen

Wenn mehrere Bedingungen für ein Ereignis eine Rolle spielen, muss man diese logisch verknüpfen. Java verwendet hierfür folgende Syntax:

Text	log. Zeichen	Java Zeichen	Bemerkung
und	$\wedge$	`&&`	Achtung kein einfaches `&` in Java verwenden, das macht etwas anderes (aber ähnliches)
oder	$\vee$	`||`	Kein einfaches `|` verwenden (analog zu und)
nicht	$\neg$	`!`	

Ein paar Beispiele:

```
if ( neo.canMove() && (neo.getCurrentlyCollectableCoins().size
    ↪ ()==7) ){

    //neo hat genau 7 Münzen UND kann nach vorne gehen.
}
```

```
if ( (!neo.canMove()) && (neo.getCurrentlyCollectableCoins().
    ↪ size()==7) ){
    //neo hat genau 7 Münzen UND kann sich NICHT nach vorne
        ↪ gehen.
}
```

```
if ( neo.canMove() || neo.getCurrentlyCollectableCoins().size()
    ↪ ==7) ){
    //neo hat genau 7 Münzen ODER kann nach vorne gehen.
}
```

Zu beachten ist außerdem, dass das „oder“ `||` kein ausschließendes „oder“ ist, daher ist `((2 < 3)|| (42 != 4))` eine wahre Aussage.

f) Bevor Neo gegen eine Wand läuft soll er sich umdrehen. Dafür musst du vor jedem `move()` Kommando prüfen, ob Neo sich überhaupt nach vorne bewegen kann.

g) Jetzt wollen wir, dass Neo auch irgendwann aufhört. Er soll aufhören zu laufen oder Münzen einzusammeln, wenn er schon 20 Münzen gesammelt hat oder wenn er auf einem Feld mit exakt 9 Münzen ankommt. Mit `neo.getCoinsInWallet()` kannst du die Anzahl der Münzen in Neos Inventory abfragen.

Tipp: Hierfür kannst du entweder das Argument der Endlosschleife ändern oder `break` Verwenden.

Aufgabe 3: Operationen

a) Passe die Main Operation an die Aufgabe an.
Verwende `Sheet3Task3` und `Sheet3Task3Verifier`.

b) Lass Neo das folgende Muster laufen:

laufe 2 Schritte
... drehe dich nach rechts
... laufe einen Schritt
... drehe dich nach links
... laufe 3 Schritte
... drehe dich nach links
... laufe 2 Schritte
... drehe dich nach rechts
... laufe 2 Schritte
... drehe dich nach rechts
... laufe einen Schritt
... drehe dich nach links.

c) Laufe das Muster aus b) mithilfe einer For-Schleife 3 Mal.

Operation um Code mehrfach zu verwenden

Wenn man den selben Code an mehreren Stellen verwenden möchte ist es nicht sinnvoll diesen mehrmals zu kopieren. Stattdessen kann man den Code in eine Operation auslagern. Dann kann man den gesamten Code an verschiedenen Stellen mit nur einer Zeile aufrufen.

```
1  public class DemoTask implements Task {
2
3      @Override
4      public void run(Simulation sim) {
5          Neo neo = new Neo();
6          // Anfang für Beispiel weggelassen
7
8          neo.collectCoin();
9          this.moveTwiceAndTurn(neo);
10         neo.collectCoin();
11         this.moveTwiceAndTurn(neo);
12     }
13
14     public void moveTwiceAndTurn(Neo neo) {
15         // move
16         neo.move();
17         neo.move();
18         // and turn
19         neo.turnClockWise();
20     }
21
22 }
```

In Aufgabenblatt 2 hast du schon eine Operation auf Neo implementiert. Da auch die Operation im Beispiel ein Kommando ist, hat sie ebenfalls keinen Rückgabewert. Deshalb ist hier vor dem Operationsnamen `moveTwiceAndTurn` `void` angegeben.

Anders als bei dem Kommando in der Klasse Neo können wir diesmal nicht mit `this` auf die Operationen von einem Objekt der Klasse `Neo` zugreifen. In einer Task Klasse haben wir mit `this` nur Zugriff auf die Operationen in der Task Klasse selbst. In dem Beispiel oben können wir `this.run(/*...*/)` oder `this.moveTwiceAndTurn(/*...*/)` aufrufen.
Achtung: Wenn ihr in `moveTwiceAndTurn` wieder `this.moveTwiceAndTurn(/*...*/)` aufruft, habt ihr eine Rekursion gebaut. Theoretisch könnte die Rekursion endlos weitergehen, aber Java bricht eine endlose Rekursion ab einem bestimmten Zeitpunkt ab. (Wenn ihr mehr über Rekursion wissen wollt, dann sucht am besten auf Google nach Rekursion.)
Da wir die Operationen von Neo nicht über `this` aufrufen können, müssen wir der neuen Operation Neo als Parameter übergeben. Parameter werden in Aufgabe 4 nochmal genauer erklärt.

d) Lagere nun den Code aus b) in die gegebene Operation `movePattern` aus.

e) Ersetze nun den Code in der Schleife aus c) durch die Operation `movePattern`.

f) In der Klasse `Sheet3Task3` findest du auch die folgende Operation.

```
1 private void dropFourCoinsAndTurnLeft(Neo neo) {
2     //write the Code for f) here
3     //make neo drop four coins
4     //make neo turn Left
5 }
```

Neo soll in dieser Operation 4 Münzen fallen lassen und sich nach links drehen. Ersetze den Kommentar in der Operation durch die passenden Kommandos.

g) Führe nun in der Schleife aus c) nach `movePattern` einmal `dropFourCoinsAndTurnLeft` aus. Der Schleifenkörper sollte beide Operationen enthalten.

h) Nun möchten wir eine eigene Operation schreiben. Kopiere dazu erst mal die `movePattern` Operation und ändere den Namen in etwas anderes (z.B. `moveStraight`). Ersetze nun in der Schleife aus c) `movePattern` durch die neue Operation.

Du solltest jetzt in der Klasse `Sheet3Task3` zwei Operationen haben, die sich nur im Namen unterscheiden.

i) Schreibe die Operation aus h) so um, dass Neo sich nur durch geradeaus laufen, nach Ausführung der Operation, an der selben Stelle befindet wie nach dem Ausführen der Operation `movePattern`. Wenn Neo an der gleichen Position startet, dann sollte er, egal ob `movePattern` oder die neue Operation ausgeführt wurde, an der gleichen Position stehenbleiben.

Teste deine Implementierung und überprüfe ob die Münzen an den gleichen Stellen abgelegt werden, wenn du in der Schleife `movePattern` durch deine neue Operation ersetzt hast.

Aufgabe 4: Operation parameters

a) Passe die Main Operation an die Aufgabe an.
Verwende `Sheet3Task4` und `Sheet3Task4Verifier`.

b) Schreibe ein Kommando das Neo einen Schritt nach vorne gehen lässt.

c) Schreibe je ein Kommando das Neo 2,3 und 4 Schritte nach vorne gehen lässt. Du solltest jetzt vier Kommandos haben, mit denen du Neo 1, 2, 3 oder 4 Schritte gehen lassen kannst.

Operations Parameter

Wenn du eine Operation schreiben möchtest die je nach Situation mit verschiedenen Variablen die selbe Operation ausführt, kannst du der Operation einen oder mehre Parameter übergeben. Bisher haben wir einer Operation immer den Neo übergeben mit dem die Operation ausgeführt werden soll. Man kann aber jede Art von Objekt als Parameter übergeben.

```
1 public void printNumbers(Neo neo, Integer from, Integer to){
2     for(Integer i=from; i <= to; i++){
3         System.out.print(i);
4         neo.turnClockWise();
5     }
6 }
```

Diese Abfrage gibt die Zahlen von `from` bis `to` aus. Z.B. `printNumbers(2,7)` gibt `234567` aus. Für jede Zahl die ausgegeben wird, dreht sich Neo ein mal nach rechts.

d) Schreibe nun ein Kommando in dem Neo `n` Schritte läuft. Dazu braucht das Kommando zwei Parameter. Einen Parameter für Neo und einen für die Anzahl der Schritte.

e) Verwende das Kommando aus d) um Neo eine größer werdende Spirale laufen zu lassen. Das heißt:

Laufe *einen* Schritt.
Drehe dich nach rechts.
Laufe *zwei* Schritte.
Drehe dich nach rechts.
Laufe *drei* Schritte.
Drehe dich nach rechts.
Laufe *vier* Schritte.
Drehe dich nach rechts.
...usw.

Du solltest mit der Spirale aufhören, wenn Neo 12 oder mehr Schritte am Stück gegangen ist.

Aufgabe 5: Inputtest and Exeptions

a) Passe die Main Operation an die Aufgabe an.
Verwende den selben Task und Verifier wie in Aufgabe 4.

b) Probier die Operation aus Aufgabe 4 d) mit unterschiedlichen Werten aus: (10, 1, 0, -1, `Integer.MAX_VALUE`)
Was passiert?

Return

Mit dem Schlüsselwort `return` kannst du den Rückgabewert einer Operation festlegen. Der Rückgabewert muss nach dem Schlüsselwort `return` und vor dem `;` stehen.
Return ist immer die letzte Anweisung einer Operation, die ausgeführt wird. Du kannst mit `return` also die Operation beenden. Das funktioniert sogar in Kommandos, die ja keinen Rückgabewert haben.
Wenn nach dem `return` noch Anweisungen stehen, die ganz sicher nicht mehr erreicht werden können, dann wird Java das Programm nicht mehr ausführen wollen. Ihr könnt die Zeilen nach dem Return (bis zum Ende des Code-Blocks der Operation) auskommentieren, damit Java das Programm wieder ausführt.
Beispiel für `return` in einer Abfrage:

```
1 public Integer getAnswer(){
2     return 42;
3 }
```

Wenn man `getAnswer()` aufruft bekommt man den Wert 42 zurück.
Beispiel für `return` in einem Kommando:

```
1 public void printNumbers(){
2     for(int i=0; i < 10; i++){
3         System.out.print(i);
4         if ( i == 5){
5             return;
6         }
7     }
8 }
```

Wenn man nun `printNumbers();` aufruft dann ist die Ausgabe in der Konsole `012345`, da die Operation bei i=5 beendet wird.

c) Wir wollen nun, dass Neo in dieser Operation nicht mehr als 50 Schritte laufen kann. Wenn Neo mehr als 50 Schritte laufen soll, dann soll er sich gar nicht bewegen. Füge ein am Anfang des Kommandos aus Aufgabe 4 d) ein IF-Statement ein, in dem du prüfst ob der Parameter für die Anzahl der Schritte größer als 50 ist. Wenn ja, dann nutze `return;` um die Ausführung der Operation direkt abzubrechen.

d) Probiere nun erneut unterschiedliche Werte, insbesondere die Werte aus b), aus. Das Verhalten der Operation sollte sich geändert haben.

Throwing Exceptions

Wir möchten nicht, dass ungültige Usereingaben einfach ignoriert werden. Darum sollte man bei ungültigen Eingaben eine Exception werfen. Für eine Erklärung von Exceptions kannst du dir nochmal Aufgabe 2 auf Blatt 2 ansehen.
Um eine Exception zu werfen verwendet man das Schlüsselwort `throw`.

```
1 public void printNumbersTo(Integer to) {
```

```
    if (i < 0) {
        throw new IllegalArgumentException("negative Number"
            ↪ );
    }
    for(int i=0; i <= to; i++) {
        System.out.print(i);
    }
}
```

Diese Operation wirft bei negativen Werten eine `IllegalArgumentException`.

e) Ändere nun die Operation so, dass zu große Zahlen die Operation nicht mehr mit einem `return;` abbrechen lassen. Stattdessen soll eine `IllegalArgumentException` mit einer passenden Nachricht geworfen werden. Du kannst die Zeile mit dem `return;` einfach durch die Zeile `throw new IllegalArgumentException("");` ersetzen.

Wähle eine hilfreiche Nachricht für deine `IllegalArgumentException`.

f) Jetzt bekommt man eine rote Fehlermeldung, wenn man versucht die Operation mit einem zu großen Parameter aufzurufen. Wir bekommen also den Fehler deutlich mit.

Noch besser wäre es, wenn wir den Fehler gar nicht erst verursachen würden. Dazu müssen wir wissen, dass diese Operation eine Exception wirft, wenn der Parameter zu groß ist. Das kann man im Javadoc Kommentar der Operation dokumentieren.

Darum erweitern wir jetzt den Javadoc Kommentar der Operation um zu erklären wann die Exception geworfen wird. Verwende dafür `@throws` in dem Javadoc Kommentar.

Hier ist nochmal ein Beispiel wie man `@throws` in einem Javadoc Kommentar verwendet.

```
/**
 * prints the numbers from 0 to 'to' in consecutive order.
 *
 * @param to the number to which to print.
 * @throws IllegalArgumentException
 *         if 'to' is negative
 */
public void printNumbersTo(int to){
    // inhalt wie im Beispiel oben
}
```

Aufgabe 6: What is `this`?

a) Passe die Main Operation an die Aufgabe an.
Verwende `Sheet3Task6` und `Sheet3Task6Verifier`.

b) Versuche die Operation aus Aufgabe 4 d) aufzurufen.

Hinweis: Dein Versuch darf auch schiefgehen. Es ist zwar möglich, aber nicht auf direktem Weg. Du kannst also nicht in der Klasse `Sheet3Task6` einfach den Namen einer Operation aus einer anderen Klasse schreiben und erwarten, dass Java schon das Richtige macht.

c) Kopiere die komplette Operation aus Aufgabe 4 d) in die Neo Klasse und versuche erneut diese aufzurufen. Achte darauf, dass die Operation `public` ist. Rufe nun diese Operation in `Sheet3Task6` auf mit `neo.<dein Operationsname>(neo, 5)`.

d) Bei dem Operationsaufruf in c) kommt die Variable `neo` zwei mal vor. Solche Dopplungen kann man fast immer vermeiden.

Gehe wieder in die Neo Klasse und lösche den Parameter `Neo neo`. Die runden Klammern und den anderen Parameter musst du dabei behalten.

Ersetze alle Vorkommen in der Operation von `neo.` mit `this.`.

Jetzt musst du noch deinen Aufruf der Operation anpassen, da die Operation nun einen Parameter weniger benötigt.

this

In Blatt 2 hast du erfahren, dass Java den Wert von `this` automatisch festlegt. Jetzt können wir erklären, wie das passiert. Dafür nehmen wir die folgenden zwei Klassen:

```
1 public class Neo extends Human {
2
3     public void turnAround() {
4         this.turnClockWise();
5         this.turnClockWise();
6     }
7 }
```

```
1 public class Sheet3Task6 implements Task {
2
3     public void run(Simulation sim) {
4         Neo neo = new Neo();
5         // other stuff
6
7         neo.turnAround();
8     }
9 }
```

In der `run(/*...*/)` Operation der Klasse `Sheet3Task6` wird in Zeile 7 `neo.turnAround();` aufgerufen. In der Variable `neo` ist das in Zeile 4 erstellte Objekt der Klasse `Neo` gespeichert. In Zeile 7 wird also auf diesem Neo Objekt die Operation `turnAround()` aufgerufen.
Die Operation `turnAround()` ist in der Klasse des Neo Objekts, also in der Klasse `Neo` in Zeile 3 definiert. Java setzt für diesen einen Aufruf der Operation `turnAround()` auf dem in `Sheet3Task6` Zeile 4 erzeugten Neo Objekt `this = Sheet3Task6::run:neo`. (Die Notation `Sheet3Task6::run:neo` ist frei erfunden und soll nur deutlich machen, dass es sich um das Objekt aus der Variable `neo` aus der Operation `run` der KlasseSheet3Task6 handelt.)

In den meisten Fällen kann man vereinfacht sagen: das `this` in einer Operation ist das, was beim Aufruf der Operation vor dem Punkt steht. Allerdings muss man beachten, das Variablen selber keine Objekte sind, sondern nur ein Name (oder eine Adresse) mit dem man ein Objekt mehrfach benutzen kann. Der Unterschied von Variablen und Objekten wird auch in PSE nochmal genauer erklärt.

e) **(Optional)** Welche Operationen kannst du mit `this.` aufrufen und welche mit `playerA.`? Kannst du unterschiedliche Operationen aufrufen, wenn ja welche?

f) Gehe zu Aufgabe 4 zurück und verwende ebenfalls die Operation die auf Neo definiert ist.

g) **(Optional)** Versuche nochmal b) zu lösen.

Hinweis: Du brauchst ein Objekt der Klasse, um einer Operation der Klasse aufrufen zu können.