Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 03

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Karsten Weihe

Übungsblattbetreuer:	Nick Steyer
Wintersemester 22/23	v1.0-SNAPSHOT
Themen:	Erste eigene Klassen
Relevante Foliensätze:	01e und 01f (und natürlich auch weiterhin 01a-01d)
Abgabe der Hausübung:	18.11.2022 bis 23:50 Uhr

Hausübung 03 — Gesamt: 25 Punkte

Ihr Upgrade in die First Class

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* in unserem Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung in der Vorlage relevanten Verzeichnisse sind `src/main/java/h03` und `src/test/java/h03`.

Einleitung

Verbindliche Anforderung für die gesamte Hausübung:

Mittels JavaDoc und den Tags `@param` und `@return` wollen wir nun im Java-Teil eine geeignete Dokumentation unserer Methoden vornehmen. Das JavaDoc können Sie automatisch vor jeder Methode mittels `/**` gefolgt von der Enter-Taste generieren (z. B. in IntelliJ). Ein Beispiel könnte wie folgt aussehen:

JavaDoc

```java
/**
* This method accepts two real numbers belonging to a
* vector and calculates the euclidean norm of said
* vector.
*
* @param x first component of two-dimensional vector (x, y)
* @param y second component of two-dimensional vector (x, y)
* @return Euclidean norm of the vector (x, y)
*/
double euclid2(float x, float y) {
    return Math.sqrt(x * x + y * y);
}
```

Die Dokumentation mit Vertrag ist Pflicht für alle Java-Methoden! Orientieren Sie sich dabei an obigem Beispiel. Für jede fehlende Dokumentation einer Methode Ihrer abgegebenen Hausübung erhalten Sie einen Punkt Abzug auf Ihre erreichten Punkte. Sie können auf diese Art und Weise bis zu 20% der Punkte einer Hausübung abgezogen bekommen!

Verbindliche Anforderung für die gesamte Hausübung:

Sie werden in dieser Übung Klassen und Attribute erstellen. Manche Attribute werden von Methoden verändert, andere bleiben konstant und verändern sich nicht. Um Fehlern vorzubeugen und den Code besser verständlich zu machen, ist es ratsam, konstante Attribute mit dem Schlüsselwort `final` zu versehen und somit vor versehentlichen Änderungen zu schützen.

Es ist daher Ihre Pflicht, unveränderliche Attribute mittels des `final`-Modifiers zu versehen. Sie werden in der Aufgabenstellung *nicht* explizit darauf hingewiesen und müssen dies *selbstständig* erledigen! Sie dürfen hierzu auch die eingebauten Analyseprogramme in Ihrer Entwicklungsumgebung verwenden.[a] Diese weisen Sie in der Regel darauf hin, wenn ein Attribut konstant ist, aber nicht mit `final` als solches markiert wurde.

[a] Für IntelliJ z.B. `https://www.jetbrains.com/help/idea/2022.2/running-inspections.html`

Bisher haben Sie Klassen verwendet, die von anderen Menschen definiert worden sind, insbesondere die Klasse `Robot`. Auf diesem Übungsblatt schreiben Sie nun erste Klassen selbst.

H1: Roboter mit Abkömmling 12 Punkte

Hier schreiben Sie schrittweise eine Klasse `RobotWithOffspring`, und zwar in einer bereits erstellten Datei `RobotWithOffspring.java` im Package `h03`.

H1.1: Abgeleitete Klasse, ihr Konstruktor und zusätzliche Attribute 4 Punkte

Analog zu Kapitel 01f, Folien 2-28 der FOP, Beispiel `SymmTurner`: Leiten Sie von Klasse `Robot` direkt die oben erwähnte `public`-Klasse namens `RobotWithOffspring` ab, wie gesagt, in Datei `RobotWithOffspring.java`.

Analog zu Kapitel 01f, Folien 29-48 der FOP, Beispiel `SlowMotionRobot`: Fügen Sie in die Klasse `RobotWithOffspring` zwei `private`-Attribute `numberOfColumnsOfWorld` und `numberOfRowsOfWorld` jeweils vom Typ `int` ein.

> **Unbewertete Verständnisfrage:**
>
> Die beiden Attribute hätte man auch einfach z. B. `m` und `n` nennen können. Was spräche dagegen? Hätte es nicht wenigstens gereicht, die beiden Attribute kürzer `numberOfColumns` und `numberOfRows` zu nennen?

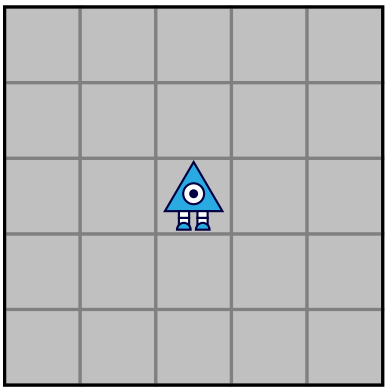
Fügen Sie als nächstes einen `public`-Konstruktor in `RobotWithOffspring` ein (vgl. Kapitel 01e, Folien 111-135 der FOP sowie Kapitel 01f, Folien 25-28 der FOP und Kapitel 01f, Folien 36-38 der FOP). Dieser hat vier formale Parameter in dieser Reihenfolge: `numberOfColumnsOfWorld` und `numberOfRowsOfWorld` jeweils vom Typ `int`, `direction` vom Typ `Direction` und `numberOfCoins` vom Typ `int`. Der Konstruktor von `RobotWithOffspring` ruft den Konstruktor von `Robot` mit folgenden aktualen Parametern auf: Die beiden formalen Parameter `direction` und `numberOfCoins` des Konstruktors von `RobotWithOffspring` werden unverändert die letzten beiden aktualen Parameter des Konstruktors von `Robot`.

Die ersten beiden Parameter im Aufruf des Konstruktors von `Robot`, also die Spalten- und Zeilenanzahl der Welt, werden etwas komplizierter berechnet: Der Konstruktor von `RobotWithOffspring` darf ohne Nachprüfung davon ausgehen, dass `numberOfColumnsOfWorld` und `numberOfRowsOfWorld` tatsächlich die Zeilen- und Spaltenzahl der `World` sind. Unter dieser Voraussetzung setzt der Konstruktor das zu konstruierende `RobotWithOffspring`-Objekt genau mittig, konkret auf die Spalte $\lfloor \text{Spaltenanzahl}/2 \rfloor$ und die Zeile $\lfloor \text{Zeilenanzahl}/2 \rfloor$[1].

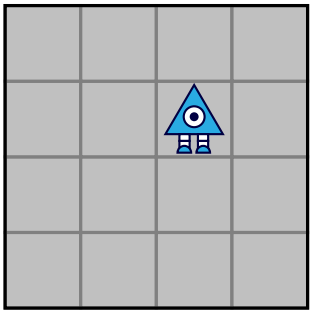
> **Hinweis:**
>
> Beachten Sie, dass die Zeilen und Spalten der `World` von unten links an jeweils bei 0 beginnend durchnummeriert werden (vgl. Kapitel 01a, Folien 12-19 der FOP). Zur Platzierung des Roboters überlegen Sie sich, in welcher Beziehung der Wert des mathematischen Ausdrucks $\lfloor \text{Spaltenanzahl}/2 \rfloor$ zum Ergebnis der Division mit Rest in Java `numberOfColumnsOfWorld/2` steht (vgl. Kapitel 01a, ab Folie 107 der FOP). Sie finden in Abbildung 1 einige Beispiele, wie der Roboter zu platzieren ist.

Außerdem werden im Konstruktor von `RobotWithOffspring` die beiden Attribute `numberOfColumnsOfWorld` und `numberOfRowsOfWorld` mit den aktualen Werten der gleichnamigen formalen Parameter initialisiert.

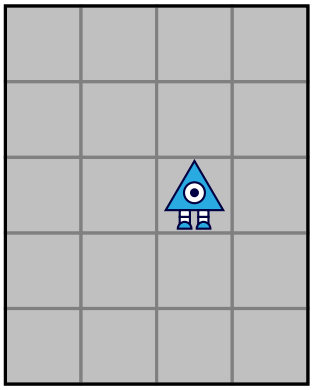
[1] Die Gaußklammern geben an, dass das Ergebnis der Division abzurunden ist. Vgl. auch `https://de.wikipedia.org/wiki/Abrundungsfunktion_und_Aufrundungsfunktion`



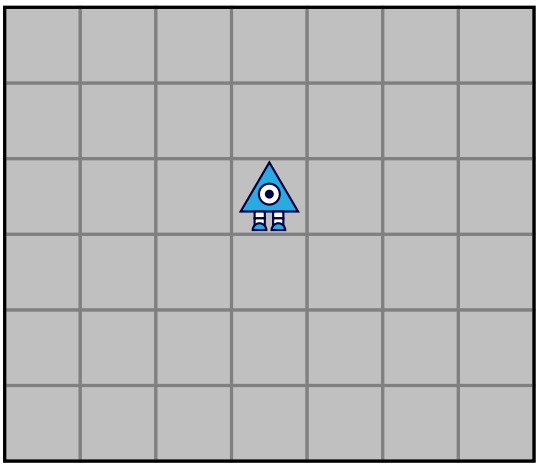
(a) Die Welt besitzt 5 Zeilen und 5 Spalten. Der Roboter befindet sich in Zeile 2, Spalte 2.



(b) Die Welt besitzt 4 Zeilen und 4 Spalten. Der Roboter befindet sich in Zeile 2, Spalte 2.



(c) Die Welt besitzt 5 Zeilen und 4 Spalten. Der Roboter befindet sich in Zeile 2, Spalte 2.



(d) Die Welt besitzt 6 Zeilen und 7 Spalten. Der Roboter befindet sich in Zeile 3, Spalte 3.

Abbildung 1: Platzierung des Roboters

Anmerkungen:

Bevor Sie mit H1.2 fortfahren, prüfen Sie, ob Ihre soweit definierte Klasse `RobotWithOffspring` fehlerfrei durch den Compiler geht. Sobald das der Fall ist, prüfen Sie einmal testweise, was der Compiler dazu sagt, wenn der Aufruf des Konstruktors der Basisklasse ***nicht*** die erste Anweisung im Konstruktor der abgeleiteten Klasse ist.

Machen Sie diesen Fehler rückgängig, sodass ihr Code wieder kompilierbar ist. Richten Sie dann in `Main.java` in der Methode `SandboxTests` an der wie üblich mit „`// Put your code here`“ bezeichneten Stelle einen Verweis und ein Objekt vom Typ `RobotWithOffspring` ein (vgl. `SymmTurner` in Kapitel 01f, ab Folie 4 der FOP). Prüfen Sie wieder per Augenschein und mithilfe von Konsolenausgaben, ob die vier Attribute des Roboters die erwarteten Werte haben und vor allem ob der Roboter tatsächlich immer mittig platziert wird. Variieren Sie wie in vorangegangenen Hausübungen die Spalten- und Zeilenzahl der `World`, indem Sie die übergebenen Parameter der Methode `World.setSize()` in der Methode `sandboxTests` anpassen. Achten Sie dabei insbesondere darauf, dass Sie sowohl gerade als auch ungerade Spalten- und Zeilenzahlen testen – denn Ihr Code könnte ja in dem einen Fall korrekt und dennoch in dem anderen Fall inkorrekt sein. Es reicht also nicht, nur einen der beiden Fälle zu testen.

H1.2: Attribut vom Referenztyp und `get`-Methoden für dessen Attribute — 4 Punkte

Fügen Sie der Klasse `RobotWithOffspring` ein `protected`-Attribut `offspring` vom Typ `Robot` hinzu. Dieses Attribut wird im Konstruktor noch nicht initialisiert (Vorgriff auf Kapitel 03b der FOP: es ist dann automatisch mit `null` initialisiert).

Fügen Sie in die Klasse `RobotWithOffspring` eine `public`-Methode `initOffspring` ein. Diese hat einen formalen Parameter `direction` vom Typ `Direction` und einen formalen Parameter `numberOfCoins` vom Typ `int` (in dieser Reihenfolge). Sie können sich hier an Kapitel 01e, ab Folie 111 der FOP orientieren. Die Methode hat keinen Rückgabetyp. Sie erstellt ein Objekt vom Typ `Robot` und lässt `offspring` darauf verweisen. Das neue `Robot`-Objekt hat dieselbe Spalten- und Zeilennummer (also dieselbe Position) wie das `RobotWithOffspring`-Objekt, auf dem `initOffspring` aufgerufen wird, ***in diesem Moment*** hat. Sie müssen hier also `getX` und `getY` geeignet in die Parameterliste im Konstruktoraufruf von `Robot` integrieren (vgl. `turnLeft` in `turnRight` von `SymmTurner` in Kapitel 01f, ab Folie 18 der FOP). Die Richtung und die Anzahl Münzen für das neue `Robot`-Objekt werden unverändert aus den beiden aktualen Parametern von `initOffspring` entnommen.

Fügen Sie nun analog zu Kapitel 01e, Folien 75-86 der FOP vier öffentliche `get`-Methoden in die Klasse `RobotWithOffspring` ein: `getXOfOffspring`, `getYOfOffspring`, `getDirectionOfOffspring` und `getNumberOfCoinsOfOffspring`. Jede dieser vier Methoden setzt ohne Nachprüfung voraus, dass `offspring` schon initialisiert ist, und liefert einfach das Ergebnis der entsprechenden Methode von `offspring` zurück.

Fügen Sie in die Klasse `RobotWithOffspring` eine `public`-Methode `offspringIsInitialized` ein. Diese hat keine formalen Parameter und den Rückgabetyp `boolean`. Die Methode liefert genau dann `true` zurück, wenn `offspring` initialisiert wurde.[2]

Hinweise:

Bevor Sie mit H1.3 fortfahren, prüfen Sie, ob Ihre soweit definierte Klasse `RobotWithOffspring` weiterhin fehlerfrei durch den Compiler geht, ob die neue boolesche Methode das erwartete Ergebnis liefert (a) vor bzw. (b) nach dem ersten Aufruf von `initOffspring` und ob die vier neuen `get`-Methoden das jeweils erwartete Ergebnis zurückliefern (***nach*** dem ersten Aufruf von `initOffspring`). Sie können `initOffspring` auch testweise mehrfach mit verschiedenen Parametern aufrufen, um sicherzustellen, dass jeder Aufruf von `initOffspring` tatsächlich das Resultat des vorhergehenden Aufrufs überschreibt.

[2] Probieren Sie hierfür ggf. einmal aus, wie sich der Wert des Attributs `offspring` vor und nach dem Aufruf von `initOffspring` unterscheidet.

Exkurs (`NullPointerException`):

Rufen Sie eine der `get`-Methoden auch einmal ***vor*** dem ersten Aufruf von `initOffspring` auf. Der Programmlauf sollte nun mit einer Fehlermeldung beendet werden. Ergibt diese Fehlermeldung für Sie Sinn?[a]

Sie haben in Ihrem Code bisher stets eine Instanz einer Klasse (also ein Objekt) erstellt und dann auf Methoden und Attribute dieser Instanz zugegriffen. Beispielsweise:

Zugriff auf eine Instanz

```
1 Robot myRobot = new Robot(2, 3, UP, 42);
2 Direction currentDirection = myRobot.getDirection();
3 myRobot.move();
```

Hier ist klar, was zu tun ist: es soll auf das in Zeile 1 erstellte Objekt `myRobot` zugegriffen und mit `myRobot.getDirection()` die Richtung ausgelesen bzw. mit `myRobot.move()` der Roboter bewegt werden. Was aber, wenn wir folgendes schreiben?

Zugriff auf `null`

```
1 Robot myRobot = null;
2 Direction currentDirection = myRobot.getDirection();
3 myRobot.move();
```

Wir wollen wieder in Zeile 2 auf die Richtung zugreifen und in Zeile 3 den Roboter bewegen. Aber welchen Roboter verwenden wir hier? In Zeile 1 wird kein Objekt erstellt; die Referenz – oder der *Pointer* – ist `null`. Dieser Code kann nicht ausgeführt werden, denn was soll bei `myRobot.getDirection()` zurückgeliefert werden, wenn `myRobot` gar keine gültige Referenz auf einen Roboter ist, sondern `null`? Daher wird an dieser Stelle eine Ausnahme ausgelöst, genauer eine `java.lang.NullPointerException`. Ausnahmen werden in Kapitel 05 der FOP und in Übungsblatt 08 genauer behandelt. An dieser Stelle sehen Sie die Ausnahme als Fehlermeldung in der Konsole.

Es gibt noch andere Fälle, die eine `NullPointerException` auslösen können. Beispielsweise führt die Zuweisung „`myArray[2] = 23;`“ zu einer solchen Ausnahme, wenn der Array `myArray` gleich `null` ist. Werfen Sie auch einen Blick in die Java-Dokumentation, wo ein paar dieser Fälle aufgelistet sind.

Eine Ausnahme wie die `NullPointerException` gibt es auch in anderen objektorientierten Programmiersprachen. Manchmal wird sie anders bezeichnet – z. B. `NullReferenceException`[b] in C# – es handelt sich aber im Kern um dasselbe.

[a] Sie sollten die Fehlermeldung in der Konsole sehen, nachdem Sie die Anwendung mit dem Gradle-Task „application/run“ gestartet haben. Ggf. müssen Sie geöffnete FOPBot-Fenster schließen, bis die Fehlermeldung angezeigt wird.
[b] vgl. https://docs.microsoft.com/dotnet/api/system.nullreferenceexception

Ausblick:

In Kapitel 05 der FOP und in späteren Hausübungen werden Sie sehen, wie man solche Fehlersituationen abfangen kann (Stichwort „Exceptions“). Der Programmlauf wird dann automatisch in eine von Ihnen zu implementierende Fehlerbehandlung umgeleitet, und nach dieser Fehlerbehandlung kann der Prozess normal weiterlaufen. Eine solche Möglichkeit ist natürlich sehr wichtig, denn viele Prozesse dürfen nicht einfach „abstürzen“ – auch dann nicht, wenn sie in einen bislang unentdeckten Programmierfehler hineingelaufen sind, weil vom Weiterlaufen des Prozesses hohe Sachwerte oder sogar Menschenleben abhängen.[a]

[a] Ein prominentes Beispiel finden Sie hier: `https://de.wikipedia.org/wiki/Ariane_V88`, siehe insbesondere `https://de.wikipedia.org/wiki/Ariane_V88#Fehlerursachen`.

H1.3: Attributwerte relativ zum momentanen Wert ändern — 4 Punkte

Hinweis:

Im folgenden Text finden Sie Ausdrücke wie `set*`. Der Asterisk (Sternchen, also „*“), dient in diesem Zusammenhang als ein Platzhalter für alle Methoden, deren Name mit `set` beginnt. Der Ausdruck `set*` steht hier konkret also für: `setX`, `setY`, `setDirection` und `setNumberOfCoins`. Ein solcher Ausdruck wird oft verwendet, wenn man nicht alle Möglichkeiten ausschreiben möchte. Eine weitere Möglichkeit, die Sie kennen sollten, nutzt eine Art Mengenschreibweise. Das würde in diesem Beispiel wie folgt aussehen: `set{X,Y,Direction,NumberOfCoins}`. Der Unterschied ist, dass die Mengenschreibweise abschließend ist, die Asterisk-Schreibweise jedoch nicht (es könnte neben diesen vier Methoden auch noch weitere geben, auf die das Namensschema `set*` passt).

Zum Setzen der vier Attribute von `offspring` würden wir eigentlich `set`-Methoden passend zu den `get`-Methoden aus H1.2 implementieren. Wir weichen hier aber leicht davon ab: Implementieren Sie vier `public`-Methoden `addToXOfOffspring`, `addToYOfOffspring`, `addToDirectionOfOffspring` und `addToNumberOfCoinsOfOffspring`. Jede dieser Methoden `addTo*OfOffspring` hat einen formalen Parameter vom Typ `int` (auch bei `addToDirectionOfOffspring`!), dessen Namen Sie beliebig wählen dürfen. Die Methoden liefern nichts zurück. Alle vier Methoden prüfen erst einmal, ob `offspring` mindestens einmal initialisiert wurde (verwenden Sie hierfür die Methode `offspringIsInitialized` aus H1.2). Falls dies nicht der Fall ist, soll die jeweilige Methode nichts weiter tun.

Verbindliche Anforderung:

Es darf keine `NullPointerException` auftreten, wenn eine `set`-Methode aufgerufen wird, bevor der `offspring` initialisiert wurde.

Hinweis:

Wir betrachten für die restliche Teilaufgabe (H1.3) den Fall, dass `offspring` initialisiert ist.

Für die Methoden `addToXOfOffspring` und `addToYOfOffspring` werden jetzt die beiden Attribute `numberOfColumnsOfWorld` und `numberOfRowsOfWorld` benötigt. Wie oben beschrieben, geht `RobotWithOffspring` davon aus, dass diese beiden Attribute tatsächlich die Anzahl der Spalten und Zeilen der `World` angeben. Sei n die Summe aus der momentanen Spaltenposition von `offspring` und dem aktualen Wert des Parameters von `addToXOfOffspring`. Falls n ein gültiger Spaltenindex der `World` ist, dann wird `offspring` mit `setX` in diese Spalte verschoben; andernfalls wird `offspring` in die Spalte der `World` verschoben, deren Spaltenindex am nächsten zu n ist (also 0, falls $n < 0$, bzw. der größte Spaltenindex, falls n größer als dieser ist). Die Werte der anderen Attribute von `offspring` werden in keinem dieser Fälle geändert. Die Methode `addToYOfOffspring` ist völlig analog zu `addToXOfOffspring` zu implementieren.

Die Methode `addToNumberOfCoinsOfOffspring` hat ebenfalls einen Parameter vom Typ `int`. Sei n die Summe aus der momentanen Anzahl Münzen von `offspring` und dem aktualen Wert dieses Parameters. Falls $n > 0$, wird die Anzahl Münzen von `offspring` mit `setNumberOfCoins` auf n gesetzt; andernfalls auf 0. Auch von `addToNumberOfCoinsOfOffspring` werden die Werte der anderen Attribute von `offspring` in keinem dieser beiden Fälle geändert.

Die Methode `addToDirectionOfOffspring` wendet modulare Arithmetik[3] für die vier Richtungen eines Roboters an, also die Art Arithmetik, die Sie z.B. von Stunden kennen. Die Stunden eines Tages werden mit $0, \ldots, 23$ durchnummeriert. Anstelle von 24 haben wir wieder die 0. Ebenso haben wir statt -1 wieder die 23 etc.

Man sagt allgemein, zwei Ganzzahlen m und n sind *kongruent* bezüglich einer positiven Ganzzahl k, wenn $|m-n|$ ohne Rest durch k teilbar ist. Dies schließt auch den Fall $|m-n| = 0$, also $m = n$, ein. Bei den Stunden eines Tages ist $k = 24$, bei den Richtungen eines Roboters ist $k = 4$. Hiermit lässt sich nun Arithmetik betreiben. Ein Beispiel auf den Stunden

[3] Für weitergehend Interessierte: `https://en.wikipedia.org/wiki/Modular_arithmetic`.

$0, \ldots, 23$ eines Tages: Sei jetzt 11:00 morgens, dann haben wir $1\,000$ Stunden später: $11 + 1\,000 = 1\,011 = 42 \cdot 24 + 3$, also 3:00 morgens (und zwar 41 Tage und 16 Stunden später).

Die vier Richtungen eines Roboters sollen im Uhrzeigersinn gesehen werden, beginnend mit `UP`. Das heißt, $n = 0$ steht für `UP`, $n = 1$ für `RIGHT`, $n = 2$ für `DOWN` und $n = 3$ für `LEFT`. Für jeden anderen Wert von n ist die Drehung dieselbe wie bei der Zahl aus $\{0, 1, 2, 3\}$, die zu n kongruent ist. Für $n \geq 0$ ist das gerade $n \,\%\, 4$ (hierbei handelt es sich um den modulo-Operator, vgl. Kapitel 01a, ab Folie 108 der FOP).

Für $n < 0$ ist die Logik offensichtlich diese: -1 ist kongruent zu 3, -2 zu 2, -3 zu 1, -4 zu 0, -5 zu 3 usw. Wir können nun den Fall $n < 0$ auf den positiven Fall $-n > 0$ zurückführen: Seien $k, \ell \in \{0, 1, 2, 3\}$, sodass k kongruent zu n und ℓ kongruent zu $-n > 0$ ist. Dann folgt aus den obigen Überlegungen offenbar $k + \ell = 4$ für $k \in \{1, 2, 3\}$ und $k + \ell = 0$ für $k = 0$. Beides lässt sich zusammenfassen: $(k + \ell) \,\%\, 4 = 0$. Sie können leicht nachprüfen, dass dies für jedes $k \in \{0, 1, 2, 3\}$ äquivalent ist zu $k = (4 - \ell) \,\%\, 4$, und dieses k ist ja der gesuchte Wert im Fall $n < 0$. Sie müssen dann nur noch für ℓ den Ausdruck $-n \,\%\, 4$ einsetzen. Damit erhalten Sie für negative n die Formel

$$k = (4 - ((-n) \,\%\, 4)) \,\%\, 4$$

, welche Sie in ihrem Programm verwenden können. Tabelle 1 veranschaulicht die Berechnungen.

n	k	$-n$	ℓ	$k + \ell$	$(k + \ell) \,\%\, 4$	$(4 - \ell) \,\%\, 4$	$(4 - ((-n) \,\%\, 4)) \,\%\, 4$
-1	3	1	1	4	0	3	3
-2	2	2	2	4	0	2	2
-3	1	3	3	4	0	1	1
-4	0	4	0	0	0	0	0
-5	3	5	1	4	0	3	3

Tabelle 1: Modulare Arithmetic der `Direction`

Sei nun $n \in \{0, 1, 2, 3\}$ die Zahl, die der momentanen Richtung von `offspring` entspricht, und m der aktuale Wert des Parameters von `addToDirectionOfOffspring`. Dann soll die Richtung von `offspring` durch `addToDirectionOfOffspring` entsprechend $m + n$ gemäß modularer Arithmetik gesetzt werden.

Hinweis:

Um an die Ganzzahl zu kommen, die einen `enum`-Wert repräsentiert, dürfen Sie die Funktion `ordinal()`[a] aus der Java-Standardbibliothek verwenden:

ordinal()

```
1 int number = Direction.DOWN.ordinal(); // number is 2
```

Um umgekehrt an den `enum`-Wert für eine Ganzzahl zu kommen, darf die `values()`-Funktion [b] verwendet werden, die ein Array zurückliefert, auf welches Sie mit der entsprechenden Ganzzahl zugreifen können.

values()

```
1 Direction direction = Direction.values()[2]; // direction is Direction.DOWN
```

[a] vgl. `java.lang.Enum#ordinal()`
[b] vgl. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.9.2`

Verbindliche Anforderung:

Funktionalität aus der Java-Standardbibliothek oder anderen Bibliotheken *für modulare Arithmetik* darf nicht verwendet werden.

Hinweis:

Die `direction` eines `Robot`-Objekts kann von außen nicht direkt gesetzt werden, da das Attribut mithilfe von `private` geschützt ist und keine `setDirection`-Methode existiert. Überlegen Sie, wie Sie dennoch Ihr Ziel erreichen können, dass der Roboter am Ende der Methode in die gewünschte Richtung zeigt. Ggf. hilft Ihnen Kapitel 01b, Folien 50-55 der FOP weiter.

In der realen Softwareentwicklung sollten Sie im Gegensatz zu hier ***immer*** schon vorhandene Funktionalität nutzen (sofern vertrauenswürdig) und **niemals** selbst implementieren. Denn die Wahrscheinlichkeit, Fehler einzubauen, ist schon bei einem so einfachen Beispiel wie hier sehr hoch bzw. in der Lebensspanne der Software kann es nachträglich irgendwann passieren, dass bei einer Revision des Softwarepakets Fehler unabsichtlich eingebaut werden.

Anmerkungen:

Bevor Sie mit H2 weitermachen, testen Sie wie üblich, ob sich `RobotWithOffspring` auch mit der zusätzlichen Funktionalität aus H1.3 weiterhin korrekt verhält. Verlassen Sie sich insbesondere nicht darauf, dass unsere obigen Überlegungen zur modularen Arithmetik und Ihre Implementierung unserer Überlegungen korrekt sind!

H2: Roboter mit überschriebenen Methoden — 6 Punkte

Sie schreiben nun eine `public`-Klasse `RobotWithOffspring2` in einer bereits erstellten Datei `RobotWithOffspring2.java` im Package `h03`. Leiten Sie die Klasse `RobotWithOffspring2` direkt von der Klasse `RobotWithOffspring` ab. Fügen Sie in `RobotWithOffspring2` ein weiteres `private`-Attribut ein, das es in `RobotWithOffspring` noch nicht gibt: `directionAccu` vom Typ `int` (`Accu` steht für *Accumulator*). Zu jedem Zeitpunkt während der Lebenszeit eines `RobotWithOffspring2`-Objektes enthält `directionAccu` die Summe der aktualen Parameter aller bis dahin getätigten Aufrufe von `addToDirectionOfOffspring` (*während* eines Aufrufs von `addToDirectionOfOffspring` gilt diese Anforderung natürlich nicht, nur *zwischen* zwei Aufrufen). Im Folgenden implementieren Sie die Logik hierfür.

Schreiben Sie auch für `RobotWithOffspring2` einen `public`-Konstruktor. Dieser soll dieselben formalen Parameter mit denselben Typen in derselben Reihenfolge wie `RobotWithOffspring` besitzen. Er ruft den Konstruktor von `RobotWithOffspring` mit seinen eigenen aktualen Parametern auf. Das neue Attribut `directionAccu` wird hier noch nicht initialisiert.

Analog zu `SlowMotionRobot` aus Kapitel 01f der FOP überschreiben Sie in `RobotWithOffspring2` die Methode `initOffspring`. Die Methode `initOffspring` von `RobotWithOffspring2` ruft die Methode `initOffspring` von `RobotWithOffspring` mit ihren eigenen aktualen Parametern auf (vgl. Kapitel 01f, ab Folie 45 der FOP). Dann initialisiert sie `directionAccu` entsprechend dem aktualen Wert des Parameters `direction` mit 0, 1, 2 oder 3.

Fügen Sie in die Klasse `RobotWithOffspring2` eine `private`-Methode `getDirectionFromAccu` ein. Diese hat keine formalen Parameter und den Rückgabetyp `Direction`. Die Methode wendet wie in H1.3 modulare Arithmetik an und liefert die Richtung zurück, die dem momentanen Wert von `directionAccu` entspricht.

Weiter überschreiben Sie die Methode `getDirectionOfOffspring` in `RobotWithOffspring2` so, dass sie einfach nur `getDirectionFromAccu` aufruft und die Rückgabe dieses Aufrufs selbst zurückliefert.

Überschreiben Sie schließlich die Methode `addToDirectionOfOffspring`. Diese prüft zunächst wie in H1.3, ob `offspring` mindestens einmal initialisiert wurde. Ist dies der Fall, addiert sie den aktualen Wert ihres Parameters auf `directionAccu` (das ist die Akkumulation im Namen `directionAccu`). Mittels `getDirectionFromAccu` berechnet sie dann noch die entsprechende neue Richtung und setzt die Richtung von `offspring` auf diese neue Richtung, damit die Richtung von `offspring` und `directionAccu` konsistent bleiben.

Unbewertete Verständnisfragen:

- Wir legen grundsätzlich Wert auf aussagekräftige Identifier, zum Beispiel `RobotWithOffspring` in Abgrenzung zu `Robot`. Warum sehen wir trotzdem kein Problem darin, die Klasse in H2 einfach nur `RobotWithOffspring2` zu nennen und die Unterschiede zu `RobotWithOffspring` nicht im Namen anzusprechen?
- Bei `RobotWithOffspring2` sehen wir ein Problem, das in der Informatik „unkontrollierte Redundanz“ genannt wird. Können Sie sich vorstellen, was damit konkret bei `RobotWithOffspring2` gemeint ist?
- Wir haben jetzt drei Klassen: `Robot`, `RobotWithOffspring` und `RobotWithOffspring2`. Welcher Konstruktor ruft nochmal welchen direkt auf? Es gibt in Java keine Möglichkeit, einen Konstruktor zu „überspringen“, also z. B. den Konstruktor von `Robot` direkt im Konstruktor von `RobotWithOffspring2` aufzurufen. Die Klasse `RobotWithOffspring` ist ein Beispiel dafür, dass das auch besser so ist. Sehen Sie, warum?

Fehlermeldungen besser verstehen:

- Versuchen Sie in `sandboxTests()` in der Datei `Main.java`, bei einem Objekt vom Typ `RobotWithOffspring2` auf das Attribut `directionAccu` einmal lesend und einmal schreibend zuzugreifen. Nun sollte `Main.java` nicht mehr durch den Compiler gehen, alle anderen Quelltextdateien hingegen schon. Warum?
- Analog: Versuchen Sie, auf einem Objekt vom Typ `RobotWithOffspring2` die Methode `getDirectionFromAccu` aufzurufen, z. B. `System.out.println(robot.getDirectionFromAccu())`. Der Kompilierversuch sollte wieder ähnlich hängen bleiben. Warum?
- Ändern Sie in `RobotWithOffspring` temporär, nur zum Zwecke dieser Verständnisfrage, das Zugriffsrecht von `initOffspring` von `public` in `private`. Nun sollte `RobotWithOffspring2.java` nicht mehr durch den Compiler gehen. Warum?
- Richten Sie Referenzen von `Robot`, `RobotWithOffspring` und `RobotWithOffspring2` ein, aber lassen Sie sie diesmal nicht auf Objekte verweisen, sondern weisen Sie ihnen mit „=“ den symbolischen Wert `null` zu. Ihr Programm sollte durch den Compiler gehen, aber beim ersten Aufruf einer Methode mit einer dieser Referenzen sollte der Prozess mit einer Fehlermeldung abbrechen. Verstehen Sie diese Fehlermeldung ungefähr?[a] (In späteren Kapiteln der FOP werden wir mehr zu `null` sehen.)

[a] Werfen Sie ggf. einen Blick in die Dokumentation: `java.lang.NullPointerException`

Exkurs (Arithmetischer Überlauf**):**

An dieser Stelle möchten wir noch einmal auf einen Stolperstein in der Softwareentwicklung eingehen, der auch in der Praxis gelegentlich übersehen wird und Probleme verursacht. In Kapitel 01b, Folien 139-142 der FOP wurde der Sachverhalt schon umrissen; hier wird er etwas näher erläutert.

Aus der Vorlesung (Kapitel 01b, ab Folie 134 der FOP) wissen Sie, dass der Datentyp `int` ganze Zahlen bis etwas über zwei Milliarden speichern kann. Nachdem Sie alle temporären Änderungen wieder rückgängig gemacht haben, richten Sie in `sandboxTests()` in der Datei `Main.java` eine `int`-Variable ein, weisen ihr „`3 * 1_000_000_000`“[a] zu und lassen sich das Ergebnis auf der Konsole wie üblich mit `System.out.println` ausgeben.

Nachdem Sie nun dieses kleine Experiment durchgeführt haben, überlegen Sie sich, welches Problem auftreten könnte, wenn ein `RobotWithOffspring2`-Objekt über einen längeren Zeitraum hinweg verwendet wird. Hat dies Auswirkungen auf das Verhalten des Roboters?

Ursache

Ein `int` wird in Java in 32 Bit (8 Byte) gespeichert. Das erste Bit gibt an, ob es sich um eine positive oder negative Zahl handelt: ist das Bit `0`, handelt es sich um eine positive Zahl, sonst um eine negative. Die restlichen Zahlen geben an, welche Zahl gespeichert wird. Beachten Sie hierzu folgende Tabelle:

Dezimalzahl	Binärrepräsentation
$-2\,147\,483\,648$	`1000_0000_0000_0000_0000_0000_0000_0000`
$-2\,147\,483\,647$	`1000_0000_0000_0000_0000_0000_0000_0001`
-1	`1111_1111_1111_1111_1111_1111_1111_1111`
0	`0000_0000_0000_0000_0000_0000_0000_0000`
1	`0000_0000_0000_0000_0000_0000_0000_0001`
$2\,147\,483\,646$	`0111_1111_1111_1111_1111_1111_1111_1110`
$2\,147\,483\,647$	`0111_1111_1111_1111_1111_1111_1111_1111`

Der maximale Betrag der negativen Zahlen ist hierbei um eins höher als der maximale Betrag der positiven Zahlen, da die 0 auch als positive Zahl mit einer `0` als erstes Bit gespeichert wird. Interessant ist nun die Binärrepräsentation der größten Dezimalzahl $2\,147\,483\,647$. Wird im Code zu dieser Zahl eins hinzuaddiert, so wird die *binäre* Zahl um eins erhöht. Aus `0111_1111_1111_1111_1111_1111_1111_1111` wird also `1000_0000_0000_0000_0000_0000_0000_0000`, da dies die nächsthöhere Zahl in der Binärrepräsentation ist. Diese Zahl repräsentiert jedoch die Dezimalzahl $-2\,147\,483\,648$, somit wird aus der größten positiven Zahl die größte negative, obwohl die Zahl um eins erhöht werden sollte. Dieses Phänomen wird arithmetischer Überlauf genannt und kann unerwartetes Verhalten im Programm verursachen.

Falls Ihnen das Ganze trivial erscheinen sollte: Solche Überläufe werden auch in der Praxis übersehen – selbst in sicherheitsrelevanten Bereichen.[b]

[a] Die Unterstriche in der Zahl dienen nur der Lesbarkeit und werden vom Compiler einfach ignoriert, als wären sie nicht da. Sie können sie genau so in Ihren Java-Code einfügen. Vgl. `https://docs.oracle.com/javaswikie/7/docs/technotes/guides/language/underscores-literals.html`

[b] Vgl. `https://www.i-programmer.info/news/149-security/8548-reboot-your-dreamliner-every-248-days-to-avoid-integer-overflow.html`

H3: Klasse *mit* Robotern und Tests — 7 Punkte

H3.1: Klasse *mit* Robotern — 4 Punkte

Um unsere beiden Klassen `RobotWithOffspring` und `RobotWithOffspring2` zu testen, implementieren wir zunächst eine weitere Klasse `TwinRobots` in einer bereits erstellten Datei `TwinRobots.java` im Package `h03`.

Fügen Sie in die Klasse `TwinRobots` ein `private`-Attribute namens `robots` ein. Dieses ist vom Typ „Array von `RobotWithOffspring`“.

Fügen Sie einen `public`-Konstruktor in `TwinRobots` ein. Dieser hat zwei formale Parameter in dieser Reihenfolge: `numberOfColumnsOfWorld` und `numberOfRowsOfWorld` jeweils vom Typ `int`. Der Konstruktor weist dem Attribut `robots` ein Array vom Typ `RobotWithOffspring` der Größe `2` zu. Anschließend wird im Index `0` dieses Arrays ein neues Objekt vom Typ `RobotWithOffspring` und im Index `1` ein neues Objekt vom Typ `RobotWithOffspring2` gespeichert[4]. Beide Roboter bekommen jeweils die Spalten- und Zeilenanzahl der Welt aus den aktualen Parametern des Konstruktors von `TwinRobots` weitergereicht. Das `RobotWithOffspring`-Objekt in Index `0` bekommt hierbei die Direction `RIGHT`, das `RobotWithOffspring2`-Objekt in Index `1` die Direction `UP`. Keiner der beiden Roboter besitzt Münzen. Schließlich wird auf beiden Robotern der Abkömmling mittels `initOffspring` initialisiert, wobei dieser bei beiden Robotern nach links zeigt und keine Münzen besitzt.

Fügen Sie in die Klasse `TwinRobots` eine `public`-Methode `getRobotByIndex` ein. Diese hat einen formalen Parameter vom Typ `int` und den Rückgabetyp `RobotWithOffspring`. Die Methode verwendet ihren aktualen Parameter, um das Roboterobjekt zurückzuliefern, das am entsprechenden Index im `robots`-Array gespeichert ist. Sie können davon ausgehen, dass die Methode immer mit einem gültigen Index aufgerufen wird.

Fügen Sie in die Klasse `TwinRobots` eine `public`-Methode `addToDirectionOfBothOffsprings` ein. Diese hat einen formalen Parameter vom Typ `int` und keinen Rückgabetyp. Die Methode ruft auf *beiden* `RobotWithOffspring`-Objekten im Array `robots` die Methode `addToDirectionOfOffspring` mit ihrem aktualen Parameter auf.

H3.2: Testen — 3 Punkte

Sie haben bisher `RobotWithOffspring` und `RobotWithOffspring2` schon ausgiebig getestet und können davon ausgehen, dass beide Klassen korrekt funktionieren – mit einem niemals vermeidbaren kleinen Restrisiko, dass Ihre Tests und Ihre Durchsicht des Codes nicht alle Fehler aufgedeckt haben.

Sie werden nun – wie oben bereits angedeutet – beide Klassen noch einmal gegeneinander testen, indem Sie mithilfe der Klasse `TwinRobots` „Zwillingspaare“ bestehend aus je einem Roboter vom Typ `RobotWithOffspring` und einem Roboter vom Typ `RobotWithOffspring2` völlig parallel dasselbe machen lassen und prüfen, ob die Methode `getDirectionOfOffspring` für beide Zwillinge eines Paares zu jedem Zeitpunkt denselben Wert zurückliefert.

Hierfür nutzen Sie eine weitverbreitete Technik, sogenannte JUnit-Tests, welche wir uns in Kapitel 05, ab Folie 132 der FOP noch genauer ansehen werden. Dafür müssen Sie zunächst in der Methode `testRobotWithOffspringTwins()` in der Datei `H3_2_UnitTest.java` im Ordner `src\test\java\h03` die auskommentierten Zeilen wieder einkommentieren. Diese sollten ohne Probleme durch den Compiler gehen. Sie können diesen JUnit-Test laufen lassen und er sollte auch erfolgreich durchlaufen.

Fügen Sie hier noch weitere Fälle ein. Sie müssen also die Richtungen beider Roboter mittels der Methode `addToDirectionOfBothOffsprings` identisch ändern und dann mittels `assertEquals` prüfen, ob

[4] Da der Array Objekte vom Typ `RobotWithOffspring` speichert und die Klasse `RobotWithOffspring2` von der Klasse `RobotWithOffspring` erbt, ist das ohne Probleme möglich.

`getDirectionOfOffspring` beider Roboter auch – wie erwartet – dasselbe zurückliefert. Zugriff auf die Roboter haben Sie mittels der Methode `getRobotByIndex`. Orientieren Sie sich hierzu am bereits gegebenen Code.

Verbindliche Anforderungen:

Fügen Sie mindestens 3 weitere Fälle ein, wobei mindestens einmal in diesen 3 zusätzlichen Fällen ein negativer Parameter beim Aufruf der Methode `addToDirectionOfBothOffsprings` verwendet werden muss. Außerdem muss in mindestens einem dieser 3 zusätzlichen Fälle der Wert des `directionAccu`-Attributs von `RobotWithOffspring2` negativ sein.