Prof. Dr. J. Giesl
J.-C. Kassing, M. Leven Rosarius, N. Lommen, E. Meyer

Allgemeine Hinweise:

- Die **Deadline** zur **Abgabe** der Hausaufgaben ist am **Donnerstag, den 13.11.2025, um 14 Uhr**.
- Der **Workflow** sieht wie folgt aus. Die Abgabe der Hausaufgaben erfolgt **im Moodle-Lernraum** und kann nur in **Zweiergruppen** stattfinden. Dabei müssen die Abgabepartner*innen **dasselbe Tutorium** besuchen. Nutzen Sie ggf. das entsprechende **Forum** im Moodle-Lernraum, um eine*n Abgabepartner*in zu finden. Es darf **nur ein*e** Abgabepartner*in die Abgabe hochladen. Diese*r muss sowohl die **Lösung** als auch den **Quellcode** der Programmieraufgaben hochladen. Die Bepunktung wird dann von uns für **beide** Abgabepartner*innen **separat** im Lernraum eingetragen. Die Feedbackdatei ist jedoch nur dort sichtbar, wo die Abgabe hochgeladen wurde und muss innerhalb des Abgabepaars **weitergeleitet** werden.
- Die **Lösung** muss als PDF-Datei hochgeladen werden. Damit die Punkte beiden Abgabepartner*innen zugeordnet werden können, müssen **oben** auf der **ersten Seite** Ihrer Lösung die **Namen**, die **Matrikelnummern** sowie die **Nummer des Tutoriums** von **beiden** Abgabepartner*innen angegeben sein.
- Der **Quellcode** der Programmieraufgaben muss als `.zip`-Datei hochgeladen werden und **zusätzlich** in der PDF-Datei mit Ihrer Lösung enthalten sein, sodass unsere Hiwis ihn mit Feedback versehen können. Auf diesem Blatt muss Ihre Codeabgabe Ihren vollständigen Java-Code in Form von `.java`-Dateien enthalten. Aus dem Lernraum heruntergeladene Klassen dürfen nicht mit abgegeben werden.

  Stellen Sie sicher, dass Ihr Programm von `javac` **in der Version 25 akzeptiert** wird. Generell sollten alle Programme für alle Eingaben terminieren, solange in der Spezifikation (bzw. der Aufgabenstellung) nicht explizit etwas anderes verlangt wird!
- Einige Hausaufgaben müssen im Spiel Codescape gelöst werden. Klicken Sie dazu im Lernraum rechts im Block "Codescape" auf den angegebenen Link. Diese Aufgaben werden getrennt von den anderen Hausaufgaben gewertet.

Übungsaufgabe 1 (Überblickswissen):

a) Was ist der grundlegende Unterschied zwischen einer Klasse und einem Objekt?

b) Variablen können in einer Klasse, aber auch in einer Methode deklariert werden. Worin besteht der Unterschied und in welchen Fällen deklariert man Variablen typischerweise an welcher Stelle?

c) Was ist der Unterschied zwischen *Call By Reference* und *Call By Value*? Inwieweit unterstützt Java die beiden Konzepte?

d) Welche Methoden kann man nicht nur auf Objekten, sondern auch auf Klassen aufrufen? Wann ergibt das Sinn?

Übungsaufgabe 2 (Verifikation):

Gegeben sei folgendes Java-Programm P über der `int[]`-Variable `a` sowie den `int`-Variablen `i` und `res`:

$\langle$ `a.length` > 0 $\rangle$ (Vorbedingung)

```
res = a[0];
i = 1;
while (i < a.length) {
    if (a[i] > res) {
        res = a[i];
    }
    i = i + 1;
}
```

$\langle$ `res` $= \max\{$ `a[`j`]` $\mid 0 \leq j <$ `a.length` $\}$ $\rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation der partiellen Korrektheit des Algorithmus im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Geben Sie bei der Anwendung der Bedingungsregel (zur Behandlung der `if`-Anweisung) an, welche Formel der Art "$\varphi \wedge \neg B \Rightarrow \psi$" hierbei gezeigt werden muss.

Hinweise:

- Gehen Sie wieder bei allen Aufgaben zum Hoare-Kalkül davon aus, dass keine Integer-Überläufe stattfinden, d.h., behandeln Sie Integers als die unendliche Menge $\mathbb{Z}$.
- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x + 1 = y + 1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen.
- Es empfiehlt sich oft, bei der Erstellung der Zusicherungen in der Schleife von unten (d. h. von der Nachbedingung aus) vorzugehen.
- Die Implikationen bei Konsequenzregeln müssen nicht separat bewiesen werden.

$\langle$`a.length` $> 0\rangle$

$\langle$_____$\rangle$
```
res = a[0];
```
$\langle$_____$\rangle$
```
i = 1;
```
$\langle$_____$\rangle$

$\langle$_____$\rangle$
```
while (i < a.length) {
```
$\langle$_____$\rangle$

$\langle$_____$\rangle$
```
    if (a[i] > res) {
```
$\langle$_____$\rangle$

$\langle$_____$\rangle$
```
        res = a[i];
```
$\langle$_____$\rangle$
```
    }
```
$\langle$_____$\rangle$
```
    i = i + 1;
```
$\langle$_____$\rangle$
```
}
```
$\langle$_____$\rangle$

$\langle$`res` $= \max\{$ `a[`j`]` $\mid 0 \leq j <$ `a.length` $\}\rangle$

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und mit Hilfe des Hoare-Kalküls die Terminierung unter der Voraussetzung `a.length` > 0 bewiesen werden. Geben Sie auch hier die Formel an, die bei der Anwendung der Bedingungsregel (zur Behandlung der `if`-Anweisung) gezeigt werden muss.

Hausaufgabe 3 (Verifikation): (15 Punkte)

Gegeben sei folgendes Java-Programm P über der `int[]`-Variable `a` sowie den `int`-Variablen `i` und `s` und der `boolean`-Variable `res`:

$\langle$ `a.length` > 0 $\rangle$ (Vorbedingung)

```
i = 1;
res = false;
s = a[0];
while (i < a.length) {
    if (s == a[i]) {
        res = true;
    }
    s = s + a[i];
    i = i + 1;
}
```

$\langle$ `res` $= \exists j$ mit $1 \leq j <$ `a.length`. `a`$[j] = \sum_{k=0}^{j-1}$ `a`$[k]$ $\rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation der partiellen Korrektheit des Algorithmus im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Geben Sie bei der Anwendung der Bedingungsregel (zur Behandlung der `if`-Anweisung) an, welche Formel der Art "$\varphi \wedge \neg B \Rightarrow \psi$" hierbei gezeigt werden muss.

Hinweise:

- Die Nachbedingung besagt, dass `res` genau dann `true` ist, wenn es einen Eintrag `a`$[j]$ mit $j \geq 1$ im Array gibt, sodass `a`$[j] =$ `a`$[0] +$ `a`$[1] + \cdots +$ `a`$[j-1]$ gilt.
- Um Platz zu sparen, können Sie statt `a.length` nur `n` schreiben.
- Gehen Sie wieder bei allen Aufgaben zum Hoare-Kalkül davon aus, dass keine Integer-Überläufe stattfinden, d.h., behandeln Sie Integers als die unendliche Menge $\mathbb{Z}$.
- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x + 1 = y + 1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen.
- Es empfiehlt sich oft, bei der Erstellung der Zusicherungen in der Schleife von unten (d. h. von der Nachbedingung aus) vorzugehen.
- Die Implikationen bei Konsequenzregeln müssen nicht separat bewiesen werden.

$\langle \mathtt{n} > 0 \rangle$

$\langle \_\_\_\_\_ \rangle$
```
i = 1;
```
$\langle \_\_\_\_\_ \rangle$
```
res = false;
```
$\langle \_\_\_\_\_ \rangle$
```
s = a[0];
```
$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$
```
while (i < a.length) {
```
$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$
```
    if (s == a[i]) {
```
$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$
```
        res = true;
```
$\langle \_\_\_\_\_ \rangle$
```
    }
```
$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$
```
    s = s + a[i];
```
$\langle \_\_\_\_\_ \rangle$
```
    i = i + 1;
```
$\langle \_\_\_\_\_ \rangle$
```
}
```
$\langle \_\_\_\_\_ \rangle$

$\langle \mathtt{res} = \exists j \text{ mit } 1 \leq j < \mathtt{n}.\ \mathtt{a}[j] = \sum_{k=0}^{j-1} \mathtt{a}[k] \rangle$

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und mit Hilfe des Hoare-Kalküls die Terminierung unter der Voraussetzung `a.length` > 0 bewiesen werden. Geben Sie auch hier die Formel an, die bei der Anwendung der Bedingungsregel (zur Behandlung der `if`-Anweisung) gezeigt werden muss.

In den Aufgaben 4 und 5 sollen Sie Speicherzustände zeichnen. Angenommen wir haben folgenden Java Code:

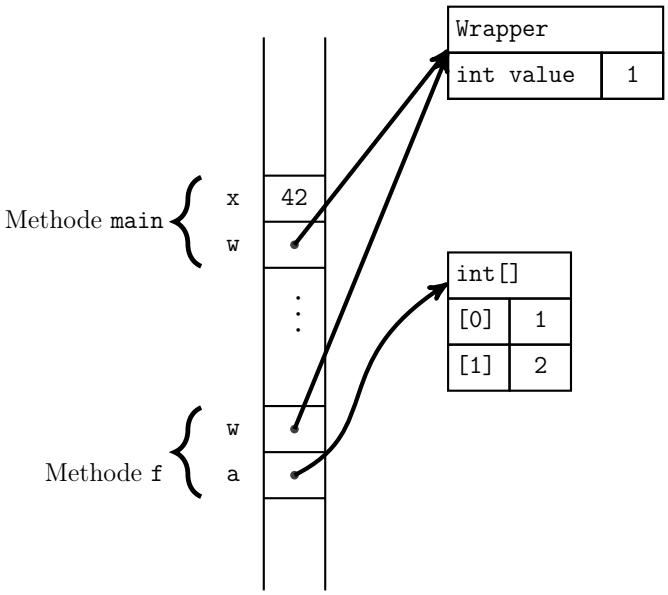
```
public class Wrapper {
      int value;
}
```

```
public class Main {
      public static void main() {
            int x = 42;
            Wrapper w = new Wrapper();
            w.value = 0;
            f(w);
      }

      public static void f(Wrapper w) {
            int[] a = {1,2};
            w.value = 1;

            // Speicherzustand hier gezeichnet
      }
}
```

Dann sieht der Speicher an der markierten Stelle wie folgt aus:



Übungsaufgabe 4 (Seiteneffekte):

Betrachten Sie das folgende Programm:

```
public class TSeiteneffekte {
    public static void main() {
        TWrapper[] ws = new TWrapper[2];
        ws[0] = new TWrapper();
        ws[1] = new TWrapper();

        ws[0].value = 2;
        ws[1].value = 1;

        f(ws[1], new TWrapper[] { ws[1], ws[0] });

        // Speicherzustand hier zeichnen
    }

    public static void f(TWrapper w1, TWrapper[] ws) {
        int sum = 0;

        // Speicherzustand hier zeichnen

        for (int j = 0; j < ws.length; j++) {
            TWrapper w = ws[j];
            sum += w.value;
            w.value = j + 2;
        }

        // Speicherzustand hier zeichnen

        w1 = ws[1];
        w1.value = -sum;
    }
}

public class TWrapper {
    int value;
}
```

Es wird nun die Methode `main` ausgeführt. Stellen Sie den Speicher an allen drei markierten Programmzuständen graphisch dar. Achten Sie darauf, dass Sie alle (implizit) im Programm vorkommenden Arrays und alle Objekte sowie die zu dem Zeitpunkt existierenden Programmvariablen darstellen.

Hausaufgabe 5 (Seiteneffekte): (18 Punkte)

Betrachten Sie das folgende Programm:

```
public class HSeiteneffekte {
        public static void main() {
                HWrapper w1 = new HWrapper();
                HWrapper w2 = new HWrapper();

                w1.i = 0;
                w2.i = 3;

                int[] a = { 1, 2 };

                f(a, w1);
                int[] b = {a[0] + a[1],a[0] * a[1]};
                f(b, w1, w2);
                f(a);
        }

        public static void f(int[] a, HWrapper... ws) {
                if(ws.length == 0) {
                        a = new int[2];
                        a[0] = 3;
                        a[1] = 2 * a[0];
                } else {
                        a[1] += a[0];
                        ws[ws.length-1].i = a[0];
                        ws[0].i += ws[ws.length-1].i;
                }
                //Speicherzustand jeweils hier zeichnen
        }
}

public class HWrapper {

    int i;

}
```

Es wird nun die Methode `main` ausgeführt. Stellen Sie den Speicher (d.h. alle (implizit) im Programm vorkommenden Arrays und alle Objekte sowie die zu dem Zeitpunkt existierenden Programmvariablen) am Ende jeder Ausführung der Methode `f` graphisch dar. Insgesamt sind also drei Speicherzustände zu zeichnen.

Übungsaufgabe 6 (Programmierung):

Aus dem Serious Game Codescape kennen Sie bereits den robotischen Begleiter, kurz RB. In dieser Aufgabe werden Sie den Codeinterpreter eines frühen Prototyps vom RB nachbauen. Dieser Prototyp versteht noch kein Java, sondern arbeitet mit einer stark vereinfachten Programmiersprache. Die Syntax der Programmiersprache ist durch folgende Grammatik in EBNF gegeben:

```
P = (("move" | "turnLeft" | "turnRight" | "pickUp") "\n"
     | "for " I " " I "\n" P "endfor " I "\n" | P P)
```

Hierbei steht `"\n"` für einen Zeilenumbruch und das Nichtterminal `I` kann durch eine beliebige natürliche Zahl ersetzt werden.
Zur Erklärung der Semantik betrachten Sie das folgende Beispiel:

```
0 for 0 2
1 turnLeft
2 for 1 3
3 move
4 endfor 2
5 turnRight
6 endfor 0
7 move
```

```
0 for (int r0 = 0; r0 < 2; ++r0) {
1   turnLeft();
2   for (int r1 = 0; r1 < 3; ++r1) {
3     move();
4   }
5   turnRight();
6 }
7 move();
```

Das Beispielprogramm auf der linken Seite hat die gleiche Semantik wie das Java-Programm auf der rechten Seite. Die Semantik von `move`, `turnLeft`, `turnRight` und `pickUp` ist, dass der RB den entsprechenden Bewegungsbefehl ausführt. Für Schleifen hat der RB eine feste Anzahl von Registern, in denen die Werte der Laufvariablen abgelegt werden. Die erste Zahl hinter einem `for` gibt das Register an, in dem die Laufvariable abgelegt wird, die zweite Zahl gibt die Schranke an, bis zu der iteriert werden soll. Die Zahl hinter einem `endfor` gibt die Zeile an, in die zurück gesprungen werden soll (also die Zeile, in der das zugehörige `for` steht). Dabei beginnt die Nummerierung der Zeilen bei `0`. Wir fordern von einem korrekten Programm, dass zu jedem `for` genau das `endfor`, das gleichzeitig aus der Grammatik erzeugt wurde, auf die Zeile dieses `for` verweist. Außerdem fordern wir, dass bei geschachtelten `for`-Schleifen unterschiedliche Register als Laufvariablen verwendet werden.

Wir verwenden für die Implementierung die Klassen `RB` und `Room` (inkl. zwei internen Klassen). Die Klasse `Room` ist als compilierte `.class` Datei verfügbar, ebenso die beiden internen Klassen `vec2d` und `InternalRB`, die innerhalb von `Room` deklariert werden. Des Weiteren steht eine Datei `RBMain` zur Verfügung, welche eine `main`-Methode enthält. Alle diese Dateien müssen im gleichen Ordner wie der restliche Code abgelegt werden. Ein `RB` hat ein gespeichertes Programm beliebiger Länge, eine maximale Anzahl von Kommandos, den Index der nächsten auszuführenden Programmzeile und eine variable Anzahl von Registern für die Laufvariablen von Schleifen, die jedes einen `int` speichern können. Außerdem hat ein `RB` zwei Methoden. Die Methode `programmHochladen` unterzieht das übergebene Programm einer einfachen Syntax-Prüfung und speichert es anschließend im Programmspeicher des `RB`. Die Methode `schritt` führt das gespeicherte Programm vom aktuellen Index ausgehend so weit aus, bis ein Bewegungsbefehl (`move`, `turnLeft`, `turnRight`, `pickUp`) erreicht wird. Dieser Bewegungsbefehl wird dann von der Methode zurückgegeben.

a) Schreiben Sie das Grundgerüst für die Klasse `RB`. Die Objekte dieser Klasse sollten vier Attribute besitzen:

- Den Programmspeicher, ein Array von Strings. Jeder String repräsentiert eine Zeile im Programm des `RB`.
- Die maximale zulässige Anzahl von Kommandos. Als "Kommando" bezeichnen wir hier die Bewegungsbefehle (`move`, `turnLeft`, `turnRight`, `pickUp`).
- Die aktuelle Position im Programm. Diese Variable sollte als Index für den Programmspeicher verwendet werden können.
- Ein Array von `int`-Werten, als die Register des `RB`.

Außerdem enthält das Grundgerüst zwei Methoden:

- Die Methode `programmHochladen`, die ein Programm (d.h. ein Array von Strings) übergeben bekommt und einen `int` zurück gibt. Im Grundgerüst sollte diese Methode immer `0` zurück geben und sonst nichts tun.
- Die Methode `schritt` ohne Parameter, die einen String zurück liefert. Im Grundgerüst gibt diese Methode immer den String `"end"` zurück.

Dieses Grundgerüst sollte von **javac** akzeptiert werden.

Hinweise:

- Beachten Sie, dass Dateiname und Klassenname in `Java` übereinstimmen müssen!

b) Vervollständigen Sie die Datei `RBMain` indem Sie den mit `TODO` markierten Block anpassen. An dieser Stelle sollte ein neues Objekt vom Typ `RB` erzeugt werden und in einer Variable `myRB` abgespeichert werden. Anschließend setzen Sie die maximale Anzahl Kommandos auf `3` und laden eines der vorgegebenen Programme in den `RB` hoch. Das Ergebnis der Überprüfung beim Hochladen speichern Sie in der Variablen `checkResult`.

Nun sollte auch die Datei `RBMain.java` von **javac** akzeptiert werden. Wenn Sie das Programm anschließend mit **java RBMain** ausführen, sollte es ohne Fehler ablaufen. Der `RB` bleibt jedoch neben dem Eingang stehen und meldet, dass das Programmende erreicht wurde.

c) Implementieren Sie nun die Methode `programmHochladen`. Sie können davon ausgehen, dass jede Zeile des Programms in einem eigenen Array-Eintrag steht.

Zunächst initialisiert die Methode die Register, sodass zwei `int`-Werte gespeichert werden können und setzt die Programmposition auf `0`.

Anschließend wird das übergebene Programm mit einer **for each** Schleife durchlaufen. Dabei soll geprüft werden, dass das *erste* Wort in jeder Zeile nach der Grammatik zulässig ist (d.h. das erste Wort muss `"move"`, ..., `"for"` oder `"endfor"` sein). Außerdem soll geprüft werden, dass in jeder Zeile die richtige Anzahl Terminalsymbole steht. Beachten Sie dabei, dass `\n` nicht explizit repräsentiert wird und beliebig große natürliche Zahlen als ein Terminalsymbol gelten. Wenn das erste Wort z.B. `"move"` ist, dann darf in der Zeile nur genau ein Terminalsymbol, nämlich `"move"` selbst, stehen. Wenn bei dieser Überprüfung ein Syntaxfehler gefunden wird, dann soll die Methode `-1` zurückgeben.

Zählen Sie außerdem, wie oft die Kommandos `move`, `turnLeft`, `turnRight` und `pickUp` im Programm auftreten. Sollte diese Zahl die maximale Anzahl zulässiger Kommandos überschreiten, dann soll die Methode `-2` zurück geben.

Wenn bei der Überprüfung kein Fehler aufgetreten ist, dann wird das übergebene Programm im Programmspeicher des `RB` abgelegt und die Methode gibt die Anzahl der Kommandos (d.h., der Bewegungsbefehle) im Programm zurück.

Wenn Sie das Programm nun neu compilieren und ausführen, sollte das Ergebnis das Gleiche sein wie beim vorherigen Aufgabenteil.

Hinweise:

- Mit der Methode `s.split(" ")` können Sie einen String `s` an Leerzeichen in ein Array aufteilen. Der String `"for 1 15"` wird z.B. in `{"for", "1", "15"}` aufgeteilt.
- Sie können `switch` auch mit Strings als Argument verwenden.
- Nutzen Sie die Methode `s.equals(andererString)`, wenn Sie Strings ohne ein `switch`-Statement vergleichen wollen!
- Die Syntax-Überprüfung in dieser Methode ist unvollständig. Es werden also manche syntaktisch falsche Programme akzeptiert, aber kein syntaktisch korrektes Programm mit dem Rückgabewert `-1` abgelehnt.

d) In dieser Teilaufgabe implementieren Sie die Methode `schritt`.

In der Methode läuft eine Schleife ab, bis das nächste auszuführende Kommando (d.h., der nächste auszuführende Bewegungsbefehl) gefunden wurde. In der Schleife wird zunächst geprüft, ob die aktuelle Programmposition noch im Programm ist. Wenn das Programmende erreicht wurde, dann wird sofort `end` zurückgegeben. Ansonsten wird der String an der aktuellen Programmposition aus dem Programmspeicher gelesen. Ist es ein Kommando, wird die Programmposition um eins erhöht und das Kommando zurückgegeben. Ist es ein `endfor`, wird die Programmposition auf die Zahl, die dem `endfor` folgt, gesetzt. Bei einem `for` wird das Register gelesen, das durch die erste Zahl hinter dem `for` gegeben ist. Anschließend wird der Wert im Register mit der zweiten Zahl hinter dem `for` verglichen. Ist der Wert im Register kleiner, werden Programmposition und Wert im Register um eins erhöht. Andernfalls wird die Programmposition so lange erhöht, bis sie eins hinter dem nächsten `endfor` steht, bei dem als Zahl die Programmposition der `for`-Anweisung gegeben ist, die gerade ausgeführt wurde. Außerdem wird das Register zurück auf `0` gesetzt.

Wenn Sie das Programm nach dieser Teilaufgabe compilieren, sollte sich der `RB` durch den Raum bewegen. Wenn Sie `program` hochladen, dann läuft er in der oberen Zeile auf und ab, bis er an der Anfangsposition stehen bleibt. Wenn Sie `program2` hochladen, sollte er die Energiezelle aufsammeln und zum Ausgang gehen. Wenn Sie dieses Ergebnis nicht erreichen, prüfen Sie auch den Code der vorherigen Teilaufgabe!

Hinweise:

- Beachten Sie die Hinweise zum vorherigen Aufgabenteil (d.h. Sie sollten wieder `split`, `equals` und `switch` auf Strings anwenden).
- Mit der Methode `Integer.parseInt(someString)` können Sie einen String, der nur eine Zahl enthält, in einen `int` umwandeln, d.h. `Integer.parseInt("123")` ergibt die `int`-Zahl `123`.
- Sie können davon ausgehen, dass der Programmspeicher ein korrektes, gültiges Programm enthält. Insbesondere brauchen Sie nicht zu prüfen, dass die Strings hinter `for` oder `endfor` gültige Registerpositionen bzw. Programmpositionen sind.

Hausaufgabe 7 (Programmierung): (17 Punkte)

Heutzutage werden sehr viele Daten gesammelt. Um einen Überblick über die Daten zu erhalten, ist es sinnvoll, sich diese automatisch zusammenfassen zu lassen.

a) Implementieren Sie die Klasse `Statistics` im Code-Fragment unten. Ein Objekt der Klasse `Statistics` kann bis zu 100 `int`-Werte zusammenfassen. Die Klasse `Statistics` soll folgende Methoden bereitstellen:

- `public void addValues(int... values) { ... }`
  Fügt eine Folge von Werten zu den bisherigen Werten im aktuellen `Statistics`-Objekt hinzu.
- `public double getAverage() { ... }`
  Gibt den durchschnittlichen Wert aller bisher hinzugefügten Werte zurück.
- `public Statistics generate(int min, int max, int size) { ... }`
  Generiert zufällig `size` viele Werte im Intervall von (jeweils inklusive) `min` bis `max` und fügt die Werte zu einem neuen `Statistics`-Objekt hinzu. Dieses neue `Statistics`-Objekt wird als Ergebnis zurückgegeben.

Implementieren Sie diese drei Methoden. Überlegen und begründen Sie, welche dieser Methoden statisch sein sollen und fügen Sie ggf. das Schlüsselwort `static` hinzu. Sie können beliebige Attribute und Hilfsmethoden einfügen. Beachten Sie außerdem folgende Randfälle:

- Wird `getAverage` aufgerufen, ohne dass zuvor Werte hinzugefügt wurden, so wird eine Fehlermeldung ausgegeben und der Wert `0` zurückgegeben.
- Wird `addValues` mit mehr Werten aufgerufen, als es noch Platz im aktuellen `Statistics`-Objekt gibt, so werden solange Werte hinzugefügt, bis das Objekt 100 Werte "enthält". Anschließend wird eine (oder mehrere) Fehlermeldung(en) ausgegeben und die weiteren Werte aus `values` werden nicht mehr hinzugefügt.
- Wird `generate` aufgerufen, sodass `size` negativ oder größer als 100 ist, oder `max` kleiner als `min` ist, dann sollen jeweils entsprechende Fehlermeldungen ausgegeben werden (und aber dennoch ein neues `Statistics`-Objekt zurückgegeben werden).

Hinweise:

- Verwenden Sie die angegebene `main`-Methode zum Testen Ihrer Implementierung.
- Bei der Berechnung des Durchschnitts dürfen Sie Variablenüberläufe ignorieren.
- Um Zufallszahlen aus $\{0, \ldots, \texttt{n}\}$ zu generieren, erzeugen Sie zuerst ein Objekt `rand` mit `Random rand = new Random();`. Nun liefert `rand.nextInt(n + 1)` eine Zufallszahl aus der geforderten Menge. Damit die Klasse `Random` verfügbar ist, importieren Sie diese mit `import java.util.Random;`.

Beim Ausführen der `main`-Methode soll Folgendes ausgegeben werden:

```
Durchschnitt: -40.8
```

b) Damit unsere Daten auch für "Laien" interpretierbar sind, ordnen wir den Daten die Farben `GRUEN`, `GELB` und `ROT` zu. Schreiben Sie ein Enum `OurColor` für diese Farben und schreiben Sie in der Klasse `Statistics` die Methode `public OurColor interpret(double ratio, int value) { ... }`. Wenn d der Durchschnitt der Werte des `Statistics`-Objekts ist, auf dem `interpret` aufgerufen wurde, dann gibt `interpret` für den Wert `value` die Farbe `GELB` zurück, falls `value` im Intervall von $d - \texttt{ratio} \cdot |d|$ bis $d + \texttt{ratio} \cdot |d|$ liegt. Hierbei nehmen wir an, dass `ratio` eine Gleitkommazahl zwischen 0 und 1 ist. Ansonsten kann sich Ihre Methode beliebig verhalten. Bei Werten, die kleiner sind als $d - \texttt{ratio} \cdot |d|$, wird die Farbe `ROT` zurückgegeben. Hingegen wird bei Werten, die größer sind als $d + \texttt{ratio} \cdot |d|$, die Farbe `GRUEN` zurückgegeben.

Hinweise:

- Sie können die Methode `Math.abs(double x)` verwenden, um den Absolutbetrag einer `double`-Zahl `x` zu berechnen.

```
import java.util.Random;

public class Statistics {

    // ...

    public static void main() {
        Statistics statistics = new Statistics();
        statistics.addValues(2,105,-366,44,11);
        IO.println("Durchschnitt: " + statistics.getAverage());
    }

    public void addValues(int... values) {
        // ...
    }

    public double getAverage() {
        // ...
    }

    public Statistics generate(int min, int max, int size) {
        // ...
    }

    public OurColor interpret(double ratio, int value) {
        // ...
    }
}
```

Hausaufgabe 8 (Deck 4): (Codescape)

Lösen Sie die Missionen von Deck 4 des Codescape Spiels. Ihre Lösung für die Codescape Missionen wird nur dann für die Zulassung gezählt, wenn sie Ihre Lösung vor der einheitlichen Codescape Deadline am Freitag, den 30.01.2026, um 23:59 Uhr abschicken.