Prof. Dr. J. Giesl
M. Hark, J. Hensel, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Dienstag, den 14.11.2017 um 08:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor oder unmittelbar vor Beginn in der Globalübung abgeben.
- In einigen Aufgaben müssen Sie in Java programmieren und `.java`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Dienstag, dem 14.11.2017 um 08:00 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Benutzen Sie in ihrem Code keine Umlaute, auch nicht in Strings und Kommentaren. Diese führen oft zu Problemen, da diese Zeichen auf verschiedenen Betriebssystemen unterschiedlich behandelt werden. Dadurch kann es dazu führen, dass Ihr Programm bei Ihrer Tutorin/Ihrem Tutor bei Verwendung von Umlauten nicht mehr von `javac` akzeptiert wird.
- Für einige Programmieraufgaben benötigen Sie die Java Klasse `SimpleIO`. Diese können Sie auf der Website herunterladen.
- Halten Sie sich beim Lösen von Programmieraufgaben an die auf der Website zur Vorlesung verfügbaren Codekonventionen. Verstöße, die zu unleserlichem Code führen, können zu Punktabzug führen.
- Einige Hausaufgaben müssen im Spiel Codescape gelöst werden:
  https://codescape.medien.rwth-aachen.de/progra/
  Diese Aufgaben werden getrennt von den anderen Hausaufgaben gewertet.

Tutoraufgabe 1 (Programmierung mit Arrays):

Bubblesort ist ein Algorithmus zum Sortieren von Arrays, der wie folgt vorgeht, um ein Array `a` zu sortieren: Das Array wird wiederholt von links nach rechts durchlaufen. Am Ende des n-ten Durchlauf gilt, dass die letzten n Array-Elemente an ihrer endgültigen Position stehen. Folglich müssen im $(n+1)$-ten Durchlauf nur noch die ersten `a.length`$-n$ Elemente betrachtet werden. In jedem Durchlauf wird in jedem Schritt das aktuelle Element mit seinem rechten Nachbarn verglichen. Falls das aktuelle Element größer ist als sein rechter Nachbar, werden sie getauscht.
Als Beispiel betrachten wir das Array `{3,2,1}`. Im ersten Durchlauf wird erst `3` mit `2` getauscht (dies ergibt `{2,3,1}`) und dann `3` mit `1`, was `{2,1,3}` ergibt. Im zweiten Durchlauf wird `2` mit `1` getauscht, was zu `{1,2,3}` führt.
Implementieren Sie eine Klasse `Bubble` mit einer Methode `public static void bubble(int[] a)`, die das Array `a` mithilfe des Algorithmus *Bubblesort* aufsteigend sortiert.

Hinweise:

- Auf der Website zur Vorlesung stehen drei Java-Dateien `Sort.java`, `Bubble.java` und `Selection.java` zum Download zur Verfügung. Speichern Sie alle drei Dateien in einem neuen Ordner. Die Klasse `Bubble` enthält eine Methode `bubble` mit leerem Rumpf. Wenn Sie die Implementierung vervollständigen und anschließend mit `javac Sort.java` kompilieren, dann können Sie Ihre Implementierung mit `java Sort bubble` testen.

Aufgabe 2 (Programmierung mit Arrays): (7 Punkte)

Selectionsort ist ein Algorithmus zum Sortieren von Arrays, der wie folgt vorgeht, um ein Array `a` zu sortieren: Das Array wird wiederholt von links nach rechts durchlaufen. Am Ende des $(n-1)$-ten Durchlaufs gilt, dass die ersten $(n-1)$ Array-Elemente an ihrer endgültigen Position stehen, d.h. bis zur Stelle $n-2$ ist das Array bereits korrekt sortiert. Folglich müssen im n-ten Durchlauf nur noch die letzten `a.length`$-n+1$ Elemente betrachtet werden. Im n-ten Durchlauf wird die Position i_{min} des kleinsten Elementes der letzten `a.length`$-n+1$ Elemente bestimmt. Danach wird das Element an Position $n-1$ mit dem Minimum an Position i_{min} vertauscht.
Als Beispiel betrachten wir das Array `{3,7,1,5,8}`.
Im ersten Durchlauf $(n=1)$ wird $i_{min}=2$ bestimmt, da die kleinste Zahl (1) des Arrays an Position 2 steht. Sie wird daher mit dem Element 3 an Position $n-1=0$ vertauscht und es entsteht das Array `{1,7,3,5,8}`.
Im zweiten Durchlauf $(n=2)$ bestimmt man $i_{min}=2$ und erhält `{1,3,7,5,8}`.
Im dritten Durchlauf ergibt sich $i_{min}=3$, dies führt zu `{1,3,5,7,8}`.
Im vierten und letzten Durchlauf $(n=4)$ wird wieder $i_{min}=3$ bestimmt. Die Vertauschung des Elements an der Position $n-1=3$ mit dem Element an der Position $i_{min}=3$ ändert daher nichts an dem Array. Das sortierte Array ist also `{1,3,5,7,8}`.
Implementieren Sie eine Klasse `Selection` mit einer Methode `public static void selection(int[] a)`, die das Array `a` mithilfe des Algorithmus *Selectionsort* aufsteigend sortiert.

Hinweise:

- Auf der Website zur Vorlesung stehen drei Java-Dateien `Sort.java`, `Bubble.java` und `Selection.java` zum Download zur Verfügung. Speichern Sie alle drei Dateien in einem neuen Ordner. Die Klasse `Selection` enthält eine Methode `selection` mit leerem Rumpf. Wenn Sie die Implementierung vervollständigen und anschließend mit `javac Sort.java` kompilieren, dann können Sie Ihre Implementierung mit `java Sort selection` testen.

Tutoraufgabe 3 (Hoare-Kalkül):

Gegeben sei folgendes Java-Programm P:

$\langle$ `true` $\rangle$ (Vorbedingung)

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

$\langle$ `res` $=$ `x` $\in \{$`a`$[j] \mid 0 \leq j \leq$ `a.length`$-1\}$ $\rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation 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.

Beachten Sie bei der Anwendung der "Bedingungsregel 1" mit Vorbedingung φ und Nachbedingung ψ, dass auch $\varphi \wedge \neg B \implies \psi$ gelten muss. D. h. die Nachbedingung ψ der `if`-Anweisung muss aus der Vorbedingung φ der `if`-Anweisung und der negierten Bedingung $\neg B$ folgen. Geben Sie beim Verwenden der Regel einen entsprechenden Beweis an.

Hinweise:

- 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.
- Der Ausdruck `x` $\in$ `M` hat den Wert `true`, wenn `x` in der Menge `M` enthalten ist, sonst hat der Ausdruck den Wert `false`.

$\langle$`true`$\rangle$

$\langle$ _____ $\rangle$
```
i = 0;
```
$\langle$ _____ $\rangle$
```
res = false;
```
$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$
```
while (i < a.length) {
```
$\langle$ _____ $\rangle$
```
    if (x == a[i]) {
```
$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$
```
        res = true;
```
$\langle$ _____ $\rangle$
```
    }
```
$\langle$ _____ $\rangle$
```
    i = i + 1;
```
$\langle$ _____ $\rangle$
```
}
```
$\langle$ _____ $\rangle$

$\langle$`res` $= x \in \{$`a`$[j] \mid 0 \leq j \leq$ `a.length`$-1\}\rangle$

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung bewiesen werden.

Geben Sie auch bei dieser Teilaufgabe einen Beweis für die Aussage $\varphi \wedge \neg B \implies \psi$ bei der Anwendung der "Bedingungsregel 1" an.

Aufgabe 4 (Hoare-Kalkül): (8+2 = 10 Punkte)

Gegeben sei folgendes Java-Programm P:

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

```
n = a.length;
```

```
i = 0;
res = true;
while(i < n - 1) {
  if(a[i]>a[i+1]) {
    res = false;
  }
  i = i + 1;
}
```

$\langle\, \texttt{res} = \forall 0 \leq k < \texttt{n}{-}1 : \texttt{a}[k] \leq \texttt{a}[k{+}1] \,\rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation 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.

Beachten Sie bei der Anwendung der "Bedingungsregel 1" mit Vorbedingung φ und Nachbedingung ψ, dass auch $\varphi \wedge \neg B \implies \psi$ gelten muss. D. h. die Nachbedingung ψ der `if`-Anweisung muss aus der Vorbedingung φ der `if`-Anweisung und der negierten Bedingung $\neg B$ folgen. Geben Sie beim Verwenden der Regel einen entsprechenden Beweis an.

Hinweise:

- 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.
- Der Ausdruck $\texttt{res} = \forall 0 \leq k < i : \texttt{a}[k] \leq \texttt{a}[k+1]$ hat den Wert `true`, wenn für jede natürliche Zahl $k \in \{0, 1, \ldots, i-1\}$ jeweils $\texttt{a}[k] \leq \texttt{a}[k+1]$ gilt, sonst hat der Ausdruck den Wert `false`. Ist i eine ganze Zahl kleiner als 1, so hat der Ausdruck ebenfalls den Wert `true`. Die Nachbedingung in unserem Beispiel besagt also, dass `res` genau dann den Wert `true` hat, wenn das Array aufsteigend sortiert ist.

```
                        ⟨a.length > 0⟩

                        ⟨_____________________________________________⟩
n = a.length;

                        ⟨_____________________________________________⟩
i = 0;

                        ⟨_____________________________________________⟩
res = true;

                        ⟨_____________________________________________⟩

                        ⟨_____________________________________________⟩
while (i < a.length - 1) {

                        ⟨_____________________________________________⟩

                        ⟨_____________________________________________⟩
    if(a[i]>a[i+1]) {

                        ⟨_____________________________________________⟩

                        ⟨_____________________________________________⟩
        res = false;

                        ⟨_____________________________________________⟩
    }

                        ⟨_____________________________________________⟩
    i = i + 1;

                        ⟨_____________________________________________⟩
}

                        ⟨_____________________________________________⟩
                        ⟨res = ∀0 ≤ k < n − 1 : a[k] ≤ a[k + 1]⟩
```

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung bewiesen werden.

Geben Sie auch bei dieser Teilaufgabe einen Beweis für die Aussage $\varphi \wedge \neg B \implies \psi$ bei der Anwendung der "Bedingungsregel 1" an.

In den nächsten beiden Aufgaben sollen sie Speicherzustände zeichnen. Angenommen wir haben folgenden

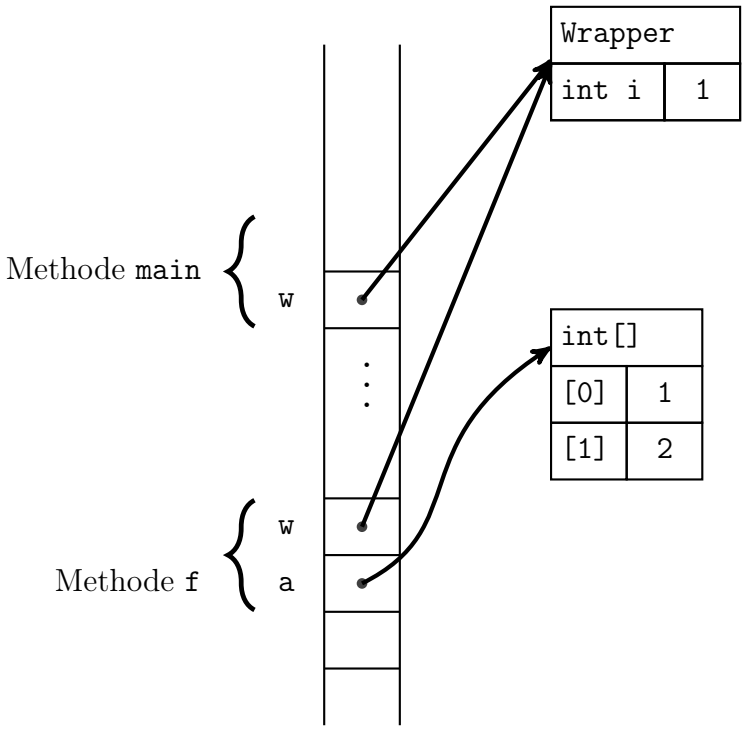
Java Code.

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

```
public class Main {
        public static void f (Wrapper w){
          int[] a = {1,2};
          w.i = 1;
          //Speicherzustand hier gezeichnet.
        }
        public static void main (String[] args){
          Wrapper w = new Wrapper();
          w.i = 0;
          f(w);
        }

}
```

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



Tutoraufgabe 5 (Seiteneffekte):

Betrachten Sie das folgende Programm:

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

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

        f(ws[1], ws[1], ws[0]);
        // Speicherzustand hier zeichnen
```

```
    }

    public static void f(Wrapper w1, Wrapper... ws) {
        int sum = 0;
        // Speicherzustand hier zeichnen
        for (int j = 0; j < ws.length; j++) {
            Wrapper w = ws[j];
            sum += w.i;
            w.i = j + 2;
        }
        // Speicherzustand hier zeichnen
        w1 = ws[1];
        w1.i = -sum;
    }
}

public class Wrapper {

    public int i;

}
```

Es wird nun die Methode `main` ausgeführt. Stellen Sie den Speicher (d.h. alle (implizit) im Programm vorkommenden Arrays (außer `args`) und alle Objekte sowie die zu dem Zeitpunkt existierenden Programmvariablen) an folgenden Programmstellen graphisch dar:

- nach der Deklaration von `sum` in dem Aufruf der Methode `f`
- nach der Schleife in der Methode `f`
- vor Ende der `main` Methode

Aufgabe 6 (Seiteneffekte): (6 Punkte)

Betrachten Sie das folgende Programm:

```
public class HSeiteneffekte {
        public static void main(String[] args) {
                Wrapper w1 = new Wrapper();
                Wrapper w2 = new Wrapper();

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

                int[] a = { 1, 2 };

                f(a, w1, w2);
                int[] b = {2*a[0],2*a[1]};
                f(b, w1);
                f(a);
        }

        public static void f(int[] a, Wrapper... ws) {
                if(ws.length==0){
                        a = new int[2];
                        a[0] = 8;
                        a[1] = 9;
```

```
                }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 Wrapper {

    public int i;

}
```

Es wird nun die Methode `main` ausgeführt. Stellen Sie den Speicher (d.h. alle (implizit) im Programm vorkommenden Arrays (außer `args`) 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.

Tutoraufgabe 7 (Einfache Klassen):

In dieser Aufgabe beschäftigen wir uns mit dem berühmten Gaunerpärchen *Bonnie und Clyde*. Wenn die beiden nicht gerade Banken ausrauben, gehen sie gerne im Wald Pilze sammeln (bzw. klauen).
Wir verwenden hier die Klassen `Main`, `Mensch` und `Pilz`, die Sie auf der Homepage herunterladen können.
Jeder dieser (beiden) **Mensch**en hat einen Korb, in den eine feste Anzahl von Pilzen passt. Weiterhin hat jeder Mensch einen Namen. Wie Sie in der Klasse `Mensch` sehen können, gibt es hierfür drei Attribute. Das Attribut `anzahl` gibt hierbei an, wie viele Pilze bereits im Korb enthalten sind.
Zu jedem **Pilz** kennen wir den Namen.

a) Vervollständigen Sie die Klasse `Main` wie folgt:

- Ergänzen Sie an den mit `TODO a.1)` markierten Stellen den Code so, dass die Variablen `steinpilz`, `champignon`, `pfifferling` auf Pilz-Objekte mit passenden Namen verweisen.
- Ergänzen Sie an den mit `TODO a.2)` markierten Stellen den Code so, dass die Variablen `bonnie` und `clyde` auf passende Mensch-Objekte zeigen. Setzen Sie hierfür jeweils den passenden Namen und sorgen Sie dafür, dass in Bonnies Korb maximal 3 Pilze Platz haben. Bei Clyde passen 4 Pilze in den Korb.

b) Gehen Sie in dieser Teilaufgabe davon aus, dass die Attribute bereits alle auf vernünftige Werte gesetzt sind.

- Erweitern Sie die Klasse `Mensch` um eine Methode `hatPlatz()`, die `true` genau dann zurückgibt, wenn im Korb Platz für einen weiteren Pilz ist. Anderenfalls wird `false` zurückgegeben.
- Schreiben Sie für die Klasse `Mensch` eine Methode `ausgabe()`. Diese gibt kein Ergebnis zurück, aber gibt den Namen und eine lesbare Übersicht der von der Person gesammelten Pilze aus. Geben Sie in der ersten Zeile den Namen der Person gefolgt von einem Doppelpunkt („:“) aus. Schreiben Sie pro Pilz im Korb eine weitere Zeile, in der (nur) der Name des jeweiligen Pilzes steht. Verwenden Sie hierzu eine `foreach` Schleife.

  Hier ist eine Beispielausgabe von einem Menschen mit Namen „Gustav“ und einem Korb, der einen Pilz mit Namen „Morchel“ und einen Pilz mit Namen „Steinpilz“ enthält:

  ```
  Gustav:
  Morchel
  Steinpilz
  ```

c) Vervollständigen Sie die Klasse `Main` wie folgt:

- Schreiben Sie eine Methode `public static void sammle(Pilz[] wald, Mensch... menschen)`. Diese arbeitet die Pilze im Wald nach und nach ab. Dabei füllt sie die Körbe der eingegebenen Menschen der Reihe nach auf. Hierbei wird zuerst überprüft, welcher der erste der eingegebenen Menschen ist, der noch Platz hat. Gibt es noch einen solchen Menschen, so wird der Pilz in seinen Korb eingefügt. Benutzen Sie hierfür auch das Attribut `anzahl` und passen Sie seinen Wert entsprechend an. Gibt es keinen Menschen, der noch Platz in seinem Korb hat, passiert nichts. Nutzen Sie hierfür eine `foreach`-Schleife. Rufen Sie nach Abarbeitung eines jeden Pilzes die Methode `ausgabe()` für jeden Menschen auf. Geben Sie anschließend eine Zeile aus, in der nur „`---`“ (drei Bindestriche) steht.

Listing 1: Main.java

```
1  public class Main {
2    public static void sammle( Pilz[] wald, Mensch... menschen){
3      // TODO c)
4
5    }
6
7
8    public static void main(String[] args) {
9      Pilz steinpilz = // TODO a.1)
10
11     Pilz champignon = // TODO a.1)
12
13     Pilz pfifferling = // TODO a.1)
14
15     Mensch bonnie = // TODO a.2)
16
17     Mensch clyde = // TODO a.2)
18
19     Pilz[] wald = {steinpilz, champignon, champignon, pfifferling,
20       steinpilz, pfifferling, champignon};
21
22     sammle(wald, bonnie, clyde);
23   }
24 }
```

Listing 2: Mensch.java

```
1  public class Mensch {
2    String name;
3    Pilz[] korb;
4    int anzahl = 0;
5  }
```

Listing 3: Pilz.java

```
1  public class Pilz {
2    String name;
3  }
```

Aufgabe 8 (Einfache Klassen): (2.5+0.5+5+3+6=17 Punkte)

Wir programmieren eine Klasse, die einen Kaffeevollautomaten simuliert. Dazu benötigen wir eine Klasse `Hotdrink` für Heißgetränke und eine Klasse `Coffeemaker` für einen Kaffeevollautomaten. Ein Kaffeevollautomat hat ein Heißgetränk-Array, das die verfügbaren Getränke enthält. Zudem gibt es Attribute, die die Größe des Milchtanks und des Wassertanks beschreiben (in Liter) und ein Attribut, das die Größe des Kaffeebohnenfachs (in Kilogramm) beschreibt. Desweiteren gibt es für jedes Fach bzw. jeden Tank Attribute, welche die aktuelle Füllung (in Liter bzw. Kilogramm) beschreiben.
Ein Heißgetränk hat einen Namen und kennt die Mengen an Wasser, Milch und Kaffeebohnen, die benötigt werden, um es in einem Kaffeevollautomaten zu produzieren.

a) Schreiben Sie die Klassen `Hotdrink` und `Coffeemaker`. Legen Sie dazu die oben beschriebenen Attribute mit geeigneten Datentypen an. Schreiben Sie außerdem eine Methode `public String toString()` in der Klasse `Hotdrink`, die bei Aufruf den Namen des Heißgetränks als String zurückgibt.

b) Schreiben Sie in der Klasse `Coffeemaker` eine Methode `public void refill()`, die den Wassertank, den Milchtank und das Kaffeebohnenfach wieder maximal auffüllt.

c) Schreiben Sie in der Klasse `Coffeemaker` eine Methode `public boolean getDrink(String hotdrink, boolean large)`. Diese gibt `true` zurück, falls das Heißgetränk im Automat zur Verfügung steht und der Kaffeevollautomat noch genügend Zutaten hat, um das Heißgetränk zu produzieren. Weiterhin aktualisiert die Methode in diesem Fall die Füllstände der verschiedenen Fächer. Gehen Sie davon aus, das ein großes Getränk doppelt so groß ist wie ein normales.

Sollte die Maschine nicht genügend Zutaten enthalten, gibt die Methode `false` zurück und druckt auf dem Bildschirm den Grund aus.

d) Schreiben Sie in der Klasse `Coffeemaker` eine Methode `public String toString()`. Die Ausgabe soll sichtbar machen, welche Heißgetränke aktuell noch in welcher Größe verfügbar sind.

e) Vervollständigen Sie die Methode `public static void simuliere(Coffeemaker machine)` in der Klasse `Main.java`. Diese Methode füllt zu Beginn alle Fächer des Kaffeevollautomaten `machine` maximal auf. Danach wird in einer Schleife zuerst ausgegeben, welche Getränke noch in welcher Größe verfügbar sind. Anschließend kann der Benutzer sein Getränk in der Größe *normal* oder *groß* wählen. Die Schleife läuft solange, bis der Benutzer ein nicht verfügbares Getränk eingegeben hat.

Ein Beispiellauf könnte dann so aussehen:

```
Was moechten Sie trinken?
Zur Auswahl stehen
Latte Macchiato Gross/Normal
Milchkaffee Gross/Normal
Espresso Gross/Normal

Eingabe des Benutzers : Latte Macchiato
Gross? (J)a/ (N)ein
Eingabe des Benutzers : J
Hier ist Ihr Latte Macchiato
Was moechten Sie trinken?
Zur Auswahl stehen
Milchkaffee nur Normal
Espresso nur Normal
Eingabe des Benutzers : Milchkaffee
Gross? (J)a/ (N)ein
Eingabe des Benutzers : J
Nicht genug Milch
```

Hinweise:

- Nutzen Sie zum Vergleich von Strings die Methode `equals`.
- Nutzen Sie für die Ein- und Ausgabe die Klasse `SimpleIO`, die auf der Website verfügbar ist.
- Nutzen Sie `foreach`-Schleifen, wann immer es möglich ist.

Aufgabe 9 (Deck 2): (Codescape)

Lösen Sie die Räume von Deck 2 des Spiels Codescape.