EINFÜHRUNG IN DIE INFORMATIK 1
– WINTERSEMESTER 2016/17 –

Lösungsvorschläge und Erklärungen
zur **Klausur** vom 21.02.2017

Note	Punkte
1,0	**70,5** – 75
1,3	**66** – 70
1,7	**61,5** – 65,5
2,0	**57** – 61
2,3	**52,5** – 56,5
2,7	**48** – 52
3,0	**43,5** – 47,5
3,3	**39** – 43
3,7	**34,5** – 38,5
4,0	**30** – 34
4,3	20 – 29,5
4,7	10 – 19,5
5,0	0 – 9,5

Alle Angaben ohne Gewähr

Stefan Berktold

Stand: 16.03.2017

Inoffizielle Lösungen. Keine Garantie für Richtigkeit.

Fragen/Nachhilfe/Übung: *s.berktold@tum.de*

Tutorium WS 16/17: tutor16.stecrz.de

Aufgabenübersicht

Aufgabe 1 – **Multiple Choice** [12]

Ausführliche Erläuterungen oder Begründungen zu den einzelnen Teilaufgaben sind unterhalb der Tabelle aufgeführt!

Beachte, dass eine falsche Antwort zu Punktabzug führt, d. h. besser nichts ankreuzen als etwas Falsches, sonst fehlen gleich zwei Punkte.

		Wahr	Falsch
1)	Der Ausdruck `"5".equals(5)` evaluiert zu `true`.	☐	☒

2)	Die folgenden Schleifen sind (semantisch) äquivalent für einen beliebigen Schleifenkörper `ss`:
	`for (int i = 0; i < 10; ++i) ss;`
	`for (int i = 0; i < 10; i++) ss;`
		☒	☐

3)	Folgende Klasse kompiliert *nicht*, da der Variable `bar` mehrfach Werte zugewiesen werden und diese `final` ist: `final class Foo { int bar; void baz() { bar = 1; bar = 2; } }`	☐	☒
4)	Folgende Klasse kompiliert *nicht*, da die Variable `bar` nicht initialisiert wurde: `class Foo { int bar; int baz() { return bar; } }`	☐	☒

5)	Folgende Klasse kompiliert:
	
	`class Foo<T> {`
	`  void bar() {`
	`  Object x = new T();`
	`  }`
	`}`
		☐	☒

6)	Die folgende Methode ist *nicht* endrekursiv:
	
	`int f(int x) {`
	`  if (x > 100)`
	`  return x;`
	`  else`
	`  return f(f(x+1));`
	`}`
		☒	☐

7)	Bei einem Integer-Overflow wird eine `IntegerOverflowException` geworfen.	☐	☒

8)	Eine Datei mit mehreren öffentlichen Klassen, wie z. B.
	
	`public class A { }`
	`public class B { }`
	
	kompiliert *nicht*.
		☒	☐

9)	Alle Objekte werden auf dem Stack allokiert.	☐	☒
10)	Wenn `A` und `B` zwei Klassen sind, die von `Object` erben, dann kompiliert folgender Code: `class Foo implements A, B { }`	☐	☒

11)	Folgender Code gibt `true` auf der Konsole aus:
	
	`int a[] = { 1,2,053,4 };`
	`int b[][] = { {1,2,4}, {2,2,1}, {0,43,2} };`
	`System.out.println(a[3] == b[0][2]);`
		☒	☐

12)	Eine Klasse kann von einer anderen Klasse erben oder ein Interface implementieren, aber sie kann *nicht* beides.	☐	☒

Erläuterungen zu Aufgabe 1

zu 1): **Zwei Objekte unterschiedlichen Typs sollten nie als gleich gewertet werden**, es sei denn sie stehen in einer Vererbungshierarchie in Relation. Die Klassen `String` und `Integer` haben nichts miteinander zu tun, weshalb auch immer `false` zurückgegeben werden sollte. In der Implementierung der `equals`-Methode wird i. d. R. ein Vergleich über die `getClass`-Methode benutzt und `false` zurückgegeben, wenn die Klassen der beiden Objekte (`this` und dem Parameter) nicht identisch sind. Alternativ wird mit dem `instanceof`-Operator gearbeitet, welcher zu `true` auswertet, wenn die Klasse des Objekts (links vom Operator) der angegebenen Klasse (rechts vom Operator) **oder einer Unterklasse** dieser entspricht.

Andere Beispiele:

- `new Double(4).equals(4) == false`
- `new Integer(4).equals(4.0) == false`
- `new Integer(4).equals(4) == true`
- `new Integer(4).equals((char) 4) == false`
- `new Integer(65).equals((int) 'A') == true`
- `5.equals(5)` geht nicht (Compiler-Fehler), da `5` primitiv (kein Objekt)

Es lohnt sich, die offizielle Java-Dokumentation zu `equals` einmal durchzulesen. Dort wird u. a. erwähnt, dass `x.equals(null)` immer zu `false` auswerten sollte und dass in der (von `Object` geerbten) Standardimplementierung von `equals` einfach auf Referenzgleichheit geprüft wird, also dass `x.equals(y) == true` gilt, genau dann wenn `x == y` gilt.

zu 2): `x++` führt dazu, dass das alte `x` (d. h. vor der Inkrementierung) zurückgegeben wird und `x` anschließend um 1 erhöht wird, während `++x` bedeutet, dass `x` zuerst erhöht und dann zurückgegeben wird, d. h.:

```
x++:  int ret = x;        ++x:  x = x + 1;
      x = x + 1;                int ret = x;
```

Ob der Inkrementoperator `++` also vor oder nach der Variable steht spielt nur eine Rolle, wenn eine Zuweisung oder ähnliches stattfinden (`ret` also verwendet werden würde), was hier nicht der Fall ist. Etwas klarer wird das, wenn man die `for`-Schleifen in äquivalente `while`-Schleifen umschreibt.

```
int i = 0;
while(i < 10) {
   ss;
   ++i;   // bzw. i++;
}
```

Es spielt also keine Rolle, wo der Operator steht, da mit dem von `++i` bzw. `i++` zurückgegebenen Wert nichts mehr gemacht wird. Das sind auch „nur“ Ausdrücke, die eben zu `x` (bei `x++`) oder `x+1` (bei `++x`) auswerten, aber beide *als Seiteneffekt* eine Inkrementierung von `x` durchführen.

zu 3): `bar` ist *nicht* `final` und kann daher sehr wohl mehrfach Werte zugewiesen bekommen. Lediglich die Klasse `Foo` ist `final`, was im Übrigen eine Spezialisierung dieser Klasse verbietet, d. h. keine Klasse kann von `Foo` erben. Das hat aber nichts mit der Variable `bar` zu tun.

zu 4): Im Unterschied zu lokalen Variablen (und Parametern) werden Membervariablen (Objektvariablen/Attribute) und Klassenvariablen (statisch) **immer implizit mit Null-Werten initialisiert**. Objekte wie beispielsweise `Strings` oder `LinkedLists` bekämen dabei tatsächlich den Wert `null`, während Variablen primitiver Datentypen einen äquivalenten primitiven Wert bekommen, d. h. `0` bei einem `int`, `0.0` bei einem `double`, `false` bei einem `boolean` usw.

zu 5): Weil `T` ein generischer Typ ist kann der Ausdruck „`new T()`“ nicht ausgeführt werden. Es ist ja überhaupt nicht klar, ob die Klasse/der Typ, der für `T` schlussletzt benutzt wird, überhaupt einen parameterlosen Konstruktor anbietet!

Beispiel: Angenommen ich schreibe eine Klasse `Kreis`, die einen Konstruktor `Kreis(double radius)` anbietet, dann wäre der Aufruf `new Kreis();` ja gar nicht möglich! Dementsprechend sind Konstruktoraufrufe für generische Typen von Haus aus „verboten“ (→ Compiler-Fehler).

zu 6): Die Methode `f` wäre endrekursiv, wenn der rekursive Aufruf (oder ein Rekursionsanker) die letzte Aktion zur Berechnung von `f` ist. Anders ausgedrückt: **Eine Methode ist endrekursiv, wenn das Ergebnis des rekursiven Aufrufs einfach zurückgegeben werden kann**. Das ist hier nicht der Fall, da wir zwei rekursive Aufrufe haben und das Ergebnis des „inneren“ Aufrufs als Parameter an den „äußeren“ rekursiven Aufruf übergeben werden muss. Etwas klarer wäre es bei der Addition zweier rekursiver Ergebnisse (`f(x-1)+f(x-2)`), was ebenfalls *nicht* endrekursiv wäre.

zu 7): Eine derartige Exception existiert nicht. Abgesehen davon wird überhaupt keine Exception geworfen, sondern (aufgrund der bitweisen Speicherung von Zahlen) mit der größtmöglichen bzw. kleinstmöglichen Zahl „fortgefahren“, d. h. insb. **`Integer.MAX_VALUE + 1 == Integer.MIN_VALUE`** ist wahr!

zu 8): Nur (maximal) eine öffentliche Klasse pro Datei. Diese Klasse muss denselben Namen haben wie die Datei (Generics zählen nicht zum Klassennamen).

zu 9): Alle Objekte werden auf dem **Heap** allokiert. Auf dem Stack werden bspw. vorhergehende Methodenaufrufe zusammen mit deren jeweiligen (lokalen) Variablen u. Ä. bzw. threadspezifische Referenzen einzelner Threads gesichert. So erhalten wir beim Auftreten einer Exception bspw. Zugriff auf die sog. „**Stack** Trace“, welche auch direkt auf der Konsole (in `rot`) ausgegeben wird, sofern die Exception nicht gefangen (`try-catch`) wird (sonst im `catch`-Block mittels `e.printStackTrace()`).

zu 10): Dieser Code würde nur kompilieren, wenn `A` und `B` Interfaces (`interface`) wären. Klassen (`class`) können nicht „implementiert“ (→ `implements`) werden; von ihnen kann man nur erben (und zwar von exakt einer).

zu 11): In Java beginnen wir mit dem Zählen bei 0 (wenn es sich wie hier um Indizes handelt). Damit ist `a[3]` das vierte Element im Array `a`, also `4`. `b[0][2]` ist das dritte Element (`[2]`) im ersten (`[0]`) Element, d. h.: `b[0]` ist `{1,2,4}`, davon das dritte Element (`[2]`) ist `4`. „`4==4`“ wertet zu `true` aus (primitiv).

Weiteres Beispiel: `System.out.println(a[2] == b[2][1]);`

Die Ausgabe ist `true`, weil `a[2]` = `053` eine Zahl im Oktalsystem ist (weil sie mit einer 0 beginnt). Wandelt man diese ins Dezimalsystem um, erhält man $a[2] = 3 * 8^0 + 5 * 8^1 = 3 + 40 =$ **43** `== b[2][1]` (Element 1 des Elements 2).

zu 12): Jede Klasse außer `Object` erbt von **exakt einer Klasse**. Ist bei einer Klassendefinition *kein* „`extends` ...“ angegeben, so erbt jede Klasse *implizit* von `Object` (oberste Oberklasse von allen Klassen). Jede Klasse kann **beliebig viele** (oder auch kein) Interfaces implementieren. D. h. insbesondere dass es *keine* Klasse gibt, die ein Interface implementieren aber nicht erben, da jede Klasse (außer `Object`) erbt.

Beispiel: `class Haus extends Gebaeude implements Bewohnbar`

Aufgabe 2 – **Rekursion** [8]

`f(2)`: **2** `f(5)`: **15**

Iterativ („Fakultät im Zweierschritt“):

```
public static int f(int n) {
    int prod = 1;
    while (n > 1) {
        prod = prod * n;
        n = n - 2;
    }
    return prod;
}
```

Rekursiv:

```
public static int fRec(int n) {
    if (n <= 1)
        return 1;
    return n * fRec(n-2);
}
```

Diese Implementierung ist *nicht* endrekursiv, da das Ergebnis des rekursiven Aufrufs noch mit `n` multipliziert werden muss und nicht einfach zurückgegeben werden kann.

Wie kommt man darauf? Spiele mehrere Beispiele im Kopf durch und du wirst feststellen, dass die Berechnung sehr an die Fakultät erinnert.

Beispiel: `f(7) = 1*7*5*3` `f(5) = 1*5*3` `f(193) = 1*193*191*189*...`

Man kann also schreiben: `f(7) = 7*f(5)` `f(5) = 5*f(3)` `f(3) = 3*f(1)`
Damit ist die Rekursionsgleichung aufgestellt. Als Anker wählen wir alle Zahlen kleiner oder gleich 1, d. h. `f(n) = n*f(n-1)` für `n>1` und `f(n) = 1` sonst (`n<=1`).

Eine mögliche **endrekursive** Implementierung wäre (nicht Teil der Aufgabe):

```
public static int fTailRec(int n) {
    return fTailRecHelper(n, 1);
}

private static int fTailRecHelper(int n, int acc) {
    if (n <= 1)
        return acc;
    return fTailRecHelper(n-2, n*acc);
}
```

Aufgabe 3 – **Ausdrücke** [12]

(i) `(++a < 4 || a < --b) ? ++a : b++ % ++a`

Antwort: **5**

(ii) `a++ + ++a - b--`

Antwort: **2**

(iii) `4 + "" + 2 * 7`

Antwort: **"414"**

(a) `b = a++ + ++b - --a`

`a =` **3** `b =` **5**

(b) `b = a++ - ++a + (b+1)`

`a =` **5** `b =` **3**

(c) `a = ++b + --a + b + b++`

`a =` **17** `b =` **6**

(1) `3 + 4 - 5 % 6`

Antwort: **(((**`3 + 4`**)** `- 5`**)** `% 6`**)**

(2) `1 - 4 + 7 / 8`

Antwort: **((**`1 -` **(**`4 + 7`**))** `/ 8`**)**

(3) `2 * 4 + "" + 4`

Antwort: **(**`2 *` **((**`4 + ""`**)** `+ 4`**))** → Fehler wegen `2 * "44"`

Aufgabe 4 – **Filter** [10]

Eine mögliche Herangehensweise ist, zuerst die Größe des neuen Arrays zu ermitteln und dieses dann zu befüllen:

```
public static int[] filter(int[] in, int n) {
    // Ermittle Größe des neuen Arrays (Anzahl an Elementen >= n)
    int size = 0;
    for (int i = 0; i < in.length; i++)
        if (in[i] >= n)
            size++;

    // Rückgabearray anlegen, Zählervariable initialisieren
    int[] ret = new int[size];
    int index = 0;

    // Befülle das Rückgabearray
    for (int i = 0; i < in.length; i++)
        if (in[i] >= n)
            ret[index++] = in[i];

    return ret;  // zurückgeben
}
```

Alternativ kann das Rückgabearray auch direkt befüllt werden, müsste anschließend aber „gekürzt“ werden. Da Arrays nicht gekürzt werden können, müsste dazu ein neues Array benutzt werden, welches dann die richtige Größe hat:

```
public static int[] filter(int[] in, int n) {
    // Neues Array anlegen, Zählervariable initialisieren
    int[] neu = new int[in.length];
    int size = 0;

    // Befülle das neue Array
    for (int i = 0; i < in.length; i++)
        if (in[i] >= n)
            neu[size++] = in[i];

    // "Kürze" das neue Array mittels Kopieren
    int[] ret = new int[size];
    while (size > 0)  // alternativ mit for-Schleife
        ret[--size] = neu[size];

    return ret;  // zurückgeben
}
```

Aufgabe 5 – **Trinäre Suche** [14]

Diese Aufgabe ist eigentlich **nicht lösbar**. Begründung auf der nächsten Seite.

```
// find sucht im gesamten Array a nach der Zahl x
public static boolean find(int[] a, int x) {
    return find(a, x, [ 0 ] , [ a.length-1 ] );
}

// Rekursive Suche im Teilbereich zwischen
// Index left und Index right (jeweils inklusive)
private static boolean find(int[] a, int x, int left, int right) {
   if (left > right) { return false; }
   if (left == right) { return [ a[left] == x ] ; }
   if ( [ left == right-1 ] ) { return a[left] == x || a[right] == x; }

   int count = 1 + (right - left);
   count = count / 3;
   int compare = a[left + count];
   if ( [ x <= compare ] ) {
       return find(a, x, left, left + count);
   }
   compare = a[right - count];
   if ( [ x >= compare ] ) {
       return find(a, x, [ right - count ] , right);
   }
   return find(a, x, left + [ count + 1 ] , right - (1 + count));
}
```

Diese Aufgabe ist eigentlich nicht lösbar, da das Array in „drei gleich große (**±1 Element**) zusammenhängende Bereiche aufgeteilt“ werden soll, wobei ein bestimmter Index „zu **genau einem** der drei neuen Bereiche gehören“ soll. Aufgrund der Vorgabe von

```
return find(a, x, left, left + count);
```

ist das schon nicht mehr möglich. Beispiel wäre ein Array der Länge 6, in welchem im Bereich von 0 (`left`) bis 5 (`right`) gesucht werden soll. `count` wäre 2, wodurch der erste Bereich bereits 3 Elemente einnehmen würde (Indizes 0 bis einschließlich 2) und die Aufgabenstellung damit nicht mehr erfüllt werden könnte, da ein Bereich höchstens die Länge 1 haben könnte (und sich damit um 2 Elemente von dem linken Bereich unterscheiden würde). Tatsächlich hätte ein Bereich unter Berücksichtigung weiterer Vorgaben sogar genau die Länge 0, die beiden anderen jeweils 3.

In der vorletzten Lücke müsste es außerdem

```
right-left==2 ? right : right - count
```

heißen, da sonst im Fall, dass der Bereich die Größe 3 hat (`right-left==2`), ein Element – nämlich das mittlere – doppelt gesucht werden würde, im linken und im rechten „Drittel“. Das mittlere Drittel ist in diesem Fall leer, wodurch das oben beschriebene Problem entsteht, d. h. man müsste hier mehrere solche Abfragen mit dem ternären Operator einbauen, um die Aufgabe komplett richtig zu lösen – eine davon müsste aber anstelle von „`left + count`“ stehen, was ja vorgegeben war.

In der vorletzten Lücke wurde aber auch „`right - count`“ akzeptiert, obwohl es die Aufgabenstellung eben auch nicht ganz erfüllt (da die Bereiche dann nicht mehr disjunkt sind).

Aufgabe 6 – **Polymorphie** [11]

Vorgehen: Finde die speziellste (d. h. am besten passende) Methode, die anhand des statischen Typs des Objekts und des statischen Typs der Parameter ausgewählt werden würde. Unterscheidet sich der dynamische Typ des Objekts von seinem statischem Typ, muss nun noch überprüft werden, ob genau diese Methode (exakt gleiche Signatur!) in der Unterklasse (oder einer spezielleren Oberklasse davon) überschrieben wird.

Statement 1: **M+**

Statement 2: **Y+M+**

Statement 3: **A+M+**

Statement 4: **M+**

Statement 5: **ClassCastException**, `da a nicht zu B gecastet werden kann`

Statement 6: **X+**

Statement 7: **Y+M+**

Statement 8: **X+**

Ausführliche Erklärung zum Vorgehen (am Beispiel von ***Statement 6***):

1. Wir bestimmen den statischen und dynamischen Typ aller im Statement verwendeter Variablen. Für den stat. Typ siehe Deklaration; der dyn. Typ („tatsächlicher" Typ zur Laufzeit) ist erkennbar an der letzten Zuweisung.

Variable	**statischer Typ**	**dynamischer Typ**
`b`	`B`	`B` (wegen `b = new B()`)
`a`	`A`	`B` (wegen `a = b`*) * und `b = new B()`

2. Nun bestimmen wir den **Methodenaufruf aus statischer Sicht** (Compiler-Sicht). Wird keine passende Methode gefunden, so gibt es einen **Compiler-Fehler** (i. d. R. rote Unterstreichung in der IDE). Wir sehen uns also von allen Variablen den stat. Typ an.

Fortsetzung auf nächster Seite

Besonderheit **Casting**: Der Cast ändert den stat. Typ *vorübergehend*, d. h. dem Compiler wird damit *einmalig* zugesichert, dass die Variable auch wirklich von dem jeweiligen Typ ist. Hier wird dem Compiler zugesichert, dass `a` wirklich den Typ `A` (oder spezieller) hat. Dieser Cast ist zwecklos, da der Compiler das sowieso weiß, nachdem ihm der stat. Typ `A` ja bekannt ist. Interessanter ist das Casting in ***Statement 7***, da dem Compiler dort zugesichert wird, dass `a` vom Typ `B` ist, was sein kann (da `A` und `B` etwas miteinander zu tun haben), aber nicht sein muss (wie in ***Statement 5***). Wäre der Cast (wie in Statement 5) nicht möglich, erhielten wir eine `ClassCastException` (Laufzeitfehler). Würde ich bspw. versuchen, `a` zu einem `Integer` (Wrapper-Klasse zu `int`) zu casten (mittels `(Integer)a`), gäbe das ein Compiler-Fehler, da der Compiler schon sieht, dass das gar nicht sein kann, nachdem die Klasse `Integer` nichts mit der Klasse `A` (stat. Typ von `a`) zu tun hat (d. h. sie stehen in keiner *Vererbungsrelation*).

Wir haben nun den Aufruf einer Methode `print` auf einem Objekt mit stat. Typ `A` (durch den Cast zugesichert, war aber davor auch schon `A`). Bzgl. der Parameter betrachten wir immer den stat. Typ. In diesem Fall suchen wir also **in der Klasse `A`** nach einer **Methode `print`**, die **genau einen Parameter vom Typ `B`** akzeptiert. Wir wählen dabei wiederum die speziellste („passendste“) Methode aus: Hier kommt nur **`print(A)`** in Frage; gäbe es aber noch eine Methode `print(Object)`, würden wir trotzdem `print(A)` wählen, obwohl `print(Object)` auch passen würde; gäbe es noch eine Methode `print(B)`, würden wir diese wählen (da unser Parameter ja den stat. Typ `B` hat, wobei `B` spezieller ist als `A` und `A` spezieller als `Object`); gäbe es noch ein `print(Integer)` würde uns das reichlich wenig interessieren (da `B` nichts mit `Integer` zu tun hat).

3. Jetzt erst werfen wir einen Blick auf den dyn. Typ des Objekts, auf dem wir die Methode `print(A)` aufrufen möchte. Entspricht der stat. Typ dem dyn. Typ, dann ändert sich an der gefundenen Methode nichts. Hier ist der dyn. Typ des Objekts, auf dem die Methode `print` aufgerufen wird (`a`) aber nicht `A` (stat. Typ), sondern `B` (vgl. Tabelle), also spezieller. Wir sehen also in der Klasse `B` nach, ob dort **exakt diese** (in Schritt 2 ermittelte) Methode (`print(A)`) überschrieben wird. Ist das wie hier der Fall, **so wählen wir diese Methode aus (`print(A)` in `B`)**.

   Was wir also machen ist, zu kucken, ob die Methode in einer spezielleren Unterklasse überschrieben wird. Gäbe es die Methode `print(A)` in `B` nicht, würde sich also an der gefundenen Methode „nichts ändern“; gäbe es aber *zusätzlich* eine Klasse `X` zwischen `A` und `B` (d. h. `B` erbt von `X` und `X` von `A`) mit einer Methode `print(A)`, die es in `B` nicht gäbe, dann würden wir die Methode `print(A)` aus `X` wählen (also immer die speziellste).

4. Nun führen wir die gefundene Methode aus; in diesem Fall `print(A)` in `B` mit der Ausgabe **`X+`**.

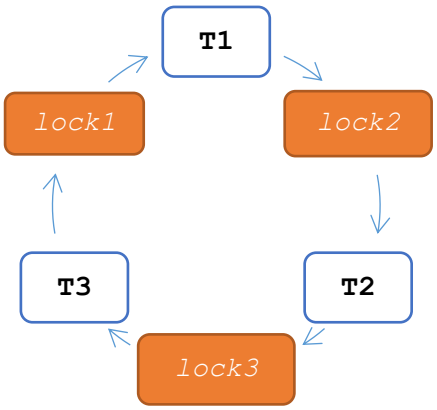
Weitere Erklärungen persönlich (Nachhilfe/Übung).

Aufgabe 7 – **Threads** [8]

Erinnerung: Ein ***Deadlock*** ist ein Zustand, in dem eine Gruppe von Threads wechselseitig auf Locks warten (d. h. versuchen ein bestimmtes Lock zu bekommen), das ein jeweils anderer Thread der Gruppe besitzt.

Warum ist das hier möglich?

`T1` holt sich (d. h. akquiriert) `lock1` und versucht dann `lock2` zu bekommen. `T2` holt sich aber *davor* `lock2` und versucht dann `lock3` zu bekommen, weshalb `T1` darauf warten muss, dass `T2` das `lock2` wieder freigibt. `T3` holt sich `lock3` jedoch wiederum *vor* `T2`, weshalb `T2` darauf warten muss, dass `T3` das `lock3` wieder freigibt. `T3` versucht anschließend das `lock1` zu bekommen, was `T1` aber besitzt. Damit müssen alle Threads aufeinander warten! Diese Situation heißt Deadlock und kann wie beschrieben eintreten (muss aber nicht!).



Ein Programm-Ablauf, der zu einem Deadlock führt, wäre also bspw.:

Thread1	Thread2	Thread3
`T1.run:Enter`		
	`T2.run:Enter`	
		`T3.run:Enter`
`lock1.lock:Enter`		
`lock1.lock:Return`		
	`lock2.lock:Enter`	
	`lock2.lock:Return`	
		`lock3.lock:Enter`
		`lock3.lock:Return`
`lock2.lock:Enter`		
	`lock3.lock:Enter`	
		`lock1.lock:Enter`

Die `run`-Methode wird nicht verlassen, daher kein `T_.run:Return`. Methodenaufrufe finden weiter keine statt. Die Aufrufe im **rot** markierten Abschnitt können beliebig „vertauscht“ werden (wobei `lock:Enter` natürlich jeweils **vor** `lock:Return` stattfinden muss). Die Aufrufe im **grün** markierten Abschnitt führen zum Deadlock und können in der Reihenfolge beliebig vertauschen werden. Es gibt aber noch weitere Möglichkeiten (bspw. `lock2.lock:Enter` von Thread1 zwei Zeilen nach oben schieben)!

Weiteres Beispiel:

Thread1	Thread2	Thread3
T1.run:Enter		
	T2.run:Enter	
		T3.run:Enter
		lock3.lock:Enter
lock1.lock:Enter		
		lock3.lock:Return
	lock2.lock:Enter	
	lock2.lock:Return	
	lock3.lock:Enter	
lock1.lock:Return		
		lock1.lock:Enter
lock2.lock:Enter		

Möglicher Ablauf **ohne Deadlock** (nicht Teil der Aufgabe):

Thread1	Thread2	Thread3
T1.run:Enter		
	T2.run:Enter	
		T3.run:Enter
	lock2.lock:Enter	
lock1.lock:Enter		
	lock2.lock:Return	
	lock3.lock:Enter	
lock1.lock:Return		
		lock3.lock:Enter
lock2.lock:Enter		
	lock3.lock:Return	
	write("Thread2")	
	lock3.unlock:Enter	
	lock3.unlock:Return	
		lock3.lock:Return
	lock2.unlock:Enter	
	lock2.unlock:Return	
	T2.run:Return	
lock2.lock:Return		
		lock1.lock:Enter
write("Thread1")		
lock2.unlock:Enter		
lock2.unlock:Return		
lock1.unlock:Enter		
lock1.unlock:Return		
		lock1.lock:Return
		write("Thread3")
T1.run:Return		
		lock1.unlock:Enter
		lock1.unlock:Return
		lock3.unlock:Enter
		lock3.unlock:Return
		T3.run:Return