Einführung in die Informatik 1

Prof. Dr. Harald Räcke, R. Palenta, A. Reuss, S. Schulze Frielinghaus 21.02.2017

Klausur

Vorname	Nachname
Matrikelnummer	Unterschrift

- Füllen Sie die oben angegebenen Felder aus.
- Schreiben Sie nur mit einem dokumentenechten Stift in schwarzer oder blauer Farbe.
- Verwenden Sie kein "Tipp-Ex" oder ähnliches.
- Die Arbeitszeit beträgt **90** Minuten.
- Prüfen Sie, ob Sie **10** Seiten erhalten haben.

vorzeitige Abgabe um ..... Hörsaal verlassen von ..... bis .....

1	2	3	4	5	6	7	Σ

Aufgabe 1 Multiple Choice [12 Punkte]

Kreuzen Sie zutreffende Antworten an. Punkte werden nach folgendem Schema vergeben:

- Richtige Antwort: 1 Punkt
- Falsche Antwort: −1 Punkt
- Keine Antwort: 0 Punkte

Eine negative Gesamtpunktzahl wird zu 0 aufgerundet.

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

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

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; } }`	☐	☒
Folgende Klasse kompiliert *nicht*, da die Variable `bar` nicht initialisiert wurde: `class Foo { int bar; int baz() { return bar; } }`	☐	☒

	Wahr	Falsch
Folgende Klasse kompiliert: `class Foo<T> { void bar() { Object x = new T(); } }`	☐	☒
Die folgende Methode ist *nicht* endrekursiv: `int f(int x) { if (x > 100) return x; else return f(f(x+1)); }`	☒	☐
Bei einem Integer-Overflow wird eine `IntegerOverflowException` geworfen.	☐	☒

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

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

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]);`
	☒	☐

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

Aufgabe 2 Rekursion [8 Punkte]

Gegeben ist die folgende Methode:

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

Welchen Rückgabewert liefert die Methode bei den folgenden Aufrufen?

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

Schreiben Sie eine Methode `public static int fRec(int n)`, die für alle Integer den gleichen Rückgabewert liefert wie die Methode `f`.

Sie dürfen weitere Methoden schreiben, aber nur selbstgeschriebene Methoden verwenden. Es dürfen keine Schleifen vorkommen.

Geben Sie für jede Ihrer Methoden an, ob diese endrekursiv ist, und begründen Sie Ihre Antwort *kurz*.

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

Da nach einem rekursiven Methodenaufruf noch Berechnungen durchgeführt werden müssen, ist die Methode nicht endrekursiv.

Aufgabe 3 Ausdrücke [12 Punkte]

Zu welchen Werten werden die folgenden Ausdrücke ausgewertet? Kennzeichnen Sie `String`s durch Anführungszeichen, z. B. `"Text123"`. Gehen Sie davon aus, dass vor jeder Teilaufgabe `a = 3` und `b = 6` gilt.

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

Antwort: 5

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

Antwort: 2

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

Antwort: "414"

Welche Werte haben die Variablen `a` und `b` nach Auswertung der jeweiligen Ausdrücke? Gehen Sie davon aus, dass vor jeder Teilaufgabe `a = 3` und `b = 4` gilt.

(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

Geben Sie durch eine vollständige Klammerung des Ausdrucks an, in welcher Reihenfolge die Teilausdrücke bei der Auswertung des gesamten Ausdrucks ausgewertet werden. Beachten Sie, dass die Operatoren die bekannte Funktionalität haben, jedoch hier eine **andere Präzedenzordnung** gelten soll. Die Präzedenzordnung ist in der Tabelle unten dargestellt. Ein Operator in einer Zeile hat eine höhere Priorität als die Operatoren in den Zeilen darunter. In einer Zeile haben die Operatoren die gleiche Priorität. Operatoren gleicher Priorität werden in der Reihenfolge von links nach rechts ausgewertet.

Beispiel: Die Auswertungsreihenfolge von `3*2-1` ist durch die vollständige Klammerung `(3*(2-1))` bestimmt.

Operatoren
`+`
`- %`
`/`
`*`

(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))`

Aufgabe 4 Filter [10 Punkte]

Implementieren Sie die Methode `public static int[] filter(int[] in, int n)`. Die Methode gibt ein `int`-Array zurück, das genau die Zahlen aus `in` enthält, die größer oder gleich `n` sind. Die Reihenfolge der Zahlen im zurückgegebenen Array muss dabei der in `in` entsprechen.

Die Größe eines Arrays `a` kann durch `a.length` bestimmt werden.

Sie dürfen keine weiteren Methoden schreiben und auch keine weiteren Methoden verwenden. Sie können davon ausgehen, dass `in` nicht `null` ist und mindestens eine Zahl enthält, die größer oder gleich `n` ist.

```
public static int[] filter(int[] in, int n) {
    int count = 0;
    for (int e : in) if (e >= n) ++count;

    int[] out = new int[count];

    for (int i = 0, j = 0; i < in.length; ++i)
        if (in[i] >= n) {
            out[j] = in[i];
            ++j;
        }

    return out;
}
```

Aufgabe 5 Trinäre Suche [14 Punkte]

Vervollständigen Sie die folgende rekursive Implementierung des Suchverfahrens *trinäre Suche*. Dabei wird in einem **aufsteigend sortierten** Array `a` mit ganzen Zahlen nach einer bestimmten Zahl `x` gesucht. Beispiel: Im Array `a=new int[]{-1,2,7,7,14,25}` wird die Zahl `x=12` gesucht.

Der Ablauf des Verfahrens ist wie folgt: Wenn die Größe des Suchbereichs (zu Beginn das komplette Array `a`) echt kleiner als drei ist, wird direkt ermittelt, ob `x` darin enthalten ist. Ansonsten wird wie folgt rekursiv gesucht: Zunächst wird der Suchbereich in drei gleich große (±1 Element) zusammenhängende Bereiche aufgeteilt. Jeder Index aus dem gegebenen Suchbereich soll zu **genau einem** der drei neuen Bereiche gehören. Dann wird ermittelt, in welchem der drei Bereiche `x` liegen kann. Danach erfolgt der entsprechende rekursive Aufruf.

Die öffentliche Methode `find` sucht im gesamten Array `a` nach der Zahl `x`. Die rekursive Hilfsfunktion sucht im Bereich zwischen den beiden Parametern `left` und `right` (jeweils inklusive).

Gehen Sie davon aus, dass `a != null` gilt. Schreiben Sie in jede Box einen Ausdruck.

```
// 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 ([ compare >= x ]) {
      return find(a, x, left, left + count);
   }
   compare = a[right - count];
   if ([ compare <= x ]) {
      return find(a, x, [ right-left==2 ? right : right - count ], right);
   }
   return find(a, x, left + [ count + 1 ], right - (1 + count));
}
```

Bemerkungen: Es gibt Fälle, in denen die Bereichsgrößen sich um mehr als ±1 Element unterscheiden. In der 7. Box wird auch nur `right-count` akzeptiert.

Aufgabe 6 Polymorphie [11 Punkte]

Betrachten Sie die Statements 1 bis 8 in dem folgenden Programm.

- Welches Statement kompiliert nicht?
- Welches Statement kompiliert und wirft eine Exception zur Laufzeit? Wieso wird hier eine Exception geworfen?
- Was ist die Ausgabe eines jeweiligen Statements, wenn dieses kompiliert und keine Exception wirft?

Nehmen Sie jeweils für ein Statement i an, dass alle anderen Statements j mit $j \neq i$ kompilieren und keine Exception werfen.

```
public class A {
   public void print()    { System.out.print("A+"); this.print(this); }
   public void print(A a) { System.out.print("M+"); }
}

public class B extends A {
   public void print()    { System.out.print("B+"); }
   public void print(A a) { System.out.print("X+"); }
   public void print(B b) { System.out.print("Y+"); super.print((A)b); }
}

public class Poly {
   public static void main (String[] args) {
      A a = new A();
      B b = new B();
      a.print(a);           // Statement 1
      b.print(b);           // Statement 2
      a.print();            // Statement 3
      a.print((A)b);        // Statement 4
      ((B)a).print((A)b);   // Statement 5
      a = b;
      ((A)a).print(b);      // Statement 6
      ((B)a).print(b);      // Statement 7
      b.print(a);           // Statement 8
   }
}
```

Statement 1: M+ 1P

Statement 2: Y+M+ 2P

Statement 3: A+M+ 2P

Statement 4: M+ 1P

Statement 5: ClassCastException: A cannot be cast to B 1P

Statement 6: X+ 1P

Statement 7: Y+M+ 2P

Statement 8: X+ 1P

Aufgabe 7 Threads [8 Punkte]

Betrachten Sie folgendes Programm:

```
public class Locks {
    public final static Object lock1 = new Object();
    public final static Object lock2 = new Object();
    public final static Object lock3 = new Object();
}

public class T1 implements Runnable {
    public void run() {
        synchronized (Locks.lock1) {
            synchronized (Locks.lock2) {
                System.out.println("Thread1");
            }
        }
    }
}

public class T2 implements Runnable {
    public void run() {
        synchronized (Locks.lock2) {
            synchronized (Locks.lock3) {
                System.out.println("Thread2");
            }
        }
    }
}

public class T3 implements Runnable {
    public void run() {
        synchronized (Locks.lock3) {
            synchronized (Locks.lock1) {
                System.out.println("Thread3");
            }
        }
    }
}

public class Deadlock {
    public static void main(String[] args) {
        new Thread(new T1()).start(); // Thread 1
        new Thread(new T2()).start(); // Thread 2
        new Thread(new T3()).start(); // Thread 3
    }
}
```

Zeigen Sie im Folgenden, dass das Programm nicht frei von Deadlocks ist. Füllen Sie dazu die Tabelle weiter unten aus. Jede Zeile steht für einen Zeitschritt und darf daher auch nur genau einen Eintrag enthalten. Als Einträge sind *ausschließlich* folgende erlaubt:

- `T1.run:Enter` / `T1.run:Return` und analog für `T2` und `T3`
- `lock1.lock:Enter` / `lock1.lock:Return` und analog für `lock2` und `lock3`
- `lock1.unlock:Enter` / `lock1.unlock:Return` und analog für `lock2` und `lock3`

Dabei bedeuten die Einträge folgendes:

- `x.foo:Enter` gibt an, dass die Methode `foo` auf dem Objekt `x` aufgerufen wurde. Das Pendant `x.foo:Return` gibt an, dass der Aufruf zum Aufrufer zurückkehrt.
- `x.lock:Enter` versucht, das Lock auf dem Objekt `x` zu akquirieren. Das Pendant `x.lock:Return` gibt an, dass das Lock akquiriert wurde.
- `x.unlock:Enter` versucht, das Lock auf dem Objekt `x` freizugeben. Das Pendant `x.unlock:Return` gibt an, dass das Lock freigegeben wurde.

Wird eine **`synchronized`**-Methode mit dem Namen `foo` von dem Objekt `x` aufgerufen, so wird erst `x.foo:Enter` ausgeführt und dann das jeweilige Lock `x.lock:Enter` akquiriert.

Geben Sie nun einen Programm-Ablauf an, sodass ein Deadlock entsteht (das Programm terminiert also nicht). Unterstreichen Sie die Lock-Statements, die zu einem Deadlock führen.

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`