Skript für den Prüfungsvoerbereitungs Workshop Einführung in die Programmierung

Stand: Herbstsemester 2017

David Blaser, Julian Croci

3. Januar 2019

Inhaltsverzeichnis

0 Vorwort und Disclaimer **5**

1 EBNF **6**

1.1 Definitionen . . . 6
1.2 Rekursion . . . 7
1.3 Tabelle . . . 7
1.4 Grafische Darstellung einer EBNF . . . 8
1.5 Übungen . . . 8
1.6 Lösungen . . . 10

2 Einfache Java Programme **11**

2.1 Einfache Programme . . . 11
2.2 Strings . . . 11
2.3 Typen und Variablen . . . 11
2.4 Arithmetische Ausdrücke . . . 11
2.5 Variablen . . . 12
2.6 Methoden . . . 12
2.7 Boolean Expressions . . . 12
2.8 if-else . . . 13
2.9 Schleifen . . . 14
2.10 Übungen . . . 15
2.11 Lösungen . . . 16

3 Arrays **17**

3.1 Eindimensionale Arrays . . . 17
3.2 Mehrdimensionale Arrays . . . 17
3.3 Übungen . . . 18

4 Klassen **19**

4.1 Einführung und Erstellung von Objekten . . . 19
4.2 Reference Semantic . . . 19

5 Input/Output **21**

5.1 Auf der Konsole . . . 21
5.2 Von einem File . . . 21

5.3 Übungen 22

6 Objekte **23**

6.1 Abstraktion 23

6.2 Attribute 23

6.3 Methoden 23

6.4 Konstruktoren 24

6.5 Namensräume 25

6.6 static 27

6.7 Module 27

6.8 Overloading 27

6.9 Übungen 27

6.10 Lösungen 29

7 Arbeiten mit Objekten und Klassen **30**

7.1 Einführung 30

7.2 Beispiel 30

7.3 Übungen 31

7.4 Lösungen 32

8 Vererbung **33**

8.1 Einführung 33

8.2 Beispiel 33

8.3 Interfaces 34

8.4 Übungen 34

8.5 Lösungen 35

9 Exceptions **36**

9.1 Einführung 36

9.2 checked und unchecked Exceptions 36

9.3 Beispiele 36

9.4 Übungen 37

9.5 Lösungen 39

10 Generische Programmierung **40**

10.1 Einführung 40

10.2 Beispiel 40

10.3 Collections . . . 40
10.4 Übungen . . . 41
10.5 Lösungen . . . 42

11 Systematisches Programmieren **43**
11.1 Pre- und Postconditions . . . 43
11.2 Hoare Tripels . . . 43
11.3 Loops . . . 43
11.4 Ein paar Worte zu JUnit-Tests . . . 44
11.5 Übungen . . . 44
11.6 Lösungen . . . 46

Vorwort und Disclaimer

Liebe Studierenden, wir freuen uns, dass du das Skript der Lernunterstützungskommission des VIS benützt und wir wünschen dir schon mal jetzt viel Erfolg bei den Prüfungen.

Wir wollen ausdrücklich darauf Hinweisen, dass dieses Skript keinen Anspruch auf Korrektheit auf und insbesondere auf Vollständigkeit erhebt. Der Relevante Prüfungsstoff wird von Professor Gross und den Hauptassistenten kommuniziert und festgelegt.

Dennoch hoffen wir, dich mit diesem Skript Unterstützen zu können. Aus Erfahrung gilt, dass die Übungsserien eine sehr gute Vorbereitung auf die Prüfung sind. In diesem Skript findet ihr noch weitere Übungen und jeweils eine kurze Zusammenfassung des Stoffes. Die Hauptkapitel sind so durch nummeriert, dass die Nummerierung der Vorlesung von Professor Gross aus dem Herbstsemester 2017 übereinstimmen. Leider ist dies bei den Unterkapiteln nicht mehr der Fall. Sollten während dem Lernen fragen auftauchen, empfehlen wir euch wärmstens, diese in den entsprechenden Channeln unter chat.vis.ethz.ch zu stellen.

Feedback zum Script gerne an blaserd@vis.ethz.ch oder crocij@vis.ethz.ch

EBNF

EBNFs können benützt werden, um die *Syntax* einer Programmiersprachen zu beschreiben. EBNFs besitzen vier Elemente ("control forms") die man auch in Java wiederfindet:

- Aufreihung (”sequence”)
- Entscheidung oder Auswahl (”decision”)
- Wiederholung (”repetition”)
- rekursion (”recursion”)

Definitionen

Eine *EBNF Beschreibung* besteht aus einer Menge EBNF Regeln. Eine *EBNF Regel* besteht aus einer *Linken Seite (LHS)* und einer *Rechten Seite (RHS)*. LHS and RHS werden durch ⇐ verbunden. Der ⇐ bedeutet “ist definiert als“. Die LHS ist der Name der EBNF Regel und wird *kursiv* oder in <> geschrieben. Die *RHS* ist die Beschreibung für die *LHS*.

Beispiele

- <Digit> ⇐ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
- *Zug* ⇐ Lok Wagen Wagen

Control Forms

Folgende control forms können auf der Rechten Seite der EBNF Regel eingesetzt werden.

- Aufreihung: Elemente in Ordnung von links nach rechts gelesen
- Auswahl: Menge von Alternativen, getrennt durch |
- Option: Element in [und], kann gewählt werden, muss aber nicht
- Wiederholung: Element in { und }, das Element kann 0, 1 oder mehrmals genommen werden

Beispiele

- Auswahl: <Digit> ⇐ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
- Aufreihung: <Zug> ⇐ Lok Wagen Wagen
- Option: <127> ⇐ [-] 127
- Wiederholung: <Positive Zahl> ⇐ <Digit> { <Digit> }

Rekursion

Rekursion ist manchmal nötig um komplizierte Symbole zu beschreiben. Es können alle Wiederholungen durch Rekursion ausgedrückt werden, hingegen nicht jede Rekursion durch Wiederholungen. Es gibt direkte Rekursion und indirekte Rekursion: Bei der direkte Rekursion erscheint die LHS direkt wieder auf der RHS. Bei der indirekten Rekursion wird der Kreis über mehrere Regeln geschlossen.

Beachte: Es sollte immer einen Ausweg aus der Rekursion geben.

Beispiele

- Direkte Rekursion: <Viele Einsen> ⇐ 1 [<Viele Einsen>]
- Indirekte Rekursion:
  - <Regel1> ⇐ Bla [<Regel2>]
  - <Regel2> ⇐ Blup [<Regel1>]
- Endlose Rekursion: <Viele Einsen> ⇐ 1 <Viele Einsen>

Tabelle

Mithilfe der Tabelle kann man zeigen, dass ein gegebenes Wort von der *EBNF Beschreibung* akzeptiert wird. Auf der ersten Zeile der Tabelle steht der Name, mit der das Symbol übereinstimmen soll. Die weiteren Zeile der Tabelle werden jeweils aus der Vorgängerzeile durch eine der folgenden Regeln abgeleitet:

1. Ersetze einen Namen (LHS) durch die entsprechende Definition (RHS)
2. Wahl einer Alternative
3. Entscheidung ob ein optionales Element dabei ist oder nicht
4. Bestimmung der Zahl der Wiederholungen

Schritt eins und zwei werden manchmal in einem Schritt gemacht.

Beispiel Wir beachten die EBNF definiert durch die folgenden Regeln und möchten zeigen, dass -127.9 akzeptiert wird. Zur Gruppierung werden (und) verwendet.

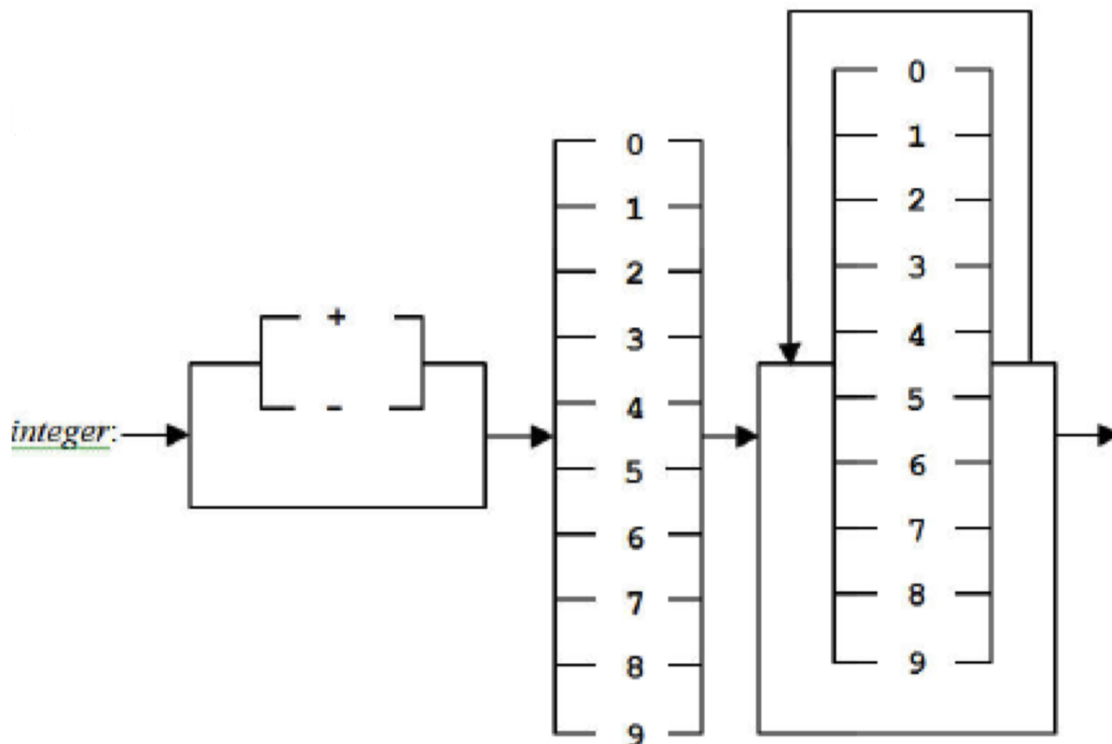
- <Zahl> ⇐ [<Vorzeichen>] <Ziffer> { <Ziffer> } [. < Ziffer> { <Ziffer> }]
- <Ziffer> ⇐ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- <Vorzeichen> ⇐ + | -

<Zahl>	Anfang
[<Vorzeichen>] <Ziffer> { <Ziffer> } [. < Ziffer> { <Ziffer> }]	Regel Zahl
<Vorzeichen> <Ziffer> { <Ziffer> } [. < Ziffer> { <Ziffer> }]	Option gewählt
(+ | -) <Ziffer> { <Ziffer> } [. < Ziffer> { <Ziffer> }]	Regel Vorzeichen
- <Ziffer> { <Ziffer> } [. < Ziffer> { <Ziffer> }]	Auswahl -
- 1 { <Ziffer> } [. < Ziffer> { <Ziffer> }]	Regel Ziffer und Auswahl 1
- 1 <Ziffer> <Ziffer> [. < Ziffer> { <Ziffer> }]	Wiederholung 2 mal
- 1 2 <Ziffer> [. < Ziffer> { <Ziffer> }]	Regel Ziffer und Auswahl 2
- 1 2 7 [. < Ziffer> { <Ziffer> }]	Regel Ziffer und Auswahl 7
- 1 2 7 . < Ziffer> { <Ziffer> }	Option gewählt
- 1 2 7 . 9 { <Ziffer> }	Regel Ziffer und Auswahl 9
- 1 2 7 . 9	Wiederholung 0 mal

Grafische Darstellung einer EBNF

Mit einem Syntax Graph lässt sich eine EBNF darstellen und es ist einfach zu erkennen, welche Zeichen und Symbole in einer Reihenfolge auftreten müssen. Von links nach rechts werden für die *Aufreihung* Elemente verbunden, eine Auswahl wird durch eine Leiter dargestellt. Eine Option wird durch eine Leiter dargestellt, die eine "leere" Zeile hat. Eine Wiederholung realisiert man gleich wie eine Option, signalisiert mit einem Pfeil zurück aber noch, dass eine Wiederholung möglich ist.

Beispiel Folgende Grafik ist aus den Slides zur Vorlesung:



Übungen

Prinzipiell sind die Übungen aus den Serien eine sehr gut Vorbereitung. Hier darum nur eine kleine Übung:

1. Die folgenden EBNFs sollten genau jene Wörter zulassen, in denen für jedes X entweder ein Y oder drei Z als Gruppe auftreten.

Beispiel für legale Symbole: ”XY”, ”XXYZZZ”, ””, ”XYYXZZZX”

Leider sind beide falsch, finde jeweils den Fehler und ein Beispielwort, welches akzeptiert wird, obwohl es nicht sollte oder umgekehrt.

1. - <xyz> ⇐ [X <xyz> <folge>]
   - <folge> ⇐ Y | ZZZ
2. - <xyz> ⇐ [{ <xy> | <xzz> }]
   - <xy> ⇐ XY | YX
   - <xzz> ⇐ XZZ | ZZX | ZXZ

2. Die folgenden EBNFs sollten genau jene Wörter akzeptieren, die folgende Anforderungen erfüllen:

- Ein Wort besteht ausschliesslich aus den Symbolen A, B, C, X, Y, Z.
- Ein Wort beginnt mit A und endet mit B.
- Nach jedem X kommt entweder ein Y oder ein Z. Y und Z können nur direkt nach einem X auftreten.
- C kann nur in Paaren auftreten (CC) und kann nicht direkt nach einem Z folgen.

Beispiele legaler Symbole: ”AB”, ”ACCB”, ”AXZB”, ”ABAB”, ”AXYXZXZXYCCB”

Leider sind alle falsch, finde jeweils den Fehler und ein Beispielwort, welches akzeptiert wird, obwohl es nicht sollte oder umgekehrt.

1. - <abc> ⇐ A {CC} <comb> B
   - <comb>⇐ [(<comb> X ((Y [CC]) | Z) <comb>) | (<comb> A [CC] <comb>) | (<comb> B [CC] <comb>)]
2. - <abc> ⇐ A [{ <c> | <xyz> | A | B }] B
   - <xyz> ⇐ XY | XZ
   - <c> ⇐ ACC | CCA | BCC | CCB | CCXY | CCXZ | YCC
3. - <abc> ⇐ A<inner>B
   - <s1> ⇐ A | B
   - <s2> ⇐ CC
   - <s3> ⇐ X (Y | Z)
   - <inner> ⇐ { [<s2>] [{<s3>} <s1>] }
4. - <abc> ⇐ A { { <buchstaben> } | {<b2>} } B
   - <buchstaben> ⇐ A | B | <spezial>
   - <spezial> ⇐ XY | XZ
   - <b2> ⇐ A | B | XY | CC
5. - <abc> ⇐ A <c> { XY <c> | XZ } B
   - <c> ⇐ { | CC}

Lösungen

1

1. Es werden nur Wörter zugelassen, die alle x-en am Anfang haben. Ein Beispiel für ein Wort, welches akzeptiert werden sollte aber nicht wird ist: XYXZZZ
2. Es werden pro X nur zwei Z produziert, nicht drei. Es könnten auch nicht drei oder mehr Y oder ZZZ "Gruppen" hintereinander gemacht werden. Ein Beispiel für ein Wort, welches akzeptiert werden sollte aber nicht wird ist: XYYYXX

2

1. Es werden nicht beliebig viele CCs akzeptiert, obwohl es so sein sollte. Ein Beispiel für ein Wort, welches akzeptiert werden sollte aber nicht wird ist: ACCACCCCB.
2. Nach einem Z sollten keine CCs möglich sein, sind aber. Folgendes Wort sollte illegal sein, wird aber akzeptiert: ACCXZCCAB
3. Ein Beispiel für ein Wort, welches akzeptiert werden sollte aber nicht wird ist: AXYCCB
4. Folgendes Wort sollte illegal sein, wird aber akzeptiert: AXZCCB
5. Die Möglichkeit A oder B mitten im Wort zu haben fehlt. Ein Beispiel für ein Wort, welches akzeptiert werden sollte aber nicht wird ist: ABAB

Einfache Java Programme

Einfache Programme

Java Programme bestehen immer aus mindestens einer Klasse mit einer main Methode:

```
public class HelloWorld {
  public static void main(String[] args) {
      System.out.println("Hello World");
  }
}
```

Der Klassen Name sollte immer mit dem Namen des entsprechenden Files übereinstimmen. Oberes Programm wäre also in dem File HelloWorld.java gespeichert.

Strings

Ein String ist eine Folge von Charakteren, eingeschlossen in Klammern. `”Hallo I bims, ein String”` ist zum Beispiel ein String. Strings können sogenannte Ersatz Darstellungen enthalten, wie zum Beispiel:

- \n Neue Zeile
- \t Tab character
- \” Quotation mark
- \\Backslash

Strings können in Java mit dem + Charakter zusammengehängt werden. Zum Beispiel `”Hello” + ”World”`.

Typen und Variablen

Typen beschrieben Eigenschaften von Daten. Der Typ schränkt die Operationen, die mit Daten gemacht werden können. Viele Programmiersprachen erfordern, dass man den Typ explizit angibt, so auch in Java. Es gibt 8 simple in Java eingebaute Typen, die wichtigsten 4 sind die folgenden:

- `int` Ganze Zahlen (1, 20, -5, etc.)
- `double` Reelle Zahlen (1.2, -6.5, 0.0, etc.)
- `char` einzelne Buchstaben (’A’, ’x’, ’*’ , etc.)
- `boolean` Logische Werte (true, false)

Arithmetische Ausdrücke

Ausdrücke (”Expressions”) sind Werte oder Operationen die einen Wert berechnen. `1 + (5 * 0.5)` ist zum Beispiel ein Ausdruck. Ausdrücke haben auch einen Typen. Es gibt die bekannten vier Grundrechenarten plus zusätzlich den Modulus (`%`) Operator, der den Rest der Division ganzer Zahlen zurück gibt. Generell gilt, dass Expressions von links nach rechts ausgewertet werden, allerdngs gilt Punkt vor Strich (Modulo gilt als Punkt). Mit Klammern kann man die Reihenfolge wie aus der Schule gewohnt anpassen. Normalerweise haben Expressions den Typ der Typen die kombiniert werden. Werden `int` und `double` gemischt hat die Epxression den Typ `double`.

Spezialfälle

- Wenn wir ganze Zahlen dividieren ist das Ergebnis auch wieder eine ganze Zahl. `14 /4` gibt also `3` und nicht `3.5`.
- Division durch 0 gibt einen Laufzeitfehler.

Variablen

Variablen speichern Werte. Um Variablen zu gebrauchen muss man sie zuerst deklarieren, dann initialisieren und dann gebrauchen. Variablen werden wie folgt:

- deklariert: `type name;`, konkret zum Beispiel `int zahl;`
- initialisiert: `name = expression;`, konkret zum Beispiel `zahl = 5 - 2;` Dies kann öfters geschehen
- gebraucht: `10 + 1 - zahl`

Methoden

Methoden sind eine Sequenz von Anweisungen, die sich über einen Namen aufrufen lassen. Sie können einen Rückgabewert haben und Parameter entgegen nehmen. Im Moment betrachten wir nur statische Methoden. Alle statischen Methoden fangen so an: `static type name(args)`. Dies nennt man die Signatur einer Methode. Methoden werden so aufgerufen: `methodenName(args);` Hier ein Beispiel um es zu verdeutlichen.

```
public class MyProgram {
  public static void main(String[] args) {
      System.out.println(mul(5, 5*3)); // Argumente werden auch hier
                                       // durch Komma getrennt
  }

  static int sqrt(int arg1, int arg2) {
                                    // das int gibt den return Type an,
                                    // int arg* ist  der type und der Name
                                    // des Arguments.
                                    // Argumente werden durch , getrennt
    int result = arg1 * arg2;
    return result;
  }
}
```

Boolean Expressions

Boolean Expressions sind Expressions, die vom type `boolean` sind. Es gibt zwei Arten von Operatoren, Operatoren, die zwei numerische Werte vergleichen und Operatoren, die booleans kombinieren. Zuerst zu den Vergleichsoperatoren.

- `zahl1 <= zahl2` kleiner gleich
- `zahl1 < zahl2` kleiner
- `zahl1 >= zahl2` grösser gleich

- `zahl1 > zahl2` grösser
- `zahl1 == zahl2` gleich

Die Ausdrücke evaluieren zu `true`, wenn die Gleichung wahr ist. Arithmetische Ausdrücke haben höhere Präzedenz als Vergleichsoperatoren. Ebenfalls gilt es zu beachten, dass Vergleichsoperatoren keine Kette bilden können wie in der Mathematik. `2 < x < 3` gehen also nicht, dies ist, weil zuerst `2 < 3` ausgewertet wird und den type `boolean` hat und man diesen nachher nicht mit 3 vergleichen kann.

Nun zu den Booleschen Operatoren:

- `bool1 && bool2` logisches und
- `bool1 || bool2` logisches oder
- `! bool` nicht

Booleasche Operatoren haben eine niedere Präzedenz als Vergleichsoperatoren.

Short Circuit Evaluation

Um Rechenzeit zu sparen, wird der zweite boolsche Ausdruck für `&&` nur ausgewertet, wenn der erste wahr ist. Gleiches gilt für `||`, der zweite boolsche Ausdruck wird nur ausgewertet, wenn der erste falsch ist. Beispiele finden sich in den kommenden Beispielen zu if-else und Schleifen.

if-else

if und else können benutzt werden, um Entscheidungen zu machen. Sie folgen intuitiv Ihrer jeweiligen Bedeutung, auf Deutsch also falls und sonst. Folgendes Beispiel illustriert dies.

```
public class MyProgram {
  public static void main(String[] args) {
      int number1 = 1;
      int number2 = 10;
      int number3 = 100;
      testNumbers(number1);
      testNumbers(number2);
      testNumbers(number3);
  }

  static void testNumbers(int arg) {
      if (arg <= 0) { // arg <= 0 ist der Test
        System.out.println(arg + " ist kleinergleich 0");
      } else if (arg < 10) {   // else und if lassen sich kombinieren
        System.out.println(arg + " ist einstellig");
      } else if (arg < 100) {
        System.out.println(arg + " ist zweistellig");
      } else { // letztes else ohne if und daher auch ohne test
        System.out.println(arg + " ist drei- oder mehrstelliger");
      }
  }
}
// Output des Programmes:
```

```
// 1 ist einstellig
// 10 ist zweistellig
// 100 ist drei- oder mehrstellig
```

Wie wir sehen muss der test für eine if Anweisung immer ein Ausdruck vom type `boolean` sein. Ebenfalls sehen wir, dass sich if und else Anweisungen zu einer else if Anweisung kombinieren lassen.

Schleifen

Schleifen ermöglichen es, einen bestimme Sequenz von Instruktionen mehrmals auszuführen. Ähnlich wie bei der if Anweisung gibt es immer einen Test (der vor jeder Ausführung der Schleife geprüft wird). In Java gibt es drei Arten von Loops.

for-Loop

Bei einem for Loop hat man die Möglichkeit, die sich ändernde Variable direkt in der for Anweisung zu definieren/initalisieren. Ebenfalls kann man das Update dieser Laufvariable direkt dort ausführen. Das ganze sieht schematisch so aus: `for (Laufvariable; Test: Update) { ... }` .

```
for (int i = 0; i < 10; i = i + 1) {
  System.out.print(i + "; ");
}
// Output  0; 1; 2; 3; 4; 5; 6; 7; 8; 9;
// 10 wird nicht mehr ausgegeben, da 10 < 10 falsch ist.
// 0 wird ausgegeben, weil das update jeweils nach der
// Ausfuehrung des Loopbodies geschieht.
```

while-Loop

Der while-Loop ist dem for-Loop relativ ähnlich, allerdings muss das Updaten der Laufvariable innerhalb des Loop-Bodies geschehen und man muss die Laufvariable ausserhalb des Loops definieren und initialisieren. Dies hat natürlich auch den Vorteil, dass man den Wert der Laufvariable auch noch nach dem Loop hat. Das ganze sieht dann schematisch wie folgt aus: `while (test) {...}`. Dieses Beispiel macht das slebe wie das Beispiel oben.

```
int i = 0;
while (i < 10) {
  System.out.println(i + "; ");
  i = i + 1;
}
```

do-while-Loop

Der do-while-Loop ist dem while-Loop ähnlich, unterscheidet sich aber insofern, dass der Loop-Body mindestens einmal ausgeführt wird. Schematisch sieht dies so aus: `do { ... } while (test);`

```
int i = 11;
do {
  System.out.println(i + "; ");
  i = i + 1;
} while (i < 10);
// Output: 11;
```

Klassifizierung von Schleifen

Man unterscheidet zwischen unbestimmten (indefinite loop) und bestimmten Schleifen (definite Loop). Bei bestimmten Schleifen ist von Anfang an bekannt, wie oft die Schleife ausgeführt wird, zum Beispiel wenn es darum geht, die Zahlen zwischen 0 und 10 auszugeben. Bei unbestimmten Schleifen ist die Anzahl der Anführungen im voraus nicht bekannt, zum Beispiel wenn man auf eine spezifische Nutzereingabe wartet.

Eine Schleife, die auf ein spezielles Hinweiszeichen, einen Sentinel, wartet heisst sentinel loop. Solch ein Hinweiszeichen kann man zum Beispiel benutzen, wenn man dem User die Möglichkeit geben möchte, das Programm durch einen Befehl zu beenden.

Übungen

1 Sind die folgenden Ausdrücke gültig und wenn ja, was ergeben sie?

1. `(5 / 2) * 2.0`
2. `(1 < 5 < 10)`
3. `3.0 + 5 / 4`
4. `3 + 5 % 4`
5. `(x < 5) || (x >= 5)`
6. `(5.0 / 2) - (10 / 2)`

2

1. Schreibe eine statische Methode, die überprüft ob eine übergebene Zahl eine Primzahl ist.
2. Schreibe ein Programm, dass die Zahlen von 1 bis n durch ein Komma getrennt ausgibt, also zum Beispiel `1, 2, 3, 4`

Lösungen

1

1. 4.0
2. Error
3. 4.0
4. 4
5. true (wenn `x` definiert ist)
6. -2.5

Arrays

Eindimensionale Arrays

In einem Array können wir mehrere Werte des selben Typs speichern. Die Elemente eines Arrays werden von 0 bis zur Länge - 1 nummeriert. Bei der Initialisierung des Arrays werden alle Werte auf einen Wert gesetzt, der 0 entspricht. Im folgenden Beispiel wird aufgezeigt, wie mit Arrays gearbeitet werden kann.

```
public static void spassMitArrays() {
  int[] zahlen = new int[10]; // Initialisierung des Arrays
  for (int i = 0; i < zahlen.length; i++) {
    System.out.print(zahlen[i] + "; ");
  }
  // Output: 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;

  System.out.println();
  for (int i = 0; i < zahlen.length; i++) {
    zahlen[i] = i;
  }
  for (int i = 0; i < zahlen.length; i++) {
    System.out.print(zahlen[i] + "; ");
  }
  // Output: 0; 1; 2; 3; 4; 5 ; 6; 7; 8 ; 9;

  System.out.println();
  double[] kommaZahlen = {1.0, 2.0, 3.0, 4.0, 5.0};
  for (int i = 0; i < kommaZahlen.length; i++) {
    System.out.print(kommaZahlen[i] + "; ");
  }
  // Output: 1.0; 2.0; 3.0; 4.0; 5.0;

}
```

Mehrdimensionale Arrays

In Java kann man ebenfalls Arrays von Arrays anlegen, dadurch kann man Matrizen "simulieren". "Simulieren", da Arrays nicht unbedingt einer klassischen Matrix aus LinAlg entsprechen müssen. Zum Beispiel können unter Arrays verschieden lang sein, wie im folgenden Beispiel illustriert wird.

```
public static void spassMitMultiArrays() {
  int[][] zahlen = new int[3][]; // Initialisierung des Arrays
  for (int i = 0; i < zahlen.length; i++) {
    System.out.print(zahlen[i] + "; ");
  }
  // Output: null; null; null;

  System.out.println();
  for (int i = 0; i < zahlen.length; i++) {
    zahlen[i] = new int[i + 1];
  }
  for (int i = 0; i < zahlen.length; i++) {
    System.out.print(zahlen[i].length + "; ");
```

```
    }
    // Output: 1; 2; 3;

    System.out.println();
    for (int i = 0; i < zahlen.length; i++) {
      System.out.print("{ ");
      for (int j = 0; j < zahlen[i].length; j++) {
        System.out.print(zahlen[i][j] + " ");
      }
      System.out.print("}");
    }
    // Output: { 0 }{ 0 0 }{ 0 0 0 }

    System.out.println();
    double[][] kommaZahlen = {{1.0}, {2.0, 2.0}, {3.0, 3.0, 3.0}};
    for (int i = 0; i < kommaZahlen.length; i++) {
      System.out.print("{ ");
      for (int j = 0; j < kommaZahlen[i].length; j++) {
        System.out.print(kommaZahlen[i][j] + " ");
      }
      System.out.print("}");
    }
    // Output: { 1.0 }{ 2.0 2.0 }{ 3.0 3.0 3.0 }

  }
```

Übungen

1. Implementiere eine Matrixmultiplikation, überlege dir, wie der User den Input eingeben kann und vergiss nicht, zu überprüfen ob sein Input Sinn macht.

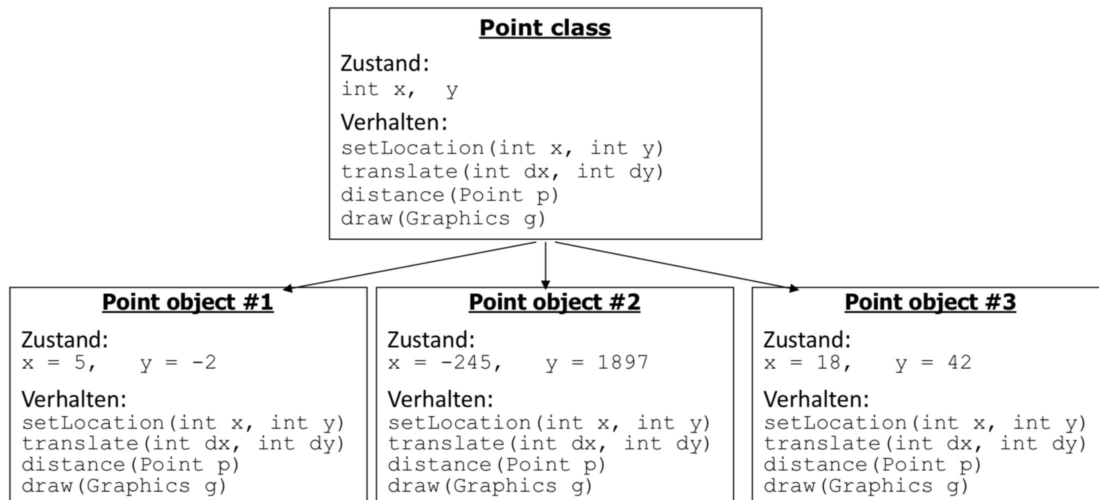
2. Schreibe eine Funktion, die als Argument ein int-Array bekommt und ein neues int Array zurückgibt, in welchem die Werte der Eingabe in umgekehrter Reihenfolge sind. Benutze keine temporären Variablen um Werte aus dem Array zu speichern.

Klassen

Dies hier ist nur ein kurzer Überblick, mehr findet ihr im Kapitel 6.

Einführung und Erstellung von Objekten

Klassen beschreiben einen Typ. Typen beschreiben Eigenschaften von Daten. Objekte sind Repräsentation von Typen zur Laufzeit des Programmes. Man könnte sagen, Klassen sind die Anleitungen um Objekte zu erschaffen und beschreiben, welche Fähigkeiten und Eigenschaften Objekte haben.



Um mit einem Objekt zu arbeiten, muss es zuerst erschaffen werden. Entweder mit dem `new` Keyword oder durch eine Initialisierung.

```
Random rand;
rand = new Random(); // Erschaffung mit new Keyword
Scanner console = new Scanner(System.in); // kann auch auf der gleichen Linie
                                          // erledigt werden
String s;
s = "Hello " + "World!"; // Erschaffung durch Initialisierung
String s2 = "Hello" + " World kompakt";
```

Objekte können von anderen Klassen benutzt werden, diese heissen Klienten oder auf Englisch client programs.

Reference Semantic

Anders als bei `int`, `boolean`, `double` und noch ein paar Anderen werden Referenzen zu den Objekten in Variablen gespeichert, nicht die Objekte selbst. Übertragen kann man es sich so vorstellen, dass nur die Adresse für den Ort gespeichert wird, an dem sich das Objekt befindet, nicht aber das Objekt selbst. Dies bedeutet, dass wenn ich etwas an dem Objekt ändere, diese Änderungen auch an anderen Stelen zum Vorschein treten können. Das gleiche gilt insbesondere wenn Funktionen Dinge an Objekten ändern, die sie als Argument bekommen haben. Da Arrays auch Objekte sind, möchte ich es in diesem Beispiel kurz illustrieren.

```
public class Main {
```

```java
    public static void main(String[] args) {
      int[] a1 = { 1, 2 };
      int[] a2 = a1;
      swapFirst2(a1);
      for (int i = 0; i < a1.length; i++) {
        System.out.print(a1[i] + " ");
      }
      System.out.println();

      for (int i = 0; i < a2.length; i++) {
        System.out.print(a2[i] + " ");
      }
    }

    public static void swapFirst2(int[] arr) {
      if (arr.length < 2) { // Nothing to swap
        return;
      } else {
        int temp = arr[0];
        arr[0] = arr[1];
        arr[1] = temp;
      }
    }
  }
  // Output of the whole program:
  // 2 1
  // 2 1
```

Generell spricht man von reference Semantics (Objekte) und value semantics (primitive types wie `int` etc.).

Input/Output

Auf der Konsole

Der Output auf die Konsole wurde bereits in einigen Beispielen benutzt. Mit `System.out.println(''MyString'');` Wird `MyString` auf der Konsole ausgegeben. Zusätzlich wird dem String noch ein `\n` (new-line character) angehängt. Mit `System.out.print(''MyString'');` wird `MyString` ausgegeben ohne new-line character.

Um Text von der Konsole einzulesen, muss man zuerst ein `Scanner` Objekt erstellen , diesem geben wir `System.in` als Parameter im Konstruktor, also haben wir insgesamt: `Scanner scanner = new Scanner(System.in);`. Um jetzt etwas einzulesen kann man die folgenden Funktionen der Scanner Klasse benutzen: `nextLine()`, `nextDouble()`, `nextInt()` etc. Werden die Werte von der Konsole eingelesen wartet das Programm auf die Eingabe. Was zu bedenken ist, ist dass eine Exception geworfen wird, sollte das gelesene nicht dem gewünschten Datentyp entspricht, doch dazu später mehr. Folgendes Beispiel liest eine Zeile von der Konsole ein und gibt sie wieder aus.

```
Scanner scanner = new Scanner(System.in);
System.out.println("Geben Sie etwas ein");
String input = scanner.nextLine();
System.out.println(input);
```

Von einem File

Die Klasse File erlaubt Operationen mit Files. Wir können wie folgend ein neues File Objekt erstellen: `File f = new File(''example.txt'');`. Nun können wir wieder einen Scanner benutzen, um auf die Inhalte des Files zuzugreifen: `Scanner scanner = new Scanner(f);` oder kombiniert: `Scanner scanner = new Scanner(new File(''example.txt''));`. Eclipse wird uns nun sagen, das wir uns um eine Exception kümmern müssen. Zu den Exceptions später mehr. Folgendes Beispielprogramm gibt ungefähr den Inhalt des Files auf der Konsole aus.

```
public static void main(String[] args)
  throws FileNotFoundException { // Falls das File nicht existiert
  Scanner scanner = new Scanner(new File(''example.txt'');
  while (scanner.hasNextLine()) {
    String input = scanner.nextLine();
    System.out.println(input);
  }
}
```

Wir sehen im Beispiel oben, dass wir eine Funktion `hasNextLine()` benützen, um zu überprüfen, ob es ein nächstes Token gibt, das den korrekten Typ hat. Sollte keines existieren, wird beim Aufruf von `next*();` eine Exception geworfen. Daher gibt es für alle `next*()` Methoden der `Scanner` Klasse auch eine entsprechende `has*();` Funktion.

Für den Output gibt es die Klasse `PrintStream`. Ein Objekt dieser Klasse können wir nachher wie das bereits bekannten `System.out` verwenden. `PrintStream output = new PrintsStream(new File(''out.txt''));` kreiert ein `PrintStream` Objekt, mit welchem man in das File out.txt zu schreiben. Um eine Zeile hinzuzufügen kann man folgendes machen: `output.println(''Hello File'');`. Doch es ist Vorsicht geboten, existiert das spezifizierte File bereits, wird es überschrieben. Folgendes Beispiel gibt die Zahlen von 0 bis 9 Zeilenweise im File `zahlen.txt` aus.

```
PrintStream output = new PrintStream(new File(''zahlen.txt''));
for (int i = 0; i < 10; i = i + 1) {
  output.println(i);
}
```

Übungen

1. Schreibe ein Programm, welches ein File liest und ausgibt, wie oft ein spezifisches Wort vorkommt. Schreibe dein eigenes File um es zu testen.

2. Schreibe ein Programm, welches ein File liest und nachher ein neues File macht, in dem alle Zeilen in umgekehrter Richtung sind. Die Funktion `toCharArray()` der Klasse `String` ist vielleicht nützlich.

Objekte

Abstraktion

Wichtig für das erstellen sinnvoller Klassen ist, dass man sich zuerst bewusst wird, welche Informationen wir für unseren Zweck überhaupt brauchen. Brauchen wir für die Analyse der Gesundheit von Personen wirklich die Farbe des T-Shirts, welches sie bei der Messung getragen haben? Wissen wir diese überhaupt? Abstraktionen sind reduzierte Beschreibungen. Sie sind weder falsch noch richtig aber können für den Zweck nützlich oder nicht sein. "Gerät das Musik spielen kann" ist für viele wohl die nützlichere Abstraktion eines iPods als "Gehäuse mit Halbleiterplatinen drin". Trotzdem sind beide Abstraktionen korrekt.

Attribute

Attribute speichern den Zustand eines Objektes. Sollte kein Konstruktor (dazu später) vorhanden sein, werden sie bei der Erstellung des Objektes auf einen Wert gesetzt, der 0 entspricht. Für Objekte ist dies `null`. Die sogenannten null Reference ist ein Wert, der vereinfacht gesagt anzeigt, dass kein Objekt existiert beziehungsweise die Variable auf kein Objekt zeigt. Es ist verboten, auf Attribute und Methoden einer null reference zuzugreifen, es erzeugt einen Laufzeitfehler. In folgendem Beispiel möchte ich aufzeigen, was Attribute sind und wie man testet, ob etwas `null` ist. Dafür kann man auch Shortcircuit evaluation ausnutzen.

```java
public class Main {
  public static void main(String[] args) {
    Car car = new Car();
    System.out.println(car.wheels + " " + car.cubicCapacity);
    if (car.brand != null) { // Check if brand String exists
      System.out.println(brand);
    } else {
      System.out.println("No Brand found");
    }
    if (car != null && car.wheels < 4) {
              // Use shortcircuit evaluation to write simpler code
      System.out.println("what a funny car");
    }
  }
}

class Car {
  int wheels;
  double cubicCapacity;
  String brand;
}
// Output of the whole program:
// 0 0.0
// No Brand found
// what a funny car
```

Methoden

Klassen können zwei verschiedene Arten von Methoden beinhalten. Statische Methoden und Instanzmethoden. Statische Methoden sind schon bekannt, `Math.abs()` ist zum Beispiel eine statische Methode. Statische Methoden existieren innerhalb einer Klasse genau einmal und können nur auf andere statische Methoden und Attribute

zugreifen. Im Gegensatz dazu hat jedes Objekt seine eigenen Instanzmethoden. Instanzmethoden können auf alle Attribute des Objektes zugreifen. Instanzmethoden müssen immer über ein Objekt aufgerufen werden, also zum Beispiel: `object.myInstanceMethod();`. Statische Methoden werden immer über die Klasse aufgerufen, zum Beispiel: `MyClass.myStaticMethod();`. Das Objekt, auf welchem die Instanzmethode aufgerufen wird, heisst Impliziter Parameter und kann über das Keyword `this` aufgerufen werden. Folgendes Beispiel soll die Unterscheide verdeutlichen.

```
public class Main {
  public static void main(String[] args) {
    Point pointObject = new Point();
    pointObject.x = 5;
    pointObject.y = 10;
    System.out.println(pointObject.distanceToOrigin());
    System.out.println(Point.constantSquare());
    System.out.println(pointObject.constantSquare());
    //System.out.println(Point.distanceToOrigin()); Doesn't work

  }
}

class Point {
  double x;
  double y;
  static int CONSTANT = 10;
  public double distanceToOrigin() {
    return Math.sqrt(x*x + y*y);    // Here I could access CONSTANT
  }
  public static int constantSquare() {
    return CONSTANT * CONSTANT; // In this method I can't access x or y
  }

}
// Output of the whole program:
// 11.180339887498949
// 100
// 100
```

Konstruktoren

Bis jetzt haben wir neue Objekte immer mit dem default Constructor konstruiert. Dies ist ein Anfang, es geht aber auch besser. Wir können nämlich auch einen eigene Konstruktoren schreiben. So sehen sie aus: `public MyClass(...) {...}`. Wie wir sehen wird kein Rückgabewert definiert. Möchte man mehrere Konstruktoren schreiben, kommen die selben Regeln zum Zug wie beim Überladen von Methoden (dazu später mehr). Wir können auch den default constructor überschreiben, dazu ebenfalls später mehr. In folgendem Beispiel füge ich der Klasse `Point` mehrere Konstruktoren hinzu.

```
public class Main {
  public static void main(String[] args) {
    Point p1, p2, p3;
    p1 = new Point(2.0, 2.5);
    p2 = new Point(3);
    p3 = new Point();
    System.out.println(p1.x + " " p1.y); // 2.0 2.5
```

```
    System.out.println(p2.x + " " p2.y); // 3.0 3.0
    System.out.println(p3.x + " " p3.y); // 1.0 1.0
  }
}

class Point {
  double x;
  double y;

  public Point(double xInit, double yInit) {
    x = xInit;
    y = yInit;
  }

  public Point(double xy) {
    x = xy;
    y = xy;
  }

  public Point() { // Override of default constructor
    x = 1;
    y = 1;
  }
}
```

Namensräume

Encapsulation

Die Idee von Encapsulation ist, die Implementationsdetails einer Klasse vor dem Klienten der Klasse zu verstecken. Dadurch werden die Daten des Objektes vor unbeabsichtigten oder unerwünschten Operationen geschützt. Durch die Trennung von Verhalten (extern sichtbar) und Zustand/Implementation (intern) wird erlaubt, dass man die Implementation einer Klasse ändert, ohne die Klienten ändern zu müssen. Dies setzt natürlich voraus, dass man sich Gedanken über das verhalten der Klasse zu Beginn macht. Um Attribute vor der Aussenwelt der Klasse zu schützen, wird das Keyword `private` benutzt. Insgesamt sieht dies dann so aus: `private type name;` als konkretes Beispiel `private int answer;`. Nur Methoden die innerhalb der Klasse definiert sind können auf private Attribute zugreifen.

Getter und Setter Methoden

Um von ausserhalb der Klasse auf private Attribute zuzugreifen, muss man Accessor und Mutator Methoden zur Verfügung stellen. Dadurch kann die Klasse zum Beispiel Buchführen über die Anzahl der Abfragen eines Wertes oder Regeln vorgeben, an welche sich die Werte halten müssen. Zur Illustration das folgende Beispiel:

```
public class Main {
  public static void main(String[] args) {
    Point p1 = new Point();
  }
}

class Point {
```

```
    private double x;
    private double y;

    public Point() { // Override of default constructor
      x = 1;
      y = 1;
    }

    public double getX() { // Getter X
      return x;
    }

    public double getY() { // Getter Y
      return Y;
    }

    public void setX(double value) {
      if (value > 0 && value < 100) // Special Rule for x
        x = value;
    }

    public void setY(double value) {
      if (value > 0 && value < 100) // Special Rule for y
        y = value;
    }
  }
```

Methoden können ebenfalls private gemacht werden.

Shadowing

Generell gilt in Java, dass eine Variable im Block, in welchem sie definiert wurde und in allen Blöcken innerhalb dieses Blockes gültig ist und dass es keine zwei gültigen Variablen mit dem selben Namen geben darf. Nun führen wir eine Ausnahme ein. Lokale Variablen und Argumente innerhalb einer Funktion oder des Konstruktors dürfen nämlich gleich heissen wie ein Attribut der Klasse der Funktion. Diese Argumente oder lokalen Variablen verdecken die Attribute, das nennt man Shadowing. Um trotzdem auf die Attribute zugreifen zu können benützt man das Keyword `this`. Folgendes Beispiel zeigt wie.

```
  public class Main {
    public static void main(String[] args) {
      Point p1;
      p1 = new Point(2.0, 2.5);
    }
  }

  class Point {
    double x;
    double y;

    public Point(double x, double y) {
      this.x = x; // this.x / this.y sind die Attribute der Klasse
      this.y = y;
    }
```

```
}
```

static

Wie bereits beschrieben sind `static` Variablen und Methoden nicht Teil eines Objekts sondern Teil einer Klasse. `static` Variablen existieren nur einmal, unabhängig wie viele Objekte der Klasse existieren. Statische Variablen und Methoden sollten die Ausnahme sein. Insbesondere ist zu beachten, dass `static` Variablen, wenn sie nicht `final` sind geändert werden können und sich folglich für alle Objekte und Klienten der Klasse, sollten die Variablen nicht `private` sein, ändern. Dadurch kann man schnell die Übersicht verlieren und mit mühsamen Bugs konfrontiert sein. Deswegen macht man sehr häufig statische Variablen, die nach aussen sichtbar sind, `final`.

Module

Module sind Teile eines Programms, welche nicht direkt ausgeführt werden können und von Klienten verwendet werden sollen. `Math` ist ein Beispiel dafür. Um diese Module zu schreiben benützt man statische Methoden und Felder. Dadurch sehen wir auch, wie wir auf statische Methoden beziehungsweise Attribute zugreifen können, nämlich so: `Class.method(parameters)` beziehungsweise `Class.attribute`. In einem konkreten Beispiel: `Math.abs(-5)` beziehungsweise `Math.PI`.

Overloading

In Java ist es möglich, dass in einer Klasse mehrere Methoden mit demselben Namen vorkommen, wenn sie über unterschiedliche Parameter verfügen. Dabei müssen sich die Parameter mindestens in ihrer Anzahl, oder in ihren Typen unterscheiden, lediglich unterschiedliche Namen sind nicht möglich. Dieses Konzept nennt sich Überladung oder Overlading von Methoden. Ein anderer Rückgabetyp ist nicht möglich. Dazu ein kleines Beispiel:

```java
public class Overloading {
  public int add(int a, int b) {
    return a + b;
  }

  public int add(int a) {
    return add(a, 1);
    // oftmals Aufruf einer Version mit mehr Parametern (mit Konstanten)
  }

  public int add (int b)  { // diese Method ist nicht erlaubt
    return add(b, 0);
  }

  public int add(boolean a) { // diese aber ist erlaubt
    return 0;
  }
}
```

Übungen

1. Finde alle Fehler im folgenden Programm:

```
1  public class Main {
2    public static void main(String[] args) {
3      Point p1;
4      p1 = new Point(2.0, 2.5);
5      System.out.println(p1.x + "␣" + p1.y + "␣" + p1.length);
6      Point[] pointArr = new Point[Random.nextInt(100) + 1]
7      for (int i = 0; i < pointArr.length - 1; i++) {
8        pointArr[i] = new Point(Random.nextInt(100), Random.nextInt(100));
9      }
10     System.out.println(pointArr[i].x + "␣" pointArr[i].y);
11   }
12 }
13
14 class Point {
15   double x;
16   double y;
17   private double length;
18
19   public Point(double x, double y) {
20     this.x = x; // this.x / this.y sind die Attribute der Klasse
21     this.y = y;
22     this.length = Math.sqrt(x * x + y * y);
23   }
24
25   public Point() {
26     x = p1.x;
27     y = p1.x
28   }
29 }
```

Lösungen

1. - In Zeile 5 kann nicht auf `p1.length` zugegriffen werden, da das Attribut `private` ist.
   - `i` ist nur innerhalb des Loops definiert, darum existiert es in Zeile 10 nicht mehr.
   - Zeile 7: Es werden nicht alle Punkte initialisiert weil der Loop nicht weit genug geht, darum gibt es bei Zeile 10 eine NullPointerException. (Wenn man annimmt, dass i korrekt ausserhalb des for-Loops definiert worden wäre.)
   - Im Namespace der Klasse `Point` ist `p1` nicht definiert, darum sind Zeilen 24 und 25 falsch.

Arbeiten mit Objekten und Klassen

Einführung

In der Vorlesung wurden in diesem Kapitel ein Paar Beispiele zu Objekten besprochen. Daher folgen nun einige weitere Übungsaufgaben.

Beispiel

Zuerst eine Beispielklasse für die nachfolgenden Übungen, eine weitere Version von `Point`:

```
public class Point {
  private int x = 1;
  private int y = 1;
  // auch beim Definieren der Attribute kann man Standardwerte angeben, das
  // erspart uns hier das Ueberschreiben des default Consturctors.

  public static final int ANTWORT = 42;

  public static String fragen() {
    return "Die Antwort ist " + ANTWORT + ".";
  }

  public int getX() {
    return x;
  }

  public void setX(int x) {
    if (x >= 0)
      this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    if (y >= 0)
      this.y = y;
  }

  public long getDistanceToOrigen() {
    return Math.round(exactDistance());
  }

  private double exactDistance() {
    return Math.sqrt(x*x + y*y);
  }
}
```

Übungen

1. Der folgende Code verwendet (als Klient) die obenstehende `Point`-Klasse. Was wird durch ihn ausgegeben?

   ```
   System.out.println(Point.fragen());
   Point p = new Point();
   Point p2 = null;
   p.setX(8);
   p.setY(-1);
   System.out.println("Der Punkt befindet sich bei (" + p.getX() + ", " +
     p.getY() + ").");
   System.out.println(p.getDistanceToOrigen());
   ```

2. Nun wird versucht, die folgenden Codezeilen jeweils einzeln direkt nach dem Code aus der vorhergehenden Aufgabe auszuführen. Gib an, ob dies erfolgreich ist. Falls es fehlschlägt, gib an, ob der Code gar nicht Kompiliert, oder ob es erst zur Laufzeit zu einem Fehler kommt.

   ```
   1 int antw = Point.ANTWORT;
   2 String was = p.fragen();
   3 int x = Point.x;
   4 int y = p.y;
   5 int y = p2.getY();
   ```

3. Pippi Langstrumpf meint, dass zwei mal drei vier ergebe. Daher ist sie mit der Multiplikation in Java unzufrieden, und du sollst ihr ein Modul schreiben, das multiplizieren kann. Dazu soll es eine Funktion anbieten, die zwei Ints als Argumente nimmt und einen Int zurückgibt, wobei sie für zwei mal drei vier zurückgeben sollte.;)
   Du darfst in dem Modul auch noch weitere Funktionen einbauen, (z.B. solle es neun geben, wenn man zu zwei mal drei nochmal drei dazu nimmt) und einige (statische) Attribute ergänzen. (z.B. eine Glückszahl)

4. Pippi hat keine Lust, ein Modul zu verwenden, denn das sei nur etwas für Erwachsene. Daher sollst du ein Programm schreiben, das das Modul verwendet. Es soll immer fortwährend zwei Zahlen von der Konsole auslesen, sie „richtig“ multiplizieren und das Ergebnis ausgeben. Da das Programm auch für Kinder geeignet sein soll, darf es unter keinen Umständen abstürzen, wenn etwas anderes als eine Zahl eingegeben wird.

5. Einige Programme enthalten sogenannte Easter Eggs. Dabei handelt es sich um verstecke, meist witzige Funktionen. Eines kann z.B. gefunden werden, indem in Firefox „about:robots“ in die Adresszeile eingegeben wird. Da das in Aufgabe 4. geschriebene Programm sowieso ein Witz ist, kann es gut mit einem Easter Egg erweitert werden: Immer wenn der Benutzer das Wort „Glück“ in die Konsole eintippt, soll umgehend die Glückszahl ausgegeben werden.

Lösungen

1. Ausgabe:

```
Die Antwort ist 42.
Der Punkt befindet sich bei (8, 1).
8
```

2. Lösung:

```
1 Erfolgreich
2 Erfolgreich (Eclipse gibt aber eine Warnung aus)
3 Kompiliert nicht, da x nicht static ist
4 Kompiliert nicht, da x nicht sichtbar ist
5 Verursacht beim Ausfuehren eine NullPointerException, da p2 == null ist
```

Vererbung

Einführung

In den vorhergehenden Kapiteln haben wir gesehen, wie Konzepte (als Abstraktion, d.h. Reduktion auf die relevanten Informationen) in Klassen dargestellt werden können. Es kommt dabei oft vor, dass ähnliche, aber doch verschiedene Konzepte dargestellt werden sollen. So könnte man in einem Beispiel Fahrzeuge haben, und als Spezialformen davon Autos und Fahrräder. Diese Konzepte haben viele gemeinsame Eigenschaften (z.B. eine Anzahl an Rädern) und Funktionen (z.B. fahren).

Um wiederzugeben, dass ein Konzept eine Spezialform von einem anderen ist, wird Vererbung (Englisch inheritance) eingesetzt. Dadurch kann auch vermieden werden, dass gemeinsame Eigenschaften und Funktionen mehrfach implementiert werden müssen. Um anzugeben, dass eine Klasse, die Subclass (oder Kindklasse) von einer anderen Klasse, der Superclass (oder Elternklasse) erbt, wird das Keyword `extends` verwendet. Dadurch übernimmt die Subclass alle Attribute und alle Methoden der Supperclass. Jedoch können in der Kindklasse die Methoden der Elternklasse überschrieben (overriden) werden, so dass sie ein anderes verhalten zeigen.

Beispiel

```java
public class Vehicle {
  public int numberOfWheels;

  public Vehicle(int wheels) {
    this.numberOfWheels = wheels;
  }

  public void drive() {
    System.out.println("Woosh");
  }
}

public class Car extends Vehicle {
  public Car() {
    super(4);
  }

  @Override
  public void drive() {
    System.out.println("Brumm");
  }
}

public class Bicycle extends Vehicle {
  public Bicycle() {
    super(2);
  }

  @Override
  public void drive() {
    System.out.println("Surrr");
  }
}
```

Interfaces

Im Zusammenhang mit Vererbung ist auch das Konzept eines Interfaces wichtig. Ein Interface ist ähnlich wie eine Klasse, die jedoch nicht instanziiert werden kann. (Also ähnlich wie eine abstrakte Klasse) Zudem dürfen Interfaces keine Implementationen für Funktionen enthalten, sie geben also nur die Signaturen von Methoden vor. Weiter erbt eine Klasse nie von einem Interface, stattdessen wird ein Interface implementiert, (Dies wird durch das `implements` Keyword angegeben) wobei eine Klasse auch mehrere Interfaces implementieren kann. (Hingegen kann eine Klasse in Java immer nur von einer Klasse erben) Im vorhergehen Beispiel könnte man, wenn man nicht will, dass generische Vehicles erstellt werden können, `Vehicle` durch ein Interface ersetzten. (Dann wäre aber die nachfolgende Übung nicht mehr ganz korrekt)

Übungen

1. Gegeben ist der folgende Code, der die Klassen aus dem vorherigen Beispiel verwendet:

```
Vehicle[] vehicles = new Vehicle[3];
vehicles[0] = new Vehicle(7);
vehicles[1] = new Car();
vehicles[2] = new Bicycle();
for (int i = 0; i < vehicles.length; i++) {
vehicles[i].drive();
System.out.println("Vehicle " + i + " has "
  + vehicles[i].numberOfWheels + " wheels.");
```

   Was wird durch diesen Code ausgegeben?

2. In dieser Aufgabe sollen einige Klassen geschrieben werden, die verschiedene Typen von Flugzeugen darstellen. Ein Flugzeug hat jeweils eine Anzahl an Sitzplätzen, eine aktuelle Geschwindigkeit sowie eine maximale Geschwindigkeit. Zudem soll eine Methode zum setzen der aktuellen Geschwindigkeit vorhanden sein, die sicherstellt, dass die maximale Geschwindigkeit nie überschritten wird. Analog dazu sollen auch Attribute und Methoden für die Flughöhe vorhanden sein. Die benötigten Flugzeugtypen und ihre Spezifikationen sind in der folgenden Tabelle angegeben. Alle Flugzeuge sollen (direkt oder indirekt) Subclassen von einer (abstrakten) Klasse `Flugzeug` sein. Dabei soll eine Vererbungsstruktur so aufgebaut werden, das möglichst wenig Code doppelt geschrieben werden muss.

Name	**Anzahl Sitzplätze**	**Max Geschwindigkeit (Knoten)**	**Max Flughöhe (Fuss)**
Kleinflugzeug	10	300	30'000
Ultraleichtflugzeug	2	130	15'000
Segelflugzeug	2	150	15'000
Grossraumflugzeug	Variabel*	600	40'000

   *Soll dem Konstruktor als Argument übergeben werden.

3. (Zusatzaufgabe) Die vorherige Aufgabe soll nun so erweitert werden, dass ein Aufruf von `System.out.println(flugzeug);` (wobei `flugzeug` eine Referenz zu einer Instanz von irgendeiner der zuvor erstellten Klassen ist) dazu führt, dass sinnvolle Informationen über die referenzierte Instanz ausgegeben werden.

Lösungen

1. Ausgabe:

```
Woosh
Vehicle 0 has 7 wheels.
Brumm
Vehicle 1 has 4 wheels.
Surrr
Vehicle 2 has 2 wheels.
```

Exceptions

Einführung

Bis jetzt wurde in einigen Beispielen (z.B. aus der Vorlesung) im Falle eines Fehlers, zum Beispiel einer fehlerhaften Eingabe oder eines ungültigen Argumentes, einfach das Programm beendet. Das ist aber oftmals keine besonders elegante Lösung. So wäre es unpraktisch, wenn ein Taschenrechnerprogramm abstürzt, wenn eine Division durch 0 auftritt. Allgemein wäre es dazu praktisch, wenn eine Funktion der Funktion, die sie aufgerufen hat, mitteilen könnte, wenn irgendetwas schief ging. (z.B weil ungültige Argumente übergeben wurden) Das wird in Java durch Exceptions ermöglicht. Eine Exception (zu Deutsch Ausnahme) kann durch eine Methode geworfen werden, wenn ein Fehler auftritt. Dadurch wird die Excpetion an die Methode, welche die werfende Methode aufgerufen hat, weitergegeben. Diese aufrufende Methode kann die Exception fangen ("catchen"), und versuchen, den Fehler zu behandeln. Falls die Exception jedoch nicht gefangen wird, wird sie wiederum an die aufrufende Methode der aufrufenden Methode weitergegeben. So wird weiter verfahren, bis eine Methode die Exception fängt, oder aber die main-Methode des Programms erreicht wird. Falls auch die main-Methode die Exception nicht fängt, wird das Programm beendet, und es wird eine Fehlermeldung auf der Konsole ausgegeben.

Zum Fangen einer Exception wird ein sogenannter try-chatch-Blcok verwendet. (Siehe Beispiel unten)

checked und unchecked Exceptions

Es gibt zwei Typen von Exceptions. Einerseits solche, die vom Programm behandelt werden können, diese heissen checked Exceptions. Immer wenn es möglich ist, dass eine solche Exception geworfen wird, muss sie entweder gefangen werden, oder die aktuelle Methode muss ankündigen, dass diese Exception geworfen wird. (Dafür wird das `throws` Keyword in der Methodensignatur verwendet) Dazu gehört beispielsweise die `FileNotFoundException`.

Anderseits gibt es aber auch unchekced Exceptions, dies werden in der Regel durch das Java Laufzeitsystem geworfen. Diese können zwar auch gefangen werden, aber das ist oft nicht sinnvoll, den oftmals ist das Programm nicht in der Lage, diese Fehler zu beheben. Denn solche Exception deuten entweder auf einen Fehler im Programm (z.B. bei einer `NullPointerException`) oder ein Problem bei der Laufzeitumgebung, die ein Programm nicht beheben kann (z.B. `OutOfMemoryError`), hin. Daher ist es bei solche Exceptions oftmals die einzige sinnvolle Lösung, das Programm abstürzen zu lassen.

Beispiele

`new FileReader("Dateiname")` kann eine FileNotFoundException werfen. Das ist eine checked Exception, d.h. sie kann entweder gefangen (und behandelt) werden, ...

```java
public static Scanner openFile() {
  try {
    return new Scanner(new FileReader("data.txt"));
  } catch (FileNotFoundException ex) {
    System.out.println("Fehler: Die Datei mit den Daten existiert nicht.");
    return null;
  }
}
```

... oder es muss angekündigt werden, dass möglicherweise eine solche Exception geworfen wird.

```java
public static Scanner openFileThrowing() throws FileNotFoundException {
  return new Scanner(new FileReader("data.txt"));
}
```

Anders hier: Dieser Code führt definitiv zu einer NullPointerException, aber diese muss weder gefangen noch angekündigt werden. Sie sollte meistens zu einem Programmabsturz führen.

```
public static int readInt() {
  Scanner sc = null; // der Scanner ist null...
  return sc.nextInt(); // ...daher wird hier eine NullPointerException geworfen.
}
```

Da eine Division durch 0 (mit Ints) nicht möglich ist, wirft diese Methode in diesem Fall eine IllegalArgumentException, die zudem auch eine hilfreiche Nachricht enthält. Die IllegalArgumentException ist aber unchecked, und muss daher nicht angekündigt werden.

```
public static int divide(int a, int b) {
  if (b == 0) {
    throw new IllegalArgumentException("Division durch 0 nicht moelglich.");
  }
  return a/b;
}
```

Wenn hingegen nicht geprüft wird, ob der Divisor 0 ist, so kann es bei der Division zu einer ArithmeticException kommen. Diese wird durch das Laufzeitsystem geworfen, und ist ebenfalls eine unchecked Exception.

```
public static int divideUnsafe (int a, int b) {
  return a/b; // Wenn b == 0 ist, so wird hier eine Exception geworfen.
}
```

Übungen

1. Was wird durch den folgenden Code, der die obenstehenden Funktionen verwendet, ausgegeben, wenn keine Datei mit dem Namen "data.txt" im Arbeitsverzeichnis existiert? Beachte dabei, dass beim Catchen eines bestimmten Typs von Exceptions auch alle Subtypen davon gecatcht werden. Zudem wird von einem try-catch-Block immer nur eine Excpetion gefangen. Denn nachdem die erste Exception geworfen wurde, wird die Ausführung des Codes im try-Block abgebrochen.

```
openFile();
try {
  openFileThrowing();
} catch (FileNotFoundException e) {
  System.out.println("Fehler: Datei existiert nicht.");
}
try {
  int i = readInt();
  divide(i, 0);
} catch (NullPointerException e) {
  System.out.println("NullPointerException! Das Program ist kaput!");
} catch (Exception e) {
  System.out.println("Irgendewas unerwartets ging schief.");
}
try {
  divide(7, 0);
  divideUnsafe(1, 0);
} catch (RuntimeException e) {
  System.out.println("Fehler: " + e.getMessage());
}
```

2. Passe deine Lösung zur Aufgabe 2. aus dem vorherigen Kapitel so an, dass jeweils eine `IllegalArgument-Exception` (mit einer hilfreichen Nachricht) geworfen wird, wenn versucht wird, die aktuelle Geschwindigkeit oder Flughöhe auf einen ungültigen Wert zu setzten.

Lösungen

1. Ausgabe:

```
Fehler: Die Datei mit den Daten existiert nicht.
Fehler: Datei existiert nicht.
NullPointerException! Das Program ist kaput!
Fehler: Division durch 0 nicht moeglich.
```

Generische Programmierung

Einführung

In einigen Übungsserien kam es vor, dass eine Struktur, die Daten speichert, z.B. eine verkettete Liste, umgeschrieben werden musste, um einen anderen Type an Daten zu speichern. So gab es zwei verschiedene verkettete Listen für Personen und Integers. Diese unterschieden sich aber im wesentlichen nur im Type des Feldes `value`, ansonsten enteilten sie fast ausschliesslich duplizierten Code. Als Lösung für dieses Problem existiert in Java das Konzept der generischen Programmierung. Dieses ermöglicht sogenannte Typenparameter, damit kann einer Klasse dann, wenn eine Instanz davon erstellt wird, ein (oder mehrere) Typen als Argument übergeben werden. Dadurch ist es möglich, die in der Klasse verwendeten Typen erst beim Instanziieren festzulegen, zuvor können sie allgemein (generisch) gehalten werden.

Dadurch wird es beispielsweise möglich, dieselbe Listenimplementation für verschiedene Datentypen zu verwenden. Dennoch kann aber sichergestellt werden, dass in einer bestimmten Listeninstanz nur Daten von einem bestimmten Type gespeichert werden. Somit lassen sich unübersichtliche Casts und Typenfehler zur Laufzeit vermeiden. Gerade auch in den Standardbibliotheken von Java existieren viele generische Klassen.

Beispiel

Im folgenden der Code eines sehr generischen Paares, in dem jegliche Kombinationen von zwei Werten gespeichert werden können.

```
public class Paar<TypeA, TypeB> {
  public TypeA a;
  public TypeB b;

  Paar(TypeA initA, TypeB initB) {
    a = initA;
    b = initB;
  }

  @Override
  public String toString() {
    return "(" + a + ", " + b + ")";
  }
}
```

Collections

Im Programmen muss man oft grössere Mengen an Daten speichern, organisieren und verarbeiten. Dafür existieren in den Java-Bibliotheken (`java.util`) verschiedenste Implementationen von Datenstrukturen, wie Listen und Sets (Mengen). Diese Implementationen basieren auf einigen abstrakten Klassen und Implementieren verschiedene allgemeine Interfaces. Dadurch entstehen die sogenannten abstrakten Datentypen. Diese erlauben es beispielsweise, dass verschiedene Implementationen von Listen als Argument für eine Funktion verwendet werden können. Gleichzeitig garantieren diese Interfaces aber auch, dass gewisse Funktionen vorhanden sind. Dadurch ist es beispielsweise möglich, mit einem for-each-Loop über alle Elemente in einer (möglicherweise abstrakten) Datenstruktur zu iterieren. Dabei werden sogenannte Iterators verwendet. Dazu ein kurzes Beispiel:

```
List<Integer> list = Arrays.asList(1, 2 ,3);
for (int i : list)
  System.out.println(i);
```

Die Implementationen und Interfaces aus diesem sogenannten Collections Framework sind generisch (d.h sie verfügen über Typenparameter). Dadurch können mit diesen Datenstrukturen alle möglichen Typen von Daten verwaltet werden.

Übungen

1. Was wird ausgegeben, wenn der folgende Code, der die vorhergehende `Punkt`-Klasse verwendet, ausgeführt wird?

   ```
   Paar<Integer, Integer> punkt = new Paar<Integer, Integer>(7, 8);
   Paar<Double, Double> genauerPunkt =
   new Paar<Double, Double>(5.3, 0.1 + punkt.b);
   Paar<Integer, String> eintrag = new Paar<Integer, String>(1, "Hallo");
   Paar<String, String> heyDu = new Paar<String, String>("Hey", "Du");
   eintrag.b = "Hans";
   punkt.b = punkt.a + 1;
   System.out.println("Punkt: " + punkt);
   System.out.println("genauer Punkt: " + genauerPunkt);
   System.out.println(eintrag.b + " ist Eintrag " + eintrag.a);
   System.out.print(heyDu);
   ```

   Falls ihr nun genug habt von derartigen Fragen: Das war die letzte in diesem Skript.

2. In der Musterlösung zur Übung 7 findet ihr in der Datei LinkedIntList.java eine Implementation einer Einfach verketteten Liste, in der Integers gespeichert werden können. Passen sie diese Liste so an, dass sie generisch wird und somit für beliebige Typen verwendet werden kann. Dazu sollte die `LinkedList`-Klasse und die dazugehörende `ListNode`-Klasse je einen Typenparameter erhalten.

Lösungen

1. Ausgabe:

```
Punkt: (7, 8)
genauer Punkt: (5.3, 8.1)
Hans ist Eintrag 1
(Hey, Du)
```

Systematisches Programmieren

Pre- und Postconditions

Oftmals müssen die Argumente, die einer Methode übergeben werden, bestimmte Kriterien erfüllen. So funktioniert z.B. die in der Vorlesung präsentierte gcd-Methode nur für x>=0 und y>0 korrekt. Eine solche Bedingung, die am Anfang einer Methode (oder allgemeiner vor einem Stück Code) gelten muss, nennt man Precondition. (zu Deutsch Vorbedingung)

Ebenfalls sollen nach einem Methodenaufruf gewisse Bedingungen (z.B. für das Resultat) gelten. Eine Bedingung, die nach einer Methode (oder einem Stück Code) gelten muss, nennt man Postcondition. (zu Deutsch Nachbedingung)

Diese Bedingungen werden als logische Formeln dargestellt, und sie müssen unter den vorliegenden Variablenwerten (bzw. Werten von Argumenten) wahr sein. Damit erlauben es derartige Pre- und Postconditions auch, formale Beweise über Programme (oder Teile davon) zu führen. Dadurch kann die Korrektheit des Codes nachgewiesen werden.

Zudem kann in Java zur Laufzeit eines Programms geprüft werden, ob die Bedingungen erfüllt sind. Dazu wird das `assert`-Keyword, gefolgt von der Bedingung, verwendet. Wenn eine so eingefügte Bedienungen nicht erfüllt ist, wird ein AssertionError geworfen. Dies allerdings nur, wenn Assertions aktiviert sind.

Hoare Tripels

Ein Hoare Tripel ist ein Programm mit einer Precondition und einer Postcondition, wir stellen es folgendermassen dar: `{P} S {Q}`, wobei P die Precondition, S das Programmsegment (bzw. Statement) und Q die Postcondition ist. Ein Hoare Triple ist genau dann gültig, wenn für jeden Zustand (Werte von Variablen), der P erfüllt, die Ausführung von S zu einem Zustand führt, der Q erfüllt. Ansonsten ist das Tripel ungültig.

Bei simplen Hoare Tripels kann man schnell sehen, ob diese gültig sind. Hingegen wird es bei etwas komplizierteren Konstrukten (z.B. mit längeren Programmsequenzen oder Loops) schnell unübersichtlich. Es existieren aber genaue Regeln, die Schritt für Schritt auf das Hoare Tripple angewendet werden können, um seine Gültigkeit zu prüfen. Die Details zu diesen Regeln werden auf den Folien der Vorlesung gut dargestellt. (Vorlesungen vom 12.12, 15.12 und 22.12.2017, bzw. im Herbstsemester 2018 vom 11.12, 14.12 und 18.12.2018)

Mit diesen Regeln kann entweder vorwärts geschlossen werden, d.h. man wendet den Effekt des Programmsegments auf die Precondition an. Hingegen kann man aber auch rückwärts schliessen, dabei wird das Programmsegment dann rückwärts auf die Postcondition angewendet. Beim Rückwertsschlissen ist das Konzept der schwächsten Precondition (weakest precondition) wichtig. Dabei wird die schwächste Precondition zu einem Programmsegment und einer Postcondition gesucht, so dass ein gültiges Hoare Tripple entsteht. Die schwächste Precondition zu einem Programmsegment `S` und einer Postcondition `Q` wird mit `wp(S, Q)` bezeichnet.

Loops

Ein interessanter (aber auch komplexer) Fall sind Loops. Bei diesen wird eine sogenannte Invariante benötigt. Dies ist eine Bedingung, die sowohl vor dem Ausführen des Loops, als auch nach jeder einzelnen Iteration durch den Loop gültig ist. Sie wird meist mit `I` bezeichnet. Die Details dazu sind auf den Folien vom 19.12.2017 (bzw. 18.12.2018) anschaulich dargestellt.

Ein paar Worte zu JUnit-Tests

In den Übungen kamen das eine oder andere mal auch JUnit-Tests vor. Auf die genauen Implementationen möchte ich hier nicht eingehen, dafür finden sich in den Übungen und Musterlösungen genügend Beispiele. Häufiger standen Studierende hingegen vor dem Problem, welches die richtige Strategie ist, um Tests zu finden, die auch wirklich Bugs offenlegen. Eine Möglichkeit ist, sich Pre- und Postconditions anzuschauen, und dann zu bestimmen, welche Werte dort am Rand des Definitionsbereich der zulässigen Werte liegen. Wenn ich Beispielsweise eine Funktion testen möchte, die herausfinden soll, ob eine gegebene positive Zahl eine Primzahl ist, würde ich 0 testen, weil es die kleinste positive Zahl ist und 2, weil es die kleinste Primzahl ist und vielleicht auch noch den grösstmöglichen Integer, obwohl dies wohl eine ganze Weile dauern würde. Natürlich sollten auch andere spezielle Fälle plus wenige Standardfälle getestet werden, nur alle Zahlen zwischen 20 und 30 zu testen ist aber weniger sinnvoll.

Übungen

1. Gib an, ob die folgenden Hoare Tripples gültig sind oder nicht. Dabei kannst du annehmen, dass es sich bei allen Variablen um Ints handelt, und es nie zu Over-/Underflow kommt.

   a)

   ```
   {true}
   x = x*x;
   {x > 0}
   ```

   b)

   ```
   {x != 0}
   x x*x;
   {x >= 0}
   ```

   c)

   ```
   {a > 0}
   if (y > 5) {
     x = y;
     z = 2;
   } else {
     x = a;
     z = a;
   }
   {x > 0 && z > 0}
   ```

   d)

   ```
   {a >= 0}
   if (y > 5) {
     x = y;
   } else {
     x = a;
   }
   {x > 0}
   ```

2. Gib eine Invariante zum Loop im nachfolgenden Hoare Triple an, so dass damit seine Korrektheit gezeigt werden kann. (Widerum unter der Annahme, dass alle Variablen Ints sind, und es nie zu Over-/Underflow kommt)

```
{z == 1 && a == x && x >= 0}
while(a > 0) {
  z = z * y;
  a--;
}
{z = y^x}
```

Lösungen

1. a) ungültig

   b) gültig

   c) gültig

   d) ungültig

2. mögliche Invariante:

   ```
   {z == pow(y, x - a) && a >= 0}
   ```