

Übungsstunde 2

Einführung in die Programmierung

Nachbesprechung Übung 1

Weitere Fragen zu Eclipse oder Git?

Lösung Übung 1, Aufgabe 2

```
/*  
 * Author: Maximiliana Muster  
 * für Einführung in die Programmierung  
 */  
public class HelloProgrammer {  
  
    public static void main(String[] args) {  
        System.out.println("Hello, my name is Max!");  
    }  
}
```

Lösung Übung 1, Aufgabe 4

Erstellen Sie eine Beschreibung `<geradezahl1>`, die als legale Symbole alle geraden Zahlen (d.h. Zahlen, die ohne Rest durch 2 teilbar sind) zulässt. Beispiele sind 2, 4, 10, -20.

Lösung Übung 1, Aufgabe 4

Zeigen Sie in einer Tabelle, dass Ihre Beschreibung das Symbol “28” als gerade Zahl erkennt.

Lösung Übung 1, Aufgabe 4

Erstellen Sie eine Beschreibung $\langle x^2y \text{gemischt} \rangle$, die als legale Symbole genau jene Wörter zulässt, in denen für jedes "X" zwei "Y" als Paar auftreten. Beispiele sind XYX , YXX , $XYXXYX$.

Lösung Übung 1, Aufgabe 4

Die folgenden EBNF-Beschreibungen sind nicht äquivalent. Finden Sie ein *kürzestmögliches* Symbol, das von der einen Beschreibung als legal erkannt wird, aber nicht von der anderen. (Fangen Sie mit einfachen Kombinationen von A und B an.)

EBNF-Beschreibung: <beispiel1>

<beispiel1> \Leftarrow [A] [B]

EBNF-Beschreibung: <beispiel2>

<beispiel2> \Leftarrow [A [B]]

Lösung Übung 1, Aufgabe 4

Erstellen Sie eine EBNF Beschreibung `<doppelt>`, die als legale Symbole genau jene Wörter zulässt, in denen die doppelte Anzahl "Y" nach einer Folge von "X" auftritt. Beispiele sind `XY`, `XXYYYY`, usw.

Theorie Recap

Rekursion

■ EBNF Regel besteht aus:

$$\underline{\text{LHS} \Leftarrow \text{RHS}}$$

- Linke-Seite (Left-Hand Side, LHS)
- Rechte-Seite (Right-Hand Side, RHS)
- \Leftarrow (trennt LHS von RHS, ausgesprochen «ist definiert als»)

Rekursive Regel

- Regel ist rekursiv: ihr Name wird in der Definition verwendet
 $\textit{pos_integer} \Leftarrow \textit{digit} [\textit{pos_integer}]$
- Beschreibung ist rekursiv: mindestens eine rekursive Regel

Rekursion

- Rekursion ist ein Programm/Regel, dass sich selbst wieder aufruft
- Rekursion ist gleich Mächtig wie Wiederholung (Iteration)
 - ist aber schöner, eleganter, einfacher zu lesen
 - und aber langsamer

keyword: Ein Bezeichner («identifizier») der reserviert ist (weil er für die Sprache eine besondere Bedeutung hat)

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	var
char	finally	long	strictfp	void
class	float	native	super	volatile
const	for	new	switch	while
continue	goto	package	synchronized	

false null

true

Kommentare («comments») sind Notizen im Programmtext, die einem Leser beim Verstehen des Programmes helfen (sollen)

- Leser: kann auch der Autor sein
- Kommentare werden nicht ausgeführt, haben keinen Einfluss auf Programm

■ **2 Varianten**

- `// Text bis zum Ende der Zeile`
- `/* Text bis
zum
naechsten

*/`

Sonderzeichen

Escape Sequences

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a form feed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

Methoden

- **Methode: Sequenz von Anweisungen mit einem Namen (dem der Methode)**
- **Methoden *strukturieren* die Anweisungen**
 - Anstatt alle Anweisungen in einer Methode (main) unterzubringen
- **Methoden erlauben es, *Wiederholungen* zu vermeiden**
 - Mehrfache Ausführung, aber nur einmal im Programm(text)
- **Eine (neue) Methode stellt eine neue Anweisung zur Verfügung**

Methoden

vorerst nur static

```
public static void foo() { ... }
```

und werden mit `foo();` aufgerufen

Typen und Variablen

Typen

- **Typen («types») beschreiben Eigenschaften von Daten**
- **Ein Typ beschreibt eine Menge (oder Kategorie) von Daten Werten.**
 - Bestimmt (beschränkt) die Operationen, die mit diesen Daten gemacht werden können
 - Viele Programmiersprachen erfordern die Angabe (Spezifikation) von Typen
 - Typen Beispiele: ganze Zahlen, reelle Zahlen, Strings
- **Typen sind Teil der Dokumentation (was verarbeitet diese Methode?)**

Basistypen in Java

- Es gibt 8 eingebaute Typen («primitive types») für Zahlen, Buchstaben, etc.

<u>Name</u>	<u>Beschreibung</u>	<u>Beispiele</u>
int	ganze Zahlen	-2147483648, -3, 0, 42, 2147483647
long	grosse ganze Zahlen	-3, 0, 42, 9223372036854775807
double	reelle Zahlen	3.1, -0.25, 9.4e3
char	(einzelne) Buchstaben	'a', 'X', '?', '\n'
boolean	logische Werte	true, false

Wer definiert Typen?

- **Verlangen alle Programmiersprachen die Spezifikation von Typen?**
 - Nein. (Mit manchmal überraschenden Folgen)
 - Java verlangt nicht immer eine Spezifikation des Typs
 - *Manchmal* kann der Compiler den Typ herausfinden
- **Welche Typen kann ein Java Programm verwenden?**
 - Typen die in der Sprache definiert sind: Basistypen («primitive types», integrierte Typen) – Beispiel: `int` und `long` für ganze Zahlen
 - Typen aus Bibliotheken, die immer verfügbar sind (z.B. `String`)
 - Benutzer-definierte Typen

Typen und **Variablen**

Variablen

Deklaration

Initialisierung

Gebrauch (und evlt. neue Zuweisungen)

Variable Deklaration und Definition:

type name = value

- **Java: können Deklaration mit Zuweisung verbinden**
- **Value ist ein (passender) Wert ...**
 - «passend» -- d.h. vom Typ `type`

Variable Deklaration: *type name*

- **Beispiel: `int laenge;`**
- **`int` – Art (Typ) der Werte für diese Variable**
 - Gleich mehr über Typen
- **`laenge` – Name der Variable**
 - Frei wählbar, mit Einschränkungen
 - Keine Java Keywords, muss mit Buchstabe anfangen, ...
 - Gross- und Kleinbuchstaben sind unterschiedlich
- **Deklaration: Erklärung, Bekanntmachung**

Zuweisungen («Assignment»)

- **Zuweisung: Anweisung die Wert in einer Variable speichert.**

- Wert kann ein Ausdruck sein, die Zuweisung speichert das Ergebnis
- Fürs erste: Zuweisungen in einer Methode

- **Syntax: *name = expression;***

- `int x;`
`x = 3;`

x	3
---	---

- `double meineNote;`
`meineNote = 3.0 + 2.25;`

meineNote	5.25
-----------	------

Zuweisung (Programm) und Algebra (Mathematik)

- Zuweisung verwendet = , aber eine Zuweisung ist keine algebraische Gleichung!

= bedeutet: «speichere den Wert der RHS in der Variable der LHS»

Die rechte Seite des Ausdrucks wird zuerst ausgewertet,
dann wird das Ergebnis in der Variable auf der linken Seite gespeichert

- Was passiert hier?

```
int x = 3;
```

```
x = x + 2;    // ???
```

x	5
---	---

Arbeiten mit einer Variablen

- **Variable muss deklariert sein bevor sie im Programm gebraucht wird**
 - Als Operand einer Zuweisung («assignment»)

```
int z;  
z = 1;
```
- **Variable muss Wert haben bevor sie als Operand anderer Operatoren gebraucht werden kann**
 - Wert kann von «Deklaration mit Initialisierung» oder Zuweisung kommen

Compiler Fehler Meldungen

- Eine Variable kann erst *nach* einer Zuweisung verwendet werden.

- `int x;`
`System.out.println(x);` **// ERROR: x has no value**

- Keine Doppeldeklarationen.

- `int x;`
`int x;` **// ERROR: x already exists**
 - `int x = 3;`
`int x = 5;` **// ERROR: x already exists**

Java Expressions (Ausdrücke)

Arithmetische Operatoren

- **Operator: Verknüpft Werte oder Ausdrücke.**
 - + Addition
 - Subtraktion (oder Negation)
 - * Multiplikation
 - / Division
 - % Modulus (Rest)
- **Während der Ausführung eines Programms werden seine Ausdrücke *ausgewertet* («evaluated»)**
 - 1 + 1 ergibt 2
 - `System.out.println(3 * 4);` ergibt (druckt) 12
 - Wie würden wir den Text `3 * 4` drucken?

Int-Division & Int-Modulo

- int dividiert durch int gibt einen int zurück
- Resultat wird auf den nächst kleineren int abgerundet

- Rest-Berechnung einer int-Division mit % (“modulo“-Operator)

int Rest mit %

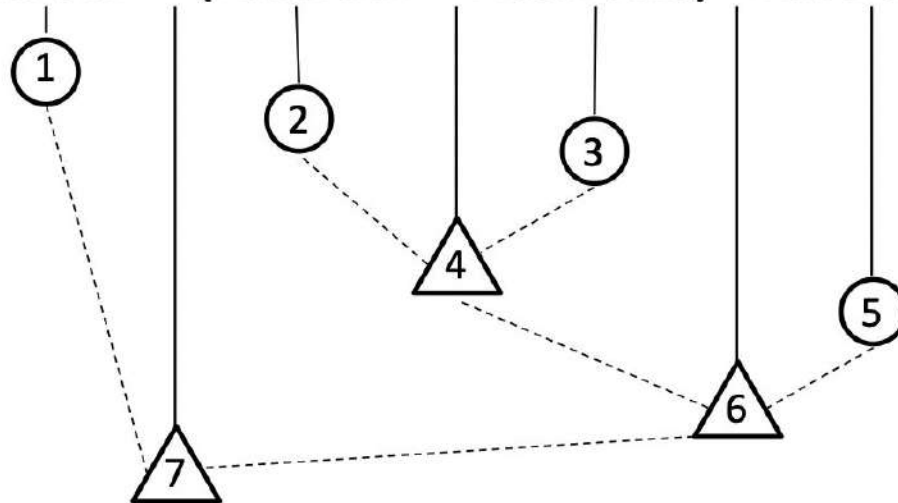
- **Einsatz des % Operators:**

- Finde letzte Ziffer einer ganzen Zahl : $230857 \% 10$ ist 7
- Finde letzte 4 Ziffern: $658236489 \% 10000$ ist 6489
- Entscheide ob Zahl gerade ist: $7 \% 2$ ergibt 1,
 $42 \% 2$ ergibt 0

Operanden und Operatoren

- Beispiel Rang Ordnung und Assoziativität:

Ausdruck1 + (Ausdruck2 + Ausdruck3) * Ausdruck4



Operator Precedence

Operators	Precedence
postfix	<i>expr</i> ++ <i>expr</i> --
unary	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

Rang Ordnung Beispiele

Welche Werte ergeben die Auswertung dieser Ausdrücke?

- $9 / 5$
- $695 \% 20$
- $7 + 6 * 5$
- $7 * 6 + 5$
- $248 \% 100 / 5$
- $6 * 3 - 9 / 4$
- $(5 - 7) * 4$
- $6 + (18 \% (17 - 12))$

$$9 / 5 ==> 1$$

$$695 \% 20 ==> 15$$

$$7 + 6 * 5 ==> 37$$

$$7 * 6 + 5 ==> 47$$

$$248 \% 100 / 5 ==> 9$$

$$6 * 3 - 9 / 4 ==> 16$$

$$(5 - 7) * 4 ==> -8$$

$$6 + (18 \% (17 - 12)) ==> 9$$

Type Casting

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size

`byte -> short -> char -> int -> long -> float -> double`

passiert “implizit”

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type

`double -> float -> long -> int -> char -> short -> byte`

muss “explizit”
gemacht werden

https://www.w3schools.com/java/java_type_casting.asp

Typ Umwandlungen

- **(type) ist ein Operator**
 - Der «cast operator»
 - Höherer Rang (Präzedenz) als arithmetische Operatoren
 - Rechts-assoziativ
 - D.h. wandelt nur den Operanden direkt rechts daneben um
- **Für alle Basistypen verfügbar**

Typ Umwandlungen

- **Syntax:**

(type) expression

- **Beispiele:**

```
(double) 19 / 5;           // 3.8  
(int) ((double) 19 / 5) ;  // 3
```

Wann brauchen wir Casting?

Bei Ausdrücken mit verschiedenen (Basis)-Typen

Passiert aber implizit, d.h.:

- ihr könnt damit immer rechnen
- bekommt aber evtl. nicht das Resultat, was ihr haben wollt

Bitte evaluieren Sie diese Ausdrücke

$4 + 8 / 3.0 * 6 + 5$

$9.0 / (2.0 / 3) + 7$

$7 \% 3 * 2 + 4.0 * 3 / 2$

$9 / (2 / 3) + 7$

$20 \% 8 + 15 / 27 / (3 \% 6)$

String Operationen

String Operationen

- **Java Regel: Für jeden Wert gibt es eine Darstellung als String**
 - Gilt für alle Typen (Basistypen, Bibliothek, selbst entwickelte Typen)
 - Ob die *default* Darstellung (D: voreingestellte *oder* standardmässig eingestellte Darstellung) sinnvoll/verständlich ist – eine andere Frage
 - Trotzdem sehr praktisch: Kann jeden Wert `W` mittels `print(W)` ausgeben
- **Der Operator `+` mit einem String und einem anderen Wert als Operanden ergibt einen längeren String**
 - Verwendet die default Darstellung

Weitere String Operationen

1 + 2 + "abc" ergibt **"3abc"**

"abc" + 9 * 3 ergibt **"abc27"**

"1" + 1 ergibt **"11"**

4 - 1 + "abc" ergibt **"3abc"**

Aussagen über Programmsegmente

Worum geht es?

Aussagen über Programme machen (unsere Java-Methoden)

Warum machen wir das?

Für einfaches Programm genau argumentieren
gute Schulung, systematisch zu programmieren
wichtig für Definition von Schnittstellen

Annahmen

Arbeiten mit Aussagen

- **Wir stellen uns vor, Positionen (Punkte) im Code haben einen Namen**
 - Point A
 - Point B
 - ...
- **Alle Anweisungen, die davor (im Programm) erscheinen, sind ausgeführt wenn wir diesen Punkt (während der Ausführung) erreichen**
- **Keine Anweisung danach wurde ausgeführt**

Annahmen

- **Alle Programm(segment)e arbeiten mit int Variablen und Werten**
 - Wir nehmen an (oder wissen) dass die endliche Darstellung kein Problem ist
 - Alle Ergebnisse können korrekt dargestellt werden

Vorwärts und Rückwärts Schliessen

Beispiel

■ Vorwärts schliessen

- Vom Zustand *vor* der Ausführung eines Programm(segments)
- Nehmen wir an wir wissen (oder vermuten) $w > 0$

// $w > 0$

$x = 17;$

// $w > 0 \wedge x$ hat den Wert 17

$y = 42;$

// $w > 0 \wedge x$ hat den Wert 17 $\wedge y$ hat den Wert 42

$z = w + x + y;$

// $w > 0 \wedge x$ hat den Wert 17 $\wedge y$ hat den Wert 42 $\wedge z > 59$

- Jetzt wissen wir einiges mehr über das Programm, u.a. $z > 59$

Beispiel

- **Rückwärts schliessen:**

- Nehmen wir an wir wollen dass z nach Ausführung negativ ist

```
// w + 17 + 42 < 0
```

```
x = 17;
```

```
// w + x + 42 < 0
```

```
y = 42;
```

```
// w + x + y < 0
```

```
z = w + x + y;
```

```
// z < 0
```

- Dann müssen wir wissen (oder vermuten) dass vor der Ausführung gilt: $w < -59$
 - Notwendig und hinreichend

Vorwärts vs Rückwärts, Teil 1

- **Vorwärts schliessen:**

- Bestimmt was sich aus den ursprünglichen Annahmen herleiten lässt.
- Sehr praktisch wenn eine *Invariante* gelten soll

- **Rückwärts schliessen:**

- Bestimmt hinreichende Bedingungen die ein bestimmtes Ergebnis garantieren
 - Wenn das Ergebniss erwünscht ist, dann folgt aus den Bedingungen die Korrektheit.
 - Ist das Ergebniss unerwünscht, dann reichen die Bedingungen um einen Bug (oder einen Sonderfall) zu generieren

Pre- und Postconditions

- **Precondition: notwendige Vorbedingungen die erfüllt sein müssen (vor Auszuführung einer Anweisung)**
- **Postcondition: Ergebnis der Ausführung (wenn Precondition erfüllt)**

Terminologie

- Die Annahme (Aussage), die vor der Ausführung eines Statements gilt, ist die *Precondition*.
- Die Aussage, die nach der Ausführung gilt (unter der Annahme dass die Precondition gültig ist und das Statement ausgeführt wurde), ist die *Postcondition*.
 - Wenn wir diesen Punkt erreichen dann gilt die Postcondition
 - Wenn wir diesen Punkt *nicht* erreichen (z.B. wegen eines Laufzeitfehlers) *dann machen wir keine Aussage*

Die übliche Notation

- **Statt die Pre/Postconditions in Kommentaren (nach //) anzugeben verwenden viele Texte {...}**

- Kein Java mehr ...
- Aber diese Schreibweise hat sich eingebürgert, lange vor Java

`{ w < -59 }`

`x = 17;`

`{ w + x < -42 }`

- **Zwischen { und } steht eine logische Aussage**

- Kein Java Code (aber wir verwenden Java's Operatoren oder die aus der Mathematik bekannten)

Hoare-Tripel

Hoare Tripel (oder 3-Tupel)

- Ein *Hoare Tripel* besteht aus zwei Aussagen und einem Programmsegment:

$$\{P\} \ S \ \{Q\}$$

- P ist die Precondition
- S das Programmsegment (bzw Statement)
- Q ist die Postcondition

Hoare Tripel (oder 3-Tupel)

- Ein *Hoare Tripel* $\{P\} \ S \ \{Q\}$ ist *gültig* wenn (und nur wenn):
 - Für jeden Zustand, für den P gültig ist, ergibt die Ausführung von S immer einen Zustand für den Q gültig ist.
 - Informell: Wenn P wahr ist vor der Ausführung von S , dann muss Q nachher wahr sein.
- Andernfalls ist das Hoare Tripel *ungültig*.

Hoare-Tripel bei Zuweisungs-Statements

Im Moment betrachten wir noch blosse Zuweisungen (später auch if-else, etc.)

Zuweisungen

$$\{P\} \ x = e; \ {Q}$$

- Bilden wir Q' in dem wir in Q die Variable x durch e ersetzen
- Das Tripel ist gültig wenn:

Für alle Zustände des Programms ist Q' wahr wenn P wahr ist

- D.h., aus P folgt Q' , geschrieben $P \Rightarrow Q'$

Zwei Beispiele

Aufgabe:

Ist das gegebene Hoare-Tripel ein gültiges Hoare-Tripel? D.h. (reminder)

- Ein **Hoare Tripel** $\{P\} \ S \ \{Q\}$ ist **gültig** wenn (und nur wenn):
 - Für jeden Zustand, für den P gültig ist, ergibt die Ausführung von S immer einen Zustand für den Q gültig ist.
 - Informell: Wenn P wahr ist vor der Ausführung von S , dann muss Q nachher wahr sein.
- Andernfalls ist das Hoare Tripel **ungültig**.

Beispiel

$\{z > 34\}$

$y = z+1;$

$\{y > 1\}$

Q' ist $\{z+1 > 1\}$

Beispiel

$\{z > 34\}$

$y = z+1;$

$\{y > 1\}$

Q' ist $\{z+1 > 1\}$

Gilt $P \Rightarrow Q'$?

Also $(z > 34) \Rightarrow (z+1) > 1$?



Beispiel

$$\{z \neq 1\}$$

$$y = z * z;$$

$$\{y \neq z\}$$

$$Q' \text{ ist } \{z * z \neq z\}$$

Beispiel

$$\{z \neq 1\}$$

$$y = z * z;$$

$$\{y \neq z\}$$

Q' ist $\{z * z \neq z\}$

Gilt $P \Rightarrow Q'$?

Also $(z \neq 1) \Rightarrow (z * z) \neq z$?
X $(z==0)$

Aufgabe 5: Postconditions

Geben Sie für die folgenden Programmsegmente die Postcondition an, welche nach der Ausführung gilt, wenn zuvor die angegebene Precondition gilt. Verwenden Sie Java Syntax. Alle Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow.

1.

```
P: { x > 10 }  
S:   y = x + 5;  
Q: { ?? }
```

2.

```
P: { x == 2 }  
S:   y = x + 5;  
Q: { ?? }
```

3.

```
P: { x > 0 && z > 0 }  
S:   y = z * x  
Q: { ?? }
```

Beispiel

```
P: { x >= 3 }  
S:   z = x/2  
Q: { ?? }
```

short-circuit

Bedingte Auswertung

- Für && und || müssen nicht immer beide Operanden ausgewertet werden, um das Ergebnis zu ermitteln
- Java *beendet* die Auswertung eines booleschen Ausdrucks sobald das Ergebnis fest steht.
- Bedingte («short-circuit») Auswertung: für boolesche Ausdrücke
- Bedingte Ausführung: für Anweisungen/Statements

Bedingte («short-circuit») Auswertung

- Für `&&` und `||` müssen nicht immer beide Operanden ausgewertet werden, um das Ergebnis zu ermitteln
- Java *beendet* die Auswertung eines booleschen Ausdrucks sobald das Ergebnis fest steht.
 - `&&` und `||` sind links-assoziativ
 - Ausdrücke werden von links nach rechts, gemäss Präzedenz und Assoziativität ausgewertet
 - `&&` stoppt sobald ein Teil(ausdruck) `false` ist
 - `||` stoppt sobald ein Teil(ausdruck) `true` ist

Bedingte («short-circuit») Auswertung

- Java *beendet* die Auswertung eines booleschen Ausdrucks sobald das Ergebnis fest steht.
 - && stoppt sobald ein Teil(ausdruck) `false` ist
 - || stoppt sobald ein Teil(ausdruck) `true` ist
- Diese Art der Auswertung heisst *bedingte Auswertung*
 - Folgende Teilausdrücke werden abhängig von zuerst ausgewerteten Ausdrücken (nicht) evaluiert

Auswertung eines Tests

- Gegeben Programm(segment) mit drei `int` Variablen `a`, `b` und `x`
- Wir wollen `x` zum Quotienten `a/b` setzen, aber nur wenn `a/b` grösser als 0 ist

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt
```

```
    // nur positive Werte sollten gespeichert werden  
    //    integer division kann 0 fuer a,b !=0 ergeben
```

```
x = ...
```

Auswertung eines Tests

- Gegeben Programm(segment) mit drei `int` Variablen `a`, `b` und `x`
- Wir wollen `x` zum Quotienten `a/b` setzen, aber nur wenn `a/b` grösser als 0 ist

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt
```

```
    // nur positive Werte sollten gespeichert werden  
    //    integer division kann 0 fuer a,b !=0 ergeben  
    // duerfen nicht durch 0 dividieren
```

```
x = ...
```

Auswertung eines Tests

- Gegeben Programm(segment) mit drei `int` Variablen `a`, `b` und `x`
- Wir wollen `x` zum Quotienten `a/b` setzen, aber nur wenn `a/b` grösser als 0 ist

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
    // nur positive Werte sollten gespeichert werden  
    //      integer division kann 0 fuer a,b !=0 ergeben  
if (b != 0) {  
  
    x = ...
```

Auswertung eines Tests

- Wollen Quotienten a/b nur speichern wenn grösser als 0

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
// nur positive Werte sollten gespeichert werden  
// integer division kann 0 fuer a,b !=0 ergeben  
if (b != 0) {  
    if (a/b > 0) {  
        x = a/b;  
    }  
}
```

Auswertung eines Tests

- Viele if—Statements machen Programm unleserlich

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
    // nur positive Werte sollten gespeichert werden  
    //      integer division kann 0 fuer a,b !=0 ergeben  
if (b != 0) {  
    if (a/b > 0) {  
        x = a/b;  
    }  
}
```

Boolsche Ausdrücke schaffen Klarheit ...

- Viele if—Statements machen Programm unleserlich

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
    // nur positive Werte sollten gespeichert werden  
    //      integer division kann 0 fuer a,b !=0 ergeben  
  
    x = a/b;
```

Boolsche Ausdrücke schaffen Klarheit ...

- Viele if-Statements machen Programm unleserlich

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
    // nur positive Werte sollten gespeichert werden  
    //      integer division kann 0 fuer a,b !=0 ergeben  
    // (a/b) > 0  
    // b != 0  
        x = a/b;
```


Boolsche Ausdrücke schaffen Klarheit ...

- Umsetzung in booleschen Ausdruck einfach – in der Logik

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
    // nur positive Werte sollten gespeichert werden  
    //      integer division kann 0 fuer a,b !=0 ergeben  
    // (a/b) > 0  
    // b != 0  
        x = a/b;
```

Boolsche Ausdrücke schaffen Klarheit ...

- Dieser Code *führt* zu einer Fehlermeldung wenn `b == 0`:

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
// nur positive Werte sollten gespeichert werden  
// integer division kann 0 fuer a,b !=0 ergeben  
  
if ( (a/b > 0) && (b != 0) ) {  
    x = a/b;  
}
```

Reihenfolge ist wichtig

- Dieser Code *führt* zu einer Fehlermeldung wenn $b == 0$:

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
// nur positive Werte sollten gespeichert werden  
// integer division kann 0 fuer a,b !=0 ergeben  
  
if ( (a/b > 0) && (b != 0) ) {  
    x = a/b;  
}
```

Bedingte («short-circuit») Auswertung

- Dieser Code führt zu *keiner* Fehlermeldung wenn $b == 0$:

```
int a;  
int b;  
int x;    // a, b werden irgendwie gesetzt  
  
// nur positive Werte sollten gespeichert werden  
// integer division kann 0 fuer a,b !=0 ergeben  
  
if ( (b != 0) && (a/b > 0) ) {  
    x = a/b;  
}
```

Vorbesprechung Übung 2

Aufgabe 1: Fehlerbehebung

Finden und beheben Sie alle Fehler im Programm "FollerVehler.java". Eclipse hilft Ihnen dabei, indem es anzeigt, wo die Fehler sind (und eine mehr oder weniger hilfreiche Fehlermeldung dazu ausgibt), aber Sie müssen selber herausfinden, was das Problem ist und wie Sie es beheben können. Wenn Sie alle Fehler behoben haben, sollte das Programm folgendes ausgeben:

```
Hello world
```

```
Gefällt Ihnen dieses Programm?
```

```
Ich habe es selbst geschrieben.
```

Aufgabe 2: Vorwärts- und Rückwärtsschrägstriche


Ergänzen Sie das Programm in "ForwardAndBackward.java" so, dass folgendes Muster als Text ausgegeben wird:

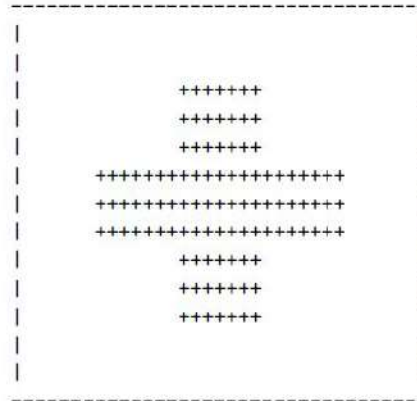
```
  \/  
  \\  
 \\\/  
 \\\///  
 ///\\\  
  ///\  
   /\
```

Das Muster muss im Ausgabefenster nicht zentriert sein, aber die Form sollte in sich so aussehen.

Aufgabe 3: Schweizerfahne (Konsole)

Statt eine Vorlage zu benutzen, schreiben Sie in dieser Aufgabe ein Programm von Grund auf selbst. Eclipse wird Ihnen allerdings etwas Schreibarbeit abnehmen.

1. Erstellen Sie eine neue Java-Datei "SwissFlag.java". Wählen Sie dafür im Menü *File* → *New* → *Class* oder klicken Sie auf das  Symbol in der Symbolleiste. Geben Sie im Dialog bei *Name* "SwissFlag" ein und drücken Sie "Finish".
2. Erstellen Sie zuerst eine leere *main*-Methode, so wie Sie es bei anderen Programmen gesehen haben. (Tipp: Sie können in Zukunft auch die Option *public static void main(String[] args)* im Dialog für neue Java-Klassen auswählen, um sich ein wenig Arbeit zu sparen.)
3. Erweitern Sie das Programm so, dass es die Schweizerfahne in der Konsole ausgibt. Die Fahne könnte ungefähr wie folgt aussehen, Sie dürfen aber auch eine grössere oder schönere Version entwerfen:



Teilen Sie das Programm in mehrere Methoden auf, welche von der `main`-Methode aufgerufen werden. Damit sorgen Sie dafür, dass weniger Wiederholungen von Code-Stücken vorkommen, was das Ändern des Programms deutlich einfacher macht.

Aufgabe 4: Berechnungen

Aufgabe 4: Berechnungen

1. Vervollständigen Sie "SharedDigit.java". In der Main-Methode sind zwei int Variablen a und b deklariert und mit einem Wert zwischen 10 und 99 (einschliesslich) initialisiert. Das Programm soll einer int Variablen r einen bestimmten Wert zuweisen. Wenn a und b eine Ziffer gemeinsam haben, dann wird r die gemeinsame Ziffer zugewiesen (wenn a und b beide Ziffern gemeinsam haben, dann kann eine beliebige Ziffer zugewiesen werden). Wenn es keine gemeinsame Ziffer gibt, dann soll -1 zugewiesen. Sie brauchen für dieses Programm keine Schleife.

Beispiele:

- Wenn a: 34 und b: 53, dann ist r: 3
- Wenn a: 10 und b: 22, dann ist r: -1
- Wenn a: 66 und b: 66, dann ist r: 6
- Wenn a: 34 und b: 34, dann ist r: 3 oder 4

Testen Sie Ihre Loesung mit a gleich 34 und b gleich 43. Was liefert Ihr Programm?

2. Vervollständigen Sie "SumPattern.java". In der Main-Methode sind drei int Variablen a, b, und c deklariert und mit irgendwelchen Werten initialisiert. Wenn die Summe von zwei der Variablen die dritte ergibt, nehmen wir an dass $a + c == b$, so soll die Methode "Moeglich. $a + c == b$ " ausgeben (wobei die Werte für a, b, und c einzusetzen sind). Wenn das nicht der Fall ist, dann soll die Methode "Unmoeglich." ausgeben.

Beispiele:

- Wenn a: 4, b: 10, c: 6, dann wird "Moeglich. $4 + 6 == 10$ " oder "Moeglich. $6 + 4 == 10$ " ausgegeben.
- Wenn a: 2, b: 12, c: 0, dann wird "Unmoeglich." ausgegeben.

3. Vervollständigen Sie "AbsoluteMax.java". In der Main-Methode sind drei int Variablen a, b, und c deklariert und mit irgendwelchen Werten initialisiert. Das Programm soll einer int Variable r den grössten absoluten Wert von a, b, und c zuweisen.

Aufgabe 5: Postconditions

Geben Sie für die folgenden Programmsegmente die Postcondition an, welche nach der Ausführung gilt, wenn zuvor die angegebene Precondition gilt. Verwenden Sie Java Syntax. Alle Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow.

1.

```
P: { x > 10 }  
S:   y = x + 5;  
Q: { ?? }
```

2.

```
P: { x == 2 }  
S:   y = x + 5;  
Q: { ?? }
```

3.

```
P: { x > 0 && z > 0 }  
S:   y = z * x  
Q: { ?? }
```

Beispiel

```
P: { x >= 3 }  
S:   z = x/2  
Q: { ?? }
```

Vorprogrammieren

Positives Maximum oder Negatives Minimum

In einer Main-Methode sind drei int Variablen a, b, und c deklariert und mit einem beliebigen Wert initialisiert. Das Programm soll einer int Variablen r einen bestimmten Wert zuweisen. Wenn der grösste Wert der Variablen a, b, und c positiv ist, dann soll r dieser Wert zugewiesen werden. Ansonsten soll r der kleinste Wert der drei Variablen zugewiesen werden.

```
public static void main(String[] args) {  
    // Aendere die Werte um verschiedene Ausfuehrungen zu testen.  
    int a = 4;  
    int b = -5;  
    int c = 3;  
  
    // TODO  
    int r;  
}
```

Kahoot