Tafelübung 01
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2020/21

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Organisatorisches

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

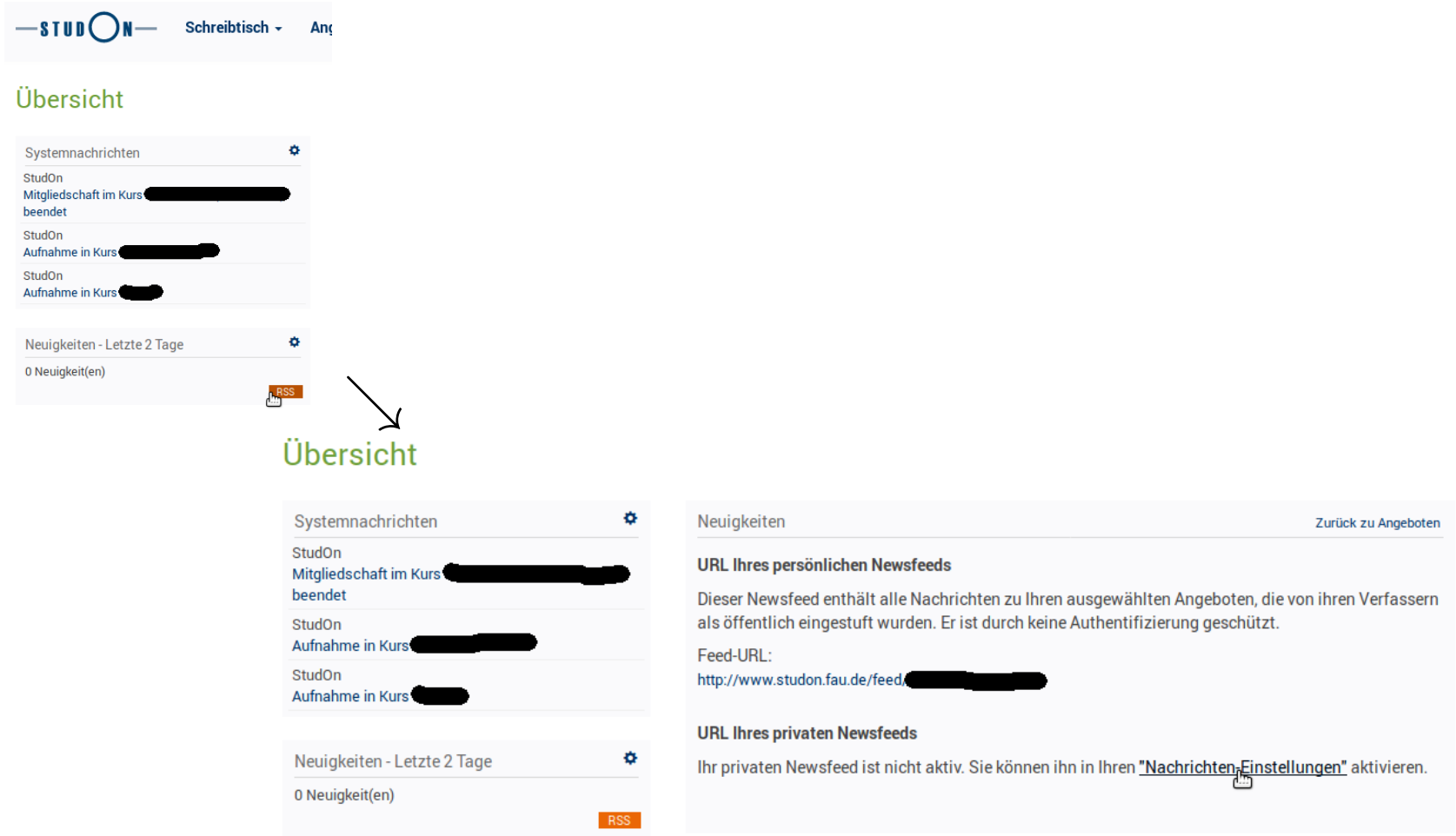
Information und Materialien

- alle Neuigkeiten und Unterlagen zu AuD:
  - StudOn-Kurs „Algorithmen und Datenstrukturen (aktuelles Semester)“
  - Kurs-Passwort: AUDFAU
- Tipp: Forum im AuD-StudOn-Kurs nutzen:
  - viel größerer Leserkreis als diverse Facebook-Gruppen!
  - auch Dozenten und Tutoren lesen mit und beantworten Fragen!
- Abgabe der Programmieraufgaben über das Exercise Submission Tool (EST): `https://est.cs.fau.de/`
- Bearbeitung/Abgabe der Theorieaufgaben über StudOn

Wichtig

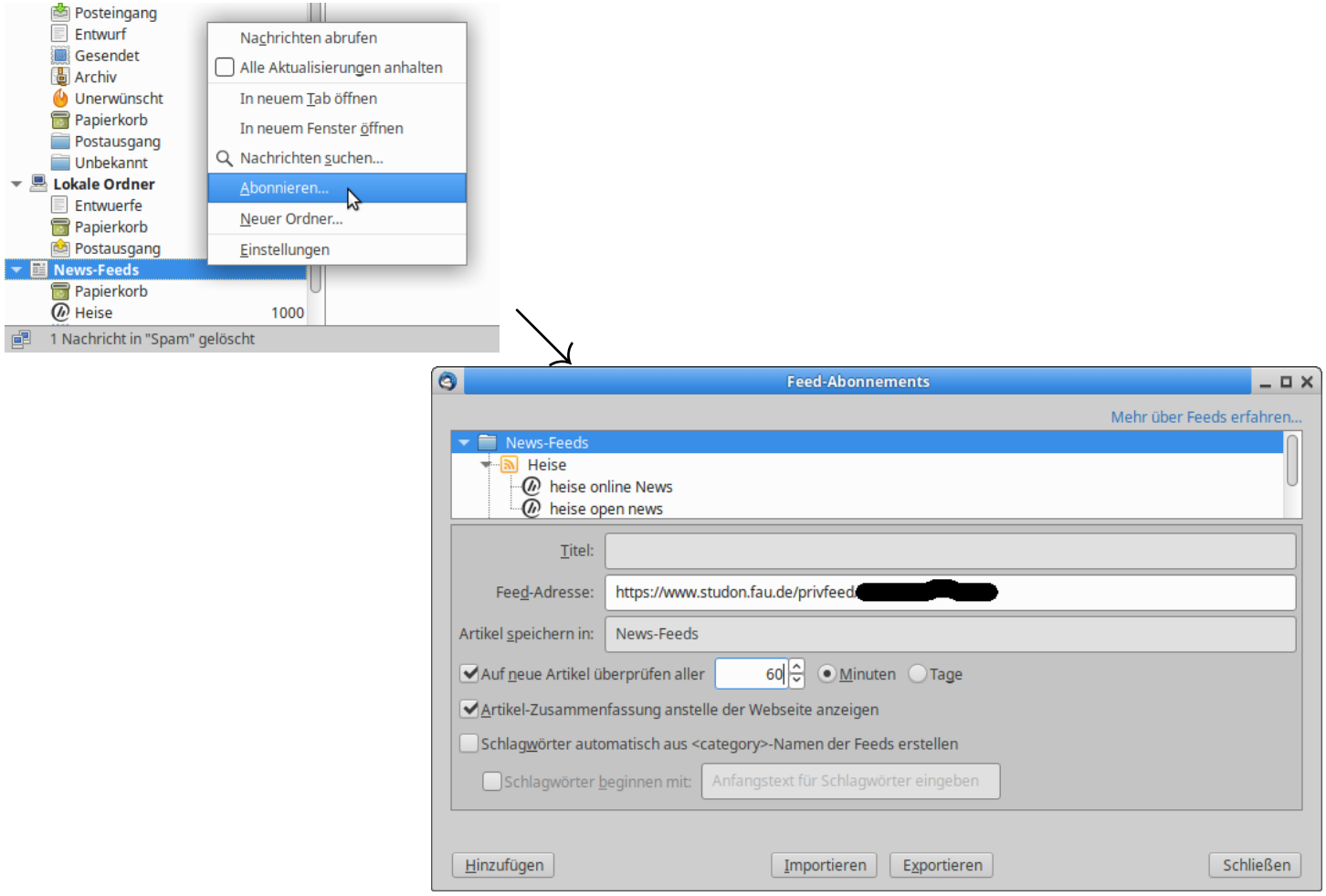
▷ Unbedingt Übungsblatt 0 (*"Wichtige Hinweise"*) und den Anhang lesen und beachten!
▷ RSS-Feed der StudOn-News abonnieren!

Neuigkeiten zeitnah: RSS-Feed abonnieren! (StudOn)



Übersicht

Neuigkeiten zeitnah: RSS-Feed abonnieren! (z.B. Thunderbird)



Plagiarismus

- Plagiieren strikt verboten!

⇝ Einzel-Aufgaben alleine bearbeiten

⇝ Gruppen-Aufgaben in 2er-Gruppen (max. 2 Pers.!) bearbeiten

- wir führen eine manuelle Überprüfung durch und setzen Software ein, um Ähnlichkeiten zwischen Abgaben zu entdecken!

Konsequenzen des Plagiierens

0 Punkte für “Spender” und “Empfänger”.

Prävention

„Sperrung“ des Home-Verzeichnisses mittels `chmod 700 ~`.

Weitere Informationen zum Thema

`https://www.ps.tf.fau.de/lehre/organisatorisches/richtlinien-zu-plagiarismus/`

Zahlensysteme

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Motivation

- für einen Menschen ist (meist) klar:
  - 100 € meint einen Betrag von einhundert Euro
  - 42 ist die Dezimalzahl Zweiundvierzig
- ein Computer arbeitet jedoch grundsätzlich „mit Nullen und Einsen“
  - 00110001 00110000 00110000 00100000 11100010 10000010 10101100 *kann* einen Betrag von einhundert Euro meinen
  - 0010 1010 *kann* die Dezimalzahl zweiundvierzig sein

Daten im Computer

- in einem Computer werden sämtliche Daten binär gespeichert
- je nach „Kontext“ werden die Daten verschieden interpretiert

Zahlen in verschiedenen Basen

Darstellung von Zahlen zu beliebigen Basen

Eine Sequenz

$$(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)_{(b)}$$

von Ziffern r_i mit der Länge l und der Basis b, $r_i \in \{0, 1, \ldots, b-1\}$, repräsentiert die vorzeichenlose, ganze Zahl

$$s = \sum_{i=0}^{l-1} (r_i \cdot b^i) \,.$$

Dabei ist b^i die Wertigkeit der Ziffer r_i. Weiter vorne stehende Ziffern haben eine höhere Wertigkeit als weiter hinten stehende *(Big Endian)*.

Angabe der Basis

Eine Zahlendarstellung muss eindeutig sein!

Wenn aus dem Kontext nicht eindeutig ersichtlich ist, um welche Basis es sich handelt, muss diese explizit angegeben werden!

Beispiele

- $1303_{(10)}$, $1303_{(8)}$
- später lernen wir noch andere Möglichkeiten für die Notation kennen

Dezimalsystem

- Dezimalsystem $\equiv$ Zahlensystem zur Basis 10
  - $r_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- unser „alltägliches“ Zahlensystem

$\rightsquigarrow$ häufig Umrechnung in dieses Zahlensystem notwendig

Beispiel $\rightsquigarrow$ Formel scheint zu stimmen ☺

$$1303_{(10)} \overset{(10)}{=} (1 \cdot 10^3) + (3 \cdot 10^2) + (0 \cdot 10^1) + (3 \cdot 10^0) = 1303$$

Binärsystem

- Binärsystem $\equiv$ Zahlensystem zur Basis 2
  - $r_i \in \{0, 1\}$
  - eine Ziffer nennen wir auch Bit (*binary digit*)
- in der Digitaltechnik verwendetes Zahlensystem
  - Form, in der Daten in einem Rechner gespeichert werden
- Möglichkeiten der Kennzeichnung:
  - $1011_{(2)}$
  - $\underline{\underline{0b}}1011$

Beispiel

$$1011_{(2)} \overset{(10)}{=} (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 11$$

Sedezimalsystem („Hexadezimalsystem“)

- Sedezimalsystem $\equiv$ Zahlensystem zur Basis 16
  - $r_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
- wir werden später sehen:
  - erlaubt kompakte Darstellung von Binärzahlen
- Möglichkeiten der Kennzeichnung:
  - $\text{BABE}_{(16)}$
  - $\underline{\underline{0x}}\text{BABE}$

Beispiel

$$\text{BABE}_{(16)} \overset{(10)}{=} (11 \cdot 16^3) + (10 \cdot 16^2) + (11 \cdot 16^1) + (14 \cdot 16^0) = 47806$$

Oktalsystem

- Oktalsystem $\equiv$ Zahlensystem zur Basis 8
  - $r_i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$
- wie bei Zahlen im Sedezimalsystem:
  - erlaubt kompakte Darstellung von Binärzahlen
- Möglichkeiten der Kennzeichnung:
  - $1303_{(8)}$
  - $\underline{0}1303$

Beispiel

$$1303_{(8)} \overset{(10)}{=} (1 \cdot 8^3) + (3 \cdot 8^2) + (0 \cdot 8^1) + (3 \cdot 8^0) = 707$$

Umrechnung: beliebiges System $\mapsto$ Dezimalsystem

Die Umrechnung von einer Zahl in einem beliebigen System in das Dezimalsystem können wir bereits ☺

Umrechnung: Dezimalsystem $\mapsto$ beliebiges System

Divisionsmethode/Modulo-Methode

Zahl N aus dem Dezimalsystem in System zur Basis b umwandeln:

- N ganzzahlig durch b teilen $\rightsquigarrow$ ganzzahliger Quotient N', Rest R
- R als Ziffer „vorne an das Ergebnis anhängen“
- falls $N' \neq 0$: mit N' nach gleichem Schema verfahren

Beispiel

N		b	Quotient N'	Rest R
4867	/	16	304	3
304	/	16	19	0
19	/	16	1	3
1	/	16	0	1

$$\rightsquigarrow 4867_{(10)} = 1303_{(16)}$$

Umrechnung: beliebiges System $\mapsto$ beliebiges System

- Umrechnung zwischen beliebigen Zahlensystemen:
  - häufig ist der „Umweg“ über das Dezimalsystem am einfachsten
  - System zur Basis $a \mapsto$ Dezimalsystem $\mapsto$ System zur Basis b
- **aber**: es gibt einen Trick bei „besonderen“ Zahlensystemen

Umrechnung „ohne Umwege“

Wenn zwei Zahlensysteme mit den Basen a und b gegeben sind, und es gilt $a^e = b$ für ein $e \in \mathbb{N}$, dann kann eine Ziffer aus dem Zahlensystem zur Basis b in e Ziffern des Zahlensystems zur Basis a umgewandelt werden.

Beispiel

- Binärsystem: Basis 2, Sedezimalsystem: Basis 16
- $2^4 = 16 \rightsquigarrow$ vier Binärziffern $\leftrightarrow$ eine Sedezimalziffer

Trick bei besonderen Zahlensystemen

Zur Erinnerung

vier Binärziffern $\leftrightarrow$ eine Sedezimalziffer

Beispiel

Umrechnung von $101010_{(2)}$ in das Sedezimalsystem:

$$\begin{array}{c|c} 0010 & 1010 \\ \updownarrow & \updownarrow \\ 2 & A \end{array}$$

$$\rightsquigarrow 101010_{(2)} = 2A_{(16)}$$

Negative Zahlen

Hinweis

Wir gehen im Folgenden von Darstellungen im Binärsystem aus. Wir können diese Darstellungen aber auch in das Sedezimalsystem oder andere Zahlensysteme umwandeln.

Wichtig

Spätestens, wenn wir auch negative Zahlen darstellen wollen, müssen wir mit Zahlen fester Länge arbeiten, d.h. die Anzahl an Bits vorgeben – insbesondere auch, weil das vorderste Bit bei den gezeigten Darstellungen das Vorzeichen angibt.

Hinweis

Im den folgenden Beispielen arbeiten wir immer mit 8 Bit langen Zahlen.

Möglichkeiten zur Darstellung negativer Zahlen

- es gibt im Wesentlichen drei verschiedene Darstellungen negativer Zahlen:
  - Vorzeichen-Betrags-Darstellung
    - vorderstes Bit gibt Vorzeichen an, restliche Bits den Betrag
  - Einerkomplement-Darstellung
    - für negative Zahlen alle Bits des Betrags invertieren
  - Zweierkomplement-Darstellung
    - für negative Zahlen alle Bits des Betrags invertieren und 1 addieren
    - diese Darstellung wird üblicherweise in Rechensystemen verwendet

Vorzeichen-Betrags-Darstellung

Vorzeichen-Betrags-Darstellung

Das vorderste Bit gibt das Vorzeichen an, wobei 0 positive Zahlen und 1 negative Zahlen kennzeichnet. Die restlichen Bits stellen den Betrag der Zahl dar.

Beispiel: ± 13

$$+13_{(10)} \mapsto \underline{0}0001101_{(2)} \mapsto 0x0D$$
$$-13_{(10)} \mapsto \underline{1}0001101_{(2)} \mapsto 0x8D$$

Probleme

Zwei Darstellungen der Null (± 0), positive und negative Zahlen müssen bei Rechnungen getrennt behandelt werden.

Einerkomplement-Darstellung

Einerkomplement-Darstellung

Positive Zahlen werden wie vorhin gezeigt dargestellt (auf führende Null achten!). Für negative Zahlen werden alle Bits der dazugehörigen positiven Zahl invertiert.

Beispiel: ± 13

$+13_{(10)} \mapsto 00001101_{(2)} \mapsto 0x0D$
$-13_{(10)} \mapsto 11110010_{(2)} \mapsto 0xF2$

Probleme

Zwei Darstellungen der Null (± 0).

Zweierkomplement-Darstellung (I)

Zweierkomplement-Darstellung

Positive Zahlen werden wie vorhin gezeigt dargestellt (auf führende Null achten!). Für negative Zahlen werden alle Bits der dazugehörigen positiven Zahl invertiert, anschließend wird 1 addiert.

Beispiel: ± 13

$$+13_{(10)} \mapsto 00001101_{(2)} \mapsto \text{0x0D}$$
$$-13_{(10)} \mapsto 11110010_{(2)} + 1 = 11110011_{(2)} \mapsto \text{0xF3}$$

Vorteile

Nur noch eine Darstellung der Null, positive und negative Zahlen können bei Rechnungen gleich behandelt werden.

Zweierkomplement-Darstellung (II)

In Java

```
byte foo = (byte)(96 + 82); // 8 Bit, vorzeichenbehaftet
System.out.format("foo = %d\n", foo);
```

Ausgabe

```
foo = -78
```

Grund

$$
\begin{array}{rcl}
0\ 1\ 1\ 0\ 0\ 0\ 0\ 0 & = & +96_{(10)} \\
+\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0 & = & +82_{(10)} \\
\hline
1\ 0\ 1\ 1\ 0\ 0\ 1\ 0 & = & -78_{(10)}
\end{array}
$$

Zeichen

- bisher: Repräsentation von Ganzzahlen im Rechner
- auch wichtig: Repräsentation von ...
  - Kommazahlen ($\rightsquigarrow$ GTI, AlgoKS)
  - Zeichen wie z.B. Buchstaben ($\rightsquigarrow$ jetzt ☺)
- auch Zeichen werden als Folge von Bits gespeichert
- verschiedene Standards, welches Zeichen zu welcher Bitfolge gehört
  - ASCII (American Standard Code for Information Interchange)
    - alphanumerische Zeichen, vor allem die im Amerikanischen übliche
    - keine Umlaute, nur wenige Sonderzeichen
  - Unicode
    - viele weitere Zeichen, darunter Umlaute, Sonderzeichen, Emoticons, ...

Beispiel

Ausschnitt aus einer ASCII-Tabelle

Buchstabe	ASCII (dezimal)	Buchstabe	ASCII (dezimal)
Leerzeichen	32	e	101
H	72	l	108
W	87	o	111
a	97	t	116

Beispiel

Geben Sie den Text an, welcher sich hinter folgenden ASCII-kodierten Zeichen verbirgt:

```
72 97 108 108 111 32 87 101 108 116
```

$\rightsquigarrow$ `Hallo Welt`

Hinweis für die Hausaufgaben

ASCII-Tabellen kann man leicht online finden (bzw. unter Linux: `man ascii`).

Eine kurze Einführung in Java

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Entwicklung in der Programmiersprache Java

- in AuD verwendete Programmiersprache: Java
  - benötigt wird das Java Development Kit (JDK)
  - im CIP vorhanden, aber auch zu Hause kostenlos nutzbar[1]
- Quelltext muss vor der Ausführung zunächst übersetzt werden
  - das Programm `javac` übersetzt den Quelltext in Java-Bytecode
- Java-Bytecode wird *nicht* nativ auf der Hardware ausgeführt
  - sondern: von der Java Virtual Machine (JVM) interpretiert
    - Start der Laufzeitumgebung zur Ausführung eines Programms mit `java`
  - Vorteil: Plattformunabhängigkeit

[1] siehe Organisatorisches → Anleitungen zur Einrichtung der Arbeitsumgebung in StudOn

Klassischer Workflow

Workflow

```
$> gedit MeinProgramm.java

... Datei bearbeiten ...

$> ls
MeinProgramm.java

$> javac MeinProgramm.java

$> ls
MeinProgramm.java   MeinProgramm.class

$> java MeinProgramm
```

Nicht vergessen...

Nach jeder Quelltext-Änderung muss das Programm neu übersetzt werden.

Aufbau einer Java-Datei

Beispiel für den Aufbau einer Java-Datei

```java
public class MeinProgramm {

    public static void main(String[] args) {
        // hier Code
    }

}
```

`main`-Methode

Die Ausführung des Programms beginnt immer in der `main`-Methode.

Wichtig

Die Datei muss genauso heißen wie die Klasse!

Fehler beim Übersetzen

- ein Programm kann nur dann übersetzt werden, wenn es den syntaktischen und semantischen Regeln von Java genügt
  - andernfalls meldet der Übersetzer einen Fehler und bricht ab
    - Fehlermeldung gibt Hinweis darauf, was kaputt ist (s.u.)
  - solche Fehler gehören dazu und passieren recht leicht $\rightsquigarrow$ keine Sorge ☺

Was tun bei einem Fehler?

Fehlermeldung lesen und versuchen, den Fehler zu beheben! ☺

Beispiele für Fehlermeldungen (I)

Strichpunkt vergessen

```
Volumen.java:4: ';' expected
        int breite = 7
                     ^
```

Was will uns der Übersetzer sagen?

- Fehler in der Datei `Volumen.java` ...
- ... in Zeile 4 ...
- ... an der gekennzeichneten Stelle (*vermutlich*...)

Unerwartetes Dateiende (geschweifte Klammer vergessen?)

```
Volumen.java:9: reached end of file while parsing
}
 ^
```

Beispiele für Fehlermeldungen (II)

Falscher Dateiname

```
Test.java:1: class Volumen is public, should be declared in a file
named Volumen.java
public class Volumen {
       ^
```

Verwendung einer nicht deklarierten Variable

```
Volumen.java:4: cannot find symbol
symbol  : variable laenge
location: class Volumen
        laenge = 5;
        ^
```

... und viele mehr ... ☺

Java-Grundlagen

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Datentypen

- Daten werden intern als Folgen von 0en und 1en gespeichert
  - *eine* 0 oder 1 wird dabei als Bit (*Binary Digit*) bezeichnet
  - eine Folge von 8 Bits wird als Byte bezeichnet
- der Datentyp einer Variable bestimmt...
  - die Größe der Variable im Speicher (Anzahl der Bits),
  - wie die gespeicherte Information zu interpretieren ist, und
  - welche Operationen auf der Variable möglich sind
- mögliche Datentypen:
  - vordefinierte Datentypen (primitive Datentypen, ...)
  - benutzerdefinierte Datentypen

Primitive Datentypen in Java (I)

- Ganzzahlen:

Typ	Größe	Wertebereich
`byte`	1 Byte	$[-128, 127]$
`short`	2 Byte	$[-32.768, 32.767]$
`int`	4 Byte	$[-2.147.483.648, 2.147.483.647]$
`long`	8 Byte	$[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]$

- Fließkommazahlen:
  - Konstanten verwenden Punkt als Dezimaltrennzeichen (z.B. `13.03`)

Typ	Größe	Genauigkeit	Wertebereich (mit Lücken)
`float`	4 Byte	einfach	$\approx \pm 3.4 \cdot 10^{38}$
`double`	8 Byte	doppelt	$\approx \pm 1.8 \cdot 10^{308}$

Primitive Datentypen in Java (II)

- Wahrheitswerte („wahr“ oder „falsch“):

Typ	Größe	Wertebereich
`boolean`	*nicht definiert*	[`true`, `false`]

- (Unicode-)Zeichen (Buchstaben, ...):

Typ	Größe	Wertebereich	Beispiel
`char`	2 Byte	$[0, 65.535]$	`'A'`

Datentypen – Beispiele und Zusammenfassung

Keyword	Größe	Werte	Literal (Bsp.)
`boolean`	1 Bit	`true` / `false`	`true` / `false`
`byte`	1 Byte (8 Bit)	$[-128; 127]$	`(byte) 5`
`short`	2 Byte (16 Bit)	$[-32768; 32767]$	`(short) 5`
`int`	4 Byte (32 Bit)	$[-2^{31}; 2^{31} - 1]$	`5`
`long`	8 Byte (64 Bit)	$[-2^{63}; 2^{63} - 1]$	`5L` / `5l`
`float`	32 Bit	32-bit IEEE 754	`5.0F` / `5.0f`
`double`[2]	64 Bit	64-bit IEEE 754	`5.0D` / `5.0d` / `5.0`
`char`	16 Bit	$[0; 65535]$	`'A'`
`String`	-	-	`"AuD"`

[2] `double` rechnet doppelt so genau wie `float`.

Arithmetische Operatoren

- für Berechnungen mit „Zahlen“ bietet Java eine Menge von Operatoren:

Operator	Bedeutung	Beispiel	Ergebnis
`+`	Addition	`13 + 3`	16
`-`	Subtraktion	`12 - 4`	8
`*`	Multiplikation	`4 * 5`	20
`/`	Division	`8 / 2`	4
`%`	Modulo (Rest)	`9 % 5`	4

Achtung: Ganzzahl-Division

Bei einer Division von ganzen Zahlen (z.B. `int`) führt Java eine Ganzzahl-Division durch, d.h. eventuelle Nachkommastellen werden abgeschnitten ($\rightsquigarrow$ `13 / 3` = 4).

Arithmetische Operatoren: Auswertungsreihenfolge

- wie in der Mathematik beachtet Java bei Berechnungen Punkt-vor-Strich
  1. „Punkt“-Operatoren: `*`, `/`, `%`
  2. „Strich“-Operatoren: `+`, `-`
- die Auswertungsreihenfolge kann durch Klammerung beeinflusst werden

Beispiele

Ausdruck	Ergebnis
`13 + 3 * 4`	25
`(13 + 3) * 4`	64

Kurz-Schreibweise: Zuweisungen mit Operation

- die Grundoperatoren können mit einer Zuweisung kombiniert werden:

Kurz-Schreibweise	entspricht[3]
`i += 5;`	`i = (T)(i + 5);`
`i -= 4;`	`i = (T)(i - 4);`
`i *= 8;`	`i = (T)(i * 8);`
`i /= 2;`	`i = (T)(i / 2);`
`i %= 6;`	`i = (T)(i % 6);`

Achtung

Es wird immer zuerst die rechte Seite vollständig ausgewertet, also:

Kurz-Schreibweise	entspricht[3]
`i *= 5 + 4;`	`i = (T)(i * (5 + 4));`

[3] dabei bezeichne `T` den Typ von `i`

Prä-/Post-Inkrement, Prä-/Post-Dekrement

- innerhalb eines beliebigen Ausdrucks kann Variable `a` verändert werden:
  - `++a`: Prä-Inkrement
    - `a` wird *zuerst* um 1 erhöht und es wird mit dem *neuen* Wert gerechnet
  - `a++`: Post-Inkrement
    - es wird mit dem *alten* Wert gerechnet und `a` *anschließend* um 1 erhöht
  - `--a`: Prä-Dekrement
    - `a` wird *zuerst* um 1 verringert und es wird mit dem *neuen* Wert gerechnet
  - `a--`: Post-Dekrement
    - es wird mit dem *alten* Wert gerechnet und `a` *anschließend* um 1 verringert
- normalerweise stehen Inkremente / Dekremente aber „alleine“: `a++;`

Prä-/Post-Inkrement, Prä-/Post-Dekrement: Beispiel

Beispiel: Prä-Inkrement

```
int a = 3;
int b = 5 * ++a;

System.out.println("a = " + a);
System.out.println("b = " + b);
```

Beispiel: Post-Inkrement

```
int a = 3;
int b = 5 * a++;

System.out.println("a = " + a);
System.out.println("b = " + b);
```

Ausgabe

```
a = 4
b = 20
```

Ausgabe

```
a = 4
b = 15
```

Vorsicht

Inkrement ist signifikant anders als:

```
int b = 5 * (a + 1); // a bleibt unverändert (meist gewünscht)
```

Vergleiche von Zahlen

- für die numerischen Datentypen gibt es die folgenden Vergleichsoperatoren:
  - das Ergebnis ist stets ein `boolean`-Wert

Operator	Bedeutung	Beispiel	Ergebnis
`==`	gleich	`13 == 3`	`false`
`!=`	ungleich	`13 != 3`	`true`
`<`	kleiner	`13 < 3`	`false`
`>`	größer	`13 > 3`	`true`
`<=`	kleiner-gleich	`13 <= 3`	`false`
`>=`	größer-gleich	`13 >= 3`	`true`

Achtung

Zuweisung mit `=`, Vergleich mit `==`.

Beispiel zum Vergleichen von Zahlen

Frage

Handelt es sich bei einem Rechteck mit gegebenen Kantenlängen um ein Quadrat?

Lösung

Ein Rechteck ist ein Quadrat, wenn seine Breite und seine Höhe gleich sind.

In Java

```
int breite = 5;
int hoehe = 6;

boolean istQuadrat = (breite == hoehe); // = false
```

Logische Operatoren

- auch für Wahrheitswerte (`boolean`) bietet Java eine Menge von Operatoren
  - verknüpfen zwei boolesche Werte zu einem neuen booleschen Wert

$\rightsquigarrow$ komplexe/zusammengesetzte boolesche Ausdrücke

Beispiel 1: Logisches „Und“

Die Verwendung des Systems wird nur gewährt, wenn...

- der Benutzername stimmt **UND**
- das Passwort stimmt.

Beispiel 2: Logisches „Oder“

Eine Ware kann nur gekauft werden, wenn...

- genügend Bargeld vorhanden ist **ODER**
- genügend Geld auf dem Konto vorhanden ist.

Logisches „Und“

- logisches „Und“ $\rightsquigarrow$ &&-Operator
  - *wahr*, wenn beide Operanden *wahr* sind
  - *falsch*, wenn mindestens ein Operand *falsch* ist

Wahrheitstabelle

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

Beispiel

```
boolean verwendungErlaubt = (benutzernameKorrekt && passwortKorrekt);
```

Logisches „Oder“

- logisches „Oder“ $\rightsquigarrow$ `||`-Operator
  - *wahr*, wenn mindestens ein Operand *wahr* ist
  - *falsch*, wenn beide Operanden *falsch* sind

Wahrheitstabelle

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

Beispiel

```
boolean kaufMoeglich = (genuegendBargeld || genuegendAufDemKonto);
```

„Exklusives Oder“

- „exklusives Oder“ $\rightsquigarrow$ $\wedge$-Operator
  - *wahr*, wenn *genau* ein Operand *wahr* ist
  - *falsch*, wenn *beide* Operanden *wahr* oder *falsch* sind

Wahrheitstabelle

A	B	A $\wedge$ B
false	false	false
false	true	true
true	false	true
true	true	false

Achtung!

Ein exklusives Oder wird manchmal fälschlicherweise für ein „hoch“ gehalten:

```
int a = 3^2; //berechnet *nicht* "3 hoch 2" sondern bitweises XOR (= 1)
```

Logische Negation

- logische Negation $\rightsquigarrow$!-Operator
  - einstelliger Operator (man sagt auch: unärer Operator), d.h. nur ein Operand
  - „dreht den Wahrheitswert um“

Wahrheitstabelle

A	!A
false	true
true	false

Beispiel

```
boolean istKeinQuadrat = !(istQuadrat);
```

Logische und arithmetische Operatoren: Auswertungsreihenfolge (I)

- Auswertungsreihenfolge:
  1. „Punkt“-Operatoren: `*`, `/`, `%`
  2. „Strich“-Operatoren: `+`, `-`
  3. Vergleichs-Operatoren: `==`, `!=`, `<`, `<=`, `>`, `>=`
  4. logisches „Und“: `&&`
  5. logisches „Oder“: `||`

- mit Klammern kann die Auswertungsreihenfolge beeinflusst werden
  - Tipp: lieber „zu viele“ Klammern verwenden als „zu wenige“...

Logische Operatoren: Auswertungsreihenfolge (II)

Natürlichsprachliche Anforderung

Eine Ware kann nur dann gekauft werden, wenn diese auf Lager ist *und* wenn genügend Bargeld *oder* genügend Geld auf dem Konto vorhanden ist.

Falsch (Wieso?)

```
boolean kaufMoeglich =
  aufLager && genuegendBargeld || genuegendAufDemKonto;
```

Richtig

```
boolean kaufMoeglich =
  aufLager && (genuegendBargeld || genuegendAufDemKonto);
```

Logische Operatoren: Kurzschlusssemantik

- Kurzschlusssemantik:
  - && und || brechen die Auswertung ab, sobald das Ergebnis feststeht
  - macht insb. dann einen Unterschied, wenn Operanden Seiteneffekte haben
    - Beispiel: rechter Operand beinhaltet einen Funktionsaufruf; auf Grund der Kurzschlusssemantik wird dieser evtl. gar nicht ausgeführt

Beispiel

```
boolean a = false;
boolean b = true;
boolean c = a && b; // da a schon false ist, muss b gar nicht
                    // angeschaut werden (c schon sicher false)
boolean d = b || a; // da b schon true ist, muss a gar nicht
                    // angeschaut werden (d schon sicher true)
```

Bitoperatoren (I)

- zur Erinnerung: sämtliche Daten werden im Computer binär gespeichert.
- manchmal möchte man explizit in dieser binären Darstellung rechnen

Bitoperatoren in Java

- bitweises unäres „not“: ~
  - invertiert alle Bits
  - Beispiel: $\sim 01001100_{(2)} = 10110011_{(2)}$
- bitweises „and“: &
  - Ergebnisbit gesetzt $\Leftrightarrow$ Bit in beiden Operanden gesetzt
  - Beispiel: $1100_{(2)}$ & $1010_{(2)} = 1000_{(2)}$
- bitweises „or“: |
  - Ergebnisbit gesetzt $\Leftrightarrow$ Bit in mindestens einem Operanden gesetzt
  - Beispiel: $1100_{(2)}$ | $1010_{(2)} = 1110_{(2)}$

Bitoperatoren (II)

Bitoperatoren in Java (Forts.)

- bitweises „xor“: ^
  - Ergebnisbit gesetzt $\Leftrightarrow$ Bit in genaum einem Operanden gesetzt
  - Beispiel: $1100_{(2)}$ ^ $1010_{(2)}$ = $0110_{(2)}$
- „left shift“: $<<$
  - es werden Nullen von rechts eingeschoben
  - überflüssige Ziffern fallen links weg
  - Beispiel (feste Länge 8 Bit): $01001101_{(2)} << 2_{(10)} = 00110100_{(2)}$
- „right shift“: $>>$, $>>>$
  - es werden Bits von links eingeschoben
    - bei $>>>$ Nullen
    - bei $>>$ Nullen oder Einsen, so dass sich das Vorzeichen nicht ändert
  - überflüssige Ziffern fallen rechts weg
  - Beispiel (feste Länge 8 Bit): $01001101_{(2)} >> 2_{(10)} = 00010011_{(2)}$

Bitmasken

- oft möchte man bestimmte Bits setzen, löschen oder prüfen
  - wiederkehrendes Problem $\Rightarrow$ selbe Lösung wiederverwenden

Bitmasken

Bit n von `eingabe` soll gesetzt/gelöscht/geprüft werden. Dabei ist Bit 0 ganz rechts und Bit 31 das höchstwertige Bit eines `int`s.

- erster Schritt: Variable erstellen, in der nur Bit n gesetzt ist
  - `int bitN = 1 << n;`
- Bit n setzen:
  - `eingabe |= bitN;`
- Bit n löschen:
  - `eingabe &= ~bitN;`
- Prüfen, ob Bit n gesetzt:
  - `if ((eingabe & bitN) == bitN) { /* bit ist gesetzt */}`

Testen mit JUnit

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

JUnit

- JUnit $\equiv$ Framework zum Testen von Java-Programmen
  - genauer: *Unit Testing* (Testen von einzelnen Software-Komponenten)
- Tests können automatisiert ausgeführt werden

$\rightsquigarrow$ Tests häufig starten

$\rightsquigarrow$ neue Fehler fallen schnell auf

- IDEs (z.B. IntelliJ) bieten häufig besondere Unterstützung (siehe unten)

JUnit: Beispiel (I)

Folgender Code soll getestet werden:

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return laenge + breite * 2; // ob das wohl so stimmt?
  }
}
```

- zum Testen könnte man theoretisch einfach eine `main`-Methode schreiben
  - Nachteil: man müsste den Test auf Korrektheit selbst implementieren

$\rightsquigarrow$ deshalb: JUnit-Testfall

JUnit: Beispiel (II)

Erster JUnit-Testfall

```
import static org.junit.Assert.*; // erforderliche Imports
import org.junit.Test; // werden von IntelliJ automatisch erzeugt

public class RechteckTest { // Test für Klasse Rechteck

  @Test // Hinweis an JUnit: diese Methode ist ein Test
  public void testeUmfang_0x0() {
    // Methode ist immer "public void" (kein static!)

    int umfang = Rechteck.umfang(0, 0);
      // Methodenaufruf: Methode umfang der Klasse Rechteck
      // mit Parameter laenge = 0 und breite = 0

    assertEquals("Umfang ist nicht 0", 0, umfang);
      // prüft, ob erwarteter Wert in Variable umfang steht
  }
}
```

JUnit: @Test

@Test

- `@Test` ist eine Annotation
  - kennzeichnet die nachfolgende Methode als JUnit-Test
- kann (optional) Parameter erhalten:
  - `@Test(timeout=42)`
    $\Rightarrow$ ist der Test nach 42 ms noch nicht fertig, bricht er ab (*Timeout*)
  - `@Test(expected=NullPointerException.class)`
    $\Rightarrow$ damit der Test erfolgreich ist, muss obige Exception geworfen werden
    (mehr zu Exceptions in ein paar Wochen...)

JUnit: Asserts (I)

assertEquals

- Methode zum Prüfen, ob zwei Variablen den gleichen Wert haben
- hat drei Parameter:
  1. Fehlermeldung (als `String`)
     - wird angezeigt, wenn Überprüfung fehlschlägt
     - dieser Parameter ist optional
  2. `expected`-Wert
     - (nahezu) beliebiger Typ
     - Wert, der *eigentlich* erwartet wird
  3. `actual`-Wert
     - selber Typ wie `expected`
     - tatsächlicher Wert, der mit `expected` verglichen wird

JUnit: Asserts (II)

Im Beispiel...

```
int umfang = Rechteck.umfang(0, 0);
assertEquals("Umfang ist nicht 0", 0, umfang);
```

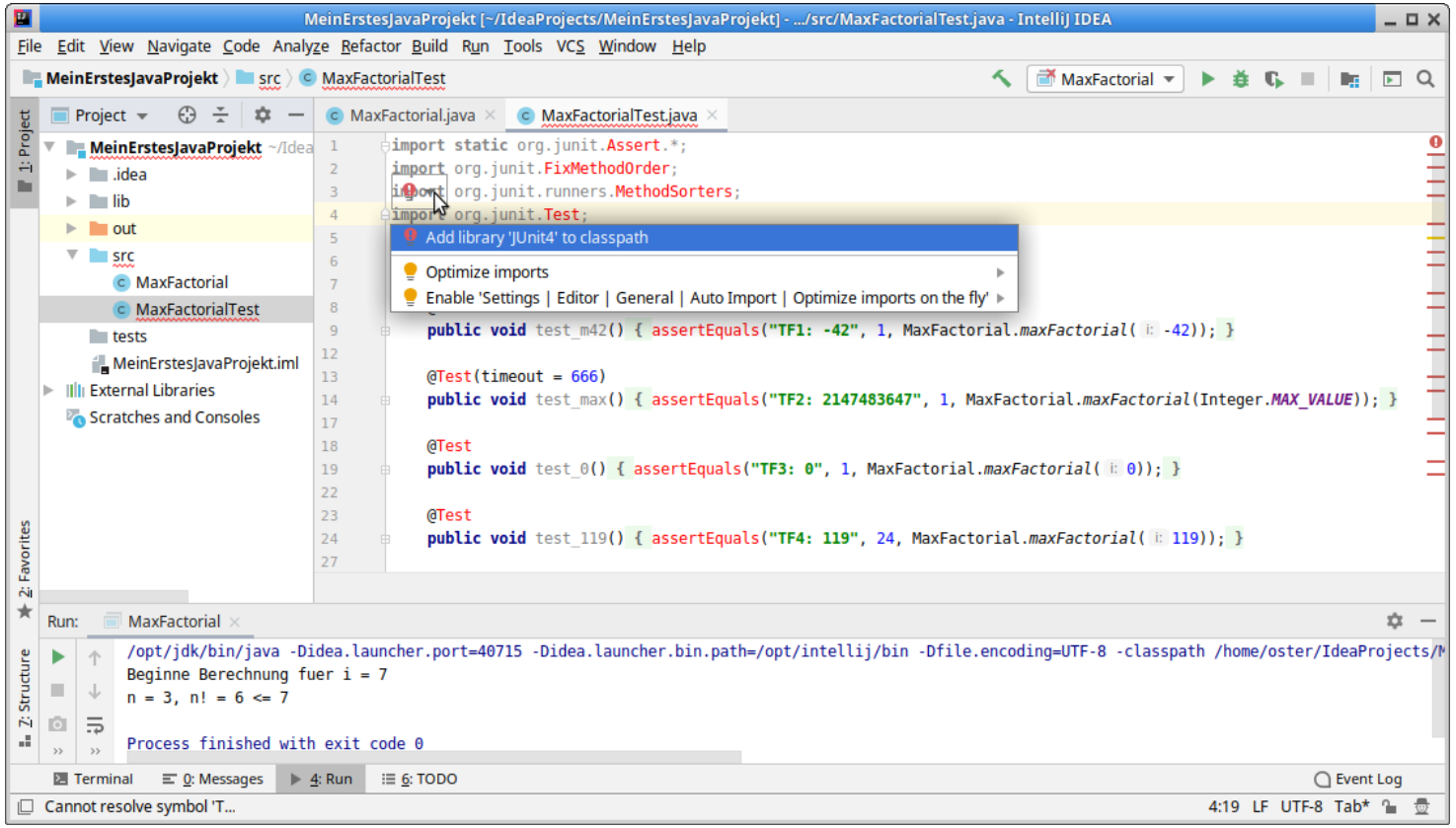
- Umfang eines 0×0-Rechtecks sollte 0 sein $\rightsquigarrow$ 2. Parameter ist 0
- tatsächlicher Wert ist in `umfang` gespeichert $\rightsquigarrow$ 3. Parameter ist `umfang`
- falls `umfang` $\neq 0$ ist, wird „Umfang ist nicht 0“ ausgegeben $\rightsquigarrow$ 1. Parameter

Weitere Assert-Methoden

- `assertNotNull` zur Prüfung, ob ein `String`/Array/... den Wert `null` hat
- `assertArrayEquals` zur Prüfung, ob zwei Arrays gleichen Inhalt haben
- `assertTrue`/`assertFalse` zur Prüfung, ob ein `boolean` den Wert `true`/`false` hat
- Details siehe `http://junit.org/apidocs/org/junit/Assert.html`

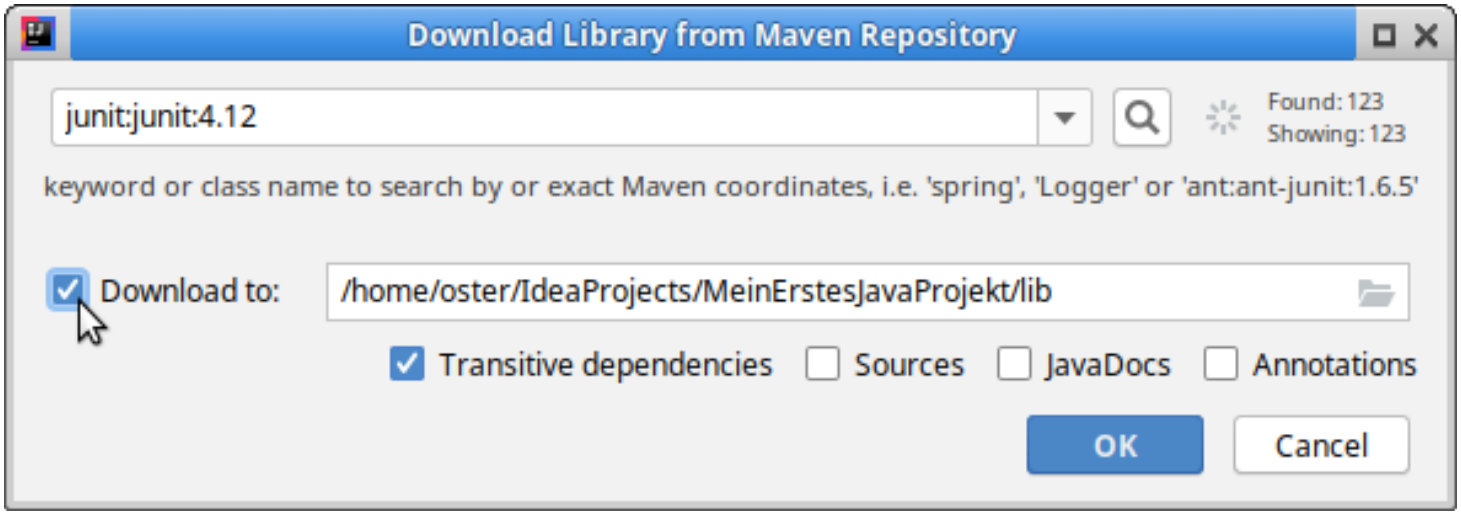
Einschub: JUnit in IntelliJ (I)

- JUnit-Lib (einmalig) zum Klassenpfad des Projekts hinzufügen lassen . . .



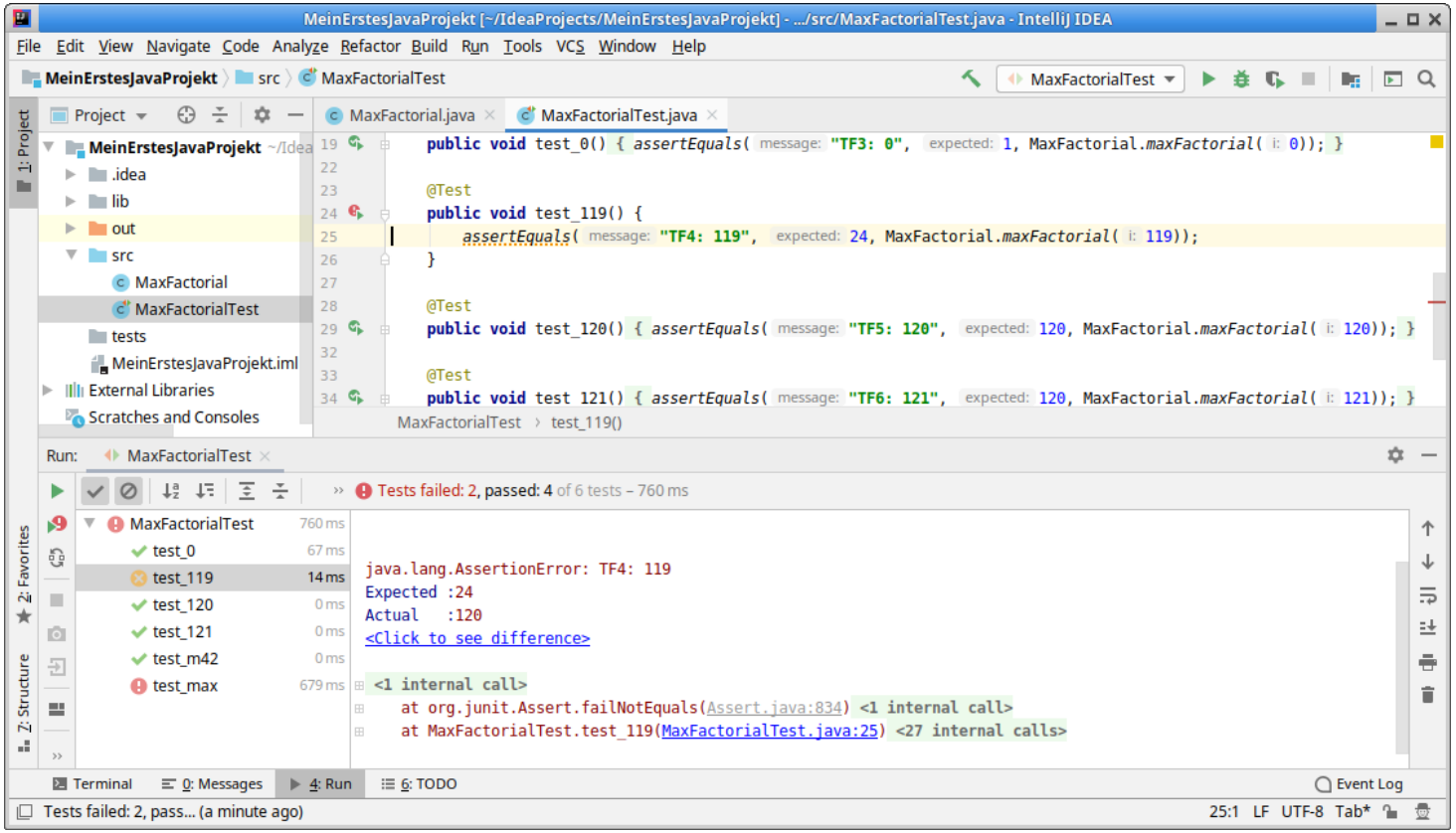
Einschub: JUnit in IntelliJ (II)

- Falls JUnit-Lib noch nicht lokal verfügbar ⇝ jetzt besorgen lassen . . .

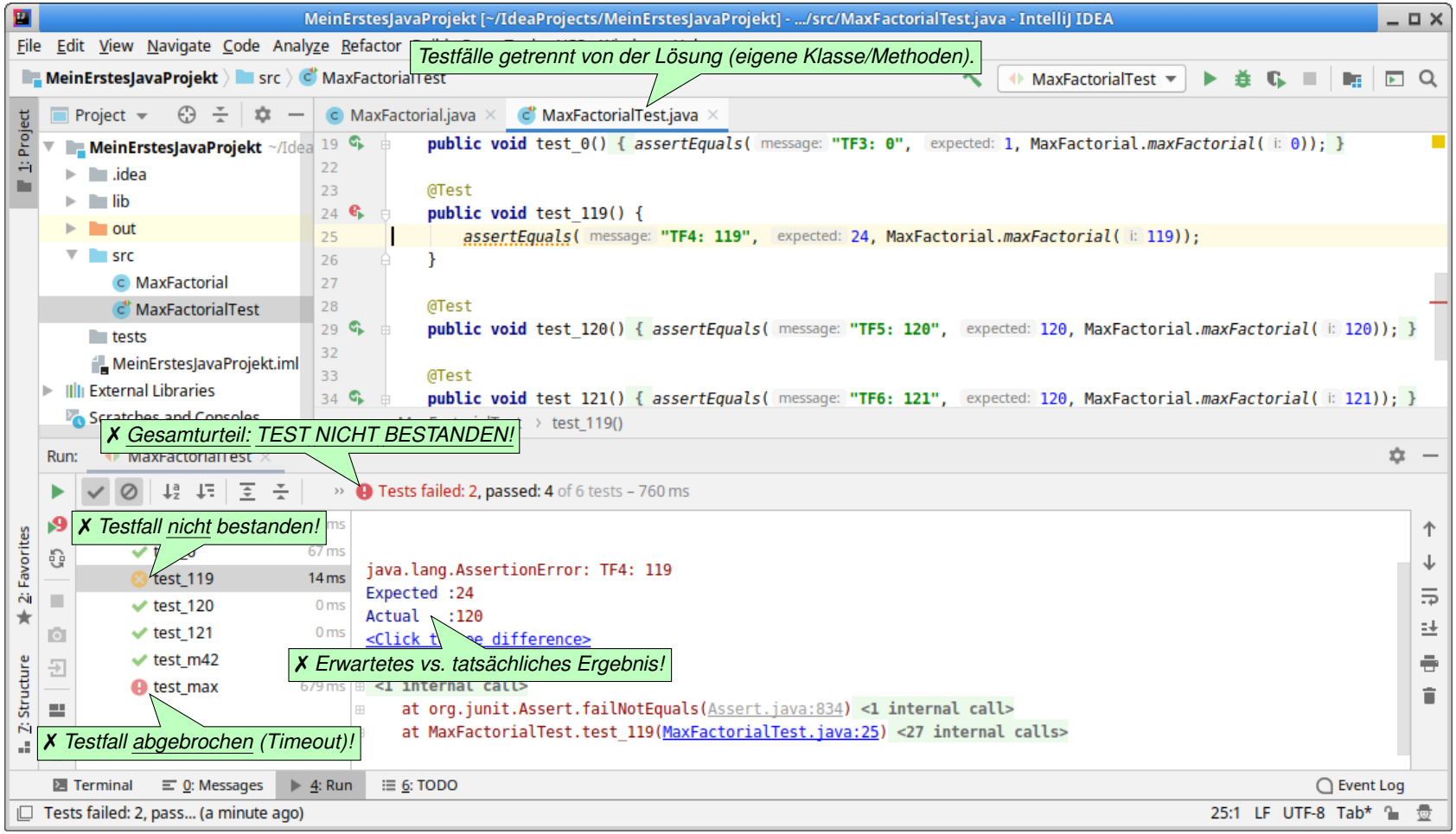


Einschub: JUnit in IntelliJ (III)

- Testklasse auswählen und auf „`Play`“ klicken ⇝ JUnit-Ansicht öffnet sich:



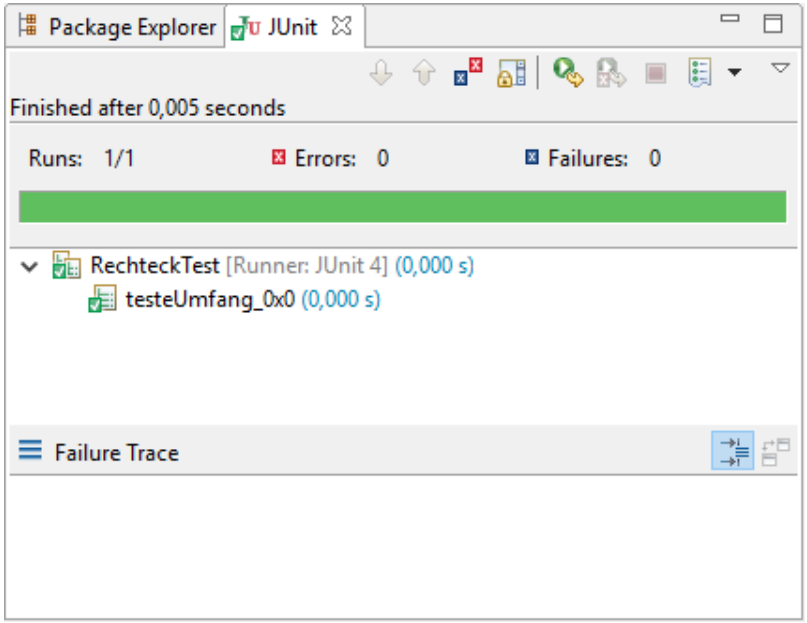
Testfälle ausführen und Ergebnisse interpretieren



Einschub: JUnit in Eclipse (I)

- Testklasse auswählen und auf „Run“ klicken
  - ggf. „`Run as JUnit Test`“ (oder ähnliches) auswählen

⇝ JUnit-Ansicht öffnet sich:



Einschub: JUnit in Eclipse (II)

- Mögliche Symbole:
  -  Test hat keine Fehler gefunden und ist erfolgreich durchgelaufen
  -  Test ist fehlgeschlagen, vermutlich ein Assert nicht erfolgreich
  -  Ausführung des Tests unterbrochen (Exception oder Timeout)

JUnit: Beispiel (III)

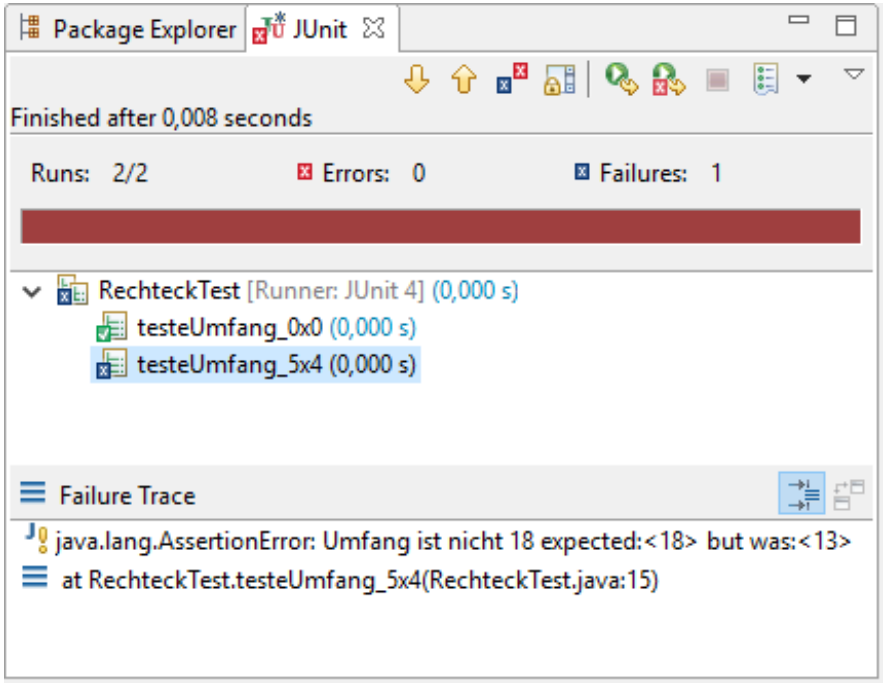
- heißt das jetzt, dass unser Beispielcode von vorhin korrekt funktioniert?
  - nein!
  - das war nur ein Test mit nur einem möglichen Eingabepaar
  - um alle Fehler zu finden, müsste man alle möglichen Eingaben testen
    - normalerweise ist das nicht möglich!
    - aber: weitere Tests schaden nicht

Weiterer Testfall

```
@Test
public void testeUmfang_5x4() { // neuer Test mit neuem Namen
  int umfang = Rechteck.umfang(5, 4);
  assertEquals("Umfang ist nicht 18", 18, umfang);
}
```

JUnit: Beispiel (IV)

- das passiert beim Ausführen:



JUnit: Beispiel (V)

- der neue Test ist fehlgeschlagen!
- im „Failure Trace“ kann man nachlesen, was passiert ist:
  - `java.lang.AssertionError` ist die Fehlerart (vorerst egal)
  - `Umfang ist nicht 18 expected <18> but was <13>`
    - unsere Fehlermeldung
    - zeigt auch den tatsächlichen Wert der Variable (hier: 13)
  - `at RechteckTest.testeUmfang_5x4(RechteckTest.java:15)`
    - Zeile, in der der Test fehlgeschlagen ist
    - auf diesen Eintrag kann man drauf klicken
      - IDE springt dann automatisch an die Fehlerstelle

  $\rightsquigarrow$ erleichtert Fehlerlokalisierung und -behebung

JUnit: Beispiel (VI)

Zurück zum Code

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return laenge + breite * 2; // ob das wohl so stimmt?
  }
}
```

Warum kommt bei `5 + 4 * 2` als Ergebnis `13` heraus?

- Punkt vor Strich: `5 + (4 * 2) = 5 + 8 = 13`
- wir wollen eigentlich: `(5 + 4) * 2 = 9 * 2 = 18`
- also: Code muss angepasst und Fehler behoben werden

JUnit: Beispiel (VII)

Angepasster Code

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return (laenge+breite) * 2; // sollte jetzt so stimmen
  }
}
```

Tipp für die Übungsaufgaben

- die öffentlichen Testcases testen *nicht* immer die gesamte Funktionalität ab
  - z.B. könnte nur testeUmfang_0x0() enthalten sein
- eigene Tests schreiben kostet oft nicht viel Zeit
- Fehler lassen sich damit oft schon vor Abgabeende finden

Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT