TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 12:
Korrekte Software

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

(In)korrektheit auf den einzelnen Abstraktionsebenen

Bei Korrektheitsbetrachtungen ist essentiell, die einzelnen beteiligten Abstraktionsebenen auseinanderzuhalten. Auf jeder Abstraktionsebene ist Korrektheit ein Thema für sich.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

Spezifikatorische Ebene

Logische Ebene

Semantische Ebene

Syntaktische Ebene

Lexikalische Ebene

Bei der Entwicklung von Software sind fünf verschiedene Abstraktionsebenen wichtig, von denen man auf die Programmieraufgabe blicken kann – und auf denen man jeweils sehr spezifisch richtig oder falsch programmieren kann.

Wir betrachten auf jeder Ebene in erster Linie typische Fehler, die häufig gemacht werden. Das ist sicherlich instruktiver als die Abhandlung von formalen Definitionen.

Wir gehen dabei von unten nach oben.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechtschreibung:

wile	....	while
For	....	for
stetic	....	static

Zuerst die unterste, die lexikalische Ebene. Aus der Betrachtung natürlicher Sprachen ist Ihnen der Begriff „Rechtschreibung“ dafür vertraut.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
    ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Die Regeln auf lexikalischer und syntaktischer Ebene können formalisiert werden. Wir sehen uns das am Beispiel Identifier einmal exemplarisch an.

Es gibt verschiedene Schreibweisen für diese formalen Regelwerke, wir verwenden hier eine gängige.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
    ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Mit diesem Zeichen werden Sprachkonstrukte formal definiert. Das Definiendum, also der Name des zu definierenden Konstrukts, steht links von diesem Zeichen, das Definiens, also der definierende Ausdruck, steht rechts.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Wird ein lexikalisches oder syntaktisches Konstrukt für die Definition eines anderen syntaktischen Konstrukts verwendet, dann wird sein Name in doppelte spitze Klammern gesetzt, also Kleiner-Größer-Zeichen.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Und natürlich muss ein so verwendetes lexikalisches oder syntatktisches Konstrukt auch genau einmal links stehen, wo es dann definiert wird.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
    ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Der vertikale Strich trennt verschiedene Alternativen. Zum Beispiel ist ein ident-char *entweder* ein letter *oder* ein digit *oder* ein Unterstrich *oder* ein Dollarzeichen.

***Erinnerung*: Wie Sie aus dem Abschnitt zu Identifiern in Kapitel 01a wissen, sind das genau die Zeichen, aus denen in Java Identifier gebildet werden dürfen.**

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
    ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Diese Zeichen sind wörtlich so zu verstehen wie sie dastehen. Ein digit, also eine Ziffer, kann somit eine 0 oder eine 9 sein; ein letter, also ein Buchstabe, kann ein kleines oder großes A oder Z sein.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
    ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Um nicht mühselig die Bereiche von a bis z und von 0 bis 9 als eine große Zahl von Alternativen mit vertikalen Strichen schreiben zu müssen, kürzen wir Aufzählungen von Alternativen mit drei Punkten ab, vorausgesetzt, es ist wie hier eindeutig, wofür die drei Punkte stehen.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Das Epsilon steht für das leere Wort, englisch empty für leer. Eine ident-char-list ist also ein syntaktisches Konstrukt, das auch leer sein kann.

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
ε | <<ident-char>> <<ident-char-list>>

letter-ext ::= <<letter>> | _ | $

ident-char ::= <<letter-ext>> | <<digit>>

letter ::= a ... z | A ... Z

digit ::= 0 ... 9

Jetzt sind wir soweit, die Definition eines Identifiers schrittweise durchzugehen. Die erste Zeile besagt, dass ein Identifier aus einem ersten Zeichen gefolgt von einer Liste von weiteren Zeichen besteht, wobei diese Liste auch leer sein kann. Das erste Zeichen ist ein Buchstabe oder der Underscore oder das Dollarzeichen, die weiteren Zeichen können auch Ziffern sein.

***Anmerkung*: Für die Menge aller Buchstaben plus Underscore und Dollarzeichen hat der Autor dieser Folien die Abkürzung letter-ext gewählt – ext für extended.**

Lexikalische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

identifier ::= <<letter-ext>> <<ident-char-list>>

ident-char-list ::=
ε | <<ident-char>> <<ident-char-list>>

c3_Po ← 3_Po ← _Po ← Po ← o ← ε

Ein kleines Beispiel für die Ableitung eines Identifiers aus diesen Syntaxregeln: Da das erste Zeichen von c3_Po ein Buchstabe (letter) ist, ist c3_Po ein Identifier genau dann, wenn 3_Po eine Liste erlaubter Zeichen (ident-char-list) ist. Da 3 ein erlaubtes Zeichen ist, ist das der Fall, wenn _Po eine ident-char-list ist. Das ist wiederum der Fall, wenn Po eine ident-char-list ist, das wiederum, wenn o das ist, und das wiederum, wenn das leere Wort eine ident-char-list ist. Und das ist ja auch der Fall, also ist im Umkehrschluss c3_Po ein korrekt gebildeter Identifier.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrekte Klammersetzung

((){}…)…[…]… {…[… {…}…]…}) … {…[…]…(…)…}

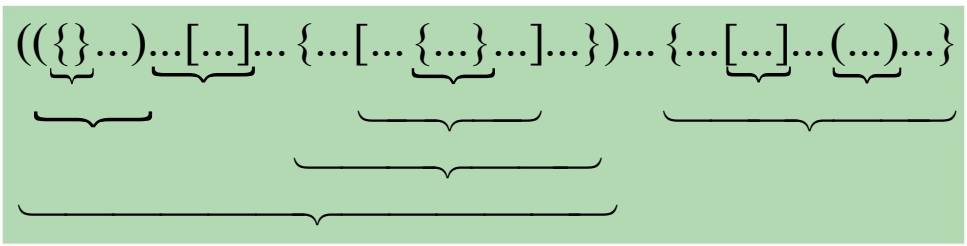
Was Sie bei natürlichen Sprachen unter Begriffen wie Satzlehre als Teil der Grammatik kennen, ist bei Programmiersprachen die Syntax, zumindest so ungefähr.

Ein wichtiger Teil der Syntax bei Programmiersprachen ist die Klammersetzung. Die Punkte deuten an, dass dazwischen noch andere Dinge stehen. Dies ist eine syntaktisch korrekte Klammersetzung. Ob der ganze Code mit diesen Klammern und den ausgelassenen Teilen Sinn macht, ist auf der syntaktischen Ebene irrelevant, das ist erst auf höheren Ebenen wichtig. Warum ist diese Klammersetzung syntaktisch korrekt?

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrekte Klammersetzung



Diese Klammersetzung ist deshalb syntaktisch korrekt, weil sie zwei einfachen Regeln folgt: Erstens gehört zu jeder öffnenden Klammer genau eine nachfolgende schließende Klammer, und umgekehrt gehört zu jeder schließenden Klammer eine vorangehende öffnende. Zweitens sind zwei Klammerpaare entweder nacheinander oder ineinander platziert, niemals teilweise überlappend.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkorrekte Klammersetzung

(...[...)...]...{...}...}...(...[...]

Um genau zu verstehen, was das heißt, schauen wir uns eine Klammersetzung an, die gegen diese beiden Regeln verstößt.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkorrekte Klammersetzung

(...[...)...]...{...}...}...(...[...]

Diese beiden Klammerpaare sind weder nacheinander noch ineinander platziert, sondern überlappen sich teilweise.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkorrekte Klammersetzung

(…[…)…]…{…}…}…(…[…]

Zur zweiten schließenden Klammer im roten Bereich findet sich nirgendwo eine zugehörige öffnende Klammer.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkorrekte Klammersetzung

(...[...)...]...{...}...}...(...[...]

Und hier fehlt zur runden öffnenden Klammer die zugehörige schließende Klammer.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntaktische Konstrukte

for (... ; ... ; ...)

while (...)

do ... while (...)

Syntaktische Konstrukte müssen gemäß der vorgegebenen Struktur gebildet werden. Hier nur drei ausgewählte Beispiele, die Köpfe bei den drei Schleifenarten in Java.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntaktische Konstrukte

for (... ; ... ; ...)

while (...)

do ... while (...)

Die for-Schleife wird mit Schlüsselwort for eingeleitet, zwingend gefolgt von einem Klammerausdruck, in dem exakt zwei Semikolons enthalten sind.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntaktische Konstrukte

for (... ; ... ; ...)

while (...)

do ... while (...)

Bei der while-Schleife kommt ein Klammerausdruck ohne Semikolons nach dem Schlüsselwort while.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntaktische Konstrukte

for (... ; ... ; ...)

while (...)

do ... while (...)

Und bei der do-while-Schleife ist die Syntax noch etwas komplizierter: zuerst Schlüsselwort do, dann der Schleifenrumpf, schließlich while und ein Klammerausdruck.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
statement ::= <<simple-statement>> ;
            | <<compound-statement>>
            | <<if-statement>>
            | <<switch-statement>>
            | <<while-loop>> | <<do-while-loop>>
            | <<for-loop>> | <<break-statement>>
            | <<continue-statement>> | .....
```

Auch auf syntaktischer Ebene kann man die grundlegenden Regeln formalisieren. Wir betrachten als Beispiel die Regeln zur Bildung von Anweisungen, und auch diese nur ausschnittweise.

Es gibt ja verschiedene Arten von Anweisungen. Daher ist der erste Schritt, eine Anweisung, also englisch statement, durch Aufzählungen der einzelnen Anweisungsarten zu definieren. Danach werden dann die einzelnen Anweisungsarten definiert.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
statement ::= <<simple-statement>> ;
            | <<compound-statement>>
            | <<if-statement>>
            | <<switch-statement>>
            | <<while-loop>> | <<do-while-loop>>
            | <<for-loop>> | <<break-statement>>
            | <<continue-statement>> | .....
```

Auch hier beginnt eine Regel mit dem Definiendum, also dem zu definierenden Wort.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
statement ::= <<simple-statement>> ;
            | <<compound-statement>>
            | <<if-statement>>
            | <<switch-statement>>
            | <<while-loop>> | <<do-while-loop>>
            | <<for-loop>> | <<break-statement>>
            | <<continue-statement>> | .....
```

Wie wir schon gesehen haben, werden die Alternativen in der hier verwendeten formalen Schreibweise durch vertikale Striche voneinander getrennt ...

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
statement ::= <<simple-statement>> ;
            | <<compound-statement>>
            | <<if-statement>>
            | <<switch-statement>>
            | <<while-loop>> | <<do-while-loop>>
            | <<for-loop>> | <<break-statement>>
            | <<continue-statement>> | .....
```

... und jeder Begriff im Definiens, der selbst durch eine Regel zu definieren ist, wird in doppelte spitze Klammern gesetzt.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
statement ::= <<simple-statement>> ;
            | <<compound-statement>>
            | <<if-statement>>
            | <<switch-statement>>
            | <<while-loop>> | <<do-while-loop>>
            | <<for-loop>> | <<break-statement>>
            | <<continue-statement>> | .....
```

Dass an dieser Stelle ein Semikolon steht, spezifiziert die allgemeine Regel, die Sie schon von Anfang an in der FOP gesehen haben: Hinter jede einfache Anweisung kommt ein Semikolon.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
statement ::= <<simple-statement>> ;
            | <<compound-statement>>
            | <<if-statement>>
            | <<switch-statement>>
            | <<while-loop>> | <<do-while-loop>>
            | <<for-loop>> | <<break-statement>>
            | <<continue-statement>> | .....
```

Diese Aufzählung von Anweisungsarten ist nicht vollständig; weitere Arten von Anweisungen lassen wir hier aus.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

compound-statement
::= { <<statement-sequence>> }

statement-sequence
::= ε
| <<statement>> <<statement-sequence>>

Als nächstes müssen wir die einzelnen Anweisungsarten definieren.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

compound-statement
::= { <<statement-sequence>> }

statement-sequence
::= ε
| <<statement>> <<statement-sequence>>

Eine Blockanweisung, englisch compound statement, ist eine Sequenz von Anweisungen in geschweiften Klammern.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

compound-statement
::= { <<statement-sequence>> }

statement-sequence
::= ε
| <<statement>> <<statement-sequence>>

Eine solche Sequenz von Anweisungen kann auch leer sein. Das Epsilon zeigt auch hier wieder an, dass das Definiendum auch aus gar nichts bestehen kann.

Dies ist ein gutes Beispiel dafür, dass man durch solche formalen Regelwerke oft

Detailfragen klären kann. Eine nicht selten gestellte Detailfrage ist eben, ob ein Anweisungsblock auch leer sein kann, also ob ein leeres geschweiftes Klammerpaar auch ein Anweisungsblock ist. Und farbig unterlegt sehen wir hier die Antwort: ja.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

compound-statement
::= { <<statement-sequence>> }

statement-sequence
::= ε
| <<statement>> <<statement-sequence>>

Erinnerung: Wir hatten bei Identifiern im Abschnitt zur lexikalischen Ebene weiter vorne schon ein erstes Beispiel einer rekursiven Definitionsregel gesehen.

Eine Anweisungssequenz ist entweder leer – das ist der Rekursionsanker – oder es ist eine Anweisung gefolgt von einer Anweisungssequenz. Jede Sequenz von Anweisungen, egal ob sie null, eine oder mehrere Anweisungen enthält, passt auf dieses Schema.

Wie Sie an diesem Beispiel sehen, lassen sich viele Detailfragen durch solche formalen Regeln klären, zum Beispiel die Frage, ob eine Blockanweisung auch aus einem leeren Klammerpaar ohne Anweisungen bestehen kann. Die Antwort ist offensichtlich: ja.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

if-statement
::= if (<<condition>>) <<statement>>
| if (<<condition>>) <<statement>>
else <<statement>>

switch-statement
::= switch (<<switch-expression>>)
{ <<switch-list>> }

Als nächstes die beiden Fallunterscheidungen, also die if-Anweisung und die switch-Anweisung.

***Erinnerung*: Die switch-Anweisung hatten wir im gleichnamigen Abschnitt in Kapitel 03c gesehen.**

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

if-statement

::= if (<<condition>>) <<statement>>

| if (<<condition>>) <<statement>>
else <<statement>>

switch-statement

::= switch (<<switch-expression>>)
{ <<switch-list>> }

Zur if-Anweisung an sich gibt es nicht viel zu sagen. Erwähnenswert ist vielleicht noch einmal als Wiederholung, dass es die if-Anweisung mit und ohne else-Teil gibt.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if-statement
      ::= if ( <<condition>> ) <<statement>>
        |  if ( <<condition>> ) <<statement>>
           else <<statement>>

switch-statement
      ::= switch ( <<switch-expression>> )
          { <<switch-list>> }
```

Das ist eine zweite Form von rekursiver Definition, ein Beispiel für *indirekte* Rekursion: Der Begriff Anweisung wird indirekt über den Begriff if-Anweisung wieder auf den Begriff Anweisung zurückgeführt. Anders formuliert: Eine Anweisung kann eine if-Anweisung sein, die wiederum eine oder zwei beliebige Anweisungen aus der Aufzählung in der Definition der Anweisung enthält.

Das ist letztendlich der Grund, warum man bei einer einzelnen Anweisung im if-Teil oder im else-Teil die geschweiften Klammern weglassen darf, aber nicht muss: Lässt man sie weg, ist die Anweisung eine einfache Anweisung (simple statement), lässt man sie *nicht* weg, ist die Anweisung ein Anweisungsblock (compound-statement) mit nur einer einzigen Anweisung darin. Der Unterschied ist natürlich nur syntaktischer Natur, in beiden Fällen passiert exakt dasselbe.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if-statement
      ::= if ( <<condition>> ) <<statement>>
        |  if ( <<condition>> ) <<statement>>
           else <<statement>>

switch-statement
      ::= switch ( <<switch-expression>> )
          { <<switch-list>> }
```

Wie gesehen, kommt nach dem Schlüsselwort switch in runden Klammern ein Ausdruck, aber nicht beliebig, sondern nur mit sehr eingeschränkten Möglichkeiten. Zur Abgrenzung verwenden wir hier nicht das allgemeine Wort expression, sondern einen spezifischeren Begriff, switch-expression.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

if-statement
::= if (<<condition>>) <<statement>>
| if (<<condition>>) <<statement>>
else <<statement>>

switch-statement
::= switch (<<switch-expression>>)
{ <<switch-list>> }

Und in geschweiften Klammern kommen dann die einzelnen Fälle.

Die exakte Definition zeigt, dass das geschweifte Klammerpaar an *dieser* Stelle *nicht* weggelassen werden darf.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

switch-list ::= <<case-list>>
| <<case-list>> <<default-case>>

case-list ::= ε | <<case-item>> <<case-list>

case-item
::= case <<case-expr>> : <<statement-sequence>>

default-case ::= default : <<statement-sequence>>

Hier nun die Details der switch-Anweisung von der letzten Folie.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

switch-list ::= <<case-list>>
| <<case-list>> <<default-case>>

case-list ::= ε | <<case-item>> <<case-list>

case-item
::= case <<case-expr>> : <<statement-sequence>>

default-case ::= default : <<statement-sequence>>

Der Rumpf der switch-Anweisung besteht aus einer Liste von Fällen, wobei ein default-Fall dabei sein kann, aber nicht sein muss.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

switch-list ::= <<case-list>>
| <<case-list>> <<default-case>>

case-list ::= ε | <<case-item>> <<case-list>

case-item
::= case <<case-expr>> : <<statement-sequence>>

default-case ::= default : <<statement-sequence>>

Eine Liste oder Sequenz von Fällen wird wieder rekursiv definiert.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

switch-list ::= <<case-list>>
| <<case-list>> <<default-case>>

case-list ::= ε | <<case-item>> <<case-list>

case-item
::= case <<case-expr>> : <<statement-sequence>>

default-case ::= default : <<statement-sequence>>

Wir haben schon gesehen, wie ein einzelner Fall gebildet werden kann: Schlüsselwort case nebst Doppelpunkt, danach die Anweisungen. Der Begriff statement-sequence war so definiert, dass die Sequenz von Anweisungen auch leer sein kann.

***Erinnerung*: Ein Beispiel für eine leere Sequenz von Anweisungen nach dem case haben wir bei der Einführung der switch-Anweisung im Beispiel mit dem Enum-Typ Month gesehen (Kapitel 03c). Das ist nützlich, wenn man zwei oder mehr Fälle völlig identisch behandeln will.**

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

switch-list ::= <<case-list>>
| <<case-list>> <<default-case>>

case-list ::= ε | <<case-item>> <<case-list>

case-item
::= case <<case-expr>> : <<statement-sequence>>

default-case ::= default : <<statement-sequence>>

Analog ist der default-Fall gebildet.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

while-loop
::= while (<<condition>>) <<statement>>

do-while-loop
::= do <<statement>> while (<<condition>>) ;

for-loop ::= <<long-for-loop>> | <<short-for-loop>>

Als nächstes die Schleifen.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

while-loop
::= while (<<condition>>) <<statement>>

do-while-loop
::= do <<statement>> while (<<condition>>) ;

for-loop ::= <<long-for-loop>> | <<short-for-loop>>

Die while-Schleife lässt sich sehr einfach definieren. Eine Bedingung ist einfach ein Ausdruck von einem booleschen Typ. Auf Ausdrücke kommen wir gleich noch ganz allgemein zu sprechen, bei einfachen Anweisungen.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

while-loop
::= while (<<condition>>) <<statement>>

do-while-loop
::= do <<statement>> while (<<condition>>) ;

for-loop ::= <<long-for-loop>> | <<short-for-loop>>

Analog die do-while-Schleife.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

while-loop
::= while (<<condition>>) <<statement>>

do-while-loop
::= do <<statement>> while (<<condition>>) ;

for-loop ::= <<long-for-loop>> | <<short-for-loop>>

Wir hatten schon festgestellt, dass die do-while-Schleife durch ein Semikolon abgeschlossen wird. Wie schon gesagt, solche Detailfragen, etwa wo ein Semikolon hingehört, lassen sich gut durch die formalen Regeln klären.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

while-loop
::= while (<<condition>>) <<statement>>

do-while-loop
::= do <<statement>> while (<<condition>>) ;

for-loop ::= <<long-for-loop>> | <<short-for-loop>>

Von der for-Schleife haben wir zwei Varianten kennen gelernt, die Normalform und eine verkürzte Form für bestimmte Fälle. Die Details der beiden Fälle sehen wir uns auf der nächsten Folie an.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

long-for-loop
::= for (<<simple-statement>> ;
<<condition>> ; <<expr-statement>>)
<<statement>>

short-for-loop
::= for (<<type-name>> <<identifier>>
: <<lvalue>>) <<statement>>

Nun also die beiden Versionen der for-Schleife.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

long-for-loop
::= for (<<simple-statement>> ;
<<condition>> ; <<expr-statement>>)
<<statement>>

short-for-loop
::= for (<<type-name>> <<identifier>>
: <<lvalue>>) <<statement>>

Diesen Begriff hatten wir schon eben auf der lexikalischen Ebene mit dem hier verwendeten Formalismus beschrieben. Wir übernehmen einfach die dortige Definition.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

long-for-loop
::= for (<<simple-statement>> ;
<<condition>> ; <<expr-statement>>)
<<statement>>

short-for-loop
::= for (<<type-name>> <<identifier>>
: <<lvalue>>) <<statement>>

DIese vier Begriffe sind noch undefiniert. Der Begriff simple-statement war auch in der Aufzählung, die den Begriff statement definiert hat, den definieren wir gleich. Da werden wir sehen, dass die Begriffe expr-statement und lvalue Spezialfälle von simple-statement sind, und type-name wird in diesem Zusammenhang ebenfalls definiert.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

long-for-loop
::= for (<<simple-statement>> ;
<<condition>> ; <<expr-statement>>)
<<statement>>

short-for-loop
::= for (<<type-name>> <<identifier>>
: <<lvalue>>) <<statement>>

***Erinnerung*: Linksausdrücke wurden im Kapitel 03c betrachtet. Dort wurde auch gesagt, dass Linksausdrücke in der englischen Fachsprache etwas ungenau lvalues heißen.**

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

simple-statement
::= <<variable-declation>> | <<const-declaration>>
| <<expr-statement>>

variable-declaration ::= <<type-name>> <<var-list>>

var-list ::= <<var-decl>> | <<var-decl>> , <<var-list>>

<<var-decl>> ::= <<identifier>>
| <<identifier>> = <<expr-statement>>

Die einfachen Ausdrücke haben wir uns bis zu Schluss aufgehoben, weil sie recht vielfältig sind.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

simple-statement
::= <<variable-declation>> | <<const-declaration>>
| <<expr-statement>>

variable-declaration ::= <<type-name>> <<var-list>>

var-list ::= <<var-decl>> | <<var-decl>> , <<var-list>>

<<var-decl>> ::= <<identifier>>
| <<identifier>> = <<expr-statement>>

Wir unterscheiden mehrere Arten von einfachen Anweisungen. Hier sehen wir auch noch einmal den Begriff expr-statement von der letzten Folie (bei der Langform der for-Schleife).

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

simple-statement
::= <<variable-declation>> | <<const-declaration>>
| <<expr-statement>>

variable-declaration ::= <<type-name>> <<var-list>>

var-list ::= <<var-decl>> | <<var-decl>> , <<var-list>>

<<var-decl>> ::= <<identifier>>
| <<identifier>> = <<expr-statement>>

Die Deklaration einer Konstante ist bekanntlich dasselbe wir die Deklaration einer Variable, nur mit Schlüsselwort final.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

simple-statement
::= <<variable-declation>> | <<const-declaration>>
| <<expr-statement>>

variable-declaration ::= <<type-name>> <<var-list>>

var-list ::= <<var-decl>> | <<var-decl>> , <<var-list>>

<<var-decl>> ::= <<identifier>>
| <<identifier>> = <<expr-statement>>

Wir haben gesehen, dass in einer einzigen Deklaration gleich mehrere Variable eingeführt werden können, separiert durch Kommas.

Syntaktische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

simple-statement
::= <<variable-declation>> | <<const-declaration>>
| <<expr-statement>>

variable-declaration ::= <<type-name>> <<var-list>>

var-list ::= <<var-decl>> | <<var-decl>> , <<var-list>>

<<var-decl>> ::= <<identifier>>
| <<identifier>> = <<expr-statement>>

Wie wir immer wieder gesehen haben, kann eine Variable entweder gleich initialisiert werden oder auch nicht, daher diese Fallunterscheidung.

Damit beenden wir unseren kleinen Einblick in die formalen Syntaxregeln.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Semantikfehler in Java:

RuntimeException wird geworfen

Syntaxfehler findet und moniert der Compiler. Semantikfehler findet der Compiler in der Regel nicht, sondern diese wirken sich erst zur Laufzeit aus. Konkret in Java heißt das, dass eine RuntimeException geworfen wird. Wird diese nicht gefangen, bricht das Programm ab.

***Erinnerung*: Im Kapitel 05, Abschnitt zu Exceptions, wurden auch RuntimeExceptions behandelt.**

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: durch 0 teilen

```
int x = 0;

...

int y = 1 / x;
```

Dieser beliebte Fehler ist also auf der semantischen Ebene: Division durch 0.

In sehr einfachen Situationen wie dieser finden heutige Compiler den Fehler schon beim Übersetzen. Aber in komplexeren Situationen, zum Beispiel wenn der Wert für x erst zur Laufzeit durch den Nutzer eingegeben wird, hat der Compiler keine Chance, den Fehler zur Übersetzungszeit zu finden.

Daher rechnet man diesen Fehler weiterhin zu den semantischen.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: falscher Arrayindex

```
int[] a = new int [10];
a[0]  = 2;
a[9]  = 3;
a[-3] = 5;
a[10] = 7;
a[1984] = 11;
```

Nun ein anderer beliebter Semantikfehler.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: falscher Arrayindex

```
int[] a = new int [10];
a[0]  = 2;
a[9]  = 3;
a[-3] = 5;
a[10] = 7;
a[1984] = 11;
```

Hier wird ein Array mit zehn Komponenten vom Typ int angelegt. Das heißt, dieses Array hat die Indizes 0 bis 9.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: falscher Arrayindex

```
int[] a = new int [10];
a[0]  = 2;
a[9]  = 3;
a[-3] = 5;
a[10] = 7;
a[1984] = 11;
```

Diese beiden Zugriffe mit Indizes 0 und 9 sind denn auch ok.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: falscher Arrayindex

```
int[] a = new int [10];
a[0]  = 2;
a[9]  = 3;
a[-3] = 5;
a[10] = 7;
a[1984] = 11;
```

Aber in diesen drei Zugriffen ist der Index außerhalb des Bereichs 0 bis 9. Jede einzelne dieser drei Zeilen führt zum Wurf einer RuntimeException, genauer: einer ArrayIndexOutOfBoundsException.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: auf null zugreifen

```
String str = null;
...

int len = str.length;
```

Ein drittes Beispiel für Semantikfehler: Zugriff auf ein nichtexistierendes Objekt.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: auf null zugreifen

```
String str = null;
...

int len = str.length;
```

Eine Variable vom Typ String wird zwar eingerichtet, aber nicht mit einem String-Objekt verbunden. Statt dessen wird der symbolische Wert null zugewiesen.

Semantische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: auf null zugreifen

```
String str = null;
...

int len = str.length;
```

Da kein String-Objekt existiert, ist der Zugriff auf die String-Länge ein semantischer Fehler, also wieder RuntimeException, genauer: NullPointerException.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;
n = true;

class X { ... }    //  hat keine Methode m

X a = new X();
a.m();
```

➢Syntaktische oder semantische Fehler?

Was ist aber beispielsweise mit Typfehlern wie diesen beiden? Solche Fehler passen nicht so recht zu den bisher betrachteten syntaktischen Fehlern. Aber semantische Fehler sind das auch nicht, da der Compiler sie garantiert immer entdeckt – zumindest in einer statisch typisierten Sprache wie Java.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition für Programmiersprachen:

- ***Syntax*: die Regeln, nach denen zu entscheiden ist, ob ein gegebener Quelltext ein korrektes Programm in der Sprache ist.**
  - **Vorausgesetzt, die lexikalische Ebene ist korrekt.**
- ***Semantik*: was ein gegebenes, sprachlich *korrektes* Programm tatsächlich macht.**

Um hier weiterzukommen, schauen wir uns jetzt doch einmal kurz an, wie die Begriffe Syntax und Semantik eigentlich definiert sind.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition für Programmiersprachen:

- ***Syntax***: **die Regeln, nach denen zu entscheiden ist, ob ein gegebener Quelltext ein korrektes Programm in der Sprache ist.**
  - **Vorausgesetzt, die lexikalische Ebene ist korrekt.**
- ***Semantik***: **was ein gegebenes, sprachlich *korrektes* Programm tatsächlich macht.**

In erster Näherung kann man also sagen: Die Syntax determiniert, ob ein Quelltext korrekt ist, also ob er vom Compiler übersetzt werden kann.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition für Programmiersprachen:

- ***Syntax*: die Regeln, nach denen zu entscheiden ist, ob ein gegebener Quelltext ein korrektes Programm in der Sprache ist.**
  - **Vorausgesetzt, die lexikalische Ebene ist korrekt.**
- ***Semantik*: was ein gegebenes, sprachlich *korrektes* Programm tatsächlich macht.**

Die Semantik sagt im Prinzip, was wir an Effekten und Ergebnissen beobachten werden, wenn wir das übersetzte Programm mit irgendwelchen Eingabedaten aufrufen.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;
n = true;

class X { ... }    //  hat keine Methode m

X a = new X();
a.m();
```

➢Also zwei syntaktische Fehler??

Nach dieser Definition von Syntax und Semantik ergibt sich *zunächst* einmal, dass ein solcher Typfehler als syntaktischer Fehler einzustufen wäre.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

Kleine Bedeutungsverschiebung in der Informatik:

- **Ein *Teil* der Syntax wird in formalen Syntaxregeln ausgedrückt.**
- **Häufig werden Begriffe wie „Syntax“ und „syntaktischer Fehler“ auf das eingeschränkt, was in den formalen Regeln festgelegt ist.**

Allerdings weicht man in der Informatik subtil, aber entscheidend ab von der Definition des Begriffs Syntax.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

Kleine Bedeutungsverschiebung in der Informatik:

- **Ein *Teil* der Syntax wird in formalen Syntaxregeln ausgedrückt.**
- **Häufig werden Begriffe wie „Syntax“ und „syntaktischer Fehler“ auf das eingeschränkt, was in den formalen Regeln festgelegt ist.**

Nicht alle Aspekte des Begriffs Syntax werden in der Informatik gleich behandelt. Manche werden in formale Regelwerke gegossen, andere bleiben dabei außen vor. Der Grund ist einfach der, dass sonst die Syntaxregeln zu komplex würden oder sogar eine komplexere Art von Regelwerk benötigt würde, um alles auszudrücken, was man *eigentlich* zur Syntax rechnet.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

Kleine Bedeutungsverschiebung in der Informatik:

- **Ein *Teil* der Syntax wird in formalen Syntaxregeln ausgedrückt.**
- **Häufig werden Begriffe wie „Syntax“ und „syntaktischer Fehler“ auf das eingeschränkt, was in den formalen Regeln festgelegt ist.**

Und nur das, was nach diesem Regelwerk inkorrekt ist, das ja nur einen Teil der Syntax abbildet, nur das wird in der Informatik typischerweise ein syntaktischer Fehler genannt.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Frage*: welcher Teil der Syntax ist das?
- *Antwort*: der *kontextfreie* Teil
- *Heißt*: für die Korrektheit eines syntaktischen Konstrukts wird nicht abgeprüft, ob es zu seinem Kontext passt
- *Beispiel*: ob eine Variable typgerecht gemäß ihrer Deklaration verwendet wird

Geklärt werden muss noch, welche Aspekte der Syntax nun typischerweise in Regelwerke gegossen werden.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Frage*: welcher Teil der Syntax ist das?
- *Antwort*: der *kontextfreie* Teil
- *Heißt*: für die Korrektheit eines syntaktischen Konstrukts wird nicht abgeprüft, ob es zu seinem Kontext passt
- *Beispiel*: ob eine Variable typgerecht gemäß ihrer Deklaration verwendet wird

Die Antwort ist: die Aspekte, die sich leicht in formale Regelwerke gießen lassen. In der Informatik lautet der Fachbegriff *Kontextfreiheit*.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Frage*: welcher Teil der Syntax ist das?
- *Antwort*: der *kontextfreie* Teil
- *Heißt*: für die Korrektheit eines syntaktischen Konstrukts wird nicht abgeprüft, ob es zu seinem Kontext passt
- *Beispiel*: ob eine Variable typgerecht gemäß ihrer Deklaration verwendet wird

Das sind einfach die Regeln, die rein lokal sind, das heißt, Zusammenhänge zwischen verschiedenen Teilen des Quelltextes – die ja vielleicht viele, viele Zeilen auseinanderliegen könnten, mit vielen damit nicht in Zusammenhang stehenden anderen syntaktischen Konstrukten dazwischen – diese Zusammenhänge werden ignoriert.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Frage*: welcher Teil der Syntax ist das?
- *Antwort*: der *kontextfreie* Teil
- *Heißt*: für die Korrektheit eines syntaktischen Konstrukts wird nicht abgeprüft, ob es zu seinem Kontext passt
- *Beispiel*: ob eine Variable typgerecht gemäß ihrer Deklaration verwendet wird

Und genau da finden wir Typfehler: Fehler, die dadurch zustande kommen, dass eine Variablendeklaration irgendwo im Qulltext und eine Verwendung dieser Variablen irgendwo ganz woanders nicht zusammenpassen.

Syntaktisch? Semantisch?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;          X a = new X();
n = true;       a.m();
```

Syntaktische Fehler im normalen Sinne

- **Zumindest in statisch typisierten Sprachen**
- **Aber nicht im engeren Sinne**

Also, sammeln wir jetzt: Typfehler sind zwar im weiteren, gewöhnlichen Sinne, aber nicht in diesem engeren Sinne syntaktische Fehler, denn nur Verstöße gegen kontextfreie Regeln werden syntaktische Fehler genannt. In einer stark typisierten Sprache wie Java sind es auch keine semantischen Fehler, denn sie werden grundsätzlich vom Compiler entdeckt, niemals zur Laufzeit. Wird der engere Begriff von Syntax zugrunde gelegt, dann passen Typfehler einfach nicht in die fünf Abstraktionsebenen, um die es in diesem Abschnitt geht.

Logische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

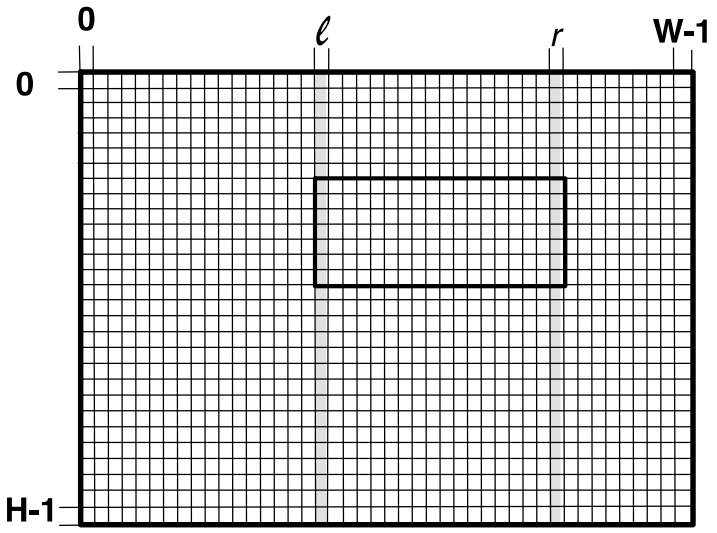
Beispielfehler: off-by-one error

Jetzt die dritte, die logische Ebene. Logische Fehler sind Umsetzungsfehler. Man weiß, was das Programm eigentlich tun soll, aber durch einen Denkfehler beim Programmieren macht das Programm etwas anderes.

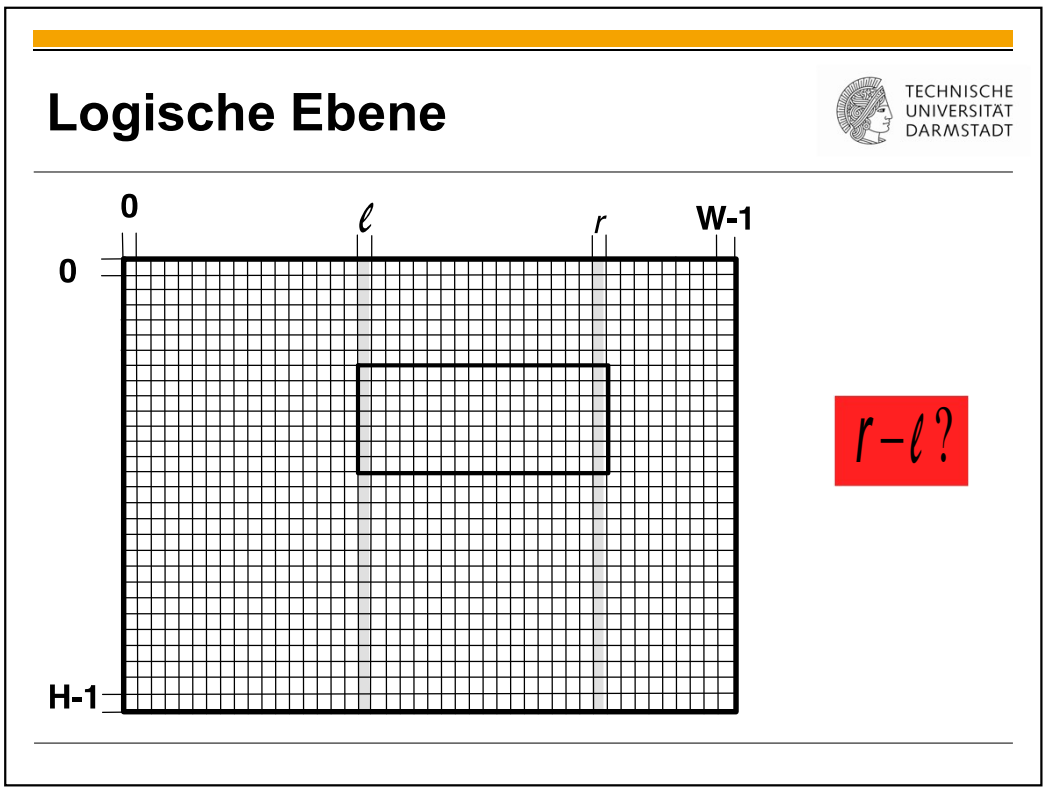
Auch auf der logischen Ebene gibt es sehr beliebte Fehler, zum Beispiel der sogenannte off-by-one error. Das heißt, man berechnet einen mathematischen Ausdruck im Prinzip richtig, vertut sich aber um eins.

Logische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT



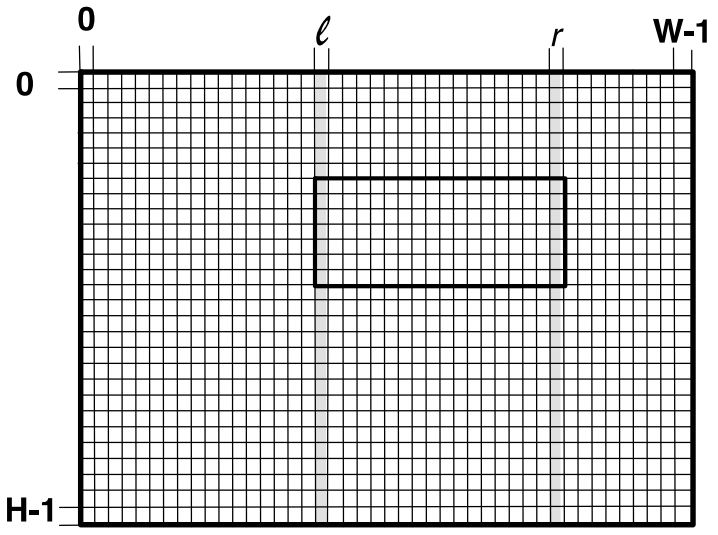
Ein anschauliches Beispiel für den off-by-one error: Dies ist schematisch das Pixelraster eines Bildschirms. Die Weite W ist die Anzahl der Pixel horizontal, die Höhe H ist die Anzahl der Pixel vertikal. Schwarz umrandet im Innern ist die Pixelmenge eines einzelnen Fensters. Das linkeste Pixel des Fensters hat Koordinate *l*, das rechteste hat Koordinate *r*.



Wie breit ist dieses Fenster nun in Anzahl Pixeln? Wer den off-by-one error noch nicht gesehen hat, wird vielleicht spontan antworten: *r* minus *l* Pixel.

Logische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT



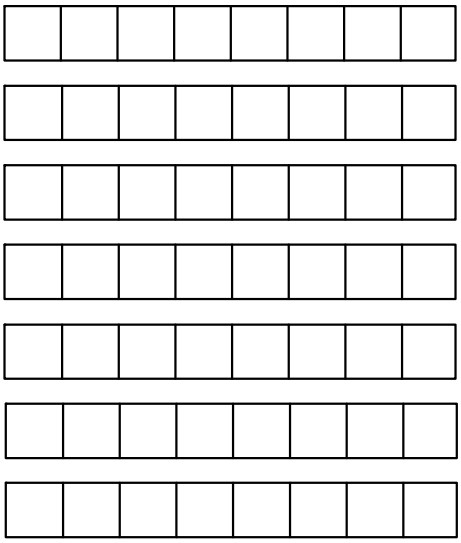
$r - \ell + 1$!

Wenn Sie nachzählen, werden Sie aber feststellen, dass man sich mit dieser Antwort um genau eins vertut. Das Fenster ist tatsächlich *r* minus *l* plus 1 Pixel breit.

Logische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler:
Wochentag
Zu lang

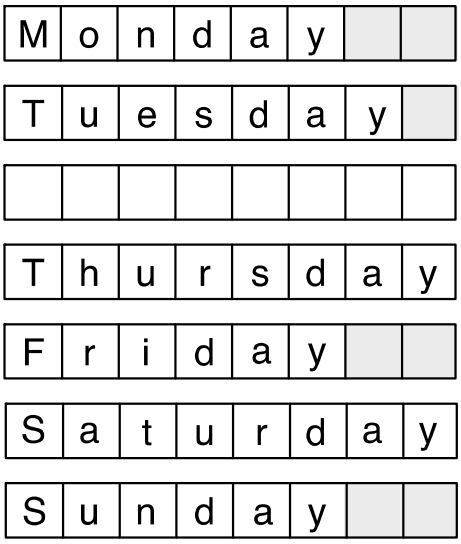


Ein weiterer logischer Fehler aus der Praxis, und zwar aus dem englischsprachigen Raum. Für den Namen des heutigen Wochentages waren acht Zeichen reserviert.

Logische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler:
Wochentag
Zu lang

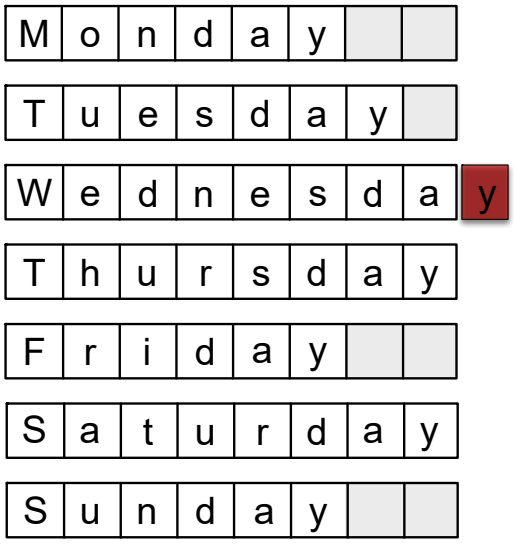


An sechs der sieben Wochentage klappt das auch wunderbar.

Logische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler:
Wochentag
Zu lang



Nur mittwochs stürzt das Programm aus unerklärlichen Gründen ab. Dieser logische Fehler äußert sich konkret in einem semantischen Fehler, nämlich Zugriff auf unzulässigen Arrayindex.

Typisch für Logikfehler ist: Selbst wenn sie durch Programmabsturz zeigen, dass sie da sind, sind sie dennoch sehr schwer zu finden. In diesem Beispiel musste man erst einmal darauf kommen, dass das Programm immer nur mittwochs abstürzt. An den anderen Wochentagen kann der Fehler nicht reproduziert und damit auch nicht gefunden werden. Wenn die interne Fehlermeldung dem Endanwender nicht vom Programm angezeigt wird und der Fehlerfall immer erst am nächsten Tag weiterverfolgt wird, findet man den Fehler nie – und weiß noch nicht einmal warum.

Spezifikatorische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler: Jahr-2000-Problem

- Jahreszahlen bis weit in die 80er mit zwei Ziffern kodiert
- Bis 2000 ist das Programm eh nicht mehr in Betrieb ...

Logikfehler waren Fehler bei der Übertragung von eigentlich richtigen Gedanken in die Programmiersprache. Spezifikatorischer Fehler heißt, dass schon der umzusetzende Gedanke falsch war.

Das wohl bekannteste Beispiel für spezifikatorische Fehler ist das Jahr-2000-Problem.

In vielen Programmen wurden Jahreszahlen mit zwei statt vier Ziffern kodiert, um Speicherplatz zu sparen. Im Jahr 2000 war das dann natürlich ein Problem.

Der spezifikatorische Fehler war, dass man glaubte, bis zum Jahr 2000 wären alle diese Programme längst außer Dienst. Waren sie aber nicht. Die zeitlichen Rahmenbedingungen wurden völlig falsch eingeschätzt.

Spezifikatorische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispielfehler:

- **Landeanflug Lufthansa-Airbus in Warschau 14.9.1993**
- **Aquaplaning und starke Seitenwinde**
- **Schubumkehr zu lange blockiert**
- ***Ursache*: Algorithmus für Entscheidung Bodenkontakt ja/nein**

Noch ein häufig zitiertes Beispiel eines Spezifikationsfehlers. Standardsituation Landeanflug eines Flugzeugs, aber unter sehr ungünstigen Rahmenbedingungen.

Nach dem Aufsetzen soll die Schubumkehr eigentlich das Flugzeug abbremsen. Die Schubumkehr war aber ein paar wertvolle Sekunden lang blockiert, so dass das Flugzeug über die Landebahn hinaus in einen Erdwall gerollt ist.

Was war passiert: Während des Fluges ist die Schubumkehr aus guten Gründen blockiert. Bei ausgefahrenen Radwerken wird permanent kontrolliert, ob die Räder Bodenkontakt haben, und dann wird die Blockierung der Schubumkehr aufgehoben. Genauer gesagt werden der Druck auf die Räder und die Drehgeschwindigkeit der Räder kontrolliert. Unter den ungünstigen Witterungsbedingungen an diesem Tag hat der Algorithmus aber nicht erkannt, dass die Räder schon auf dem Boden waren.

Spezifikatorische Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Gewünscht*: ein Tool zur Kommunikation (=Werbung) nach außen
- *Angegangen*: ein Tool zur abteilungsinternen Kommunikation

Zum Abschluss das denkwürdigste Beispiel, von dem ich je gehört habe, so geschehen in einem interdisziplinären Forschungsprojekt.

Aus Sicht der Anwender sollte das Programm die Kommunikation nach außen unterstützen, also die Öffentlichkeitsarbeit.

Gebastelt wurde aber an einem Programm zur Unterstützung der internen Kommunikation. Es hat etliche Monate gedauert, bis dieser Spezifikationsfehler endlich aufgefallen ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Was heißt eigentlich

Korrektheit von Software?

Lassen Sie uns als nächstes kurz klären, was wir eigentlich meinen, wenn wir sagen, Software ist korrekt beziehungsweise *nicht* korrekt.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kein Programmabbruch durch Fehler
- Termination
  - wenn Aufgabe erledigt oder
  - wenn Befehl zur Termination von außen
- Korrekte Ausgaben und Effekte

Die Antwort besteht im Prinzip aus diesen drei Kriterien.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kein Programmabbruch durch Fehler
- Termination
  - wenn Aufgabe erledigt oder
  - wenn Befehl zur Termination von außen
- Korrekte Ausgaben und Effekte

Bei Racket heißt das, dass DrRacket den Programmlauf nicht mit einer Fehlermeldung abbricht. So etwas passiert, wenn der Typ eines Operanden nicht zur vordefinierten Operation passt.

Bei Java hingegen wird der Programmlauf dadurch abgebrochen, dass eine Exception geworfen, aber nicht gefangen wurde. Wie wir aus Kapitel 05 wissen, müssen alle Exception-Klassen außer RuntimeException beziehungsweise die von Klasse RuntimeException direkt oder indirekt abgeleiteten Klassen gefangen oder weitergereicht werden und können auch nicht über die vom Laufzeitsystem aufgerufene Methode hinaus weitergereicht werden. Daher bedeutet Programmabbruch in Java ganz konkret: Eine Exception von Klasse RuntimeException oder einer davon abgeleiteten Klasse ist von keiner Methode, die momentan auf dem Call-Stack ist, gefangen worden.

***Nebenbemerkung*: Es ist natürlich nicht völlig ausgeschlossen, dass die Laufzeitumgebung selbst fehlerhaft ist und allein deshalb Ihr Programmlauf abgebrochen wird. Aber das ist natürlich nicht mehr Thema der FOP.**

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kein Programmabbruch durch Fehler
- Termination
  - wenn Aufgabe erledigt oder
  - wenn Befehl zur Termination von außen
- Korrekte Ausgaben und Effekte

Beim Thema Termination müssen wir zwei Fälle unterscheiden: ob eine Softwarekomponente einfach nur eine bestimmte Aktion durchführen und dann zu Ende sein soll oder ob sie potentiell unendlich lange laufen soll, weil sie einen permanenten Prozess steuert, zum Beispiele eine Ampel. Im ersten Fall heißt korrekte Termination einfach nur, dass sie nach endlich vielen Schritten tatsächlich beendet ist, also beispielsweise nicht in eine Endlosschleife läuft. Im zweiten Fall heißt korrekte Termination, dass die Komponente genau dann terminiert, wenn sie von autorisierter Seite den Befehl dazu bekommt, zum Beispiel über ein Event wie bei GUIs.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kein Programmabbruch durch Fehler
- Termination
  - wenn Aufgabe erledigt oder
  - wenn Befehl zur Termination von außen
- Korrekte Ausgaben und Effekte

Das, was ein Stück Software an Daten ausgibt, muss natürlich korrekt sein. Eine Softwarekomponente kann auch Effekte auf die Außenwelt haben, beispielsweise Steuerungssoftware für Geräte. Diese Effekte müssen selbstverständlich ebenfalls korrekt sein.

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von Klassen

Wir unterscheiden zwischen verschiedenen Ebenen und beginnen mit der Ebene von Klassen. Danach schauen wir uns Subroutinen *als Ganzes* an, das sind Methoden in Java und Funktionen in Racket, und schließlich schauen wir in Subroutinen hinein.

***Nebenbemerkung*: In Racket gibt es Klassen analog zu Java, aber da wir diese nicht behandelt haben, schauen wir uns in diesem Abschnitt nur Klassen in Java an. Alles hier Gesagte lässt sich aber mehr oder weniger eins-zu-eins auf Klassen in Racket und anderen Sprache übertragen.**

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Darstellungsinvariante (representation invariant) von Klassen und Interfaces
- Implementationsinvariante (implementation invariant) von Klassen

Wie geht man überhaupt an das Thema Korrektheit von Klassen heran? Der Einstiegspunkt ist die Formulierung zweier Sammlungen von Aussagen über die Klasse. Die beiden Sammlungen nennt man *Darstellungsinvariante* beziehungsweise *Implementationsinvariante*.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Darstellungsinvariante (representation invariant) von Klassen und Interfaces
- Implementationsinvariante (implementation invariant) von Klassen

Die Darstellungsinvariante beschreibt, wie Objekte der Klasse sich dem Nutzer der Klasse darstellen sollen. Konkreter formuliert heißt das: die Sicht, die die Attribute und Methoden vermitteln, die public definiert sind. Selbstverständlich sehen wir gleich Beispiele dazu.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Darstellungsinvariante (representation invariant) von Klassen und Interfaces
- Implementationsinvariante (implementation invariant) von Klassen

Die Implementationsinvariante ist analog zur Darstellungsinvariante, aber behandelt eben den Teil der Klassendefinition, der *nicht* public ist. Die Implementationsinvariante sollte nicht unbedingt für Nutzer der Klasse zugänglich sein. Sie kann zum Beispiel als ganz normaler Java-Kommentar in der Quelldatei der Klasse stehen.

***Nebenbemerkung*: Seit Java 9 sind private-Methoden auch in Interfaces möglich. Da wir solche Möglichkeiten bisher nicht behandelt haben, lassen wir sie auch hier aus und konzentrieren uns bei der Implementationsinvariante auf Klassen. Der Transfer zu Interfaces a la Java 9 sollte trivial sein.**

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public class DMatrix {
   private double[ ][ ] matrix;
   .........
}
```

Als erstes Beispiel für Darstellungs- und Implementationsinvariante definieren wir eine selbstgebaute Matrixklasse, deren Einträge vom primitiven Datentyp double sind.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public class DMatrix {
  private double[ ][ ] matrix;
  ..........
}
```

Die Matrix wird naheliegenderweise in einem private-definierten Array von Arrays von double gehalten.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public class DMatrix {
   private double[ ][ ] matrix;
   ..........
}
```

Die Methoden der Klasse schauen wir uns auf den nächsten Folien an, ...

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public DMatrix ( int numberOfRows, int numberOfColumns )
                                throws IndexOutOfBoundsException {
   if ( numberOfRows <= 0 || numberOfColumns <= 0 )
      throw new IndexOutOfBoundsException();
   matrix = new double [ numberOfRows ] [ ];
   for ( int i = 0; i < numberOfRows; i++ )
      matrix[i] = new double [ numberOfColumns ];
}
```

... als erstes den Konstruktor.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public DMatrix ( int numberOfRows, int numberOfColumns )
                                throws IndexOutOfBoundsException {
   if ( numberOfRows <= 0 || numberOfColumns <= 0 )
      throw new IndexOutOfBoundsException();
   matrix = new double [ numberOfRows ] [ ];
   for ( int i = 0; i < numberOfRows; i++ )
      matrix[i] = new double [ numberOfColumns ];
}
```

Die Anzahl der Zeilen und die Anzahl der Spalten werden zweckmäßigerweise dem Konstruktor als Parameter übergeben.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public DMatrix ( int numberOfRows, int numberOfColumns )
                                    throws IndexOutOfBoundsException {
  if ( numberOfRows <= 0 || numberOfColumns <= 0 )
     throw new IndexOutOfBoundsException();
  matrix = new double [ numberOfRows ] [ ];
  for ( int i = 0; i < numberOfRows; i++ )
     matrix[i] = new double [ numberOfColumns ];
}
```

Natürlich sichern wir die Klasse gegen eine nichtpositive Anzahl der Zeilen oder Spalten ab.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public DMatrix ( int numberOfRows, int numberOfColumns )
                                throws IndexOutOfBoundsException {
   if ( numberOfRows <= 0 || numberOfColumns <= 0 )
      throw new IndexOutOfBoundsException();
   matrix = new double [ numberOfRows ] [ ];
   for ( int i = 0; i < numberOfRows; i++ )
      matrix[i] = new double [ numberOfColumns ];
}
```

Wir stellen hier schon einmal fest, dass offenkundig der Index im Hauptarray die Zeilenzahl angeben soll, so dass die Länge des Hauptarrays also gleich der Zeilenzahl ist. Dann wird also der Index in den Komponenten des Hauptarrays die Spaltenzahl angeben. Natürlich hätten wir das auch umdrehen können, aber für eine der beiden Möglichkeiten müssen wir uns halt entscheiden – und diese dann auch in allen Methoden der Klasse konsequent beibehalten.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public DMatrix ( int numberOfRows, int numberOfColumns )
                                throws IndexOutOfBoundsException {
   if ( numberOfRows <= 0 || numberOfColumns <= 0 )
      throw new IndexOutOfBoundsException();
   matrix = new double [ numberOfRows ] [ ];
   for ( int i = 0; i < numberOfRows; i++ )
      matrix[i] = new double [ numberOfColumns ];
}
```

Die Komponenten des Hauptarrays werden dann also allesamt so dimensioniert, dass für jede Spalte ein Index vorhanden ist.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public int getNumberOfRows () {
   return matrix.length;
}

public int getNumberOfColumns () {
   return matrix[0].length;     //  matrix != null &&
                                //  matrix.length > 0  !!!
}
```

Natürlich soll es jederzeit nach Konstruktion des Matrix-Objektes möglich sein, die Zeilenzahl beziehungsweise die Spaltenzahl abzufragen. Dies ist ein gutes Beispiel dafür, dass man solche Funktionalität immer in Methoden packen sollte, denn dahinter können sich schon kitzlige Implementationsdetails verbergen, auf die man besser keinen direkten public-Zugriff erlauben sollte. Zumindest bei Methode getNumberOfColumns kann man das sicherlich so sagen, wie wir gleich besprechen werden.

***Erinnerung*: Im Fallbeispiel GeomShape2D in Kapitel 02 hatten wir den Sinn von get- und set-Methoden schon einmal anhand von konkreten Beispielen besprochen.**

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public int getNumberOfRows () {
   return matrix.length;
}

public int getNumberOfColumns () {
   return matrix[0].length;     //  matrix != null &&
                                //  matrix.length > 0  !!!
}
```

Die Zeilenzahl zurückzuliefern, geht ganz einfach: Wir haben ja schon festgelegt, dass das Hauptarray die Zeilen repräsentiert.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public int getNumberOfRows () {
   return matrix.length;
}

public int getNumberOfColumns () {
   return matrix[0].length;    //  matrix != null &&
                               //  matrix.length > 0  !!!
}
```

Folgender Punkt ist jetzt wichtig: An dieser Stelle setzen wir voraus, dass die Zeilenzahl niemals 0 sein kann, denn sonst gäbe es keinen Index 0 im Array namens matrix. Solche Feinheiten, die man leicht übersieht, müssen unbedingt dokumentiert werden, denn wenn man später die Matrix-Klasse so abändert, dass Zeilenzahl 0 *doch* möglich wird, weist uns die Dokumentation darauf hin, dass es vorher anders war und dass möglicherweise – so wie hier – irgendwo in der Matrix-Klasse darauf vertraut wird, dass die Zeilenzahl unmöglich 0 sein kann. Wir kommen darauf gleich zurück, bei der Formulierung der Darstellungs- und Implementationsinvariante von Klasse DMatrix.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public double getEntry ( int row, int column )
                              throws IndexOutOfBoundsException {
   if ( row < 0 || row >= matrix.length ||
        column < 0 || column >= matrix[0].length )
      throw new IndexOutOfBoundsException();
   return matrix [ row ] [ column ];
}
```

Jetzt aber erst einmal weiter mit den Methoden der Klasse DMatrix, als nächstes die Methode zum lesenden Zugriff auf einen Eintrag.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public double getEntry ( int row, int column )
                          throws IndexOutOfBoundsException {
  if ( row < 0 || row >= matrix.length ||
       column < 0 || column >= matrix[0].length )
     throw new IndexOutOfBoundsException();
  return matrix [ row ] [ column ];
}
```

Auch hier wird als erstes wieder geprüft, ob die beiden Parameter passen, und andernfalls eine Exception geworfen.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public double getEntry ( int row, int column )
                              throws IndexOutOfBoundsException {
   if ( row < 0 || row >= matrix.length ||
        column < 0 || column >= matrix[0].length )
      throw new IndexOutOfBoundsException();
   return matrix [ row ] [ column ];
}
```

Beim Rückgabewert verlassen wir uns darauf, dass der erste Index tatsächlich die Zeile und der zweite die Spalte darstellt und nicht etwa umgekehrt.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public void setEntry ( int row, int column, double value )
                              throws IndexOutOfBoundsException {
   if ( row < 0 || row >= matrix.length ||
        column < 0 || column >= matrix[0].length )
      throw new IndexOutOfBoundsException();
   matrix [ row ] [ column ] = value;
}
```

Die Methode zum Überschreiben eines Eintrags ist völlig analog dazu, so dass hier nichts zu erläutern ist.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public Object clone () {
   DMatrix result
         = new DMatrix ( matrix.length, matrix[0].length );
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         result.matrix[i][j] = matrix[i][j];
   return result;
}
```

Zur Korrektheit einer Klasse gehört auch, dass eine ererbte Methode überschrieben wird, wenn die Implementation in der Basisklasse nicht passt. Das ist bei den Methoden equals und clone von Klasse Object definitiv der Fall. Als erstes betrachten wir die Methode clone.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public Object clone () {
   DMatrix result
         = new DMatrix ( matrix.length, matrix[0].length );
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         result.matrix[i][j] = matrix[i][j];
   return result;
}
```

Der Kopievorgang besteht aus zwei Schritten, die typisch für Kopiermethoden sind. Der erste Schritt richtet das neue Objekt ein, in diesem Fall also das Hauptarray namens matrix und die Komponenten des Array matrix. Das geht zweckmäßigerweise mit dem Konstruktor.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public Object clone () {
   DMatrix result
         = new DMatrix ( matrix.length, matrix[0].length );
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         result.matrix[i][j] = matrix[i][j];
   return result;
}
```

Im zweiten Schritt werden dann alle Werte im Detail kopiert, hier also die einzelnen Matrixeinträge.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public boolean equals ( Object otherMatrix ) {
   if ( ! otherMatrix instanceof DMatrix )
      return false;
   DMatrix m = (DMatrix) otherMatrix;
   if ( matrix.length != m.matrix.length || matrix[0].length != m.matrix[0].length )
      return false;
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         if ( matrix[i][j] != m.matrix[i][j] )
            return false;
   return true;
}
```

Als zweites und letztes nun Methode equals.

***Nebenbemerkung*: Wenn man die von Object ererbte Methode equals überschreibt, ist es dringend zu empfehlen, dass eine weitere von Object ererbte Methode namens hashCode dann ebenfalls überschrieben wird. Diese Methode hatten wir bisher nicht besprochen, lassen wir auch hier aus.**

***Vorgriff*: Die Hintergründe zu hashCode werden im zweiten Fachsemester in der Lehrveranstaltung *Algorithmen und Datenstrukturen* im Zusammenhang mit Hashtabellen betrachtet.**

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public boolean equals ( Object otherMatrix ) {
   if ( ! otherMatrix instanceof DMatrix )
      return false;
   DMatrix m = (DMatrix) otherMatrix;
   if ( matrix.length != m.matrix.length || matrix[0].length != m.matrix[0].length )
      return false;
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         if ( matrix[i][j] != m.matrix[i][j] )
            return false;
   return true;
}
```

Der Parameter der Methode equals von Klasse Object hat formalen Typ Object. Eine überschreibende Methode muss sich daran halten.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public boolean equals ( Object otherMatrix ) {
   if ( ! otherMatrix instanceof DMatrix )
      return false;
   DMatrix m = (DMatrix) otherMatrix;
   if ( matrix.length != m.matrix.length || matrix[0].length != m.matrix[0].length )
      return false;
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         if ( matrix[i][j] != m.matrix[i][j] )
            return false;
   return true;
}
```

Wir müssen daher erst einmal prüfen, ob der Parameter von der richtigen Klasse ist. Falls nein, sind die beiden Objekte natürlich von vornherein ungleich. Falls doch, müssen wir einen Downcast durchführen, um auf die Attribute des Parameters zuzugreifen.

***Nebenbemerkung*: In der Doku zu Klasse java.lang.Object finden Sie Anforderungen an eine Implementation von Methode equals, insbesondere dass equals eine Äquivalenzrelation im mathematischen Sinne konstituieren, also reflexiv, symmetrisch und transitiv sein soll. Solange keine Klassen von DMatrix abgeleitet werden beziehungsweise keine von DMatrix abgeleitete Klasse ihrerseits equals überschreibt, leistet unsere Implementation von equals das auch. Symmetrie wird allerdings gebrochen, wenn in einer von DMatrix abgeleiteten Klasse X Methode equals so überschrieben wird, dass auf „instanceof X“ statt „instanceof DMatrix“ getestet wird, was ja eigentlich naheliegend wäre.**

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public boolean equals ( Object otherMatrix ) {
   if ( ! otherMatrix instanceof DMatrix )
      return false;
   DMatrix m = (DMatrix) otherMatrix;
   if ( matrix.length != m.matrix.length || matrix[0].length != m.matrix[0].length )
      return false;
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         if ( matrix[i][j] != m.matrix[i][j] )
            return false;
   return true;
}
```

Wenn Zeilenzahl oder Spaltenzahl nicht übereinstimmen, wissen wir wieder, dass die beiden Matrizen nicht gleich sein können.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public boolean equals ( Object otherMatrix ) {
   if ( ! otherMatrix instanceof DMatrix )
      return false;
   DMatrix m = (DMatrix) otherMatrix;
   if ( matrix.length != m.matrix.length || matrix[0].length != m.matrix[0].length )
      return false;
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         if ( matrix[i][j] != m.matrix[i][j] )
            return false;
   return true;
}
```

Ansonsten sind die beiden Matrizen ungleich, wenn sie sich in mindestens einem Eintrag unterscheiden.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

```
public boolean equals ( Object otherMatrix ) {
   if ( ! otherMatrix instanceof DMatrix )
      return false;
   DMatrix m = (DMatrix) otherMatrix;
   if ( matrix.length != m.matrix.length || matrix[0].length != m.matrix[0].length )
      return false;
   for ( int i = 0; i < matrix.length; i++ )
      for ( int j = 0; j < matrix[0].length; j++ )
         if ( matrix[i][j] != m.matrix[i][j] )
            return false;
   return true;
}
```

Und wenn nirgendwo ein Unterschied ist, sind die beiden Matrizen gleich in dem Sinne, für den equals eigentlich gedacht ist – nämlich Wertgleichheit.

Damit ist die Klasse DMatrix fertig, ...

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

... und wir können zur Darstellungs- und Implementationsinvariante der Klasse DMatrix kommen.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Es ist generell eine gute Idee, die Darstellungsinvariante mit dieser oder einer ähnlichen Formulierung einzuleiten. Unabhängig davon, um welche Klasse oder welches Interface es gerade geht, ist damit erst einmal grundlegend gesagt, worauf sich alles Weitere bezieht.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Als nächstes kommt dann zweckmäßigerweise eine allgemeine Beschreibung, was man sich unter einem Objekt dieser Klasse vorzustellen hat.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Das ist eine wichtige Information, die nicht selbstverständlich ist: dass die Zeilen- und Spaltenzahl weder durch Methoden noch sonst irgendwie jemals während der Lebenszeit eines Matrix-Objektes geändert wird.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Sie erinnern sich: Unter anderem bei der Methode getNumberOfColumns unserer Matrixklasse eben haben wir vorausgesetzt, dass es mindestens eine Zeile gibt, und wir haben gesagt, dass das dann dokumentiert werden muss. Denn später, wenn man bei einer eventuellen Weiterentwicklung der Klasse DMatrix auch null Zeilen oder Spalten zulassen möchte, muss man daran denken und den Code der Klasse DMatrix daraufhin abklopfen, ob diese Voraussetzung nicht irgendwo verwendet wird. Der richtige Platz für diese Dokumentation ist die Darstellungsinvariante.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Auch dieser Punkt ist alles andere als selbstverständlich, sollte also explizit dokumentiert werden. In Java sind wir zwar daran gewöhnt, dass Indizes mit 0 beginnen, zum Beispiel bei eingebauten Arrays und beim Interface List aus Package java.util. Aber der Entwickler der Klasse DMatrix ist nicht gezwungen, sich an diese allgemeine Konvention zu halten. In der Mathematik und auch bei manchen anderen Programmiersprachen würde man die Indizes eher mit 1 beginnen lassen.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Nun zur Implementationsinvariante.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

***Alle* private-Attribute sollten in der Implementationsinvariante angesprochen werden. Ist ein Attribut nicht wichtig genug, um es hier anzusprechen, dann ist auch nicht wichtig, dass es in der Klasse ist, und dann sollten Sie es aus der Klasse entfernen. Umgekehrt gesprochen: Ist ein Attribut für die Klasse wesentlich, dann sollte es auch in der Implementationsinvariante vorkommen. Im konkreten Beispiel gibt es nur ein Attribut.**

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Die Typinformation ist natürlich redundant, könnte man auch weglassen.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Sie denken vielleicht, diese Informationen sind überflüssig, kann man ja auch im Konstruktor nachlesen. Umgekehrt wird ein Schuh draus: Diese Informationen sind zu verstehen als Instruktionen dafür, wie der Konstruktor und andere Methoden, die Attributwerte lesen oder überschreiben, zu implementieren sind.

Diese Dokumentation ist insbesondere wichtig, wenn man Fehler in der Implementation der Klasse suchen muss oder wenn man die Klasse weiterentwickeln will. Denn hier steht, auf welcher Logik der Code der Klasse momentan beruht. Bei der Fehlersuche kann man das tatsächliche Verhalten der Klasse Schritt für Schritt mit der Implementationsinvariante abgleichen, um den Fehler zu finden. Bei der Weiterentwicklung der Klasse ist die Implementationsinvariante der Ausgangspunkt aller Überlegungen und muss zusammen mit der Klasse weiterentwickelt werden – am Besten wirklich simultan, Schritt für Schritt.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel eigene Matrixklasse

- ***Darstellungsinvariante*: Ein Objekt von Klasse DMatrix repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von double; Zeilen- und Spaltenzahl sind konstant und größer 0; die Indizes sind ab 0 aufsteigend.**
- ***Implementationsinvariante*: Attribut matrix vom Typ double[ ][ ] hat als Länge die Zeilenzahl, und seine Komponenten haben als Länge die Spaltenzahl; matrix[i][j] ist der Eintrag in Zeile i und Spalte j.**

Auch solche Details sollte man bei der Formulierung der Implementationsinvariante nicht vergessen. Es wäre ja genauso gut möglich, dass die Rollen von i und j genau vertauscht sind. Eine Klarstellung kostet nicht viel Schreibaufwand und kann helfen, Missverständnisse zu vermeiden.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ableitung von Klassen / Implementation von Interfaces:

- **Die Darstellungsinvariante muss für die abgeleitete / implementierende Klasse übernommen werden.**
- **Die Implementationsinvariante muss bei protected-Attributen der Basisklasse übernommen werden.**
- **„Übernommen werden“ heißt:**
  - **Darf erweitert und verfeinert werden;**
  - **aber *nichts* darf zurückgenommen werden!**

Nach diesem ersten Beispiel wieder eine Grundsatzfrage: Wie sieht das mit Darstellungs- und Implementationsinvariante aus, wenn eine Klasse von einer anderen abgeleitet wird beziehungsweise ein Interface von einer Klasse implementiert wird?

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ableitung von Klassen / Implementation von Interfaces:

- **Die Darstellungsinvariante muss für die abgeleitete / implementierende Klasse übernommen werden.**
- **Die Implementationsinvariante muss bei protected-Attributen der Basisklasse übernommen werden.**
- **„Übernommen werden“ heißt:**
  - **Darf erweitert und verfeinert werden;**
  - **aber *nichts* darf zurückgenommen werden!**

Die abgeleitete beziehungsweise implementierende Klasse muss selbstverständlich die Darstellungsinvariante einhalten. Wird eine Methode der Basisklasse in der abgeleiteten Klasse überschrieben, dann muss der Effekt der überschreibenden Methode auf die Darstellungsinvariante der Basisklasse exakt derselbe sein wie bei der überschriebenen Methode.

Dies ist der erste Teil einer Regel, die Sie in der Literatur unter dem Namen Liskov Substitution Principle finden, benannt nach der Informatik-Professorin Barbara Liskov, abgekürzt LSP. Üblicherweise wird das LSP ungefähr so formuliert: Jede Aussage über das logische Verhalten der Basisklasse muss auch für die abgeleitete Klasse gelten. Gleich, bei der Korrektheit von Methoden, kommen wir darauf noch einmal zurück und schauen uns den zweiten Teil des LSP an.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ableitung von Klassen / Implementation von Interfaces:

- **Die Darstellungsinvariante muss für die abgeleitete / implementierende Klasse übernommen werden.**
- **Die Implementationsinvariante muss bei protected-Attributen der Basisklasse übernommen werden.**
- **„Übernommen werden“ heißt:**
  - **Darf erweitert und verfeinert werden;**
  - **aber *nichts* darf zurückgenommen werden!**

Ist ein Attribut protected definiert, dann kann es in der abgeleiteten Klasse ja gelesen und überschrieben werden. Dieses Attribut muss aber dennoch weiterhin die internen Regeln der Basisklasse erfüllen, und die sind in deren Implementationsinvariante festgelegt.

Für die private-Attribute der Basisklasse ist natürlich nichts zu übernehmen, da diese weiterhin hundertprozentig unter der Kontrolle der Basisklasse stehen.

Man muss sich also sehr genau überlegen, ob man ein Attribut wirklich protected und nicht private definiert. Denn falls man daran später etwas ändert, muss man potentiell alle abgeleiteten Klassen ebenfalls ändern.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ableitung von Klassen / Implementation von Interfaces:

- **Die Darstellungsinvariante muss für die abgeleitete / implementierende Klasse übernommen werden.**
- **Die Implementationsinvariante muss bei protected-Attributen der Basisklasse übernommen werden.**
- **„Übernommen werden“ heißt:**
  - **Darf erweitert und verfeinert werden;**
  - **aber *nichts* darf zurückgenommen werden!**

Zusammenfassend kann man es sicher so formulieren: Die Darstellungsinvariante und – soweit es die protected-Attribute betrifft – auch die Implementationsinvariante müssen in jedem einzelnen Punkt eingehalten werden.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ableitung von Klassen / Implementation von Interfaces:

- **Die Darstellungsinvariante muss für die abgeleitete / implementierende Klasse übernommen werden.**
- **Die Implementationsinvariante muss bei protected-Attributen der Basisklasse übernommen werden.**
- **„Übernommen werden“ heißt:**
  - **Darf erweitert und verfeinert werden;**
  - **aber *nichts* darf zurückgenommen werden!**

Was es mit dem Wort „verfeinert“ genau auf sich hat, sehen wir uns beim nächsten Beispiel an.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DSquareMatrix extends DMatrix {
   public DSquareMatrix ( int size ) {
      super ( size, size );
   }
   public int getSize () {
      return getNumberOfColumns ();
   }
}
```

Dieses nächste Beispiel greift noch einmal die eben definierte Matrix-Klasse auf.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DSquareMatrix extends DMatrix {
   public DSquareMatrix ( int size ) {
      super ( size, size );
   }
   public int getSize () {
      return getNumberOfColumns ();
   }
}
```

Und zwar leiten wir jetzt eine andere Klasse davon ab.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DSquareMatrix extends DMatrix {
   public DSquareMatrix ( int size ) {
      super ( size, size );
   }
   public int getSize () {
      return getNumberOfColumns ();
   }
}
```

Die abgeleitete Klasse verfeinert die Basisklasse dahingehend, dass Zeilenzahl und Spaltenzahl identisch sein müssen. Mit anderen Worten: Die Matrix muss quadratisch sein.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DSquareMatrix extends DMatrix {
   public DSquareMatrix ( int size ) {
      super ( size, size );
   }
   public int getSize () {
      return getNumberOfColumns ();
   }
}
```

Dies drückt sich implementatorisch einfach dadurch aus, dass der Konstruktor der abgeleiteten Klasse nur einen einzigen Parameter hat und der Konstruktor der Basisklasse denselben Wert für seine beiden Parameter, Zeilenzahl und Spaltenzahl, übergeben bekommt.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DSquareMatrix extends DMatrix {
   public DSquareMatrix ( int size ) {
      super ( size, size );
   }
   public int getSize () {
      return getNumberOfColumns ();
   }
}
```

Wir fügen noch etwas Funktionalität in Form dieser Methode hinzu, die offensichtlich redundant zu den beiden ererbten Methoden getNumberOfRows und getNumberOfColumns ist.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DSquareMatrix extends DMatrix {
   public DSquareMatrix ( int size ) {
      super ( size, size );
   }
   public int getSize () {
      return getNumberOfColumns ();
   }
}
```

Der Hintergedanke ist, dass size anstelle der Zeilen- und Spaltenzahl ja auch schon im Konstruktor der Name war. Nutzern, die nur die abgeleitete Klasse verwenden wollen, nicht die Basisklasse, bieten der Konstruktor und die Methode getSize ein einheitliches Look&Feel, sozusagen.

Das Liskov Substitution Principle ist erfüllt. Beachten Sie, dass das LSP von vornherein unmöglich zu erfüllen wäre, wenn die Basisklasse DMatrix Funktionalität zum Ändern der Zeilenzahl unabhängig von der Spaltenzahl oder umgekehrt bieten würde. Bei einer quadratischen Matrix lassen sich Zeilenzahl und Spaltenzahl prinzipiell nicht unabhängig voneinander ändern.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Jetzt ein Beispiel vom Beginn des Semester.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Die einleitende Formulierung ist dieselbe wie bisher auch.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Es ist vielleicht etwas unbefriedigend, dass hier nur von einem *Roboter* die Rede ist, ohne genauere Spezifikation. Aber letztendlich erfüllt die Formulierung der Darstellungsinvariante sicherlich auch so ihren Zweck, ...

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

... denn die nachfolgenden Informationen sollten eigentlich gut präzisieren können, was wir mit einem Roboter eigentlich meinen. Salopp gesagt, ist ein Roboter die Summe seiner Attributwerte.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Diese vier Attribute sind Ihnen aus den Kapiteln 01 wohlvertraut.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Die Veränderlichkeit dieser vier Attribute ist nicht selbstverständlich und sollte daher explizit erwähnt werden.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Offensichtlich gibt es aber noch einen anderen Bestandteil, der nach außen hin sichtbar ist, nämlich die Beziehung zur World, in der der Roboter zu sehen ist. Das Attribut selbst ist zwar im private-Bereich, aber sein Effekt ist nach außen hin zu sehen, nämlich das Zeichnen in die FopBot-World. Ein solcher nach außen sichtbarer Effekt sollte natürlich ebenfalls in die Darstellungsinvariante.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Auch hier ist wieder wichtig, explizit festzuhalten, ob die World ein für allemal fest oder auswechselbar ist.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

FopBot hat noch weitere Aspekte, die in die Darstellungsinvariante gehören, aber nicht im Kapiteln 01 thematisiert wurden und daher auch hier ausgelassen werden.

Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Klasse Robot aus FopBot:

- ***Darstellungsinvariante*: Ein Objekt von Klasse Robot repräsentiert zu jedem Zeitpunkt seiner Lebenszeit einen Roboter mit vier jederzeit veränderbaren Attributen: Zeile, Spalte, Richtung und Anzahl Münzen, sowie einem konstanten Attribut: der FopBot-World, zu der der Roboter gehört. .....**
- ***Implementationsinvariante*: abhängig von der tatsächlichen Implementation der Klasse Robot.**

Da wir die genaue Implementation der Klasse Robot nicht kennen, lassen wir die Implementationsinvariante hier aus.

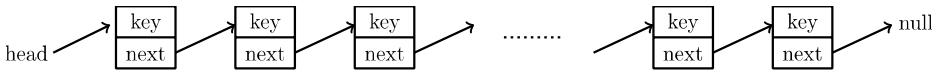
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



Als nächstes ein Beispiel aus der Standardbibliothek, mit dem wir uns schon in Kapitel 07 vertraut gemacht hatten.

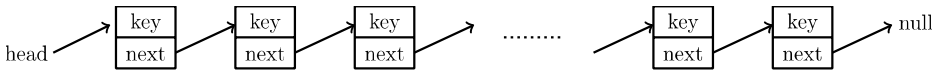
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



***Erinnerung*: List aus Package java.util ist ein Interface, das Klasse Collection erweitert.**

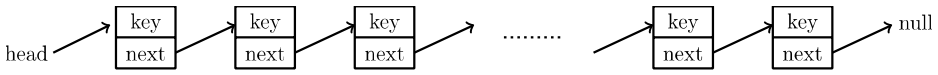
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



Die hier formulierte Darstellungsinvariante ist nicht offiziell, sondern vom Autor dieses Kapitels nach eigenem Verständnis des Interface List formuliert worden.

Zur Vermeidung von unkontrolliert redundanten Texten sollte man in der Darstellungsinvariante von List eigentlich auf die Darstellungsinvariante von Collection verweisen und nur die zusätzlichen Regeln für List gegenüber Collection – nämlich dass die Elemente Positionen haben – in die Darstellungsinvariante von List schreiben. Aber hier geht es nur ums Prinzip, deshalb schreiben wir hier die Darstellungsinvariante von List von Grund auf neu.

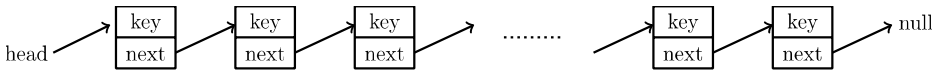
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



Wie immer dieselbe einleitende Formulierung für die Darstellungsinvariante.

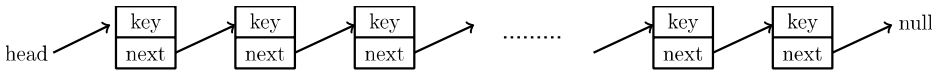
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



So hatten wir das Interface List und seine Implementationen kennen gelernt. Die implementierenden Klassen realisieren mit verschiedenen Implementationen – also mit völlig verschiedenen Implementationsinvarianten – jeweils dieselbe Art von Entität: Mengen, deren Elemente in eine Reihenfolge gebracht sind, im Gegensatz zu ungeordneten mathematischen Mengen.

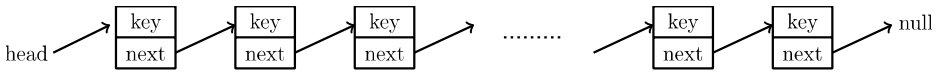
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



Da man sich in der Informatik mindestens seit den sechziger Jahren mit verschiedenen Implementationen von geordneten Sequenzen befasst, konnte man gut vorhersehen, dass die hier genannten Standardoperationen nicht in jeder Klasse, die irgendwann geschrieben wird, um das Interface List zu implementieren, in voller Allgemeinheit sinnvoll sind. Zum Beispiel wird nicht jede Implementation Duplikate erlauben; versucht man ein schon in der Liste vorhandenes Element nochmals einzufügen, wird eine Exception geworfen. Aber in der Darstellungsinvariante von List wird schon gesagt, dass kein try-catch notwendig ist, solange man mit Interface List und nicht explizit mit implementierenden Klassen arbeitet.

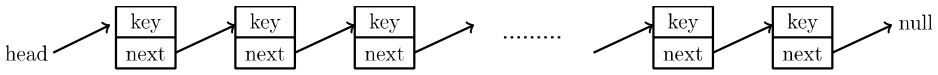
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf einer Runtime-Exception führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



Ein paar weitere Punkte müsste man hier wohl noch erwähnen, um die Darstellungsinvariante vollständig zu formulieren, lassen wir hier aus.

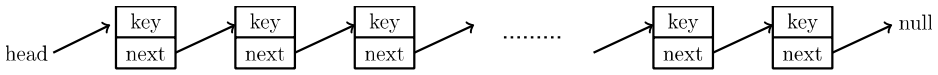
Korrektheit von Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel java.util.List

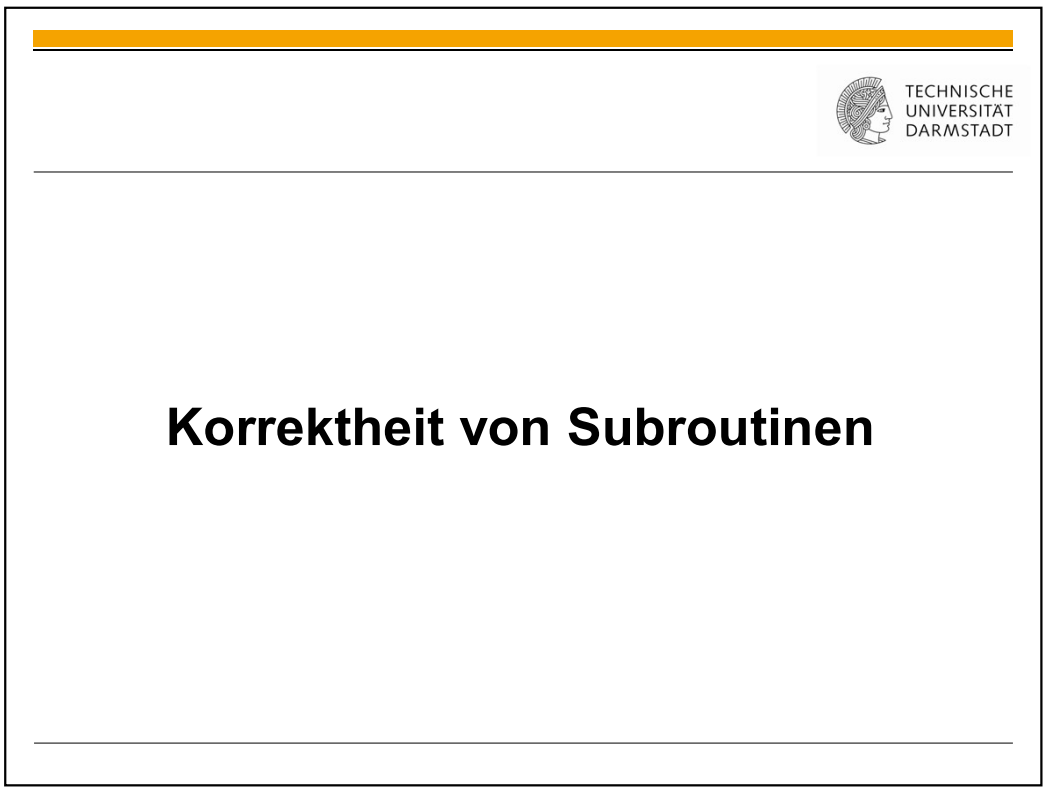
- ***Darstellungsinvariante*: Ein Objekt von List repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine geordnete Sequenz (die auch leer sein kann) seines generischen Elementtyps; die Positionen sind ab 0 aufsteigend; Suchen, Einfügen und Entfernen von Elementen kann zum Wurf von Exceptions führen; ...**

- ***Implementationsinvariante* von MyLinkedList:**



Wir hatten schon festgestellt, dass eine Implementationsinvariante für ein Interface wenig Sinn macht. Für eine Klasse, die das Interface implementiert, sieht das natürlich anders aus. In Kapitel 07 hatten wir eine Klasse ansatzweise definiert, die das Interface List implementiert und intern sehr ähnlich zu Klasse LinkedList organisiert ist.

Die Implementationsinvariante von MyLinkedList müsste also so etwas besagen wie, dass das Attribut head auf das erste Element einer korrekt gebildeten, einfach (nicht doppelt) verzeigerten linearen Liste verweist, deren Elementtytp ListItem ist, und das erste Element dieser internen Liste hat nach außen hin Position 0, das zweite Position 1, und so weiter.



Jetzt wenden wir uns Subroutinen zu. Den Begriff Subroutine verwenden wir hier als Oberbegriff für Funktionen in Racket und Methoden in Java sowie analoge Konstrukte in anderen Programmiersprachen.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorbedingungen:
  - Objektmethode → Implementationsinvariante
  - Parameter
  - Variable/Konstante außerhalb der Klasse
  - Externe Datenquellen (z.B. Dateien)
- Nachbedingungen:
  - Objektmethode → Implementationsinvariante
  - Rückgabewert
  - Variable außerhalb der Klasse
  - Externe Datensenken (z.B. Dateien)

***Erinnerung*: In Kapitel 04a hatten wir das Konzept des Vertrags eingeführt.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Vorbedingungen:**
  - Objektmethode → Implementationsinvariante
  - Parameter
  - Variable/Konstante außerhalb der Klasse
  - Externe Datenquellen (z.B. Dateien)
- **Nachbedingungen:**
  - Objektmethode → Implementationsinvariante
  - Rückgabewert
  - Variable außerhalb der Klasse
  - Externe Datensenken (z.B. Dateien)

Grob gesprochen stellt man sich vor, dass der Entwickler einer Subroutine und der Nutzer dieser Subroutine einen Vertrag schließen: Wenn der Aufruf der Subroutine alle Vorbedingungen erfüllt, dann muss die Subroutine alle Nachbedingungen erfüllen. Wir gehen alle Aspekte von Vor- und Nachbedingung kurz durch.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorbedingungen:
  - Objektmethode → Implementationsinvariante
  - Parameter
  - Variable/Konstante außerhalb der Klasse
  - Externe Datenquellen (z.B. Dateien)
- Nachbedingungen:
  - Objektmethode → Implementationsinvariante
  - Rückgabewert
  - Variable außerhalb der Klasse
  - Externe Datensenken (z.B. Dateien)

Methoden sind ja Subroutinen, die zu einer bestimmten Klasse gehören und Zugriff auf den nichtöffentlichen Teil der Klasse haben. Daher muss eine Methode einerseits als Vorbedingung darauf vertrauen, dass die Implementationsinvariante unmittelbar *vor* Aufruf der Methode eingehalten ist, und die Methode muss ihrerseits gewährleisten, dass die Implementationsinvariante unmittelbar *nach* Beendigung der Methode eingehalten ist.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorbedingungen:
  - Objektmethode → Implementationsinvariante
  - Parameter
  - Variable/Konstante außerhalb der Klasse
  - Externe Datenquellen (z.B. Dateien)
- Nachbedingungen:
  - Objektmethode → Implementationsinvariante
  - Rückgabewert
  - Variable außerhalb der Klasse
  - Externe Datensenken (z.B. Dateien)

So kennen Sie Verträge im Zusammenhang mit Racket-Funktionen aus Kapitel 04a: Die Parameter müssen Vorbedingungen erfüllen, die Rückgabe muss ein bestimmter Wert und kein anderer sein. Natürlich sind auch für Methoden in Java – und für Subroutinen jedweder Art in anderen Programmiersprachen – einerseits die Vorbedingungen an Parameter und andererseits die Spezifikation des Rückgabewertes hoch relevant. Der wesentliche Unterschied zwischen Racket und Java in diesem Punkt ist, dass in Racket auch die Typen in den Vertrag müssen, während in Java die Typen durch den Compiler geprüft werden und daher kein Gegenstand des Vertrags zwischen Entwickler und Nutzer der Subroutine sind.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorbedingungen:
  - Objektmethode → Implementationsinvariante
  - Parameter
  - Variable/Konstante außerhalb der Klasse
  - Externe Datenquellen (z.B. Dateien)
- Nachbedingungen:
  - Objektmethode → Implementationsinvariante
  - Rückgabewert
  - Variable außerhalb der Klasse
  - Externe Datensenken (z.B. Dateien)

Bei Java haben wir gesehen, dass eine Methode auch auf Variable und Konstante außerhalb der Klasse, zu der die Methode gehört, lesend und auf Variable auch schreibend zugreifen kann, nämlich public-Attribute anderer Klassen.

***Erinnerung*: Das gibt es prinzipiell auch bei Funktionen in Racket, siehe Kapitel 04d, Abschnitt „Aufweichung der reinen funktionalen Lehre in Racket“.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorbedingungen:
  - Objektmethode → Implementationsinvariante
  - Parameter
  - Variable/Konstante außerhalb der Klasse
  - Externe Datenquellen (z.B. Dateien)
- Nachbedingungen:
  - Objektmethode → Implementationsinvariante
  - Rückgabewert
  - Variable außerhalb der Klasse
  - Externe Datensenken (z.B. Dateien)

***Erinnerung*: In Kapitel 08 hatten wir ausführlich das Lesen von und das Schreiben auf Dateien behandelt und kurz angedeutet, dass sich auch bei Racket wie bei Java hinter Streams durchaus Dateien verbergen können.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vertrag:

- **Type**
  - **Nicht bei expliziter statischer Typisierung**
- **Precondition**
- **Returns**
- **Postcondition**
  - **Nicht bei referentieller Transparenz**

Schauen wir uns noch einmal das Thema Vertrag zusammenfassend an.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vertrag:

- **Type**
  - ➢ **Nicht bei expliziter statischer Typisierung**
- **Precondition**
- **Returns**
- **Postcondition**
  - ➢ **Nicht bei referentieller Transparenz**

Da der Java-Compiler die Typen prüft, gehören diese in Java nicht zum Vertrag, in Racket aber schon.

***Erinnerung*: Statische vs. dynamische Typisierung hatten wir im Kapitel 04d thematisiert und dabei auch kurz gesehen, dass es nicht nur Programmiersprachen wie Java mit *expliziter* statischer Typisierung gibt, sondern auch solche mit *impliziter*, Stichwort Typinferenz.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vertrag:

- **Type**
  - ➢ **Nicht bei expliziter statischer Typisierung**
- **Precondition**
- **Returns**
- **Postcondition**
  - ➢ **Nicht bei referentieller Transparenz**

Man könnte auch den Rückgabewert unter den Nachbedingungen subsumieren. Traditionell unterscheidet man hier aber.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vertrag:

- **Type**
  - ➢ **Nicht bei expliziter statischer Typisierung**
- **Precondition**
- **Returns**
- **Postcondition**
  - ➢ **Nicht bei referentieller Transparenz**

Neben der Rückgabe kann eine Subroutine auch Seiteneffekte haben, sofern nicht referentielle Transparenz eingehalten wird (vgl. Kapitel 04d).

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vor-/Nachbedingung von Methoden bei Ableitung von Basisklasse / Implementation von Interface:

- ***Vorbedingung*: darf nur abgeschwächt, nicht (teilweise) verschärft oder ersetzt werden.**
- ***Nachbedingung*: darf nur verschärft, nicht (teilweise) abgeschwächt oder ersetzt werden.**

Wir kommen nun zum angekündigten zweiten Teil des Liskov Substitution Principles, abgekürzt LSP. Wenn die abgeleitete beziehungsweise implementierende Klasse logisch konform zur Basisklasse beziehungsweise zum Interface sein soll, dann müssen auch die Methoden ihren Teil dazu beitragen.

Wird eine Methode von der Basisklasse an eine abgeleitete Klasse vererbt, dann erfüllt sie natürlich automatisch das Liskov Substiution Principle; problematisch kann es nur werden, wenn die Methode in der abgeleiteten Klasse überschrieben beziehungsweise in der ein Interface implementierenden Klasse implementiert wird.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vor-/Nachbedingung von Methoden bei Ableitung von Basisklasse / Implementation von Interface:

- ***Vorbedingung***: **darf nur abgeschwächt, nicht (teilweise) verschärft oder ersetzt werden.**
- ***Nachbedingung***: **darf nur verschärft, nicht (teilweise) abgeschwächt oder ersetzt werden.**

Für die Vorbedingung heißt das, dass sie gleich bleiben muss oder höchstens abgeschwächt werden darf. Jedenfalls darf die Vorbedingung in keinem einzigen Detail verschärft werden beziehungsweise es dürfen keine verschärfenden Details hinzukommen. Dann ist gewährleistet, dass der Nutzer der Methode den Vertrag vollständig einhält und nicht aus Versehen verletzt, wenn er die Methode nur über eine Referenz der Basisklasse beziehungsweise des Interface sieht und guten Glaubens den Vertrag gemäß Basisklasse beziehungsweise Interface einhält.

Ein einfaches Beispiel: Das Interface java.util.List hat eine Version der Methode add, die neben dem einzufügenden Wert noch den Index als Parameter hat, an dem der Wert einzufügen ist. In einer das Interface List implementierenden Klasse darf es keine zusätzlichen Bedingungen an den Index geben, etwa dass der Index nur gradzahlig sein darf oder ähnliche Scherze.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vor-/Nachbedingung von Methoden bei Ableitung von Basisklasse / Implementation von Interface:

- ***Vorbedingung***: **darf nur abgeschwächt, nicht (teilweise) verschärft oder ersetzt werden.**
- ***Nachbedingung***: **darf nur verschärft, nicht (teilweise) abgeschwächt oder ersetzt werden.**

Umgekehrt darf die Nachbedingung nur gleich bleiben oder enger gefasst werden. Damit ist gewährleistet, dass der Nutzer nach Aufruf der Methode über eine Referenz der Basisklasse beziehungsweise des Interface keine Überraschungen bei der Rückgabe oder den Seiteneffekten erlebt. Denn wenn die Nachbedingung auch nur in einem einzigen Detail abgeschwächt würde beziehungsweise Details aus der Nachbedingung ganz weggelassen würden, dann dürfte die Methode ja Effekte haben, die nicht dem Vertrag gemäß Basisklasse beziehungsweise Interface entsprechen.

Wieder ein einfaches Beispiel, ebenfalls Methode add von Interface java.util.List: In einer das Interface List implementierenden Klasse muss die Methode add den neuen Wert an dem Index einfügen, an dem er gemäß Vertrag in Interface List einzufügen ist. Die Methode add darf weder dieses Einfügen unterlassen noch den Wert an einem anderen Index einfügen noch einen weiteren Effekt über dieses Einfügen hinaus haben.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Fibonacci-Zahlen:

long fibonacci (int n) { ..... }

- ***Vorbedingung*: n ist nichtnegativ und darf nicht größer als 92 sein.**
- ***Nachbedingung*: Der Fibonacci-Wert von n wird zurückgeliefert.**

Ein erstes konkretes Beispiel zum Thema Korrektheit von Subroutinen, Sie erinnern sich: Die Fibonacci-Zahl von 0 und 1 ist jeweils 1, und die Fibonacci-Zahl einer Zahl n größer 1 ist die Summe aus den Fibonacci-Zahlen von n minus 1 und n minus 2.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Fibonacci-Zahlen:

long fibonacci (int n) { ..... }

- ***Vorbedingung*: n ist nichtnegativ und darf nicht größer als 92 sein.**
- ***Nachbedingung*: Der Fibonacci-Wert von n wird zurückgeliefert.**

Natürlich darf n nicht negativ sein.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Fibonacci-Zahlen:

long fibonacci (int n) { ..... }

- ***Vorbedingung*: n ist nichtnegativ und darf nicht größer als 92 sein.**
- ***Nachbedingung*: Der Fibonacci-Wert von n wird zurückgeliefert.**

Diese Vorbedingung ist vielleicht erst einmal rätselhaft. Die Auflösung des Rätsels ist aber ganz einfach: Die Folge der Fibonacci-Zahlen wächst so rasant schnell, dass selbst long – immerhin der größte primitive Datentyp für ganze Zahlen in Java – die Fibonacci-Zahlen ab 93 nicht mehr darstellen kann.

***Nebenbemerkung*: In Package java.math finden Sie eine Klasse namens BigInteger, die potentiell beliebig große ganze Zahlen darstellen kann, allerdings mit einer erheblich höheren Laufzeit für arithmetische Operationen, Vergleichsoperationen, und so weiter.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Fibonacci-Zahlen:

long fibonacci (int n) { ..... }

- ***Vorbedingung*: n ist nichtnegativ und darf nicht größer als 92 sein.**
- ***Nachbedingung*: Der Fibonacci-Wert von n wird zurückgeliefert.**

Die Nachbedingung gibt explizit an, was die Subroutine alles macht. Sie besagt aber immer auch implizit, dass die Subroutine *keine anderen* Effekte hat. Der Nutzer einer Subroutine darf darauf vertrauen, dass die Subroutine nur die in der Nachbedingung aufgelisteten Effekte und keine weiteren hat.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: **cmp.compare darf für *kein* Paar von Listenelementen eine ClassCastException werfen.**
- ***Nachbedingung***: **dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.**

Als nächstes ein Beispiel aus der Java-Standardbibliothek.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: cmp.compare darf für *kein* Paar von Listenelementen eine ClassCastException werfen.
- ***Nachbedingung***: dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.

***Erinnerung*: In Kapitel 07, Abschnitt zu Sortieren mit Comparator, hatten wir diese Klassenmethode der Klasse Collections aus dem Package java.util schon genau so wie hier gesehen.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: cmp.compare darf für ***kein*** Paar von Listenelementen eine ClassCastException werfen.
- ***Nachbedingung***: dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.

Wie die zu sortierende Liste und der Comparator, der die Sortierlogik vorgibt, zustande kommen, muss hier nicht nochmals thematisiert werden.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: **cmp.compare darf für *kein* Paar von Listenelementen eine ClassCastException werfen.**
- ***Nachbedingung***: **dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.**

In der Oracle-Dokumentation von Interface Comparator finden Sie diese Vorbedingung, die letztendlich besagt, dass der Comparator wirklich zum Referenztyp MyType passt.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: cmp.compare darf für *kein* Paar von Listenelementen eine ClassCastException werfen.
- ***Nachbedingung***: dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.

Das ist die konstituierende Nachbedingung für eine Sortiermethode: Die Elemente der Liste müssen hinterher aufsteigend sortiert sein gemäß Logik der verwendeten Comparator-Klasse.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: cmp.compare darf für *kein* Paar von Listenelementen eine ClassCastException werfen.
- ***Nachbedingung***: dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.

Dieser Punkt wird häufig übersehen, ist aber eigentlich unabdingbar: dass die Liste hinterher aus exakt denselben Elementen wie vorher besteht, also keines verschwunden und keines hinzugekommen ist, nur eben umgeordnet.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
List<MyType> list = .....;
Comparator<MyType> cmp = .....;
Collections.sort ( list, cmp );
```

- ***Vorbedingung***: cmp.compare darf für *kein* Paar von Listenelementen eine ClassCastException werfen.
- ***Nachbedingung***: dieselben Elemente, aber in aufsteigender Reihung, das heißt, für jede Position i > 0 ist cmp.compare (list,get(i-1), list.get(i)) ≤ 0; der Sortieralgorithmus ist stabil.

Es gibt Sortieralgorithmen, die in folgendem Sinne stabil sind: Wenn zwei Elemente in der Liste sind, bei denen Methode compare des Comparators den Wert 0 zurückliefert, dann ändert sich ihre relative Reihenfolge durch den Sortieralgorithmus nicht. Das heißt, das Element, das unmittelbar *vor* Aufruf von sort das vordere dieser beiden ist, ist auch unmittelbar *nach* dem Aufruf von sort das vordere dieser beiden.

Dieser Teil der Nachbedingung besagt also, dass der Implementierer der Methode sort nicht ganz frei in der Wahl des Sortieralgorithmus ist, sondern einen in diesem Sinne stabilen Sortieralgorithmus wählen muss.

Beachten Sie, dass bei einem stabilen Sortieralgorithmus nur *eine* mögliche Umordnung als Ergebnis korrekt ist, während bei einem nichtstabilen Sortieralgorithmus Werte, die vom Comparator nicht unterschieden werden können, in beliebiger Reihenfolge aufeinander folgen dürfen.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation “precedes“ is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;        list, but sorted in ascending order according to “precedes“;
;;        the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

Dasselbe Beispiel, Sortieren einer Liste, jetzt noch einmal, aber in Racket.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation "precedes" is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;       list, but sorted in ascending order according to "precedes";
;;       the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

***Erinnerung*: „iff“ ist eine gängige Abkürzung für „if and only if“, also „genau dann, wenn“.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation “precedes“ is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;        list, but sorted in ascending order according to “precedes“;
;;        the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

Eine Liste kommt hinein, eine Liste kommt heraus.

***Erinnerung*: Mit einzelnen Großbuchstaben als Typnamen wird per Konvention in Racket-Kommentaren angegeben, dass alle Elemente der Liste vom selben Typ sind, dieser Typ aber offengehalten ist.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation "precedes" is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;       list, but sorted in ascending order according to "precedes";
;;       the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

Hinzu kommt eine Funktion, die dasselbe leisten soll wie Comparator in Java.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation "precedes" is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;       list, but sorted in ascending order according to "precedes";
;;       the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

Die Logik des zweiten Parameters soll analog zu Methode compare von Comparator sein.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation "precedes" is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;       list, but sorted in ascending order according to "precedes";
;;       the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

Diesen Passus hatten wir nicht bei Methode sort in Java; brauchten wir auch nicht, denn zum Beispiel in der Oracle-Dokumentation von Interface Comparator in Package java.util können Sie Anforderungen an Methode compare nachlesen, die von jeder Klasse, die Comparator implementiert, erfüllt werden müssen. Daher mussten diese Anforderungen nicht noch einmal bei den Vorbedingungen für die Methode sort von Klasse Collections stehen.

Für die Summe dieser Anforderungen gibt es den Begriff strikte schwache Ordnung in der Mathematik. Grob gesprochen, ist dies eine lineare Ordnung, bei der auch mehrere Werte an derselben Stelle in der Ordnung sein können. Die Werte an derselben Stelle sind dann natürlich untereinander unvergleichbar.

***Erinnerung*: In Kapitel 07, Abschnitt zu Sortieren mit Comparator, hatten wir ein Beispiel dafür gesehen: eine Comparator-Klasse, deren Methode compare nur Nach- und Vorname zweier Personen miteinander vergleicht und daher zwei Personen mit demselben Nach- und Vornamen nicht unterscheiden kann.**

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

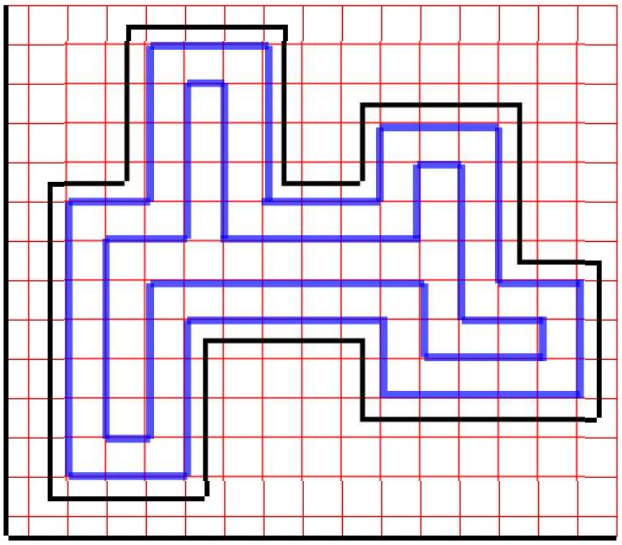
Sortieren von Listen:

```
;; Type: ( list of X ) ( X X -> integer ) -> ( list of X )
;; Precondition: cmp returns a value < 0 iff the first parameter precedes
;;       the second one and a value > 0 iff the second parameter precedes
;;       the first one; the relation "precedes" is a strict weak ordering.
;; Returns: a list whose elements are exactly the elements of
;;       list, but sorted in ascending order according to "precedes";
;;       the sorting algorithm is stable.

( define ( sort list cmp ) ( ..... ) )
```

Die Nachbedingung reduziert sich auf eine Beschreibung des Rückgabewertes und ist analog zur Nachbedingung bei der Java-Methode sort, die wir eben gesehen hatten.

Korrektheit von Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT



Noch ein letztes Beispiel in diesem Abschnitt.

***Erinnerung*: In Kapitel 01c hatten wir uns die Aufgabe gestellt, einen abgegrenzten Raum mit Beepern zu füllen. Das war ein Beispiel für eine komplexere Vorbedingung: Die Räume mussten im Prinzip so wie hier strukturiert sein, das heißt, die Abstandslinien müssen sich überall berühren, und keine Abstandslinie darf auf sich selbst verlaufen. Auch die Nachbedingung war komplexer: Auf jeder Kreuzung innerhalb des abgegrenzten Raumes soll genau ein Beeper platziert sein.**

Dies ist ein Beispiel dafür, dass es bei Vor- und Nachbedingung nicht immer nur um Parameter und Rückgabe geht, sondern beispielsweise so wie hier auch um die Attribute einer anderen Klasse, in diesem Fall der Klasse World.

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von rekursiven Subroutinen

Beim Thema Korrektheit von Subroutinen erfordert ein Fall besondere Aufmerksamkeit: Rekursion. Rekursion haben wir in Kapitel 04a ausgiebig im Zusammenhang mit Java und Racket kennen gelernt. Hier nehmen wir uns vorrangig Beispiele aus Racket her.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion

- **Rekursionsabbruch**
- **Rekursionsschritt**

```
( define ( factorial n ) ( if ( = n 0 ) 1 ( * n ( factorial ( − n 1 ) ) ) ) )
```

```
( define ( sum list )
    ( if ( empty? list ) 0 ( + ( first list ) ( sum ( rest list ) ) ) ) )
```

Die beiden Beispiele auf dieser Folie sollten wohlbekannt sein aus Kapitel 04a beziehungsweise 04b.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion

- Rekursionsabbruch
- Rekursionsschritt

```
( define ( factorial n ) ( if ( = n 0 ) 1 ( * n ( factorial ( − n 1 ) ) ) ) )
```

```
( define ( sum list )
    ( if ( empty? list ) 0 ( + ( first list ) ( sum ( rest list ) ) ) ) )
```

Damit eine Rekursion ordentlich terminiert, muss es einen Rekursionsabbruch geben. Das heißt, eine Bedingung wird abgefragt, und falls diese erfüllt ist, bricht die Rekursion ab.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion

- **Rekursionsabbruch**
- **Rekursionsschritt**

```
( define ( factorial n ) ( if ( = n 0 ) 1 ( * n ( factorial ( − n 1 ) ) ) ) )
```

```
( define ( sum list )
    ( if ( empty? list ) 0 ( + ( first list ) ( sum ( rest list ) ) ) ) )
```

Der für uns hier wichtige Punkt ist, dass der rekursive Aufruf ein ausreichendes Stück näher an der Abbruchbedingung dran ist, so dass nach endlich vielen Schritten der Rekursionsabbruch erreicht ist.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion

- **Rekursionsabbruch**
- **Rekursionsschritt**

```
( define ( factorial n ) ( if ( = n 0 ) 1 ( * n ( factorial ( − n 1 ) ) ) ) )

( define ( sum list )
    ( if ( empty? list ) 0 ( + ( first list ) ( sum ( rest list ) ) ) ) )
```

Bei der Fakultätsfunktion wird n um 1 heruntergezählt. Vorbedingung bei der Fakultätsfunktion ist ja, dass der Parameter eine nichtnegative ganze Zahl sein muss. Sofern diese Vorbedingung erfüllt ist, führt Herunterzählen um 1 nach so vielen Schritten zur Abbruchbedingung, wie die Zahl n beim ersten rekursiven Aufruf groß ist.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion

- **Rekursionsabbruch**
- **Rekursionsschritt**

```
( define ( factorial n ) ( if ( = n 0 ) 1 ( * n ( factorial ( − n 1 ) ) ) ) )
```

```
( define ( sum list )
    ( if ( empty? list ) 0 ( + ( first list ) ( sum ( rest list ) ) ) ) )
```

Ähnlich sieht das bei den typischen rekursiven Durchläufen durch Listen aus, so wie auch hier. Die Länge der Liste wird bei jedem rekursiven Aufruf um 1 kürzer und erreicht daher nach so vielen Schritten die Abbruchbedingung, wie die Liste ursprünglich lang war.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define epsilon 0.00001 )

;; Type: ( real -> real ) real real -> real
;; Precondition:
;;      a < b;
;;      f is a continuous function in the interval [ a ... b ];
;;      it is f(a) * f(b) <= 0.
;; Returns: a value in the interval [ a ... b ] that differs from
;; some zero of f by no more than epsilon.

( define ( find-zero f a b ) ( ..... ) )
```

***Erinnerung*: Die Funktion find-zero mit diesem Vertrag hatten wir in Kapitel 04a gesehen, Abschnitt zu Rekursion in Racket. Implementiert wird das Bisektionsverfahren aus der Mathematik zum näherungsweisen Finden einer Nullstelle einer stetigen Funktion in einem Intervall mit Vorzeichenwechsel.**

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( cond
        [ ( < ( − b a ) epsilon ) m ]
        [  ( > ( * ( f a ) ( f m ) ) 0 ) ( find-zero f m b ) ]
        [ else ( find-zero f a m ) ] ) ) )
```

***Weiter Erinnerung*: Das war im Prinzip die Implementation von find-zero, nur dass die beiden ineinandergeschachtelten if-Ausdrücke durch einen äquivalenten cond-Ausdruck ersetzt sind.**

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von find-zero:

- **Kein Programmabbruch mit Fehlermeldung.**
- **Terminiert nach endlich vielen Schritten.**
- **Ergebnis bei Termination ist korrekt.**

.....

[(< (− b a) epsilon) m]

.....

Unser Thema in diesem Kapitel ist natürlich die Korrektheit der Funktion find-zero. Sie erinnern sich, dass drei Kriterien überprüft werden müssen.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von find-zero:

- **Kein Programmabbruch mit Fehlermeldung.**
- **Terminiert nach endlich vielen Schritten.**
- **Ergebnis bei Termination ist korrekt.**

.....

[(< (− b a) epsilon) m]

.....

Sie können sich durch nochmalige Inspektion der Implementation von Funktion find-zero sicherlich selbst davon überzeugen, dass das erste Kriterium durch find-zero eingehalten wird, falls die Vorbedingung von find-zero erfüllt ist. Denn das einzig kritische bei der doch recht einfachen Funktion find-zero ist die Anwendung von arithmetischen Operatoren und Vergleichsoperatoren; aber nach Vorbedingung sind die Typen der Parameter kompatibel mit den Operatoren. Es kann nichts passieren, solange der Nutzer seinen Teil des Vertrags einhält.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von find-zero:

- **Kein Programmabbruch mit Fehlermeldung.**
- **Terminiert nach endlich vielen Schritten.**
- **Ergebnis bei Termination ist korrekt.**

.....

[(< (− b a) epsilon) m]

.....

Dreh- und Angelpunkt für das zweite und dritte Kriterium ist der Rekursionsabbruch, daher ist der zur Erinnerung auf dieser Folie noch einmal zitiert.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von find-zero:

- Kein Programmabbruch mit Fehlermeldung.
- Terminiert nach endlich vielen Schritten.
- Ergebnis bei Termination ist korrekt.

.....

[(< (− b a) epsilon) m]

.....

Wir haben gesehen, dass die Differenz zwischen a und b in jedem rekursiven Schritt halbiert wird. Falls die ursprünglichen Werte von a und b um nicht mehr als epsilon auseinanderliegen, gibt es *null* rekursive Schritte; falls nicht mehr als zweimal epsilon auseinander, gibt es *einen* rekursiven Schritt; falls nicht mehr als viermal epsilon auseinander, gibt es *zwei* rekursive Schritte; falls nicht mehr als achtmal epsilon auseinander, gibt es *drei* rekursive Schritte; und so weiter.

Dieses Bildungsgesetz lässt sich verallgemeinern: Falls die ursprünglichen Werte von a und b um nicht mehr als 2 hoch n mal epsilon auseinanderliegen, gibt es nur n rekursive Schritte. Diesen Sachverhalt kann man natürlich auch mit der Umkehrfunktion der Exponentiation formulieren: Sei x der Quotient aus b minus a im Zähler und epsilon im Nenner, dann gibt es Zweier-Logarithmus von x aufgerundet viele rekursive Schritte.

Insbesondere ist Termination gewährleistet.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von find-zero:

- **Kein Programmabbruch mit Fehlermeldung.**
- **Terminiert nach endlich vielen Schritten.**
- **Ergebnis bei Termination ist korrekt.**

.....

[(< (− b a) epsilon) m]

.....

Dass Termination gewährleistet ist, bedeutet nichts anderes, als dass die Abbruchbedingung irgendwann erreicht wird. Nach Vorbedingung wissen wir, dass zwischen a und b mindestens eine Nullstelle von f zu finden ist. Die Funktion find-zero liefert bei Rekursionsabbruch einen Wert zurück, der garantiert nicht weiter als epsilon von jedem Punkt im Intervall von a nach b entfernt ist, also auch von mindestens einer Nullstelle der Funktion f nicht weiter als epsilon entfernt.

***Nebenbemerkung*: Natürlich hätten wir uns noch einen rekursiven Schritt sparen können, denn tatsächlich ist der Rückgabewert höchstens epsilon-halbe von einer Nullstelle entfernt.**

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Induktionsbehauptung***: Für Induktionsparameter = Problemgröße h ≥ 0 erfüllt die Subroutine ihren Vertrag.
- ***Induktionsanfang***, also h = 0: Die Anweisungen im Rekursionsabbruch sorgen dafür, dass der Vertrag erfüllt ist.
- ***Induktionsvoraussetzung*** für h > 0: Der Vertrag gelte für 0, 1, ..., h-1.
- ***Induktionsschritt*** h > 0: Unter der Voraussetzung, dass der Vertrag für 0, 1 ..., h-1 gilt, sorgen die Anweisungen im Rekursionsschritt dafür, dass er auch für h gilt.

Zum Abschluss des Abschnitts über die Korrektheit von rekursiven Subroutinen noch eine kurze theoretische Einordnung dessen, was wir hier an verschiedenen Beispielen gesehen haben.

Auch wenn man es sich häufig nicht bewusst macht: Hinter der Korrektheit einer rekursiven Subroutine steckt grundsätzlich immer das aus der Mathematik bekannte Beweisprinzip der vollständigen Induktion.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Induktionsbehauptung***: **Für Induktionsparameter = Problemgröße h ≥ 0 erfüllt die Subroutine ihren Vertrag.**
- ***Induktionsanfang***, **also h = 0: Die Anweisungen im Rekursionsabbruch sorgen dafür, dass der Vertrag erfüllt ist.**
- ***Induktionsvoraussetzung*** **für h > 0: Der Vertrag gelte für 0, 1, ..., h-1.**
- ***Induktionsschritt*** **h > 0: Unter der Voraussetzung, dass der Vertrag für 0, 1 ..., h-1 gilt, sorgen die Anweisungen im Rekursionsschritt dafür, dass er auch für h gilt.**

Mit vollständiger Induktion lassen sich Aussagen über die natürlichen Zahlen beweisen. Wir brauchen daher einen natürlichzahligen Induktionsparameter. Im Zusammenhang mit rekursiven Subroutinen bezeichnen wir den Induktionsparameter etwas lebensnäher als *Problemgröße*. Wir hatten drei Beispiele für die Korrektheit von rekursiven Subroutinen gesehen:

Beim Beispiel Fakultät ist einfach der Eingabeparameter, für den die Fakultät berechnet werden soll, die Problemgröße.

Beim Beispiel Summation der Listenelemente ist die Länge der Liste die Problemgröße.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Induktionsbehauptung***: **Für Induktionsparameter = Problemgröße h ≥ 0 erfüllt die Subroutine ihren Vertrag.**
- ***Induktionsanfang***, **also h = 0: Die Anweisungen im Rekursionsabbruch sorgen dafür, dass der Vertrag erfüllt ist.**
- ***Induktionsvoraussetzung*** **für h > 0: Der Vertrag gelte für 0, 1, ..., h-1.**
- ***Induktionsschritt*** **h > 0: Unter der Voraussetzung, dass der Vertrag für 0, 1 ..., h-1 gilt, sorgen die Anweisungen im Rekursionsschritt dafür, dass er auch für h gilt.**

Bei der Intervallhalbierung ist die geeignete Wahl des Induktionsparameters weniger offensichtlich, denn wir haben es hier mit reellen und nicht mit natürlichen Zahlen zu tun, und da wird auch nichts einfach hochgezählt. Aber wir haben den geeigneten Parameter schon herauskristallisiert: der aufgerundete Logarithmus des Quotienten aus Intervallgröße und epsilon. Bei der Besprechung der weiteren Punkte auf dieser Folie wird klarwerden, warum diese etwas komplizierte Zahl geeignet ist, aber im Grunde wird das nur die Systematisierung unserer seinerzeitigen informellen Argumentation sein, warum die Intervallhalbierung terminiert.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Induktionsbehauptung***: Für Induktionsparameter = Problemgröße h ≥ 0 erfüllt die Subroutine ihren Vertrag.
- ***Induktionsanfang***, also h = 0: Die Anweisungen im Rekursionsabbruch sorgen dafür, dass der Vertrag erfüllt ist.
- ***Induktionsvoraussetzung*** für h > 0: Der Vertrag gelte für 0, 1, ..., h-1.
- ***Induktionsschritt*** h > 0: Unter der Voraussetzung, dass der Vertrag für 0, 1 ..., h-1 gilt, sorgen die Anweisungen im Rekursionsschritt dafür, dass er auch für h gilt.

Bei der Fakultätsfunktion wurde im Fall, dass der Parameter 0 ist, die korrekte Zahl 1 zurückgeliefert, das passt.

Beim Beispiel Summation der Listenelemente wurde 0 zurückgeliefert für den Fall, dass die Liste leer, die Listengröße also 0 ist. Auch das passt.

Bei der Intervallhalbierung bedeutet Induktionsparameter 0, dass die Intervallgröße nicht größer als epsilon ist, was ebenfalls zum passenden Ergebnis führt.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Induktionsbehauptung***: Für Induktionsparameter = Problemgröße h ≥ 0 erfüllt die Subroutine ihren Vertrag.
- ***Induktionsanfang***, also h = 0: Die Anweisungen im Rekursionsabbruch sorgen dafür, dass der Vertrag erfüllt ist.
- ***Induktionsvoraussetzung*** für h > 0: Der Vertrag gelte für 0, 1, ..., h-1.
- ***Induktionsschritt*** h > 0: Unter der Voraussetzung, dass der Vertrag für 0, 1 ..., h-1 gilt, sorgen die Anweisungen im Rekursionsschritt dafür, dass er auch für h gilt.

Der entscheidende Kniff beim allgemeinen Beweisprinzip der vollständigen Induktion ist die Induktionsvoraussetzung: Wir nehmen uns im Induktionsschritt irgendeinen beliebigen echt positiven Wert h her, denn der Fall h = 0 wird ja im Induktionsanfang behandelt. Wir nehmen einfach an, dass die zu beweisende Aussage schon für alle natürlichen Zahlen kleiner h gilt.

Das dürfen wir aufgrund folgender Überlegung: Nehmen wir uns h = 357 her, nur so als Beispiel. Nach Induktionsanfang gilt der Vertrag für 0, nach Induktionsschritt also auch für 1, nach Induktionsschritt also auch für 2, nach Induktionsschritt also auch für 3, und so weiter, dann kommen wir irgendwann zu 356, und nach Induktionsschritt gilt der Vertrag dann auch für 357.

Korr. von rekursiven Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Induktionsbehauptung***: Für Induktionsparameter = Problemgröße h ≥ 0 erfüllt die Subroutine ihren Vertrag.
- ***Induktionsanfang***, also h = 0: Die Anweisungen im Rekursionsabbruch sorgen dafür, dass der Vertrag erfüllt ist.
- ***Induktionsvoraussetzung*** für h > 0: Der Vertrag gelte für 0, 1, ..., h-1.
- ***Induktionsschritt*** h > 0: Unter der Voraussetzung, dass der Vertrag für 0, 1 ..., h-1 gilt, sorgen die Anweisungen im Rekursionsschritt dafür, dass er auch für h gilt.

Wir müssen nur einmal für beliebiges, aber festes h größer 0 einsehen, dass der Vertrag für h erfüllt ist, sofern er für alle natürlichen Zahlen kleiner h erfüllt ist. Genau dafür muss der Rekursionsschritt sorgen. Mit anderen Worten: Der Rekursionsschritt muss die Problemgröße h auf eine Problemgröße kleiner h korrekt zurückführen.

Bei der Fakultätsfunktion wird die Fakultät rekursiv mit dem um 1 verringerten Parameterwert aufgerufen, das passt.

Bei der Summation der Listenelemente wird die Liste für den rekursiven Aufruf um ein Element verkürzt, auch das passt.

Und schließlich bei der Intervallhalbierung sorgt die Halbierung der Intervallgröße dafür, dass der aufgerundete Logarithmus des Quotienten aus Intervallgröße und epsilon genau um 1 geringer wird, passt also ebenfalls wieder.

Mit dieser allgemeinen Betrachtung beenden wir den Abschnitt zur Korrektheit rekursiver Funktionen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit innerhalb von Subroutinen

Zur Analyse der Korrektheit von Software reicht es natürlich nicht, sich Klassen und Subroutinen als Black Boxes anzuschauen; wir müssen auch *hinein*schauen.

Korrektheit *in* Subroutinen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Korrektheit von Ausdrücken: völlig analog zu Subroutinen mit Rückgabe
- Korrektheit bei Verzweigungen (if, cond, switch): genau dann, wenn jede Alternative korrekt ist.

Fangen wir mit den zwei einfachsten Fällen an: Ausdrücke und Verzweigungen. Dazu brauchen wir wohl weder genauere Erläuterungen noch Beispiele.

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit speziell von Schleifen

Der eigentlich schwierige Fall sind Schleifen, daher widmen wir Schleifen einen eigenen Abschnitt.

Korrektheit von Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Invariante (Schleifeninvariante)
- Variante (Schleifenvariante)

Zwei Begriffe sind für die Analyse der Korrektheit von Schleifen grundlegend: die Schleifeninvariante und die Schleifenvariante. Wie die beiden Namen schon sagen, beinhaltet die Schleifeninvariante Aussagen darüber, was sich während der Schleife *nicht* ändert, und die Schleifenvariante, was sich in jedem Schleifendurchlauf *ändert*.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a )
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Hier ein erstes, einfaches Beispiel: In einer Schleife sollen alle Komponenten eines Arrays von double aufsummiert werden. Das Ergebnis soll in der lokalen Variable result stehen.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a )
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Die Variante ist bei Zählschleifen in der Regel trivial und nicht weiter zu analysieren. Gleich werden wir sehen, dass das bei while-Schleifen anders ist.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a )
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Die Invariante ist bei dieser Schleife wesentlich interessanter.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a )
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

So leiten wir die Formulierung eine Schleifeninvariante immer ein. Dazu nehmen wir uns eine Variable her, die im Code nicht vorkommt, um Mehrdeutigkeiten, die zu Missverständnissen führen können, zu vermeiden.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a ) {
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Das ist der eigentliche Inhalt der Invariante. Und wenn diese Aussage für jedes h stimmt, „„

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a ) {
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

... dann stimmt sie auch nach dem letzten Schleifendurchlauf, also wenn die Schleife so oft durchlaufen ist, wie das Array lang ist. Die Invariante geht dann nahtlos über in die Aussage, dass tatsächlich in result wie gefordert die Summe *aller* Arraykomponenten steht.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a ) {
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Die Frage ist noch, was es mit dem Fall h gleich 0 auf sich hat, also nach 0 Schleifendurchläufen.

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a ) {
   double result = 0;
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Nach 0 Durchläufen heißt: unmittelbar vor dem ersten Durchlauf. In dieser Situation steht in result der Wert 0. Laut Invariante müsste in result die Summe der Komponenten von a vom Index 0 bis zum Index -1 stehen, also die Summe über einen leeren Indexbereich. In der Mathematik ist die Summe über einen leeren Indexbereich als 0 definiert, passt also.

***Nebenbemerkung*: Die Definition in der Mathematik, dass eine Summe über einen leeren Indexbereich den Wert 0 hat, ist selbstverständlich nicht willkürlich, sondern so, dass sie eben mit Summen über *nicht*leere Indexbereiche bei allen mathematischen Rechnungen zusammenpasst. Nur ein Beispiel: Damit ergibt die Vereinigung eines leeren Indexbereichs mit einem nichtleeren Indexbereich dieselbe Summe wie der nichtleere Indexbereich allein.**

Beispiel Summe im Array

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double sum ( double[ ] a ) {
  double result = 0;
  for ( int i = 0; i < a.length; i++ )
    result += a[i];
  return result;
}
```

- Invariante: Nach h ≥ 0 Schritten ist result == a[0] + ... + a[h-1].
- Variante: h steigt um 1.

→ Nach Schleifenende ist result == a[0] + ... + a[a.length-1].

Man kann es auch so herum sehen: Aus der Invariante leitet sich ab, wie die betroffenen Variablen vor Beginn der Schleife zu initialisieren sind und wie deren Werte in jedem Schleifendurchlauf abzuändern sind – nämlich gerade so, dass die Invariante von Anfang an und von Schleifendurchlauf zu Schleifendurchlauf eingehalten wird.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Nächstes Beispiel, wieder schrittweise Berechnung eines Zahlenwertes in einer Schleife, aber etwas komplizierter als beim letzten Beispiel.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Berechnet werden soll die Fibonacci-Zahl des Parameters. Wir hatten diese Methode schon weiter vorne in diesem Kapitel, bei der Korrektheit von rekursiven Subroutinen, hier jetzt aber realisiert durch eine Schleife.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Der Parameter ist vom primitiven Datentyp int, und der Laufindex i kann daher ebenfalls einfach vom Typ int sein. Denn wie wir weiter vorne schon festgestellt hatten, muss der Vertrag n auf den Bereich von 0 bis 92 begrenzen, ...

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

... und auch das nur, wenn für alle Variablen, in denen das Endergebnis aufgebaut wird, Datentyp long verwendet wird.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Die Grundidee ist, dass *nach* jedem Schleifendurchlauf in fibOfI die Fibonacci-Zahl des aktuellen Wertes des Laufindex i steht und in fibOfIMinus1 entsprechend der vorhergehende Fibonacci-Wert.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Die jeweils nächste Fibonacci-Zahl wird gemäß Definition durch Summation der beiden vorhergehenden Fibonacci-Zahlen berechnet.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Jetzt müssen noch die beiden Variablen aktualisiert werden, damit sie nach dem Schleifendurchlauf die Werte enthalten, die sie enthalten sollen. Machen Sie sich klar, dass die Reihenfolge dieser beiden Anweisungen wichtig ist!

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Auch in diesem Beispiel ist die Schleife wieder eine einfache Zählschleife, so dass die Variante wieder trivial ist.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Die Invariante ist eins-zu-eins identisch mit unserer Grundidee. Das ist kein Zufall, sondern sogar eher die Regel. In gewisser Weise ist die Invariante die abstrakte Idee hinter einer Schleife.

Beispiel Fibonacci

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static long fibonacci ( int n ) {
   long fibOfIMinus1 = 1;
   long fibOfI = 1;
   for ( int i = 2; i <= n; i++ ) {
      long nextFib = fibOfI + fibOfIMinus1;
      fibOfIMinus1 = fibOfI;
      fibOfI = nextFib;
   }
   return fibOfI;
}
```

Für n >= 0:
fib(n) = n <= 1 ? 1 : fib(n-1) + fib(n-2)

- Invariante: Nach h ≥ 0 Schritten ist
  - fibOfI == Fibonacci-Wert von h+1 und
  - fibOfIMinus1 == Fibonacci-Wert von h.
- Variante: h steigt um 1.

→Nach Schleifenende ist fibOfI == Fibonacci-Wert von n.

Und sobald das Abbruchkriterium der Schleife erreicht ist, folgt aus der Invariante unmittelbar Korrektheit des Ergebnisses. Die Schleife endet nach h == n minus 1 Durchläufen, in diesem Moment ist h plus 1 gleich n, passt also.

Beispiel Nullstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double findZero ( DoubleToDoubleFunction f,
                  double a, double b, double epsilon ) {
   double left = a;
   double right = b;
   while ( right – left > epsilon ) {
      double m = ( right – left ) / 2;
      if ( f.apply(left) * f.apply(m) > 0 )
         left = m;
      else
         right = m;
   }
   return ( right – left ) / 2;
}
```

- Invariante: mindestens eine Nullstelle im Intervall [left...right] existiert
- Variante: die Intervallbreite right – left wird halbiert

Als nächstes nicht noch ein weiteres Beispiel mit einer einfachen Zählschleife, sondern jetzt mit einer while-Schleife. Das Beispiel hatten wir schon weiter vorne in diesem Kapitel, aber in Racket und rekursiv. Die Funktion hieß find-zero. Jetzt betrachten wir eine Realisierung in Java mit einer Schleife.

Beispiel Nullstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double findZero ( DoubleToDoubleFunction f,
                  double a, double b, double epsilon ) {
   double left = a;
   double right = b;
   while ( right – left > epsilon ) {
      double m = ( right – left ) / 2;
      if ( f.apply(left) * f.apply(m) > 0 )
         left = m;
      else
         right = m;
   }
   return ( right – left ) / 2;
}
```

- Invariante: mindestens eine Nullstelle im Intervall [left...right] existiert
- Variante: die Intervallbreite right – left wird halbiert

Die Methode in Java ist analog zu der vorhin in Racket: Sie bekommt eine Funktion f und ein Intervall von a nach b, und sofern f die Vorbedingung in diesem Intervall erfüllt, liefert findZero einen Wert sehr nahe einer Nullstelle von f im Intervall von a nach b zurück.

Beispiel Nullstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double findZero ( DoubleToDoubleFunction f,
                  double a, double b, double epsilon ) {
   double left = a;
   double right = b;
   while ( right – left > epsilon ) {
      double m = ( right – left ) / 2;
      if ( f.apply(left) * f.apply(m) > 0 )
         left = m;
      else
         right = m;
   }
   return ( right – left ) / 2;
}
```

- Invariante: mindestens eine Nullstelle im Intervall [left...right] existiert
- Variante: die Intervallbreite right – left wird halbiert

***Erinnerung*: Dieses Interface hatten wir in Kapitel 09, Abschnitt zu Parallelisierung, selbst definiert in Anlehnung an die entsprechenden Interfaces in Package java.util.function.**

Beispiel Nullstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double findZero ( DoubleToDoubleFunction f,
                  double a, double b, double epsilon ) {
   double left = a;
   double right = b;
   while ( right – left > epsilon ) {
      double m = ( right – left ) / 2;
      if ( f.apply(left) * f.apply(m) > 0 )
         left = m;
      else
         right = m;
   }
   return ( right – left ) / 2;
}
```

- Invariante: mindestens eine Nullstelle im Intervall [left...right] existiert
- Variante: die Intervallbreite right – left wird halbiert

In Anlehnung an die entsprechenden Interfaces in Package java.util.function heißt die funktionale Methode auch hier apply. Damit ist diese Abfrage identisch mit der entsprechenden Alternative im cond-Ausdruck in der Racket-Funktion weiter vorne in diesem Kapitel.

Beispiel Nullstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double findZero ( DoubleToDoubleFunction f,
                  double a, double b, double epsilon ) {
   double left = a;
   double right = b;
   while ( right – left > epsilon ) {
      double m = ( right – left ) / 2;
      if ( f.apply(left) * f.apply(m) > 0 )
         left = m;
      else
         right = m;
   }
   return ( right – left ) / 2;
}
```

- Invariante: mindestens eine Nullstelle im Intervall [left...right] existiert
- Variante: die Intervallbreite right – left wird halbiert

Diese Java-Implementation von findZero ist, wie gesagt, nicht rekursiv wie die Racket-Implementation, sondern durch eine Schleife realisiert. In jedem Schleifendurchlauf wird einer der beiden Begrenzer des Intervalls enger gesetzt für den nächsten Schleifendurchlauf.

Die Schleifenvariante ist, dass die Differenz right minus left in jedem Durchlauf halbiert wird.

Beispiel Nullstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

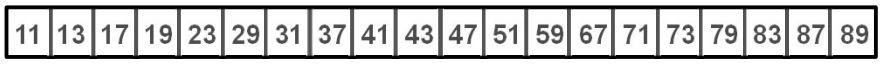
```
double findZero ( DoubleToDoubleFunction f,
                  double a, double b, double epsilon ) {
   double left = a;
   double right = b;
   while ( right – left > epsilon ) {
      double m = ( right – left ) / 2;
      if ( f.apply(left) * f.apply(m) > 0 )
         left = m;
      else
         right = m;
   }
   return ( right – left ) / 2;
}
```

- Invariante: mindestens eine Nullstelle im Intervall [left...right] existiert
- Variante: die Intervallbreite right – left wird halbiert

Auch der letztendliche Rückgabewert nach Erreichen der Abbruchbedingung ist eins-zu-eins wie in der rekursiven Racket-Funktion.

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

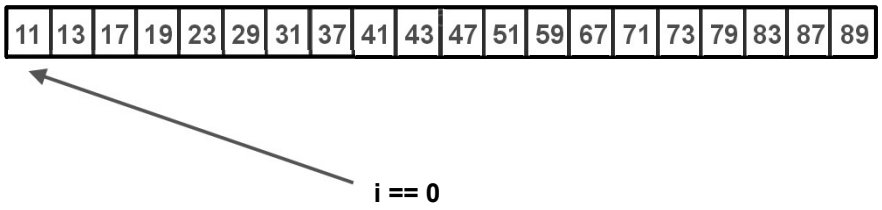


Gesucht: 87

Als nächstes Beispiel nehmen wir uns eine Grundoperation vor: Suchen eines Wertes in einem aufsteigend sortierten Array. In diesem Zahlenbeispiel sehen wir ein aufsteigend sortiertes Array von ganzen Zahlen, und gesucht ist ein Wert, der zufälligerweise sehr weit hinten im Array zu finden ist.

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

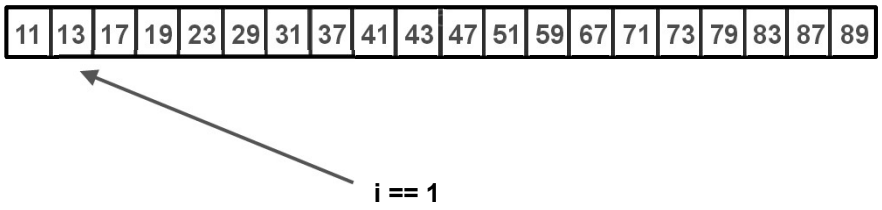


Gesucht: 87

Als erstes verwenden wir lineare Suche. Diese Suchstrategie ist extrem einfach: Wir gehen Index für Index durch das Array und schauen an jedem Index, ob der gesuchte Wert dort zu finden ist. Als erstes am Index 0, ...

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

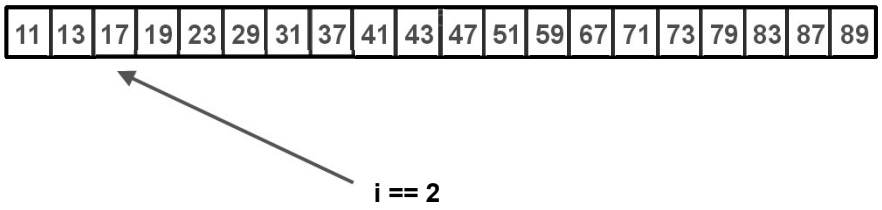


Gesucht: 87

... dann am Index 1, ...

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

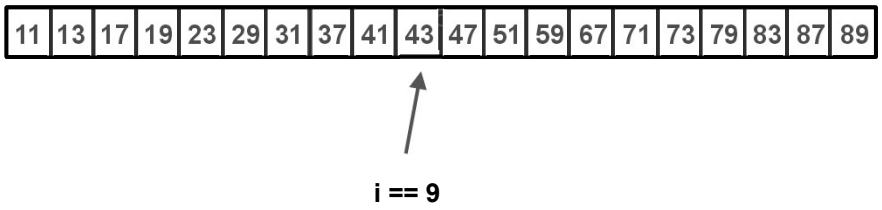


Gesucht: 87

... am Index 2, ...

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

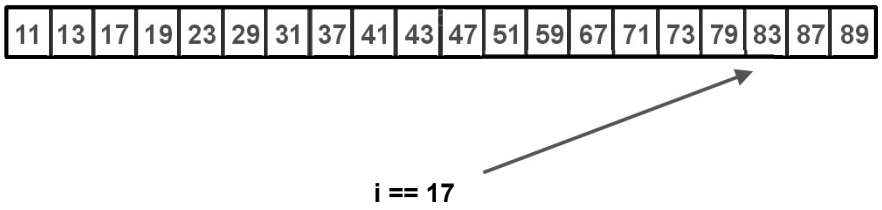


Gesucht: 87

... irgendwann später am Index 9, da der gesuchte Wert immer noch nicht gesehen wurde, ...

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

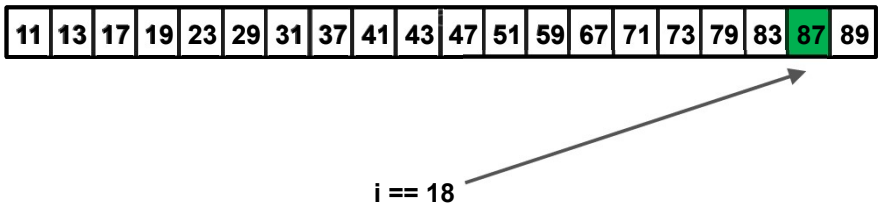


Gesucht: 87

... und dann kommen wir schon langsam in die Größenordnung des gesuchten Wertes ...

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

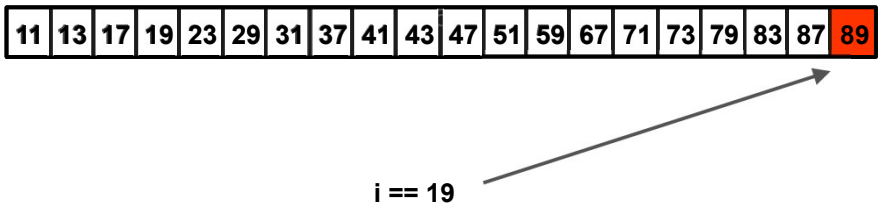


Gesucht: 87

... und haben ihn schlussendlich gefunden.

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT



Eine kleine Variation des Zahlenbeispiels: Nun suchen wir einen Wert, der *nicht* im Array enthalten ist, die 88. Durch die aufsteigende Sortierung kann die Suche sofort stoppen, sobald ein Wert gefunden wurde, der größer als der gesuchte ist. In diesem Zahlenbeispiel ist das dummerweise erst der allerletzte Wert im Array.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean contains ( int [ ] a, int n ) {
   for ( int i = 0; i < a.length; i++ ) {
      if ( a[i] == n )
         return true;
      if ( a[i] > n )
         return false;
   }
   return false;
}
```

So einfach, wie sie aussieht, ist die lineare Suche auch zu implementieren.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean contains ( int [ ] a, int n ) {
    for ( int i = 0; i < a.length; i++ ) {
        if ( a[i] == n )
            return true;
        if ( a[i] > n )
            return false;
    }
    return false;
}
```

Der Einfachheit halber implementieren wir hier nicht eine Klasse, die einen sortieren Array einkapselt, sondern einfach eine statische Methode, die das Array und den gesuchten Wert als zwei Parameter erhält. Ebenfalls zur Vereinfachung ist diese Methode nicht generisch, sondern nur speziell für den primitiven Datentyp int definiert.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean contains ( int [ ] a, int n ) {
  for ( int i = 0; i < a.length; i++ ) {
     if ( a[i] == n )
        return true;
     if ( a[i] > n )
       return false;
  }
  return false;
}
```

Hier wird explizit deutlich, dass wir im schlimmsten Fall alle Komponenten des Arrays durchlaufen müssen.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean contains ( int [ ] a, int n ) {
   for ( int i = 0; i < a.length; i++ ) {
      if ( a[i] == n )
         return true;
      if ( a[i] > n )
         return false;
   }
   return false;
}
```

Aber falls der gesuchte Wert schon vorher gefunden wurde, kann die Schleife auch schon vorher beendet werden.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean contains ( int [ ] a, int n ) {
   for ( int i = 0; i < a.length; i++ ) {
      if ( a[i] == n )
         return true;
      if ( a[i] > n )
         return false;
   }
   return false;
}
```

Da die Komponenten laut Vertrag aufsteigend sortiert sind, kann die Schleife ebenfalls vorher abgebrochen werden, falls ein Wert gesehen wird, der größer als der gesuchte Wert ist.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean contains ( int [ ] a, int n ) {
   for ( int i = 0; i < a.length; i++ ) {
      if ( a[i] == n )
         return true;
      if ( a[i] > n )
         return false;
   }
   return false;
}
```

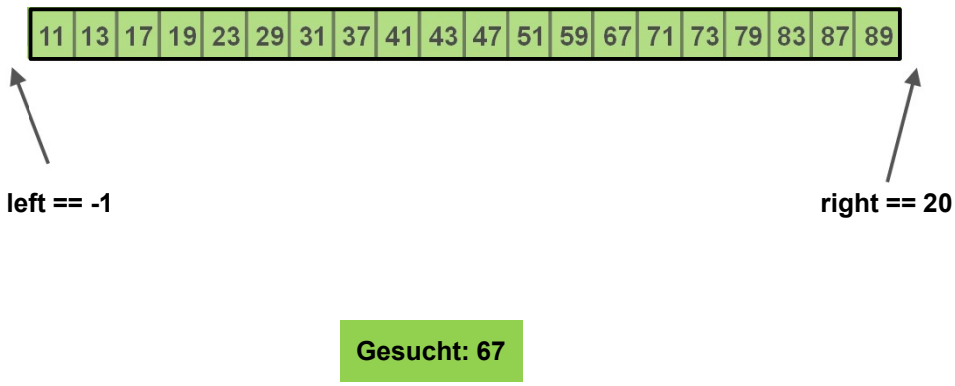
Invariante: Nach h ≥ 0 Iterationen gilt:
Falls n in a, dann ist n einer der Werte a[h-1] ... a[a.length-1].

Die Invariante ist, dass der gesuchte Wert n entweder gar nicht im Array vorhanden ist *oder* dass n im allerletzten Schleifendurchlauf gesehen – und daraufhin true zurückgeliefert – wurde *oder* dass n in den noch nicht gesehenen Arraykomponenten zu finden ist.

Sollte die Schleife bis um Ende durchlaufen, ohne dass n gesehen wird, folgt aus der Invariante zwingend, dass n nicht in a enthalten sein kann.

Binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

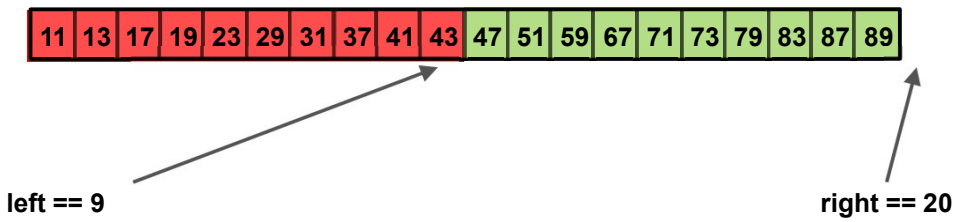


Jetzt zum Vergleich die binäre Suche. Diese Suchstrategie verwaltet zwei Indizes, die auch außerhalb des Bereichs der Arrayindizes sein dürfen – und dies zu Beginn auch sind.

Die Schleifeninvariante bei dieser konkreten Implementation der binären Suche wird sein: Falls der gesuchte Wert überhaupt im Array enthalten ist, dann ist er in dem Indexbereich des Arrays, der von left und right eingegrenzt wird, also der Indexbereich von left plus eins bis right minus eins.

Binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

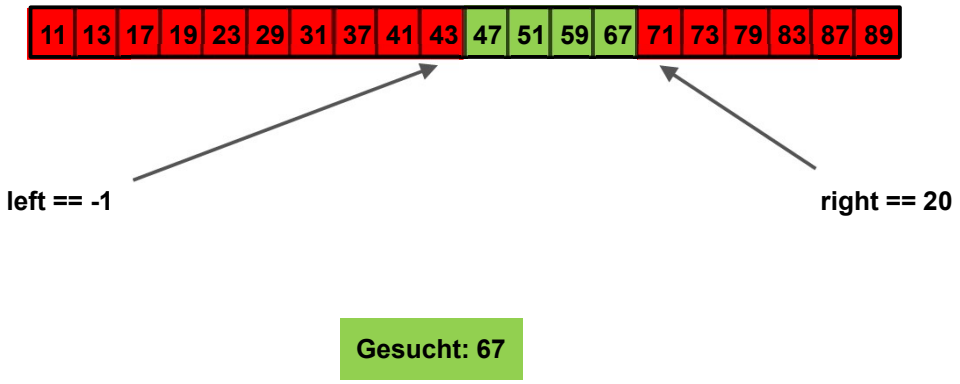


Gesucht: 67

Und die Schleifen*variante* ist, dass die Größe des eingegrenzten Bereich in jedem Schleifendurchlauf mindestens halbiert wird. Das ist so ähnlich wie bei der Intervallhalbierung bei reellwertigen Funktionen in Methode findZero soeben: Je nachdem, ob der Wert in der Mitte des eingegrenzten Bereichs größer oder kleiner als der gesuchte Wert ist, wird entweder die hintere oder die vordere Hälfte des momentanen Indexbereichs herausgenommen durch Setzung von right beziehungsweise left. Hier wurde left auf die Mitte des Indexbereichs gesetzt.

Binäre Suche

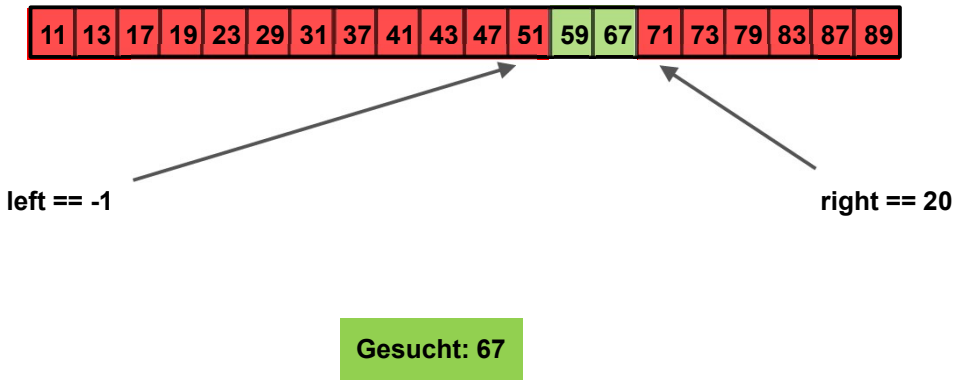
TECHNISCHE UNIVERSITÄT DARMSTADT



Als nächstes wäre dann right in die Mitte des verbliebenen Intervalls umzusetzen, ...

Binäre Suche

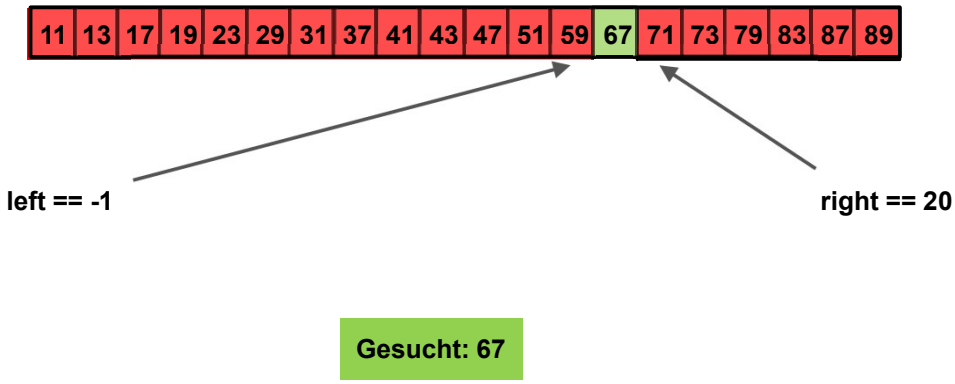
TECHNISCHE UNIVERSITÄT DARMSTADT



... jetzt noch einmal left ...

Binäre Suche

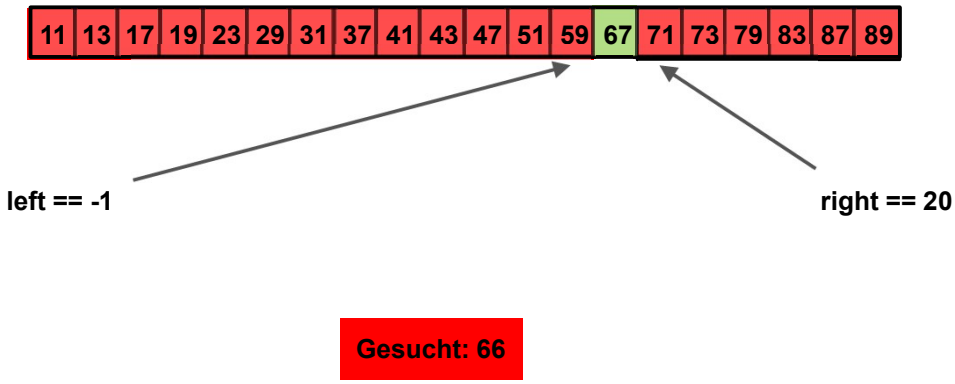
TECHNISCHE UNIVERSITÄT DARMSTADT



... und die gesuchte Zahl ist eingegrenzt.

Binäre Suche

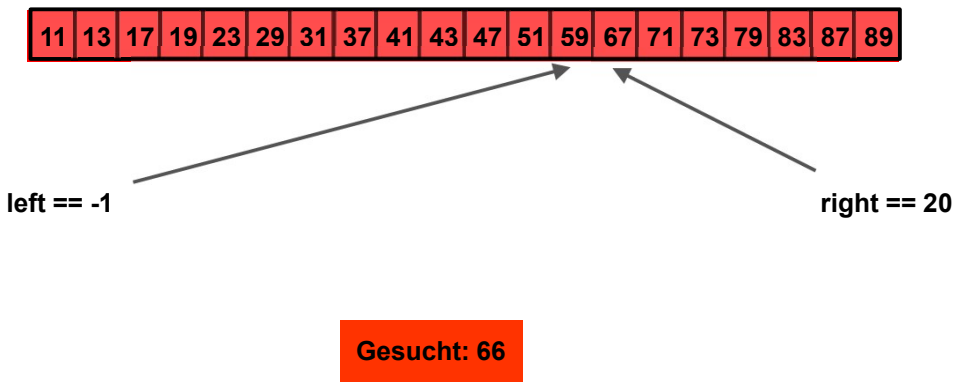
TECHNISCHE UNIVERSITÄT DARMSTADT



Wie bei linearer Suche eine kleine Variation des Zahlenbeispiel: Nun soll der um 1 kleinere Wert gefunden werden, der ist aber nicht drin.

Binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT



Das Ergebnis ist, dass der eingegrenzte Indexbereich leer ist. Die Invariante sagt ja, dass der gesuchte Wert im eingegrenzten Indexbereich ist, falls er überhaupt im Array ist. Daraus folgt logisch zwingend: Ist der Indexbereich leer, dann ist der gesuchte Wert *nicht* im Array.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
   int left = –1;
   int right = a.length;
   while ( right – left > 1 ) {
      int m = ( right – left ) / 2;
      if ( a[m] == n )
         return true;
      if ( a[m] < n )
         left = m;
      else
         right = m;
   }
   return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Jetzt die Implementation der binären Suche als Schleife.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
    int left = –1;
    int right = a.length;
    while ( right – left > 1 ) {
        int m = ( right – left ) / 2;
        if ( a[m] == n )
            return true;
        if ( a[m] < n )
            left = m;
        else
            right = m;
    }
    return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Derselbe Methodenkopf wie bei linearer Suche.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
  int left = –1;
  int right = a.length;
  while ( right – left > 1 ) {
    int m = ( right – left ) / 2;
    if ( a[m] == n )
      return true;
    if ( a[m] < n )
      left = m;
    else
      right = m;
  }
  return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Analog zur Intervallhalbierung definieren wir einen linken und einen rechten Begrenzer. Das sind jetzt ganze Zahlen.

Die Schleifeninvariante war ja: Falls n in a enthalten ist, dann findet sich n im Indexbereich left plus eins bis right minus eins.

Um diese Schleifeninvariante nach 0 Durchläufen, also unmittelbar vor dem ersten Durchlauf, zu erfüllen, werden die beiden Variablen so initialisiert, dass der Indexbereich left plus eins bis right minus eins exakt gleich dem gesamten Indexbereich des Arrays ist.

Beachten Sie, wie wichtig das explizite Hinschreiben der Schleifeninvariante ist: Die beiden Begrenzer left und right könnten ja auch so definiert sein, dass der fragliche Indexbereich left bis right oder left bis right minus eins oder left plus eins bis right ist. Verwechslungen hier sind eine häufige Fehlerquelle in Subroutinen, die mit Indexbereichen herumhantieren. Die Invariante stellt klar, was genau der fragliche Indexbereich in Abhängigkeit von left und right ist, und wenn man sich in allen Anweisungen in der Subroutine konsequent daran hält, ist diese Fehlerquelle eliminiert.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
   int left = –1;
   int right = a.length;
   while ( right – left > 1 ) {
      int m = ( right – left ) / 2;
      if ( a[m] == n )
         return true;
      if ( a[m] < n )
         left = m;
      else
         right = m;
   }
   return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Die Fortsetzungsbedingung ist gleichbedeutend damit, dass der Indexbereich left plus eins bis right minus eins nicht leer ist. Nach Schleifeninvariante heißt das also:

Wenn die Schleife abbricht, können wir uns sicher sein, dass n *nirgendwo* in a zu finden ist, denn wäre n *irgendwo* in a, dann müsste n auch mindestens einmal im Bereich left plus eins bis right minus eins zu finden sein. In einem leeren Indexbereich ist aber nun einmal kein n zu finden.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
   int left = –1;
   int right = a.length;
   while ( right – left > 1 ) {
      int m = ( right – left ) / 2;
      if ( a[m] == n )
         return true;
      if ( a[m] < n )
         left = m;
      else
         right = m;
   }
   return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Nach Schleifeninvariante ist es also korrekt, false zurückzuliefern, falls die Schleife regulär durch Abfrage der Fortsetzungsbedingung abbricht.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
   int left = –1;
   int right = a.length;
   while ( right – left > 1 ) {
      int m = ( right – left ) / 2;
      if ( a[m] == n )
         return true;
      if ( a[m] < n )
         left = m;
      else
         right = m;
   }
   return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Hier findet die eigentliche binäre Suche statt. Offensichtlich ist sie sehr ähnlich zur Intervallhalbierung, nur auf ganzen statt auf gebrochenen Zahlen. Beachten Sie, dass daher auch die Division ganzzahlig ist.

Die Schleifenvariante ist, wie gesagt, dass die Differenz right minus left in jedem Durchlauf mindestens halbiert wird.

Beispiel binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( int[ ] a, int n ) {
   int left = –1;
   int right = a.length;
   while ( right – left > 1 ) {
      int m = ( right – left ) / 2;
      if ( a[m] == n )
         return true;
      if ( a[m] < n )
         left = m;
      else
         right = m;
   }
   return false;
}
```

Invariante: Falls n in a, dann ist n einer der Werte a[left+1] ... a[right-1].

Ist das Element gefunden, wird sofort die dann korrekte Antwort true zurückgeliefert. Wir hatten schon geklärt, dass in dem Moment, wenn die Fortsetzungsbedingung nicht mehr erfüllt ist, die Rückgabe false statt dessen korrekt ist.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

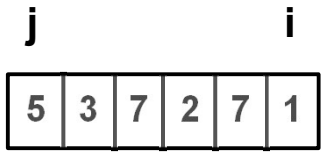
```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length-1; i > 0; i – – ) {
      for ( int j = 0; j < i; j++ )
         if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
            T tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
   }
}
```

Zum Abschluss des Abschnitts über Korrektheit von Schleifen noch zwei Beispiele, Selection Sort und Bubblesort, die zwei wichtige Aspekte beleuchten sollen: erstens Korrektheit von ineinander geschachtelten Schleifen. Zweitens sind die beiden Beispiele sich so ähnlich, dass man daran gut studieren kann, wie die Unterschiede in der Implementation und die Unterschiede bei der Schleifeninvariante miteinander korrespondieren.

Zuerst Bubblesort. Wir schauen uns auf den nächsten Folien erst einmal ein Beispiel an und kommen dann auf den hier schon einmal kurz gezeigten Code zurück.

Beispiel Bubblesort

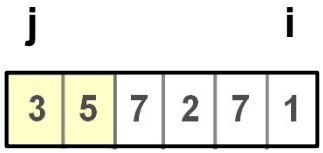
TECHNISCHE UNIVERSITÄT DARMSTADT



Der Algorithmus Bubblesort besteht im Wesentlichen aus zwei ineinandergeschachtelten Schleifen. In der äußeren Schleife wird i heruntergezählt, in der inneren Schleife wird j hochgezählt, und zwar bis i minus 1 einschließlich.

Beispiel Bubblesort

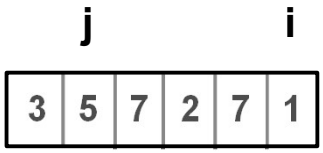
TECHNISCHE UNIVERSITÄT DARMSTADT



Bei Bubblesort wird immer der Wert an Index j mit dem Wert am Index j plus 1 vertauscht, falls der Wert an Index j größer als der an Index j plus 1 ist. Das ist hier gleich der Fall, also 3 und 5 vertauscht.

Beispiel Bubblesort

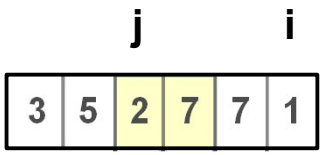
TECHNISCHE UNIVERSITÄT DARMSTADT



Bei 5 und 7 passiert nichts, da 5 kleiner-gleich 7 ist.

Beispiel Bubblesort

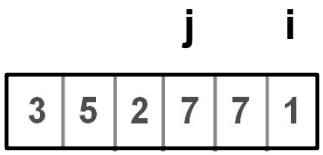
TECHNISCHE UNIVERSITÄT DARMSTADT



Jetzt wird wieder vertauscht Wir sehen: Der Wert an Index j plus 1 ist immer der größte bisher gesehene Wert.

Beispiel Bubblesort

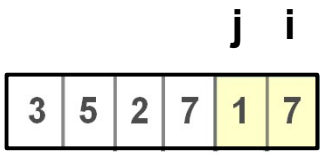
TECHNISCHE UNIVERSITÄT DARMSTADT



Bei Gleichheit wird *nicht* getauscht. Dadurch bleibt die Reihenfolge gleicher Werte erhalten, unsere Implementation des Algorithmus Bubblesort ist stabil.

Beispiel Bubblesort

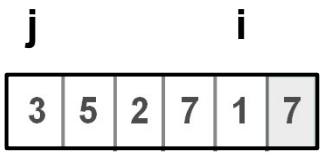
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier wird noch einmal getauscht, und damit ist der erste Durchlauf durch die äußere Schleife zu Ende. Die Invariante der äußeren Schleife ist: Die Menge von Werten im Array bleibt immer dieselbe, nur umgeordnet, und nach $h \geq 0$ Durchläufen durch die äußere Schleife sind die letzten h Indizes im Array korrekt besetzt. Und für die Stabilität muss die Invariante noch eine dritte Aussage enthalten: Die relative Reihenfolge gleich großer Elemente bleibt gleich.

Beispiel Bubblesort

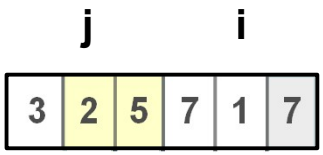
TECHNISCHE UNIVERSITÄT DARMSTADT



Es geht wieder von vorne los. Hier ist nichts zu tauschen, ...

Beispiel Bubblesort

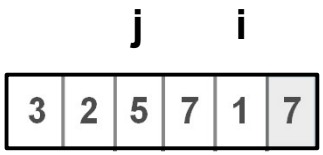
TECHNISCHE UNIVERSITÄT DARMSTADT



... hier muss jetzt wieder getauscht werden, ...

Beispiel Bubblesort

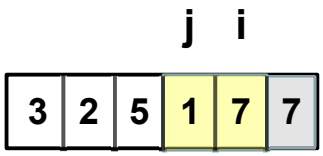
TECHNISCHE UNIVERSITÄT DARMSTADT



... hier nicht, ...

Beispiel Bubblesort

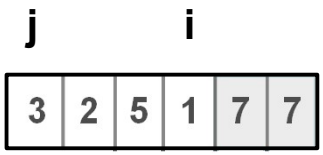
TECHNISCHE UNIVERSITÄT DARMSTADT



... aber dafür hier wieder.

Beispiel Bubblesort

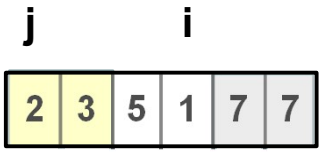
TECHNISCHE UNIVERSITÄT DARMSTADT



Nun der dritte Durchlauf, der also die 5 an die drittletzte Position bringen soll.

Beispiel Bubblesort

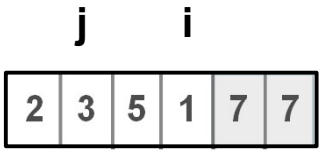
TECHNISCHE UNIVERSITÄT DARMSTADT



Gleich ein erster Tausch, ...

Beispiel Bubblesort

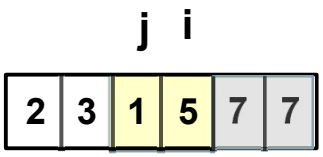
TECHNISCHE UNIVERSITÄT DARMSTADT



... hier jetzt kein Tausch nötig, ...

Beispiel Bubblesort

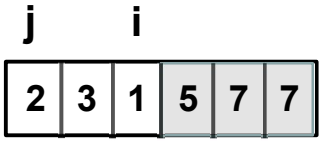
TECHNISCHE UNIVERSITÄT DARMSTADT



... und ein letzter Tausch bringt die 5 in die korrekte Position.

Beispiel Bubblesort

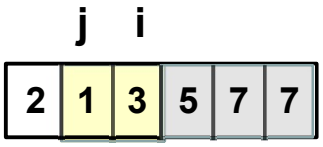
TECHNISCHE UNIVERSITÄT DARMSTADT



Nun der vierte Durchlauf, erst einmal kein Tausch.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT



Mit diesem Tausch ist der vierte Durchlauf auch schon wieder beendet.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

Der fünfte Durchlauf wird jetzt ganz kurz, ...

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

... ein Tausch und schon fertig. Offenbar wird kein sechster Durchlauf benötigt, denn wenn alle Werte bis auf einen an ihren korrekten Positionen sind, dann ist natürlich auch dieser eine Wert an seiner korrekten Position.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length-1; i > 0; i – – ) {
      for ( int j = 0; j < i; j++ )
         if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
            T tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
   }
}
```

Jetzt gehen wir an den Quelltext. Neben dem zu sortierenden Array hat die Methode einen Comparator für den Vergleich.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
  for ( int i = a.length-1; i > 0; i – – ) {
     for ( int j = 0; j < i; j++ )
        if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
           T tmp = a[j];
           a[j] = a[j+1];
           a[j+1] = tmp;
        }
  }
}
```

Schleifeninvariante: Nach h >= 0 Iterationen gilt:
1.die Gesamtmenge der Arraykomponenten ist gleichgeblieben;
2.die relative Reihenfolge gleicher Elemente ist gleichgeblieben;
3.an den letzten h Positionen stehen die korrekten Werte.

Wir haben im Zahlenbeispiel für Bubblesort soeben gesehen: i wandert vom letzten zum zweiten Index im Array. Unten sehen Sie die Invariante. Sie besteht aus drei Aussagen, wobei die Idee der Schleife in der dritten Aussage formuliert ist. Alle drei Aussagen der Invariante sind für h = 0 trivialerweise erfüllt, so dass keine Initialisierung vor der Schleife notwendig ist.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length-1; i > 0; i – – ) {
      for ( int j = 0; j < i; j++ )
         if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
            T tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
   }
}
```

Schleifeninvariante: Nach h >= 0 Iterationen gilt:
1.die Gesamtmenge der Arraykomponenten ist gleichgeblieben;
2.die relative Reihenfolge gleicher Elemente ist gleichgeblieben;
3.an Index h steht das Maximum aller Werte a[0]...a[h].

Jetzt sehen Sie unten die Invariante der inneren Schleife. Nur die dritte Aussage ist anders als bei der Invariante der äußeren Schleife. Für h = 0 sind alle drei Aussagen der Invariante trivialerweise erfüllt, daher ist bei Bubblesort keine Initialisierung für die innere Schleife notwendig. (Wir werden gleich sehen, dass das bei Selection Sort anders ist.)

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length-1; i > 0; i – – ) {
      for ( int j = 0; j < i; j++ )
         if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
            T tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
   }
}
```

Dank der Schleifeninvariante können wir davon ausgehen, dass unmittelbar vor dem h-ten Schleifendurchlauf an Stelle h minus eins das Maximum aus dem Bereich 0 bis h minus eins steht. Sollte der Wert an Index h größer oder gleich sein, ist nichts zu tun, dieser Wert ist dann das Maximum aus dem Bereich 0 bis h. Ist der Wert an Index h hingegen *nicht* größer, dann ist das Maximum aus dem Bereich 0 bis h-1 zugleich auch das Maximum aus dem Bereich 0 bis h und muss daher an die Position h getauscht werden.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length-1; i > 0; i – – ) {
      for ( int j = 0; j < i; j++ )
         if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
            T tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
   }
}
```

Vertauschung der Werte zweier Variable ist ein gängiges Programmiermuster mit drei Anweisungen: In der ersten Anweisung wird der Wert der einen der beiden Variablen in einer temporären Hilfsvariable gespeichert; dann kann man ihren Wert in der zweiten Anweisung ohne Verlust mit dem Wert der zweiten Variable überschreiben; schlussendlich kann man den Wert der zweiten Variable ohne Verlust mit dem ursprünglichen Wert der ersten Variable überschreiben, der ja in der ersten Anweisung in der temporären Variable gerettet wurde und nun dort zur Verfügung steht.

Beispiel Bubblesort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void bubbleSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length-1; i > 0; i – – ) {
      for ( int j = 0; j < i; j++ )
         if ( cmp.compare ( a[j], a[j+1] ) > 0 ) {
            T tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
   }
}
```

Die Stabilität des Algorithmus ist durch die Wahl des Vergleichsoperators gewährleistet. Durch den strikten Vergleich werden zwei ununterscheidbare Werte nicht vertauscht, die Reihenfolge ununterscheidbarer Werte bleibt erhalten.

Beispiel Selection Sort

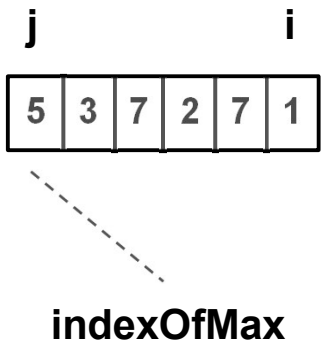
TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Jetzt zum zweiten angekündigten Sortieralgorithmus, Selection Sort.

Beispiel Selection Sort

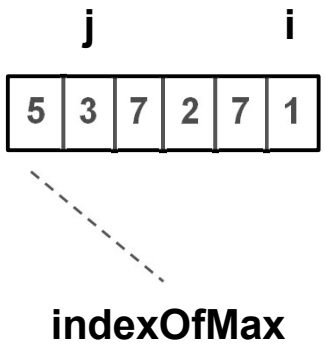
TECHNISCHE UNIVERSITÄT DARMSTADT



Unsere Implementation von Selection Sort verwaltet drei Laufindizes, die so wie hier gezeigt initialisiert werden: zwei auf dem ersten Index, einer auf dem letzten Index. Der Index i wird bei Selection Sort dieselbe Funktion haben wie bei Bubblesort und sich auch genau so verhalten.

Beispiel Selection Sort

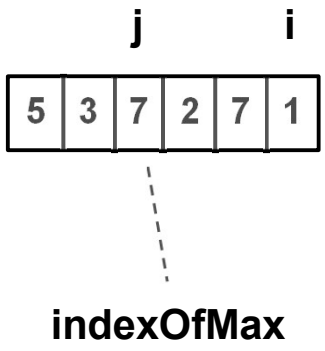
TECHNISCHE UNIVERSITÄT DARMSTADT



Der Index j läuft jetzt los. Wann immer der Wert, auf dem j steht, größer oder gleich dem Wert ist, auf den indexOfMax verweist, wird indexOfMax gleich j gesetzt. Das ist hier nicht der Fall, ...

Beispiel Selection Sort

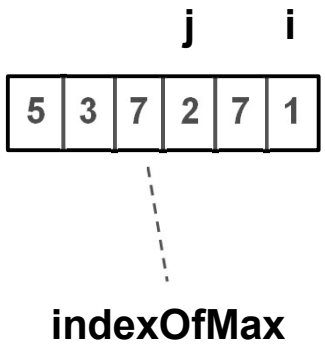
TECHNISCHE UNIVERSITÄT DARMSTADT



... aber *hier* ist das der Fall. Die 7 ist größer-gleich der 5, also wird indexOfMax auf j umgesetzt. Wir sehen: indexOfMax ist immer der Index eines größten Wertes unter allen von j durchlaufenen Werten. Das ist die vorläufige Formulierung der Invariante.

Beispiel Selection Sort

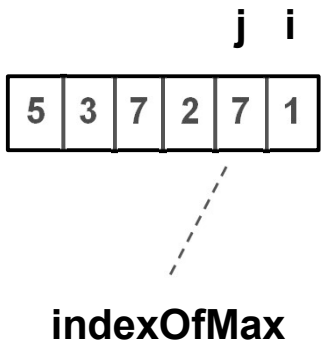
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier passiert wieder nichts.

Beispiel Selection Sort

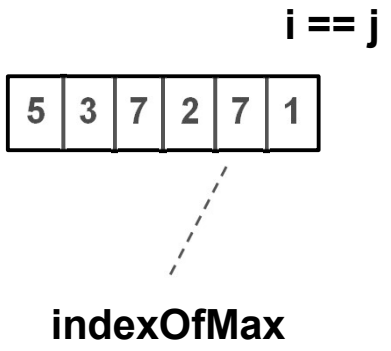
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier wird indexOfMax wieder umgesetzt. Wir sehen also: Wenn es mehrere größte Werte unter den bisher von j durchlaufenen gibt, dann ist indexOfMax immer der größte Index eines solchen Wertes. Das ist die endgültige Formulierung der Invariante der inneren Schleife. Ohne diese vermeintliche Feinheit wäre der Algorithmus nicht stabil.

Beispiel Selection Sort

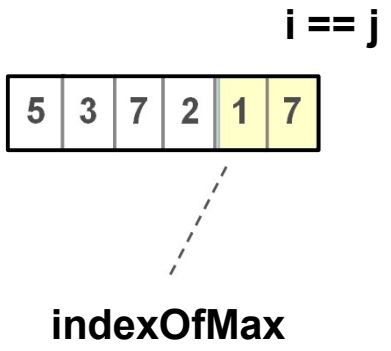
TECHNISCHE UNIVERSITÄT DARMSTADT



Ein Durchlauf der inneren Schleife ist zu Ende, wenn j gleich i ist. Da die 1 kleiner als die 7 ist, werden die letzten beiden Komponenten des Arrays nicht vertauscht.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

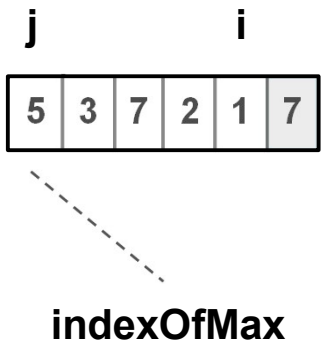


Jetzt werden die beiden Werte an i und an indexOfMax vertauscht. Wir sehen: Am Ende der Schleife steht am letzten Index der größte Wert. Das ist auch korrekt.

Warum ist es wichtig, dass die zweite 7 getauscht wird und nicht die erste? Nur so kann der Sortieralgorithmus stabil werden, denn die Reihenfolge der beiden Siebener bleibt erhalten.

Beispiel Selection Sort

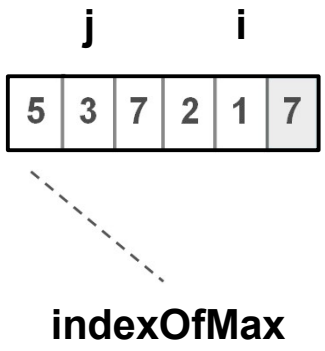
TECHNISCHE UNIVERSITÄT DARMSTADT



Wir wiederholen jetzt den Durchlauf mit j. Da der letzte Index korrekt besetzt ist, geht der zweite Durchlauf nur bis zum vorletzten Index. Daher ist i im zweiten Durchlauf der vorletzte Index, nicht mehr der letzte wie im ersten Durchlauf.

Beispiel Selection Sort

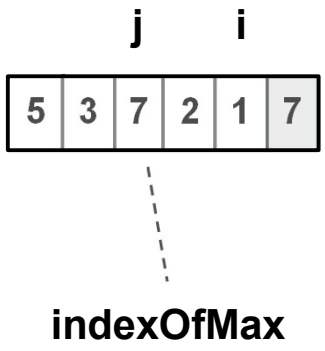
TECHNISCHE UNIVERSITÄT DARMSTADT



Ansonsten ist alles genau so wie im ersten Durchlauf: Falls wie hier der nächste Wert kleiner ist, passiert nichts mit indexOfMax, ...

Beispiel Selection Sort

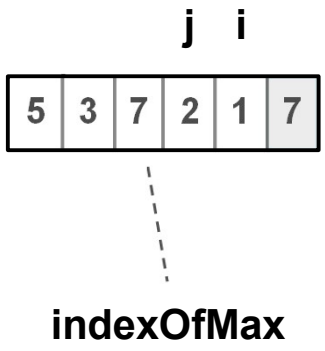
TECHNISCHE UNIVERSITÄT DARMSTADT



... und bei einem Wert größer-gleich wird indexOfMax wie gehabt umgesetzt.

Beispiel Selection Sort

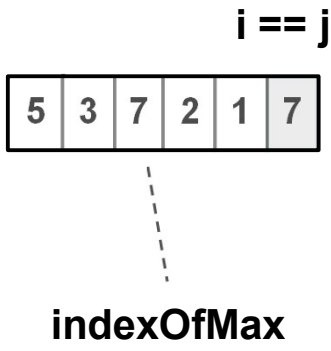
TECHNISCHE UNIVERSITÄT DARMSTADT



Wieder keine Umsetzung von indexOfMax ...

Beispiel Selection Sort

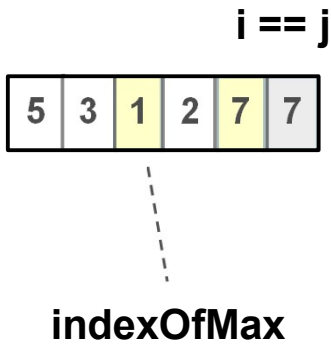
TECHNISCHE UNIVERSITÄT DARMSTADT



... und auch keine in dem Moment, in dem j den Index i erreicht.

Beispiel Selection Sort

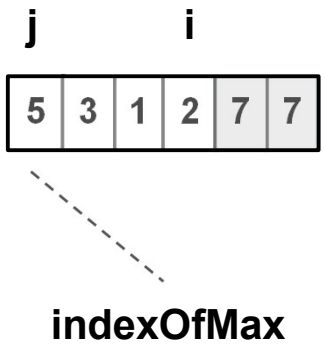
TECHNISCHE UNIVERSITÄT DARMSTADT



Auch nach der zweiten inneren Schleife werden die Werte am Index i und am Index indexOfMax vertauscht. Jetzt sind die beiden größten Werte an den beiden letzten Indizes, und das ist auch korrekt. Und diese beiden Werte sind in derselben Reihenfolge wie in der Eingabe, was ja für die Stabilität notwendig ist.

Beispiel Selection Sort

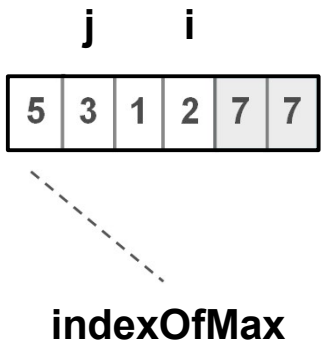
TECHNISCHE UNIVERSITÄT DARMSTADT



Jetzt beginnt der dritte Durchlauf durch die äußere Schleife. Wir sehen schon, dass am Ende die 5 an Index i stehen sollte.

Beispiel Selection Sort

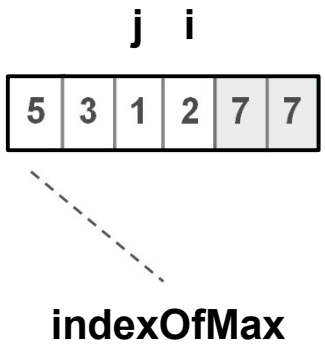
TECHNISCHE UNIVERSITÄT DARMSTADT



Der Index namens indexOfMax bleibt auch an der 5 kleben, ...

Beispiel Selection Sort

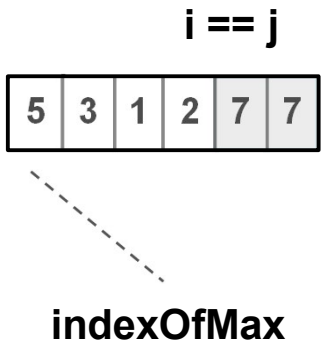
TECHNISCHE UNIVERSITÄT DARMSTADT



... und kleben...

Beispiel Selection Sort

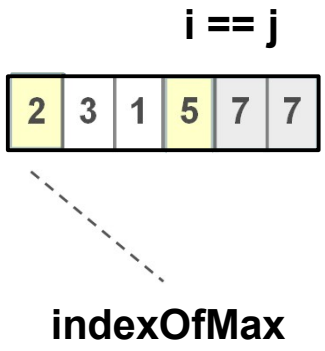
TECHNISCHE UNIVERSITÄT DARMSTADT



... und kleben.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

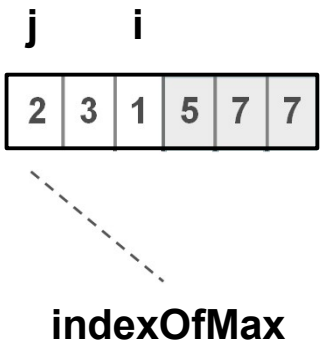


Daher wird die 5 wie gewünscht an den drittletzten Index getauscht.

Wir sehen jetzt auch schon, dass die Invariante der äußeren Schleife offensichtlich dieselbe ist wie bei Bubblesort: Nach h >= 0 Iterationen sind die korrekten Werte an den letzten h Positionen, wobei die Gesamtmenge an Werten und auch die relative Reihenfolge ununterscheidbarer Werte gleichgeblieben sind.

Beispiel Selection Sort

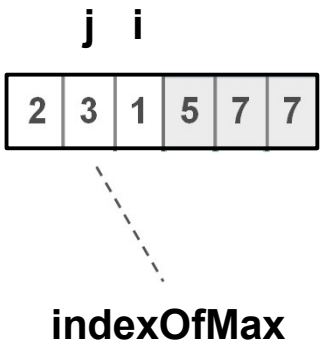
TECHNISCHE UNIVERSITÄT DARMSTADT



Nach dieser Vertauschung beginnt wieder alles von vorne, der vierte Durchlauf durch die äußere Schleife.

Beispiel Selection Sort

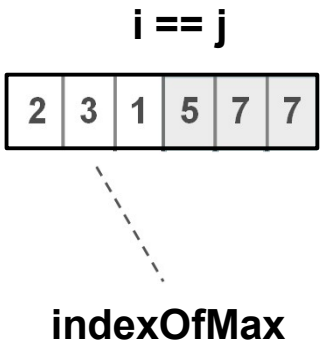
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier wird indexOfMax korrekterweise wieder auf j umgesetzt.

Beispiel Selection Sort

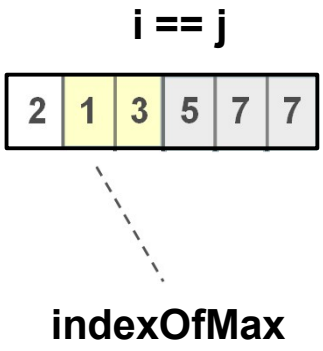
TECHNISCHE UNIVERSITÄT DARMSTADT



Und schon sind wir in diesem kleinen Beispiel bei i angelangt.

Beispiel Selection Sort

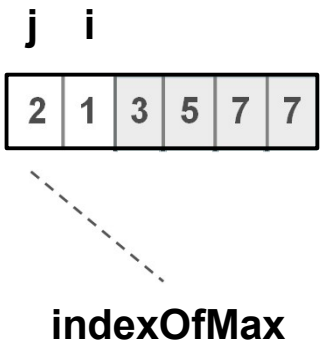
TECHNISCHE UNIVERSITÄT DARMSTADT



Wieder Austausch ...

Beispiel Selection Sort

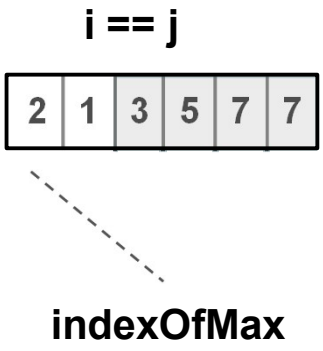
TECHNISCHE UNIVERSITÄT DARMSTADT



... und Beginn des letzten Durchlaufs.

Beispiel Selection Sort

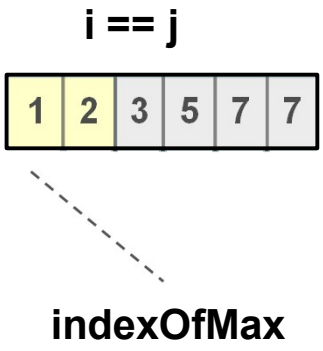
TECHNISCHE UNIVERSITÄT DARMSTADT



Der letzte Durchlauf ist ganz schnell zu Ende, ...

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT



... ein letzter Tausch ...

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Und mit dem Wert am zweiten Index ist natürlich zwangsläufig auch der Wert am ersten Index an der korrekten Position.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Jetzt haben wir ein kleines Beispiel durchexerziert und sollten gewappnet sein für den Quelltext.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Die äußere Schleife durchläuft dieselben Indizes in derselben Reihenfolge wie bei Bubblesort.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Die Suche nach dem maximalen Wert wird in eine Methode ausgelagert, die wir uns gleich ansehen werden.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Um Stabilität zu garantieren, reicht es im Falle von mehreren Vorkommen des maximalen Wertes nicht, *irgendeinen* Index eines Vorkommens des maximalen Wertes zurückzuliefern, sondern es muss der *höchste* solche Index sein. Auch das hatten wir schon im Beispiel gesehen.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Die Vertauschung der beiden Komponenten wird ebenfalls in eine eigene Methode ausgelagert.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Falls die Indizes der beiden auszutauschenden Komponenten identisch sind, kann man sich die Laufzeit für den Methodenaufruf natürlich sparen, denn das Ergebnis wäre ja dasselbe wie wenn man nichts macht. Kann natürlich sein, dass dieser Test vor jeder Vertauschung unterm Strich mehr Laufzeit kostet als gelegentliche überflüssige Vertauschungen oder dass man den Unterschied gar nicht merkt. Egal, wir müssen entscheiden, ob wir eine Komponente mit selbst tauschen oder nicht, und wir haben uns jetzt halt so entschieden.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = getHighestIndexOfMaximum ( a, 0, i, cmp );
      if ( indexOfMax != i ) {
         swapComponents ( a, indexOfMax, i );
         rotateDuplicatesLeft ( a, indexOfMax, i - 1 );
      }
   }
}
```

Diesen Punkt haben wir im Beispiel noch nicht gesehen. Wenn der Wert an Position i mit dem Wert an Position indexOfMax vertauscht wird, dann kann es auch bei dem Wert, der vor dieser Vertauschung an Index i stand, eine Verletzung der Stabilität geben. Das muss dann nach der Vertauschung repariert werden. Auch dies ist in eine eigene Methode ausgelagert, die wir uns gleich ansehen werden.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: 0 <= firstIndex <= lastIndex < a.length.
// Returns: the index in firstIndex ... lastIndex where the maximum
//          value out of a[firstIndex] ... a[lastIndex] is found; in case
//          of a tie, the highest index of such a maximum is returned.

public static <T> int getHighestIndexOfMaximum
                    ( T[ ] a, int firstIndex, int lastIndex, Comparator<T> cmp ) {
   int indexOfMax = firstIndex;
   for ( int i = firstIndex + 1; i <= lastIndex; i++ )
       if ( cmp.compare ( a[indexOfMax], a[i] ) <= 0 )
          indexOfMax = i;
   return indexOfMax;
}
```

Nun zu den drei in Methoden ausgelagerten Schritten, als erstes die Suche nach dem Maximum.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: 0 <= firstIndex <= lastIndex < a.length.
// Returns: the index in firstIndex ... lastIndex where the maximum
//          value out of a[firstIndex] ... a[lastIndex] is found; in case
//          of a tie, the highest index of such a maximum is returned.

public static <T> int getHighestIndexOfMaximum
                    ( T[ ] a, int firstIndex, int lastIndex, Comparator<T> cmp ) {
   int indexOfMax = firstIndex;
   for ( int i = firstIndex + 1; i <= lastIndex; i++ )
      if ( cmp.compare ( a[indexOfMax], a[i] ) <= 0 )
         indexOfMax = i;
   return indexOfMax;
}
```

Der Vertrag kodifiziert das, was wir schon informell gesagt hatten: Unter allen Vorkommen des maximalen Wertes in einem Indexbereich des Arrays soll dasjenige mit dem höchsten Index gefunden werden. Dass zuerst gesagt wird, worum es prinzipiell geht (den Maximalwert) und danach dann mit so etwas wie „in case of a tie“ genauer gesagt wird, welcher von mehreren gesucht ist (den am höchsten Index) ist eine bewährte Formulierungsweise für solche Fälle.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: 0 <= firstIndex <= lastIndex < a.length.
// Returns: the index in firstIndex ... lastIndex where the maximum
//          value out of a[firstIndex] ... a[lastIndex] is found; in case
//          of a tie, the highest index of such a maximum is returned.

public static <T> int getHighestIndexOfMaximum
                    ( T[ ] a, int firstIndex, int lastIndex, Comparator<T> cmp ) {
  int indexOfMax = firstIndex;
  for ( int i = firstIndex + 1; i <= lastIndex; i++ )
      if ( cmp.compare ( a[indexOfMax], a[i] ) <= 0 )
         indexOfMax = i;
  return indexOfMax;
}
```

Die Berechnung des Maximums aus einer Sequenz ist ein gängiges Programmiermuster. Die Invariante lautet: Nach h >= 0 Durchläufen durch die Schleife ist indexOfMax der höchste Index im Indexbereich von firstIndex bis firstIndex plus h , an dem das Maximum aus diesem Indexbereich zu finden ist. Für h gleich 0 folgt daraus zwingend, dass indexOfMax gleich firstIndex sein muss, das heißt, so muss indexOfMax vor der Schleife initialisiert werden.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: 0 <= firstIndex <= lastIndex < a.length.
// Returns: the index in firstIndex ... lastIndex where the maximum
//          value out of a[firstIndex] ... a[lastIndex] is found; in case
//          of a tie, the highest index of such a maximum is returned.

public static <T> int getHighestIndexOfMaximum
                    ( T[ ] a, int firstIndex, int lastIndex, Comparator<T> cmp ) {
   int indexOfMax = firstIndex;
   for ( int i = firstIndex + 1; i <= lastIndex; i++ )
      if ( cmp.compare ( a[indexOfMax], a[i] ) <= 0 )
         indexOfMax = i;
   return indexOfMax;
}
```

Dadurch, dass hier kleiner-gleich statt kleiner abgefragt wird, ist gewährleistet, dass indexOfMax nicht einfach nur auf *irgendein* Vorkommen des größten bisher gesehenen Wertes verweist, sondern der größte solche Index ist, der bisher gesehen wurde.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: index1 and index2 are in 0 ... a.length – 1.
// Postcondition: the components of array a at index1 and index2
//                are exchanged.

private static <T> void swapComponents ( T[ ] a, int index1, int index2 ) {
    T tmp = a [ index1 ];
    a [ index1 ] = a [ index2 ];
    a [ index2 ] = tmp;
}
```

Das ist die in eine eigene Methode ausgelagerte Vertauschung zweier Komponenten des Arrays. Hatten wir schon bei Bubblesort gesehen, braucht hier nicht noch einmal erläutert werden.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//      left-rotated, that is, their indices in a are permuted such that the first
//     occurence will be the last one, the second one will be the first one, the
//     third one will be second one, etc.

private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
      int indexOfToBeExchangedNext = indexOfFirstDuplicate;
      for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
          if ( a[indexOfToBeExchangedNext] == a[i] ) {
             swapComponents ( a, indexOfToBeExchangedNext, i );
             indexOfToBeExchangedNext = i;
     }
   }
}
```

Es bleibt noch die Methode, die sicherstellen soll, dass die relative Reihenfolge auch bei dem Wert gewahrt bleibt, der von der Position i an die Position indexOfMax getauscht wird. Es kann ja sein, dass weitere Vorkommen dieses Wertes an Indizes von indexOfMax plus 1 bis i minus 1 stehen. Die Idee hinter dieser Methode ist, dass das Vorkommen, das von i an indexOfMax getauscht wurde, schrittweise durch eventuelle Vorkommen desselben Wertes sozusagen hindurchgetauscht wird.

Der Index i aus der Methode selectionSort taucht hier nicht mehr explizit auf, aber der formale Parameter lastIndexToBeConsidered wird ja beim Aufruf in selectionSort mit dem aktualen Parameterwert i minus 1 initialisiert, also wie der Name schon sagt, der letzte zu betrachtende Index.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//     left-rotated, that is, their indices in a are permuted such that the first
//    occurence will be the last one, the second one will be the first one, the
//    third one will be second one, etc.
private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
      int indexOfToBeExchangedNext = indexOfFirstDuplicate;
      for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
          if ( a[indexOfToBeExchangedNext] == a[i] ) {
             swapComponents ( a, indexOfToBeExchangedNext, i );
             indexOfToBeExchangedNext = i;
     }
   }
}
```

Hier steht genauer, was mit Linksrotation gemeint ist. Das von Index i weggetauschte Element wird unter allen identischen Vorkommen wieder an den höchsten Index kleiner i getauscht, und zwar so, dass die ursprüngliche Reihenfolge aller Vorkommen dieses Wertes wieder erreicht ist.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//      left-rotated, that is, their indices in a are permuted such that the first
//     occurence will be the last one, the second one will be the first one, the
//     third one will be second one, etc.

private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
    int indexOfToBeExchangedNext = indexOfFirstDuplicate;
    for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
        if ( a[indexOfToBeExchangedNext] == a[i] ) {
            swapComponents ( a, indexOfToBeExchangedNext, i );
            indexOfToBeExchangedNext = i;
    }
  }
}
```

Es werden immer zwei aufeinanderfolgende Vorkommen desselben Wertes vertauscht. In der hier eingerichteten Variable steht immer der Index des ersten der beiden Indizes.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//      left-rotated, that is, their indices in a are permuted such that the first
//     occurence will be the last one, the second one will be the first one, the
//     third one will be second one, etc.

private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
    int indexOfToBeExchangedNext = indexOfFirstDuplicate;
    for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
        if ( a[indexOfToBeExchangedNext] == a[i] ) {
            swapComponents ( a, indexOfToBeExchangedNext, i );
            indexOfToBeExchangedNext = i;
    }
  }
}
```

Die Invariante der Schleife ist erstens, dass an indexOfToBeExchangedNext dasjenige Vorkommen des Wertes ist, das ganz ans Ende rotiert werden soll; zweitens stehen an kleineren Indizes als indexOfToBeExchangedNext alle Vorkommen des Wertes an ihren richtigen Positionen bezüglich stabiler Sortierung. Bei Initialisierung von indexOfToBeExchanged auf indexOfFirstDuplicate sind beide Teile trivialerweise erfüllt. Diese Schleife nun muss so implementiert werden, dass die Invariante bis zum Ende der Schleife erfüllt bleibt.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//      left-rotated, that is, their indices in a are permuted such that the first
//     occurence will be the last one, the second one will be the first one, the
//     third one will be second one, etc.

private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
    int indexOfToBeExchangedNext = indexOfFirstDuplicate;
    for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
        if ( a[indexOfToBeExchangedNext] == a[i] ) {
            swapComponents ( a, indexOfToBeExchangedNext, i );
            indexOfToBeExchangedNext = i;
    }
  }
}
```

Natürlich ist nur dann etwas zu tun, wenn die Schleife an einem Vorkommen des Wertes vorbeikommt.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//      left-rotated, that is, their indices in a are permuted such that the first
//     occurence will be the last one, the second one will be the first one, the
//     third one will be second one, etc.

private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
      int indexOfToBeExchangedNext = indexOfFirstDuplicate;
      for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
          if ( a[indexOfToBeExchangedNext] == a[i] ) {
            swapComponents ( a, indexOfToBeExchangedNext, i );
            indexOfToBeExchangedNext = i;
     }
   }
}
```

Und was dann zu tun ist, sollte nach dem bisher Gesagten klar sein: den Wert am aktuellen Index vertauschen mit dem letzten Vorkommen dieses Wertes.

Beispiel Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

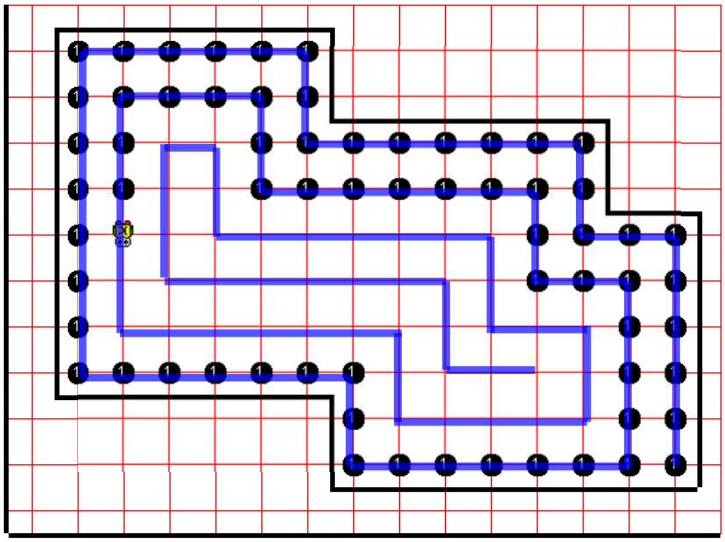
```
// Postcondition: the duplicates of a[indexOfToBeShifted] in array a within the
//      index range indexOfToBeShifted ... lastIndexToBeConsidered are
//      left-rotated, that is, their indices in a are permuted such that the first
//     occurence will be the last one, the second one will be the first one, the
//     third one will be second one, etc.

private static <T> void rotateDuplicatesLeft
                        ( T[ ] a, int indexOfFirstDuplicate, int lastIndexToBeConsidered ) {
      int indexOfToBeExchangedNext = indexOfFirstDuplicate;
      for ( int i = indexOfFirstDuplicate + 1; i <= lastIndexToBeConsidered; i++ )
          if ( a[indexOfToBeExchangedNext] == a[i] ) {
             swapComponents ( a, indexOfToBeExchangedNext, i );
             indexOfToBeExchangedNext = i;
     }
   }
}
```

Und dann natürlich den Index des letzten Vorkommens nachziehen.

Damit ist das Fallbeispiel Selection Sort beendet.

Beispiel KarelJ

TECHNISCHE UNIVERSITÄT DARMSTADT



Noch ein illustratives Beispiel, jetzt aus Kapitel 01c. Die Schleifeninvariante hatten wir dort schon formuliert:

Nach $h \geq 0$ Durchläufen sind Beeper auf den ersten h Kreuzungen der Spirale abgelegt; der Roboter steht auf der (h+1)-ten Kreuzung der Spirale und schaut entgegengesetzt zum letzten platzierten Beeper.

Aufgrund der Invariante kann die Implementation der Schleife davon ausgehen, dass die Kreuzungen auf der Spur rechts neben den bisher durchlaufenen Kreuzungen alle schon besetzt sind, so dass in jedem Schritt die in Laufrichtung am weitesten rechte freie Kreuzung die nächste Kreuzung auf der Spur ist. Und genau zu dieser Kreuzung geht der Roboter in der Implementation, die wir in Kapitel 03c gesehen hatten.

Korrektheit von Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Invariante* = *Induktionsbehauptung*: „Nach h ≥ 0 Schleifendurchläufen gilt ....."
- *Induktionsanfang*, also h = 0: Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Schleifendurchlauf erfüllt ist.
- *Induktionsvoraussetzung* für h > 0: Die Invariante gelte für h-1 (also unmittelbar vor dem h-ten Durchlauf).
- *Induktionsschritt*: Unter der Voraussetzung, dass die Invariante nach h-1 Durchläufen gilt, sorgt der Body der Schleife dafür, dass die Invariante auch nach h Durchläufen weiterhin gilt.

Zum Abschluss des Abschnitts über die Korrektheit von Schleifen machen wir dieselbe theoretische Einordnung wie bei rekursiven Subroutinen. Auch hinter der Argumentation, warum eine Schleife korrekt ist, steht immer eine vollständige Induktion.

Korrektheit von Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Invariante* = *Induktionsbehauptung*: „Nach h ≥ 0 Schleifendurchläufen gilt .....“**
- ***Induktionsanfang*, also h = 0: Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Schleifendurchlauf erfüllt ist.**
- ***Induktionsvoraussetzung* für h > 0: Die Invariante gelte für h-1 (also unmittelbar vor dem h-ten Durchlauf).**
- ***Induktionsschritt*: Unter der Voraussetzung, dass die Invariante nach h-1 Durchläufen gilt, sorgt der Body der Schleife dafür, dass die Invariante auch nach h Durchläufen weiterhin gilt.**

Die Schleifeninvariante hat dabei eine herausragende Bedeutung, das ist nämlich die per vollständiger Induktion zu beweisende Behauptung.

Korrektheit von Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Invariante* = *Induktionsbehauptung*: „Nach h ≥ 0 Schleifendurchläufen gilt .....“
- *Induktionsanfang*, also h = 0: Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Schleifendurchlauf erfüllt ist.
- *Induktionsvoraussetzung* für h > 0: Die Invariante gelte für h-1 (also unmittelbar vor dem h-ten Durchlauf).
- *Induktionsschritt*: Unter der Voraussetzung, dass die Invariante nach h-1 Durchläufen gilt, sorgt der Body der Schleife dafür, dass die Invariante auch nach h Durchläufen weiterhin gilt.

Bei jedem Beispiel hatten wir immer auch die Initialisierung vor der Schleife angesprochen. Im Begriffsrahmen der vollständigen Induktion ist das, was wir dazu gesagt hatten, schlicht und einfach der Induktionsanfang.

Korrektheit von Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Invariante* = *Induktionsbehauptung*: „Nach h ≥ 0 Schleifendurchläufen gilt ....."
- *Induktionsanfang*, also h = 0: Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Schleifendurchlauf erfüllt ist.
- *Induktionsvoraussetzung* für h > 0: Die Invariante gelte für h-1 (also unmittelbar vor dem h-ten Durchlauf).
- *Induktionsschritt*: Unter der Voraussetzung, dass die Invariante nach h-1 Durchläufen gilt, sorgt der Body der Schleife dafür, dass die Invariante auch nach h Durchläufen weiterhin gilt.

Die Induktionsvoraussetzung ist dann ebenso einfach die Annahme, dass die Invariante unmittelbar *vor* dem betrachteten Schleifendurchlauf erfüllt ist.

Korrektheit von Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Invariante* = *Induktionsbehauptung*: „Nach h ≥ 0 Schleifendurchläufen gilt .....“**
- ***Induktionsanfang*, also h = 0: Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Schleifendurchlauf erfüllt ist.**
- ***Induktionsvoraussetzung* für h > 0: Die Invariante gelte für h-1 (also unmittelbar vor dem h-ten Durchlauf).**
- ***Induktionsschritt*: Unter der Voraussetzung, dass die Invariante nach h-1 Durchläufen gilt, sorgt der Body der Schleife dafür, dass die Invariante auch nach h Durchläufen weiterhin gilt.**

Und der Induktionsschritt beinhaltet dann, dass die Invariante in einem Schleifendurchlauf erfüllt bleibt.

Damit beenden wir den Abschnitt zur Korrektheit von Schleifen und das ganze Kapitel.