

IMPERATIVE PROGRAMMIERUNG

Kurzbeschreibung

Ausgehend von den Variablen und Zuweisungen werden in dieser Kurseinheit Sprachelemente und Datenstrukturen der imperativen Programmierung eingeführt. Aspekte wie Zuweisungssemantik, Zusicherung, Schleifeninvarianten und Effizienz werden behandelt.

Schlüsselwörter

Variable, Verzweigung, Schleife, Zusicherung, Effizienz, Wertemenge, Array, mehrdimensionales Array, Zeichen, Zeichenkette, String, Stringoperation

Lernziele

1. Das Konzept der Variablen und Zuweisungen als Kern der imperativen Programmierung wird verstanden.
2. Die wichtigsten Anweisungen imperativer Programmierung sowie der Methodenaufruf können zur Erstellung eigener Programme genutzt werden.
3. Zusicherungen und Schleifeninvarianten können zu einem imperativen Programm formuliert werden.
4. Datenobjekte vom Typ Array bzw. String können innerhalb der imperativen Programmierung deklariert und verwendet werden.

Hauptquellen

- Manfred Broy: Informatik – Eine grundlegende Einführung, Band 1: Programmierung und Rechstrukturen, Springer Verlag 1998.
- Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.

Inhaltsverzeichnis

1	VARIABLEN UND ZUWEISUNGEN	3
1.1	Motivation von Variablen	4
1.2	Semantik von Zuweisungen	7
2	ZUSAMMENGESETZTE ANWEISUNGEN	8
2.1	Verzweigungen	9
2.2	Wiederholungsanweisungen (Schleifen)	12
3	METHODEN	17
4	DATENSTRUKTUREN	19
4.1	Arrays	20
4.1.1	Arbeiten mit eindimensionalen Arrays	20
4.1.2	Freigabe von Arrays	22
4.1.3	Suchen in Arrays	23
4.1.4	Zeichen-Arrays	26
4.1.5	Mehrdimensionale Arrays	28
4.2	Strings	29
4.2.1	Stringvergleich	31
4.2.2	Stringmanipulationen	31
4.2.3	Stringkonversion	33
	VERZEICHNISSE	34
	Abkürzungen und Glossar	34

Index	35
Informationen und Interaktionen	35
Literatur	36

- VARIABLEN UND ZUWEISUNGEN
 - Wiederverwendung von Identifikatoren, Zustandsmenge, Semantik
- ZUSAMMENGESetzte ANWEISUNGEN
 - Verzweigungen, Zusicherungen, Schleifen, Schleifeninvarianten
- METHODEN
 - Überladen, Problemzerlegung mittels Methoden
- DATENSTRUKTUREN
 - Arrays, Strings

Information 1: IMPERATIVE PROGRAMMIERUNG

1 VARIABLEN UND ZUWEISUNGEN

In dieser Kurseinheit werden die konzeptionellen Grundlagen der imperativen Programmierung eingeführt und am Beispiel der Programmiersprache Java praktisch umgesetzt. Die Ausführungen zu den Variablen und Zuweisungen orientieren sich an [Br98], während die Java-orientierten Inhalte an [Mö03] angelehnt sind.

- Imperative oder Zuweisungsorientierte Programme: Folge von Anweisungen
- Anweisungen dienen u.a. zur Kontrolle und Steuerung von Abläufen
- Durch die Ausführung einer Anweisung ändert sich der Zustand
- Anweisungen werden häufig in Abhängigkeit vom Speicher- und Ablaufzustand erteilt
- Zustand betrifft zwei Aspekte
 - Speicherzustand (wird auch Datenzustand genannt)
 - Ablaufzustand (wird auch Kontrollzustand genannt)
- Anweisungen ändern gewisse Teile des Speicher- und Ablaufzustands

Information 2: VARIABLEN UND ZUWEISUNG- Zentraler Begriff der Anweisung

Der zuweisungsorientierten Programmierung liegt ein operativer, maschinenorientierter und weniger ein formaler, mathematischer Zugang zur Programmierung zugrunde. Dieses Programmierkonzept hat sich in der Praxis gegenüber dem in der Kurseinheit FUNKTIONALE PROGRAMMIERKONZEPTE UND REKURSION [C&M-FR] behandelten Konzept der funktionalen Programmierung durchgesetzt.

Die Vorgänge, die Maschinenzustandsübergänge bewirken, werden als Anweisungen bezeichnet. Information 2 beinhaltet Aussagen über Anweisungen, durch die diese näher charakterisiert werden.

Es wird deutlich, dass der Zustand bei der zuweisungsorientierten Programmierung eine ganz zentrale Rolle einnimmt. Bislang wurde der Zustand durch den Speicherinhalt des Rechensystems definiert. Bei näherer Analyse, was den Zustand ausmacht, wird eine Zweiteilung sichtbar: der Speicher- oder Datenzustand und der Ablauf- oder Kontrollzustand.

Häufig wird eine Anweisung nur dann erteilt, wenn der Ausgangszustand, der geändert werden soll, bestimmte Eigenschaften aufweist. Hinter diesem Aspekt verbirgt sich offensichtlich die bedingte Zuweisung, die ja gerade in ihrem Bedingungsteil solche Eigenschaften abzurufen erlaubt.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
Statement =
    {Ident ":"}
    ( Block
    | [Expr] ";"
    | "if" "(" Expr ")" Statement ["else" Statement]
    | "switch" "(" Expr ")" "{" {switchGroup} }"
    | "while" "(" Expr ")" Statement
    | "for" "(" [ForInit] ";" [Expr] ";" [ForUpdate] ")" Statement
    | "do" Statement "while" "(" Expr ")"
    | "try" Block (Catch {Catch} [Finally] | Finally )
    | "synchronized" "(" Expr ")" Block
    | "return" [Expr] ";"
    | "throw" Expr ";"
    | "break" [Ident] ";"
    | "continue" [Ident] ";"
    ).
```

Information 3: Anweisungen der Sprache Java im Überblick

Information 3 führt die Anweisungen, die in der Sprache Java bereitgestellt werden, in Erweiterter Backus-Naur-Form (EBNF) auf.

Einige grundlegende Anweisungen, wie z.B. die if- oder while-Anweisung, sind bereits aus der Kurseinheit PROGRAMMIERGRUNDLAGEN [C&M-PG] bekannt. Einige der aufgeführten Anweisungen werden in anderen Kurseinheiten (z.B. FORTGESCHRITTENE PROGRAMMIERKONZEPTE [C&M-FP]) behandelt.

1.1 Motivation von Variablen

Ein offensichtliches Ziel, das mit gebundenen Bezeichnern erreicht werden kann, besteht darin, Zwischenergebnisse mehrfach verwenden zu können. Hierin können Zwischenergebnisse und somit bereits geleistete Berechnungsarbeit gespeichert werden. Das führt zu einer effizienteren Arbeitsweise, weil dadurch die Arbeit gespart wird, das Zwischenergebnis von neuem berechnen zu müssen.



- Mit den gebundenen Bezeichnungen verknüpftes Ziel:
 - Mehrfachverwendung von Zwischenergebnissen
- Berechnung des Ausdrucks: $E1 * E1 - E2 * E2$
 - klassische Notation der Funktionsabstraktion
 $((\text{int } a, b) \text{ int}: a * a - b * b) (E1, E2)$
 - Programm mit Deklarationen (Konstanten, statische Variablen)
 - Konstanten werden in Java durch das Schlüsselwort "final" gekennzeichnet

```
final int a = 17 - 4 * 3;    // E1
final int b = (9 - 5) * 2; // E2
result = a * a - b * b;
```

Die Deklaration der Größe result
ist im Java-Programm zu ergänzen

Interaktion 1: Gebundene Bezeichnungen

Dieser Sachverhalt wird in Interaktion 1 an einem einfachen Beispiel aufgezeigt. Das Ziel bei der Berechnung des Ausdrucks (*Expression*) $E1 * E1 - E2 * E2$ besteht darin, dass jeder der Ausdrücke $E1$ und $E2$ nur einmal ausgewertet werden soll. Erreicht wird dieses Ziel durch die Funktionsabstraktion $((\text{int } a, \text{int } b) \text{ int}: a * a - b * b) (E1, E2)$. Das gleiche Ergebnis wird durch Elementdeklarationen erzielt, hier Deklaration der Konstanten a und b .

Das Java-Programm zeigt das Vorgehen am Beispiel von zwei einfachen Ausdrücken. Der in diesem Programm verwendete Bezeichner `result` ist in Interaktion 1 geeignet zu deklarieren.

Das Problem der Mehrfachberechnung von Teilausdrücken tritt beispielsweise ganz massiv bei der Berechnung von Polynomen auf. Bei traditioneller Berechnung der einzelnen Ausdrücke $a_i x^i$ führt man zwangsläufig eine unnötige Mehrfachberechnung durch, d.h. man muss mehrfach jeweils x^{i-1} ausrechnen. Hier schafft das in Information 4 beschriebene so genannte Horner Schema Abhilfe.

- Bei folgendem Ausdruck (Polynom) müssen die x^i mehrfach berechnet werden:

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

- Umgehung der Mehrfachberechnung durch das Horner Schema

$$(\dots(a_n \cdot x + a_{n-1}) \cdot x + \dots + a_1) \cdot x + a_0$$

- Aufbrechen der Formel, die durch das Horner Schema vorgegeben ist

$$y_{n+1} = 0$$

$$y_n = y_{n+1} \cdot x + a_n$$

$$y_{n-1} = y_n \cdot x + a_{n-1}$$

...

$$y_0 = y_1 \cdot x + a_0$$

- Die Wiederverwendung von Hilfsidentifikatoren führt zu den Variablen

Information 4: Horner Schema als weiteres Beispiel

Eine direkte Übertragung des Horner Schemas mit einer entsprechenden Folge von Vereinbarungen führt zu einer Berechnungsfolge, wie sie in Information 4 angegeben ist. Das Ergebnis ist ein gestaffeltes System von Deklarationen.

Es fällt auf, dass jeder Hilfsidentifikator y_i nach seiner Einführung ausschließlich in der darauf folgenden Zeile verwendet wird. Es liegt also nahe, einen einmal eingeführten Identifikator wieder zu verwenden.

Zur Umsetzung des Horner Schemas wird nur ein Identifikator benötigt. Dieser Identifikator wird mit wechselnden, also variablen Werten belegt, weshalb er als Variable oder Programmvariable bezeichnet wird. Eine Anweisung der Form $v=E$ heißt Zuweisung.

```
final int x = 3; final int a4 = 2; final int a3 = 1; final int a2 = 3; final int a1 = 4; final int a0 = 6;
int v = 0;           // declaration of variable v
v = v*x + a4;
v = v*x + a3;
v = v*x + a2;
v = v*x + a1;
v = v*x + a0;
// Polynomial: _____
// Result: _____
```

- v heißt Programmvariable, $v = E$; ist eine Zuweisung (spezielle Form einer Anweisung)
- Syntax (Ausschnitt)

Assignment	=	AssignmentOp Expr.
AssignmentOp	=	"=" "+=" "-=" "/="

Interaktion 2: Variablen und Zuweisungen

In Interaktion 2 wird das Java-Programm angegeben, durch das die Berechnung eines Polynoms gemäß dem vorgestellten Hornerschema erfolgt. Dabei wird ein festes $n = 4$ angenommen. Nach der Deklaration der Variablen v folgen entsprechende Zuweisungen (*Assignments*), durch die das Hornerschema realisiert wird. Das durch das Programm berechnete Polynom und das Resultat sind als Kommentar im Programm zu ergänzen.

Wie die in Interaktion 2 angegebene Syntaxbeschreibung von Zuweisungen in Java zeigt, sind neben dem Zuweisungsoperator $=$ weitere Zuweisungsoperatoren definiert, durch die sich die Zuweisung mit einer arithmetischen Operation kombinieren lässt.

1.2 Semantik von Zuweisungen

Im obigen Beispiel kommt die Variable v in der Berechnung sowohl auf der linken als auch auf der rechten Seite vor. Betrachtet man das Konzept der Variable und Zuweisung aus der Sicht der klassischen Mathematik, so stößt man bei der Interpretation auf Schwierigkeiten.

Offensichtlich ist bei einer durchaus zulässigen Zuweisung

$$x = x+1$$

die Interpretation des Zuweisungszeichens als Gleichheitszeichen mathematisch nicht möglich. Vielmehr muss die Interpretation in folgender Weise unter Zuhilfenahme des Gleichheitszeichens erfolgen:

$$x_{\text{neu}} = x_{\text{alt}} + 1$$

Die Größe x existiert somit in zwei Zuständen mit zwei Werten: der Wert von x im alten Zustand (x_{alt}) und der Wert von x im neuen Zustand (x_{neu}).

An dieser Stelle wird erneut der Sachverhalt deutlich, dass Anweisungen – in diesem Fall Zuweisungen – Zustände ändern.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Im Zusammenhang mit Anweisungen betrachtete Zustände
 - Belegungen der Identifikatoren, die als Programmvariablen auftreten
 - wird als Menge ENV bezeichnet
 - Hinzufügen eines speziellen Zustands \perp
 - führt insgesamt zu einer Menge STATE = ENV \cup $\{\perp\}$
 - Definition einer partiellen Ordnung \sqsubseteq auf STATE
- Funktionale Bedeutung der Ausführung einer Anweisung
 - Änderung eines Zustands
 - Interpretationsfunktion I: $\langle \text{statement} \rangle \rightarrow (\text{STATE} \rightarrow \text{STATE})$
 - Zustandsabbildung: Zustand vor Anweisung \rightarrow Zustand nach Anweisung
- Operationale Semantik von Anweisungen
 - wird üblicherweise nicht durch (sehr komplexe) Termersetzungssysteme, sondern durch abstrakte Maschinenmodelle beschrieben

Information 5: Funktionale Bedeutung von Anweisungen

Um die Semantik von Anweisungen festlegen zu können, muss zunächst geklärt sein, was formal unter einem Zustand zu verstehen ist. Das führt zur Werte-Belegung der die Variablen darstellenden Identifikatoren und zu der Menge ENV (steht für *Environment*). Dieser Menge wird das so genannte *Bottom*-Symbol oder undefiniert-Symbol \perp hinzugefügt.

Die Zustandsdefinition wird dazu genutzt, die funktionale Bedeutung von Anweisungen anzugeben: Als Semantik (Interpretationsfunktion) einer Anweisung wird eine Zustandsabbildung betrachtet, die für jeden Anfangszustand (Zustand vor der Ausführung der Anweisung) einen Endzustand (Zustand nach Ausführung der Anweisung) erzeugt.

Dieses Vorgehen basierend auf den Zustandsabbildungen ließe sich weiterführen, um die operationelle Semantik von Anweisungen zu beschreiben. Hierzu wäre notwendig, die „Rechenstruktur der Zustände“ in allen Einzelheiten zu beschreiben und dafür ein Termersetzungssystem anzugeben. Aufgrund der Komplexität geht man in der Regel einen anderen Weg, der über die abstrakten Maschinenmodelle führt. Hier treten die Kontroll- und Speicherzustände im Gegensatz zu den Termersetzungssystemen explizit auf, was den Konstruktionsprozess erheblich vereinfacht.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

$$I[x = E](\sigma) = \begin{cases} \sigma[I_{\sigma}[E]/x] & \text{falls } \sigma \neq \perp \wedge I_{\sigma}[E] \neq \perp \\ \perp & \text{falls } \sigma = \perp \vee I_{\sigma}[E] = \perp \end{cases}$$

- Zuweisung $x = E$ bewirkt eine Zustandsänderung von einem Ausgangszustand in einen Nachfolgezustand
 - Ausgangszustand stimmt mit dem Nachfolgezustand für alle Identifikatoren bis auf x überein
 - Bedeutung von $\sigma[I_{\sigma}[E]/x]$: Im Nachfolgezustand liefert x den Wert von E
- Nachfolgezustand ist undefiniert, wenn
 - Ausgangszustand undefiniert ist
 - Semantik von E undefiniert ist

Information 6: Semantik einer Zuweisung

An dieser Stelle wird die Semantik der in der imperativen (zuweisungsorientierten) Programmierung zentralen Anweisung, der Zuweisung, näher betrachtet.

Allgemein lässt sich die Semantik der Zuweisung durch eine in Information 6 angegebene Interpretationsvorschrift formulieren. Diese besagt, dass bei einem Zustand σ die Ausführung der Anweisung $x = E$ zu einem Folgezustand führt, bei dem jedes x gegen $I_{\sigma}[E]$ ersetzt wird.

Das gilt allerdings nur für den Fall, dass weder σ noch $I_{\sigma}[E]$ dem undefinierten Zustand entsprechen. In diesem Fall liefert die Interpretationsvorschrift als Folgezustand der Zuweisung ebenfalls den undefinierten Zustand \perp . Allgemein bestehen zuweisungsorientierte Programme aus einer Folge von Zustandsänderungen, die einer Folge von Zuweisungen entsprechen.

2 ZUSAMMENGESetzte ANWEISUNGEN

Neben den einfachen Anweisungen (wie z.B. die im letzten Kapitel behandelte Zuweisung) bestehen die zusammengesetzten Anweisungen, die aus den einfachen Anweisungen durch verschiedene Formen der Komposition hervorgehen [Br96, Mö03].

- Durch verschiedene Formen der Komposition (Sequenz, Bedingung, Wiederholung) lassen sich aus gegebenen Anweisungen zusammengesetzte Anweisungen erzeugen
 - werden auch als sequentielle Komposition bezeichnet
- Syntax

```
SequentialComposition = Statement ";" {Statement ";"}
```

- Semantik

$$I[S1; S2](\sigma) =_{\text{def}} I[S2](I[S1](\sigma))$$
- Anweisungen werden nacheinander (sequentiell) ausgeführt

Information 7: ZUSAMMENGESetzte ANWEISUNGEN - Syntax und Semantik

Eine Form der Komposition ist die Sequenz oder Aufeinanderfolge. Die Syntax sieht vor, die aufeinander folgenden Anweisungen durch einen Strichpunkt (;) voneinander zu trennen. Semantisch bedeutet die sequentielle Komposition, dass der durch die zuerst aufgeschriebene Anweisung S1 erzeugte Folgezustand der Zustand ist, auf dem die zweite Anweisung S2 aufsetzt.

2.1 Verzweigungen

Eine wichtige zusammengesetzte Anweisung ist die bereits in den PROGRAMMIERGRUNDLAGEN [C&M-PG] eingeführte bedingte Anweisung, durch die eine Zweiweg-Verzweigung realisiert wird.

- Aufteilung des Programmflusses in zwei oder mehr Zweige
 - Zweiweg-Verzweigung: if-Anweisung
 - Mehrweg-Verzweigung: switch-Anweisung
- Semantische Interpretation der if-Anweisung

$$I[\text{if } (C) \text{ S1; else S2;}] (\sigma) =_{\text{def}} \begin{cases} I[S1](\sigma) & \text{falls } \sigma \neq \perp \text{ und } I_{\sigma}[C] = L \\ I[S2](\sigma) & \text{falls } \sigma \neq \perp \text{ und } I_{\sigma}[C] = 0 \\ \perp & \text{sonst} \end{cases}$$

- Kurzschlussauswertung

```
if (y != 0 && x / y > 10) ...;
/* if first comparison y != 0 is false (i.e. y is 0) the second comparison
x / y > 10 is not evaluated */
```

Information 8: Verzweigungen

Bedingte Anweisungen werden in ihrer Bedeutung auf die Fallunterscheidung zurückgeführt. Hat der Ausdruck C in dem vor der Ausführung der bedingten Anweisung bestehenden Zustand den Wert `true`, so wird $S1$ ausgeführt, andernfalls $S2$.

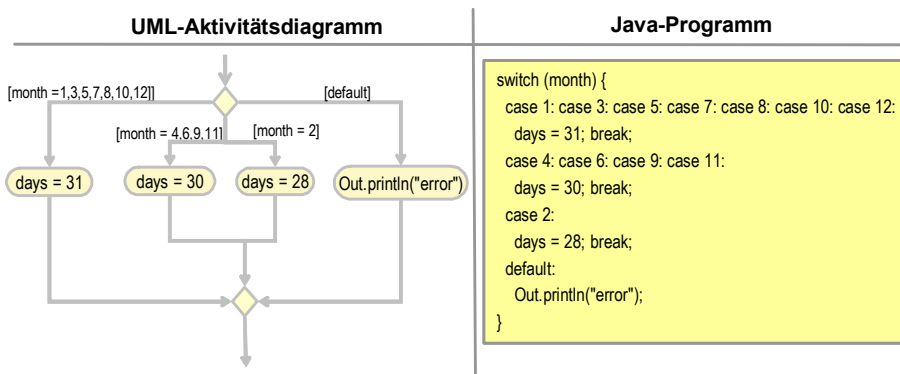
Ist der Wert von C für den Anfangszustand undefiniert (\perp), also weder `true` noch `false`, so ist der Endzustand der bedingten Anweisung der undefinierte Zustand \perp .

Der Programmausschnitt in Information 8 zeigt, wie diese zu einem Laufzeitfehler führende Situation vermieden werden kann. Das Beispiel nutzt dabei die von Java durchgeführte Kurzschlussauswertung – auch als bedingte Auswertung bezeichnet – aus: In der im Beispiel auftretenden Und-Verknüpfung `y != 0 && x / y > 10` wird der zweite Vergleich `x / y > 10` gar nicht mehr ausgewertet, wenn der erste Vergleich `y != 0` den Wert `false` ergibt.

Besteht aufgrund des umzusetzenden Algorithmus die Anforderung, den Programmfluss an einer Stelle in mehrere Zweige zu spalten, bietet sich die Mehrweg-Verzweigung in Form der `switch`-Anweisung an.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- `switch`-Anweisung prüft den Wert eines Ausdrucks
 - Ausdruck kann vom Typ `int`, `short`, `byte` oder `char` sein
 - schlägt in Abhängigkeit davon einen von mehreren möglichen Wegen ein
- Beispiel:



Information 9: Mehrwegverzweigung `switch`-Anweisung

In Information 9 wird die Syntax der `switch`-Anweisung durch ein einfaches Beispiel, das die Anzahl der Tage (Variable `days`) eines Monats (Variable `month`, wobei die Monate Januar bis Dezember von 1 bis 12 durchnummeriert sind).

Ist in `month` z.B. der Wert 1 gespeichert, so steht dieser Wert für den Monat Januar. Die `switch`-Anweisung wählt den ersten Zweig aus (wegen `case 1`) und weist der Variablen `days` den Wert 31 zu (weil der Januar 31 Tage hat). Durch die `break`-Anweisung wird danach an das Ende der `switch`-Anweisung gesprungen.

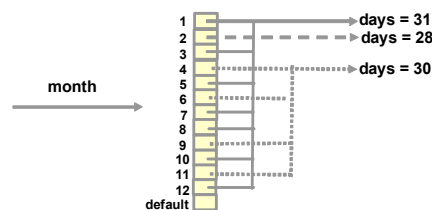
Würde die `break`-Anweisung fehlen – ein häufiger Programmierfehler im Zusammenhang mit der `switch`-Anweisung – käme es zur Bearbeitung der folgenden `case`-Anweisungen und damit zu einem fehlerhaften Programmverhalten.



```
if (month == 1 || month == 3 || ...) days = 31;
else if (month == 4 || month == 6 || ...) days = 30;
else if (month == 2) days = 28;
else Out.println("error");
```

- Jede switch-Anweisung kann in eine semantisch gleichwertige if-Anweisung übertragen werden

- Der vom Compiler erzeugte Code ist unterschiedlich
 - if-Anweisung: sequentielle Prüfung der Fälle
 - switch-Anweisung: jeder Fall ist ein Eintrag in einer vom Compiler angelegten Tabelle
 - Woraus besteht ein Tabelleneintrag?



Interaktion 3: switch-Anweisung und if-Anweisung

Offensichtlich lässt sich eine switch-Anweisung problemlos in eine if-Anweisung überführen, wie an dem Beispiel in Interaktion 3 aufgezeigt wird. Die Programmausschnitte der if-Anweisung und der in Information 9 angegebenen switch-Anweisung sind zwar semantisch gleichbedeutend, werden aber vom Compiler ganz unterschiedlich behandelt: Eine switch-Anweisung wird nicht wie eine if-Anweisung sequentiell durchlaufen, sondern mittels einer Tabelle abgearbeitet, deren Einträge auf den dazu gehörigen Zweig verweist.

- Die Lösung mittels switch-Anweisung
 - hat im Mittel eine kürzere Ausführungszeit als die Lösung mittels if-Anweisung
 - verbraucht im Mittel mehr Speicherplatz als die Lösung mittels if-Anweisung
- Ausführungszeit und Speicherplatz sind die zwei konkurrierenden Optimierungskriterien von Algorithmen
- Bei weit auseinander liegenden case-Marken entstehen große Tabellen
 - if-Anweisung in diesem Fall angemessener
 - gute Compiler übersetzen in einem solchen Fall die switch-Anweisung wie eine if-Anweisung

Information 10: Eigenschaften der beiden Lösungen

Durch die Einführung der Tabelle wird bei der Umsetzung der switch-Anweisung im Mittel der zu bearbeitende Zweig schneller als bei der Lösung mittels if-Anweisung erreicht. Dieser Vorteil der besseren Laufzeit wird erkaufte durch einen im Mittel größeren Speicherplatzverbrauch, der durch die Tabelle verursacht wird.

Durch Zusicherungen (*Assertion*) lassen sich explizite Aussagen über den Programmzustand treffen.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Zusicherungen (engl. Assertions) machen eine Aussage über den Zustand (STATE) des Programms an einer Programmstelle
 - können in Programmen als Kommentare geschrieben werden
 - in diesem Fall keine Auswertung durch den Compiler
- Beispiel: Zusicherungen zum then- und else-Fall

```
int x;
...
if (0 <= x && x < 10)
  // assertion:      0 <= x && x < 10
  //                value of x is in [0 .. 9]
else
  // assertion:      !(0 <= x && x < 10)
  //                !(0 <= x) || !(x < 10) (DeMorgan)
  //                (x < 0) || (x >= 10)
```

Information 11: Zusicherungen

Im ersten Kapitel dieser Kurseinheit wurde der Zustand STATE eingeführt. Hierüber wurde die Interpretationsfunktion I und damit die Semantik der Anweisung definiert.

Wie das Beispiel in Information 11 zeigt, können so genannte Zusicherungen als Kommentare in das Programm eingefügt werden. Eine Auswertung durch den Compiler erfolgt dabei allerdings nicht.

Im Zusammenhang mit der if-Anweisung können bei der Zusicherung zum else-Fall die Regeln von DeMorgan hilfreich sein, um das Negationszeichen zu eliminieren.

2.2 Wiederholungsanweisungen (Schleifen)

In vielen Programmier-Situationen wünscht man die Ausführung einer Anweisung S , bis der erreichte Zustand eine gewisse Bedingung erfüllt. Die Anweisung, durch die diese Art von Aufgabe gelöst wird, ist die Wiederholungsanweisung. Neben der in den PROGRAMMIERGUNDLAGEN [C&M-PG] bereits behandelten while-Anweisung (Abweisschleife) stehen in Java mit der do-while-Anweisung (Durchlaufschleife) und der for-Anweisung (Zählschleife) zwei weitere Schleifenformen zur Verfügung.



- Durch die while-Anweisung wird eine Abweisschleife realisiert
- do-while-Anweisung (Durchlaufschleife)
 - im Unterschied zur while-Schleife wird der Rumpf mindestens einmal durchlaufen

- Beispiel:

```
int n = In.readInt();
if (n < 0) n = -n;    // no negative values
do {
    Out.print(n % 10); // rest from n / 10
    n = n / 10;
while (n > 0);
```

Eingabe: n = 123 Ausgabe: _____

n % 10	n	Durchlauf

Was leistet das Programm? _____

Interaktion 4: do-while-Anweisung

Anhand des in Interaktion 4 angegebenen Beispiels kann man sich leicht klar machen, dass es Aufgabenstellungen gibt, in denen in jedem Fall ein Durchlauf durch die Schleife stattfinden sollte. Falls im obigen Beispiel eine while-Anweisung gewählt worden wäre, würde der Algorithmus für die Eingabe 0 nicht korrekt arbeiten.

Die dritte Schleifenvariante, die Zählschleife, wird häufig dann verwendet, wenn die Anzahl der Schleifendurchläufe im Voraus bekannt ist.

- Die eine Zählschleife realisierende for-Anweisung besteht aus den folgenden Teilen:
 - (1) Initialisierungsteil
 - wird vor dem Betreten der Schleife ausgeführt
 - Laufvariable wird initialisiert
 - (2) Abbruchbedingung
 - wird jedes Mal vor Betreten der Schleife überprüft
 - (3) Inkrementierungsteil
 - wird am Ende jedes Schleifendurchlaufs ausgeführt
 - typische Verwendung: Erhöhen der Laufvariable
 - (4) Schleifenrumpf
- Syntax
"for" "(" Initialisierungsteil";" Abbruchbedingung";" Inkrementierungsteil ")"
Schleifenrumpf ";"

Information 12: for-Anweisung

Die Auflistung der Bestandteile einer for-Anweisung in Information 12 deutet bereits an, dass es sich hierbei um eine komplexere Anweisung handelt, die Java zur Verfügung stellt.

Es ist davon abzuraten, die von Java zugelassenen, umfangreichen Möglichkeiten der for-Schleife auch tatsächlich zu nutzen, da hierdurch die Lesbarkeit erschwert wird. Sinnvoll ist die Beschränkung des Initialisierungs- und Inkrementierungsteils auf eine Anweisung. Komplexere Schleifen sollten deshalb durch eine while-Anweisung realisiert werden.

- Programmbeschreibung:
 - Eingabe: positive ganze Zahl n
 - Ausgabe: $(n \times n)$ - Matrix m mit $m[i, j] = i * j$

- Beispiel für $n = 3$:

1	2	3
2	4	6
3	6	9

- Das die Aufgabe lösende Java-Programm ist zu vervollständigen

```
int n = In.readInt(); if (n < 0) n = -n;           // no negative values
for
```

Interaktion 5: Beispiel zur for-Anweisung

Interaktion 5 zeigt die Aufgabenbeschreibung eines Programms, die durch eine Schachtelung von zwei for-Anweisungen zu realisieren ist.

Schleifen können bewirken, dass das Ende der Schleife niemals erreicht wird, d.h. das Programm terminiert nicht. Die Semantik der Schleife und damit des ganzen Programms ist in diesem Fall undefiniert, was dem weiter oben eingeführten \perp -Symbol entspricht. Ein praktisches Vorgehen, die Terminierung sicher zu stellen, besteht in der Nutzung von Zusicherungen (*Assertions*), die bereits im Zusammenhang mit den Verzweigungen eingeführt wurden.

```

i = 1; sum = 0;
while (i <= n) {
  // Assertion when entering the loop body: i <= n
  sum = sum + i;
  i = i + 1;
}
// Assertion after leaving the loop: i > n

```

- Zwei triviale Zusicherungen (analog Verzweigungen)
 - am Anfang des Schleifenrumpfes
 - nach Verlassen des Schleife
- Eine interessante weitere Zusicherung ist die Schleifeninvariante
 - Aussage über das Ergebnis der Schleife, die in jedem Durchlauf gültig bleibt

Information 13: Zusicherung bei Schleifen

Die in Information 13 enthaltenen zwei Zusicherungen betreffen zum einen Aussagen zu der Abbruchbedingung der Schleife. Eine weitere Zusicherung, die Schleifeninvariante, ist sehr viel schwieriger zu ermitteln und bedarf einiger Erfahrungen.

Am Beispiel einer einfachen Summierungsschleife soll das Prinzip der Schleifeninvariante illustriert werden.



```

i = 1; sum = 0;
while (i <= n) {
  // sum == add(1 .. i - 1)
  sum = sum + i;
  // sum == add(1 .. i)
  i = i + 1;
  // sum == add(1 .. i - 1)
}
// sum == add(1 .. n)

```

- Schleifeninvariante $sum = add(1 .. i - 1)$
 - $add(1 .. x)$ berechnet die Summe $1 + \dots + x$
- Es ist zu zeigen, dass die Schleifeninvariante durch den Schleifenrumpf erhalten bleibt

Interaktion 6: Schleifeninvariante zu einem Beispiel-Programm

Die Frage, was die in Interaktion 6 angegebene Schleife berechnet, lässt sich unmittelbar beantworten. Im Falle einer bereits nur wenig komplexeren Schleife wäre die Antwort und damit die Schleifeninvariante bereits sehr viel schwieriger zu ermitteln.

Nach der Festlegung der Schleifeninvariante $sum == add(1 .. i - 1)$ ist nachfolgend zu beweisen, dass es sich auch tatsächlich um eine Invariante handelt. Hierzu zeigt man, dass nach einem Durchlauf durch die Schleife die Invariante erhalten bleibt.

Konkret müssen zum Nachweis der Schleifeninvariante die zwei Anweisungen, die den Schleifenrumpf bilden, untersucht werden.

Die Zusicherung nach Verlassen der Schleife ($sum = add(1 .. n)$) wird durch eine (Und-) Verknüpfung der Schleifeninvariante und der Zusicherung erreicht, die in Information 13 im Zusammenhang mit der Schleifenbedingung aufgestellt wurde.

Die Terminierung der Schleife lässt sich ebenfalls leicht nachweisen, da zum einen mit n eine obere Schranke gegeben ist und zum anderen der Wert von i , der gegen diese obere Schranke geprüft wird, in jedem Schleifendurchlauf um 1 erhöht wird.

Bei allen drei Schleifenformen (`while`, `do-while`, `for`) wird durch die in Java angebotene `break`-Anweisung die Möglichkeit zugelassen, nicht über die Schleifenbedingung, sondern über eine im Rumpf durchgeführte Abprüfung einer Bedingung die Schleife zu verlassen.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)



- Abbruch innerhalb des Rumpfes durch den Befehl `break`
 - sollte wegen der daraus resultierenden komplexen Programmlogik möglichst vermieden werden

```
// program using break
int sum = 0;
int x = In.readInt(); if (x < 0) x = -x;
while (In.done()) {
    sum = sum + x;
    if (sum < 1000)
        Out.println("sum is " + sum);
    else {
        Out.println("error: sum is too big");
        break;
    }
    x = In.readInt();
}
```

```
// equivalent program without break
int sum = 0;
int x = In.readInt(); if (x < 0) x = -x;
while (In.done() && sum + x < 1000) {
    sum = sum + x;
    Out.println("sum is " + sum);
    x = In.readInt();
}
sum = sum + x;
if _____ // add condition
    Out.println("error: sum is too big");
```

Interaktion 7: Abbruch von Schleifen

In dem in Interaktion 7 gezeigten linken Programmausschnitt, der die `break`-Anweisung beinhaltet, ist die Bedingung für den Abbruch der Schleife dann gegeben, wenn die Variable `sum` nicht kleiner als 1000 ist.

Zum Verständnis des Programms ist ein Blick in die Klasse `In` und der von ihr bereitgestellten Methode `done()` hilfreich. Im Java-Quellcode findet sich die folgende Erklärung:

```
/** Check if the previous operation was successful.
This method returns true if the previous read operation was able
to read a token of the requested structure. It can also be called
after open() and close() to check if these operations were successful.
If done() is called before any other operation it yields true.
```

```
*/
```

Ein Blick auf die Erklärung der Methode `readInt()` verschafft dann die endgültige Klarheit:

```
/** Read an integer.
```

```
This method skips white space and tries to read an integer. If the  
text does not contain an integer or if the number is too big, the  
value 0 is returned and the subsequent call of done() yields false.  
An integer is a sequence of digits, possibly preceeded by '-'.  
*/
```

Die Schleife wird also verlassen, sobald der Benutzer keine korrekte *Integer*-Zahl eingibt.

Wie die gleichwertige Programmvariante auf der rechten Seite zeigt, lässt sich die Verwendung der `break`-Anweisung in manchen Fällen dadurch umgehen, dass die Abbruchbedingung in der Schleifenbedingung untergebracht wird. Das Programm wird dadurch meist übersichtlicher, da die Zusicherung nach der Schleife klarer formuliert werden kann.

Im Beispiel ist nach dem Durchlauf durch die Schleife in einer `if`-Anweisung zu ermitteln, ob die Schleife "korrekt" (d.h. `ln.done() == false`) oder aufgrund der zur Schleifenbedingung hinzugefügten Abbruchbedingung (d.h. `(sum + x < 1000) == false`) beendet wurde. Die entsprechende Bedingung ist in Interaktion 7 entsprechend zu ergänzen.

3 METHODEN

Methoden bieten eine wichtige Grundlage zur Strukturierung von Programmen. Nachfolgend werden die in den PROGRAMMIERGRUNDLAGEN [C&M-PG] gemachten Ausführungen um einige fortgeschrittene Konzepte zu den Methoden ergänzt [Mö03].

Zu einem Methodennamen können durch das so genannte Überladen mehrere Methodenausführungen, bestehend aus der formalen Parameterliste und dem Methodenrumpf, in einem Block existieren.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Es darf ein Methodennamen mehrfach in einem Block auftreten, wenn sich diese Methoden in ihrer Parameterliste unterscheiden
- Das damit verknüpfte Konzept wird als Überladen von Methoden bezeichnet
 - mit einem Methodennamen sind verschiedene Bedeutungen verbunden
- Es wird diejenige Methode beim Aufruf ausgewählt, deren formale Parameterliste zu der aktuellen Parameterliste am besten passt

Information 14: METHODEN – Überladen von Methoden

Die Auswahl der jeweiligen Methodenausführung erfolgt beim Aufruf aufgrund der aktuellen Parameterliste, wie am nachfolgenden Beispiel verdeutlicht wird.

```

static void print(int i) {...}
static void print(float f) {...}           // overload ok: different type
static void print(int i, int width) {...}  // overload ok: different number of parameters

print(100);                               // calls print(int i)
print(3.14f);                              // calls print(float f)
print(100, 5);                             // calls print(int i, int width)

```

- Kriterien zur Unterscheidung der Parameterliste
 - Typ der formalen Parameter
 - Anzahl der formalen Parameter
- Die zur Ausgabe verwendete Methode `Out.print()` ist ein weiteres Beispiel einer überladenen Methode

Information 15: Beispiele für das Überladen von Methoden

Die in Information 15 angegebenen Aufruf-Beispiele ermöglichen eine eindeutige Zuordnung zu einer der drei Varianten der überladenen Methode `print()`. Die Typangabe müsste hierbei gar nicht zwingend exakt übereinstimmen, da z.B. der Aufruf der `print()`-Methode mit einer Variablen vom Typ `short` aufgrund der zulässigen Konvertierung nach `int` mit der ersten Variante `void print (int i) {...}` ausgeführt wird.



- Methoden können zur Zerlegung von Problemen genutzt werden
- Beispiel: Methode zum Kürzen eines Bruchs nutzt die Methode zur Ermittlung des ggT

```

static int gcd (int x, int y) {           // compute greatest common divisor of x and y
    int rest = x % y;                    // modulo operation (rest of x / y)
    while (rest != 0) {
        _____; _____; _____; // add three missing statements
    }
    return y;
}

static void reduce (int numerator, int denominator) {
    int x = _____; // add right side of assignment
    Out.print(numerator / x) + "/" + (denominator / x));
}

```

Interaktion 8: Problemzerlegung mittels Methoden

Am Beispiel des in Interaktion 8 aufgezeigten Kürzen von Brüchen soll verdeutlicht werden, dass Methoden dazu geeignet sind, die aus der Problemzerlegung resultierenden Teilprobleme wiederum in Form von Methoden zu lösen. In diesem Fall besteht das Problem aus dem Kürzen

eines Bruchs. Ein hierin auftretendes Teilproblem ist die Ermittlung des größten gemeinsamen Teilers (ggT, *Greatest Common Divisor* GCD), das durch eine separate Methode gelöst wird.

Die zum Kürzen erforderliche Ermittlung des größten gemeinsamen Teilers erfordert den bereits kennen gelernten und bekannten Euklidischen Algorithmus, der in der Methode `gcd()` in Interaktion 8 realisiert ist und in der zweiten Methode `reduce()`, die das Kürzen eines übergebenen Bruchs realisiert, genutzt wird.

Falls die Methode `gcd()` bereits bestehen sollte (z.B. innerhalb einer Bibliothek mathematischer Funktionen), würde diese zur Lösung des Problems "Kürzen von Brüchen" wieder verwendet werden.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Wiederverwendung von Code
 - Programm wird kürzer und lesbarer
 - Nutzung von Bibliotheken verkürzt diestellungszeit und macht die Programme robuster
- Erweiterung des Angebots an Operationen
 - die von einer Sprache angebotenen Grundoperationen lassen sich durch eigene Operationen erweitern
 - Durchbrechen der Begrenzungen der Sprache
- Strukturierung von Programmen
 - Aufteilen einer unstrukturierten langen Anweisungsfolge in überschaubare und logisch zusammen gehörige Programmstücke
 - gezielte Verwendung von sprechenden Methodennamen ermöglicht ein Verständnis auch ohne die genaue Analyse der einzelnen Anweisungen

Information 16: Gründe für die Nutzung von Methoden

Die drei in Information 16 genannten Gründe, die für den Einsatz von Methoden sprechen, treffen alle für das eben behandelte Beispiel zu.

4 DATENSTRUKTUREN

Neben den ablauforientierten Sprachelementen werden in der Programmierung auch datenorientierte Sprachelemente benötigt, durch die Einzelwerte in Datenstrukturen zusammen geführt werden.

- Die zunächst behandelten Datenstrukturen, die intensiv in der imperativen Programmierung genutzt werden, sind Arrays und Strings
- Array (Feld)
 - eine ein- oder mehrdimensionale Tabelle von Elementen
 - Elemente haben alle den gleichen Typ
 - Zugriff auf die Array-Werte über Nummern (Index)
- String (Zeichenkette)
 - vergleichbar einem Array von Zeichen
 - aufgrund der hohen Bedeutung dieser Datenstruktur wird in vielen Sprachen (wie auch in Java) ein eigener Typ eingeführt
 - Zahlreiche Operationen (z.B. Vergleiche)

Information 17: DATENSTRUKTUREN - Überblick

Mit den Arrays und den Strings werden im Folgenden die zwei elementaren Datenstrukturen behandelt, die in jeder modernen imperativen Programmiersprache angeboten werden.

4.1 Arrays

Die Datenstruktur der Arrays bietet die Möglichkeit, Variablen gleichen Typs zu einer Wertemenge zusammen zu fassen.

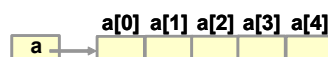
4.1.1 Arbeiten mit eindimensionalen Arrays

Ein Array kann aus mehreren Dimensionen bestehen. Zunächst wird die einfachste Form, das eindimensionale Array, das einem Vektor von Werten entspricht, näher betrachtet.

- Zunächst Betrachtung von eindimensionalen Arrays
- Wichtigste Eigenschaften eines Arrays
 - hat einen Namen
 - ermöglicht den Zugriff auf Array-Elemente über einen Index
 - alle Elemente sind vom gleichen Typ
 - Elemente sind Speicherzellen und verhalten sich wie namenlose Variablen

- Beispiel:


```
int[] a;           // Declaration of an array of int elements
a = new int[5];    // Generation of an array with 5 int elements
```



Information 18: Arrays

Wie Information 18 zeigt, erfolgt der Zugriff auf die einzelnen Elemente des Arrays über einen Index. Die Deklaration eines Arrays erkennt man an den eckigen Klammern [], die an einen beliebigen Typ angehängt werden. Im Beispiel wird mit

```
int[] a;
```

ein Array aus int-Elementen mit der Bezeichnung a deklariert.

Die eigentliche Speicherreservierung wird durch den new-Operator erbracht, der im Beispiel

```
a = new int[5];
```

Speicher für fünf int-Elemente vorsieht, die über die Array-Variable a unter Angabe eines Indexes zugegriffen werden können.

- Einer Array-Variablen kann zu einem späteren Zeitpunkt ein neues Array zugeordnet werden

```
int[] a;
a = new int[5];
...
a = new int[100];
```

- Die Länge eines Arrays kann abgefragt werden

```
a.length;
```

- Ein Index kann ein beliebiger ganzzahliger Ausdruck sein

```
a[2 * i + 1] = a[j];
a[max(i, j)] = 100
```

- Die for-Anweisung bietet sich an, um Arrays zu durchlaufen, Beispiele:

- Einlesen von Array-Elementen
- **Summieren von Array-elementen (zu ergänzen)**

```
for (int i = 0; i < a.length; i++)
    a[i] = In.readInt();
// Calculate sum of array elements
```

Interaktion 9: Arbeiten mit Arrays

Wie Interaktion 9 verdeutlicht, kann einer Array-Variablen, der bereits ein erzeugtes Array zugewiesen wurde, zu einem späteren Zeitpunkt ein anderes Array zugewiesen werden. Dieses Array muss zu dem in der Deklaration angegebenen Typ kompatibel sein und kann kürzer oder länger als das zuvor zugewiesene Array sein. Auf Werte, die eventuell in dem zuvor zugewiesenen Array eingetragen wurden, lässt sich nach einer Zuweisung eines neuen Arrays nicht mehr über diese Array-Variable zugreifen.

Wie das in Interaktion 9 entsprechend zu ergänzende Beispiel einer Array-Bearbeitung zeigt, bietet sich die for-Anweisung zum Durchlaufen der Array-Elemente an.



```
int[] a; b

a = new int[3];

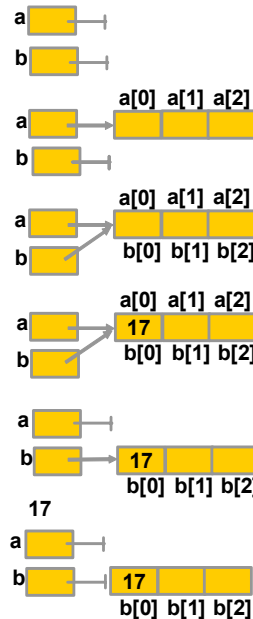
b = a;

a[0] = 17;

a = null;

Out.print(b[0]);

b = null;
```



- Bei der Deklaration einer Array-Variablen wird diese mit null vorbelegt
- Mehrere Array-Variablen können auf dasselbe Array zeigen
- Wie kann auf das Array nach der letzten Anweisung (`b = null;`) zugegriffen werden?

Interaktion 10: Array-Zuweisung

Anhand eines konkreten Ablaufs sollen die verschiedenen Formen der Array-Zuweisung verdeutlicht werden (siehe Interaktion 10). Zunächst werden zwei Array-Variablen `a` und `b` deklariert. Anschließend wird `a` ein mittels `new` erzeugtes 3-elementiges Array zugeordnet. Durch die Array-Zuweisung

```
b = a;
```

zeigt `b` auf dasselbe Array wie `a`. Hierdurch kann auch dann noch auf die Werte dieses Arrays zugegriffen werden, selbst wenn `a` ein anderes Array (im Beispiel das `null`-Array) zugewiesen werden sollte. `null` ist der so genannte *Nullpointer*, was bedeutet, dass der Array-Variablen kein Array zugeordnet ist.

4.1.2 Freigabe von Arrays

Nach Durchführung der obigen Anweisungsfolge zeigt keine Array-Variable auf den Array-Speicher, womit dieser Speicher nicht mehr zugänglich ist und damit frei gegeben werden kann.

- Sobald ein Array von keiner Variablen mehr referenziert wird, kann dieser Speicherplatz freigegeben werden
- Die Freigabe von dynamisch erzeugtem Speicher (`new`-Anweisung) ist nicht Aufgabe des Programmierers sondern wird von Java automatisch durchgeführt
- Vorteile einer automatischen Speicherbereinigung
 - Entlastung des Programmierers
 - Vermeidung von Programmierfehlern
 - Freigabe eines Speicherbereichs, obwohl noch ein oder mehrere Zeiger auf diesen Bereich zeigen

Information 19: Speicherbereinigung (*Garbage Collection*)

Wie in Information 19 ausgeführt ist, erfolgt die Freigabe und Bereinigung des Speichers, die *Garbage Collection*, automatisch durch das Java-Laufzeitsystem, was erhebliche Vorteile für den Programmierer mit sich bringt. In Sprachen wie z.B. C bedeutet die durch den Programmierer selbst durchzuführende Speicherfreigabe einen erheblichen Aufwand und eine Quelle schwer zu behobender Programmierfehler.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Es bestehen verschiedene Möglichkeiten, ein initialisiertes Array zu erstellen
 - Langform: Deklaration, Erzeugung, Initialisieren der einzelnen Array-Elemente
 - Kurzform 1: Deklaration, Initialisierung aller Elemente bei der Erzeugung
 - Kurzform 2: Erzeugung und Initialisierung aller Elemente bei der Deklaration

```
// long variant
int[] primes;
primes = new int[5];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
primes[4] = 11;
```

```
// short variant 1
int[] primes;
primes = new int[]{2, 3, 5, 7, 11};
```

```
// short variant 2
int[] primes = {2, 3, 5, 7, 11};
```

Information 20: Varianten zur Erstellung von initialisierten Arrays

Am Beispiel eines 5-elementigen Primzahlen-Arrays wird in Information 20 verdeutlicht, dass Java elegante und die Lesbarkeit erhöhende Möglichkeiten anbietet, die Initialisierung eines Arrays mit der Erzeugung (Kurzform 1) oder mit der Deklaration und der Erzeugung in einer Anweisung zu verbinden.

4.1.3 Suchen in Arrays

Arrays werden zur Lösung zahlreicher Programmierprobleme benötigt. Ein typisches solches Beispiel ist das Suchen eines Eintrags in einem Katalog (z.B. Telefonbuch, Warenbestand, Kurskatalog).



- Eingabe von seqSearch()
 - gesuchter Zahlwert
 - Array mit ganzen Zahlen, in dem der Zahlenwert gesucht wird
- Ausgabe von seqSearch()
 - Position des gesuchten Zahlwerts im Array, falls der Zahlwert im Array enthalten ist
 - -1, falls Zahlwert nicht im Array enthalten ist

```
/* searches elem in a[] sequentially
 * returns position in a[], if found and -1, if not found
 */
static int seqSearch (int elem, int[] a) {
    int pos = a.length - 1;           // pos initialized to position of last array element
    while (pos >= 0 && a[pos] != elem)
        pos--;
    // afterloop assertion: _____
    return pos;
}
```

Interaktion 11: Sequentielle Suche

Das Beispiel in Interaktion 11 zeigt eine einfache Such-Funktion, die eine Zahl x in einem Array `int[] a` von ganzen Zahlen sucht, indem die Zahlen im Array von der letzten Position `pos` beginnend mit dem gesuchten Zahlenwert verglichen werden.

Die angegebene Lösung liefert zwar das gewünschte Ergebnis, ist aber offensichtlich nicht effizient. Insbesondere bei großen Katalogen (z.B. mit 10^4 Einträgen und mehr) stellt sich das sequentielle Durchlaufen des Arrays als nicht akzeptabel – weil zu aufwändig – heraus.

Durch eine Anforderung an die Eingabe lässt sich die Effizienz der Suche erheblich steigern, wie in Information 21 näher ausgeführt wird.

- Analoge Beschreibung der Such-Funktion mit folgender neuer Anforderung an die Eingabe:
 - das Array, in dem der Zahlenwert gesucht werden soll, liegt sortiert vor

```
/* searches elem in a[] by binary intersection of search interval
 * param first, last define search interval
 * returns position in a[], if found and -1, if not found
 */
static int binSearch (inte elem, int[] a, int first, int last) {
    if (first > last) return -1;    // search interval empty; elem not found
    int m = (first + last) / 2;    // select middle position m in search interval
    if (elem == a[m]) return m;   // found
    if (elem < a[m]) return binSearch(elem, a, first, m-1);    // search elem in first half
    return binSearch(elem, a, m+1, last); // only reached if elem > a[m]
}
```

Information 21: Binäre Suche

Die Anforderung besteht darin, dass das Array $a[]$, in dem der Zahlenwert $elem$ gesucht wird, sortiert ist. In diesem Fall kann anstelle der sequentiellen Suche eine so genannte binäre Suche erfolgen, die den Namen aufgrund der fortlaufenden Zweiteilung (binäre Zerlegung) des Arrays erhält.

Es bietet sich an, die binäre Suche durch eine Methode `binSearch()` zu realisieren, wie in Information 21 angegeben ist. Diese Methode hat die interessante Eigenschaft, dass sie sich selbst aufruft, weshalb sie als rekursive Methode bezeichnet wird. Auf Rekursionen wird in der Kurseinheit FUNKTIONALE PROGRAMMIERKONZEPTE UND REKURSION [C&M-FR] genauer eingegangen.

Aufgrund der Sortiert-Eigenschaft des Arrays lässt sich nach Auswahl eines Elements – im Algorithmus wird das bezüglich der Position in der Mitte ($int\ m = (first + last) / 2$) des Arrays liegende Element gewählt – entscheiden, ob das gewählte Element das gesuchte Element ist oder falls nicht, ob das Element in der ersten oder zweiten Hälfte des Arrays zu suchen ist.

Das auf diese Weise entsprechend eingegrenzte Suchintervall erfolgt durch die Angabe der Position des ersten (`first`) und des letzten (`last`) Elements im Array als Parameter der Methode `binSearch()`.

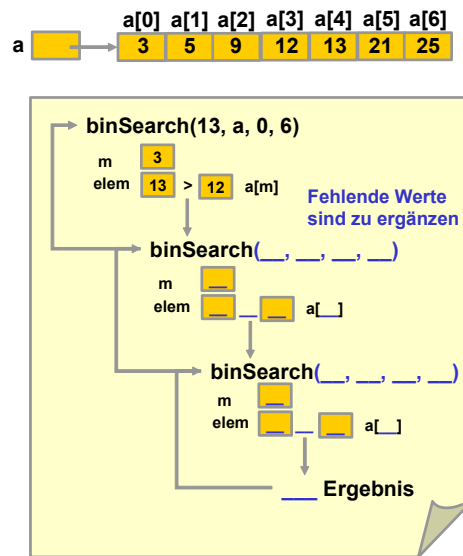


```

...
int a[] = {3, 5, 9, 12, 13, 21, 25};
int pos = binSearch(13, a, 0, 6);
Out.print(pos);
...

```

- Die binäre Suche ist effizienter als die sequentielle Suche, da weniger Vergleiche relativ zur Anzahl n der Array-Elemente erforderlich sind
- sequentielle Suche:
durchschnittlich $n/2$ Vergleiche
- binäre Suche:
durchschnittlich $\log_2 n$ Vergleiche



Interaktion 12: Beispielablauf zur binären Suche

Die Arbeitsweise des Algorithmus, der der binären Suche zugrunde liegt, wird am Beispiel des in Interaktion 12 skizzierten Beispielablaufs verdeutlicht.

Da bei der binären Suche jeder Vergleich bewirkt, dass die Anzahl der noch zu durchsuchenden Elemente im Durchschnitt halbiert wird, ist dieses Vorgehen im Vergleich zur sequentiellen Suche sehr viel effizienter. Ist n die Anzahl der Elemente des Arrays, so ist der Aufwand des Algorithmus, der die sequentielle Suche ausführt, von linearer Ordnung, also $O(n)$, während der Algorithmus zur binären Suche nur einen Aufwand logarithmischer Ordnung, also $O(\log n)$ bedeutet.

Es sei daran erinnert, dass die effiziente binäre Suche nur dann erfolgen kann, wenn das Array sortiert ist. Sortieralgorithmen stellen aus diesem Grund eine wichtige Algorithmenklasse in der Informatik dar.

4.1.4 Zeichen-Arrays

Array-Elemente können von einem beliebigen Typ sein, so z.B. auch Zeichen, was zu den so genannten Zeichen-Arrays oder char-Arrays führt.

- Deklaration von Zeichen-Arrays analog zu bisher behandelten Zahlen-Arrays
- In einem neu erzeugten Array enthalten alle Elemente den Wert `\u0000`
- Kurzformen zur Deklaration und Erzeugung bzw. zur Deklaration, Erzeugung und Initialisierung

```
char[] s; // declaration
           // no array generated so far

s = new char[20];
           // 20 array elements generated
           // and assigned to s

char[] s1 = new char[20];
           // declaration and generation

char[] s2 = {'a', 'b', 'c'};
           // declaration, generation
           // and initialization
```

Information 22: Zeichen-Arrays

Im Grundsatz lassen sich die auf die Zahlen-Arrays bezogenen Ausführungen zur Deklaration, zur Erzeugung und zur Initialisierung auch auf Zeichen-Arrays (oder auch Arrays eines anderen Typs) anwenden.

Solange die Elemente in einem erzeugten Zeichen-Array nicht initialisiert wurden, weist Java diesen das voreingestellte (Default) Unicode-Zeichen `\u0000` zu.

Auch in Zeichen-Arrays ist wie in Zahlen-Arrays die Suche eine nahe liegende Problemstellung. Die in Interaktion 13 gestellte Aufgabe besteht darin, das Auftreten einer Zeichenkette `pat` (steht für *Pattern*, d.h. Muster) in einer Zeichenkette `t` zu finden.

- Eingabe
 - zwei Zeichen-Arrays `t` und `pat`
- Ausgabe
 - Position des ersten Auftretens von `pat` in `t`, falls `pat` in `t` vorkommt
 - -1 sonst

```
static int stringPos (char[] t, char [] pat) {
    int i, j;
    int last = t.length - pat.length;      // last possible position of pat in t
    for (i=0; i <= last; i++) {
        if (t[i] == pat[0]) {              // first character of pat matches
            j = 1;
            while (j < pat.length && pat[j] == t[i+j]) j++;
            _____ // found
        } // if
    } // for
    _____ // not found
}
```

Interaktion 13: Suche in einem Zeichen-Array

Der Algorithmus zu diesem Problem liegt auf der Hand: Bis zu der letzten möglichen Position `last`, in der `pat` in `t` auftreten kann, wird in einer äußeren `for`-Schleife überprüft, ob das erste Zeichen von `pat` (also `pat[0]`) mit dem untersuchten $(i+1)$ -ten Zeichen in `t` (also `t[i]`) übereinstimmt.

Wird ein solches erstes übereinstimmendes Zeichen gefunden, wird in der inneren `while`-Schleife dann festgestellt, ob die nachfolgenden Zeichen in `t` ebenfalls mit den weiteren Zeichen in `pat` übereinstimmen. Kann eine Übereinstimmung für alle Zeichen in `pat` festgestellt werden ist die gesuchte Position gefunden. Hierbei ist zu beachten, dass `pat[length(pat)-1]` das letzte Zeichen in `pat` ist.

&3

Zusicherung am Schleifenende

```
j == pat.length || pat[j] != t[i+j]
```

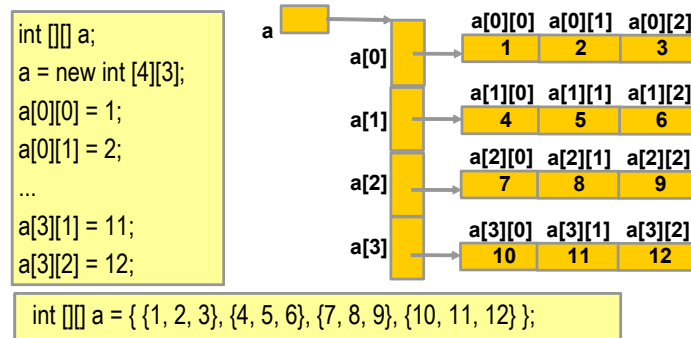
In Interaktion 13 ist das Java-Programm um die entsprechenden Anweisungen zu ergänzen, die insbesondere die ErgebnISRückgabe der Methode `stringPos()` realisieren.

4.1.5 Mehrdimensionale Arrays

Arrays können als Elemente wiederum Arrays enthalten, was zu den mehrdimensionalen Arrays führt.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Bislang betrachtete eindimensionale Arrays entsprechen einem Vektor
- Sind die Vektorelemente wiederum Vektoren, handelt es sich um zweidimensionale Arrays
 - entsprechen den Matrizen
 - das Prinzip kann auf n-dimensionale Arrays ausgedehnt werden



Information 23: Mehrdimensionale Arrays

Die beiden in Information 23 gezeigten Programmausschnitte sind gleichbedeutend: Ein zweidimensionales Array mit 4 Zeilen und 3 Spalten, also eine 4x3-Matrix, wird nach dessen Deklaration (`int a[][]`) und Erzeugung (`a = new int [4][3]`) mit den Werten 1 bis 12 (zeilenweise) initialisiert. Wie bereits bei den eindimensionalen Arrays kennen gelernt, lässt sich dieser Vorgang in einer einzigen Anweisung zusammenfassen.

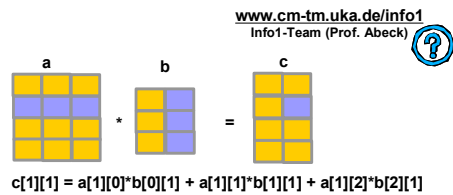
Der Umgang mit mehrdimensionalen Arrays soll am Beispiel der Matrixmultiplikation verdeutlicht werden.

- Eingabe

- Matrizen $a[n][y]$ und $b[y, m]$

- Ausgabe

- Matrixprodukt $c[n,m]$ von a und b



```
static float[][] matMult (float [][] a, float [][] b);
// assertion: _____
float [][] c = new float[a.length][b[0].length];
int i, j, k
for (i = 0; i < a.length; i++) // for all rows of a
  for (j = 0; j < b[0].length; j++) { // for all columns of b
    float sum = 0;
    for (k = 0; k < b.length; k++) // compute c[i, j]
      sum = sum + a[i, k] * b[k, j];
    c[i, j] = sum;
  }
return c;
}
```

Interaktion 14: Matrixmultiplikation

Die Multiplikation von zwei Matrizen a und b setzt voraus, dass die Spaltenanzahl der Matrix a mit der Zeilenanzahl der Matrix b übereinstimmt. In Interaktion 14 ist das Prinzip der Matrixmultiplikation für eine 4×3 -Matrix a und eine 3×2 -Matrix b gezeigt, deren Matrixprodukt $a * b$ eine 4×2 -Matrix c ergibt.

Die Methode `matMult()` erhält die zwei zu multiplizierenden Matrizen als Eingabe und liefert das Matrixprodukt als Ergebnis. Die oben beschriebene Anforderung an die zu multiplizierenden Matrizen ist als eine Zusicherung im Programm zu ergänzen.

4.2 Strings

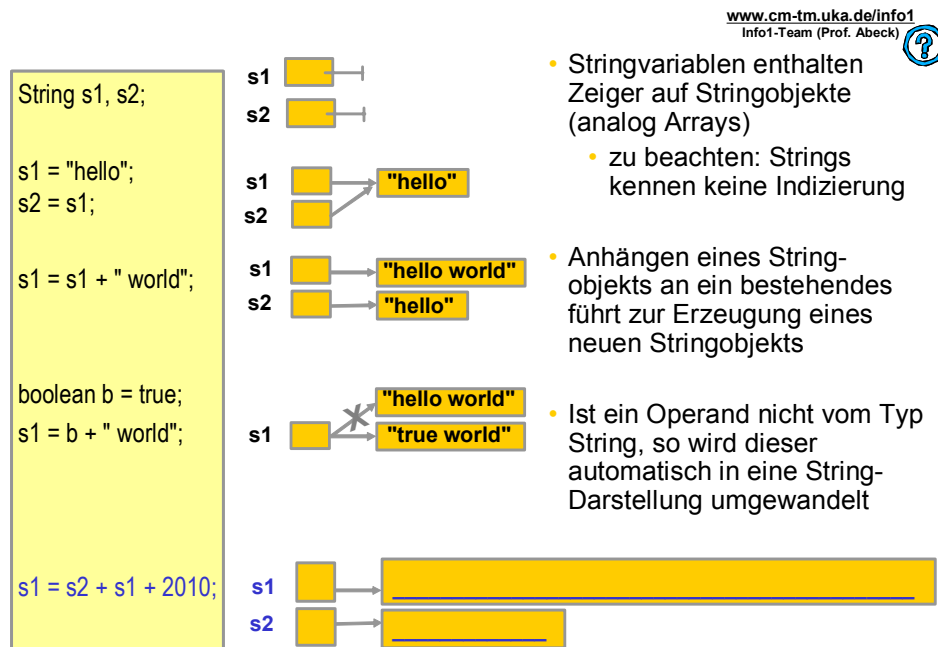
Im vorhergehenden Abschnitt wurden mit den Zeichen-Arrays bereits eine mit den Strings vergleichbare Art einer Datenstruktur vorgestellt. Da diese Datenstruktur, die als Zeichenketten bezeichnet werden, sehr häufig auftritt, wurde in Java ein eigener Bibliothekstyp `String` vorgesehen.

- String ist ein von Java bereit gestellter Bibliothekstyp
 - teilweise in die Sprache integriert, weil der Compiler in bestimmten Fällen speziellen Code für Stringoperationen erzeugt
- Stringkonstanten sind Zeichenfolgen, die in doppelte Hochkommata gestellt sind
 - z.B. "a string", "containing a \" character", "\u03c0 is pi"
- Unterschied "x" und 'x'
 - "x" ist eine Stringkonstante mit nur einem Zeichen
 - 'x' ist eine Zeichenkonstante vom Typ char

Information 24: Strings

String ist nicht wie die bislang kennen gelernten Datentypen (z.B. int, float, char) ein in der Sprache Java verankerter Typ, sondern ein Bibliothekstyp. Da der Compiler speziellen Code für Stringoperationen erzeugen kann, besteht aber ein enger Bezug mit der Sprache Java.

Stringkonstanten sind an den doppelten Hochkommata zu erkennen und enthalten eine endliche Menge von Zeichen. Es können die zur Darstellung von Zeichen bestehende *Escape*-Sequenzen in der in Information 24 gezeigten Art und Weise in Stringkonstanten genutzt werden. Stringkonstanten bestehen aus beliebig vielen Zeichen; sie können also auch nur aus einem Zeichen oder auch aus keinem Zeichen ("") bestehen, was als leerer String bezeichnet wird.



Interaktion 15: Stringvariablen

Strings weisen eine zu den Arrays ähnliche Zeiger-Charakteristik auf. Durch den Operator + lassen sich Strings verketteten. Hieran wird deutlich, dass ein Compiler den String-Typ kennen muss, obwohl dieser nicht in der Sprache, sondern nur in der Bibliothek definiert ist.

4.2.1 Stringvergleich

Im Zusammenhang mit Strings sind zwei Arten von Vergleichen zu unterscheiden. Die Operation `==` führt einen Vergleich auf der Zeigerebene durch – d.h. sie liefert dann `true`, wenn die miteinander verglichenen Strings auf dasselbe Stringobjekt zeigen. Sollen die Strings daraufhin verglichen werden, ob sie die gleichen Zeichenfolgen als Werte haben, ist nicht ein Zeiger-Vergleich, sondern ein Werte-Vergleich erforderlich. Hierzu stellt die Bibliothek für den Stringtyp eine geeignete Methode `equals()` zur Verfügung.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)



- Bei Strings ist zu unterscheiden zwischen

- Zeigervergleich
- Wertevergleich

```
String s = "Hel"; s = s + "lo";
Out.print(s == "Hello");    // prints _____
Out.print(s.equals("Hello")); // prints _____
```

- `s.length()` liefert die Länge des Strings `s`

```
String s = "this is a string";
int i = s.length();    // i == _____
```

- `s.charAt(int pos)` liefert das Zeichen, das sich auf Position `pos` im String `s` befindet

```
String s = "this is a string";
char ch = s.charAt(3);    // ch == _____
```

- `s1.startsWith(String s2)` liefert `true`, wenn `s1` mit `s2` anfängt

```
String s1 = "this is a string";
String s2 = "this is";
bool b = s1.startsWith(s2); // b == _____
```

Interaktion 16: Stringvergleich und weitere Stringoperationen

Der Werte-Vergleich `equals()` ist nur eine von zahlreichen Operationen, die im Zusammenhang mit Strings dem Java-Programmierer zur Bearbeitung von Strings von der Bibliothek angeboten werden. Interaktion 16 beschreibt eine kleine Auswahl von Methoden, deren Aufruf jeweils in einem Programmausschnitt verdeutlicht wird.

4.2.2 Stringmanipulationen

Da das einer Stringvariablen zugewiesene Objekt konstant ist, eignen sich diese Variablen nicht für dynamische Veränderungen, durch die das Objekt schrittweise manipuliert werden kann.

- Stringobjekte sind konstant und eignen sich nicht dazu, flexible Stringmanipulationen anzubieten
- Zwei Möglichkeiten: char-Arrays oder StringBuffer
- char-Arrays
 - Zusammensetzen des Strings im char-Array und anschließende Umwandlung in einen String

```
char[] a = new char[80];
for (int i = 0; i < 80; i++) a[i] = ln.read();
String s1 = new String(a);           // generation of string object
String s2 = new String(a, 0, 40);    // first 40 characters a[0] ... a[39]
```

Information 25: Stringmanipulationen

Information 25 zeigt mit den char-Arrays eine von zwei Möglichkeiten auf, solche Manipulationen durchzuführen. In dem Beispiel wird die Zeichenkette in einer Schleife zeichenweise eingelesen. Anschließend wird ein Stringobjekt s1 erzeugt, dem der Inhalt des gesamten char-Arrays als Wert zugewiesen wird. Wie anhand der Erzeugung des zweiten Stringobjekts s2 ersichtlich, kann durch die Angabe zweier Positionsangaben nur ein Teil eines char-Arrays in ein Stringobjekt umgewandelt werden.



- StringBuffer wird wie String als Typ von der Java-Bibliothek angeboten
 - liefert flexible Möglichkeiten zur Manipulation der Zeichenkette

```
StringBuffer b = new StringBuffer("sBuf cont"); // declaration, generation and initialization
Out.println(b.length()); // length of b; prints _____
Out.println(b.append("ent")); // concatenates " ent" at the end of b
// prints: _____
Out.println(b.delete(2, 4)); // deletes characters from index 2 to index 3
// prints: _____
Out.println(b.replace(0, 2, "strBuf")); // replaces characters from position 0 to 1
// string "strBuf"
// prints: _____
String s1 = b.toString(); // convert buffer string to string
String s2 = b.substring(5, 7); // substring from position 5 to 6 is converted
```

0	1	2	3	4	5	6	7	8
s	B	u	f		c	o	n	t

Interaktion 17: StringBuffer

Der Typ StringBuffer wird wie der Typ String durch die Java-Bibliothek zur Verfügung gestellt. Ein Objekt dieses Typs verhält sich im Wesentlichen wie ein String und bietet zusätzlich umfangreiche Operationen zur Manipulation des zugewiesenen Stringwertes. An dem in Interaktion 17 gezeigten Programmausschnitt werden einige dieser vom Typ StringBuffer bereitgestellten Operationen an einem konkreten Beispiel verdeutlicht.

Die von `StringBuffer` bereitgestellten Methoden, die Manipulationen an dem im Puffer gehaltenen Wert vornehmen, sind Funktionen, die jeweils den veränderten Pufferinhalt zurückgeben.

Die im Beispiel benutzten Methoden stellen nur eine kleine Auswahl dar. Es sei angemerkt, dass in vielen Methoden neben Strings auch andere Typen als Parameter zulässig sind. So erlaubt beispielsweise die Methode `append(x)` mittels Überladung, dass `x` vom Typ `char`, `int`, `long`, `float`, `double`, `boolean`, `String` oder `char[]` ist.

4.2.3 Stringkonversion

Die Konversion, also die Umwandlung von Strings in andere Typen und umgekehrt, ist eine häufig in einem Programm auftretende Aufgabe. Daher stellt die Java-Bibliothek für diesen Zweck geeignete Funktionen zur Verfügung.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Die Java-Bibliothek bietet geeignete Funktionen, um den Wert eines bestimmten Typs in einen String umzuwandeln und umgekehrt

```
String s = String.valueOf(x);           // converts value of x to a string
                                        // type of x can be char, int, long, float, double,
                                        // boolean or char[]

int i = Integer.parseInt("123");       // converts an integer-like char sequence to its
                                        // corresponding integer value

float f = Float.parseFloat("3.14")     // converts a float-like char sequence to its
                                        // corresponding float value

char[] a = s.toCharArray();           // provides char array containing value of string s
```

Information 26: Stringkonversionen

Zur Konversion in einen String stellt der `String`-Typ die Methode `valueOf()` bereit. Aufgrund der angenehmen Eigenschaft, dass der Operator `+` (String-Konkatenation) Werte automatisch in Strings konvertiert, muss diese Methode nicht so häufig genutzt werden.

Die Umwandlung eines Strings in einen Wert eines bestimmten anderen Typs erfolgt gemäß der in Information 26 aufgeführten Methoden. Bei der Umwandlung in die Typen `int` bzw. `float` ist zu beachten, dass der an die Methoden `parseInt()` bzw. `parseFloat()` übergebene String auch tatsächlich eine zulässige "*Integer-Zeichenkette*" bzw. "*Float-Zeichenkette*" beinhaltet. Andernfalls liefert das Java-Laufzeitsystem eine so genannte Ausnahme (*Exception*).

Ausnahmen und deren Behandlung werden in der Kurseinheit FORTGESCHRITTENE PROGRAMMIERKONZEPTE [C&M-FP] behandelt.

Mit den in dieser Kurseinheit eingeführten Anweisungen und Datenstrukturen lassen sich aufbauend auf den in der Kurseinheit PROGRAMMIERGRUNDLAGEN [C&M-PG] beschriebenen Sprachelemente bereits größere imperative Java-Programme entwickeln.

VERZEICHNISSE

Abkürzungen und Glossar

Abkürzung oder Begriff	Langbezeichnung und/oder Begriffserklärung
\perp	<i>Bottom-Element</i> Symbolisiert im Zusammenhang mit Zuständen den gedachten „Endzustand“ nicht-terminierender Programme. Synonymer Begriff: undefiniert-Element
Array	Eine Datenstruktur, durch die Variablen gleichen Typs zu einer Wertemenge zusammengefasst werden können.
BNF	Backus-Naur-Form Schreibweise für Regeln einer Grammatik
EBNF	Erweiterte BNF Erweiterung der BNF um Metaregeln zur ausdrucksstärkeren Formulierung von Regeln einer Grammatik.
ENV	<i>Environment</i> Menge der Werte-Belegung von Identifikatoren, die durch Variablen dargestellt werden.
Hornerschema	Schema zur effizienten Berechnung von Polynomen.
Methode	Aus der objektorientierten Programmierung stammende Bezeichnung für eine Rechenvorschrift, die aus einem Methodennamen und formalen Parametern besteht. Handelt es sich bei der Methode um eine Funktion, so ist zusätzlich der Typ des von der Methode zurückgegebenen Ergebnisses anzugeben. Methoden sind in Java das Mittel, mit dem Botschaften realisiert werden [MS+03].
<i>Nullpointer</i>	Zeiger (<i>Pointer</i>), dessen Wert undefiniert ist, da ihm noch keine Speicheradresse zugewiesen wurde.
O(n)	lineare Ordnung Schreibweise (O-Notation), durch die der Aufwand eines Algorithmus ausgedrückt wird.
Rekursive Methode	Eine Methode, die sich in ihrem Methodenrumpf selbst aufruft.
UML	<i>Unified Modeling Language</i> Sprache, die mittels graphischer Elemente eine semi-formalen Beschreibung von beliebigen Gegenständen (z.B. Software-Systeme oder Geschäftsbereiche) ermöglicht.
WWW, Web	World Wide Web Wichtigste Anwendung des Internet.

Zusicherung	Explizite Aussage über den Programmzustand in Form eines booleschen Ausdrucks über Werte der im Programm genutzten Größen. Englischer Begriff: <i>Assertion</i>
Zuweisung	Anweisung, durch die einer Variablen ein Wert zugeordnet wird. Englischer Begriff: <i>Assignment</i>

Index

⊥	7	Methoden	17
Arrays	20	Nullpointer	22
<i>Environment</i>	7	O(n)	26
Erweiterter Backus-Naur-Form	4	rekursive Methode	25
Garbage Collection	23	Zuweisung	6
Hornerschema	5		

Informationen und Interaktionen

Information 1: IMPERATIVE PROGRAMMIERUNG.....	3
Information 2: VARIABLEN UND ZUWEISUNG- Zentraler Begriff der Anweisung.....	3
Information 3: Anweisungen der Sprache Java im Überblick.....	4
Information 4: Hornerschema als weiteres Beispiel.....	6
Information 5: Funktionale Bedeutung von Anweisungen.....	7
Information 6: Semantik einer Zuweisung.....	8
Information 7: ZUSAMMENGESETZTE ANWEISUNGEN - Syntax und Semantik.....	9
Information 8: Verzweigungen.....	9
Information 9: Mehrwegverzweigung switch-Anweisung.....	10
Information 10: Eigenschaften der beiden Lösungen.....	11
Information 11: Zusicherungen.....	12
Information 12: for-Anweisung.....	13
Information 13: Zusicherung bei Schleifen.....	15
Information 14: METHODEN – Überladen von Methoden.....	17
Information 15: Beispiele für das Überladen von Methoden.....	18
Information 16: Gründe für die Nutzung von Methoden.....	19
Information 17: DATENSTRUKTUREN - Überblick.....	20
Information 18: Arrays.....	20
Information 19: Speicherbereinigung (<i>Garbage Collection</i>).....	22
Information 20: Varianten zur Erstellung von initialisierten Arrays.....	23
Information 21: Binäre Suche.....	25
Information 22: Zeichen-Arrays.....	27
Information 23: Mehrdimensionale Arrays.....	28
Information 24: Strings.....	30
Information 25: Stringmanipulationen.....	32
Information 26: Stringkonversionen.....	33
Interaktion 1: Gebundene Bezeichnungen.....	5
Interaktion 2: Variablen und Zuweisungen.....	6
Interaktion 3: switch-Anweisung und if-Anweisung.....	11
Interaktion 4: do-while-Anweisung.....	13
Interaktion 5: Beispiel zur for-Anweisung.....	14
Interaktion 6: Schleifeninvariante zu einem Beispiel-Programm.....	15

Interaktion 7: Abbruch von Schleifen	16
Interaktion 8: Problemzerlegung mittels Methoden	18
Interaktion 9: Arbeiten mit Arrays	21
Interaktion 10: Array-Zuweisung	22
Interaktion 11: Sequentielle Suche	24
Interaktion 12: Beispielablauf zur binären Suche	26
Interaktion 13: Suche in einem Zeichen-Array	27
Interaktion 14: Matrixmultiplikation	29
Interaktion 15: Stringvariablen	30
Interaktion 16: Stringvergleich und weitere Stringoperationen	31
Interaktion 17: StringBuffer	32

Literatur

- [Br98] Manfred Broy: Informatik – Eine grundlegende Einführung, Band 1: Programmierung und Rechstrukturen, Springer Verlag 1998.
- [C&M-FP] Cooperation&Management: FORTGESCHRITTENE PROGRAMMIERKONZEPTE, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [C&M-FR] Cooperation&Management: FUNKTIONALE PROGRAMMIERKONZEPTE UND REKURSION, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [C&M-PG] Cooperation&Management: PROGRAMMIERGRUNDLAGEN, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [Mö03] Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.
- [MS+03] Stefan Middendorf, Reiner Singer, Jörn Heid: Java – Programmierhandbuch für die Java-2-Plattform, Standard Edition, dpunkt.verlag, 2003.