LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Skript zur Vorlesung:

Einführung in die Programmierung

WiSe 2009 / 2010

Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Gliederung

4. Imperative Programmierung

4.1 Grunddatentypen und Ausdrücke

4.2 Imperative Variablenbehandlung

4.3 Anweisungen, Blöcke und Gültigkeitsbereiche

4.4 Klassenvariablen

4.5 Reihungen

4.6 (Statische) Methoden

4.7 Kontrollstrukturen

4.8 … Putting the pieces together …

Vorbemerkung

- Induktion, Rekursion und das Konzept der (induktiv definierten) Ausdrücke sind funktionale Konzepte und werden als solche in der Einführungsvorlesung “Programmierung und Modellierung” vertieft.
- Allerdings spielen diese (und andere funktionale) Konzepte auch im imperativen Programmierparadigma eine wichtige Rolle.
- Auch als Entwurfsmuster spielt Rekursion in imperativen Sprachen eine wichtige Rolle.
- Wir wenden uns nun aber dem imperativen Programmier-paradigma und der Realisierung am Beispiel von Java zu.

Grunddatentypen

- Typischerweise stellt jede höhere Programmiersprache gewisse *Grunddatentypen* zur Verfügung. (Rein konzeptionell handelt es sich dabei um eine Menge von Sorten).
- Zusätzlich werden auch gewisse *Grundoperationen* bereitgestellt, also eine Menge von (teilweise überladenen) Operationssymbolen.
- Die Semantik dieser Operationen ist durch den zugrundeliegenden Algorithmus zur Berechnung der entsprechenden Funktion definiert.
- Aus den Literalen der Grunddatentypen und den zugehörigen Grundoperationen können nun, wie im vorherigen Abschnitt definiert, Ausdrücke gebildet werden.
- **Achtung**: Diese Ausdrücke enthalten zunächst noch keine Variablen. Zu Variablen kommen wir später.

Grunddatentypen (primitive Typen) in Java

- Wir wissen bereits, dass es in Java Grunddatentypen (auch *atomare* oder *primitive Typen*) für $\mathbb{B}$, *CHAR*, eine Teilmenge von $\mathbb{Z}$ und für eine Teilmenge von $\mathbb{R}$ gibt, aber keinen eigenen Grunddatentyp für $\mathbb{N}$.
- Die Werte der primitiven Typen werden intern binär dargestellt.
- Die Datentypen unterscheiden sich u.a. in der Anzahl der Bits, die für ihre Darstellung verwendet werden.
- Die Anzahl der Bits, die für die Darstellung der Werte eines primitiven Datentyps verwendet werden, heißt *Länge* des Typs.
- Die Länge eines Datentyps hat Einfluss auf seinen Wertebereich.
- Als *objektorientierte* Sprache bietet Java zusätzlich die Möglichkeit, benutzerdefinierte (sog. *abstrakte*) Datentypen zu definieren. Diese Möglichkeiten lernen wir im Teil über objektorientierte Modellierung genauer kennen.

Grunddatentypen (primitive Typen) in Java

Überblick

Typname	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`,`false`}
`char`	2 Byte	Alle Unicode-Zeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

Wahrheitswerte in Java

- Java hat einen eigenen Typ **`boolean`** für Wahrheitswerte.
- Die beiden einzigen Werte (*Konstanten*) sind **`true`** für “wahr” und **`false`** für “falsch”.

Boole'sche Operatoren in Java

Operator	Bezeichnung	Bedeutung
`!`	logisches NICHT (¬)	`!a` ergibt **`false`** wenn `a` wahr ist, sonst **`true`**.
`&`	logisches UND (∧)	`a & b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`&&`	sequentielles UND	`a && b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Ist `a` bereits falsch, wird **`false`** zurückgegeben und `b` nicht mehr ausgewertet.
`|`	logisches ODER (∨)	`a | b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`||`	sequentielles ODER	`a || b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Ist bereits `a` wahr, so wird **`true`** zurückgegeben und `b` nicht mehr ausgewertet.
`^`	exklusives ODER (XOR)	`a ^ b` ergibt **`true`**, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wahrheitswert haben.

Zeichen (Character) in Java

- Java hat einen eigenen Typ **char** für (Unicode-)Zeichen.
- Werte (*Konstanten*) werden in einfachen Hochkommata gesetzt, z.B. `'A'` für das Zeichen "A".
- Einige Sonderzeichen können mit Hilfe von *Standard-Escape-Sequenzen* dargestellt werden:

Sequenz	Bedeutung
`\b`	Backspace (Rückschritt)
`\t`	Tabulator (horizontal)
`\n`	Newline (Zeilenumbruch)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	doppeltes Anführungszeichen
`\'`	einfaches Anführungszeichen
`\\`	Backslash

Ganze Zahlen in Java

- Java hat vier Datentypen für ganze (vorzeichenbehaftete) Zahlen:
  - **`byte`** (Länge: 8 Bit)
  - **`short`** (Länge: 16 Bit)
  - **`int`** (Länge: 32 Bit)
  - **`long`** (Länge: 64 Bit)
- Werte (*Konstanten*) können geschrieben werden in
  - Dezimalform: bestehen aus den Ziffern `0`, . . ., `9`
  - Oktalform: beginnen mit dem Präfix `0` und bestehen aus Ziffern `0`, . . ., `7`
  - Hexadezimalform: beginnen mit dem Präfix `0x` und bestehen aus Ziffern `0`, . . ., `9` und den Buchstaben `a`, . . ., `f` (bzw. `A`, . . ., `F`)
- Negative Zahlen erhalten ein vorangestelltes -.
- **Gehört dieses - zum Literal?**

Vorzeichen-Operatoren in Java

- Vorzeichen-Operatoren haben die Signatur

$$S \rightarrow S$$

  mit $S \in \{$**`byte`**, **`short`**, **`int`** $\ldots\}$.
- Operationen:

Operator	Bezeichnung	Bedeutung
+	Positives Vorzeichen	+n ist gleichbedeutend mit n
-	Negatives Vorzeichen	-n kehrt das Vorzeichen von n um

- **Bemerkung**: Offenbar sind die Operatoren überladen!!!

Gleitkommazahlen in Java

- Java hat zwei Datentypen für Fließkommazahlen:
  - **`float`** (Länge: 32 Bit)
  - **`double`** (Länge: 64 Bit)
- Werte (*Konstanten*) werden immer in Dezimalnotation geschrieben und bestehen maximal aus
  - Vorkommateil
  - Dezimalpunkt (*)
  - Nachkommateil
  - Exponent `e` oder `E` (Präfix – möglich) (*)
- Negative Zahlen erhalten ein vorangestelltes -.
- Beispiele:
  - double: `6.23, 623E-2, 62.3e-1`
  - float: `6.23f, 623E-2F, 62.3e-1f`

(*) mindestens einer dieser Bestandteile muss vorhanden sein.

Arithmetische Operationen in Java

- Arithmetische Operationen haben die Signatur

$$S \times S \rightarrow S$$

mit $S \in \{$**`byte`**, **`short`**, **`int`** $\ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
`+`	Summe	`a+b` ergibt die Summe von `a` und `b`
`-`	Differenz	`a-b` ergibt die Differenz von `a` und `b`
`*`	Produkt	`a*b` ergibt das Produkt aus `a` und `b`
`/`	Quotient	`a/b` ergibt den Quotienten von `a` und `b`
`%`	Modulo	`a%b` ergibt den Rest der ganzzahligen Division von a durch b

- **Bemerkung**: Offenbar sind die Operatoren überladen!!!

Vergleichsoperationen in Java

- Vergleichsoperatoren haben die Signatur

$$S \times S \rightarrow \texttt{boolean}$$

  mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int} \ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
`==`	Gleich	`a==b` ergibt **`true`**, wenn `a` gleich `b` ist
`!=`	Ungleich	`a!=b` ergibt **`true`**, wenn `a` ungleich `b` ist
`<`	Kleiner	`a<b` ergibt **`true`**, wenn `a` kleiner `b` ist
`<=`	Kleiner gleich	`a<=b` ergibt **`true`**, wenn `a` kleiner oder gleich `b` ist
`>`	Größer	`a>b` ergibt **`true`**, wenn `a` größer `b` ist
`>=`	Größer gleich	`a>=b` ergibt **`true`**, wenn `a` größer oder gleich `b` ist

- **Bemerkung**: Offenbar sind die Operatoren ebenfalls überladen!!!

Ausdrücke in Java

- Wir können auch in Java aus Operatoren Ausdrücke (zunächst ohne Variablen) bilden, so wie im vorherigen Kapitel besprochen.
- Laut induktiver Definition von Ausdrücken ist eine Konstante ein Ausdruck.
- Interessanter ist, aus Konstanten (z.B. den **`int`** Werten `6` und `8`) und mehrstelligen Operatoren (z.B. `+`, `*`, `<`, `&&`) Ausdrücke zu bilden. Ein gültiger Ausdruck hat selbst wieder einenWert (der über die Semantik der beteiligten Operationen definiert ist) und einen Typ (der durch die Ergebnissorte des angewendeten Operators definiert ist):
  - `6 + 8 //Wert: 14 vom Typ int`
  - `6 * 8 //Wert: 48 vom Typ int`
  - `6 < 8 //Wert: true vom Typ boolean`
  - `6 && 8 //ungültiger Ausdruck` **Warum?**

Typkonversion

- Motivation:
  - Was passiert eigentlich, wenn man verschiedene Sorten / Typen miteinander verarbeiten will?
  - Ist der Java-Ausdruck `6 + 7.3` erlaubt?
  - Eigentlich nicht: Die Operation + ist zwar überladen, d.h.

$$+ : S \times S \to S$$

    ist definiert für beliebige primitive Datentypen S, aber es gibt keine Operation

$$+ : S \times T \to U$$

    für *verschiedene* primitive Datentypen $S \neq T$.
- Trotzdem ist der Ausdruck `6 + 7.3` in Java erlaubt.
- *Wann* passiert *was* und *wie* bei der Auswertung des Ausdrucks `6 + 7.3`?

Automatische Typkonversion in Java

- **Wann**:
  Während des Übersetzens des Programmcodes durch den Compiler.
- **Was**:
  Der Compiler kann dem Ausdruck keinen Typ (und damit auch keinen Wert) zuweisen. Solange kein *Informationsverlust* auftritt, versucht der Compiler diese Situation zu retten.
- **Wie**:
  Der Compiler konvertiert automatisch den Ausdruck `6` vom Typ **`int`** in einen Ausdruck vom Typ **`double`**, so dass die Operation

$$+ : \texttt{double} \times \texttt{double} \rightarrow \texttt{double}$$

  angewendet werden kann.

Automatische Typkonversion in Java

- Formal gesehen ist diese Konvertierung eine Operation `i->d` mit der Signatur

$$\texttt{i->d} : \textbf{int} \rightarrow \textbf{double}$$

  d.h. der Compiler wandelt den Ausdruck
  `6 + 7.3`
  um in den Ausdruck
  `i->d(6) + 7.3`

- Dieser Ausdruck hat offensichtlich einen eindeutigen Typ und damit auch einen eindeutig definierten Wert.

Typkonversion in Java

- Was bedeutet “Informationsverlust”?
- Es gilt folgende “Kleiner-Beziehung” zwischen Datentypen:

`byte` < `short` < `int` < `long` < `float` < `double`

- Beispiele:
  - `1 + 1.7` ist vom Typ **`double`**
  - `1.0f + 2` ist vom Typ **`float`**
  - `1.0f + 2.0` ist vom Typ **`double`**
- Java konvertiert Ausdrücke automatisch in den allgemeineren (“größeren”) Typ, da dabei kein Informationsverlust auftritt. **Warum?**

Explizite Typkonversion in Java: Type-Casting

- Will man eine Typkonversion zum spezielleren Typ durchführen, so muss man dies in Java explizit angeben.
- Dies nennt man allgemein *Type-Casting*.
- In Java erzwingt man die Typkonversion zum spezielleren Typ `type` durch Voranstellen von (`type`).
- Der Ausdruck
  `(type) a`
  wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um.
- Beispiele:
  - `(`**`byte`**`) 3` ist vom Typ **`byte`**
  - `(`**`int`**`) (2.0 + 5.0)` ist vom Typ **`int`**
  - `(`**`float`**`) 1.3e-7` ist vom Typ **`float`**

Explizite Typkonversion in Java: Type-Casting

- Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
- Beispiele:
  - `(int) 5.2` ergibt `5`
  - `(int) -5.2` ergibt `-5`

Der Type-Cast-Operator in Java

Im Ausdruck
`(type) a`
ist `(type)` ein Operator. Type-Cast-Operatoren bilden zusammen mit einem Ausdruck wieder einen Ausdruck.
Der Typ des Operators ist z.B.:

```
(int)   : char∪byte∪short∪int∪long∪float∪double → int
(float) : char∪byte∪short∪int∪long∪float∪double → float
```

Sie können also z.B. auch **`char`** in **`int`** umwandeln.
Klingt komisch? Ist aber so! Und was passiert da?

Das Konzept Typkonversion im Allgemeinen

- Ganz allgemein nennt man das Konzept der Umwandlung eines Ausdrucks mit einem bestimmten Typ in einen Ausdruck mit einem anderen Typ *Typkonversion*.
- In vielen Programmiersprachen gibt es eine automatische Typkonversion meist vom spezielleren in den allgemeineren Typ.
- Eine Typkonversion vom allgemeineren in den spezielleren Typ muss (wenn erlaubt) sinnvollerweise immer explizit durch einen *Typecasting*-Operator herbeigeführt werden.
- Es gibt auch Programmiersprachen, in denen man grundsätzlich ein entsprechendes Typecasting explizit durchführen muss.
- Unabhängig davon kennen Sie jetzt das allgemeine Konzept und die Problematik solch einer Typkonversion. Unterschätzen Sie diese nicht als Fehlerquelle bei der Entwicklung von Algorithmen bzw. der Erstellung von Programmen!

Ausdrücke mit Variablen

- Im vorherigen Kapitel haben wir Ausdrücke (in Java) nur mit Operationssymbolen (Konstanten und mehrstelligen Operatoren) gebildet.
- Warum haben wir dann die Ausdrücke vorher induktiv so definiert, dass darin auch Variablen vorkommen können?
- Antwort: Weil wir natürlich auch Variablen in (Java-)Ausdrücken zulassen wollen.

Konstanten und Variablen

- Wozu sind Variablen gut?
- Betrachten wir als Beispiel folgenden (in funktionaler Darstellung) angegebenen Algorithmus:
  - Berechne die Funktion $f(x)$ für $x \neq -1$ mit $f: \mathbb{R} \rightarrow \mathbb{R}$ gegeben durch

$$f(x) = \left(x + 1 + \frac{1}{x+1}\right)^2 \quad \text{für } x \neq -1$$

  - Eine imperative Darstellung erhält man durch Aufteilung der Funktionsdefinition in:

$$y_1 = x + 1$$
$$y_2 = y_1 + \frac{1}{y_1}$$
$$y_3 = y_2 * y_2$$
$$f(x) = y_3.$$

Konstanten und Variablen

- Intuition des Auswertungsvorgangs der imperativen Darstellung:
  - y_1, y_2 *und* y_3 repräsentieren drei Zettel.
  - Auf diese Zettel werden der Reihe nach Rechenergebnisse geschrieben.
  - Bei Bedarf wird der Wert auf dem Zettel "abgelesen".
- Formal steckt hinter dieser Intuition eine Substitution.
  - x wird durch den Eingabewert substituiert.
  - y_1 wird mit dem Wert des Ausdrucks $x + 1$ substituiert, wobei x bereits substituiert wurde.
  - Mit y_2 und y_3 kann man analog verfahren.
- y_1, y_2 und y_3 heißen *Konstanten*.

Konstanten und Variablen

- Bei genauerer Betrachtung:
  - Nachdem der Wert von y_1 zur Berechnung von y_2 benutzt wurde, wird er im folgenden nicht mehr benötigt.
  - Eigentlich könnte der Zettel nach dem Verwenden (Ablesen) “radiert” und für die weiteren Schritte wiederverwendet werden.
  - In diesem Fall kämen wir mit einem Zettel y aus.
- y heißt im Kontext imperativer Programmierung *Variable*.
- Im Unterschied zu Konstanten sind Variablen “radierbar”.

Konstanten und Variablen

- Die imperative Lösung zur Berechnung von $f(x)$ mit einer Variable y:

$$y = x + 1$$
$$y = y + \frac{1}{y}$$
$$y = y * y$$

- Wenn wir im Folgenden von Konstanten und Variablen sprechen, verwenden wir immer die imperativen Konzepte (außer wir weisen explizit darauf hin), d.h. wir sprechen dann immer von Variablen ("Platzhalter") der Menge V, über die wir Ausdrücke bilden können.
- Variablen sind überschreibbare, Konstanten sind nicht überschreibbare Platzhalter.

Konstanten und Variablen

- Variablen und Konstanten können
  - *deklariert* werden, d.h. ein "leerer Zettel" (Speicherzelle) wird angelegt.
    Formal: Der Bezeichner der Menge der zur Verfügung stehenden Variablen V, die wir in Ausdrücken verwenden dürfen, wird hinzugefügt.
  - *initialisiert* werden, d.h. ein Wert wird auf den "Zettel" (in die Speicherzelle) geschrieben.
    Formal: Die Variable ist mit einem entsprechenden Wert belegt.
- Der Wert einer Variablen kann später auch noch durch eine (neue) *Wertzuweisung* verändert werden.
- Die Initialisierung entspricht einer (initialen)Wertzuweisung.
- Nach der Deklaration kann der Wert einer *Konstanten* nur noch einmalig durch eine Wertzuweisung verändert / initialisiert werden.
- Nach der Deklaration kann der Wert einer *Variablen* beliebig oft durch eine Wertzuweisung verändert werden.

Konstanten und Variablen in Java

- Eine Variablendeklaration hat in Java die Gestalt

  ```
  Typname variablenName;
  ```

  Konvention: Variablennamen beginnen mit kleinen Buchstaben.
- Eine Konstantendeklaration hat in Java die Gestalt

  ```
  final Typname KONSTANTENNAME;
  ```

  Konvention: Konstantennamen bestehen komplett aus großen Buchstaben.
- Das bedeutet: in Java hat jede Variable / Konstante einen Typ.
- Eine Deklaration ist also das Bereitstellen eines "Platzhalters" des entsprechenden Typs.

Konstanten und Variablen in Java

- Eine Wertzuweisung (z.B. Initialisierung) hat die Gestalt

```
variablenName = NeuerWert;
```

bzw.

```
KONSTANTENNAME = Wert;
```
(nur als Initialisierung)

- Eine Variablen- bzw. Konstantendeklaration kann auch mit der Initialisierung verbunden sein, d.h. der ersten Wertzuweisung.

```
Typname variablenName = InitialerWert;
```

(Konstantendeklaration mit Initialisierung analog mit Zusatz **`final`**)

- Formal gesehen ist eine Wertzuweisung eine Funktion

$$=: S \to S$$

mit beliebigem Typ *S*.

- Damit ist eine Wertzuweisung auch wieder ein Ausdruck.

Abkürzungen für Wertzuweisungen in Java

Für bestimmte einfache Operationen (Addition und Subtraktion mit 1 als zweitem Operanden) gibt es gängige Kurznotationen:

Operator	Bezeichnung	Bedeutung
`++`	Präinkrement	`++a` ergibt `a+1` und erhöht `a` um `1`
`++`	Postinkrement	`a++` ergibt `a` und erhöht `a` um `1`
`--`	Prädekrement	`--a` ergibt `a-1` und verringert `a` um `1`
`--`	Postdekrement	`a--` ergibt `a` und verringert `a` um `1`

Abkürzungen für Wertzuweisungen in Java

Wenn man eine Variable nicht nur um 1 erhöhen oder verringern, sondern allgemein einen neuen Wert zuweisen will, der aber vom alten Wert abhängig ist, gibt es Kurznotationen wie folgt:

Operator	Bezeichnung	Bedeutung
`+=`	Summe	`a+=b` weist `a` die Summe von `a` und `b` zu
`-=`	Differenz	`a-=b` weist `a` die Differenz von `a` und `b` zu
`*=`	Produkt	`a*=b` weist `a` das Produkt aus `a` und `b` zu
`/=`	Quotient	`a/=b` weist `a` den Quotienten von `a` und `b` zu
(Auch für weitere Operatoren möglich...)		

Konstanten und Variablen in Java

- Konstanten:

```
final <typ> <NAME> = <ausdruck>;
```

```
final double Y_1 = x + 1; //Achtung: was ist x???
final double Y_2 = Y_1 + 1 / Y_1;
final double Y_3 = Y_2 * Y_2;
final char NEWLINE = '\n';
final double BESTEHENSGRENZE_PROZENT = 0.5;
final int GESAMTPUNKTZAHL = 80;
```

- Variablen:

```
<typ> <name> = <ausdruck>;
```

```
double y = x + 1; //Achtung: was ist x???
int klausurPunkte = 42;
boolean klausurBestanden =
        ((double) klausurPunkte) /
        GESAMTPUNKTZAHL >= BESTEHENSGRENZE_PROZENT;
```

Zustandsänderungen von Variablen

Beispiel:

Anweisung	x	y	z
`int x;`			
`int y = 5;`		5	
`x = 0;`	0	5	
`final int z = 3;`	0	5	| 3 |
`x += 7 + y;`	12	5	| 3 |
`y = y + z - 2;`	12	6	| 3 |
`x = x * y;`	72	6	| 3 |
`x = x/z;`	24	6	| 3 |

Legende: [ ] = radierbar [\| \|] = nicht radierbar

Anweisungen

- In imperativen Programmen sind *Anweisungen* die elementaren Einheiten.
- Eine Anweisung steht für einen einzelnen Abarbeitungsschritt in einem Algorithmus.
- Anweisungen können unter anderem aus Ausdrücken gebildet werden.
- Im folgenden: Anweisungen in Java.
- Eine Anweisung wird immer durch das Semikolon begrenzt.

Die leere Anweisung

- Die einfachste Anweisung ist die leere Anweisung:
  ```
  ;
  ```
- Die leere Anweisung hat keinen Effekt auf das laufende Programm, sie bewirkt nichts.
- Oftmals benötigt man tatsächlich eine leere Anweisung, wenn von der Logik des Algorithmus *nichts* zu tun ist, die Programmsyntax aber eine Anweisung erfordert.

Der Block

- Ein Block wird gebildet von einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer, die eine beliebige Menge von Anweisungen umschließen:

```
{
 Anweisung1;
 Anweisung2;
 ...
}
```

- Die Anweisungen im Block werden nacheinander ausgeführt.
- Der Block als ganzes gilt als eine einzige Anweisung, kann also überall da stehen, wo syntaktisch eine einzige Anweisung verlangt ist.
- Eine Anweisung in einem Block kann natürlich auch wieder ein Block sein.

Lebensdauer, Gültigkeit, Sichtbarkeit

- Die *Lebensdauer* einer Variablen ist die Zeitspanne, in der die virtuelle Maschine der Variablen einen *Speicherplatz* zu Verfügung stellt.
- Die *Gültigkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen der Name der Variablen dem Compiler durch eine Vereinbarung (*Deklaration*) bekannt ist.
- Die *Sichtbarkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf ihren Wert zugreifen kann.

Gültigkeitsbereich von Variablen

- Eine in einem Block deklarierte (lokale) Variable ist ab ihrer Deklaration bis zum Ende des Blocks gültig und sichtbar.
- Mit Verlassen des Blocks, in dem eine lokale Variable deklariert wurde, endet auch ihre Gültigkeit und Sichtbarkeit.
- Damit oder kurz danach endet normalerweise auch die Lebensdauer der Variablen, da der Speicherplatz, auf den die Variable verwiesen hat, im Prinzip wieder freigegeben ist und für neue Variablen verwendet werden kann.
- Solange eine lokale Variable sichtbar ist, darf keine neue lokale Variable gleichen Namens angelegt werden.

Gültigkeitsbereich von Variablen

- Beispiel:

```
...
int i = 0;
{
int i = 1; // nicht erlaubt
i = 1;     // erlaubt
int j = 0;
}
j = 1;     // nicht moeglich
...
```

Ausdrucksanweisung

Aus einem Ausdruck `<ausdruck>` wird durch ein angefügtes Semikolon eine Anweisung. Allerdings spielt dabei der Wert des Ausdrucks im weiteren keine Rolle. Daher ist eine solche Ausdrucksanweisung auch nur sinnvoll (und in Java nur dann erlaubt), wenn der Ausdruck einen Nebeneffekt hat.
Solche Ausdrücke sind:

- Wertzuweisung
- (Prä- / Post-)Inkrement
- (Prä- / Post-)Dekrement
- Methodenaufruf (werden wir später kennenlernen)
- Instanzerzeugung (werden wir später kennenlernen)

Global-lokal

- Mit den bisherigen Konzepten können wir theoretisch einfache imperative Algorithmen und Programme schreiben.
- Wir werden später weitere Konzepte kennenlernen, um komplexere Algorithmen / Programme zu strukturieren:
  - Prozeduren (in Java statische Methoden genannt).
    Beispiel: die Prozedur (Methode) `main`. Ein Algorithmus ist in einer Prozedur (Methode) verpackt.
  - Module (in Java Pakete bzw. Packages genannt)
- Variablen, so wie wir sie bisher kennengelernt haben, sind *lokale* Variablen und Konstanten, d.h. sie sind nur innerhalb des Blocks (Algorithmus, Prozedur, Methode), der sie verwendet, bekannt.
- Es gibt auch *globale* Variablen und Konstanten, die in mehreren Algorithmen (Prozeduren, Methoden, Modulen) bekannt sind.
- Diese globalen Größen sind z.B. für den Datenaustausch zwischen verschiedenen Algorithmen geeignet.

Globale und lokale Größen in Java

Globale Variablen heißen in Java *Klassenvariablen*. Diese Variablen gelten in der gesamten Klasse und ggf. auch darüberhinaus. Eine Klassenvariable definiert man üblicherweise am Beginn einer Klasse, z.B.:

```
public class HelloWorld
{
  public static String gruss = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(gruss);
  }
}
```

Die Definition wird von den Schlüsselwörtern **`public`** und **`static`** eingeleitet. Deren Bedeutung lernen wir später kennen.

Konstante Klassenvariablen

Klassenvariablen kann man auch als Konstanten definieren.
Wie bei lokalen Variablen dient hierzu das Schlüsselwort **`final`**:

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

Auch bei Klassenvariablen schreibt man Namen von Konstanten in Großbuchstaben.

Klasse und Programm

- Zur Erinnerung: Ein Java-Programm besteht aus Klassen – einer oder mehreren, in größeren Projekten oft hunderten oder tausenden.
- Da eine Klasse auch einen Block bildet, ist der Gültigkeitsbereich einer Klassenvariablen klar: Sie gilt im gesamten Programmcode innerhalb der Klasse nach ihrer Deklaration.
- Darüberhinaus gilt sie aber in der gesamten Programmlaufzeit, d.h. solange das Programm ausgeführt wird, "lebt" eine Klassenvariable.
- Die Sichtbarkeit in anderen als ihrer eigenen Klasse kann man aber einschränken.
- Eine Klasse gehört immer zu einem Package. Die Klassendatei liegt in einem Verzeichnis, das genauso heißt wie das Package.
- Der Package-Name gehört zum Klassennamen.

Nutzen von Klassenvariablen

Als Klassenvariablen definiert man z.B. gerne Werte, die von universellem Nutzen sind. Die Klasse `java.lang.Math` definiert die mathematischen Konstanten *e* und π:

```
package java.lang;

public final class Math {
   ...
   /**
     * The <code>double</code> value that is closer than any other to
     * <i>e</i>, the base of the natural logarithms.
     */
   public static final double E = 2.7182818284590452354;

   /**
     * The <code>double</code> value that is closer than any other to
     * <i>pi</i>, the ratio of the circumference of a circle to its
     * diameter.
     */
   public static final double PI = 3.14159265358979323846;
   ...
}
```

Automatische Initialisierung

Im Gegensatz zu lokalen Variablen muss man Klassenvariablen nicht explizit initialisieren. Sie werden dann automatisch mit ihren Standardwerten initialisiert:

Typname	Standardwert
`boolean`	`false`
`char`	`\u0000`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0`
`double`	`0.0`

Klassenkonstanten müssen dagegen explizit initialisiert werden.

Namen von lokalen Variablen und Klassenvariablen

- Lokale Variablen innerhalb einer Klasse können genauso heißen wie eine Klassenvariable.

```
public class Sichtbarkeit
{
    public static int variablenname;

    public static void main(String[] args)
    {
        boolean variablenname = true;
    }
}
```

- Das bedeutet: Während bei lokalen Variablen *Sichtbarkeit* und *Gültigkeit* zusammenfallen, muss man zwischen beiden Eigenschaften bei Klassenvariablen prinzipiell unterscheiden.

Namen von lokalen Variablen und Klassenvariablen

- Das ist aber kein Widerspruch zum Verbot, den gleichen Namen innerhalb des Gültigkeitsbereichs einer Variable nochmal zu verwenden, denn genau genommen heißt die Klassenvariable doch anders:
- Zu ihrem Namen gehört der vollständige Klassenname (inklusive des Package-Namens).
  Der vollständige Name der Konstanten `PI` aus der `Math`-Klasse ist also:
  `java.lang.Math.PI`
- Unter dem vollständigen Namen ist eine globale Variable auch dann sichtbar, wenn der innerhalb der Klasse geltende Name (z.B. `PI`) durch den identisch gewählten Namen einer lokalen Variable verdeckt ist.

Namen von lokalen Variablen und Klassenvariablen

```
public class Sichtbarkeit
{
    public static int variablenname;

    public static void main(String[] args)
    {
        boolean variablenname = true;
        System.out.println(variablenname); // Ausgabe: true
        System.out.println(Sichtbarkeit.variablenname); // Ausgabe?
    }
}
```

Eingebaute Datenstruktur

- In einer Programmiersprache ist üblicherweise eine einfache Datenstruktur eingebaut, die es ermöglicht, eine Reihe von Werten (gleichen Typs) zu modellieren.
- Im imperativen Paradigma ist das normalerweise das Array (Reihung, Feld).

Vorteil: direkter Zugriff auf jedes Element möglich

Nachteil: dynamisches Wachstum ist nicht möglich
(In Java sind Arrays *semidynamisch*, d.h., ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden.)

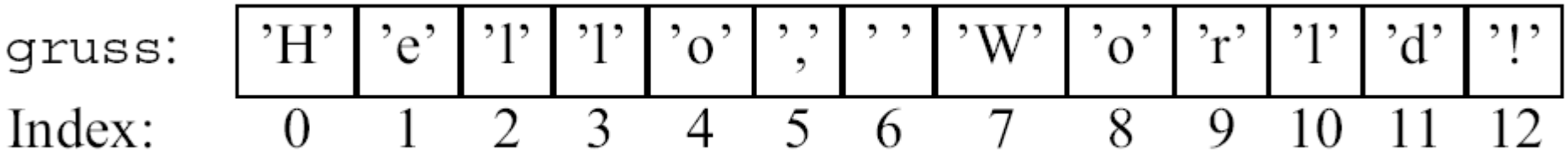
Reihung

Definition

Eine Reihung (Feld, Array) ist ein Tupel von Komponentengliedern gleichen Typs, auf die über einen Index direkt zugegriffen werden kann.

Darstellung

Eine **`char`**-Reihung `gruss` der Länge 13:



Mathematische Darstellung

Allgemein

Eine Reihung mit n Komponenten vom Typ `<typ>` ist eine Abbildung von der Indexmenge I_n auf die Menge `<typ>`.

Beispiel

$$\texttt{gruss} \;:\; \{0,1,\ldots,12\} \rightarrow \textbf{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ \vdots & \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

Deklaration von Arraytypen und Arrayvariablen in Java

- Der Typ eines Arrays, das den Typ

  `<typ>`

  enthält, ist in Java:

  `<typ>[] variablenName`.

- Der Wert des Ausdrucks

  `variablenName[i]`

  ist der Wert des Arrays an der Stelle `i`.

Deklaration von Arraytypen und Arrayvariablen in Java

Ein Array wird zunächst behandelt wie Variablen von primitiven Typen auch. Wir können also z.B. deklarieren:

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc[0]);      // gibt den Character 'a'
                               // aus, den Wert des Array-
                               // Feldes mit Index 0.
                               // Allgemein: array[i] ist
                               // Zugriff auf das i-te
                               // Element
char d = 'd';
char e = 'e';
char[] de = {d, e};
abc = de; // die Variable abc hat jetzt den gleichen
          // Wert wie die Variable de

System.out.print(abc[0]); // Ausgabe ?
```

Länge einer Reihung

Ein Array hat immer eine feste Länge, die in einer Variablen `length` festgehalten wird. Diese Variable `length` wird mit einem Array automatisch miterzeugt. Der Name der Variablen ist zusammengesetzt aus dem Namen des Arrays und dem Namen `length`:

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc.length); // gibt 3 aus
char[] de = {d, e};
abc = de; // gleicher Variablenname,
          // haelt aber als Wert ein
          // anderes Array
System.out.print(abc.length); // Ausgabe ?
```

Reihungen dynamisch anlegen

- Oft legt man ein Array an, bevor man die einzelnen Elemente kennt. Die Länge muss man dabei angeben:

```
char[] abc = new char[3];
```

- Dass Arrays in Java *semidynamisch* sind, bedeutet: Es ist möglich, die Länge erst zur Laufzeit festzulegen.

```
// x ist eine Variable vom Typ int
// deren Wert bei der Ausfuehrung
// feststeht, aber nicht beim
// Festlegen des Programmcodes
char[] abc = new char[x];
```

- Dann kann man das Array im weiteren Programmverlauf füllen:

```
abc[0] = 'a';
abc[1] = 'b';
abc[2] = 'c';
```

Leere Reihungen?

- Wenn man ein Array anlegt:

```
int[] zahlen = new int[10];
```

  aber nicht füllt – ist es dann leer?
- Es gibt in Java keine leeren Arrays. Ein Array wird immer mit den Standardwerten des jeweiligen Typs befüllt.
- Das spätere Belegen einzelner Array-Zellen ist also immer eine Änderung eines Wertes.

```
int[] zahlen = new int[10];
System.out.print(zahlen[3]); // gibt 0 aus
zahlen[3] = 4;
System.out.print(zahlen[3]); // gibt 4 aus
```

Array-Variablen als Konstanten

Auch Array-Variablen kann man als Konstanten deklarieren. Dann kann man der Variablen keinen neuen Wert zuweisen:

```
final char[] ABC = {'a', 'b', 'c'};
final char[] DE = {'d', 'e'};
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

Aber einzelne Array-Komponenten sind normale Variablen. Man kann ihnen also einen neuen Wert zuweisen. Die Länge des Arrays ändert sich dadurch nicht:

```
ABC[0] = 'd';
ABC[1] = 'e'; // erlaubt
System.out.print(ABC.length); // gibt 3 aus
```

Array als Funktion

Rufen wir uns die abstrakte Betrachtungsweise eines Arrays als Funktion in Erinnerung:

```
final char[] ABC = {'a', 'b', 'c'};
```

$$\texttt{ABC} : \{0,1,2\} \rightarrow \textbf{char}$$

$$i \mapsto \begin{cases} \text{'a'} & \text{falls} \quad i = 0 \\ \text{'b'} & \text{falls} \quad i = 1 \\ \text{'c'} & \text{falls} \quad i = 2 \end{cases}$$

Was bedeutet die Neubelegung einzelner Array-Zellen?

```
ABC[0] = 'd';
ABC[1] = 'e';
```

Der Typ `String` als spezielles Array

- In der Deklaration und Initialisierung

```
public static String gruss = "Hello, World!";
```

  entspricht der Ausdruck "Hello, World!" einer speziellen Schreibweise für ein konstantes Array **`char`**`[13]`, das in einen Typ `String` gekapselt wird.

- **Achtung**: Die Komponenten dieses Arrays können nicht mehr (durch Neuzuweisung) geändert werden.
- Der Typ **`String`** ist kein *primitiver* Typ, sondern eine Klasse von Objekten.
- Werte dieses Typs können aber – wie bei primitiven Typen – durch Literale gebildet werden.
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (abermals überladenen!) Operator + konkateniert werden.

Mehrdimensionale Reihungen

Da auch Arrays einen bestimmten Typ haben
z.B. `gruss : char[]`
kann man auch Reihungen von Reihungen bilden. Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren:

```
int[] m0 = {1, 2, 3};
int[] m1 = {4, 5, 6};
int[][] m = {m0, m1};
```

Prozeduren

- Das Konzept der Prozedur dient zur Abstraktion von Algorithmen (oder von einzelnen Schritten eines Algorithmus).
- Durch Parametrisierung wird von der Identität der Daten abstrahiert: die Berechnungsvorschriften werden mit abstrakten Parametern formuliert
  - konkrete Eingabedaten bilden die aktuellen (Parameter-) Werte.
- Durch Spezifikation des (Ein- / Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert: Vorteile sind
  - **Örtliche Eingrenzung (Locality)**: Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden, ohne die Implementierungen anderer Abstraktionen kennen zu müssen.
  - **Änderbarkeit (Modifiability)**: Jede Abstraktion kann reimplementiert werden, ohne dass andere Abstraktionen geändert werden müssen.
  - **Wiederverwendbarkeit (Reusability)**: Die Implementierung einer Abstraktion kann beliebig wiederverwendet werden.

Funktionen und Prozeduren

- Im funktionalen Programmierparadigma werden Algorithmen als Funktionen dargestellt.
- Das imperative Pendant dazu ist die Prozedur, die sogar ein etwas allgemeineres Konzept darstellt: Eine Funktion kann man als Prozedur bezeichnen, aber nicht jede Prozedur ist eine Funktion.
- Eine Funktion stellt nur eine Abbildung von Elementen aus dem Definitionsbereich auf Elemente aus dem Bildbereich dar.
- Es werden aber keine Werte verändert.
- Im imperativen Paradigma können Werte von Variablen verändert werden (durch Anweisungen). Dies kann Effekte auf andere Bereiche eines Programmes haben.
- Treten in einer Prozedur solche *Seiteneffekte* (oder Nebeneffekte) auf, kann man nicht mehr von einer Funktion sprechen.
- Eine Funktion ist also als eine Prozedur ohne Seiteneffekte.

Funktionen und Prozeduren

- Wir haben bereits Prozeduren mit Seiteneffekten verwendet:

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

Funktionen und Prozeduren

- Bei einer Funktion ist der Bildbereich eine wichtige Information:

$$f : D \rightarrow B$$

- Bei einer Prozedur, die keine Funktion ist, wählt man als Bildbereich oft die leere Menge:

$$p : D \rightarrow \emptyset$$

  Dies signalisiert, dass die Seiteneffekte der Prozedur zur eigentlichen Umsetzung eines Algorithmus gehören, dagegen aber kein (bestimmtes) Element aus dem Bildbereich einer Abbildung als Ergebnis des Algorithmus angesehen werden kann.

- Sehr häufig findet man in imperativen Implementierungen von Algorithmen aber eine Mischform, in der eine Prozedur sowohl Seiteneffekte hat als auch einen nicht-leeren Bildbereich.

Prozeduren in Java

- In Java werden Prozeduren durch *Methoden* realisiert.
- Eine Methode wird definiert durch den Methodenkopf:

  ```
  public static <typ> <name>(<parameterliste>)
  ```

  und den Methodenrumpf, einen Block, der sich an den Methodenkopf anschließt.
- Als besonderer Ergebnis-Typ einer Methode ist **`void`** möglich. Dieser Ergebnis-Typ steht für die leere Menge als Bildbereich.
- Eine Methode mit Ergebnistyp **`void`** gibt *kein* Ergebnis zurück. Hier liegt der Sinn also ausschließlich in den Nebeneffekten.
- Eine Methode, deren Nebeneffekte den weiteren Programmverlauf beeinflussen, sollte als Ergebnis-Typ stets **`void`** haben.

Prozeduren in Java

- Manchmal wird auch **`boolean`** als Ergebnistyp gewählt. und zeigt mit dem Ergebnis*wert*, ob der beabsichtigte Nebeneffekt erfolgreich war.
- Das Ergebnis einer Methode ist der Ausdruck nach dem Schlüsselwort **`return`**. Nach Auswertung dieses Ausdrucks endet die Ausführung der Methode.

Prozeduren in Java

```
public class Streckenberechnung
{
  /**
   * Methode zur Berechnung der zurueckgelegten Strecke
   * nach Einwirken der Kraft <code>k</code>
   * auf einen Koerper der Masse <code>m</code>
   * fuer die Zeitdauer <code>t</code>.
   *
   * @param m Masse des Koerpers
   * @param t Zeitdauer
   * @param k Kraft
   * @return zurueckgelegte Strecke
   */
  public static double strecke(double m, double t, double k)
  {
    double b = k / m;
    return 0.5 * b * (t * t);
  }
}
```

Prozeduren in Java

Beispiel: Algorithmus für die Funktion

$$f(x) = \left(x + 1 + \frac{1}{x+1}\right)^2 \quad \text{für } x \neq -1$$

```
public class Funktionsberechnung
{
  /**
   * Methode zur Berechnung der Funktion
   * <i>f(x) = ((x + 1) + 1 / (x + 1))<sup>2</sup></i>.
   *
   * @param x der Eingabewert
   * @return Wert der Funktion f an der Stelle x
   */
  public static double f(double x)
  {
    double y = x + 1;
    y = y + 1/y;
    y = y * y;
    return y;
  }
}
```

Methodenaufruf

Wie wir im Abschnitt über Anweisungen gesehen haben, bildet ein Methodenaufruf eine Anweisung. Ein Methodenaufruf kann also überall da stehen, wo eine Anweisung möglich ist.
Beispiel für einen Methodenaufruf:

```
public class HelloWorld
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
      // auch println(...); ist ein Methodenaufruf
  }

  public static void main(String[] args)
  {
    // Aufruf der Methode gruessen
    gruessen("Hello, World!"); // abstrakter Parameter gruss
                               // wird mit konkretem Wert belegt
  }
}
```

Die `main`-Methode

- In Programmbeispielen haben wir bereits die `main`-Methode gesehen. Sie ermöglicht das selbständige Ausführen eines Programmes.
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus.
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array. Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.

Parameter für die `main`-Methode

Beispiel für einen Zugriff der `main`-Methode auf das Parameterarray:

```
public class Gruss
{
  public static void gruessen(String gruss)
  {
    System.out.println(gruss);
  }

  public static void main(String[] args)
  {
    gruessen(args[0]);
  }
}
```

Dadurch ist eine vielfältigere Verwendung möglich:

- `java Gruss "Hello, World!"`
- `java Gruss "Hallo, Welt!"`
- `java Gruss "Servus!"`

Verarbeitung von Parametern

Zur Verarbeitung der Parameter von Prozeduren kennen Programmiersprachen zwei grundsätzliche Möglichkeiten:

- **Call-by-value**
  Im Aufruf `methodenName(parameter)` wird für `parameter` im Methoden-Block eine neue Variable angelegt, in die der Wert von `parameter` geschrieben wird. Auf diese Weise bleibt die ursprüngliche Variable `parameter` von Anweisungen innerhalb der Methode unberührt.
- **Call-by-reference**
  Im Aufruf `methodenName(parameter)` wird die Variable `parameter` weiter verwendet. Wenn innerhalb der Methode der Wert von `parameter` verändert wird, hat das auch Auswirkungen außerhalb der Methode.
  Achtung: call-by-reference ist daher eine potentielle Quelle unbeabsichtigter Seiteneffekte.

In Java: call-by-value

Java wertet Parameter call-by-value aus. Für den Aufruf der Methode `swap` im folgenden Beispiel werden also Kopien der Variablen `x` und `y` angelegt.

```
public class Exchange
{
  public static void swap(int i, int j)
  {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args)
  {
    int x = 1;
    int y = 2;
    swap(x,y);
    System.out.println(x); // Ausgabe?
    System.out.println(y); // Ausgabe?
  }
}
```

Ausblick: call-by-reference-Effekt bei Objekten

Wenn der Parameter kein primitiver Typ ist, sondern ein Objekt (also z.B. ein Array – was genau Objekte sind, betrachten wir später) dann wird zwar in der Methode ebenfalls mit einer Kopie des Parameters gearbeitet, aber es handelt sich um eine Kopie der Speicheradresse.

```
public static void changeValues(int[] werte, int index, int wert)
{
  werte[index] = wert;
}

public static void main(String[] args)
{
  int[] werte = {0, 1, 2};
  changeValues(werte, 1, 3);
  System.out.println(werte[1]); // Ausgabe ?
}
```

Obwohl also auch hier die Parameterauswertung nach dem Prinzip call-by-value erfolgt, ist der Effekt der gleiche wie bei call-by-reference. Wir werden auf diesen Effekt zurückkommen.

Bedingte Anweisungen

Motivation:

- In vielen Algorithmen benötigt man eine Fallunterscheidung zur Lösung des gegebenen Problems.
- Beispiel: Falls . . . dann . . . im Algorithmus *Wechselgeld 1*

  Führe folgende Schritte der Reihe nach aus:

  1. Setze $w = ()$.
  2. Falls die letzte Ziffer von r eine 2, 4, 7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
  3. Falls die letzte Ziffer von r eine 1 oder 6 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  4. Falls die letzte Ziffer von r eine 3 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
  5. Solange $r < 100$: Erhöhe r um 5 und nimm 5 zu w hinzu.

Bedingte Anweisungen

- Im einfachsten Fall (Pseudo-Code):

  IF <Bedingung> **THEN** <Ausdruck1> **ELSE** <Ausdruck2> **ENDIF**

- Dies entspricht einer echten *Fallunterscheidung*:
  - Ist <Bedingung> wahr, dann wird <Ausdruck1> ausgewertet.
  - Ist <Bedingung> falsch, dann wird <Ausdruck2> ausgewertet.
- Im funktionalen Paradigma modelliert man mit einer Fallunterscheidung eine Abbildung. Deshalb müssen dort im Allgemeinen <Ausdruck1> und <Ausdruck2> den selben Typ haben.
- In der imperativen Programmierung ist dies nicht der Fall:
  - In den verschiedenen Zweigen stehen *Anweisungen* anstelle von Ausdrücken.
  - Damit entfällt die Forderung nach gleichen Typen.

Bedingte Anweisungen

- Man spricht daher im imperativen Paradigma von *bedingten Anweisungen*.
- Die Fallunterscheidung ist ein Spezialfall der bedingten Anweisung.
- Die einfachste Form von bedingten Anweisungen ist:

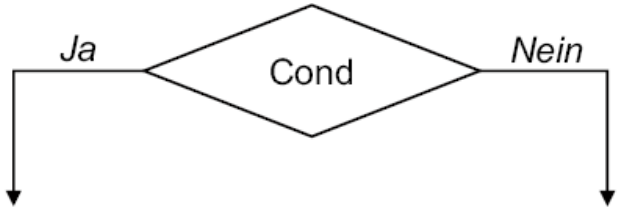
```
IF <Bedingung> THEN <Anweisungsfolge> ENDIF
```

- Bedingte Anweisungen können beliebig oft verzweigt werden:

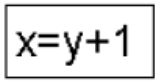
```
IF <Bedingung1> THEN <Anweisungsfolge1>
ELSE IF <Bedingung2> THEN <Anweisungsfolge2>
.
.
.
ELSE <AnweisungsfolgeN>
ENDIF
```

- Die einzelnen Zweige nennt man auch *bewachte Anweisungen*.

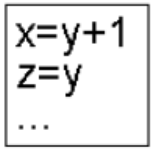
Graphische Darstellung: Kontrollflussdiagramme



Fallunterscheidung bzgl. der Bedingung `Cond`

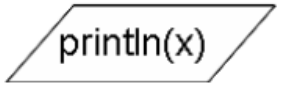


Anweisung



Block

```
{
  x = y + 1;
  z = y;
  ...
}
```

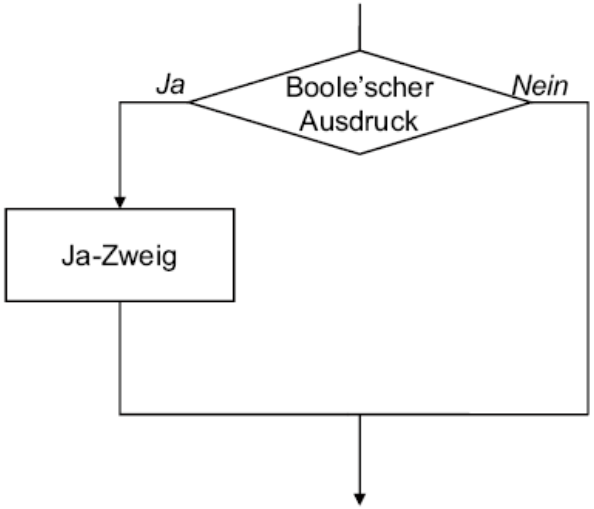


Ein- / Ausgabe

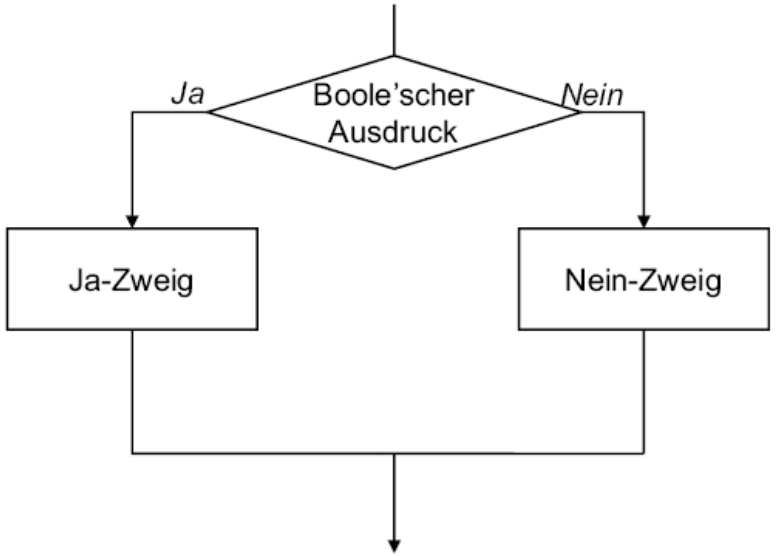
Sequentielle Abfolge

Kontrollflussdiagramme für bedingte Anweisungen

Einfache bedingte Anweisung:



Zweifache bedingte Anweisung:



Bedingte Anweisungen und Rekursion

- Rekursion ist ein nützliches und elegantes Entwurfskonzept für Algorithmen.
- Ein rekursiver Entwurf einer Methode verwendet das Ergebnis eines Aufrufs dieser Methode selbst in ihrem Rumpf wieder.
- Damit die Rekursion terminiert, benötigt man in der Regel einen oder mehrere Basisfälle (neben einem oder mehreren Rekursionsfällen).
- Diese verschiedenen Fälle werden typischerweise durch bedingte Anweisungen realisiert.

Erinnerung: Fakultäts-Funktion

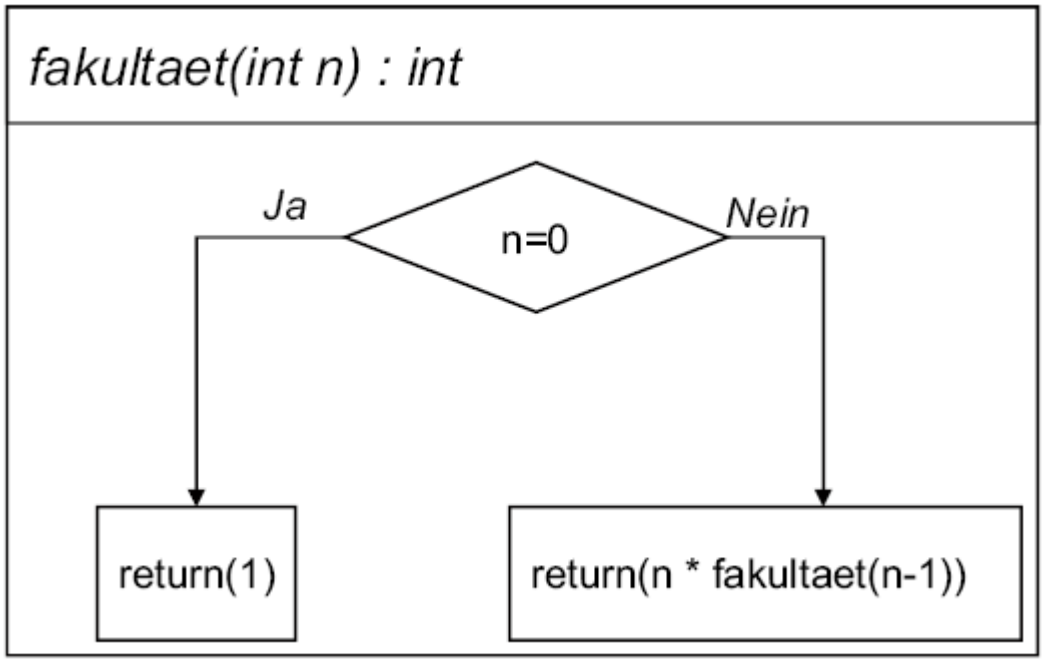
Eine rekursive Definition in der Mathematik haben wir am Beispiel der Fakultätsfunktion betrachtet:

- Die Fakultäts-Funktion !: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$ ist rekursiv definiert wie folgt:
  - $0! = 1$
  - $(n + 1)! = (n + 1) \cdot (n!)$
- Oft wird äquivalent statt der Rückführung von $n + 1$ auf n der Fall $n \neq 0$ auf n – 1 zurückgeführt:

$$n! = \begin{cases} 1, & \text{falls } n = 0, \\ n \cdot (n-1)! & \text{sonst.} \end{cases}$$

Bedingte Anweisungen und Rekursion

Beispiel:
Algorithmus zur Berechnung der Fakultät für eine natürliche Zahl $n \in \mathbb{N}$ als Kontrollflussdiagramm:



Bedingte Anweisungen in Java

- Java erlaubt zwei Formen von bedingten Anweisungen:
  1. Eine einfache Verzweigung:

```
if (<Bedingung>)
    <Anweisung>
```

  2. Eine zweifache Verzweigung:

```
if (<Bedingung>)
    <Anweisung1>
else
    <Anweisung2>
```

  Wobei
  - `<Bedingung>` ein Ausdruck vom Typ **`boolean`** ist,
  - `<Anweisung>`, `<Anweisung1>` und `<Anweisung2>` jeweils einzelne Anweisungen (also möglicherweise auch einen Block mit mehreren Anweisungen) darstellen.

Bedingte Anweisungen in Java

- Beispiel: Der Algorithmus zur Berechnung der Fakultät einer natürlichen Zahl $n \in \mathbb{N}$ kann in Java durch folgende Methode umgesetzt werden:

```
public static int fakultaet01(int n)
{
   if(n==0)
   {
      return 1;
   }
   else
   {
      return n * fakultaet01(n-1);
   }
}
```

Bedingte Anweisungen in Java

- Bedingte Anweisungen mit mehr als zwei Zweigen müssen in Java durch Schachtelung mehrerer **`if`**-Konstrukte ausgedrückt werden:

```
if (<Bedingung1>)
    <Anweisung1>
else if (<Bedingung2>)
   <Anweisung2>
.
.
.
else if (<BedingungN>)
    <AnweisungN>
else
    <AnweisungN+1>
```

Dangling Else

- Gegeben:

```
if (a)
  if (b)
    s1;
  else
    s2;
```

- **Zu welchem `if`-Statement gehört der `else`-Zweig?**

Dangling Else

- Antwort: Zur inneren Verzweigung **`if`** (b). (Die (falsche!) Einrückung ist belanglos für den Compiler und verführt den menschlichen Leser hier, das Programm falsch zu interpretieren.)
- Tipp: Immer Blockklammern verwenden!

```
if (a)
{
  if (b)
  {
    s1;
  }
  else
  {
    s2;
  }
}
```

Mehrfachanweisungen

- Beispiel: Ein Kunde hebt einen Betrag (`betrag`) von seinem Konto (`kontoStand` speichert den aktuellen Kontostand) ab. Falls der Betrag nicht gedeckt ist, wird eine Überziehungsgebühr fällig. Die fälligen Gebühren werden über einen bestimmten Zeitraum akkumuliert (`gebuehren`) und am Ende des Zeitraums in Rechnung gestellt. **Was ist falsch in folgender Berechnung?**

```
if (kontoStand >= betrag)
{
  double neuerStand = kontoStand - betrag;
  kontoStand = neuerStand;
}
else
kontoStand = kontoStand - betrag;
gebuehren = gebuehren + UEBERZIEH_GEBUEHR;
```

Mehrfachanweisungen

- Problem im vorherigen Beispiel:
  Überziehungsgebühr wird immer verlangt, auch wenn das Konto gedeckt ist.
- Lösung: Blockklammern setzen.

```
if (kontoStand >= betrag)
{
  double neuerStand = kontoStand - betrag;
  kontoStand = neuerStand;
}
else
{
  kontoStand = kontoStand - betrag;
  gebuehren = gebuehren + UEBERZIEH_GEBUEHR;
}
```

- Nun wird die Überziehungsgebühr nur verlangt, wenn das Konto *nicht* gedeckt ist.

Spezielle Mehrfachverzweigung in Java

- In Java gibt es eine weitere Möglichkeit, spezielle Mehrfach-verzweigungen auszudrücken.
- Die sog. **`switch`**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen.
- Syntax:

```
switch (ausdruck)
{
  case konstante1 : anweisung1
  case konstante2 : anweisung2
    .
    .
    .
  default : anweisungN
}
```

Spezielle Mehrfachverzweigung in Java

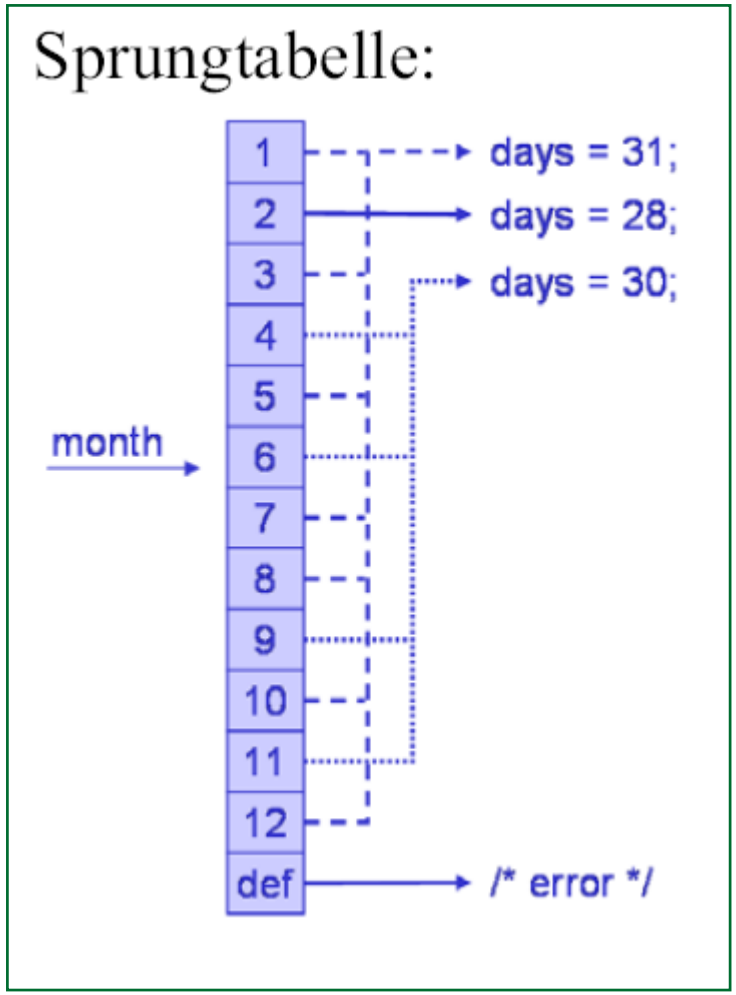
- Bedeutung:
  - Abhängig vom Wert des Ausdrucks ausdruck wird die *Sprungmarke* angesprungen, deren Konstante mit dem Wert von `ausdruck` übereinstimmt.
  - Die Konstanten und der Ausdruck müssen den selben Typ haben.
  - Die Anweisungen nach der Sprungmarke werden ausgeführt.
  - Die optionale **`default`**-Marke wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.
  - Fehlt die **`default`**-Marke und wird keine passende Sprungmarke gefunden, so wird keine Anweisung innerhalb der **`switch`**-Anweisung ausgeführt.

Spezielle Mehrfachverzweigung in Java

- Besonderheiten:
  - Der Ausdruck `ausdruck` darf nur vom Typ **`byte`**, **`short`**, **`int`** oder **`char`** sein.
  - Die **`case`**-Marken sollten alle verschieden sein, müssen aber nicht.
  - **Achtung**: Wird zu einer Marke gesprungen, werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt **keine** Unterbrechung, wenn das nächste Label erreicht wird, sondern es wird dort fortgesetzt! Dies ist eine beliebte Fehlerquelle!
  - Eine Unterbrechung kann durch die Anweisung **`break;`** erzwungen werden. Jedes **`break`** innerhalb einer **`switch`**-Anweisung verzweigt zum Ende der **`switch`**-Anweisung.
  - Nach einer Marken-Definition **`case`** muss nicht zwingend eine Anweisung stehen.

Beispiel

```
Switch (month)
{
  case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      days = 31; break;
  case 4: case 6: case 9: case 11:
      days = 30; break;
  case 2:
      if(leapYear)
      {
        days = 29;
      }
      else
      {
        days = 28;
      }
      break;
    default:
      /* error */
}
```



Wiederholungsanweisungen (Schleifen)

Motivation:

- Im Algorithmus *Wechselgeld 2* hatten wir eine Anweisung der Form **Solange** . . . : . . .
  Führe folgende Schritte der Reihe nach aus:
  1. Setze $w = ()$.
  2. Solange $r < 100$: Führe jeweils (wahlweise) einen der folgenden Schritte aus:
     1. Falls die letzte Ziffer von r eine 2, 4, 7 oder 9 ist, dann erhöhe r um 1 und nimm 1 zu w hinzu.
     2. Falls die letzte Ziffer von r eine 1, 2, 3, 6, 7 oder 8 ist, dann erhöhe r um 2 und nimm 2 zu w hinzu.
     3. Falls die letzte Ziffer von r eine 0, 1, 2, 3, 4 oder 5 ist, dann erhöhe r um 5 und nimm 5 zu w hinzu.

Wiederholungsanweisungen (Schleifen)

- Dabei handelt es sich um eine sog. *bedingte Wiederholungsanweisung* (*Schleife*).
- Allgemeine Form:

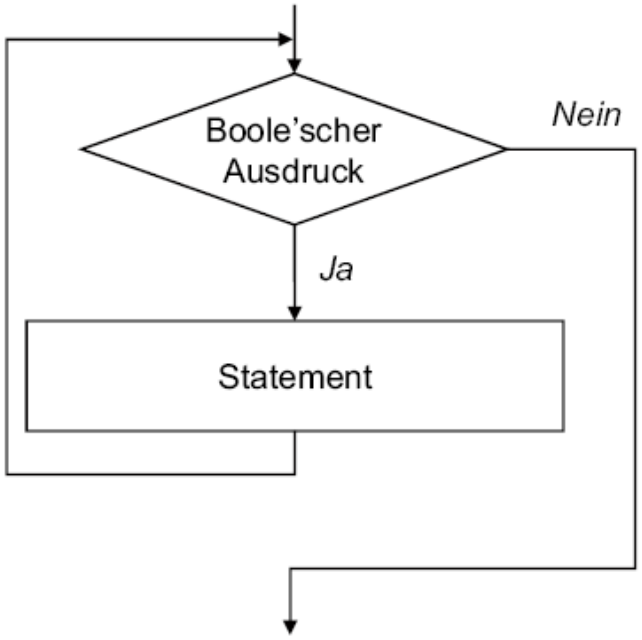
  WHILE <Bedingung> **DO** <Anweisung> **ENDDO**

- <Bedingung> heißt *Schleifenbedingung*, <Anweisung> heißt *Schleifenrumpf* und kann natürlich auch wieder ein Block sein, also aus mehreren Anweisungen bestehen.
- Variante:

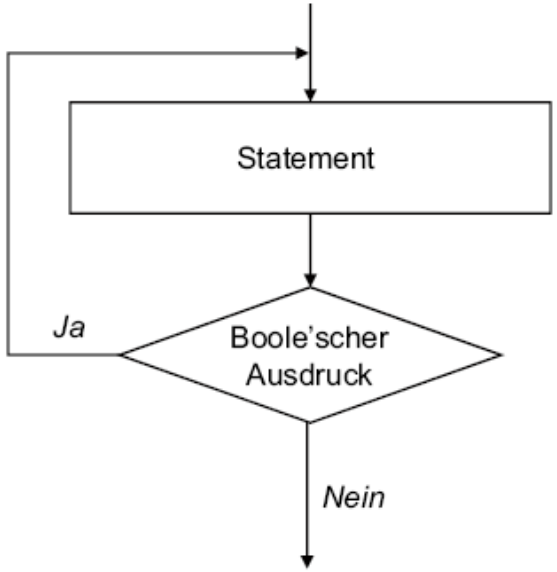
  DO <Anweisung> **WHILE** <Bedingung> **ENDDO**

Kontrollflussdiagramm für bedingte Wiederholungsanweisungen

WHILE-Schleife

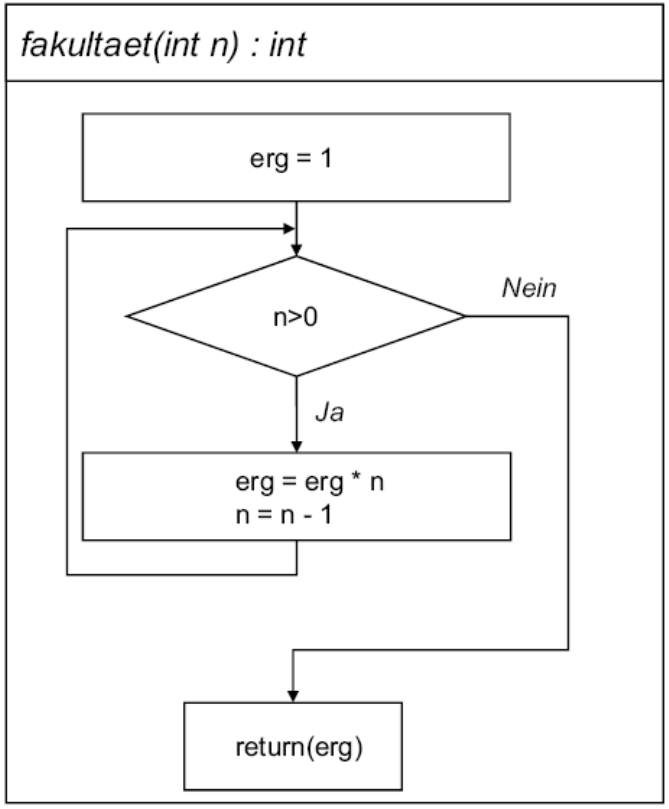


DO-Schleife



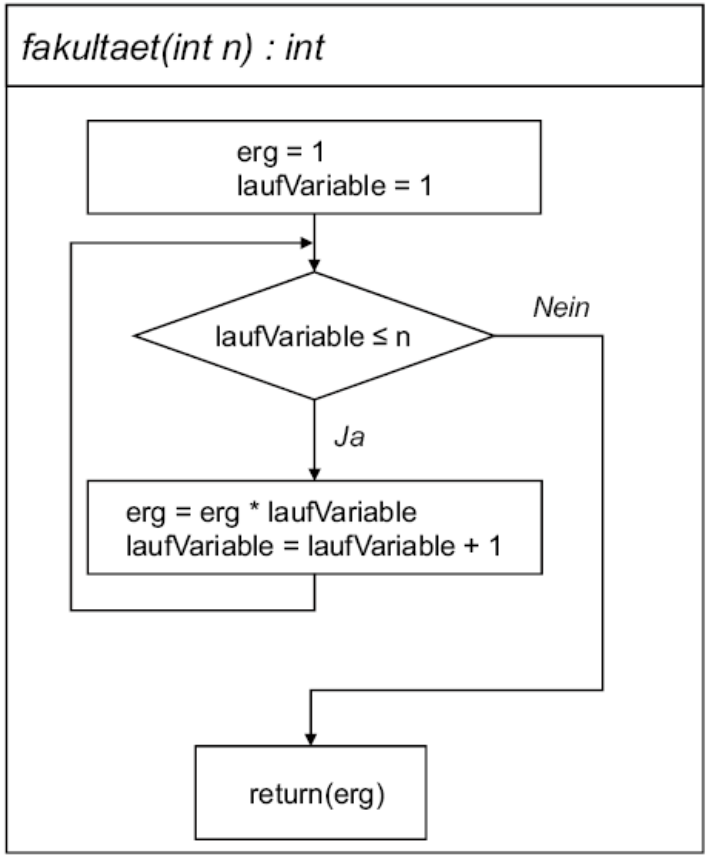
Wiederholungsanweisungen (Schleifen)

Beispiel: Fakultätsfunktion (nicht rekursiv) mit bedingter Wiederholungsanweisung



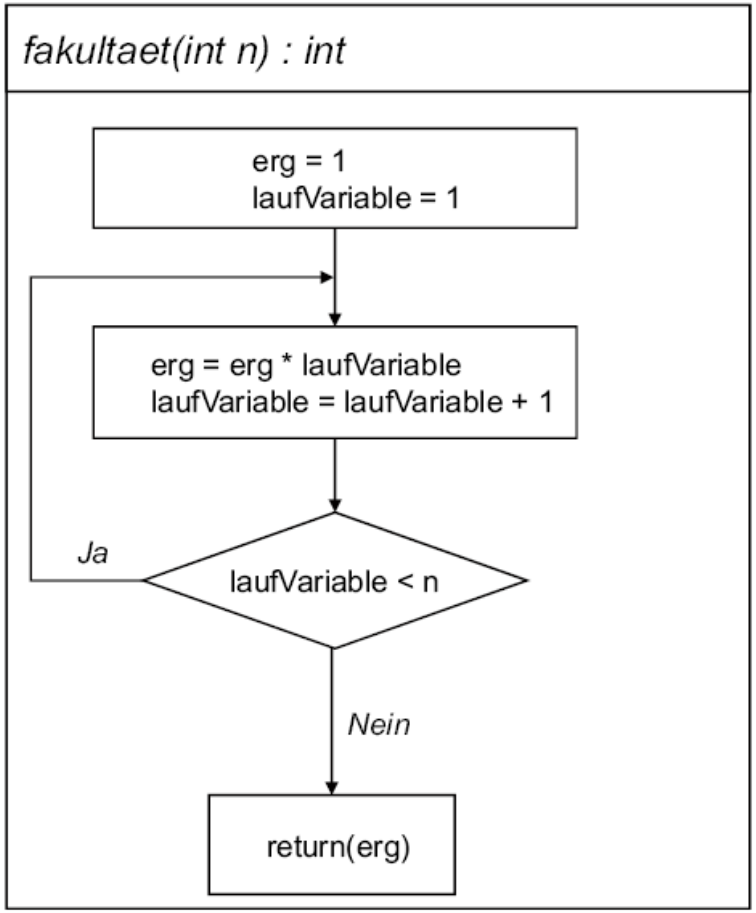
Wiederholungsanweisungen (Schleifen)

Variante: Mitzählen der durchgeführten Schritte



Wiederholungsanweisungen (Schleifen)

Variante: `Do`-Schleife



Wiederholungsanweisungen (Schleifen)

- Die Laufvariable (Zählen der Schritte) wurde in den letzten Beispielen zur Berechnung benutzt.
- Äquivalent zu den Formulierungen der Algorithmen zur Berechnung der Fakultätsfunktion durch die **`while`**- oder **`do`**-Schleife kann man das Ergebnis `erg` durch folgende Anweisungsfolge berechnen:

```
erg := 1;
erg := 1 * erg;
erg := 2 * erg;
.
.
.
erg := n * erg;
```

Wiederholungsanweisungen (Schleifen)

- Diese Folge von Zuweisungen könnte man wie folgt imperativ notieren:
  Führe für $i = 1,\ldots, n$ **nacheinander aus** : `erg` := i `*` `erg`.
- Dabei handelt es sich um eine sog. *gezählte Wiederholungsanweisung* (*Schleife*) (auch *Laufanweisung*).
- Allgemeine Form:

```
FOR <Zaehler> FROM <Startwert> TO <Endwert>
BY <Schrittweite> DO <Anweisung> ENDDO
```

- Analog zur bedingten Schleife heißt `<Anweisung>` *Schleifenrumpf* und kann wiederum aus mehreren Anweisungen (bzw. einem Block) bestehen.

Bedingte Schleifen in Java

- Java kennt mehrere Arten von bedingten Schleifen.
- Schleifen mit dem Schlüsselwort **`while`**:
  - Die klassische While-Schleife:

```
while (<Bedingung>)
  <Anweisung>
```

  - Die Do-While-Schleife:

```
do
  <Anweisung>
while (<Bedingung>);
```

- Dabei bezeichnet `<Bedingung>` einen Ausdruck vom Typ **`boolean`** und `<Anweisung>` ist entweder eine einzelne Anweisung, oder ein Block mit mehreren Anweisungen.
- Unterschied: `<Anweisung>` wird vor bzw. nach der Überprüfung von `<Bedingung>` ausgeführt.
- Ist `<Bedingung>` **`false`**, wird die Schleife verlassen.

Bedingte Schleifen in Java

Beispiel (Fakultät):

```
public static int fakultaet02(int n)
{
   int erg = 1;
   while (n > 0)
   {
      erg = erg * n;
      n--;
   }
   return erg;
}
```

Bedingte Schleifen in Java

Variante: Mitzählen der durchgeführten Schritte

```
public static int fakultaet03(int n)
{
  int erg = 1;
  int laufVariable = 1;
  while (laufVariable <= n)
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  return erg;
}
```

Bedingte Schleifen in Java

Variante: `Do`-Schleife

```
public static int fakultaet04(int n)
{
  int erg = 1;
  int laufVariable = 1;
  do
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  while (laufVariable < n);
  return erg;
}
```

Bedingte Schleifen in Java

- Eine weitere bedingte Schleife kann in Java mit dem Schlüsselwort **`for`** definiert werden:

```
for (<Initialisierung>; <Bedingung>; <Update>)
  <Anweisung>
```

- Alle drei Bestandteile im Schleifenkopf sind Ausdrücke (nur `<Bedingung>` muss vom Typ **`boolean`** sein).
- **Vorsicht**: Dieses Konstrukt ist keine klassische gezählte Schleife (auch wenn es **`for`**-Schleife genannt wird).

Die Bestandteile im Einzelnen

- Der Ausdruck `<Initialisierung>`
  - wird einmal vor dem Start der Schleife aufgerufen
  - darf Variablendeklarationen mit Initialisierung enthalten (um einen Zähler zu erzeugen); diese Variable ist nur im Schleifenkopf und innerhalb der Schleife (`<Anweisung>`) sichtbar
  - darf auch fehlen
- Der Ausdruck `<Bedingung>`
  - ist ähnlich wie bei den While-Konstrukten die Testbedingung
  - wird zu Beginn jedes Schleifendurchlaufs überprüft
  - die Anweisung(en) `<Anweisung>` wird (werden) nur ausgeführt, wenn der Ausdruck `<Bedingung>` den Wert **`true`** hat
  - kann fehlen (gleichbedeutend mit dem Ausdruck **`true`**)
- Der Ausdruck `<Update>`
  - verändert üblicherweise den Schleifenzähler (falls vorhanden)
  - wird am Ende jedes Schleifendurchlaufs ausgewertet
  - kann fehlen

Laufanweisungen in Java

- Eine gezählte Schleife wird in Java wie folgt mit Hilfe der **`for`**-Schleife notiert:

```
for (<Zaehler>=<Startwert>;
     <Zaehler> <= <Endwert>;
     <Zaehler> = <Zaehler> + <Schrittweite>)
<Anweisung>
```

- Beispiel (Fakultät):

```
public static int fakultaet05(int n)
{
  int erg = 1;
  for(int i=1; i<=n; i++)
  {
    erg = erg * i;
  }
  return erg;
}
```

Sprungbefehle: `break` und `continue`

- In Java gibt es Möglichkeiten, die normale Auswertungs-reihenfolge innerhalb einer **`do`**-, **`while`**- oder **`for`**-Schleife zu verändern.
- Der Befehl **`break`** beendet die Schleife sofort. Das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.
- Der Befehl **`continue`** beendet die aktuelle Iteration und beginnt mit der nächsten Iteration, d.h. es wird an den Beginn des Schleifenrumpfes “gesprungen”.
- Sind mehrere Schleifen ineinander geschachtelt, so gilt der **`break`** bzw. **`continue`**-Befehl nur für die aktuelle (innerste) Schleife.

Sprungbefehle: `break` und `continue`

- Mit einem *Sprungbefehl* kann man an eine beliebige Stelle in einem Programm "springen".
- Die Befehle **`break`** und **`continue`** können in Java auch für (eingeschränkte) Sprungbefehle benutzt werden.
- Der Befehl **`break <label>;`** bzw. **`continue <label>;`** muss in einem Block stehen, vor dem die Marke <**`label`**> vereinbart ist. Er bewirkt einen Sprung an das Ende dieses Anweisungsblocks.

Sprungbefehle: `break` und `continue`

Beispiel

```
int n = 0;
loop1:
for (int i=1; i<=10, i++)
{
  for (int j=1, j<=10, j++)
  {
    n = n + i * j;
    break loop1;
  }
}
System.out.print(n); // n = 1
```

- Der Befehl **`break`** `loop1;` erzwingt den Sprung an das Ende der äußeren **`for`**-Schleife.

Sprungbefehle: `break` und `continue`

Anmerkung:
Durch die Sprungbefehle (vor allem bei Verwendung von Labeln) wird ein Programm leicht unübersichtlich und Korrektheits-überprüfung wird schwierig. Wenn möglich, sollten Sprungbefehle vermieden werden.

Unerreichbare Befehle

- *Unerreichbare Befehle* sind Anweisungen, die (u.a.)
  - hinter einer **`break`**- oder **`continue`**-Anweisung liegen, die ohne Bedingung angesprungen werden,
  - in Schleifen stehen, deren Testausdruck zur Compile-Zeit **`false`** ist.
- Solche unerreichbaren Anweisungen sind in Java nicht erlaubt, sie werden vom Compiler nicht akzeptiert.
- Einzige Ausnahme sind Anweisungen, die hinter der Klausel

  ```
  if (false)
  ```

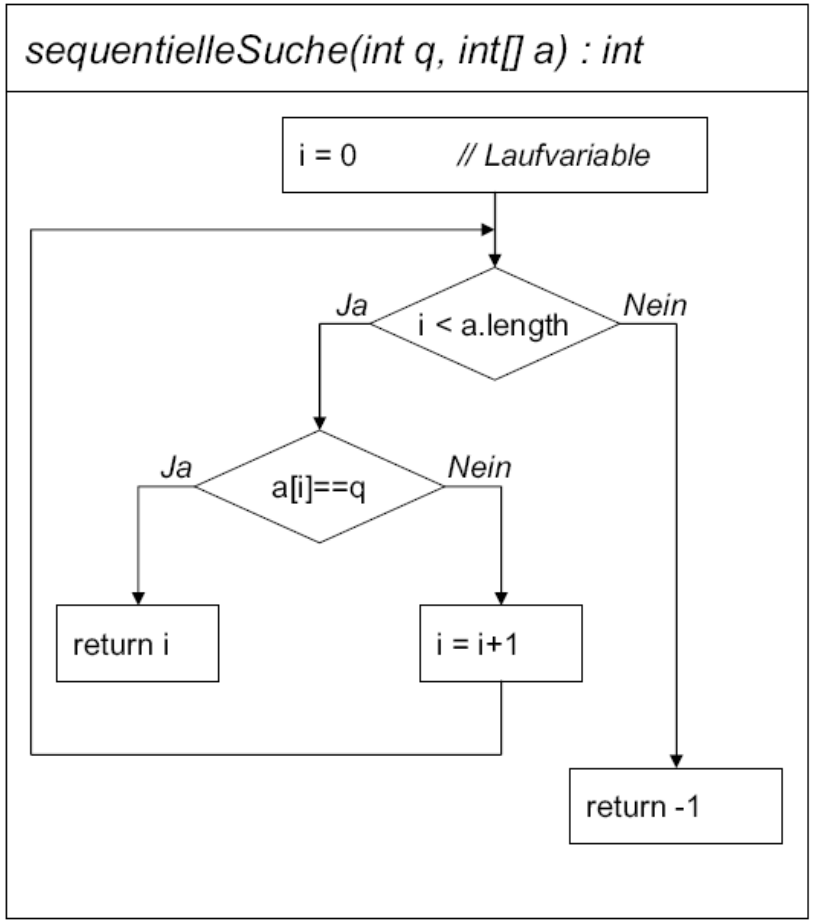
  stehen. Diese Anweisungen werden von den meisten Compilern nicht in den Bytecode übernommen, sondern einfach entfernt. Man spricht von *bedingtem Kompilieren*.

Beispiel: Suche in einem Array

- Aufgabe:
  - Sei A ein Array der Länge n mit Werten aus $\mathbb{N}$ und $q \in \mathbb{N}$.
  - Keiner der Werte kommt mehrfach vor.
  - Suche q in A und gib, falls die Suche erfolgreich ist, die Position i mit $A[i] = q$ aus. Falls die Suche erfolglos ist, gib den Wert -1 aus.
- Einfachste Lösung: *Sequentielle Suche*
  - Durchlaufe A von Position $i = 0, \ldots, n - 1$.
  - Falls $A[i] = q$, gib i aus und beende den Durchlauf.
  - Falls q nicht gefunden wird, gib -1 aus.

Beispiel: Suche in einem Array

- Algorithmus:



Beispiel: Suche in einem Array

- Java-Methode:

```
public static int sequentielleSuche(int q, int[] a)
{
  for(int i = 0; i < a.length; i++)
  {
    if(a[i]==q)
    {
      return i;
    }
  }
  return -1;
}
```

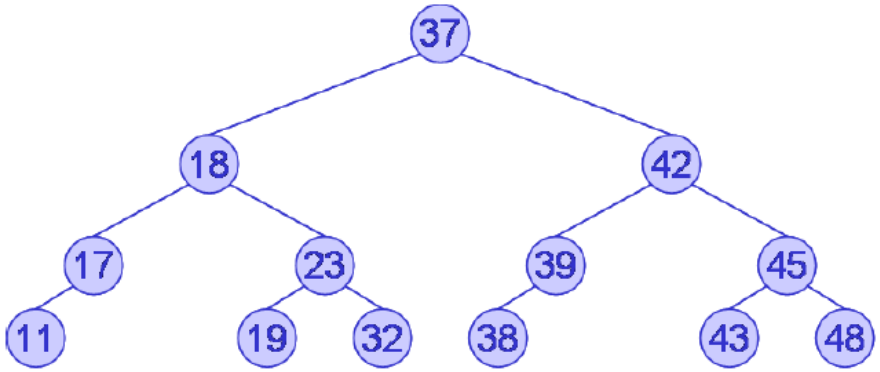
Beispiel: Suche in einem Array

- Anzahl der Vergleiche (= Analyse der Laufzeit):
  - Erfolgreiche Suche: n Vergleiche maximal, $n/2$ im Durchschnitt.
  - Erfolglose Suche: n Vergleiche.
- Falls das Array sortiert ist, funktioniert auch folgende Lösung: *Binäre Suche*
  - Betrachte den Eintrag $A[i]$ mit $i = n/2$ in der Mitte des Arrays
    - Falls $A[i] = q$, gib i aus und beende die Suche.
    - Falls $A[i] > q$, suche in der linken Hälfte von A weiter.
    - Falls $A[i] < q$, suche in der rechten Hälfte von A weiter.
  - In der jeweiligen Hälfte wird ebenfalls mit binärer Suche gesucht.
  - Falls die neue Hälfte des Arrays leer ist, gib -1 aus.

Beispiel: Suche in einem Array

Beispiel: A:	11	17	18	19	23	32	37	38	39	42	43	45	48

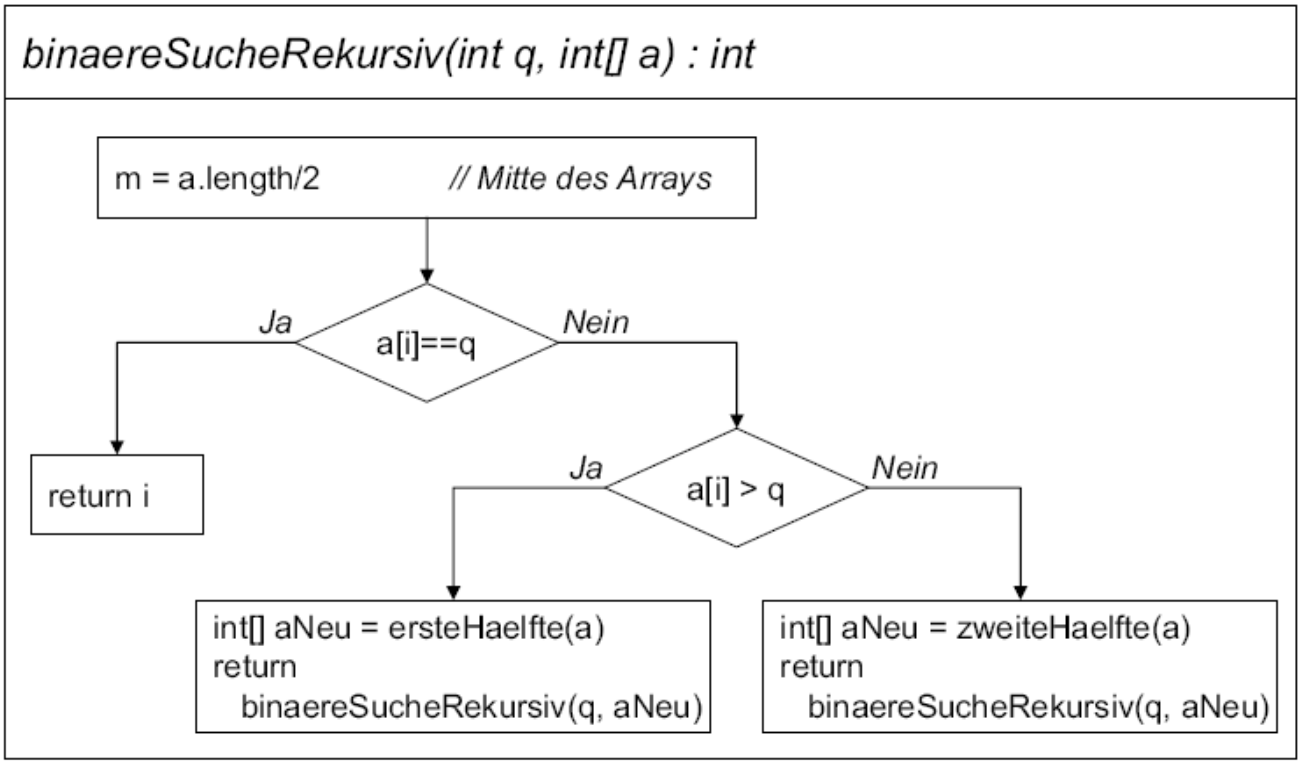
- Entscheidungsbaum:



- Analyse der Laufzeit: Maximale Anzahl von Vergleichen entspricht Höhe h des Entscheidungsbaums: $h = \lceil \log_2(n+1) \rceil$
- Vergleich der Verfahren:
  - $n = 1.000$:
    Sequentielle Suche: 1.000 Vergleiche – Binäre Suche: 10 Vergleiche
  - $n = 1.000.000$:
    Sequentielle Suche: 1.000.000 Vergleiche – Binäre Suche: 20 Vergleiche

Beispiel: Suche in einem Array

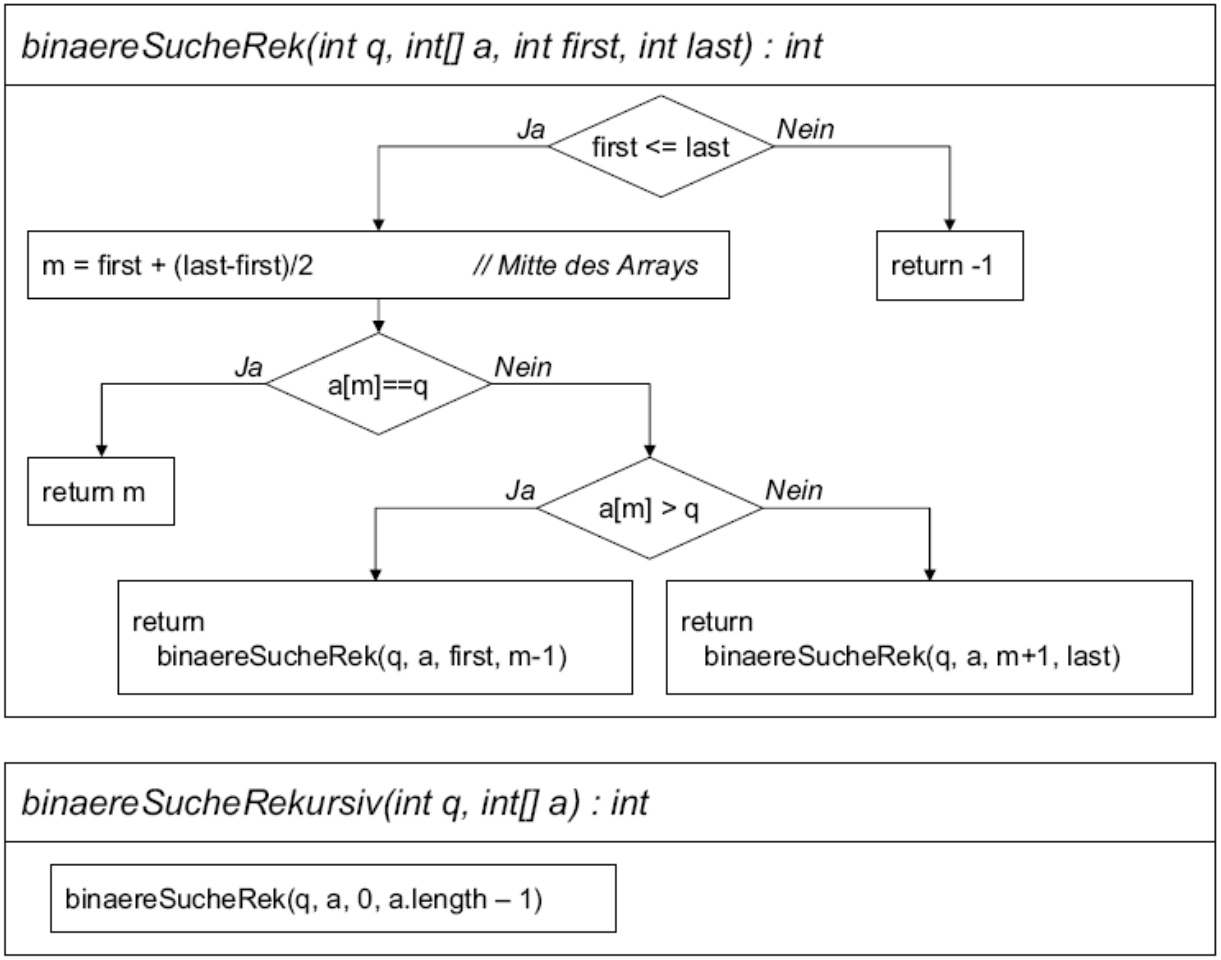
Rekursiver Algorithmus:



Probleme?

Beispiel: Suche in einem Array

Alternativer rekursiver Algorithmus mit nur einem Array:



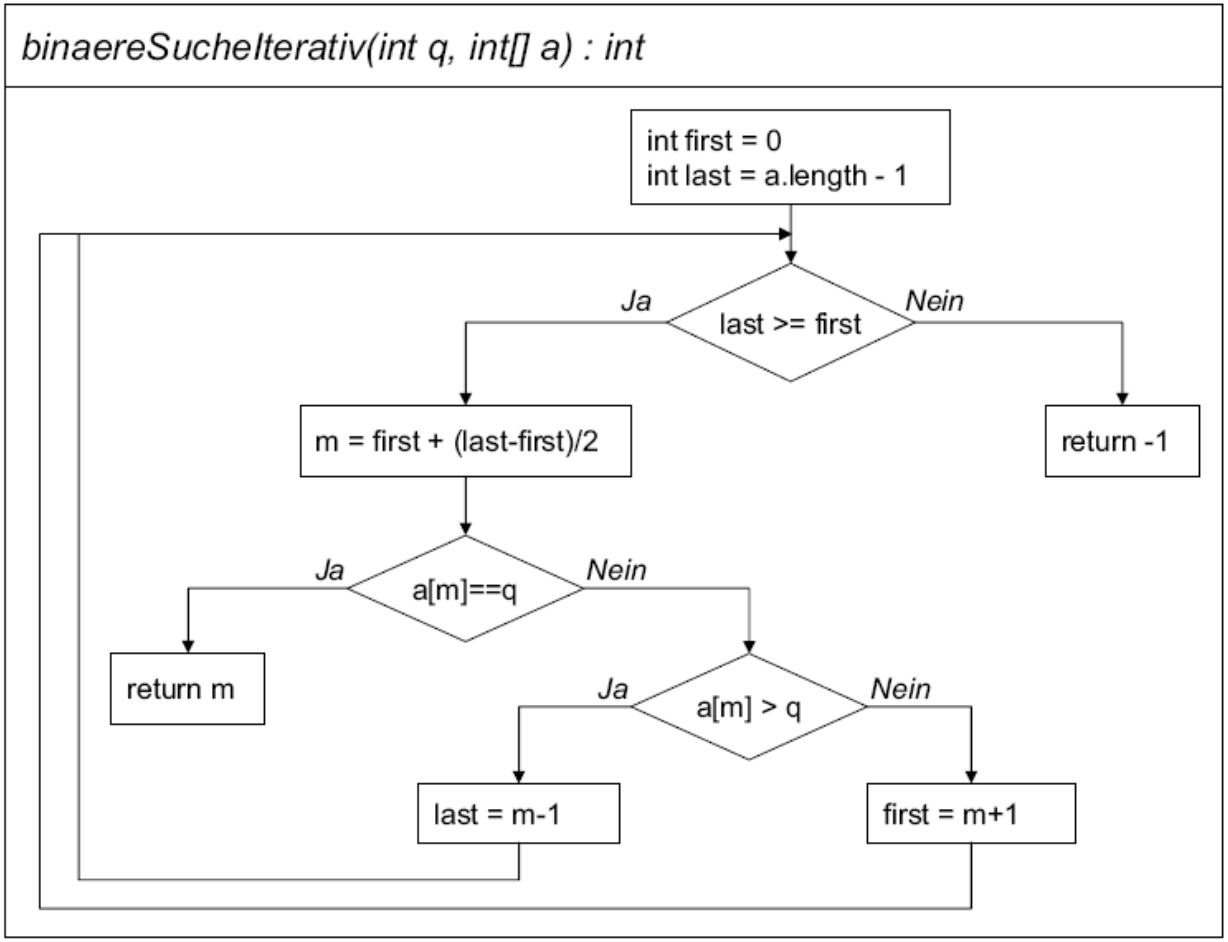
Beispiel: Suche in einem Array

Alternativer rekursiver Algorithmus in Java:

```
public static int binaereSucheRek (int q, int[] a, int first, int last)
{
  if (first > last)
  {
    return -1;  // leeres Array
  }
  int m = first + (last-first) / 2;
  if (q == a[m])
  {
    return m;   // q gefunden
  }
  else if (a[m] > q)
  {
    return binaereSucheRek(q, a, first, m-1);
  }
  else /* a[m] < q */
  {
    return binaereSucheRek(q, a, m+1, last);
  }
}
public static int binaereSucheRekursiv (int q, int[] a)
{
  return binaereSucheRek (q, a, 0, a.length - 1);
}
```

Beispiel: Suche in einem Array

Alternativer *iterativer* Algorithmus mit nur einem Array:



Beispiel: Suche in einem Array

Iterativer Algorithmus in Java:

```
public static int binaereSucheIterativ (int q, int[] a)
{
  int first = 0;
  int last = a.length - 1;
  while (last >= first)
  {
    int m = (last + first) / 2;
    if (q == a[m])
    {
      return m;
    }
    else if (q < a[m])
    {
      last = m - 1;
    }
    else /*q > a[m]*/
    {
      first = m + 1;
    }
  }
  return -1;  // not found
}
```

Zusammenfassung

Sie kennen jetzt die Grundkonzepte imperativer Programmierung:

- Grunddatentypen und ihre typischen Operationen als elementare Bestandteile der meisten Programmiersprachen,
- das Konzept von (lokalen und globalen) Variablen und Konstanten (und deren Unterschied) zur sequentiellen Verarbeitung von Information in imperativen Programmen,
- das Konzept von (sequentiellen) Anweisungen,
- Blöcke als Zusammenfassung mehrerer Anweisungen,

  ...

Zusammenfassung

- …
- die Gültigkeitsbereiche verschiedener Arten von Variablen,
- Arrays (Reihungen) als grundlegende Datenstruktur zur Verwaltung einer Menge gleichartiger Werte im imperativen Programmierparadigma,
- Methoden (Prozeduren) als Mittel zur Abstrahierung von Algorithmen und einzelnen Berechnungsschritten sowie (durch die Verwendung formaler Parameter) von den konkreten Daten,
- typische imperative Kontrollstrukturen zur Implementierung von bedingten, kontrollierten, iterierten Anweisungen und können somit im Prinzip einen Algorithmus imperativ in Java implementieren.