TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Programmparameter in Java-Programmen verwenden
- Typumwandlungen verstehen und anwenden
- Begriff des Algorithmus kennen und anwenden können
  - Ausführungszustand = Speicherzustand + Steuerungszustand
  - Terminierung, Determinismus, Determiniertheit
- Anweisungen in Java-Programmen verwenden
  - Deklaration und Zuweisung
  - Verzweigungen
  - Schleifen
  - Sprung- und Auswahlanweisungen

Praxisbeispiel: Schaltjahre

Wann ist ein Jahr ein Schaltjahr?

Wir wollen Programm schreiben, das für eine eingegebene Jahreszahl bestimmt, ob es sich um ein Schaltjahr handelt. Wenn der erste Programmparameter ein Schaltjahr ist, soll `true` ausgegeben werden, andernfalls `false`.

Aus Wikipedia-Artikel *Schaltjahr*:

- Die durch 4 ganzzahlig teilbaren Jahre sind Schaltjahre. [...]
- Die durch 100 ganzzahlig teilbaren Jahre (z.B. 1700, 1800, 1900, 2100 und 2200) sind keine Schaltjahre. [...]
- Schließlich sind die ganzzahlig durch 400 teilbaren Jahre doch Schaltjahre. Damit sind 1600, 2000, 2400, ... jeweils wieder Schaltjahre. [...]

Ausgabe von Daten

```
System.out.println(Ausdruck);
System.out.print(Ausdruck);
```

Ausgabe von Daten auf der Konsole.

Der Ausdruck kann von einem beliebigen Typ sein und wird automatisch in einen `String` umgewandelt.

Eingabe von Daten: Programmparameter

Beim Starten des Programms können **Programmparameter** (auch: Befehlszeilenargumente) angegeben werden.

```
public class Beispiel {
  public static void main(String[] args) {
    System.out.println(args[0]);
    System.out.println(args[1]);
  }
}
```

Der Ausdruck `args[0]` liefert den ersten Programmparameter.
Der Ausdruck `args[1]` den zweiten, usw.

```
> javac Beispiel.java
> java Beispiel 17 hallo
17
hallo
```

Programmparameter und Typumwandlungen

Programmparameter sind vom Typ `String`.

Umwandeln von `String` zu `int`: `Integer.parseInt(`Ausdruck`)`

Beispiel:

```
public class Add {
  public static void main(String[] args) {
    System.out.println(args[0] + args[1]);
    System.out.println(Integer.parseInt(args[0])
                     + Integer.parseInt(args[1]));
  }
}
```

Programmparameter und Typumwandlungen

Programmparameter sind vom Typ `String`.

Umwandeln von `String` zu `int`: `Integer.parseInt(`Ausdruck`)`

Beispiel:

```
public class Add {
  public static void main(String[] args) {
    System.out.println(args[0] + args[1]);
    System.out.println(Integer.parseInt(args[0])
                     + Integer.parseInt(args[1]));
  }
}
```

```
> javac Add.java
> java Add 4 2
```

Programmparameter und Typumwandlungen

Programmparameter sind vom Typ `String`.

Umwandeln von `String` zu `int`: `Integer.parseInt(`Ausdruck`)`

Beispiel:

```
public class Add {
  public static void main(String[] args) {
    System.out.println(args[0] + args[1]);
    System.out.println(Integer.parseInt(args[0])
                     + Integer.parseInt(args[1]));
  }
}
```

```
> javac Add.java
> java Add 4 2

42
6
```

Typumwandlungen: Explizite Typumwandlung

Zur Umwandlung von Strings in numerische Werte:

`int Integer.parseInt(String s)`	Wandelt `s` in `int`-Wert um
`double Double.parseDouble(String s)`	Wandelt `s` in `double`-Wert um
`boolean Boolean.parseBoolean(String s)`	Wandelt `s` in `boolean`-Wert um

Beispiele:

`Integer.parseInt("123")`	ergibt sich zu
`Integer.parseInt("007")`	ergibt sich zu
`Double.parseDouble("4")`	ergibt sich zu
`Integer.parseInt(10)`	ergibt sich zu
`Integer.parseInt("ten")`	ergibt sich zu

Typumwandlungen: Explizite Typumwandlung

Zur Umwandlung von Strings in numerische Werte:

`int Integer.parseInt(String s)`	Wandelt `s` in `int`-Wert um
`double Double.parseDouble(String s)`	Wandelt `s` in `double`-Wert um
`boolean Boolean.parseBoolean(String s)`	Wandelt `s` in `boolean`-Wert um

Beispiele:

`Integer.parseInt("123")`	ergibt sich zu	123
`Integer.parseInt("007")`	ergibt sich zu	
`Double.parseDouble("4")`	ergibt sich zu	
`Integer.parseInt(10)`	ergibt sich zu	
`Integer.parseInt("ten")`	ergibt sich zu	

Typumwandlungen: Explizite Typumwandlung

Zur Umwandlung von Strings in numerische Werte:

`int Integer.parseInt(String s)`	Wandelt `s` in `int`-Wert um
`double Double.parseDouble(String s)`	Wandelt `s` in `double`-Wert um
`boolean Boolean.parseBoolean(String s)`	Wandelt `s` in `boolean`-Wert um

Beispiele:

`Integer.parseInt("123")`	ergibt sich zu	`123`
`Integer.parseInt("007")`	ergibt sich zu	`7`
`Double.parseDouble("4")`	ergibt sich zu	`4.0`
`Integer.parseInt(10)`	ergibt sich zu	Übersetzungsfehler
`Integer.parseInt("ten")`	ergibt sich zu	Laufzeitfehler

Typumwandlungen: Expliziter Cast

- Java bietet integrierte Typumwandlung für primitive Datentypen
- *Beispiel:* `(int)2.71828` ergibt den `int`-Wert `2`
- **Achtung: Informationsverlust**
- Auf Klammerung achten, da Casts stärker binden als arithmetische Operationen

- *Hinweis:* Runden mittels der Bibliotheksfunktion `long Math.round(double d)` und explizitem Cast von `long` auf `int`
  Beispiel: `(int) Math.round(2.71828)` liefert `3`

Typumwandlungen: Automatische Promotion für Zahlen

- Daten von primitiven numerischen Typen können in einem Kontext verwendet werden, wenn in diesem ein Wert eines Datentyps verwendet wird, der einen größeren Wertebereich abdeckt
- *Beispiel:* Bei `4.0 * 3` wird `3` automatisch in `double`-Wert umgewandelt, danach Multiplikation auf `double`
- *Beispiel:* Bei `"Ocean's " + 11` wird `11` automatisch in `String`-Wert umgewandelt, danach Konkatenation auf `String`

Typumwandlungen: Automatische Promotion für Zahlen

- Daten von primitiven numerischen Typen können in einem Kontext verwendet werden, wenn in diesem ein Wert eines Datentyps verwendet wird, der einen größeren Wertebereich abdeckt
- *Beispiel:* Bei `4.0 * 3` wird `3` automatisch in `double`-Wert umgewandelt, danach Multiplikation auf `double`
- *Beispiel:* Bei `"Ocean's " + 11` wird `11` automatisch in `String`-Wert umgewandelt, danach Konkatenation auf `String`

Auch bei der automatischen Promotion von Zahlen entsteht unter Umständen ein Informationsverlust!

```
int x = 16777218;
int y = 16777217;
float xx = x;
System.out.println(xx - y);
```

Welcher Wert wird hier ausgegeben?

Typumwandlungen: Automatische Promotion für Zahlen

- Daten von primitiven numerischen Typen können in einem Kontext verwendet werden, wenn in diesem ein Wert eines Datentyps verwendet wird, der einen größeren Wertebereich abdeckt
- *Beispiel:* Bei `4.0 * 3` wird `3` automatisch in `double`-Wert umgewandelt, danach Multiplikation auf `double`
- *Beispiel:* Bei `"Ocean's " + 11` wird `11` automatisch in `String`-Wert umgewandelt, danach Konkatenation auf `String`

Auch bei der automatischen Promotion von Zahlen entsteht unter Umständen ein Informationsverlust!

```
int x = 16777218;
int y = 16777217;
float xx = x;
System.out.println(xx - y);
```

Welcher Wert wird hier ausgegeben?

Der ausgegebene Wert ist 2.0.

Variablen

Werte können in **Variablen** gespeichert und später verwendet werden:

```
public class Add {
  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println(x + y);
  }
}
```

Variablen

Der Wert einer Variablen kann verändert werden:

```
public class Add {
  public static void main(String[] args) {
    int res = 0;
    res = res + Integer.parseInt(args[0]);
    res = res + Integer.parseInt(args[1]);
    System.out.println(res);
  }
}
```

Variablen - Syntax in Java I

```
Deklaration →
      Typ  ≪ Bezeichner ≫ ;
    | Typ  ≪ Bezeichner ≫  =  Ausdruck;

Typ → PrimitiverTyp | ≪ Bezeichner ≫

PrimitiverTyp →
      byte | short | int | long
    | float | double | char | boolean

Zuweisung →
     ≪ Bezeichner ≫ = Ausdruck;

Ausdruck →
     ≪ Bezeichner ≫
```

Variablen - Syntax in Java II

Der **Bezeichner** (engl. identifier) ist hier der Name der Variable.

- Bezeichner bestehen aus Buchstaben, Ziffern und Unterstrich.
- Sie starten nicht mit einer Ziffer.
- Bezeichner dürfen nicht mit Schlüsselwörtern übereinstimmen.
- Groß- und Kleinschreibung sind entscheidend:
  `a` und `A` sind unterschiedliche Bezeichner

Variablen

Charakteristische Operationen auf einer Variablen v sind:

- **Zuweisen** eines Werts w an v;
- **Lesen** des Wertes, den v enthält/speichert/hat.

Zustand einer Variablen: gespeicherter Wert

Graphische Darstellung	**Sprachgebrauch**
v: true	`v` enthält den Wert `true` `v` speichert den Wert `true`
x: 7	`x` hat den Wert 7 `x` ist 7

Testen von Programmen

Testen von Programmen

Softwaretests sind eine der wichtigsten Maßnahmen zur Qualitätssicherung in der Software-Entwicklung. Sie helfen uns Fehler in der Software zu erkennen.

> *"Ein Test [...] ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen" (Denert, 1991)*

Verschiedene Fehlerquellen

Lexikalische und syntaktische Fehler:

Programmtext entspricht nicht den Regeln der Programmiersprache.
Fehler wird beim Übersetzen vom Compiler erkannt.

- Fehlende Klammern:

```
Beispiel.java:5: error: ')' expected
                int x = (3 + 4 * 5;
                                   ^
```

- Typfehler:

```
Beispiel.java:5: error: incompatible types: int cannot
be converted to String
                int x = Integer.parseInt(42);
                                         ^
```

Verschiedene Fehlerquellen

Logische Fehler:
Programm hat nicht das gewünschte Verhalten.

Designfehler: Fehler, die bereits bei der Definition der Anforderungen an die Software oder auch bei der Entwicklung des Design entstehen.

Laufzeitfehler:
Programm-Ausführung bricht wegen Fehler ab.

- `Integer.parseInt("blub")` führt zu Laufzeitfehler
  `java.lang.NumberFormatException: For input string: "blub"`
- `int y = 10 / Integer.parseInt(args[0]);`
  Führt zu Laufzeitfehler, wenn erster Programmparameter `0` ist:
  `java.lang.ArithmeticException: / by zero`

Testen

> *"Ein Test [...] ist der überprüfbare und* **jederzeit wiederholbare** *Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen" (Denert, 1991)*

Logische Fehler können durch Testen gefunden werden.

Idee: Ausführen des Programms mit verschiedenen Eingaben.
Vergleich der Ausgabe mit **erwarteter** Ausgabe.

Testen – Wählen von Eingaben

In der Regel können nicht alle möglichen Eingaben getestet werden.

> *„Program testing can be used to show the presence of bugs, but never show their absence! “(Edsger W. Dijkstra)*

Ziel: trotzdem möglichst viele Fehler finden /
geringe Wahrscheinlichkeit für unentdeckte Fehler.

- Bei Fallunterscheidungen mindestens eine Eingabe pro Fall
- Randfälle testen

Wenn sich die Testfälle aus der Problembeschreibung abgeleitet werden, da Implementierungsdetails nicht bekannt sind, spricht man von *Black-Box Tests*. Bei *White-Box Tests* ist die Implementierung bekannt, und man kann aus ihr Testfälle ableiten.

Testen – Wählen von Eingaben

Frage: Warum sind die Eingabedaten 1992, 2016, 2040 **allein** keine geeigneten Testdaten, um das Verhalten des `Schaltjahr`-Programms zu testen?

Testen – Wählen von Eingaben

Frage: Warum sind die Eingabedaten 1992, 2016, 2040 **allein** keine geeigneten Testdaten, um das Verhalten des `Schaltjahr`-Programms zu testen?

Antwort: Die Jahre 1992, 2016, 2040 sind alles Schaltjahre. Daher wird durch diese Eingaben nur ein Teil der Spezifikation des Programms abgedeckt.
Besser wären:
1992 (durch 4 teilbar, aber nicht durch 100),
1990 (nicht durch 4 teilbar),
1900 (durch 100 teilbar, aber nicht durch 400),
2000 (durch 4, 100 und 400 teilbar).

Testen

1978: Ein Vorzeichenfehler in der Software des Kampfflugzeug F-16 sorgte dafür, dass der Autopilot das Flugzeug in Rückenlage brachte, wenn es den Äquator überflog. Der Fehler entstand, da man keine negativen Breitengrade als Eingabedaten bedacht hatte. Diese kritische Situation wurde erst spät in der Entwicklungsphase des F-16 während einer Simulation entdeckt.

Algorithmus

Der Begriff des Algorithmus

- *Prozedurale* Modellierung und Programmierung baut auf klassischem Algorithmusbegriff
- Berechnung ist dabei *zustandsändernder* Ablauf
- Orientiert sich am Berechnungskonzept von Rechnern

Begriffsklärung: Algorithmus

Ein **Algorithmus** ist ein Verfahren zur schrittweisen **Ausführung** von (**Berechnungs-**) **Abläufen**, das sich präzise und endlich beschreiben lässt, so dass:

- die Beschreibung auf wohlverstandenen, ausführbaren ("effektiven") Einzelschritten basiert;
- in jedem Schritt eine oder mehrere Aktionen (ggf. parallel) ausgeführt werden;
- jede Aktion von einem Zustand in einen Nachfolgezustand führt.

Terminierung

Die Ausführung eines Algorithmus **terminiert**, wenn sie nach endlich vielen Schritten beendet ist;
andernfalls spricht man von einer **nicht-terminierenden** Ausführung.

Beschreibung eines Algorithmus

Algorithmen lassen sich mit unterschiedlichen Sprachmitteln beschreiben:

- umgangssprachlich
- mit mathematischer Sprache
- in graphischer Notation
- mit programmiersprachlichen Mitteln

Ein Algorithmus ist *unabhängig* von der verwendeten Beschreibungstechnik bzw. Sprache.

Algorithmus in Umgangssprache

Berechnung des größten gemeinsamen Teilers:

Seien m, n, v Variablen für Integer-Werte.

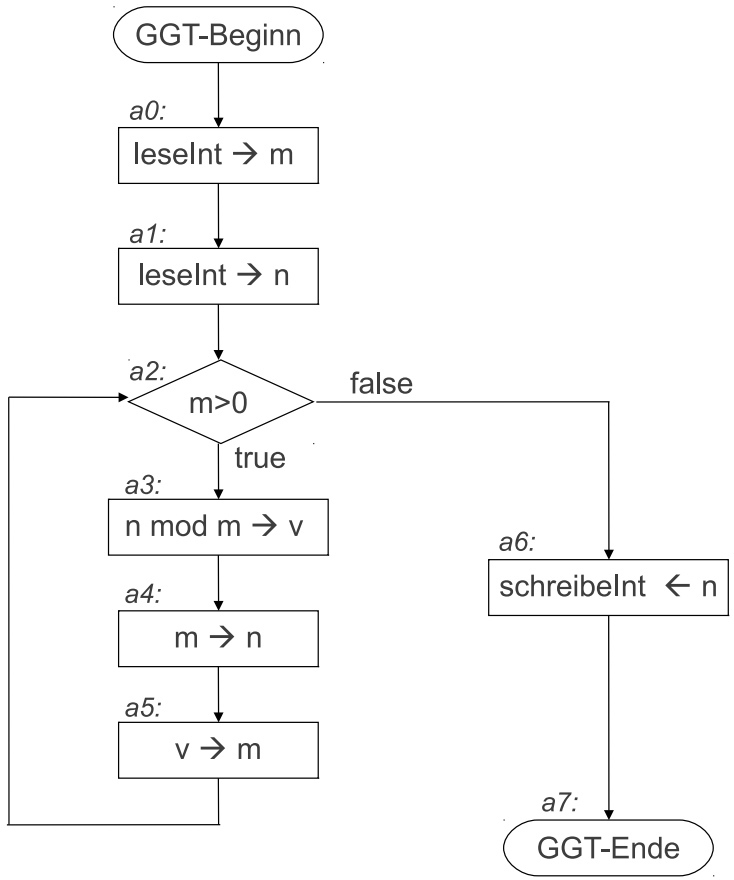
Lese die Werte $w1$ und $w2$ ein, für die der ggT berechnet werden soll und weise $w1$ an m und $w2$ an n zu.

Solange der Wert von m größer als 0 ist, tue Folgendes und prüfe danach wieder die Bedingung:

- Berechne $n \, mod \, m$ und weise das Ergebnis an v zu;
- Weise den Wert von m an n zu;
- Weise den Wert von v an m zu;

Gebe den Wert aus, den n enthält.

Algorithmus als Flussdiagramm



Algorithmus als Java-Programm

```
public class GGT { // Berechnet ggT fuer 2 gelesene Werte
 public static void main(String[] args) {
   int m = Integer.parseInt(args[0]);
   int n = Integer.parseInt(args[1]);

   while (m > 0) {
     int v = n % m;
     n = m;
     m = v;
   }
   System.out.println(n);
}
```

Begriffsklärung: Zustände

Jeder Schritt bei der Ausführung eines Algorithmus führt von einem **Ausführungszustand** zum **Nachfolgezustand**.
Ein Ausführungszustand ist gekennzeichnet durch

- den **Speicherzustand**
  im Wesentlichen der Zustand der Variablen
- den **Steuerungszustand**
  vereinfacht gesagt, die Stelle im Programm, an der die Ausführung angekommen ist

Ein Ausführungsschritt führt zu einer *Zustandsänderung*.

Begriffsklärung: Aktion

In einem Ausführungsschritt wird üblicherweise eine **Aktion** ausgeführt.

- Zuweisungen an Variablen
- Prüfen von Bedingungen
- Kommunikation mit der Umgebung (Ein- und Ausgabe)

Begriffsklärung: Ablauf

Der **Ablauf** eines Algorithmus zu gegebenen Eingaben wird charakterisiert durch

- die Sequenz der Ausführungszustände und
- die Sequenz der ausgeführten Aktionen

Begriffsklärung: Deterministisch

Ein Algorithmus heißt **deterministisch**, wenn für alle Eingabedaten der Ablauf des Algorithmus eindeutig bestimmt ist.

Andernfalls heißt er **nicht-deterministisch**.

Beispiel: Nicht-deterministischer Algorithmus

Aufgabe:
Erkenne, ob eine Zeichenkette z eine andere Zeichenkette s enthält.

Eingabe:
Zeichenketten z und s

Ausgabe:
ja/nein

Beispiel: Nicht-deterministischer Algorithmus

- $P := \{0, \ldots, \text{länge}(z) - \text{länge}(s)\}$
- Solange P nicht leer ist, tue folgendes:
  - Wähle ein x aus P aus
  - Entferne x aus P
  - $w := z$ ohne die ersten x Buchstaben
  - Prüfe, ob w mit der Zeichenkette s beginnt
  - Falls ja, terminiert der Algorithmus mit "ja"
- Wenn keine Position mehr in der Menge P ist: terminiere mit der Ausgabe "nein".

Beispiel: Nicht-deterministischer Algorithmus

- $P := \{0, \ldots, \text{länge}(z) - \text{länge}(s)\}$
- Solange P nicht leer ist, tue folgendes:
  - Wähle ein x aus P aus
  - Entferne x aus P
  - $w := z$ ohne die ersten x Buchstaben
  - Prüfe, ob w mit der Zeichenkette s beginnt
  - Falls ja, terminiert der Algorithmus mit "ja"
- Wenn keine Position mehr in der Menge P ist: terminiere mit der Ausgabe "nein".

Frage

Warum ist der Algorithmus nicht-deterministisch?

Begriffsklärung: Determiniert

Ein Algorithmus heißt **determiniert**, wenn er bei gleichen zulässigen Eingabewerten stets das gleiche Ergebnis liefert.

Andernfalls heißt er **nicht-determiniert**.

Beispiele:

- Jeder Algorithmus, der eine math. Funktion berechnet, ist determiniert.
- Der nicht-deterministische Algorithmus des obigen Beispiels ist determiniert.

Anweisungen in Java

Begriffsklärung: Anweisung

Anweisungen (engl. *statements*) sind programmiersprachliche Beschreibungsmittel.

- **Einfache** Anweisungen beschreiben Aktionen.
- **Zusammengesetzte** Anweisungen beschreiben, wie mehrere Aktionen auszuführen sind.

Wiederholung: Ausdruck

Ausdrücke sind das Sprachmittel zur Beschreibung von Werten.

Ein **Ausdruck** (engl. **expression**) in Java ist (u.a.)

- ein Literal,
- ein Bezeichner (Variable, Name),
- die Anwendung einer Operation auf einen oder mehrere Ausdrücke,
- oder aufgebaut aus Sprachmitteln, die erst später behandelt werden.

Unterschied: Anweisung / Ausdruck

- Syntaktischer Unterschied:
  können an verschiedenen Stellen verwendet werden.
- Die Auswertung von *Ausdrücken* liefert ein *Ergebnis*.
- Die Ausführung von *Anweisungen verändert den Zustand*. Im Allgemeinen liefern sie kein Ergebnis.

Beispiele für Anweisungen in Java

- Variablendeklaration

```
Deklaration →
        Typ   ≪ Bezeichner ≫  ;
      | Typ   ≪ Bezeichner ≫   =   Ausdruck;

Typ → PrimitiverTyp  |  ≪ Bezeichner ≫

PrimitiverTyp →
     byte | short | int | long | float
   | double | char | boolean
```

- Zuweisungen[1]

```
Zuweisung → ≪ Bezeichner ≫   =   Ausdruck  ;
```

[1] In Java ist eine Zuweisung syntaktisch ein Ausdruck, liefert also einen Wert und zwar das Ergebnis der Auswertung von der rechten Seite der Zuweisung.

Begriffsklärung: L-/R-Wert

In einer Zuweisung (und anderen Sprachkonstrukten) kann eine Variable `v` mit zwei Bedeutungen vorkommen:

$$a = 7 + a;$$

L-Wert R-Wert

1. Das Vorkommen links vom Zuweisungszeichen meint die Variable. Man spricht vom L-**Wert** (engl. l-*value*) des Ausdrucks v .
2. Das Vorkommen rechts vom Zuweisungszeichen meint den in v gespeicherten Wert. Man spricht vom R-**Wert** (engl. r-*value*) des Ausdrucks v.

Anweisungsblöcke

Ein **Anweisungsblock (engl. block statement)** ist eine Liste bestehend aus Deklarationen und Anweisungen.

Syntax in Java:

```
Anweisung → { DeklAnweisListe }

DeklAnweisListe → ε
                | Deklaration DeklAnweisListe
                | Anweisung DeklAnweisListe
```

Semantik:

Stelle den Speicherplatz für die Variablen bereit und führe die Anweisungen der Reihe nach aus.

Beispiel

Anweisungsblock:

```
{ int i; i = 7; int a; a = 27 % i; }
```

Üblicherweise schreibt man Deklarationen und Anweisungen untereinander, so dass ein Textblock entsteht. Anweisungen in einem Block werden in der Regel eingerückt, um die Lesbarkeit zu verbessern.

```
{
   int i;
   int x;
   i = 34;
   x = i * 5;
}
```

Kontrollfluss: Verzweigungen und Schleifen

Bedingte Anweisung

Syntax in Java:

Anweisung → `if` `(` Ausdruck `)` Anweisung

Semantik:
Werte den boolschen Ausdruck aus.
Ist das Ergebnis `true`, führe die `Anweisung` aus.

Beispiel:

```
// Absolutwert
if (x < 0) { x = -x; }

// Sortieren von x und y: x soll kleiner als y sein
if (x > y) {
  int t = x;
  x = y;
  y = t;
}
```

Fallunterscheidung

Syntax in Java:

```
Anweisung →
    if ( Ausdruck )  Anweisung
    else Anweisung
```

Semantik:
Werte den boolschen Ausdruck aus. Ist das Ergebnis `true`, führe die erste Anweisung, andernfalls die zweite Anweisung aus.

Beispiel: Münzwurf

```
if (Math.random() < 0.5) {
  System.out.println("Kopf");
} else {
  System.out.println("Zahl");
}
```

Math.random() liefert eine Zufallszahl aus dem Bereich [0.0, 1.0)

Beispiel: Maximumsberechnung

Beispiel: Maximum zweier Zahlen

```
// Maximum von x und y
int x = ...;  // Initialisieren von x
int y = ...;  // Initialisieren von y

int max;
if (x > y) {
  max = x;
} else {
  max = y;
}
```

Frage

Wie ermittelt man das Maximum dreier Zahlen (x,y,z)?

Beispiel: Maximum dreier Zahlen

Eine mögliche Implementierung:

```
int x = ...;  // Initialisieren von x
int y = ...;  // Initialisieren von y
int z = ...;  // Initialisieren von z

int max;
if (x > y) {
  if (x > z) {
    max = x;
  } else {
    max = z;
  }
} else {
  if (y > z) {
    max = y;
  } else {
    max = z;
  }
}
```

Schleifenanweisungen

Schleifenanweisungen (engl. *loop statements*) steuern die iterative, d.h. *wiederholte Ausführung* von Anweisungen/Anweisungsblöcken.

Wir betrachten hier die folgenden Schleifenanweisungen:

- while-Anweisung
- do-Anweisung
- Zählanweisung (for-Anweisung) ($\Rightarrow$ nächste Vorlesung)

while-Anweisung

Syntax in Java:

```
Anweisung →
    while( Ausdruck ) Anweisung
```

Semantik:
Werte den boolschen Ausdruck aus, die sogenannte **Schleifenbedingung**.
Ist die Bedingung erfüllt, führe die Anweisung aus, den sogenannten **Schleifenrumpf**, und wiederhole den Vorgang.
Andernfalls beende die Ausführung der Schleifenanweisung.

Beispiel: ggt

Aus der Prozedur zur Berechnung des ggT:

```
while (m > 0)
{
    int v = n % m;
    n = m;
    m = v;
}
```

```
while (m > 0) {
    int v = n % m;
    n = m;
    m = v;
}
```

Bemerkung:

Der Schleifenrumpf ist in den meisten Fällen ein Anweisungsblock.
Syntaktisch korrekt ist aber auch z.B.:

```
while (true)  System.out.println(i);
```

Beispiel: Berechnung von Zweierpotenzen

```
/*
* Berechnet die Tabelle aller Zweierpotenzen,
* die kleiner als 2^n sind.
* Der Wert von n wird dabei als Programmparameter uebergeben.
*/

public class Zweierpotenz {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    int v = 1;
    int i = 0;

    while (i < n) {
      System.out.println(i + " " + v);
      v = 2 * v;
      i = i + 1;
    }
  }
}
```

Terminierung

Wird die Ausführung der Schleife irgendwann abgebrochen?

1. Welche Werte müssen die Variablen in der Schleifenbedingung annehmen, damit diese nicht mehr erfüllt ist?
2. Wie verändern sich die Werte dieser Variablen in jedem Schleifendurchlauf?
3. Wird die Schleifenbedingung daher irgendwann nicht mehr erfüllt?

Hilfestellung: Welche Werte nehmen die Variablen an, nachdem die Schleife das erste Mal, das zweite, etc., das letzte Mal durchlaufen wird?

Beispiel: Zweierpotenz

1. Die Schleifenbedingung ist nicht mehr erfüllt, wenn die Variable `i` größer oder gleich der Eingabe `n` ist.
2. In jedem Schleifendurchlauf wird der Wert der Variablen `i` um 1 erhöht; der Wert von `n` verändert sich nicht.
3. Daher ist `i` irgendwann größer als `n` und die Schleifenbedingung ist nicht mehr erfüllt.
4. Das Programm terminiert also für alle Eingaben.

```
while (i < n) {
  System.out.println(i + " " + v);
  v = 2 * v;
  i = i + 1;
}
```

Beispiel: ggT

1. Die Schleifenbedingung ist nicht mehr erfüllt, wenn die Variable `m` kleiner oder gleich 0 ist.
2. Nach jedem Schleifendurchlauf wird `m` auf den alten Wert von `n % m` gesetzt. Dies liefert einen Wert zwischen `0` und `m-1` (da `m > 0`).
3. Daher wird der Wert von `m` in jedem Durchlauf mindestens um `1` kleiner und erreicht irgendwann den Wert 0.
4. Das Programm terminiert also für alle Eingaben.

```
while (m > 0) {
  int v = n % m;
  n = m;
  m = v;
}
```

do-Anweisung

Syntax in Java:

```
Anweisung →
   do Anweisung while( Ausdruck ) ;
```

Semantik:

Führe die Anweisung aus.
Werte danach den booleschen Ausdruck aus.
Ist die Bedingung erfüllt ist, wiederhole den Vorgang.
Andernfalls beende die Ausführung der Schleifenanweisung.

Sprung- und Auswahlanweisungen

Sprunganweisungen

Sprunganweisungen (engl. *jump statements*) legen eine Fortsetzungsstelle der Ausführung fest, die möglicherweise weit von der aktuellen Anweisung entfernt liegt.

Wir betrachten hier nur Sprünge, die der Programmstruktur folgen.

Abbruchanweisung

Syntax in Java:

Anweisung → `break ;`

Semantik:
Die Ausführung wird mit der Anweisung fortgesetzt, die nach der umfassenden Schleife oder Auswahlanweisung kommt.

Typische Einsatzgebiete:

1. In Auswahlanweisung: siehe unten.
2. In Schleifenanweisungen:

```
while (...) {
    ...
    if (...) {
      break;
    }
    ...
}
```

Auswahlanweisung

Auswahlanweisungen erlauben es, in Abhängigkeit vom Wert eines Ausdrucks direkt in einen von endlich vielen Fällen zu verzweigen.

Beispiel:

```
String eingabe = args[0];
switch (eingabe) {
  case "a":
    System.out.println("A wie Apfel");
    break;
  case "b":
    System.out.println("B wie Banane");
    break;
  case "c":
    System.out.println("C wie Clementine");
    break;
  default:
    System.out.println("Falsches Eingabezeichen");
}
```

Übersicht: Anweisungen I

```
Anweisung →
      { DeklAnweisListe }
    | Zuweisung
    | if ( Ausdruck ) Anweisung
    | if ( Ausdruck ) Anweisung else Anweisung
    | while ( Ausdruck ) Anweisung
    | do Anweisung while ( Ausdruck ) ;
    | break ;
    | switch ( Ausdruck ) { FallListe }

DeklAnweisListe →
      Deklaration DeklAnweisListe
    | Anweisung DeklAnweisListe
    | ϵ

Deklaration →
      Typ  ≪ Bezeichner ≫ ;
    | Typ  ≪ Bezeichner ≫  =  Ausdruck;

Typ → PrimitiverTyp | ≪ Bezeichner ≫
```

Übersicht: Anweisungen II

```
PrimitiverTyp →
    byte | short | int | long | float | double | char |
  boolean

Zuweisung → ≪ Bezeichner ≫  =  Ausdruck ;

FallListe →
    Fall FallListe
  | ϵ

Fall →
    case Ausdruck : DeklAnweisListe
  | default : DeklAnweisListe
```