Universität Bremen

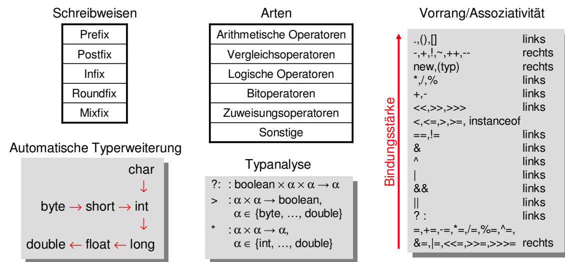
Anweisungen, Verzweigungen und Schleifen

Thomas Röfer

Anweisungen
Verzweigungen
Schleifen
Abbruch/Fortsetzung/Rückgabe
Zusicherung
Ausnahmen

Universität Bremen

Rückblick „Operatoren und Ausdrücke“



Universität Bremen

Anweisungen

- **Leere Anweisung**
  - ;
- **Variablendeklaration**
  - *typ bezeichner* [= *ausdruck*] { , *bezeichner* [= *ausdruck*] } ;
- **Zuweisungen**
  - *l-wert* [*operator*] = *ausdruck* ;
- **Erhöhen/verringern einer Variablen**
  - **++** *bezeichner*
  - **--** *bezeichner*
  - *bezeichner* **++**
  - *bezeichner* **--**
- **Funktionsaufruf**
  - [*ausdruck* .] *bezeichner* ([*ausdruck* { , *ausdruck* }]) ;
- **Zusicherung**
  - **assert** *bedingung* [: *ausdruck*] ;

z. B.
```
;;;;;;;;;;

int i = 0, j = 2;

i = j * 2;

++i;
--i;
i++;
i--;

System.out.println(i);

assert i >= 0;
```


Universität Bremen

Kontrollanweisungen

- **Verzweigungen**
  - **if (** *bedingung* **)** *anweisung* [**else** *anweisung*]
  - **switch (** *ausdruck* **) {** { (**case** *konstausdruck* | **default**) **:** { *anweisung* } } **}**
- **Schleifen**
  - **while (** *bedingung* **)** *anweisung*
  - **do** *anweisung* **while (** *bedingung* **) ;**
  - **for (** [*initialisierungen*] **;** [*bedingung*] **;** [*aktualisierungen*] **)** *anweisung*
- **Ausnahmen fangen**
  - **try {** { *anweisung* } **}**
    { **catch (** *typ bezeichner* **) {** { *anweisung* } **}** } [**finally {** { *anweisung* } **}**]
- **Marke, Abbruch, Fortsetzung, Rückgabe, Ausnahme**
  - *bezeichner* **:**
  - **break** [*bezeichner*] **;**
  - **continue** [*bezeichner*] **;**
  - **return** [*ausdruck*] **;**
  - **throw** *ausdruck* **;**
- **Block**
  - **{** { *anweisung* } **}**

Universität Bremen

Verzweigung

- **EBNF**
  - **if (** *bedingung* **)** *anweisung* [**else** *anweisung*]
- **Bedeutung**
  - Wenn die Bedingung wahr ist, wird die Anweisung (oder der Block) hinter der if-Bedingung ausgeführt
  - Falls sie falsch ist und ein else-Zweig angegeben wurde, wird dieser ausgeführt
- **Anmerkung**
  - Ein else-Zweig gehört immer zum zuletzt geöffneten if.

Universität Bremen

Verzweigung – Beispiele

```
if(a == 1)
  System.out.println("a ist eins");
```

```
if(a == 1)
  System.out.println("a ist eins");
else
  System.out.println("a ist nicht eins");
```

```
if(a == 1)
  if(b == 1)
    System.out.println("a und b sind eins");
else
  System.out.println("a ist nicht eins");
```

→

```
if(a == 1)
{
  if(b == 1)
    System.out.println("a und b sind eins");
}
else
  System.out.println("a ist nicht eins");
```

```
if(a == 1)
  System.out.println("a ist eins");
else if(a == 2)
  System.out.println("a ist zwei");
else if(a == 3)
  System.out.println("a ist drei");
else
  System.out.println("a ist etwas anderes");
```

Universität Bremen

Mehrfachverzweigung

- **EBNF**
  - **switch (** *ausdruck* **) {** { (**case** *konstausdruck* | **default**) **:** { *anweisung* } } **}**
- **Bedeutung**
  - Der hinter switch stehende Ausdruck wird ausgewertet
  - Dann wird zu der case-Marke gesprungen, die den gleichen Wert wie der Ausdruck hat
  - Falls hinter keinem case ein passender Wert angegeben wurde und eine default-Marke vorhanden ist, wird zu dieser gesprungen
- **Anmerkungen**
  - Der Ausdruck muss vom Typ int (oder automatisch erweiterbar) oder ein Aufzählungstyp (enum) sein
  - Hinter case können nur Kompilierzeitkonstanten von kompatiblem Typ stehen
  - Es darf maximal eine default-Marke angeben werden
  - Die Ausführung wird auch über Marken hinweg fortgesetzt, entweder bis zum Ende der switch-Anweisung oder bis zur Anweisung break

Universität Bremen

Mehrfachverzweigung – Beispiel 1

```
if(a == 1)
  System.out.println("a ist eins");
else if(a == 2)
  System.out.println("a ist zwei");
else if(a == 3 || a == 4)
  System.out.println("a ist drei oder vier");
else
  System.out.println("a ist etwas anderes");
```

In switch-Anweisungen müssen die einzelnen case-Blöcke fast immer mit einem break abgeschlossen werden!

```
switch(a)
{
  case 1:
    System.out.println("a ist eins");
    break;
  case 2:
    System.out.println("a ist zwei");
    break;
  case 3:
  case 4:
    System.out.println("a ist drei oder vier");
    break;
  default:
    System.out.println("a ist etwas anderes");
}
```


Universität Bremen

Mehrfachverzweigung – Beispiel 2

```
class Calendar
{
  enum Month {JANUARY, FEBRUARY, MARCH,
    APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
    OCTOBER, NOVEMBER, DECEMBER}

  static int daysOfMonth(Month month, int year)
  {
    switch(month)
    {
      case JANUARY:
      case MARCH:
      case MAY:
      case JULY:
      case AUGUST:
      case OCTOBER:
      case DECEMBER:
        return 31;
      case APRIL:
      case JUNE:
      case SEPTEMBER:
      case NOVEMBER:
        return 30;
      case FEBRUARY:
        return year % 4 == 0 &&
          year % 100 != 0 ||
          year % 400 == 0 ? 29 : 28;
      default:
        return 0;
    }
  }
```

- Calendar.daysOfMonth(Calendar .Month.FEBRUARY, 2006)

Universität Bremen

Abweisende Schleife

- **EBNF**
  - **while (** *bedingung* **)** *anweisung*
- **Bedeutung**
  - Wenn die Bedingung wahr ist, wird die Anweisung (bzw. der Anweisungsblock) ausgeführt
  - Dies wird solange wiederholt, bis die Bedingung nicht mehr wahr ist
- **Anmerkungen**
  - Falls die Bedingung bereits zu Anfang falsch ist, wird die Anweisung (bzw. der Anweisungsblock) niemals ausgeführt
  - Die Anweisung (bzw. der Anweisungsblock) sollte die in der Bedingung verwendeten Variablen verändern, sonst terminiert die Schleife nicht (es sei denn, durch break, continue, return oder throw)

```
while(i > 0)
{
  System.out.println(100 / i);
  --i;
}
```


Universität Bremen

Annehmende Schleife

- **EBNF**
  - **do** *anweisung* **while (** *bedingung* **) ;**
- **Bedeutung**
  - Die Anweisung (bzw. der Anweisungsblock) wird ausgeführt
  - Danach wir die Bedingung getestet
  - Ist sie wahr, werden die Anweisung (bzw. der Anweisungsblock) und der Bedingungstest wiederholt, solange, bis die Bedingung nicht mehr wahr ist
- **Anmerkungen**
  - Falls die Bedingung bereits zu Anfang falsch ist, wird die Anweisung (bzw. der Anweisungsblock) dennoch einmal ausgeführt
  - Die Anweisung (bzw. der Anweisungsblock) sollte die in der Bedingung verwendeten Variablen verändern, sonst terminiert die Schleife nicht (es sei denn, durch break, continue , return oder throw)

```
do
{
  System.out.println(100 / i);
  --i;
}
while(i > 0);
```

Fehler bei i = 0 zu Beginn!

Universität Bremen

Zählschleife

- **Motivation**
  - Aus der Mathematik: $\sum_{i=1}^{n} v_i$
  - In BASIC: **FOR** *variable* **=** *anfangswert* **TO** *endwert*
- **EBNF**
  - **for (** [*initialisierungen*] **;** [*bedingung*] **;** [*aktualisierungen*] **)** *anweisung*
  - *initialisierungen* = [*typ*] *zuweisung* { **,** *zuweisung* }
  - *aktualisierungen* = *aktualisierung* { **,** *aktualisierung* }
  - *aktualisierung* = (**++** | **--**) *bezeichner* | *bezeichner* (**++** | **--**) | *zuweisung*
- **Bedeutung**
  - Zuerst wird die Initialisierung ausgeführt
  - Werden dabei Variablen deklariert, endet ihre Lebenszeit nach Ende der Schleife
  - Ist die Bedingung wahr, werden erst die Anweisung (bzw. der Anweisungsblock) und dann die Aktualisierungen ausgeführt
  - Dann wird wieder mit der Überprüfung der Bedingung fortgefahren

```
for(int i = 10; i > 0; --i)
  System.out.println(100 / i);
```

Universität Bremen

Zählschleife – Beispiele

```
static void primes(int n)
{
  boolean[] sieve = new boolean[n + 1];
  for(int i = 2; i <= n; ++i)
    if(!sieve[i])
    {
      System.out.println(i);
      for(int j = 2 * i; j <= n; j += i)
        sieve[j] = true;
    }
}
```

```
for(;;)
{ // …
  if(bedingung) break;
  //…
}
```

```
static int search(String[] a, String b)
{
  int i;
  for(i = 0; i < a.length && !a[i].equals(b); ++i)
    ;
  return i < a.length ? i : -1;
}
```

```
static int search(String[] a, String b)
{
  int i;
  for(i = a.length; i-- > 0 && !a[i].equals(b);)
    ;
  return i;
}
```


Universität Bremen

Zählschleife über Reihungen

- **Motivation**
  - Oft benötigt man Operationen, die alle Elemente eines Arrays betreffen
- **EBNF**
  - **for (** *typ bezeichner* **:** *bezeichner* **)** *anweisung*
- **Bedeutung**
  - Schleife läuft über alle Elemente der Reihung von Index 0 bis Index *length* – 1
  - Jedes Element wird vor Schleifendurchlauf an Platzhalter zugewiesen
  - Veränderungen am Platzhalter wirken sich nicht auf die Elemente der Reihung aus
- **Hinweis**
  - Funktioniert auch mit Klassen, die die Schnittstelle *Iterable<T>* implementieren

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

```
int sum(int[] a)
{
  int s = 0;
  for(int e : a)
    s += e;
  return s;
}
```


Universität Bremen

Analogien zwischen Schleifen

```
while(bedingung)
  anweisung
```
≙
```
if(bedingung)
  do
    anweisung
  while(bedingung);
```

```
do
  anweisung
while(bedingung)
```
≙
```
boolean first = true;
while(first || bedingung)
{
  anweisung
  first = false;
}
```

```
for(initialisierungen; bedingung; aktualisierungen)
  anweisung
```
≙
```
{
  initialisierungen;
  while(bedingung)
  {
    anweisung
    aktualisierungen; // ',' → ';'
  }
}
```


Universität Bremen

Abbruch-/Fortsetzungsanweisung

- **Abbruchanweisung**
  - EBNF
    - ***break*** [bezeichner] **;**
  - Bedeutung
    - *break ohne Angabe einer Marke setzt die Ausführung hinter der aktuellen Schleife/switch-Anweisung fort*
- **Fortsetzungsanweisung**
  - EBNF
    - ***continue*** [bezeichner] **;**
  - Bedeutung
    - *continue ohne Angabe einer Marke leitet den nächsten Schleifendurchlauf ein*

```
while(true)
{
  if(i <= 0)
    break;
  else
    System.out.println(100 / i);
}
```

```
for(int i = 10; i >= -10; --i)
{
  if(i == 0)
    continue;
  else
    System.out.println(100 / i);
}
```


Universität Bremen

Abbruch-/Fortsetzung mit Marke

- **EBNF**
  - *bezeichner* **:**
- **Bedeutung**
  - Vor nahezu jeder Anweisung kann eine Marke stehen (nur vor Deklarationen nicht)
  - break mit Angabe einer Marke setzt die Ausführung hinter der markierten Anweisung (Block) fort
  - continue mit Angabe einer Marke leitet den nächsten Durchlauf der markierten Schleife ein

```
OuterLoop:
for(int x = 0; x < 10; ++x)
{
  for(int y = 0; y < 10; ++y)
  {
    if(a[x][y] == 42)
      break;
    else if(a[x][y] == 43)
      break OuterLoop;
  }
}
```

```
OuterLoop:
for(int x = 0; x < 10; ++x)
{
  for(int y = 0; y < 10; ++y)
  {
    if(a[x][y] == 42)
      continue;
    else if(a[x][y] == 43)
      continue OuterLoop;
  }
}
```


Universität Bremen

Rückgabeanweisung

- **EBNF**
  - **return** [*ausdruck*] **;**
- **Bedeutung**
  - Die return-Anweisung beendet die Ausführung der aktuellen Funktion und kehrt zum Aufrufer zurück
- **Anmerkungen**
  - Der Typ des Ausdrucks hinter return muss kompatibel zum Rückgabetyp der Funktion sein
  - Ist der Rückgabetyp einer Funktion void, darf hinter return kein Ausdruck stehen
  - Ist der Rückgabetyp nicht void, muss jeder mögliche Ausführungspfad durch die Funktion in einer return-Anweisung enden

```
static int factorial(int n)
{
  if(n > 0)
    return n * factorial(n - 1);
  else
    return 1.0;
}
```
Falscher Rückgabetyp!

```
static int factorial(int n)
{
  if(n > 0)
    return n * factorial(n - 1);
}
```
Fehlendes return!

Zusicherung

- **EBNF**
  - **assert** *bedingung* [**:** *ausdruck*] **;**
- **Bedeutung**
  - assert überprüft eine Zusicherung
  - Ist diese nicht gegeben, wird eine sog. Ausnahme erzeugt
- **Anmerkungen**
  - Man kann Java-Programm auch so übersetzen, dass assert-Anweisungen ignoriert werden (nicht mit BlueJ)
  - Im Modultest werden fehlgeschlagene Zusicherungen als Fehler vermerkt (nicht als „nicht bestanden“)

```
static int factorial(int n)
{
  assert n >= 0 && n <= 12 :
    "Falscher Wertebereich";
  if(n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```


Ausnahmen (Exceptions)

- **EBNF**
  - **try {** { *anweisung* } **}**
    { **catch (** *typ bezeichner* **) {** { *anweisung* } **}** } [**finally {** { *anweisung* } **}**]
  - **throw** *ausdruck* **;**
  - *funktionskopf* [**throws** *typ* { , *typ* }] *funktionsrumpf*
- **Bedeutung**
  - Erzeugen (throw)
    - *Exceptions dienen zum Mitteilen von Fehlerzuständen*
    - *An der Stelle, an der die Exception ausgelöst wird, wird die Ausführung des Programms unterbrochen und an anderer Stelle wieder aufgenommen*
  - Fangen (catch)
    - *Das Behandeln einer Exception bezeichnet man als „fangen“*
    - *Wird eine Exception nicht gefangen, wird das Programm abgebrochen und eine Fehlermeldung ausgegeben*
  - Erzwingen der Behandlung
    - *Durch die Angabe einer throws-Klausel hinter dem Funktionskopf werden Aufrufer einer Funktion gezwungen, die genannte Exception zu behandeln*
    - *Allerdings kann der Aufrufer ebenfalls eine throws-Klausel hinter seinem Funktionskopf verwenden, um die Behandlung an seinen Aufrufer weiterzuleiten*
  - „Geworfene“ Objekte müssen einen Typ haben, der von Throwable *abgeleitet* ist

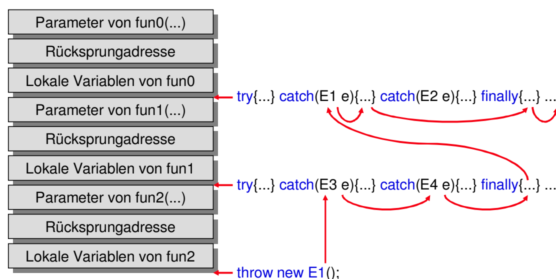
Merchandising mit Exceptions

```
class Buchführung
{
  static double berechneVKPreis(double ekpreis) throws Exception
  {
    final int gewinnspanne = 150,
              mwst = 16;
    if(ekpreis <= 0)
      throw new Exception("Negativer oder Gratiseinkaufspreis ist nicht erlaubt");
    if(ekpreis * 100 - (int) (ekpreis * 100) > 0)
      throw new Exception("Mehr als zwei Nachkommastellen sind nicht erlaubt");
    double zwischenpreis = ekpreis * (1 + gewinnspanne / 100.0);
    double ergebnis = zwischenpreis * ( 1 + mwst / 100.0);
    return (int) (ergebnis * 100) / 100.0;
  }
```


Merchandising mit Exceptions

```
  static void berechneVKPreise(double[] ekpreise)
  {
    for(int i = 0; i < ekpreise.length; ++i)
      try
      {
        System.out.println("EK-Preis " + ekpreise[i] + " € -> VK-Preis " +
                           berechneVKPreis(ekpreise[i]) + " €");
      }
      catch(Exception e)
      {
        System.out.println(e.getMessage() + ": " + ekpreise[i] + " €");
      }
  }
}
```


Der Weg einer Ausnahme



Die Exception-Hierarchie