universität innsbruck

Anweisungen

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Elementare Anweisungen

- Leere Anweisung

```
;
```

- Ausdrucksanweisung

```
Ausdruck;
```

  - Nur zielführend, wenn der Ausdruck einen Nebeneffekt hat
    - Zuweisung
    - Inkrement und Dekrement
    - Methodenaufruf und Instanzerzeugung
- return-Anweisung
  - Rücksprung zur aufrufenden Methode aus einer Methode
- Block
  - Zusammenfassung von Anweisungen, die nacheinander ausgeführt werden

```
{
  Anweisung1;
  Anweisung2;
  …
}
```

if-Anweisung

```
if (Bedingung1) {
    Anweisungsfolge1;
} else if (Bedingung2) {
    Anweisungsfolge2;
    …
} else {
    AnweisungsfolgeX;
}
```

- Ausführung
  - Die Bedingung (boolescher Ausdruck!) wird ausgewertet und dann entsprechend verzweigt.
  - Klammern (Block) können weggelassen werden, wenn nur eine Anweisung folgt (Achtung: geringere Lesbarkeit!).
  - `Der else`-Zweig bzw. `else if`-Zweige können weggelassen werden.

Beispiel (if-Anweisung)

```
public static int computeGrade(int pointsReached) {
    int grade;
    if (pointsReached < 50) {
        grade = 5;
    } else if (pointsReached < 65) {
        grade = 4;
    } else if (pointsReached < 75) {
        grade = 3;
    } else if (pointsReached < 85) {
        grade = 2;
    } else {
        grade = 1;
    }
    return grade;
}
```

Dangling else

- Bekanntes Problem bei der Programmierung
- Ein `else`-Zweig bei zwei `if`-Anweisungen
  → Wohin gehört der `else`-Zweig?

```
if (counter < 5)
    if (counter % 2 == 0)
        System.out.println("Position 1");
else
    System.out.println("Position 2");
```

- Auflösung durch Compiler
  - `else`-Zweig gehört zum textuell letzten freien `if` im selben Block!
  - `counter` mit Wert 7 führt zu keiner Ausgabe.
  - `counter` mit Wert 3 führt zur Ausgabe `"Position 2"`.

Always Use Brackets

```
if (counter < 5) {
    if (counter % 2 == 0) {
        System.out.println("Position 1");
    } else {
        System.out.println("Position 2");
    }
}
```

- Vorher:
  - Abarbeitungsabfolge nicht klar
- Nachher:
  - Abarbeitung durch Klammerung klar
  - Gilt nicht nur für `if`-Blöcke, sondern für jegliche Anweisungen

Bedingungsoperator

- Erlaubt den Wert eines Ausdrucks von einer Bedingung abhängig zu machen, ohne eine `if`-Anweisung zu verwenden.
- Form

```
Bedingung ? Ausdruck1 : Ausdruck2;
```

- Beispiel

```
int a, b;
…
int max = (a > b) ? a : b;
```

```
int a, b;
…
int max;
if (a > b) {
  max = a;
} else {
  max = b;
}
```

switch-Anweisung (1)

- Verzweigt den Kontrollfluss anhand des Wertes einer Bedingung zu einem oder mehreren Fällen (cases).
- Den Körper einer switch-Anweisung gibt es in den zwei Formen Anweisungsgruppen und switch-Regeln.

```
switch (Bedingung) {
    case Wert1: {Anweisungenfolge1; break;}
    case Wert2, Wert3: {Anweisungenfolge2; break;}
    …
    default: {AnweisungenfolgeX; break;}
}
```

```
switch (Bedingung) {
    case Wert1 -> Ausdruck1;
    case Wert2, Wert3 -> {Anweisungenfolge2;}
    …
    default -> AusdruckX;
}
```

switch-Anweisung (2)

- Der Typ der Bedingung muss eine Ganzzahl (`short`, `byte`, `int`, `char`), ein Wrapper-Typ einer Ganzzahl, ein Aufzählungstyp (enum) oder ein `String` sein.
- Kann beliebig viele `case`-Lables enthalten.
- `break` kann beim switch mit Anweisungsgruppen verwendet werden und sorgt dafür, dass die Ausführung **nicht** in das folgende `case`-Label weiterspringt.
- Das `default`-Label ist optional und wird ausgeführt, wenn kein `case`-Label ausgeführt werden kann.
- Ein case-Label hat eine oder mehrere case-Konstanten, welche durch ein Komma getrennt werden.
- Jeder Wert eines case-Labels muss ein konstanter Ausdruck oder ein Bezeichner einer enum-Konstante sein.

Beispiel (switch-Anweisung)

```
public static String getGradeDescription(int grade) {
    String gradeString;
    switch (grade) {
        case 1:
            gradeString = "Sehr Gut";
        case 2:
            gradeString = "Gut";
            break;
        case 3:
            gradeString = "Befriedigend";
            break;
        case 4:
            gradeString = "Genügend";
            break;
        default:
            gradeString = "Nicht Genügend";
            break;
    }
    return gradeString;
}
```

Avoid Switch Fallthrough

```
…
case 1:
    gradeString = "Sehr Gut";
    break;
…
```

- Vorher:
  - Abarbeitungsabfolge für switch wird ohne break nicht eingehalten
- Nachher:
  - Abarbeitung wie gedacht

switch-Ausdruck (1)

- Verzweigt den Kontrollfluss anhand des Wertes einer Bedingung zu einem Fall.
- Den Körper eines switch-Ausdrucks gibt es in den zwei Formen Anweisungsgruppen und switch-Regeln.

```
switch (Bedingung) {
    case Wert1: {Anweisungenfolge1; yield Resultat1;}
    case Wert2, Wert3: {Anweisungenfolge2; yield Resultat2;}
    …
    default: {AnweisungenfolgeX; yield ResultatX;}
}
```

```
switch (Bedingung) {
    case Wert1 -> Ausdruck1;
    case Wert2, Wert3 -> {Anweisungenfolge2; yield Resultat2;}
    …
    default -> AusdruckX;
}
```

switch-Ausdruck (2)

- Jeder Fall muss ein Ergebnis für das Resultat des switch-Ausdrucks liefern.
  - Switch-Regel:
    - Sofern ein Block verwendet wird, kann mit dem yield-Ausdruck ein Ergebnis bereitgestellt werden.
    - Wird eine Ausdruck verwendet, ist der Ausdruck das Ergebnis.
  - Anweisungsgruppen:
    - Das Ergebnis kann mit der yield-Anweisung bereitgestellt werden.
- Alle möglichen Werte der Bedingung müssen behandelt werden.

Beispiel (switch-Ausdruck)

```
public static String getGradeDescription(int grade) {
    return switch (grade) {
        case 1 -> "Sehr Gut";
        case 2 -> "Gut";
        case 3 -> "Befriedigend";
        case 4 -> "Genügend";
        default -> "Nicht Genügend";
    };
}
```

Schleifen

- Anweisungsfolgen beliebig oft wiederholen
- `while`-, `do-while`- und `for`-Schleifen bestehen aus
  - Schleifenkörper
  - Abbruchbedingung

`while`	`do-while`	`for`
• Zuerst Abbruchbedingung abfragen • Dann Schleifenkörper ausführen	• Zuerst Schleifenkörper ausführen • Dann Abbruchbedingung abfragen	• Geschlossene Schleife • Vorbereitung (Initialisierung) und Fortschalten ist Teil der Schleifenkonstruktion

while-Schleife

- Form

```
while (Bedingung) {
    Anweisungsfolge;
}
```

- Ablauf
  1. Die Bedingung (boolescher Ausdruck) wird ausgewertet.
  2. Ist sie wahr:
     - Der Schleifenkörper (Anweisungsfolge) wird ausgeführt.
     - Nach der Ausführung geht es wieder bei 1. weiter.
  3. Wenn die Bedingung falsch ist, dann wird die nächste Anweisung nach der Schleife ausgeführt.

Beispiel (while-Schleife)

```
int[] toSum = {1, 2, 3, 6, 10, 12};
int sum = 0;
int i = 0;
while (i < toSum.length) {
    sum += toSum[i];
    ++i;
}
```

Die Länge eines Arrays kann über das Datenelement length, welches jedes Array besitzt, ermittelt werden.

do-while-Schleife

- Form

```
do {
    Anweisungsfolge;
} while (Bedingung);
```

- Ablauf
  1. Der Schleifenkörper (Anweisungsfolge) wird ausgeführt.
  2. Die Bedingung (boolescher Ausdruck) wird ausgewertet.
  3. Ist sie wahr, dann Sprung nach 1.
  4. Ist sie falsch, dann wird die nächste Anweisung ausgeführt

for-Schleife

- Allgemeine Form

```
for (Initialisierung; Bedingung; Inkrementierung) {
    Anweisungsfolge;
}
```

- Aufbau
  - Initialisierung: Wird vor dem Betreten der Schleife ausgeführt.
  - Bedingung: Wird immer vor der Ausführung der Schleife überprüft.
  - Inkrementierung: Wird am Ende jedes Schleifendurchlaufs ausgeführt.
- Ablauf
  1. Initialisierung wird ausgeführt.
  2. Die Bedingung wird ausgewertet:
     1. Bedingung erfüllt (wahr)
        - Schleifenkörper (Anweisung bzw. Block) wird ausgeführt.
        - Inkrementierung wird ausgeführt.
        - Es geht wieder bei der Bedingung weiter.
     2. Bedingung nicht erfüllt: Die nächste Anweisung wird ausgeführt.

Beispiel (for-Schleife)

```
int[] toSum = {1, 2, 3, 6, 10, 12};
int sum = 0;
for (int i = 0; i < toSum.length; ++i) {
    sum += toSum[i];
}
```

Erweiterte for-Schleife

- Form

```
for (Datentyp Bezeichner: Ausdruck) {
    Anweisungsfolge;
}
```

- Erweiterte `for`-Schleife (foreach-Schleife)
  - Erleichtert den Umgang mit Arrays und listenartigen Datenstrukturen (z.B. Collections).
  - Wird in den entsprechenden Kapiteln über Arrays bzw. Collections noch genau besprochen.

Beispiel (erweiterte for-Schleife)

```
int[] toSum = {1, 2, 3, 6, 10, 12};
int sum = 0;
for (int value : toSum) {
    sum += value;
}
```

break und continue

- `break`
  - Führt zum sofortigen Ausstieg aus dem aktuellen Schleifendurchlauf (oder aus der `switch`-Anweisung).
  - Wird benutzt in: `while`-, `do-while`- und `for`-Schleifen sowie `switch`-Anweisungen
- `continue`
  - Überspringt die restlichen Anweisungen im aktuellen Schleifendurchlauf.
  - Wird benutzt in: `while`-, `do-while`- und `for`-Schleifen

- Verwendung
  - Bei `switch`–Anweisungen mit Anweisungsgruppen meist notwendig (`break`)!
  - In Schleifen nur sparsam verwenden!
    - Kann übermäßige Verschachtelung vermeiden!
    - Kann durch entsprechende Formulierung der Schleifenbedingung umgangen werden!

Ausblick - Anweisungen

- `assert`
  - Mit der `assert`-Anweisung kann eine Zusicherung überprüft werden.
- `throw`
  - Mit einer `throw`-Anweisung kann eine Ausnahme geworfen werden.
- `try`
  - Mit einer `try`-Anweisung können Ausnahmen gefangen werden (Exception Handling).