Selektion

In [**Programmablaufstrukturen**](http://freiheit.f4.htw-berlin.de/prog1/start/#programmablaufstrukturen) haben wir uns die drei Kontrollstrukturen angeschaut, die in Programmen vorkommen können:

* die [**Sequenz**](http://freiheit.f4.htw-berlin.de/prog1/start/#die-sequenz),
* die [**Iteration**](http://freiheit.f4.htw-berlin.de/prog1/start/#die-iteration) und
* die [**Selektion**](http://freiheit.f4.htw-berlin.de/prog1/start/#die-selektion).

Wir betrachten nun die Selektion genauer und schauen uns an, wie wir sie in Java umsetzen.

if...else

Bei der Selektion ist die Ausführung von Anweisungen von einer Bedingung abhängig. Angenommen, wir wollen erreichen, dass eine Zahl number halbiert wird, wenn sie gerade ist oder sie wird mit 3 multipliziert und 1 addiert, wenn sie ungerade ist. Es findet also eine Selektion der Anwesiungen statt, je nachdem ob number gerade ist oder nicht. Als "Pseudocode" sieht das so aus:

wenn (number ist gerade)

dann teile number durch 2

sonst

multipliziere number mit 3 und addiere 1

In Java gibt es dafür die if ... else-Anweisung:

if(number%2 == 0)

{

number = number / 2;

}

else

{

number = 3 \* number +1;

}

Das heißt, es wird zunächst eine Bedingung (ein Ausdruck vom Typ boolean) geprüft. Ist der Wert dieser Bedingung true, dann wird der erste Anweisungsblock ausgeführt. Ist der Wert der Bedingung jedoch false, dann wird der zweite Anweisungsblock (der nach dem else) ausgeführt. Es wird also genau einer der beiden Anweisungsblöcke ausgeführt - entweder der eine oder der andere (je nach Wert der Bedingung).

Noch ein Beispiel aus dem [**euklidischen Algorithmus**](http://freiheit.f4.htw-berlin.de/prog1/start/#beispiel-euklidischer-algorithmus):

if(a > b)

{

a = a - b;

}

else

{

b = b - a;

}

Innerhalb der Anweisungsblöcke können natürlich jeweils mehrere Anweisungen stehen. Als Information sei hier gesagt, dass es theoretisch möglich ist, bei nur einer Anweisung die Klammern wegzulassen, also z.B.

if(a > b) a = a - b;

else b = b - a;

Aber wir machen das nicht, da solche Programme nicht gut erweiterbar sind! Es gibt genügend Beispiele, in denen so etwas schief lief (z.B. [**bei Apple**](https://www.spiegel.de/netzwelt/web/goto-fail-apples-furchtbarer-fehler-a-955154.html)).

Die allgemeine Syntax einer solchen Selektion sieht also wie folgt aus:

if(bedingung)

{

/\*

\* Anweisungsblock, der ausgeführt wird,

\* wenn die bedingung true ist.

\* das können beliebig viele Anweisungen sein:

\* Sequenzen, Iterationen und/oder Selektionen

\*/

}

else

{

/\*

\* Anweisungsblock, der ausgeführt wird,

\* wenn die bedingung false ist.

\* das können beliebig viele Anweisungen sein:

\* Sequenzen, Iterationen und/oder Selektionen

\* dieser else-Block kann aber auch ganz weg-

\* gelassen werden - sehen wir gleich

\*/

}

Bedingungen

Bevor wir uns noch weitere Varianten der if...else-Anweisung anschauen, gehen wir nochmal darauf ein, was eine *Bedingung* ist.

Eine Bedingung ist ein logischer Ausdruck,

- d.h. der Datentyp des Wertes einer Bedingung ist `boolean`,

- d.h. der Wert einer Bedingung ist entweder `true` oder `false`

Wir betrachten einige Beispiele von Bedingungen (wir nehmen an, dass die Methoden isOdd(int) und notZero(int) existieren und ein boolean zurückgeben):

boolean cond1 = true; // kann genutzt werden als if(cond1) oder if(true)

boolean cond2 = cond1; // kann genutzt werden als if(cond2) oder if(cond1)

boolean cond3 = cond1 && true; // kann genutzt werden als if(cond3) oder if(cond1 && true)

boolean cond4 = (7 >= 4); // kann genutzt werden als if(cond4) oder if(7 >= 4)

boolean cond5 = (7 >= 4) && (3 < 5); // kann genutzt werden als if(cond5) oder if((7 >= 4) && (3 < 5))

boolean cond6 = cond5 || cond3; // kann genutzt werden als if(cond6) oder if(cond5 || cond3)

boolean cond7 = isOdd(3) && notZero(3); // kann genutzt werden als if(cond7) oder if(isOdd(3) && notZero(3))

Wichtig ist also, dass alle Werte, die vom Typ boolean sind, als Bedingungen verwendet werden können. Achten Sie auch darauf, dass die Ausdrücke selbst (also die Bedingungen) kein Semikolon am Ende haben - Ausdrücke sind keine Anweisungen.

**Beachte**

Vergleichen Sie logische Ausdrücke nicht mit true! Man sieht häufig so etwas wie isOdd(3) == true. Das ist unnötig! Angenommen, der Wert von isOdd(3) ist true, dann ist auch der Vergleich isOdd(3) == true true. Angenommen, der Wert von isOdd(3) ist false, dann ist auch der Vergleich isOdd(3) == true false. Das bedeutet, der Wert des Vergleiches entspricht exakt dem Wert von isOdd(3), also benutzen wir nur isOdd(3).

if...else ohne else-Block

Der else-Block ist nicht zwingend nötig. Es gibt Beispiele, in denen etwas getan werden soll, wenn eine bestimmte Bedingung gilt, aber wenn sie nicht gilt, dann muss auch nichts getan werden, z.B. (Methoden ausgedacht):

if(fileOpen(file))

{

closeFile(file);

}

if(connectionEstablished(database))

{

disconnect(database);

}

if(divisor == 0)

{

System.out.println("Divsion durch 0 nicht möglich!");

}

Verschachtelte Selektionen

Die Anweisungsblöcke sowohl im true-Zweig als auch im else-Zweig können beliebige Kontrollstrukturen enthalten:

* nur eine Anweisung oder
* Sequenzen von Anweisungen und/oder
* Iterationen und/oder
* Selektionen.

Ist eine Selektion innerhalb einer Selektion, spricht man auch von *verschachtelten* Selektionen. Angenommen, wir haben drei int-Variablen a, b, c und wollen diese der Größe nach sortieren (und wollen auch noch prüfen, ob die Werte evenzuell gleich sind):

// a, b, c vom Typ `int`

if(a > b)

{

if(b > c)

{

System.out.println("a > b > c");

}

else // c >= b

{

if(c > b)

{

if(a > c)

{

System.out.println("a > c > b");

}

else // c >= a

{

if(c > a)

{

System.out.println("c > a > b");

}

else // c == a

{

System.out.println("c = a > b");

}

}

}

else // c == b

{

System.out.println("a > b = c");

}

}

}

else // b >= a

{

if(b > a)

{

if(b > c)

{

if(a > c)

{

System.out.println("b > a > c");

}

else // c >= a

{

if(c > a)

{

System.out.println("b > c > a");

}

else // a == c

{

System.out.println("b > a = c");

}

}

}

else // c >= b

{

if(c == b)

{

System.out.println("b = c > a");

}

else // c > b

{

System.out.println("c > b > a");

}

}

}

else // b == a

{

if(b > c)

{

System.out.println("a = b > c");

}

else // c >= b

{

if(c == b)

{

System.out.println("a = b = c");

}

else // c > b

{

System.out.println("c > b = a");

}

}

}

}

Sie sehen, so etwas wird sehr schnell sehr unübersichtlich. Wir sollten versuchen, solche Verschachtelungen nur auf sehr geringe Verschachtelungstiefen zu beschränken. Die beste Möglichkeit, solche Verschachtelungstiefen zu vermeiden, besteht darin, die Bedingungen komplexer zu gestalten indem wir logische Operatoren verwenden, z.B. if(a > b) && (b > c)). Wir zeigen das mal am obigen Beispiel, in dem wir gar keine Verschachtelungstiefe haben:

// a, b, c vom Typ `int`

if(a > b && b > c)

{

System.out.println("a > b > c");

}

if(a > b && c > b && a > c)

{

System.out.println("a > c > b");

}

if(a > b && c > b && c > a)

{

System.out.println("c > a > b");

}

if(a > b && c > b && c == a)

{

System.out.println("a = c > b");

}

if(a > b && b == c)

{

System.out.println("a > b = c");

}

if(a == b && b == c)

{

System.out.println("a = b = c");

}

if(a == b && b > c)

{

System.out.println("a = b > c");

}

if(a == b && c > b)

{

System.out.println("c > a = b");

}

if(b > a && a > c)

{

System.out.println("b > a > c");

}

if(b > a && c > a && b > c)

{

System.out.println("b > c > a");

}

if(b > a && c > a && b == c)

{

System.out.println("b = c > a");

}

if(b > a && c == a)

{

System.out.println("b > c = a");

}

if(b > a && c > b)

{

System.out.println("c > b > a");

}

Übung

Das Problem mit komplexen Bedingungen besteht darin, dass diese recht schwer zu verstehen sind. Für komplexe Bedingungen sollten wir stets eigene Methoden definieren (die ein booleanzurückgeben) und die mit ihrem Namen verraten, was die Bedingung prüft. Darauf kommen wir später nochmal zurück.

**switch-Anweisung**

Die switch-Anweisung war in Java lange unglücklich gelöst. Die switch-Anweisung kann verwendet werden, wenn Sie viele Fallunterscheidungen haben. Wir geben zunächst ein Beispiel in der alten Form der switch-Anweisung, die immer noch funktioniert und die Sie so wahrscheinlich auch noch sehr häufig antreffen werden:

// monat vom Typ `int`

switch(monat)

{

case 1: System.out.println("Januar");

break;

case 2: System.out.println("Februar");

break;

case 3: System.out.println("März");

break;

case 4: System.out.println("April");

break;

case 5: System.out.println("Mai");

break;

case 6: System.out.println("Juni");

break;

case 7: System.out.println("Juli");

break;

case 8: System.out.println("August");

break;

case 9: System.out.println("September");

break;

case 10: System.out.println("Oktober");

break;

case 11: System.out.println("November");

break;

case 12: System.out.println("Dezember");

break;

default: System.out.println("kein Monat");

}

Betrachten wir die Anweisung zunächst etwas genauer. Die Fallunterscheidungen betreffen den Wert der int-Variablen monat. Das bedeutet schonmal, dass in den runden Klammern der switch()-Anweisung keine Bedingung steht, sondern ein Ausdruck, der von verschiedenen Datentypen sein kann, z.B. byte, short, int und String.

Der Ablauf einer switch()-Anweisung ist nun so, dass für den entsprechenden Wert der case gesucht wird, also z.B. wenn month den Wert 6 hat, dann ist case 6: der "Einsprungspunkt". Dort werden nun alle Anweisungen abgearbeitet, bis

* entweder ein break; kommt oder
* die switch()-Anweisung beendet ist.

break; stoppt also die Abarbeitung. Würde in unserem Beispiel kein break enthalten sien und wäre monthz.B. 6, dann würden alle Monate ab und inkl. Juli ausgegeben werden. Der default:-Fall ist dafür, falls keine der case passt. Der default:-Fall ist optional.

Dass die Anweisungen nach case x: nicht in Anweisungsblöcken stehen und dass die Verwendung von break; erforderlich ist, um die Ausführung von Anweisungen zu stoppen, macht diese alte Syntax der switch()-Anweisung unschön. Wir betrachten noch ein Beispiel in der alten Syntax, um den Unterschied zur neuen noch besser zu erläutern:

// monat vom Typ `int`

int anzahlTageImMonat = 0;

switch(monat)

{

case 1:

case 3:

case 5:

case 7:

case 8:

case 10:

case 12: anzahlTageImMonat = 31;

break;

case 4:

case 6:

case 9:

case 11: anzahlTageImMonat = 30;

break;

case 2: anzahlTageImMonat = 28;

}

Ab Java 12 wurde die switch()-Anweisung gründlich überarbeitet. Wir betrachten das so eben gezeigte Beispiel nun in der neuen Schreibweise:

int anzahlTageImMonat = switch(monat)

{

case 1, 3, 5, 7, 8, 10, 12 -> 31;

case 4, 6, 9, 11 -> 30;

case 2 -> 28;

default -> 0;

};

Mehrere Dinge fallen auf:

1. kann eine switch()-Anweisung nun auch als Ausdruck verwendet werden und somit z.B. einer Variablen einen Wert zu weisen (das machen wir hier)
2. hat sich die Schreibweise beim case geändert. Aus dem Doppelpunkt wurde ein Pfeil und es können mit einem case auch mehrere durch Komma getrennte Werte aufgeführt werden
3. und - besonders gut - wir benötigen kein breakmehr. Es wird immer nur der case ausgeführt. Die Abarbeitung der dort beschriebenen Anweisungen stoppt beim nächsten case bzw. bei default oder am Ende der switch()-Anweisung

**Success**

Wir haben 2 Möglichkeiten kennengelernt, die Selktion in Java zu implementieren. Die eine (und bedeutendste) Möglichkeit ist die if...else-Anweisung. Dort wird in Abhängigkeit vom Wahreheitswert einer Bedingung entweder der eine oder der andere Anweisungsblock ausgeführt. Der else-Block kann auch weggelassen werden. Innerhalb eines Anweisungsblockes können beliebige Kontrollstrukturen stehen: Sequenzen, Iterationen und Selektionen. Die zweite Möglichkeit ist die switch()-Anweisung, die seit Java 12 auch selbst ein Ausdruck sein kann. Wir haben die alte Syntax der switch()-Anweisung kritisch betrachtet und die neue Syntax kennengelernt.