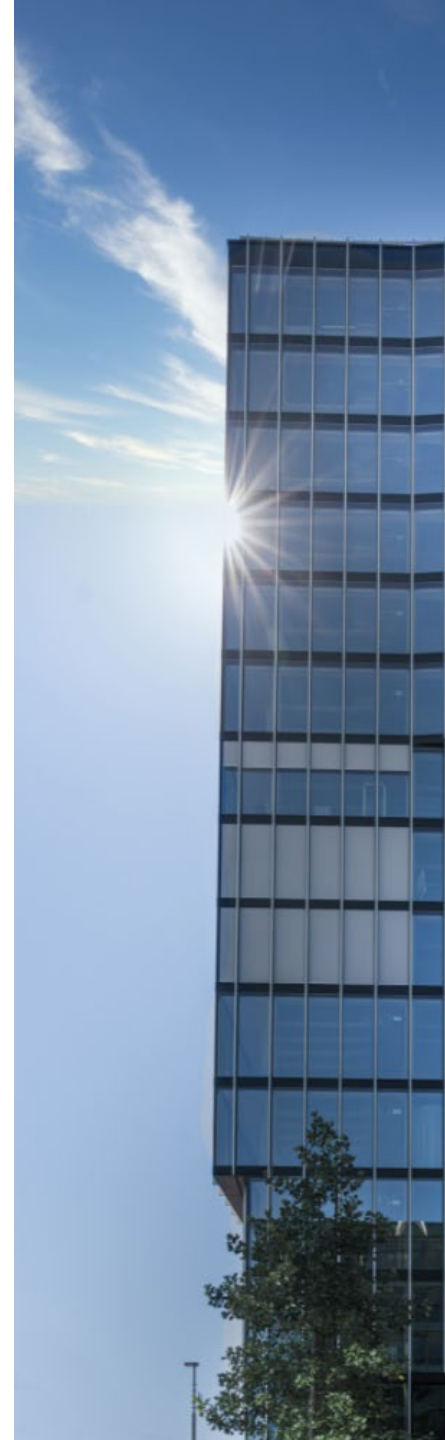


Objektorientierte Programmierung

# Selektionen

## Steuerung des Kontrollflusses

Roland Gisler



# Inhalt

- Selektion (Auswahl) mit **if-then-else**
- Bedingungen und bool'sche Operationen
- Exkurs: Logische Operatoren
- Empfehlungen zu **if**-Statements
- Problematik von verschachtelten **if**-Statements
- **else-if**-Statement
- **switch-Statement**
- Zusammenfassung

# Lernziele

- Sie können einfache Bedingungen und bool'sche Ausdrücke formulieren.
- Sie kennen das **if**-Statement.
- Sie kennen das **else if**-Statement.
- Sie kennen das **switch**-Statement und die **switch**-Expression.
- Sie kennen die Vor- und Nachteile der verschiedenen Selektionen und können gezielt die jeweils am Besten geeignete Variante auswählen.

# **Einfache Selektion mit `if`**

# Selektion (Auswahl)

- Bei der Programmierung erfolgen sehr viele Aktionen in Abhängigkeit von logischen Bedingungen.
- Zum Beispiel um eine Division durch Null zu verhindern:

**Wenn** `divisor` den Wert ungleich 0 hat,

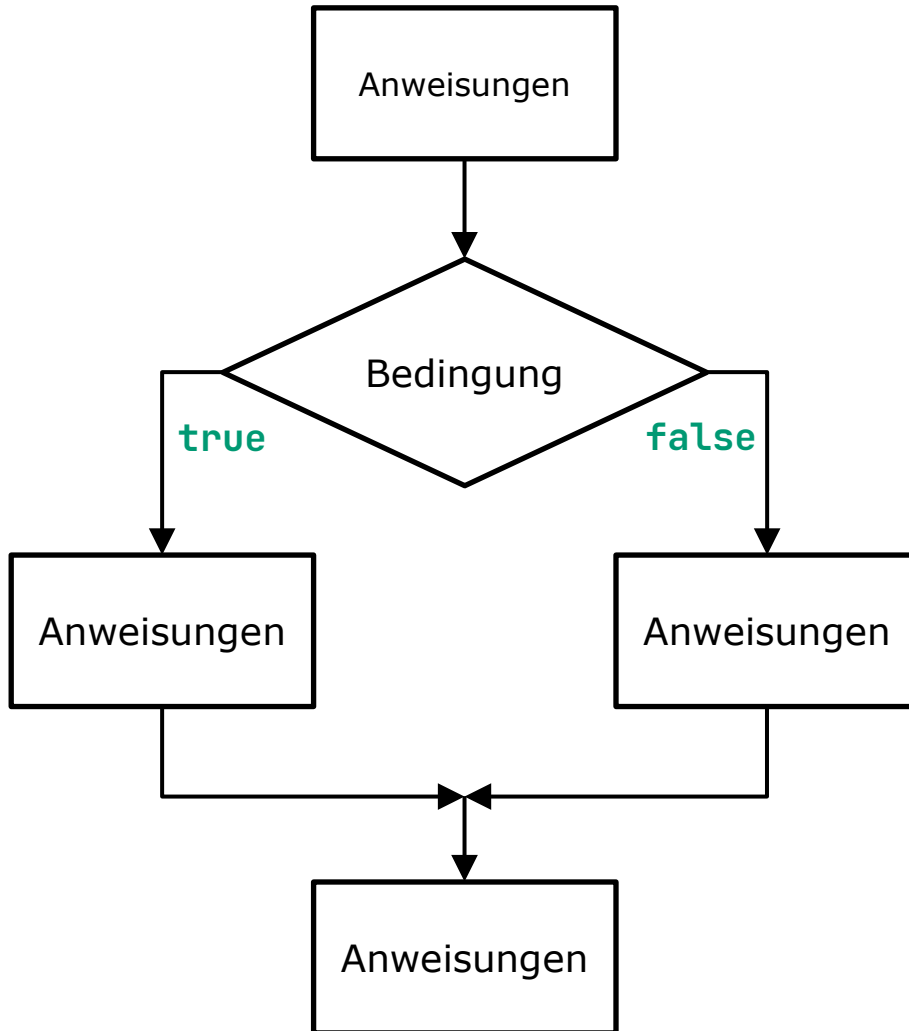
**Dann:** Berechne `quotient = dividend / divisor`

**Sonst:** Melde einen Fehler!

Es sind also zwei Dinge notwendig:

- Die Formulierung einer Selektion, z.B.:  
`if (<expression>) then { ... } else { ... }`
- Beispiel für eine logischen Bedingungen (expression):  
`divisor != 0`

# Alternative Programmflüsse



- Darstellung als Verzweigung mit Bedingung:
  - Hier gezeigt mit einem Flussdiagramm.
- Der **true**-Pfad entspricht dem **then**-Block, der **false**-Pfad in der Regel dem **else**-Block.
- Einer der Pfade könnte natürlich auch keine (leer) Anweisungen enthalten.
  - Natürlicherweise ist das meist der **else**-Block.

# Einfache Selektion in Java: if/else-Statements

```
if (<expression>) {    // Achtung: Bei Java kein "then"!
    <statements>       // Ausführung wenn Bedingung true.
} else {
    <statements>       // Ausführung wenn Bedingung false.
}
```

- Die Bedingung (expression) **muss** ein bool'sches Resultat ergeben.
- Die Bedingung muss in Klammern ( ... ) eingefasst sein.
- Tipp: Auch wenn in einem Anweisungsblock { ... } nur ein einziges Statement steht, verwendet man trotzdem **immer** die geschweiften Klammern.
  - Code wird dadurch wartungsfreundlicher.
- Das **else**-Statement (inkl. Anweisungsblock) ist **optional**!
  - Wird es nicht benötigt, lassen wir es vollständig weg.

# **Bedingungen und logische Operatoren**



# Bedingungen

- Eine Bedingung kann **wahr** oder **falsch** sein (erfüllt oder nicht erfüllt) → zwei mögliche Werte.
- Datentyp: **boolean**  
Werte: **true** (= wahr, erfüllt) und **false** (= falsch, nicht erfüllt).
- Eine häufige Variante Bedingungen zu formulieren, sind Vergleiche zweier numerischer Werte. Beispiele von Operatoren:

<	kleiner als	<=	kleiner oder gleich als
==	gleich (identisch)	!=	ungleich (nicht identisch)
>	grösser als	>=	grösser oder gleich als

- Einfache Beispiele:

(x == 5)

(a > b)

(d != c)

# Logische (bool'sche) Operatoren

- Operatoren für bool'sche Variablen oder Ausdrücke.
  - Ergebnis all dieser Operatoren ist wiederum ein bool'scher Wert.
- Damit lassen sich Einzelbedingungen logisch zu komplexeren Bedingungen verknüpfen.
- Übersicht der wichtigsten Operatoren:

<i>Op</i>	<i>Funktion</i>	<i>Kurzbeschreibung</i>
<b>&amp;&amp;</b>	<b>UND</b> (AND)	Beide Argumente müssen <b>true</b> sein.
<b>  </b>	<b>ODER</b> (OR)	Mindestens eines der Argument muss <b>true</b> sein.
<b>^</b>	<b>Exklusiv-ODER</b> (XOR)	Nur genau eines der Argument darf <b>true</b> sein.
<b>!</b>	<b>Negation</b> (NOT)	Negiert das Argument (Unäre Operation).

# **Kleiner Exkurs: Logische Operationen**

# Logisches UND (AND)

- Wahrheitstabelle:

1. Argument (a)	2. Argument (b)	Resultat (a && b)
false	false	false
false	true	false
true	false	false
true	true	true

- Operator: &&

- et-Zeichen (Kaufmanns-Und), Eingabe Windows: <Shift>+<6>

- Der Operator ist bei Java **optimiert**: Wenn das **erste** Argument **false** ist, wird das **zweite** Argument **nicht** mehr ausgewertet.

- ➔ Weil das Resultat dann nicht mehr **true** werden kann!

# Logisches ODER (OR)

- Wahrheitstabelle:

1. Argument (a)	2. Argument (b)	Resultat (a    b)
false	false	false
false	true	true
true	false	true
true	true	true

- Operator: ||

- Pipe-Zeichen (Unix), Eingabe Windows: <Alt Gr> + <7>

- Der Operator ist bei Java optimiert: Wenn das **erste** Argument **true** ist, wird das **zweite** Argument **nicht** mehr ausgewertet.  
➔ Weil das Resultat dann nicht mehr **false** werden kann.

# Logisches Exklusiv-ODER (Ex-OR)

- Wahrheitstabelle:

1. Argument (a)	2. Argument (b)	Resultat (a ^ b)
false	false	false
false	true	true
true	false	true
true	true	false

- Operator: ^
  - Zirkumflex-Zeichen, Eingabe bei Windows: <^> mit anschliessendem Space.
- Hier ist keine Optimierung möglich, es werden immer beide Argumente ausgewertet.

# Logisches NICHT (NOT, negation)

- Wahrheitstabelle:

<i>1. Argument (a)</i>	<i>Resultat (! a)</i>
<b>false</b>	<b>true</b>
<b>true</b>	<b>false</b>

- Operator: !

  - Ausrufezeichen

- Hinweis: Achten Sie auf saubere Klammersetzung, auch zur besseren Verständlichkeit, z.B. wenn ganze Ausdrücke negiert werden sollen.

# Das Gesetz von De Morgan

- Zwei gleichwertige Formulierungen:

$$\neg(a \ \&\& \ b) \ == \ \neg a \ || \ \neg b$$

$$\neg(a \ || \ b) \ == \ \neg a \ \&\& \ \neg b$$

- Einfach formuliert:
  - Eine **AND**-Operation lässt sich durch eine **OR**-Operation (und umgekehrt!) ersetzen, wenn man **gleichzeitig** sowohl die einzelnen **Argumente** als auch das **Resultat negiert**.
- Praktischer Nutzen:

Ermöglicht gegebenenfalls die Umformulierung von Ausdrücken in deutlich leichter verständliche, aber gleichwertige Varianten!



## Beispiel – Anwendung von De Morgan

- Es soll geprüft werden, ob ein Wert gültig (valid) ist.  
Anforderung: Das Resultat soll **false** sein, wenn der Wert (**value**) kleiner als **1** oder grösser als **6** ist.

- Direkt nach diesen Anforderungen umgesetzt ergibt das:

```
boolean valid = !((value < 1) || (value > 6))
```

- Speziell die Negation des Gesamtausdruckes ist unübersichtlich!
- Anwendung von De Morgan:  
Damit die Gesamtnegation entfällt, negieren wir die beiden Argumente und «drehen» den Operator.

- Ergebnis:

```
boolean valid = (value >= 1) && (value <= 6)
```

# **Zurück zum `if`-Statement**

# Empfehlungen zum `if`-Statement – 1



- Anweisungsblöcke **immer** mit geschweiften Klammern, sonst passieren bei Erweiterungen sehr schnell hässliche Fehler!
- Versuchen Sie, die Bedingung immer so klar und einfach wie möglich zu formulieren.
  - ➔ Das De Morgan'sche Gesetz kann helfen.
- Versuchen Sie, den «normalen» oder häufigeren Fall möglichst im **then**-Block zu implementieren, und Ausnahmen bzw. den selteneren Fall eher im **else**-Block.
  - **if/else**-Statements mit leerem **then**-Block sind verpönt!
- Leere **else**-Blöcke lassen Sie immer ganz weg.

## Empfehlungen zum if-Statement – 2



- Wenn Sie bereits eine Variable/Parameter vom Typ `boolean` haben, können Sie diese **direkt** als Expression verwenden!
  - Direkt mit `if (<boolean>) {...}`
  - Oder bei Negation mit `if (!<boolean>) {...}`
  - Vermeiden Sie `if (<boolean> == true) {...}` (dito `false`)
- Wenn Sie einen bool'schen Wert als Resultat benötigen, brauchen Sie kein `if`-Statement, sondern können die Expression direkt zuweisen bzw. verwenden:

**Nicht empfohlen:**

```
if (x < 3) {  
    return true;  
} else {  
    return false;  
}
```



**Besser:**

```
return (x < 3);
```

# Problematik von `if`-Statements

- Bei logisch etwas komplexeren Abläufen treten `if`-Statements sehr schnell ineinander verschachtelt auf:

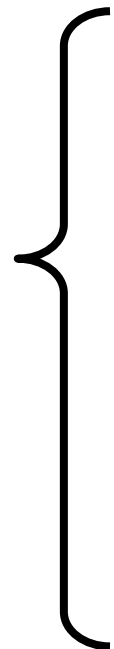
```
if (x > 5) {  
    doSomethingA();  
} else {  
    if (x > 4) {  
        doSomethingB();  
    } else {  
        if (x > 3) {  
            doSomethingC();  
        }  
    }  
}
```

- Problem: Schon ab drei Einrückungsebenen (ist hier der Fall) empfindet man den Code als sehr schwer verständlich.

# Vermeiden tiefer Verschachtelung von if-Statements

- Man begrenzt die Verschachtelung, in dem man tief verschachtelte **if**-Statements in eigenständige Methoden auslagert, die man dann aufruft.
- So lässt sich der Code von oben wie folgt vereinfachen:

```
if (x > 5) {  
    doSomethingA();  
} else {  
    doOther(x);  
}
```



```
private void doOther(int x) {  
    if (x > 4) {  
        doSomethingB();  
    } else {  
        if (x > 3) {  
            doSomethingC();  
        }  
    }  
}
```

- Herausforderung: Namensgebung der Methode(n) und Parameter.

# **else-if-Statement**

# Einsatz von `else-if`-Statement

- Java kennt ein speziell verkürztes `else-if`-Statement, mit welchem sich gegenseitig ausschliessende Optionen auf nur einer einzigen Einrückungsebene formulieren lassen.
- Gleiches Beispiel wie oben, nun aber mit `else-if`-Statement :

```
if (x > 5) {  
    doSomethingA();  
} else if (x > 4) {  
    doSomethingB();  
} else if (x > 3) {  
    doSomethingC();  
}
```

- Vorteil: Kann in manchen Fällen deutlich übersichtlicher sein!
- Nachteil: Nur bei voneinander unabhängig formulierten Optionen.



# **switch-Statement**

# Selektion auf Basis absoluter Werte: switch-Statement

- Java kennt neben **if-else** und **else-if** mit dem **switch**-Statement noch ein drittes Konstrukt für Selektionen.
- Im Unterschied zu den **if**-Statements ist es aber beschränkt auf den Vergleich von **absoluten Werten** (keine Bedingungen).
  - andere Programmiersprachen sind hier deutlich flexibler!
- Es funktioniert nur mit einer **eingeschränkten** Menge von **Datentypen**:
  - Primitive: **byte, short, char, int** (also **ohne long!**)
  - Klassen: **String** und Enumerations-Typen (folgen später)
- Somit nur geeignet für einfache Fallunterscheidungen auf Basis von einzelnen Werten!

# Beispiel 1: switch-Statement - Einfachselektion

- Liefert zu einer Tagnummer (`int value`) den Wochentag:

```
switch (value) {  
    case 1:  
        tag = "Montag";  
        break;  
    case 2:  
        tag = "Dienstag";  
        break;  
    ... <Zeilen aus Platzgründen entfernt>  
    case 6:  
        tag = "Samstag";  
        break;  
    case 7:  
        tag = "Sonntag";  
        break;  
    default:  
        tag = "nicht erlaubte Tagnummer";  
}
```

## **switch-Statement: break nicht vergessen!**

- Das **switch**-Statement zeigt ein spezielles Verhalten:  
Sobald ein **case** zutrifft, werden die Statements dieses, aber auch **aller folgenden** Fälle abgearbeitet!
  - Darum ist in den meisten Fällen zwingend am Ende jedes **case** ein **break**-Statement notwendig.
- Dieses auf den ersten Blick sonderbare Verhalten lässt sich auch bewusst und geschickt ausnutzen, wie das Beispiel auf der nächsten Seite zeigt.
- Es gibt Sprachen, in welchen das **switch**-Statement wesentlich leistungsfähiger ist als in Java, z.B. in Bezug auf nutzbare Datentypen oder die Angabe von Wertebereichen für die einzelnen Fälle.
- Seit Java 14 wurde das Switch-Statement erweitert!

## Beispiel 2: switch-Statement – mit «fall-through»

- Liefert zu einer Tagnummer (`int value`) die Art des Tages:

```
switch (value) {  
    case 1:  
    case 2:  
    case 3:  
    case 4:  
    case 5:  
        typ = "Arbeitstag";  
        break;  
    case 6:  
    case 7:  
        typ = "Wochenende";  
        break;  
    default:  
        typ = "nicht erlaubte Tagnummer";  
}
```

# **Seit Java 14: switch-Expressions**

# switch-Statement und switch-Expressions

- Erweiterung [JEP 361](#) wurde in Java **14** finalisiert.
  - Erster folgender LTS war Java **17**, aktuell stehen wir bei **21**.
- Das **switch**-Statement wurde funktional erweitert. Neu sind pro **case** auch mehrere Optionen möglich.
  - Diese Optionen werden durch Kommas separiert.
  - Labels (mit Doppelpunkt) mit «fall through» (**break** notwendig).
- Das **switch**-Statement kann neu auch als **Expression** (mit einem Rückgabewert) geschrieben werden.
  - Verkürzte Schreibweise mit Arrow-Labels (->), dann **kein** «fall through mehr! **yield** zur direkten Rückgabe aus Codeblöcken.
  - **switch**-Expressions müssen **vollständig** sein, d.h. alle möglichen Fälle/Werte müssen geprüft werden!
- Das sind alles vorbereitende Änderung für «pattern matching».

## Beispiel 1: switch-Statement - Mehrfachselektion

- Liefert zu einer Tagnummer (`int value`) die Art des Tages:

```
switch (value) {  
    case 1, 2, 3, 4, 5:           // Neu mit multi case  
        typ = "Arbeitstag";  
        break;                  // break notwendig  
    case 6, 7:  
        typ = "Wochenende";  
        break;  
    default:  
        typ = "Unerlaubte Tagnummer";  
}
```

- Das `default` ist in diesem Fall **optional**, da nicht alle `int`-Werte berücksichtigt werden müssen.
- Das `break` ist aber weiterhin **notwendig**, da wir ansonsten «fall through» hätten.



## Beispiel 2: switch-Expression – mit Arrow Labels

- Stark verkürzte Syntax mit Arrow-Labels (**ohne** «fall through»!):

```
String daytype = switch (value) {  
    case 1, 2, 3, 4, 5 -> "Arbeitsstag";  
    case 6, 7          -> "Wochenende";  
    default            -> "Unerlaubte Tagnummer";  
}
```

- **Expression:** `switch` liefert direkt das Resultat als Rückgabewert zurück (hier z.B. `String`).
- Das `default` ist in diesem Fall **zwingend**, da alle möglichen `int`-Werte berücksichtigt werden müssen.
- Ein `break` ist hingegen nicht mehr notwendig, da wir hier **kein** «fall through» mehr haben!

## Beispiel 3: switch-Expression – mit Codeblock und yield

- Pro **case** (oder **default**) sind auch Codeblöcke {...} möglich:

```
String daytype = switch (value) {  
    case 1, 2, 3, 4, 5 -> {  
        String name = "Arbeitstag(" + value + ")";  
        LOG.info("Es ist {}", name);  
        yield name;  
    }  
    case 6, 7           -> "Wochenende";  
    default             -> "Unerlaubte Tagnummer";  
}
```

- Bei Verwendung eines (optionalen) Codeblockes muss das Schlüsselwort **yield** genutzt werden, um die **switch**-Expression mit dem gewünschten Wert zu verlassen.
  - Analog zu einem **return** aus einer ganzen Funktion.

# Empfehlungen zu `switch`-Statement



- Das `switch`-Statement sollte möglichst **zurückhaltend** und nur in wohlüberlegten Situationen eingesetzt werden.
  - Manche Programmier\*innen stellen `switch` sogar auf die gleiche Ebene wie das legendär verpönte `goto`-Statement!
- Klassische `switch`-Statements blähen den Code sehr stark auf, mit zunehmender Optionsanzahl wird es noch schlimmer:
  - Es provoziert schlecht wart- und erweiterbaren Quellcode!
  - Es ist leicht, einen (unabsichtlichen) Fehler einzubauen.
- Es gibt elegante, objektorientierte Design(-muster), mit welchen sich `switch`-Statements vermeiden/reduzieren lassen.
  - Stichwort: Designpatterns, Strategy etc. → Modul **VSK**
- Die neue `switch`-Expression bringt hingegen in ausgewählten Fällen (z.B. «pattern matching») durchaus grosses Potential mit.

# Zusammenfassung

- Drei verschiedene Statements zur Steuerung des Kontrollflusses abhängig von Bedingungen:
  - **if**-Statement: **else**-Zweig ist optional.
  - **else-if**-Statement: kürzere Schreibweise, weniger Einrückung
  - **switch**-Statement: mehrere Alternativen, einfache Selektion
- Bei **if**-Statements die logische Bedingung möglichst einfach verständlich formulieren.
- Häufigere bzw. Normalfälle eher im **then**-Zweig abhandeln.
- **switch**-Statement zurückhaltend einsetzen, weil es produziert sehr schnell viel Code. ➔ Designpatterns.
  - Neue Alternative: **switch**-Expression wieder attraktiver.



**Fragen?**

Fragen bitte im  
**ILIAS-Forum**

