

Programmieren lernen mit JavaScript

Copyright © Gerd Wagner. 2006-2013. [Lehrstuhl Internet-Technologie](#), Brandenburgische Technische Universität Cottbus, [Some rights reserved](#).

Letzte Änderung am 17.4.2013.

Inhalt

- [Teil 1 Einführung](#)
- [Teil 2 Algorithmen und Programme](#)
- [Teil 3 Prozeduren und Funktionen](#)
- [Teil 4 Eingabe und Ausgabe in HTML-Formularen](#)
- [Teil 5 String-Operationen und reguläre Ausdrücke](#)

Teil 1 Einführung

Inhalt

- [Geschichte](#)
- [Ein erstes JavaScript-Programm](#)
- [Einfache Berechnungsprogramme](#)

Geschichte

Die Skriptprogrammiersprache *JavaScript* wurde 1995 von der damals führenden Browserhersteller-Firma *Netscape* eingeführt und später von der Organisation ECMA als Industriestandard definiert. Die aktuelle [JavaScript-Version 1.5](#) entspricht der Spezifikation ECMA-262, Edition 3. Alle Web-Browser (sowie einige weitere Softwarewerkzeuge wie z.B. Adobe Acrobat) können JavaScript-Programme mit Hilfe eines eingebauten JavaScript-Interpreters ausführen.

Obwohl der Name *JavaScript* eine Verwandtschaft zu der Programmiersprache *Java* suggeriert, gibt es bis auf eine gewisse Übereinstimmung in der Syntax der beiden Sprachen nicht viele Gemeinsamkeiten. Insbesondere ist JavaScript im Gegensatz zu Java keine wirklich objektorientierte Sprache, auch wenn sie Ansätze zur Objektorientierung hat.

JavaScript als Einstiegs-Programmiersprache

Dieses Tutorium ist kein JavaScript-Kurs, sondern eine Einführung in die Grundbegriffe und Techniken des Programmierens mit Hilfe der Sprache JavaScript. JavaScript bietet sich als Einstiegs-Programmiersprache an, weil JavaScript relativ einfach ist, einem viele Fehler verzeiht und eine schnelle Programmentwicklung erlaubt.

Ein erstes JavaScript-Programm

JavaScript-Code wird in ein HTML-Dokument mit Hilfe des `script`-Elements eingebettet, und zwar in der Regel innerhalb des `head`-Elements (alternativ kann auch eine separate JavaScript-Datei mit einem HTML-Dokument verknüpft werden). Ein JavaScript-Programm besteht aus einer Folge von Anweisungen, die der JavaScript-Interpreter in Maschinencode umsetzt, der dann auf dem Rechner des Anwenders ausgeführt wird. Jede JavaScript-Anweisung wird mit einem Strichpunkt ";" abgeschlossen.

Das folgende Beispiel zeigt, wie mit Hilfe des Bildschirmausgabe-Befehls `alert` der Text "Hallo Welt!" in einem Meldungsfenster auf dem Bildschirm ausgegeben wird:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de">
<head>
  <title>Einführung in die Programmierung</title>
  <meta charset="UTF-8" />
<script>
  //<![CDATA[
    alert("Hallo Welt!");
  //]]>
</script>
</head>
<body>
<p>Weiterer Text ...</p>
</body>
</html>
```

Testen Sie, wie Ihr Browser dieses Skript ausführt: [Ausführen/anzeigen](#)

Die Umklammerung des JavaScript-Programms durch `//<![CDATA[` und `//]]>` dient dazu, den JavaScript-Code vor der XHTML-Validierung abzuschotten, so dass keine Probleme mit XML-kritischen Zeichen wie `&` und `<` entstehen.

Einfache Berechnungsprogramme

Als erste Art von *Programmanweisung* (oder *Befehl*) behandeln wir *Wertzuweisungen*. Mit ihrer Hilfe kann man Variablen einen Wert zuweisen, um dann Berechnungen durchzuführen.

Datenwerte, Variablen und Wertzuweisungen

Datenverwaltung und *Rechnen* sind neben den Funktionen des *Steuerns* und der *Kommunikation* die Themen der Informatik. Man kann die Informatik als eine Herstellungswissenschaft zur Konstruktion von Datenverarbeitungs-, Rechen-, Steuerungs-, und Kommunikationsanlagen begreifen.

Beispiele für Datenwerte sind: 20061017, 3.1416, "Asamoah", und "Beckenbauer kandidiert erneut", also Zahlen und Zeichenfolgen ("Strings") sowie Texte. Mit Hilfe von Computersystemen ist es möglich, große Datenmengen zu verwalten. Das heißt vor allem, die Daten so zu speichern, dass bei Bedarf gezielt auf sie zugegriffen werden kann. Datenwerte sind auch die Grundlage für das maschinelle Rechnen ("Computing"). Dabei geht es nicht nur um numerische Berechnungen (von Summen, Mittelwerten, oder komplexer Formeln), sondern auch um Zeichenfolgen- bzw. String-Operationen wie z.B. die String-Verkettung oder die Teilstringsuche sowie um symbolische Berechnungen wie z.B. in der Computeralgebra.

Eine Variable hat einen Namen und einen Wert. Sie ist über ihren Namen mit einer Adresse im Hauptspeicher verknüpft, wo ihr Wert wie in einem Behälter abgelegt ist. Der Wert einer Variablen kann durch eine Wertzuweisung geändert werden. Bevor eine Variable in einer Berechnung verwendet werden kann, muss sie mit Hilfe des Schlüsselworts `var` deklariert werden. Bei der Deklaration einer Variablen kann man ihr einen Initialwert zuweisen, wie die folgenden Beispiele zeigen:

```
var i = 0;
var Betrag = 231.17;
var x = 2.175e5;
var Eingabe = "";
var Ausgabe = "Unsinn";
```

Die drei Variablen `i`, `Betrag` und `x` sind Beispiele für *numerische* Variablen (sie haben einen Zahlenwert), während die beiden Variablen `Eingabe` und `Ausgabe` Beispiele für *String*-Variablen sind (sie haben einen Zeichenfolgen- bzw. Text-Wert). Die Zuweisung `Eingabe = ""` weist der Variablen `Eingabe` den Leerstring als Wert zu. Wenn eine Variable deklariert wird, ohne ihr einen Initialwert zuzuweisen, dann hat sie so lange den speziellen Wert `undefined`, bis ihr ein richtiger Wert zugewiesen wird.

Bei den numerischen Variablen-Deklarationen haben wir es mit drei unterschiedlichen Zahlenformaten zu tun:

1. Der Datenwert `0` ist eine Ganzzahl.
2. Der Datenwert `231.17` ist eine Gleitkommazahl in Dezimalschreibweise.
3. Der Datenwert `2.175e5` ist eine Gleitkommazahl im Zehnerpotenzformat, d.h. sie bedeutet $2.175 * 10^5$, also 217500.

Generell gibt es in einer Programmiersprache *vordefinierte* und *benutzerdefinierte* Namen. So sind z.B. die Schlüsselworte `var` zur Deklaration von Variablen oder `alert` für die Prozedur zur Bildschirmausgabe vordefinierte Namen, während `i`, `Betrag`, `x`, etc. benutzerdefiniert sind. Für benutzerdefinierte Namen gelten in JavaScript folgende Regeln:

- sie dürfen keine Leerzeichen enthalten;
- sie dürfen nur aus Buchstaben und Ziffern sowie den beiden Sonderzeichen "_" und "\$" bestehen, das erste Zeichen darf jedoch keine Ziffer sein;
- es sind Groß- und Kleinbuchstaben erlaubt, wobei Groß- und Kleinschreibung unterschieden werden (die Namen "Betrag" und "betrag" bezeichnen also zwei unterschiedliche Variablen);
- ab der JavaScript-Version 1.5 dürfen sie auch Sonderzeichen wie deutsche Umlaute und das "ß" enthalten;
- sie dürfen nicht mit einem vordefinierten Namen identisch sein.

Ohne treffende, sich selbst erklärende Namen für Variablen (und Prozeduren) werden Sie nach einigen Monaten die größte Mühe haben, in Ihrem eigenen Programm zu verstehen, welche Bedeutung die Bezeichner haben.

Achten Sie darauf, dass Sie sich selbst erklärende Namen für Variablen (und Prozeduren) vergeben. Dadurch wird die Lesbarkeit eines Programms stark verbessert. Und davon profitieren nicht nur andere, die Ihr Programm vielleicht irgendwann mal lesen sollen oder wollen, sondern auch Sie selbst als Autor Ihres eigenen Programms.

Einfache Eingabe-Ausgabe-Programme

Wir können nun einfache Programme schreiben, die Werte einlesen und in Variablen speichern, um sie dann wieder auszugeben. Die Eingabe erfolgt mit Hilfe des Befehls `prompt` und die Ausgabe mit Hilfe des Befehls `document.write`, der direkt in das Browserfenster schreibt:

```
<script>//
01  var Eingabe;
02  Eingabe = prompt("Geben Sie Ihren Vornamen ein");
03  document.write("Hallo ", Eingabe, "!");
//]]&gt;&lt;/script&gt;</pre>
</div>
<div data-bbox="101 664 292 682" data-label="Section-Header">
<h2><a href="#">Ausführen/anzeigen</a></h2>
</div>
<div data-bbox="101 698 936 735" data-label="Text">
<p>Beachten Sie, dass die JavaScript-Prozedur <code>prompt</code> in Zeile 02 den eingegebenen Text als Wert an die Variable <code>Eingabe</code> übergibt.</p>
</div>
<div data-bbox="101 768 346 798" data-label="Section-Header">
<h1>Ausdrücke</h1>
</div>
<div data-bbox="101 822 938 877" data-label="Text">
<p>Ein Ausdruck ist eine Kombination von Datenwerten, Variablen, Konstanten, Operatoren, Funktionen und Klammern. Ausdrücke werden dazu verwendet, um Berechnungen durchzuführen. Beispiele für numerische Ausdrücke sind:</p>
</div>
<div data-bbox="136 896 356 932" data-label="List-Group">
<ul>
<li>• <code>23.5 + 7</code></li>
<li>• <code>1.2 * (Zahl1 - Zahl2)</code></li>
</ul>
</div>
```

Beispiele für String-Ausdrücke sind:

- "Hallo " + "Du!"
- Vorname + Zuname + "!"
- "23.5" + "7"

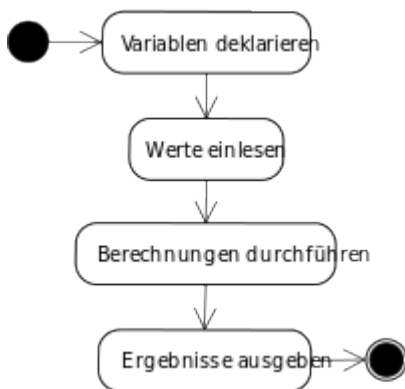
Während der numerische Ausdruck $23.5 + 7$ als die Zahl 30.5 ausgewertet wird, wird der String-Ausdruck "23.5" + "7" als der String "23.57" ausgewertet. Das heißt, das Operator-Zeichen "+" bedeutet im ersten Fall die arithmetische Addition und im zweiten Fall die Zeichenverkettung (Konkatenation).

Außer den vier Operatoren für die Grundrechenarten Addition (+), Subtraktion (-), Multiplikation (*) und Division (/) gibt es als weiteren numerischen Operator die Modulo-Operation %, mit der man den Rest bei ganzzahliger Division berechnet. Zum Beispiel ergibt $12 \% 5 = 2$, da nach ganzzahliger Division von 12 durch 5 der Rest 2 bleibt.

Kommentare

Kommentare sind Erläuterungstexte zur Information des Programmlesers. Sie beeinträchtigen die Programmausführung nicht und erlauben die Dokumentation des Programms. Wie z.B. unten im [Würfelberechnungsprogramm](#) zu sehen ist, können einzeilige Kommentare durch // eingeleitet werden. Ein mehrzeiliger Kommentar wird mit /* eingeleitet und mit */ beendet.

Einfache Berechnungen



Einfache Berechnungsprogramme haben folgende vierteilige Struktur:

1. Variablen deklarieren
2. Werte einlesen
3. Berechnungen durchführen
4. Ergebnisse ausgeben

Eine solche Folge von Berechnungsschritten kann auch mit Hilfe eines *Aktivitätsdiagramms* beschrieben werden. Dabei beginnt der Kontrollfluss bei dem schwarz ausgefüllten Kreis und führt dann über die Pfeile zu weiteren Aktivitäten bzw.

Berechnungsschritten, die durch Rechtecke mit abgerundeten Ecken dargestellt werden. Das Ende des Programms wird durch einen umrandeten schwarzen Kreis dargestellt.

In unseren einfachen Berechnungsprogrammen kann die Eingabe und die Ausgabe wie oben beschrieben mit Hilfe von `prompt` und `document.write` programmiert werden.

Würfelberechnung

Das Volumen und die Oberfläche eines Würfels lassen sich durch einfaches Multiplizieren aus der Kantenlänge berechnen.

```
<script>
```

```
04 var GanzzahligeZufallszahl = Math.floor(Math.random() * (max - min + 1) + min);
05 document.writeln("Zufallszahl zwischen ",min," und ",max,"<br />");
06 document.writeln(Zufallszahl, " ",GanzzahligeZufallszahl);
```

[Ausführen/anzeigen](#)

Übung 1: Schreiben Sie ein Programm, das die Kapitalverzinsung (inklusive Zinseszinsen) berechnet. Dabei wird ein Anfangskapital K mit jährlich $p\%$ verzinst. Wie hoch ist das Kapital nach n Jahren?

Fehlersuche

Beim Programmieren macht man immer wieder alle möglichen Fehler. Deshalb ist es wichtig zu wissen, wie man Fehler gegebenenfalls auf systematische Weise aufspüren kann. Bei den meisten Programmiersprachen gibt es dazu spezielle Werkzeuge zur Fehlersuche ("Debugger"). Bei JavaScript-Programmen ist man auf die Unterstützung durch Web-Browser angewiesen. So bietet z.B. der Firefox-Browser unter *Tools* eine *JavaScript Console*, d.i. ein separates Fenster, in dem Fehlermeldungen und Warnungen angezeigt werden.

Viele Fehler sind banale Syntax-Verletzungen, wie z.B. das abschließende Semikolon am Ende einer Anweisung weglassen oder einen JavaScript-Befehl falsch zu schreiben. Solche **Syntaxfehler** sind relativ einfach zu finden, da sie in der JavaScript-Konsole angezeigt werden. Das folgende fehlerhafte Programm illustriert den Fall eines falsch geschriebenen JavaScript-Befehls:

```
01 var Eingabe;
02 Eingabe = prompt("Geben Sie Ihren Vornamen ein");
03 document.write("Hallo ", Eingabe, "!");
```

[Ausführen/anzeigen](#)

Wo ist der Schreibfehler?

Schwieriger ist es bei einem **Semantik-Fehler**, wenn man eine Anweisung falsch verwendet und dann nicht die gewünschten Resultate erhält. Bei solchen Fehlern hilft meist nur eine gründliche Analyse des Programms, am besten in einer Diskussion mit anderen Programmiererinnen.

String-Manipulation

Strings können auf verschiedene Weise manipuliert werden: man kann sie z.B. zerlegen oder miteinander verketteten oder man kann in ihnen nach einem Teilstring suchen. Für die Stringverkettung gibt es den Operator `+`. Zur Ermittlung der Position eines Teilstrings innerhalb eines Strings gibt es die Funktion `indexOf(suchString)`, deren Verwendung in folgendem Programm illustriert wird:

```
var pos;
var Zitat =
    "Was hülfe es dem Menschen, so er die ganze Welt gewönne und nähme doch schaden an
```

```
seiner Seele?";  
pos = Zitat.indexOf("Welt");  
document.write("Gefunden bei Position: ", pos);
```

[Ausführen/anzeigen](#)

Zur Extraktion eines Teilstrings aus einem String gibt es die Funktion `slice(anfang, ende)`. Der Parameter *anfang* gibt die Position des ersten zu extrahierenden Zeichens an, während *ende* die Position des ersten nicht mehr zu extrahierenden Zeichens angibt (in beiden Fällen beginnt die Positionszählung bei 0). Mit Hilfe der Funktion `toUpperCase()` können alle Buchstaben eines Strings in Großbuchstaben umgewandelt werden. Die Verwendung dieser beiden Funktionen wird in folgendem Programm illustriert:

```
var dieGanzeWelt;  
var DIEGANZEWELT;  
var Zitat = "Was hülfe es dem Menschen, so er die ganze Welt gewönne und nähme doch schaden  
an seiner Seele?";  
dieGanzeWelt = Zitat.slice(33,47);  
DIEGANZEWELT = dieGanzeWelt.toUpperCase();  
document.write("DIEGANZEWELT = ", DIEGANZEWELT);
```

[Ausführen/anzeigen](#)

Übung 2: Schreiben Sie ein Programm, das aus einer Eingabe von Namen in der Form "Vorname Nachname" eine Ausgabe der Form "NACHNAME, Vorname" erzeugt.

Teil 2 Algorithmen und Programme

Inhalt

- [Was ist ein Algorithmus?](#)
- [Wie kann ein Algorithmus beschrieben werden?](#)
- [Programmablaufpläne in der Form von UML-Aktivitätsdiagrammen](#)
- [Fallunterscheidungen](#)

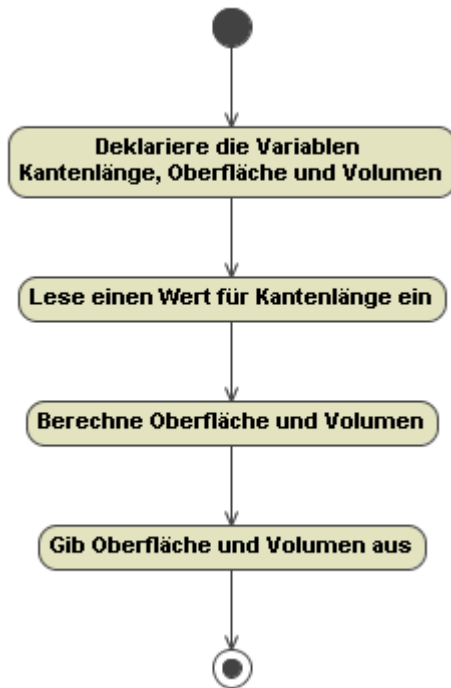


Abb. 1 Algorithmus zur Berechnung von Oberfläche und

Volumen eines Würfels

Was ist ein Algorithmus?

Allgemein kann man einen Algorithmus als ein *Verhaltensskript*, d.h. als eindeutige Beschreibung einer systematischen Vorgehensweise auf der Grundlage einer vorgegebenen Menge von möglichen Aktionen, auffassen.

Beispiele für Algorithmen sind:

- Kochrezepte:
 1. 500g Mehl, 100g Butter und 1 Ei verkneten.
 2. Wenn der Teig nicht zusammengeht, 100 ml Milch dazugießen.
 3. Teig 1 Stunde zugedeckt kühl stellen.
 4. Im Ofen bei 175 C in 40 min backen.
- IKEA-Möbel-Aufbauanleitungen
- Berechnungsverfahren:
 1. Den größten gemeinsamen Teiler zweier Zahlen bestimmen.
 2. Zwei Brüche addieren.
 3. Feststellen, ob eine gegebene natürliche Zahl eine Primzahl ist.

Anforderungen an Algorithmen:

- Ein Algorithmus ist mit endlich vielen Anweisungen zu formulieren.
- Es gibt eine eindeutig bestimmte Anweisungen, die als erste auszuführen ist.
- Es ist eine Reihenfolge bei der Anwendung der Anweisungen vorgegeben, d.h. nach Anwendung einer Anweisung kann festgestellt werden, ob der Algorithmus zu Ende ist oder welche Anweisung als nächste auszuführen ist.

In der Informatik geht es um die Ausführbarkeit von Algorithmen durch Maschinen wie Digital-Computer bzw. durch abstrakte Maschinen wie z.B. *Turing-Maschinen* (einem

abstrakten Modell eines Digital-Computers).




Wie kann ein Algorithmus beschrieben werden?

Ein Algorithmus kann auf verschiedene Weise ausgedrückt werden:

1. Als Text mit Hilfe einer eingeschränkten eindeutigen Sprache (die natürlichen Sprachen wie Deutsch oder Englisch erlauben viele Arten von Mehrdeutigkeiten, die es zu vermeiden gilt).
2. Grafisch mit Hilfe einer Diagramm-Notation (z.B. Programmablaufpläne in der Form von UML-Aktivitätsdiagrammen).
3. Mit Hilfe einer formalen Sprache (wie z.B. einer konkreten oder abstrakten Programmiersprache).

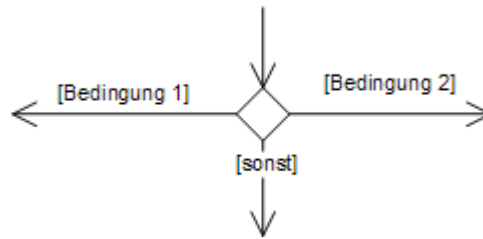
Programmablaufpläne in der Form von UML-Aktivitätsdiagrammen

In der Informatik werden verschiedene Arten von Systemen, vor allem aber Softwaresysteme, mit Hilfe der *Unified Modeling Language* ([UML](#)) modelliert. UML besteht aus mehreren Diagrammsprachen zur Modellierung von Systemstrukturen und Systemverhalten. UML-Aktivitätsdiagramme erlauben die Modellierung von Prozessen und auch von Algorithmen. Deswegen verwenden wir Aktivitätsdiagramme als graphische Notation zur Darstellung von Programmablaufplänen. Ein Beispiel eines Programmablaufplans für den Algorithmus zur Berechnung von Oberfläche und Volumen eines Würfels ist in [Abb. 1](#) zu sehen.

Name	Grafisches Symbol	Bedeutung
Ablauf-Start		Drückt den Beginn des Algorithmus aus. Gibt es nur einmal pro Ablaufplan.
Ablauf-Ende		Drückt das Ende des Algorithmus aus. Kann pro Ablaufplan mehrfach auftreten.
Handlungsschritt		Rechtecke mit abgerundeten Ecken stellen Handlungsschritte dar, die eine oder mehrere Anweisungen beinhalten.

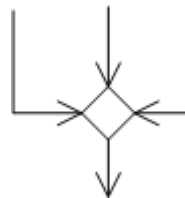
Name**Grafisches Symbol****Bedeutung**

Fallunterscheidung



Die Raute kann zwei oder mehr ausgehende Ablaufpfeile mit jeweils einer Bedingung haben (wobei sich die jeweiligen Bedingungen wechselseitig ausschließen). Einer dieser Pfeile kann auch ohne Bedingung sein oder mit dem Schlüsselwort "sonst" (Englisch "else") versehen sein; in beiden Fällen handelt es sich um einen "sonst"-Ablaufpfeil. Je nachdem, welche Bedingung gilt, wird entlang des entsprechenden Pfeiles verzweigt. Wenn keine Bedingung gilt, wird entlang des "sonst"-Pfeiles verzweigt.

Wiederezusammenführung



Die Wiederezusammenführungs-Raute kann zwei oder mehr eingehende Ablaufpfeile haben. Sie erlaubt es, zwei oder mehr alternative Abläufe wieder zusammenzuführen.

Fallunterscheidungen

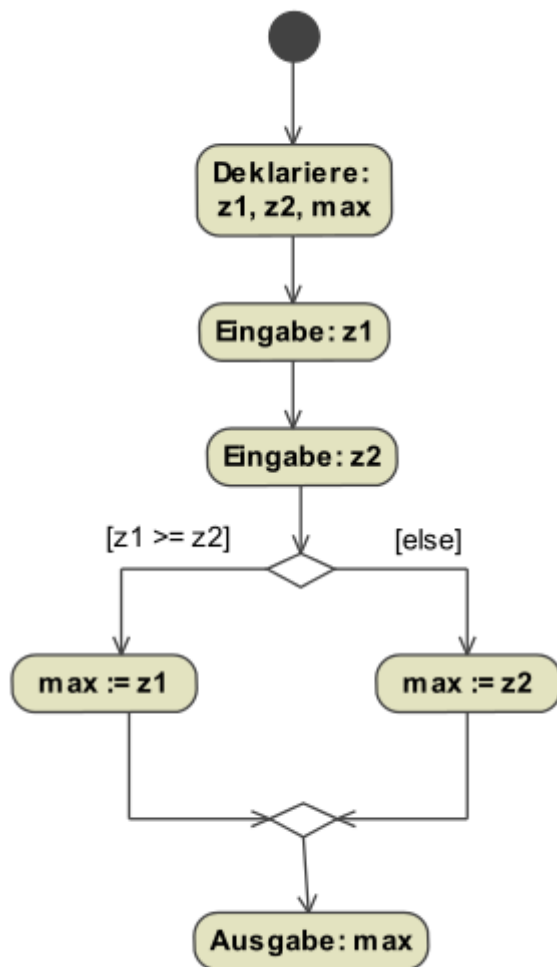


Abb. 2 Algorithmus zur Berechnung des

Maximums zweier Zahlen

Fallunterscheidungen (auch *bedingte Verzweigung* genannt) sind wichtige Bestandteile von Algorithmen. Sie basieren auf der Auswertung von logischen Bedingungen, die nach dem englischen Logiker [George Boole](#) auch *Boolesche* Ausdrücke genannt werden. Bei einer einfachen Fallunterscheidung gibt es zwei ausgehende Ablaufpfeile. Wenn nur einer von ihnen mit einer Bedingung versehen ist, wie z.B. in [Abb. 3](#), dann stellt der andere einen "sonst"-Ablaufpfeil dar, dem die Negation der Bedingung zugrundeliegt. Bei einer solchen einfachen Fallunterscheidung sprechen wir auch von einem *Ja-* bzw. einem *Nein-Ablaufzweig* (bzw. *Sonst-Ablaufzweig*).

Berechnung des Maximums zweier Zahlen

Zum Beispiel muss bei der Berechnung des Maximums zweier Zahlen z_1 und z_2 der Fall, dass z_1 größer oder gleich z_2 ist, von dem Fall, dass z_1 kleiner als z_2 ist, unterschieden werden. Im ersten Fall ist z_1 das Maximum, im zweiten Fall z_2 . Entsprechend verzweigt das Ablaufdiagramm in [Abb. 2](#). In Worten lässt sich dieser Algorithmus folgendermaßen ausdrücken:

1. Deklariere die Variablen z_1 , z_2 und max
2. Gib einen Wert für z_1 ein
3. Gib einen Wert für z_2 ein
4. **Wenn** $z_1 \geq z_2$,
 dann setze $max := z_1$,
 sonst setze $max := z_2$
5. Gib max aus

Dabei ist zu beachten, dass wir jetzt "!=" als Symbol für die Wertzuweisung verwenden, um eine Verwechslung mit dem Gleichheitssymbol zu vermeiden. Während man mit dem Gleichheitssymbol eine logische Bedingung ausdrückt, drückt man mit dem Wertzuweisungssymbol eine Programmanweisung (also einen Handlungsschritt) aus.

Die Implementierung des Algorithmus in JavaScript

Die logische *Gleichheit* darf auf keinen Fall mit der *Wertzuweisung* verwechselt werden, auch wenn in manchen Programmiersprachen dasselbe Zeichen sowohl für die Gleichheit als auch für die Wertzuweisung verwendet wird. In JavaScript wird das Zeichen "=" für die Wertzuweisung und "==" für die Gleichheit verwendet.

Bei der Implementierung dieses einfachen Algorithmus mit JavaScript ist zu beachten, dass Benutzereingaben als Strings interpretiert werden. Wenn man aber nicht Strings, sondern ganze Zahlen miteinander vergleichen will, muss man Benutzereingaben mit Hilfe der in JavaScript vordefinierten Funktion `parseInt` zuerst in ganze Zahlen umwandeln, bevor man sie vergleicht. Andernfalls ist "2" > "12", obwohl ja $2 < 12$ ist!

```
var max=0;
var z1 = parseInt( prompt("z1:"));
var z2 = parseInt( prompt("z2:"));

if (z1 >= z2) max = z1;
else max = z2;

document.writeln("max = " + max);
```

[Ausführen/Anzeigen](#)

Beachten Sie die Syntax der `if`-Anweisung: die Bedingung steht in runden Klammern; danach kommt entweder eine mit Semikolon abgeschlossene Anweisung oder Anweisungsfolge.

Eine *Anweisungsfolge* (im Ja- oder Nein-Verzweigungsteil) muss in geschweiften Klammern eingeschlossen sein, wie in dem folgenden Beispiel zu sehen ist:

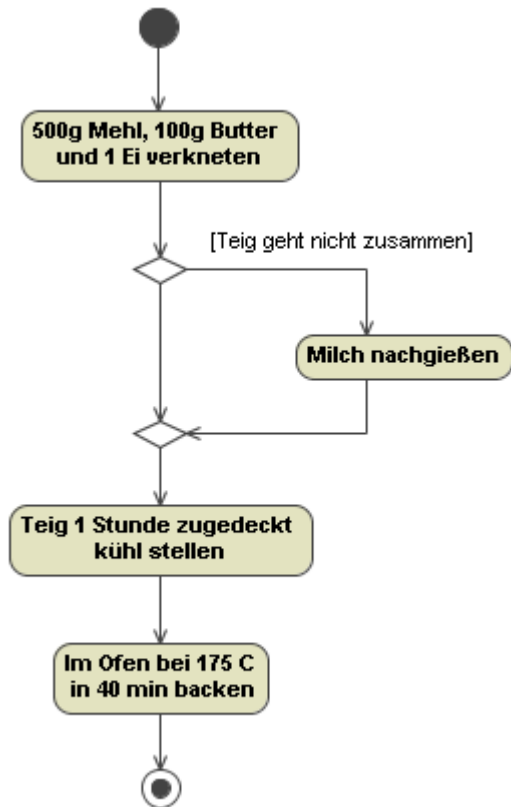


Abb. 3 Algorithmus zum Kuchenbacken

```

var max,min = 0;
var z1 = parseInt( prompt("z1:"));
var z2 = parseInt( prompt("z2:"));
if (z1 >= z2) {
    max = z1;
    min = z2;
} else {
    max = z2;
    min = z1;
}
document.writeln("max = " + max + " min = " + min);
  
```

[Ausführen/Anzeigen](#)

Kuchen backen

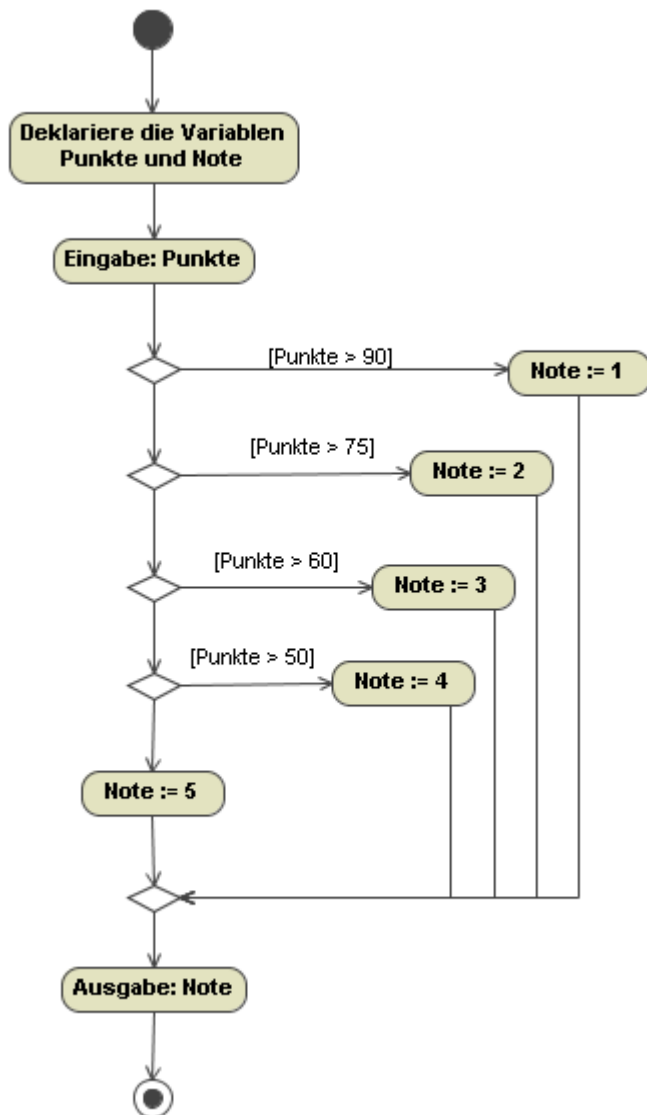
Das Ablaufdiagramm in [Abb. 3](#) zeigt eine einfache Fallunterscheidung, bei der nur im Ja-Fall ein spezieller Handlungsschritt ("Milch nachgießen") durchgeführt wird. Im Nein-Fall geht es einfach dort weiter, wo es auch nach dem Ja-Fall-Handlungsschritt weitergeht ("Teig 1 Stunde zugedeckt kühl stellen"). In Worten lässt sich dieser Algorithmus folgendermaßen ausdrücken:

1. Verknete 500g Mehl, 100g Butter und 1 Ei
2. **Wenn** der Teig nicht zusammengeht,
 dann gieße 100 ml Milch dazu
3. Stelle den Teig 1 Stunde zugedeckt kühl
4. Backe den Kuchen im Ofen bei 175 C in 40 min

Noten berechnen

Das Ablaufdiagramm in [Abb. 4a](#) zeigt eine verkettete Fallunterscheidung, die sich in Worten folgendermaßen ausdrücken lässt:

1. Deklariere die Variablen *Punkte* und *Note*
2. Gib einen Wert für *Punkte* ein
3. **Wenn** *Punkte* > 90, **dann** setze *Note* := 1,
sonst wenn *Punkte* > 75, **dann** setze *Note* := 2,
sonst wenn *Punkte* > 60, **dann** setze *Note* := 3,
sonst wenn *Punkte* > 50, **dann** setze *Note* := 4,
sonst setze *Note* := 5
4. Gib *Note* aus



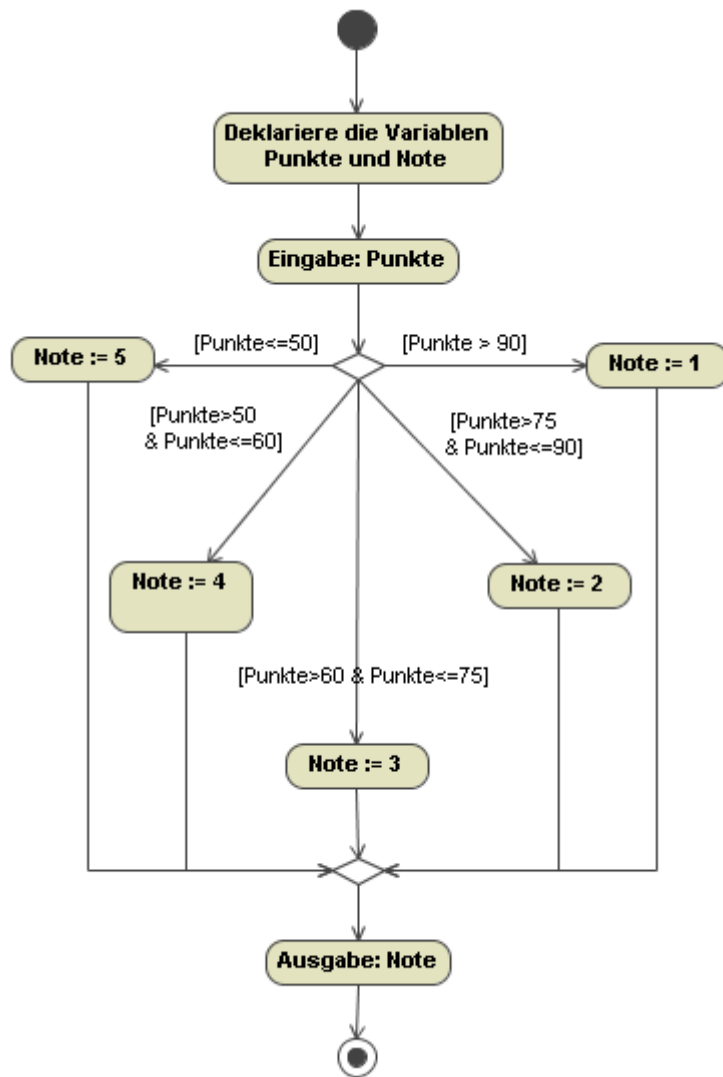


Abb. 4a + 4b: Algorithmen zur

Notenberechnung

Dieselbe fallweise Berechnung lässt sich auch als Mehrfach-Fallunterscheidung wie im Ablaufdiagramm in [Abb. 4b](#) durchführen. In Worten ausgedrückt:

1. Deklariere die Variablen *Punkte* und *Note*
2. Gib einen Wert für *Punkte* ein
3. **Wenn** *Punkte* > 90, **dann** setze *Note* := 1
4. **Wenn** *Punkte* > 75 und *Punkte* <= 90, **dann** setze *Note* := 2
5. **Wenn** *Punkte* > 60 und *Punkte* <= 75, **dann** setze *Note* := 3
6. **Wenn** *Punkte* > 50 und *Punkte* <= 60, **dann** setze *Note* := 4
7. **Wenn** *Punkte* <= 50, **dann** setze *Note* := 5
8. Gib *Note* aus

Implementierung einer verketteten Fallunterscheidung (Abb. 4a)

Für die Lesbarkeit einer verketteten `if-else-if`-Konstruktion ist es wichtig, dass man auf geeignete Weise geschweifte Klammern setzt:

```
var Punkte = prompt("Punkte:");
```

```

var Note=0;
if (Punkte > 90) {
    Note = 1;
} else if (Punkte > 75) {
    Note = 2;
} else if (Punkte > 60) {
    Note = 3;
} else if (Punkte > 50) {
    Note = 4;
} else {
    Note = 5;
}
document.write("Note = " + Note);

```

[Ausführen/Anzeigen](#)

Einrückungen sind ein wichtiges Gestaltungsmittel für Programmtexte. Sie dienen der besseren Lesbarkeit.

Implementierung einer Mehrfach-Fallunterscheidung (Abb. 4b)

Anstatt einer verketteten *if-then-else*-Anweisung kann man auch eine Folge von *if*-Anweisungen verwenden, bei denen die Bedingungen nicht überlappen:

```

var Punkte = prompt("Punkte:");
var Note=0;
if (Punkte > 90) {
    Note = 1;
}
if (Punkte > 75 && Punkte <= 90) {
    Note = 2;
}
if (Punkte > 60 && Punkte <= 75) {
    Note = 3;
}
if (Punkte > 50 && Punkte <= 60) {
    Note = 4;
}
if (Punkte <= 50) {
    Note = 5;
}
Ausführen
document.write("Note = ", Note);

```

[Ausführen/Anzeigen](#)

Beachten Sie, wie in JavaScript der logische Operator *Konjunktion* (UND) notiert wird, nämlich als doppeltes kaufmännisches Und (&&).

Vergleichsrelationen und logische Operatoren

Vergleichsrelationen, wie z.B. *Gleichheit* und *Ungleichheit*, dienen zur Formulierung von Bedingungen, die mit Hilfe von logischen Operatoren, wie z.B. *Konjunktion* und *Negation*, zu komplexen Bedingungen (*Booleschen Ausdrücken*) zusammengesetzt werden können.

Logische Relation	Symbol
-------------------	--------

Gleichheit	===
------------	-----

Ungleichheit	!==
--------------	-----

Kleiner als	<
-------------	---

Kleiner als oder gleich	<=
-------------------------	----

Größer als	>
------------	---

Größer als oder gleich	>=
------------------------	----

Die in JavaScript ebenfalls vorhandene Pseudo-Gleichheitsrelation `==`, sowie die entsprechende Pseudo-Ungleichheitsrelation `!=` sollten nicht verwendet werden, da sie z.B. so unerwünschte Ergebnisse wie `2 == "2"` liefern.

Die Ordnungsrelationen `<`, `<=`, `>` und `>=` gelten nicht nur für Zahlen, sondern auch für Strings, bei denen sie sich auf die lexikografische (alfabetische) Ordnung beziehen. Es gibt drei logische Operatoren:

Logischer Operator	Symbol
--------------------	--------

Konjunktion (UND)	&&
-------------------	----

Disjunktion (ODER)	
--------------------	--

Negation (NICHT)	!
------------------	---

Z.B. ist der Boolesche Ausdruck `!(x === y)` gleichbedeutend mit `(x !== y)`. Da Zahlen und Strings linear geordnet sind, ist `!(x < y)` gleichbedeutend mit `(x >= y)` und analog `!(x > y)` gleichbedeutend mit `(x <= y)`.

Mehrfach-Fallunterscheidungen mit Gleichheitsbedingungen

Wenn ein Ausdruck auf Gleichheit mit einem Datenwert aus einer Menge von möglichen Werten getestet werden soll, dann bietet sich dafür anstatt einer Folge von

if-Anweisungen die einfachere `switch`-Anweisung mit folgender Syntax an:

```
switch (Ausdruck) {  
  case Wert1:  
    Anweisungen1;  
    break;  
  case Wert2:  
    Anweisungen2;  
    break;  
  ...  
  default:  
    Anweisungen-für-den-Sonst-Fall;  
}
```

Dabei wird der Ausdruck ausgewertet und der resultierende Wert dann mit den in den `case`-Klauseln aufgeführten Werten Wert1, Wert2, etc. verglichen. Wenn der resultierende Wert mit einem der `case`-Werte übereinstimmt, dann werden die für diesen Fall angegebenen Anweisungen durchgeführt und mit der folgenden `break`-Anweisung wird die `switch`-Anweisung beendet und mit der ihr folgenden Anweisung fortgefahren. Ansonsten, wenn keiner der definierten Fälle zutrifft, kann man am Ende mit Hilfe der `default`-Klausel definieren, was dann passieren soll.

Bedingte Wertausdrücke

Eine bedingte Wertzuweisung kann man entweder mit Hilfe von `if-else`-Anweisungen oder einfacher mit Hilfe eines *bedingten Wertausdrucks* in einer einzigen Anweisung programmieren. Ein bedingter Wertausdruck hat folgende Syntax:

```
Bedingung ? Ja-Ausdruck : Nein-Ausdruck
```

Wenn die *Bedingung* zutrifft, dann bestimmt der *Ja-Ausdruck* den Wert, sonst bestimmt der *Nein-Ausdruck* den resultierenden Wert. In folgendem Beispiel hängt die Kundenkategorie "Gold-Kunde" davon ab, ob der Umsatz mit einem Kunden größer als 1000 Euro ist oder nicht:

```
kundenKategorie = (umsatz > 1000) ? "Gold-Kunde" : "Silber-Kunde"
```

Teil 3 Prozeduren und Funktionen

Inhalt

- [Eine Prozedur/Funktion definieren](#)

Eine Prozedur ist wie ein kleines, in sich abgeschlossenes Programm. Sie besteht aus einem Prozedur-Kopf und einem Prozedur-Rumpf. Der Rumpf ist einfach ein Anweisungsblock, also eine Folge von Anweisungen, die in JavaScript (wie auch in vielen anderen Programmiersprachen) in geschweiften Klammern eingeschlossen und durch Semikolon voneinander getrennt sind. Diese Anweisungen werden jedesmal, wenn die Prozedur aufgerufen wird, ausgeführt. Der Kopf einer Prozedur dient vor allem dazu, die

Prozedur mit Startwerten für ihre Variablen zu versorgen. Wenn eine Prozedur nur den Zweck hat, für gegebene Eingangswerte einen bestimmten Ausgangswert zu berechnen, dann kann man sie als algorithmische Realisierung einer Funktion im Sinne der Mathematik begreifen. Weil Prozeduren oft genau das tun, heißen sie in JavaScript "Funktionen" (während sie in Java "Methoden" heißen). Eine JavaScript-Funktion ist also eigentlich eine Prozedur und entspricht nicht immer einer wirklichen Funktion (im Sinne der Mathematik).

Durch die Verwendung von benutzerdefinierten Prozeduren und Funktionen kann man große Programme nach dem Prinzip "teile und herrsche" überschaubar halten. Richtig eingesetzt helfen sie also dabei, ein Programm gut zu strukturieren und seine Lesbarkeit bzw. Verständlichkeit zu erhöhen.

Eine Prozedur/Funktion definieren

Die Definition einer JavaScript-Funktion beginnt mit dem Schlüsselwort `function`, gefolgt von

- einem benutzerdefinierten Namen der JavaScript-Funktion, wie z.B. "max";
- einer in runden Klammern eingeschlossenen, komma-separierten Liste von *Parametern*, wie z.B. "(x,y)";
- einem Anweisungsblock, der definiert, was die JavaScript-Funktion macht, wie z.B.
`{ var m; if (x>y) m=x; else m=y; return m; }`

Eine JavaScript-Funktion kann einen mit Hilfe des Schlüsselworts `return` definierten *Rückgabewert* haben (und immer wenn sie eine wirkliche Funktion ist, hat sie auch einen solchen Rückgabewert). Ein *Parameter* ist wie eine Variable, d.h. er hat einen Namen und erhält beim Aufruf der JavaScript-Funktion einen Wert. In folgendem Beispiel wird eine JavaScript-Funktion `max` in den Zeilen 01 bis 06 zuerst definiert und dann bei der Ausgabe des Maximums zweier eingegeben Zahlen in Zeile 09 aufgerufen:

```
01 function max(x,y){
02     var m=0;
03     if (x>y) m=x;
04     else m=y;
05     return m;
06 }
07 var z1 = prompt("z1:");
08 var z2 = prompt("z2:");
09 document.writeln("max = " + max(z1, z2);
```

Bei dieser JavaScript-Funktion handelt es sich um eine echte Funktion: es werden nur Berechnungsschritte (nicht jedoch zustandsverändernde Handlungen) durchgeführt und es gibt einen Rückgabewert (als den *Funktionswert*). Beachten Sie, wie beim Aufruf der Funktion `max` in Zeile 09 die beiden Ausdrücke `z1` und `z2` an die Stelle der Parameter `x` und `y` getreten sind. Im allgemeinen können beliebige passende Ausdrücke beim Aufruf einer JavaScript-Funktion verwendet werden.

Ein Beispiel für eine JavaScript-Funktion ohne Rückgabewert ist das folgende:

```
01 function max(x,y){
02     var m=0;
```

```

03   if (x>y) m=x;
04   else m=y;
05   return m;
06 }
07 function output( AusgabeText, Ausgabe){
08     document.writeln(AusgabeText + Ausgabe);
09 }
10 var z1 = prompt("z1:");
11 var z2 = prompt("z2:");
12 output("Maximum = ", max(z1,z2));

```

Bei der in den Zeilen 07-09 definierten JavaScript-Funktion output handelt es sich um eine Prozedur: es geht nicht um die Berechnung eines Funktionswerts für gegebene Argumente, sondern es werden irgendwelche Handlungsschritte durchgeführt. Zeile 12 zeigt den Aufruf dieser Prozedur mit einem String-Argument und einem numerischen Argument.

Teil 4 Eingabe und Ausgabe in HTML-Formularen

Inhalt

HTML-Formulare erlauben eine einfache Programmierung von Benutzerschnittstellen, über die ein Benutzer ein Webanwendungsprogramm bedient. Wir können HTML-Formulare als Alternative zu den Eingabe-/Ausgabe-Anweisungen, die wir bisher verwendet haben, einsetzen.

```

<form action="hello.html">
<p>
<label for="name">Name:</label>
<input type="text" id="name" name="Name" />
<label for="bio">Biographie:</label>
<textarea name="Bio" id="bio" cols="50" rows="10">
<input type="submit" value="Absenden">
</p>
</form>

```

[Ausführen/anzeigen](#)

Teil 5 String-Operationen und reguläre Ausdrücke

Inhalt

Bei der Stringsuche ist es oft praktisch, ein Stringmuster für die Suche angeben zu können. Zu diesem Zweck gibt es eine spezielle Syntax für StringmusterAusdrücke, auch

reguläre Ausdrücke (regular expressions) genannt. Z.B.

Operation	Erklärung
<code>strVar.length</code>	Liefert die Anzahl der Zeichen (also die Länge) einer Zeichenfolge <code>strVar</code> .
<code>strVar.toLowerCase()</code> <code>strVar.toUpperCase()</code>	Wandelt alle Buchstaben einer Zeichenfolge <code>strVar</code> in Kleinbuchstaben bzw. Großbuchstaben um.
<code>strVar.indexOf(suchStr)</code> <code>strVar.lastIndexOf(suchStr)</code>	Ermittelt das erste (bzw. letzte) Vorkommen eines Zeichens oder einer Zeichenfolge <code>suchStr</code> innerhalb einer Zeichenfolge <code>strVar</code> und gibt zurück, an wie vielter Stelle <code>suchStr</code> in der Zeichenfolge <code>strVar</code> steht. Die Zählung beginnt bei 0. Wenn die Suche erfolglos ist, wird -1 zurückgegeben.
<code>strVar.search(regExp)</code>	Durchsucht eine Zeichenfolge <code>strVar</code> mit Hilfe eines regulären Ausdrucks <code>regExp</code> . Liefert -1 zurück, wenn der reguläre Ausdruck <code>regExp</code> nicht passt. Wenn er passt, wird die Position des ersten Vorkommens zurück geliefert.
<code>strVar.replace(regExp, ersatzStr)</code>	Durchsucht eine Zeichenfolge <code>strVar</code> mit Hilfe eines regulären Ausdrucks <code>regExp</code> und ersetzt Zeichenfolgen, auf die der reguläre Ausdruck passt durch <code>ersatzStr</code> .
<code>strVar.slice(anfang, ende)</code>	Extrahiert aus einer Zeichenfolge <code>strVar</code> einen Teilstring, der mit der Position <code>anfang</code> beginnt und an der Position <code>ende-1</code> aufhört (wobei bei 0 zu zählen begonnen wird). Wenn man den zweiten Parameter weglässt, dann wird bis zum Ende der Zeichenfolge extrahiert.
<code>strVar.substr(anfang, zeichenAnzahl)</code>	Extrahiert aus einer Zeichenfolge <code>strVar</code> einen Teilstring, der mit der Position <code>anfang</code> beginnt und <code>zeichenAnzahl</code> viele Zeichen hat.
<code>strVar.split(separator)</code>	Zerlegt eine Zeichenfolge <code>strVar</code> in mehrere Teile, sofern sie durch ein Begrenzerzeichen oder eine Begrenzerzeichenfolge <code>separator</code> voneinander getrennt sind. Die erzeugten Teilstrings werden in einem Array gespeichert.

Die folgende Tabelle beschreibt die wichtigsten Möglichkeiten der Suche mit Hilfe von regulären Ausdrücken.

Symbo l	Beispie l	Beschreibung
-	/aus/	findet "aus", und zwar in "aus", "Haus", "auserlesen" und "Banause".
^	/^aus/	findet "aus" am Anfang des zu durchsuchenden Wertes, also in "aus" und "auserlesen", sofern das die ersten Wörter im Wert sind.
\$	/aus\$/	findet "aus" am Ende des zu durchsuchenden Wertes, also in "aus"

		und "Haus", sofern das die letzten Wörter im Wert sind.
*	/aus*/	findet "aus", "auss" und "aussssss", also das letzte Zeichen vor dem Stern 0 oder beliebig oft hintereinander wiederholt.
+	/aus+/?	findet "auss" und "aussssss", also das letzte Zeichen vor dem Stern mindestens einmal oder beliebig oft hintereinander wiederholt.
.	/ .aus/	findet "Haus" und "Maus", also ein beliebiges Zeichen an einer bestimmten Stelle.
.+	/ .+aus/	findet "Haus" und "Kehraus", also eine beliebige Zeichenfolge an einer bestimmten Stelle. Zusammensetzung aus <i>beliebiges Zeichen</i> und <i>beliebig viele davon</i> .
\b	/\baus\b/	findet "aus" als einzelnes Wort. \b bedeutet eine Wortgrenze.
\B	/\Baus\B/	findet "aus" nur innerhalb von Wörtern, z.B. in "hausen" oder "Totalausfall". \B bedeutet <i>keine Wortgrenze</i> .
\d	/\d.+ \B/	findet eine beliebige ganze Zahl. \d bedeutet eine Ziffer (0 bis 9)
\D	/\D.+/?	findet "-fach" in "3-fach", also keine Ziffer.
\s	/\s/	findet jede Art von Leerzeichen ("white space"), also Tabulatorzeichen, Zeilenvorschubzeichen, etc. sowie normale Leerzeichen.
\S	/\S.+/?	findet ein beliebiges einzelnes Zeichen, das kein white space ist.
/.../i	/aus/i	findet "aus", "Aus" und "AUS", also unabhängig von Groß-/Kleinschreibung.