Zeichenketten und JavaScript

Ole Vanhoefer

17. Januar 2022

Inhaltsverzeichnis

Kapitel 1

Erste Schritte

Schon sehr früh beherrschten Webbrowser die Fähigkeit neben der reinen Anzeige von Webseiten auch dort eingebettete kleine Programme, genannt Skripte, auszuführen und damit eine gewisse Interaktivität mit dem Benutzer zu ermöglichen. Die meisten Webbrowser erlauben es auch diese Skripte ohne den Inhalt von Webseiten auszuführen. Damit sind Webbrowser eine ideale Plattform um sich mit dem Programmieren bekannt zu machen, da sie heute auf fast jedem Rechner installier sind.

Die in den heutigen Webbrowsern integrierte Skriptsprache ist JavaScript. Diese Sprache ist nicht mit Java zu verwechseln, obwohl sie von der Syntax (Aussehen und Regeln) inspiriert wurde. Du brauchst also nur einen Webbrowser und einen Editor. Als Webbrowser verwenden ich hier für die Beispiele den Firefox. Als Editor kann unter Windows z.B. der Editor Notepad++ benutzt werden. Unter Linux bietet sich z.B. der Webeditor Quantas oder der Universaleditor Kate an. Unter beiden Systemen läuft der Editor Bluefish.

1.1 Hallo Welt!

Webbrowser stellen normalerweise HTML-Seite dar. Deshalb bekommt die Programmdatei die Endung `.html`. Für diesen Kurs solltest Du Dir ein Verzeichnis anlegen und dort die Dateien speichern. Speichere die Dateien bitte unter dem beim Listing angegebenen Namen ab, da eventuell andere Skripte und Seiten auf diese Dateien zurückgreifen wollen. Jetzt wollen wir anfangen. Dazu gebe das folgende Listing im Editor ein und speichere es dann unter dem Namen `hallowelt.html` ab. Die Nummern (`1:` bis `3:`) vor der Zeile sind nicht Bestandteil des Programms. Sie werden bei den Listings als Zeilennummerierung mitgeschrieben, damit Du besser bei den Erklärungen erkennen kannst, auf welche Zeilen diese sich beziehen.

Listing 1.1. hallowelt.html

```
1: <script type="text/javascript">
2: document.writeln("Hallo Welt!");
3: </script>
```

Das erste Programm beginnt mit der Zeile `<script type="text/javascript">`. Damit wird dem Browser mitgeteilt, dass er den folgenden Text bis zum Endtag `</script>` als Programm interpretieren und auch

Abbildung 1.1: Struktogramm und Programmablaufplan zu Listing 1.1

ausführen soll. Ebenfalls wird dem Browser mitgeteilt, dass die hier verwendete Programmiersprache JavaScript ist. Daraufhin übergibt der Browser seinem eingebauten JavaScript-Interpreter diesen Teil der Webseite, damit dieser die dort geschriebenen Anweisungen ausführt.

Der Befehl `document.writeln("Hallo Welt!")` schreibt den Text *Hallo Welt!* in das HTML-Dokument. Dieser Text wird dann schließlich vom Browser interpretiert und dann dargestellt.

Am Schluß wird mit `</script>` dem Browser das Ende des Programmcodes mitgeteilt.

Also lassen wir doch mal den Browser auf die Datei los. Im Firefox über `Datei > Datei öffnen` die Datei `hallowelt.html` laden.

In einem Skript werden die Befehle nacheinander von oben nach unten abgearbeitet. Ergänzen wir das erste Skript doch mal mit einem zweiten Ausgabebefehl.

Listing 1.2. hallowelt_v2.html

```
1: <script type="text/javascript">
2: document.writeln("Hallo Welt!");
3: document.writeln("Willkommen bei JavaScript!");
4: </script>
```

Im HTML-Dokument sollte nun stehen.

```
Hallo Welt!
Willkommen bei JavaScript!
```

Überraschenderweise stellt der Browser die beiden Texte nicht untereinander da, sondern schreibt sie in eine Zeile. Wenn man sich die Arbeitsweise des Browsers noch mal anschaut, ist dieses Vorgehen aber erklärbar. Der Interpreter liefert tatsächlich beide Texte in zwei Zeilen zurück. Allerdings werden diese beiden Textzeilen dann noch vom HTML-Parser des Browers interpretiert. Und für den Browser sind Zeilenumbrüche nichts anderes als Leerzeichen. Daher trennt er die beiden Sätze nicht auf zwei Zeilen auf, sondern fügt nur ein Leerzeichen ein.

Um einen Zeilenumbruch zu erhalten muss im Text der HTML-Befehl für den Zeilenumbruch eingefügt werden. Dieser sogenannte Tag (ausgeprochen: Täck) lautet `<br>`.

Listing 1.3. hallowelt_v3.html

```
1: <script type="text/javascript">
2: document.writeln("Hallo Welt!<br>");
3: document.writeln("Willkommen bei JavaScript!");
4: </script>
```

Dann sieht das HTML-Dokument so aus.

```
Hallo Welt!<br>
Willkommen bei JavaScript!
```

Ausgabe “Hallo Welt!”
Ausgabe “Willkommen bei JavaScript!”

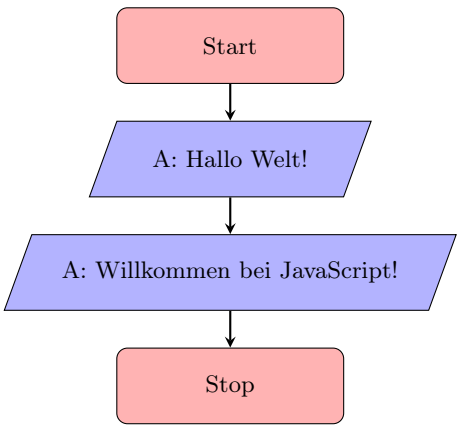

Abbildung 1.2: Struktogramm und Programmablaufplan zu Listing 1.2

Wenn jetzt der HTML-Parser des Browsers das Dokument bearbeitet, bekommt er die Anweisung nach `Hallo Welt!` einen Zeilenumbruch einzufügen. Umd damit werden jetzt beide Sätze in zwei Zeilen dargestellt.

Ein andere Methode die Zeilenumbrüche hinzubekommen ist die Verwendung eines vorformatierten Bereichs. In dieser Umgebung stellt der Browser sämtlichen Text in Schreibmaschinenschrift da und berücksichtigt jedes Leerzeichen und jeden Zeilenumbruch. Der Bereich wird durch die Tags `<pre>` und `</pre>` definiert.

Listing 1.4. hallowelt_v3a.html

```
1: <pre>
2: <script type="text/javascript">
3: document.writeln("Hallo Welt!");
4: document.writeln("Willkommen bei JavaScript!");
5: </script>
6: </pre>
```

Achte bitte darauf, dass `<pre>` und `</pre>` nur außerhalb eines Skriptbereichs benutzt werden dürfen.

1.2 Alert-Dialogfenster

Eine andere Möglichkeit eines Skriptes Informationen auszugeben ist das Alert-Dialogfenster. Durch das Kommando `alert("Hallo Welt!");` wird der Browser angewiesen ein Fenster mit dem angegebenen Text zu öffnen. Die Ausführung des Skriptes stoppt, bis der Benutzer den OK-Knopf gedrückt hat.

Listing 1.5. hallowelt_v4.html

```
1: <script type="text/javascript">
2: alert("Hallo Welt!");
3: alert("Willkommen bei JavaScript!\nWillkommen!");
4: </script>
```

Ein solches Dialogfenster wird nicht durch den HTML-Parser geschickt. Deshalb bringt auch die Verwendung von `<br>` für einen Zeilenumbruch nichts. Hier nimmt man das sogenannte Steuerzeichen `\n` (new line) um eine Zeilenumbruch im Text des Alert-Dialogfensters zu erreichen.

1.3 Ausgabe in der Webconsole

Eine weitere Möglichkeit der Ausgabe ist die Benutzung der Web-Konsole. Die Web-Konsole ist ein Tool für Entwickler, um sich Meldungen und Fehler anzeigen zu lassen. Im Firefox erreicht man die Web-Konsole über `Extras > Web-Entwickler > Web-Konsole` oder über das Hauptmenü und den Punkten `Web-Entwickler > Web-Konsole`. Alternativ auch über `Strg`+`Umschalt`+`K`.

In der Web-Konsole werden Fehler in den Webseiten und damit auch in den ausgeführten JavaScript-Programmen angezeigt. Eine wichtige Hilfe bei der Fehlersuche, da auch die Zeile, in der der Fehler auftaucht, angezeigt wird.

Mit dem Befehl `console.log()` kannst Du selber in diese Konsole Texte schreiben. Dies ist nützlich, wenn man Zwischenergebnisse und eigene Fehlermeldung bei der Programmentwicklung sichtbar machen will.

Listing 1.6. hallowelt_v5.html

```
1: <script type="text/javascript">
2: console.log("Hallo Welt!");
3: console.log("Willkommen bei JavaScript!\nWillkommen!");
4: </script>
```

1.4 Datentypen, Literale und Variablen

1.4.1 Datentypen

In der Programmierung werden meistens Daten verarbeitet. In JavaScript können diese Daten in vier Datentypen eingeteilt werden.

- Zahlen
- Zeichenketten
- Wahrheitswerte
- `null`-Wert

Die Werte können als Literale direkt ins Programm geschrieben werden oder in einer Variablen gespeichert werden.

Datentyp Number

Zahlen in JavaScript sind Gleitkommazahlen. Für Menschen aus dem deutschen Sprachraum ist es etwas gewöhnungsbedürftig, dass die Nachkommastellen nicht durch ein Komma sondern durch einen Punkt abgetrennt werden.

Der Nachkommastellenanteil ist aber bei der Darstellung nicht Pflicht. Zahlen ohne Nachkommastellen werden als Ganzzahlen (Integer) im Bereich von -2^{53} bis 2^{53} behandelt.

Neben der normalen Zahlendarstellung beherrscht JavaScript auch die Exponentialschreibweise. Diese Art der Darstellung wird bei sehr großen Zahlen verwendet. Dabei wird ausgenutzt, dass mehr die Größenordnung (Million oder Millarde) interessiert als z.B. die fünfte Ziffer (1,2345 oder 1,2346). Zahlen in Exponentialschreibweise bestehen aus einem Faktor und einer 10er-Potenz wie z.B. $1{,}234 \cdot 10^5$ was normal geschrieben 123 400 lauten würde. Um die Schreibweise zu verkürzen wird für das 10 hoch ein `e` eingesetzt. Also würde unsere Beispielzahl in JavaScript lauten `1.234e5`.

Datentyp String

Eine Zeichenkette, die in der Fachsprache als String bezeichnet wird, wird als Literal durch einfache oder doppelte Anführungszeichen eingeschlossen.

```
alert('Hallo Welt!');
alert("Willkommen bei JavaScript!\nWillkommen!");
```

Es ist dabei völlig egal, welche Anführungszeichen man verwendet, man muss nur den String mit dem gleichen Anführungszeichen beenden mit dem man ihn auch begonnen hat.

Neben den normalen Zeichen kann ein String auch Sonderzeichenkombinationen enthalte, die sogenannten *Escape-Zeichen*. Ein solcher Vertreter ist z.B. `\n`, das für das Zeilenende steht. Wichtig werden diese, wenn z.B. Anführungszeichen im Text verlangt werden.

```
alert('Dieser "Beamte" war falsch!');
alert("Er sagte: \"Hans kommt bald wieder!\"");
```

Datentyp Boolean

Der Datentyp Boolean ermöglicht die Verwendung von Wahrheitswerten. Dieser Datentyp kann zwei Werte annahmen: `true` und `false`, wobei `true` für *wahr* steht und `false` für *falsch*. Diese beiden Werten dürfen als Literal nicht in Anführungszeichen gesetzt werden, denn `false` ist etwas völlig anderes als `"false"`.

`null` und `undefined`

Der `null`-Wert steht für eine nicht vorher definierte Variable. Ihr Einsatz führt in den meisten Fällen zu einer Fehlermeldung des JavaScript-Interpreters. Dagegen ist der Wert `undefined` für vorher eingeführte Variablen vorgesehen, denen aber noch kein Wert zugeordnet wurde.

1.4.2 Variablen

Die Variablen sind das Gedächtnis des Programms. Hier werden Werte abgespeichert um später auf sie zugreifen zu können. Die Variablen sind dabei Stellvertreter für die Werte, da diese zum Zeitpunkt der Programmierung noch gar nicht bekannt sind.

Als Analogie können wir die Variablen mit Karteikarten vergleichen. Sie haben einen Namen, anhand dessen wir sie identifizieren können, und besitzen einen Inhalt, den man lesen und ändern kann.

Variablen müssen vor ihrem ersten Gebrauch deklariert und initialisiert werden. Dies kann zu einem durch das Schlüsselwort `var` erfolgen oder indem den Variablen ein Wert zugewiesen wird.

Listing 1.7. variablen.html

```
 1: <pre><script type="text/javascript">
 2: // Ausgabe von Literalen
 3: document.writeln(42);            // Ein Zahlenliteral
 4: document.writeln("Hallo Welt!"); // Ein Zeichenkettenliteral
 5:
 6: // Deklaration und Initialisierung von Variablen
 7: var a = 42;                      // Zahl
 8: var b = "Hallo Welt!";           // Zeichenkette
 9: var c = true;                    // Wahrheitswert
10: var d = null;                    // Null-Wert
11:
12: // Ausgabe von Variablen
13: document.writeln(a);
14: document.writeln(b);
15: document.writeln(c);
16: document.writeln(d);
17: </script></pre>
```

Das Struktogramm und den Programmablaufplan zu dem Listing findest Du in Abbildung 1.3 auf Seite 10.

Vielleicht ist Dir aufgefallen, daß in Listing 1.7 erläuternder Klartext steht. Dies sind Kommentare, die den Quellcode erläutern und damit leichter verständlich machen sollen. Zeilenkommentare beginnen mit der Zeichenkette `//`. Der Browser wird damit angewiesen alle Zeichen von hier bis zum Ende der Zeile zu ignorieren.

Für einen Kommentar, der über mehrere Zeilen geht, stellt man den Kommentartext zwischen die Zeichen `/*` und `*/`, das sieht dann wie folgt aus.

```
/*
  Ein großes Programm bedarf vieler Kommentare um es verstehen zu können. Dabei
  besteht der Quellcode gut programmierter Programme zu mehr als drei Viertel
  aus Kommentar.
*/
```

Im Listing ist jede Programmzeile mit einem Semikolon beendet worden. Mit dem Semikolon wird angezeigt, daß der Ausdruck bzw. der Befehl zu Ende ist. Bei JavaScript ist der Befehl nicht zwingend notwendig, da das Ende der Zeile auch das Ende des Befehls ist. Das Semikolon wird aber gebraucht, wenn mehrere Befehle in einer Zeile stehen sollen.

```
var a = 42
var b = "Hallo Welt!"
var c = true
var d = null
```

Diese Schreibweise ist für JavaScript in Ordnung. Das Ende der Zeile beendet den Befehl.

```
var a = 42; var b = "Hallo Welt!"; var c = true; var d = null;
```

Ausgabe 42
Ausgabe “Hallo Welt!”
a ← 42
b ← “Hallo Welt!”
c ← true
d ← null
Ausgabe Variable a
Ausgabe Variable b
Ausgabe Variable c
Ausgabe Variable d

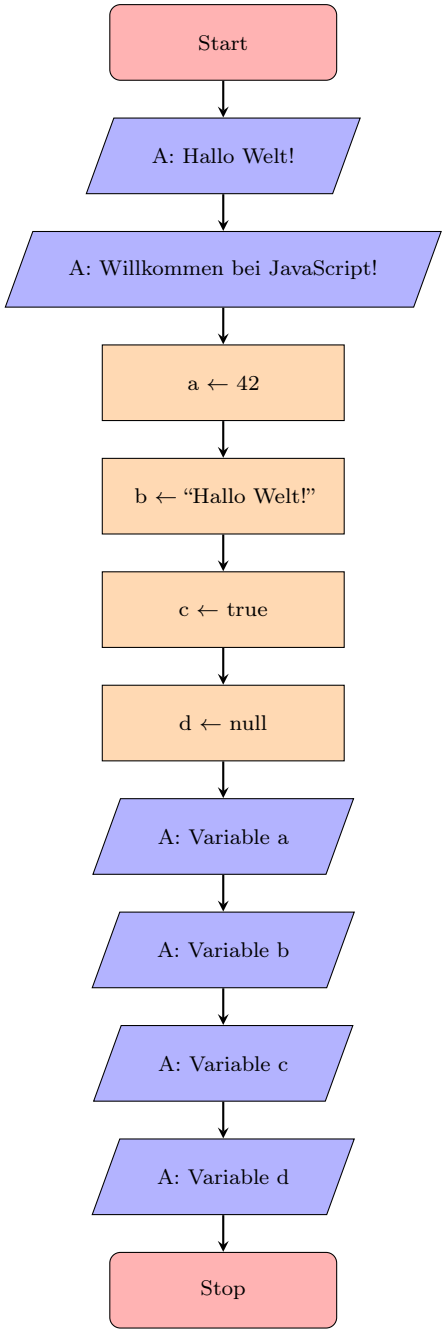

Abbildung 1.3: Struktogramm und Programmablaufplan zu Listing 1.7

Werden alle Befehle in einer Reihe geschrieben, dann müssen die Ausdrücke durch Semikola getrennt werden. Von dieser Schreibweise ist aber abzuraten, da sie den Programmcode sehr schwer lesbar macht. Du solltest jedem Befehl eine eigene Zeile gönnen.

```
var a = 42;
var b = "Hallo Welt!";
var c = true;
var d = null;
```

So ist es am schönsten. Jeder Ausdruck hat eine eigene Zeile und wird mit einem Semikolon beendet. Das Semikolon wird hier zwar nicht zwingend gebraucht, da es aber in anderen Sprachen wie Perl, PHP, Java, C, C++ etc. zwingend notwendig ist, solltest Du Dich gleich daran gewöhnen.

1.5 Eingabe: prompt()

Neben dem Ausgabefenster gibt es auch ein Fenster in dem der Benutzer etwas eingeben kann. Dieses Fenster wird mit dem Befehl `prompt()` geöffnet. Schauen wir uns mal das folgende Listing an. Das Programm soll ein Eingabefenster öffnen und nach dem Namen des Benutzers fragen. Dann soll das Programm den Benutzer namentlich begrüßen.

Listing 1.8. hallo_duda.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var name = "";
 4:
 5: // Programm ------------------------------------------------------------- //
 6: // Eingabe des Namens
 7: name = prompt("Gebe Deinen Namen ein!","");
 8: // Ausgabe über Alert-Fenster
 9: alert("Hallo " + name + "!\nWillkommen bei JavaScript!");
10: </script>
```

In Zeile 2 beginnt unser Programm. Hier passiert aber noch nichts. Alle Zeichen hinter dem doppelten Schrägstrich werden vom Browser ignoriert. Dies nennt man, wie Du schon gelernt hast, einen Kommentar. Das Programm funktioniert auch ohne Kommentare. Diese erhöhen aber die Lesbarkeit und erleichtern das Verstehen von Programmen. Die vielen Striche damit auch nicht notwendig, aber beim Lesen des Textes im Editor teilt diese Linie das Programm übersichtlicher in Abschnitte.

In Zeile 3 legen wir eine Variable im Speicher des Rechners an und geben ihr den Namen `name`. Diesen Vorgang nennen wir *Deklarieren*. In dieser Variablen soll später der eingegebene Name gespeichert werden.

Eingabe “Name” → name
Ausgabe “Hallo”, Variable name und “Willkommen bei JavaScript!”

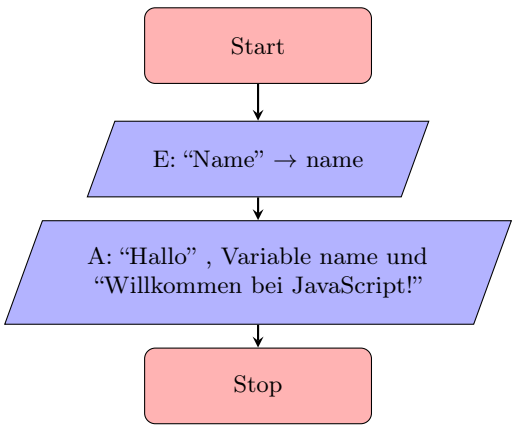

Abbildung 1.4: Struktogramm und Programmablaufplan zu Listing 1.8

Die Anweisung, ein Eingabefenster zu öffnen, erfolgt in Zeile 7. Nach der Eingabe des Namens gibt `prompt` die eingegebene Zeichenkette zurück. Durch das Gleichheitszeichen wird der Variablen `name` diese Zeichenkette zugewiesen.

Die Ausgabe erfolgt in Zeile 9. Da die Anweisung `alert` nur ein Argument erwartet, müssen die Zeichenketten zu einer zusammengefasst werden. Dies erfolgt mit Hilfe des Operators `+`.

Werden zwei Eingaben von einem Programm – wie in Listing 1.9 – gefordert, so müssen auch zwei Variablen verwendet und somit auch deklariert werden.

Listing 1.9. hallo_duda_v2.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var nachname = "";
 4: var vorname = "";
 5: var name = "";
 6:
 7: // Programm ------------------------------------------------------------- //
 8: // Eingabe des Namens
 9: nachname = prompt("Gebe Deinen Nachnamen ein!","");
10: vorname = prompt("Gebe Deinen Vornamen ein!","");
11:
12: // Erstelle den vollständigen Namen
13: name = vorname + " " + nachname;
14:
15: // Ausgabe über Alert-Fenster
16: alert("Hallo " + name + "!\nDu hier?");
17: </script>
```

A 1.1. Beschreibe in Stichworten die Funktion des obigen Programms Zeile für Zeile!

A 1.2. Erstelle für das obige Programm einen Programmablaufplan und ein Struktogramm.

1.6 Operatoren

Wie in Listing 1.8 und 1.9 zu sehen, ist es oft nötig Werte miteinander zu verbinden. Dies erfolgt durch sogenannte Operatoren, die Literale und Variablen untereinander und miteinander verbinden. Die meisten in JavaScript verwendeten Operatoren sind aus dem Matheunterricht bekannt. Eine Auswahl der wichtigsten Operatoren zeigt Tabelle 1.1.

Sollen zwei Werte miteinander verknüpft werden, so müssen sie mit einem Operator verbunden werden. Der Verknüpfungsoperator für Zeichenkette `+` wurde auch schon in den Listings 1.8 und 1.9 verwendet. Allerdings wird das Zeichen `+` auch als Additionsoperator verwendet.

Lassen wir doch mal den Browser für uns rechnen.

Listing 1.10. operator_rechnen.html

```
 1: <pre>
 2: <script type="text/javascript">
 3: // Rechnen
 4: document.writeln(17+4);
 5: document.writeln(17-4);
 6: document.writeln(17*4);
 7: document.writeln(17/4);
 8: document.writeln(17%4);
 9: </script>
10: </pre>
```

Sehr selten wird man in einem Programm zwei Literale berechnen lassen, denn das Ergebnis wäre ja immer gleich. Du wirst also in den meisten Fällen Operatoren im Zusammenhang mit Variablen verwenden. Ein

Operator	Name	Beispiel		Wert des Ausdrucks
Arithmetische Operatoren				
+	Addition	10 + 4		14
-	Subtraktion	10 - 4		6
*	Multiplikation	10 * 4		40
/	Division	10 / 40		2.5
%	Modulo	10 % 4		2
-	Negation	- (4 + 10)		-14
		- (4 - 10)		6
++	Inkrement	x = 10	x == 10	
		++x	x == 11	11
		x = 10	x == 10	
		x++	x == 11	10
--	Dekrement	x = 10	x == 10	
		--x	x == 9	9
		x = 10	x == 10	
		x--	x == 9	10
Zeichenketten Operatoren				
+	Konkatenation	"Rübe" + "zahl"		Rübezahl
		"Hallo" + " " + "Welt!"		Hallo Welt!

Tabelle 1.1: Operatoren

sehr wichtiger Operator ist dabei der Zuweisungsoperator =. Auf seiner linken Seite muss immer genau eine Variable stehen. Dieser Variable wird dann der Wert, der auf der rechten Seite steht, zugewiesen. Schau Dir das im folgenden Beispiel mal an.

Listing 1.11. operator_variablen.html

```
 1: <pre>
 2: <script type="text/javascript">
 3: // Rechnen mit Variablen und Operatoren
 4: var a = 13;
 5: var b = 7;
 6: var addition = a + b;
 7: var subtraktion = a - b;
 8: var multiplikation = a * b;
 9: var division = a / b;
10: var modulo = a % b;
11:
12: document.writeln(addition);
13: document.writeln(subtraktion);
14: document.writeln(multiplikation);
15: document.writeln(division);
16: document.writeln(modulo);
17: </script>
18: </pre>
```

Schauen wir uns die Operatoren mal in einer Anwendung an. Das folgende Listing fragt nach zwei Zahlen und gibt dann die Summe der Zahlen aus.

Listing 1.12. addierezweizahlen.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------- //
3: var zahl_1 = 0;
4: var zahl_2 = 0;
5: var ergebnis = 0;
```

```
 6: var ausgabe = "";
 7:
 8: // Eingabe ------------------------------------------------------------- //
 9: zahl_1 = Number(prompt("Die erste Zahl:", ""));
10: zahl_2 = Number(prompt("Die zweite Zahl:", ""));
11:
12: // Verarbeitung --------------------------------------------------------- //
13: // Berechnung
14: ergebnis = zahl_1 + zahl_2;
15: // Ausgabetext
16: ausgabe = "Die Summe aus " + zahl_1 + " und " + zahl_2 + " ist ";
17: ausgabe = ausgabe + ergebnis + "!";
18:
19: // Ausgabe -------------------------------------------------------------- //
20: alert(ausgabe);
21: </script>
```

A 1.3. Entferne die Anweisung `Number()` aus den Zeilen 9 und 10,

```
 9: zahl_1 = prompt("Die erste Zahl:", "");
10: zahl_2 = prompt("Die zweite Zahl:", "");
```

Teste nun das Script. Erkläre die Aufgabe der Funktion `Number()`.

Etwas verwirrend in JavaScript ist die doppelte Verwendung des Operators +. Einmal steht er für die Addition von zwei Werten. Wenn es aber um Texte geht, dann addiert er diese nicht (Wie soll man denn auch zwei Texte addieren können?), sondern fügt die Texte aneinander, wie im folgenden Beispiel zu sehen ist.

Listing 1.13. operator_text.html

```
 1: <pre>
 2: <script type="text/javascript">
 3: // Zeichenketten zusammenfügen ------------------------------------------ //
 4: var vorname = "B&auml;rtram";
 5: var nachname = "von Brumm";
 6: document.writeln("Lirum"+"larum"+"l&ouml;ffel"+"stiel");
 7: document.writeln("Mein Name ist " + nachname + ". " + vorname + " "
 8:                  + nachname + "!");
 9: </script>
10: </pre>
```

Die Zeilen 7 bis 8 enthalten einen Befehl. Früher hat JavaScript das Ende eines Befehls dann erkannt, wenn die Zeile zu Ende war. Da manche Befehlszeilen aber durchaus länger werden können, führte da oft zu ungewollten und nervigen Aufteilungen eines Befehls auf mehrere Befehlen. In den neueren Versionen unterstützt JavaScript auch die Aufteilung eines Befehls auf zwei Zeilen. Der Interpreter untersucht den laufen Befehl, ob er überhaupt beendet werden kann. Dann schaut er in der folgenden Zeile nach, ob der Befehl hier weitergeführt wird. Ist dies der Fall, dann geht der Befehl halt über zwei oder mehr Zeilen, wie es schon lange in anderen Programmiersprachen üblich ist.

A 1.4. Diese doppelte Belegung des Operators + kann zu ungewollten Ergebnissen führen. Erkläre das unterschiedliche Verhalten der beiden Ausgaben im folgenden Listing 1.14.

Listing 1.14. operator_falle.html

```
1: <pre>
2: <script type="text/javascript">
3: // Vorsicht Falle
4: document.writeln("2 + 3 = "+(2+3));
5: document.writeln("2 + 3 = "+2+3);
6: </script>
7: </pre>
```

A 1.5. Erweitere das Skript 1.12 so, dass hintereinander Addition, Subtraktion, Multiplikation, Division und Modulo für die eingegebenen Zahlen ausgeführt wird.

1.6.1 Postfix und Präfix Dekrement und Inkrement

Eine der häufigsten arithmetischen Operationen ist das Erhöhen oder Erniedrigen eines Variableninhalts um den Wert 1. Dafür wurden die Operatoren ++ und -- eingeführt. Für die Funktionsweise ist es wichtig, ob der Operator vor oder hinter der Variablen steht.

Beispiel:

```
a = 5;
b = a++;
```
$\equiv$
```
a = 5;
b = a;
a = a + 1;
```

```
a = 5;
b = ++a;
```
$\equiv$
```
a = 5;
a = a + 1;
b = a;
```

Listing 1.15. inkrement.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var x = 0;
 4:
 5: // Prozeduren ---------------------------------------------------------- //
 6: x++;
 7: alert("A " + x);
 8: alert("B " + x++);
 9: alert("C " + x);
10: alert("D " + ++x);
11: alert("E " + x);
12: alert("F " + --x);
13: alert("G " + x);
14: alert("H " + x--);
15: alert("I " + x);
16: x--;
17: alert("J " + x);
18: </script>
```

A 1.6. Führt das obige Listing 1.15 aus und notiert die Ergebnisse. Macht euch daran die Funktionsweise der ++ und -- Operatoren klar.

A 1.7. Die Fläche und der Umfang eines Rechtecks ergibt sich aus der Breite und Länge eines Rechtecks. Es gilt

$$\text{Fläche} \qquad A = a \cdot b$$

$$\text{Umfang} \qquad A = 2 \cdot a + 2 \cdot b$$

Schreibt ein Skript, dass die Seiten als Eingabe erwartet und die Fläche und den Umfang des Rechtecks als Ergebnis ausgibt.

1.6.2 eval()

Das nächste Listing stellt einen kleinen Taschenrechner da. Nach der Deklaration der Variablen wird ein mathematischer Ausdruck mittels der bekannten Funktion `prompt()` *eingegeben*. Die neue Funktion `eval()` sorgt für die *Verarbeitung* und berechnet den mathematischen Ausdruck. Mittels `alert()` wird dann das Ergebnis *ausgegeben*. Das folgende Listing sollte sich damit von selbst erklären.

Listing 1.16. taschenrechner.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------- //
3: var eingabe = "";
4: var ergebnis = "";
```

```
 5:
 6: // Eingabe ------------------------------------------------------------------ //
 7: eingabe = prompt("Taschenrechner", "");
 8:
 9: // Verarbeitung --------------------------------------------------------------- //
10: ergebnis = eval(eingabe);
11:
12: // Ausgabe -------------------------------------------------------------------- //
13: alert(eingabe + " = " + ergebnis);
14: </script>
```

Die Funktion `eval()` in Zeile 10 bewirkt, daß die als Parameter angegebene Zeichenkette so ausgeführt wird, als ob sie direkt im JavaScript-Code stehen würde.

So etwas kann man natürlich auch "quick and dirty" programmieren.

Listing 1.17. taschenrechner2.html

```
1: <script type="text/javascript">
2:  alert(eval(prompt("Taschenrechner","")));
3: </script>
```

A 1.8. Probieren Sie doch mal die folgenden Zeichenketten im Taschenrechner aus.

a) `740736/6`
b) `8641969%7`
c) `1+6*6`
d) `(1+6)*6`
e) `"Java" + "Script"`
f) `screen.height`
g) `screen.width * screen.height`
h) `a=4; b=3; a*b`
i) `a=5; a++`
j) `a=5; ++a`

1.7 Zahlen: Die Klasse `Number`

Die Funktion `Number()` haben wir schon in Listing 1.12 kennengelernt, in dem wir die Eingabe in Textform in eine Zahl umgeformt haben. Die Klasse `Number` stellt weitere Eigenschaften und Methoden für den Umgang mit Zahlen zur Verfügung. Die Methoden der Klasse können direkt auf jede Variable angewendet werden, die einen Wert vom Typ Zahl hat.

1.7.1 Eigenschaften

Die Eigenschaften `MAX_VALUE` und `MIN_VALUE` geben die größte Zahl und die kleinste von Null verschiedene Zahl wieder, die JavaScript verarbeiten kann.

Listing 1.18. number_values.html

```
1: <pre><script type="text/javascript">
2: document.writeln("Die gr&ouml;&szlig;te Zahl: " + Number.MAX_VALUE);
3: document.writeln("Die kleinste von Null verschiedene Zahl: " + Number.MIN_VALUE);
4: </script></pre>
```

Das Ergebnis sieht dann z.B. so aus:

```
Die größte Zahl: 1.7976931348623157e+308
Die kleinste von Null verschiedene Zahl: 5e-324
```

Daneben gibt es auch Entsprechungen für bestimmte Zustände, die Zahlenvariablen annehmen können. So ist das Resultat keine gültige Zahl (`NaN`), unterschreitet den Wertebereich (`NEGATIVE_INFINITY`) oder überschreitet ihn (`POSITIVE_INFINITY`).

1.7.2 toExponential()

Die Methode `toExponential()` liefert die Zahl in der Exponentialschreibweise zurück. So werden selbst Zahlen, die JavaScript normal darstellen würde, in der exponentiellen Schreibweise dargestellt. Z.B. würde die Zahl 2 300 000 als 2.3e6 wiedergegeben oder 0.002 als 2e-3.

Listing 1.19. number_toexponential.html

```
1: <script type="text/javascript">
2: // Variablen und Eingabe ------------------------------------------------- //
3: var zahl = parseFloat(prompt("Gebe eine Zahl ein!", ""));
4: // Verarbeitung und Ausgabe ------------------------------------------------ //
5: alert (zahl.toExponential());
6: </script>
```

A 1.9. Teste die folgenden Zahlen mit dem obigen Skript: 10, 100, 1000, 1 000 000, 0.006, 1234567, 0.0023, 42.0815

1.7.3 toFixed()

Oft liefert eine Multiplikation eine Zahl mit vielen Nachkommastellen. Meistens werden davon aber nur wenige gebraucht. So muss z.B. auf den Nettopreis 19% Mehrwertsteuer aufgeschlagen werden. Diese Rechnung führt in den meisten Fällen zu sehr krummen Ergebnissen. Im alltäglichen Gebrauch wird aber ein Preis auf zwei Nachkommastellen (Cents) genau angegebenen. Die Methode `toFixed(`*`stellen`*`)` rundet kaufmännisch die Zahl auf die angegebene Anzahl von Nachkommastellen.

Das folgende Beispiel berechnet den Bruttopreis aus dem Nettopreis. Dabei gibt das Programm den berechneten Bruttopreis und den gerundeten Bruttopreis wieder.

Listing 1.20. number_tofixed.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var netto = 0;
 4: var brutto = 0;
 5: var mwst = 19;  // Mehrwertsteuersatz in Prozent
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: netto = parseFloat(prompt("Gebe den Nettopreis an!", ""));
 9:
10: // Verarbeitung ------------------------------------------------------------- //
11: brutto = netto * (1 + mwst/100);
12:
13: // Ausgabe ------------------------------------------------------------- //
14: alert ("Bruttopreis: " + brutto + " => " + brutto.toFixed(2) + " Euro");
15: </script>
```

A 1.10. Teste das Skript aus Listing 1.20 mit Nettopreisen wie z.B. 19.99, 99.99 und 21.11.

1.7.4 toPrecision()

Während `toFixed()` die Zahl auf eine vorgegebene Anzah von Nachkommazahlen reduziert, reduziert die Methode `toPrecision()` die Zahl auf eine bestimmte Anzahl signifikanter Ziffer. So hat z.B. die Zahl 3 200 000 nur zwei signifikante Ziffer, da man die Zahl auch als 3,2 Millionen bzw. 3.2e6 angegeben kann. Reduziert

man eine Zahl z.B. auf drei signifikante Ziffer, dann wird aus 3,567 die Zahl 3,57, aus 54 321 die Zahl 54 300 und aus 0,001234 die Zahl 0,00123.

Das folgende Beispiel zeigt eine Anwendungsmöglichkeit. In dem Skript wird aus der zurückgelegten Strecke und der dafür benötigten Zeit die Durchschnittsgeschwindigkeit berechnet. Bei realen Messungen kann es aber bei einer Division zu einer unendlichen Anzahl von Ziffern kommen, von denen die meisten keine Relevanz fürs Messergebnis haben. Mit der Methode `toPrecision()` können wir das Ergebnis auf eine bestimmte Zahl von signifikanten Ziffern reduzieren.

Listing 1.21. number_toprecision.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------- //
var s = 0;         // Strecke in Meter
var t = 0;         // Zeit in Sekunden
var v = 0;         // Geschwindigkeit in Meter pro Sekunde

// Eingabe --------------------------------------------------------------- //
s = parseFloat(prompt("Strecke in Metern", 100));
t = parseFloat(prompt("Zeit in Sekunden", 9.84));

// Verarbeitung ---------------------------------------------------------- //
v = s / t;

// Ausgabe --------------------------------------------------------------- //
document.writeln("Die durchschnittliche Geschwindigkeit betrug ... <br>");
document.writeln("... auf 1 Stelle genau " + v.toPrecision(1) + " m/s<br>");
document.writeln("... auf 2 Stellen genau " + v.toPrecision(2) + " m/s<br>");
document.writeln("... auf 3 Stellen genau " + v.toPrecision(3) + " m/s<br>");
document.writeln("... auf 4 Stellen genau " + v.toPrecision(4) + " m/s<br>");
document.writeln("... auf 5 Stellen genau " + v.toPrecision(5) + " m/s<br>");
document.writeln("... auf 6 Stellen genau " + v.toPrecision(6) + " m/s<br>");
document.writeln("... auf 7 Stellen genau " + v.toPrecision(7) + " m/s<br>");
document.writeln("... auf 8 Stellen genau " + v.toPrecision(8) + " m/s<br>");
</script>
```

1.7.5 `toString()`

Um die Methoden für Bearbeitung von Texten zu nutzen, müssen Zahlenwerte in Zeichenketten umgewandelt werden. Dies erfolgt durch die Methode `toString()`. Ohne Angabe eines Wertes gibt die Methode die Zahl im Dezimalsystem zurück. Übergibt man an die Methode eine Zahl, dann wird diese Zahl als Basis des Zahlensystems angesehen, in der die Zahl zurückgegeben wird. Also liefert `toString(2)` die Zahl als Binärzahl zurück und als `toString(16)` als Hexadezimalzahl.

Listing 1.22. number_tostring.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------- //
var zahl = 0;

// Eingabe --------------------------------------------------------------- //
zahl = parseFloat(prompt("Geben Sie eine Zahl ein!", 42));

// Ausgabe --------------------------------------------------------------- //
document.writeln("Die eingegebene Zahl lautet im ... <br>");
document.writeln("... Bin&auml;rsystem: " + zahl.toString(2) + "<br>");
document.writeln("... Hexalsystem: " + zahl.toString(6) + "<br>");
document.writeln("... Octalsystem: " + zahl.toString(8) + "<br>");
document.writeln("... Dezimalsystem: " + zahl.toString(10) + "<br>");
document.writeln("... Duodezimalsystem: " + zahl.toString(12) + "<br>");
```

```
15: document.writeln("... Hexadezimalsystem: " + zahl.toString(16) + "<br>");
16: </script>
```

1.8 Mathematische Funktionen: Die Klasse Math

Natürlich stehen außer den Operatoren für Addition, Subtraktion, Multiplikation, Division und Modulo in JavaScript auch weitere mathematische Methoden und Konstanten zur Verfügung. Diese werden in JavaScript durch die Klasse `Math` zur Verfügung gestellt. Im folgenden werden einige Methoden und Eigenschaften vorgestellt. Auf weitere Methoden gehen wir später ein.

1.8.1 Eigenschaften

Die Eigenschaften der Klasse `Math` sind mathematische Konstanten wie die Kreiszahl π (Pi) und die Eulersche Konstante e oder auch häufig verwendete Ergebnisse wie die Wurzel aus 2 ($\sqrt{2}$). Das folgende Skript gibt einen Überblick über alle Konstanten.

Listing 1.23. math_eigenschaften.html

```
 1: <pre><script type="text/javascript">
 2: // Die Eigenschaften des Objekts Math ---------------------------------------- //
 3: document.writeln("Math.E       (Eulersche Konstante) ......... " + Math.E);
 4: document.writeln("Math.LN2     (natürlicher Logarithmus von 2)  " + Math.LN2);
 5: document.writeln("Math.LN10    (natürlicher Logarithmus von 10) " + Math.LN10);
 6: document.writeln("Math.LOG2E   (Logarithmus zur Basis 2 von E)  " + Math.LOG2E);
 7: document.writeln("Math.LOG10E  (Logarithmus zur Basis 10 von E) " + Math.LOG10E);
 8: document.writeln("Math.PI      (Konstante PI) .................. " + Math.PI);
 9: document.writeln("Math.SQRT1_2 (Quadratwurzel aus 0,5) ........ " + Math.SQRT1_2);
10: document.writeln("Math.SQRT2   (Quadratwurzel aus 2): ......... " + Math.SQRT2);
11: </script></pre>
```

A 1.11. Der Umfang und die Fläche eines Kreises hängen von seinem Radius r ab. Im Gegensatz zum Quadrat wird für die Berechnung dieser Werte die Kreiskonstante π benötigt. Es gilt:

$$\text{Fläche} \quad A = 2\pi \cdot r \qquad \text{Umfang} \quad A = \pi \cdot r \cdot r$$

Schreibt ein Skript, dass den Radius r als Eingabe erwartet und die Fläche und den Umfang des Kreises als Ergebnis ausgibt.

1.8.2 Die größere oder die kleinere Zahl?

Die Methode `Math.min(Zahl1, Zahl2)` liefert den Wert der kleineren Zahl zurück und `Math.max(Zahl1, Zahl2)` den Wert der größeren Zahl.

Listing 1.24. math_max.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var a = 0;
 4: var b = 0;
 5: var max = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: a = Number(prompt("Geben Sie eine Zahl ein!", ""));
 9: b = Number(prompt("Geben Sie eine weitere Zahl ein!", ""));
10:
11: // Verarbeitung ------------------------------------------------------------- //
12: max = Math.max(a, b);
13:
```

```
14: // Ausgabe ------------------------------------------------------------- //
15: alert("Die groessere Zahl von " + a + " und " + b + " ist " + max + "!");
16: </script>
```

A 1.12. Verändere das obige Skript so, dass die kleinere von beiden Zahlen ausgegeben wird.

1.8.3 Runden

Für das Runden von Zahlen stehen uns in JavaScript drei Methoden zur Verfügung. Die Methode `Math.ceil` rundet auf die nächste ganze Zahl auf, die Methode `Math.floor` rundet ab und die kaufmännische Rundung übernimmt die Methode `Math.round`. Bitte denkt daran, dass JavaScript anstatt eines Kommas einen Punkt verwendet um die Nachkommastellen von der ganzen Zahl abzutrennen. Also für z.B. 5,34 muss 5.34 eingegeben werden.

Listing 1.25. math_runden.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl = 0;
 4: var gerundet = 0;
 5: var aufgerundet = 0;
 6: var abgerundet = 0;
 7:
 8: // Eingabe ------------------------------------------------------------- //
 9: zahl = Number(prompt("Geben Sie eine Zahl mit Nachkommastellen ein!", ""));
10:
11: // Verarbeitung ---------------------------------------------------------- //
12: gerundet = Math.round(zahl);
13: aufgerundet = Math.ceil(zahl);
14: abgerundet = Math.floor(zahl);
15:
16: // Ausgabe -------------------------------------------------------------- //
17: alert("Die Zahl " + zahl + " ist gerundet " + gerundet + "!");
18: alert("Die Zahl " + zahl + " ist abgerundet " + abgerundet + "!");
19: alert("Die Zahl " + zahl + " ist aufgerundet " + aufgerundet + "!");
20: </script>
```

Diese Methoden runden immer bis zur nächsten Ganzzahl auf oder ab. Oft wird aber gefordert, dass ein Zahlen auf zwei Nachkommastellen gerundet werden soll. Hier verwendet man ein einfaches Verfahren. Um auf zwei Nachkommastellen zu Runden wird die Zahl erst mit 100 multipliziert, dann gerundet und schließlich das Ergebnis wieder durch 100 geteilt. Heraus kommt eine Zahl mit zwei (oder weniger) Nachkommastellen. Bei drei Nachkommastellen muss man mit 1000 multiplizieren und dividieren.

Listing 1.26. math_runden_2nks.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl = 0;
 4: var gerundet = 0;
 5:
 6: // Eingabe ------------------------------------------------------------- //
 7: zahl = Number(prompt("Geben Sie eine Zahl mit Nachkommastellen ein!", ""));
 8:
 9: // Verarbeitung ---------------------------------------------------------- //
10: gerundet = Math.round(zahl*100)/100;
11:
12: // Ausgabe -------------------------------------------------------------- //
13: alert("Die Zahl " + zahl + " auf 2 Nachkommstellen gerundet: " + gerundet);
14: </script>
```

A 1.13. Verändere das Skript 1.26 so, dass die Zahl auf eine (drei) Nachkommastellen gerundet ausgegeben wird.

A 1.14. Untersuche ob die Verwendung von `.toFixed()` besser oder schlechter geeignet ist als Math.round().

1.8.4 Potenz

Im Skript 1.26 wird für ein Runden auf zwei Nachkommastellen der Faktor 100 verwendet. Für drei Nachkommastellen müsste man 1000 und für vier Nachkommastellen 10 000 verwenden. Es ist sind also immer Potenzen von 10 gefragt, also z.B. 10^2, 10^3 bzw. 10^4. Zur Berechnung von Potenzen wird die Methode `Math.pow(Basis, Exponent)` verwendet. Für 3^4 müsste man dann die Anweisung schreiben `Math.pow(3,4)`.

Verbessern wir das Skript 1.26 so, dass auch die Anzahl der Nachkommastellen eingetragen werden kann.

Listing 1.27. math_runden_pow.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var zahl = 0;
 4: var nks = 0;
 5: var gerundet = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: zahl = Number(prompt("Geben Sie eine Zahl mit Nachkommastellen ein!", ""));
 9: nks = Math.floor(Number(prompt("Anzahl der Nachkommastellen eingeben", "")));
10:
11: // Verarbeitung ------------------------------------------------------------- //
12: gerundet = Math.round(zahl*Math.pow(10,nks))/Math.pow(10,nks);
13:
14: // Ausgabe ------------------------------------------------------------- //
15: alert(zahl + " auf " + nks + " Nachkommstellen gerundet: " + gerundet);
16: </script>
```

A 1.15. Erläutere, warum in Zeile 9 die Methode `Math.floor` verwendet wird.

1.8.5 Betragsfunktion

Die Betragsfunktion macht aus einer negativen Zahl eine positive Zahl. Dies wird z.B. benötigt, wenn der Abstand zwischen zwei Punkten bestimmt werden soll. Berechnet man den Abstand durch Differenz, so kann das Ergebnis positiv oder negativ sein. Da ein Abstand immer nur positiv ist muss das Ergebnis vom Vorzeichen befreit werden. Im folgenden Beispiel wird der Abstand zwischen zwei Zahlen ermittelt. Egal, ob die erste oder die zweite Zahl die größere ist, das Ergebnis ist immer positiv.

Listing 1.28. math_betrag.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var a = 0;
 4: var b = 0;
 5: var unterschied = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: a = Number(prompt("Gebe eine Zahl ein!", ""));
 9: b = Number(prompt("Gebe zweite Zahl ein!", ""));
10:
11: // Verarbeitung ------------------------------------------------------------- //
12: unterschied = Math.abs(a-b);
13:
14: // Ausgabe ------------------------------------------------------------- //
15: alert("Die Differenz von " + a + " und " + b + " ist " + unterschied + ".");
16: </script>
```

1.8.6 Zufallsfunktion

Der Zufall ist bei den meisten Spielen ein wichtiges Element. Oder kannst Du Dir Mau-Mau ohne gemischte Karten oder Meiern ohne Würfel vorstellen? Auch für die Erzeugung von Zufallszahlen hat JavaScript in der Math-Klasse eine Methode. Diese Methode heißt `random()` und erzeugt eine Kommazahl zwischen 0 und 1.[1]

Um damit einen Würfel zu simulieren, müssen wir noch etwas rechnen. Als erstes multiplizieren wir die Zahl mit sechs. Dann sollte das Ergebnis zwischen 0 und 6 liegen. Da es keinen Würfel mit Kommazahlen gibt, sondern ganzzahlige Ergebnisse gefragt sind, runden wir das Ergebnis mit `Math.floor()` ab. Jetzt sind 0, 1, 2, 3, 4 und 5 die möglichen Ergebnisse. Also addieren wir noch eins dazu und schon haben wir unseren Würfel.

Listing 1.29. math_random.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------ //
wuerfel_1 = 0;
wuerfel_2 = 0;

// Verarbeitung --------------------------------------------------------- //
wuerfel_1 = Math.floor(Math.random()*6)+1;
wuerfel_2 = Math.floor(Math.random()*6)+1;

// Ausgabe -------------------------------------------------------------- //
alert("Ihr Wurf: " + wuerfel_1 + " " + wuerfel_2);
</script>
```

1.8.7 Quadratwurzel

Die Fläche A eines Quadrats mit der Seitenlänge x erhält man, wenn man diese mit sich selbst multipliziert: $A = x \cdot x$. In JavaScript kann man dies mit der Anweisung

```
flaeche = Math.pow(seite,2);
```

erreichen. Allerdings ist die Berechnungsroutine in der Potenzfunktion etwas ungenau. Für das Quadrieren bietet sich die genauere und schnellere Variante

```
flaeche = seite * seite;
```

an. Aber natürlich sollte es auch umgekehrt möglich sein aus der Fläche eines Quadrats seine Seitenlänge zu bestimmen. Dafür müssen wir die Gegenoperation zum Quadrieren verwenden, das sogenannte Wurzelziehen oder Radizieren (von lat. Radix, die Wurzel). Es gilt: $x = \sqrt{A}$. Dabei ist in diesem Fall A der Radikant, aus dem die Wurzel gezogen werden soll. Die Wurzelmethode heißt in JavaScript `Math.sqrt(Radikant)`.

Listing 1.30. math_wurzel.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------ //
var flaeche = 0;
var seite = 0;
var umfang = 0;

// Eingabe -------------------------------------------------------------- //
flaeche = Number(prompt("Bitte Fläche des Quadrats eingeben:", ""));

// Verarbeitung --------------------------------------------------------- //
seite = Math.sqrt(flaeche);
umfang = 4 * seite;
```

[1] Das Ergebnis 1 ist in dieser Menge von Zufallszahlen nicht dabei.

```
13:
14: // Ausgabe ------------------------------------------------------------------ //
15: document.writeln("Quadrat<br>");
16: document.writeln("Fl&auml;che: " + flaeche + "<br>");
17: document.writeln("Seite: " + seite + "<br>");
18: document.writeln("Umfang: " + umfang);
19: </script>
```

1.9 Zeichenketten-Funktionen: Die Klasse `String`

Verschlüsselung ist eine der Anwendungen, mit der wir den Umgang mit Zeichenketten und JavaScript üben wollen. Dafür brauchen wir Werkzeuge die Texte in Form Zeichenketten bearbeiten können.

Für Zeichenketten gibt es ebenfalls eine Klasse. Diese Klasse trägt den Namen `String`. Allerdings taucht sie selten explizit mit Ihrem Namen auf, wie `Math` sondern die Methoden werde direkt auf Variablen angewendet. So gibt die Methode `toString()` die Zeichenkette als Zeichenkette zurück. Mhhh, nicht gerade spannend. Deshalb kann man den Befehl auch weglassen und nur den Variablennamen nehmen. Die Eigenschaft `length` liefert die Länge der Zeichenkette zurück.

Listing 1.31. string.html

```
1: <script type="text/javascript">
2: // Eingabe ------------------------------------------------------------------
3: var str = prompt("Einen Text eingeben.");
4:
5: // Ausgabe -----------------------------------------------------------------
6: alert("Der Text '" + str.toString() + "' hat " + str.length + " Zeichen.");
7: </script>
```

1.9.1 Zeichen herauspicken

Eine Zeichenkette ist für den Computer einfach eine Liste von Zeichen, die durch Zahlen repräsentiert werden. JavaScript nutzt für die Codierung UTF-16, die zwei Bytes (16 Bit) pro Zeichen benutzt. Mit den Methoden `charAt()` und `charCodeAt()` kann auf die einzelnen Zeichen zurückgegriffen werden. Die Nummerierung der Zeichen erfolgt ab der 0. Also greift man mit `charAt(0)` auf das erste Zeichen der Zeichenkette zurück. Während `charAt()` eine Zeichenkette mit dem Zeichen an der angegeben Stelle zurückgibt, gibt `charCodeAt()` den dafür passenden Zahlenwert der Codierung zurück. Dieser liegt zwischen 0 und 65535.

Listing 1.32. string_char.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------------
 3: var str = "Das also ist des Pudels Kern.";
 4:
 5: document.writeln("<pre>");
 6: document.writeln("Der Text lautet: '" + str + "'");
 7:
 8: // Zeichen an einer bestimmten Position.
 9: document.writeln("charAt(): " + str.charAt());
10: document.writeln("charAt(0): " + str.charAt(0));
11: document.writeln("charAt(17): " + str.charAt(17));
12:
13: // Die UTF-16 Codierung des Zeichens an einer bestimmten Position
14: document.writeln("charCodeAt(): " + str.charCodeAt());
15: document.writeln("charCodeAt(0): " + str.charCodeAt(0));
16: document.writeln("charCodeAt(17): " + str.charCodeAt(17));
17:
18: document.writeln("</pre>");
19: </script>
```

A 1.16. Geben Sie an, welchen Index die Methoden `charAt()` und `charCodeAt()` verwenden, wenn keiner angegeben worden ist.

A 1.17. Schreiben Sie ein Programm, mit dem Sie die Zahlencodierung der Großbuchstaben A-Z, Kleinbuchstaben a-z und das Leerzeichen ermitteln können. Notieren Sie die ermittelten Werte sinnvoll.

1.9.2 Großbuchstaben und Kleinbuchstaben

Damit die Beispiele für die Verschlüsselung einfach bleiben, werden wir nur mit Großbuchstaben, Zahlen und Leerzeichen arbeiten. Die Umwandlung in Großbuchstaben stellt JavaScript mit der String-Methode `toUpperCase()` zur Verfügung. Diese liefert einen String zurück, in dem alle Kleinbuchstaben zu Großbuchstaben umgewandelt worden sind.

Listing 1.33. string_upper.html

```
 1: <script type="text/javascript">
 2: // Eingabe ------------------------------------------------------------------
 3: var str = prompt("Einen Text eingeben.");
 4:
 5: // In Großbuchstaben umwandeln
 6: var gross = str.toUpperCase();
 7:
 8: // Ausgabe -----------------------------------------------------------------
 9: alert(gross);
10: </script>
```

Es gibt natürlich auch eine String-Methode, die einen Text in Kleinbuchstaben umwandelt. Diese heißt `toLowerCase()`.

A 1.18. Ändere das Programm so ab, dass es den eingegeben Text in Kleinbuchstaben ausgibt.

Kapitel 2

Entscheidungen und Verzweigungen

Die Programme, die bis jetzt behandelt wurden, waren linear aufgebaut. Es erfolgte eine Eingabe, dann wurde gerechnet und dann wurde der Wert ausgegeben. Für reine Berechnungen reicht das, aber schon die Eingabe stellt uns vor Probleme. Denn was passiert, wenn der Anwender anstatt die gewünschte Zahl in Ziffern einzugeben die Zahl mit Buchstaben ausschreibt. Die Zeichenfolge "fünf" wird die Funktion `Number()` nicht als Zahl erkennen und einen falschen Wert zurückgeben, der dann natürlich auch zu einem falschen Ergebnis führt. Hier müsste das Programm entscheiden, ob die Eingabe auch sinnvoll ist. Schauen wir uns nun Strukturen an, die einem Skript erlauben Entscheidungen zu treffen.

2.1 Wahrheitswerte und Vergleichsoperatoren

Entscheidungen werden in einem Programm dadurch getroffen, daß ein Ausdruck entweder *wahr* oder *falsch* ist. Für diese Werte stehen die reservierten Worte `true` und `false`. Die Aussagen ergeben sich dadurch, dass zwei Werte mit einem Vergleichsoperator in Verbindung gebracht werden. Die dadurch entstehende Aussage ist entweder wahr oder falsch. Z.B. ist die Aussage *5 ist größer als 4* wahr, die Aussage *7 ist kleiner als 6* dagegen falsch. Tabelle 2.1 zeigt eine Liste der Vergleichsoperatoren mit Beispielen für ihre Anwendung.

2.2 Der dreifach Operator ? :

Der dreifach Operator `? :` ist die einfachste Form einer Entscheidung. Je nachdem ob der Vergleich *wahr* oder *falsch* ist, wird entweder der erste oder der zweite Wert zurückgegeben.

Vergleich `?` *WertWennWahr* `:` *WertWennFalsch*

Das folgende Script sucht aus zwei Zahlen die größere heraus.

Listing 2.1. entscheidung_max.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var a = 0;
 4: var b = 0;
 5: var max = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: a = Number(prompt("Geben Sie eine Zahl ein!", ""));
 9: b = Number(prompt("Geben Sie eine weitere Zahl ein!", ""));
10:
11: // Verarbeitung ---------------------------------------------------------- //
12: max = (a>b) ? a : b;
13:
14: // Ausgabe -------------------------------------------------------------- //
15: alert("Die groessere Zahl von " + a + " und " + b + " ist " + max + "!");
16: </script>
```

Operator	Name	Beispiel	Wert des Ausdrucks
==	Gleichheit	`1 == 1`	true
		`true == true`	true
		`"Rose" == "Rose"`	true
		`2 + 2 == 5`	false
		`true == false`	false
		`"Rose" == "Tulpe"`	false
!=	Ungleichheit	`1 != 1`	false
		`true != true`	false
		`"Rose" != "Rose"`	false
		`2 + 2 != 5`	true
		`true != false`	true
		`"Rose" != "Tulpe"`	true
<	kleiner als	`4 < 5`	true
		`7 < 5`	false
		`5 < 5`	false
		`"abc" < "abd"`	true
		`"abc" < "ABC"`	false
>	größer als	`4 > 5`	false
		`7 > 5`	true
		`5 > 5`	false
		`"abc" > "abd"`	false
		`"abc" > "ABC"`	true
<=	kleiner gleich	`4 <= 5`	true
		`7 <= 5`	false
		`5 <= 5`	true
		`"abc" <= "abd"`	true
		`"abc" <= "ABC"`	false
>=	größer gleich	`4 >= 5`	false
		`7 >= 5`	true
		`5 >= 5`	true
		`"abc" >= "abd"`	false
		`"abc" >= "ABC"`	true

Tabelle 2.1: Vergleichsoperatoren

Das Script kommt Dir bekannt vor? Natürlich! Es ist fast identisch mit Listing 1.24 und hat die gleiche Aufgabe. Die Anweisung

```
max = (a>b) ? a : b;
```

erledigt die gleiche Aufgabe wie die Anweisung

```
max = Math.max(a, b);
```

Allerdings können wir die erste Anweisung flexibler gestalten. In JavaScript können die eigentlich für Zahlen gedachten Operatoren wie *größer als* und *kleiner als* auch auf Zeichenketten (Texte) angewendet werden. Aber was bedeutet denn in diesem Fall größer oder kleiner? Dazu müssen wir uns einfach mal anschauen, wie ein Rechner mit Texten umgeht. Für einen Rechner sind Buchstaben nichts anderes als Zahlen. Jedem Buchstaben wird ein bestimmter Zahlenwert zugeordnet. Da dies alphabetisch passiert hat der Buchstabe a einen kleineren Wert als der Buchstabe b. Und so bedeutet die Aussage: *Wort A ist kleiner als Wort B*, dass Wort A vor Wort B in der alphabetischen Sortierung kommt.

Im folgenden Script wurde der Code aus dem vorherigen Listing so angepasst, dass er auch für Namen passt.

Listing 2.2. entscheidung_alphabet.html

```
1: <script type="text/javascript">
```

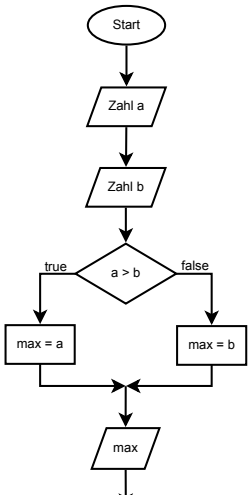

Abbildung 2.1: Ablaufdiagramm zu Listing 2.1

```
 2: // Variablen ------------------------------------------------------------- //
 3: var a = "";
 4: var b = "";
 5: var max = "";
 6: var min = "";
 7:
 8: // Eingabe -------------------------------------------------------------- //
 9: a = prompt("Geben Sie einen Namen ein!", "");
10: b = prompt("Geben Sie einen weiteren Namen ein!", "");
11:
12: // Verarbeitung ---------------------------------------------------------- //
13: max = (a>b) ? a : b;
14: min = (a<b) ? a : b;
15:
16: // Ausgabe ---------------------------------------------------------------- //
17: alert(min + " kommt vor " + max + "!");
18: </script>
```

Die beiden Skripte enthalten jeweils eine Verzweigung im Programmablauf. Bei einer Verzweigungen des Typs Wenn-Dann-Sonst b ist ein Skript noch übersichtlich. Häufen sich die Verzweigungen, dann ist es notwendig vorher den Programmablauf zu planen. Dies kann auf verschiedene Arten und Weisen erfolgen. Eine Methode sind die sogenannten Struktogramme bzw. Ablauflaufdiagramme, wie in Abbildung 2.1 zu sehen. Die Aufgaben des Programms werden dort durch Pfeile verbundene Figuren dargestellt. Die Pfeile zeigen den Weg des Programms. Ein Oval stellt Anfang oder Ende des Programms dar. Rechtecke enthalten die allgemeine Anweisungen an das Programm, wie z.B. Berechnungen. Parallelogramme stehen für Eingabe und Ausgabe und die Raute für Verzweigungen.

Eine weitere Methode ein Programm zu planen ist der Pseudocode. Hier werden in einer mehr oder minder allgemein gehaltenen Sprache die Programmanweisungen beschrieben. Wie detailliert man die Anweisungen dann aufschlüsselt, ist einem selbst überlassen. Der Pseudocode für das Listing 2.1 könnt so aussehen:

- Erwarte vom Benutzer eine Zahl a.
- Erwarte vom Benutzer eine Zahl b.
- Wenn a größer ist als b ...
  - ... dann:
    - Merke maximale Zahl ist a.
  - ... sonst:

- Merke maximale Zahl ist b.

- Gebe die maximale Zahl aus.

Oft wird nicht das ganze Programm so geplant, sondern nur die Grobstruktur und knifflige Detailprobleme werden so vorher durchgeplant. Ihr könnt euch sicherlich vorstellen, dass ein Struktogramm für ein Programm mit einer Million Programmzeilen nicht praktikabel ist.

Im Prinzip sollte die Entwicklung kleiner Programme so ablaufen:

1. Verstehen des Problems
2. Erarbeiten der Lösung
3. Zerlegen der Lösung in Einzelanweisungen (Algorithmus)
4. Anpassen des Algorithmus an die Programmiersprache
5. Schreiben des Codes
6. Testen und Fehlerbeseitigung
7. Evt. Optimierung
8. Freigabe des Programms

Am Schluss noch ein Beispiel für ein einzeiliges Programm.

Listing 2.3. entscheidung_kopf_oder_zahl.html

```
1: <script type="text/javascript">
2: // Die Virtuelle Münze ------------------------------------------------------ //
3: alert((Math.floor(Math.random()*2)>0) ? "Kopf" : "Zahl");
4: </script>
```

A 2.1. Erläutere die Codezeile 3 aus dem obigen Listing und die Aufgabe des Skripts.

2.2.1 Anwendung: Caesar-Verschlüsselung

Eine einfache Art der Verschlüsselung ist die nach Julius Caesar benannte Caesar-Verschlüsselung. Dabei werden die Buchstaben einfach im Alphabet verschoben. Bei einer Verschiebung um 1 wird z.B. aus einem A ein B, aus einem B ein C und aus einem V ein W. Das folgende Skript führt diese Verschiebung für den ersten Buchstaben des eingegebenen Textes aus.

Listing 2.4. caesar_1.html

```
 1: <script type="text/javascript">
 2: // Eingabe ---------------------------------------------------------------
 3: var txt = prompt("Text eingeben");
 4:
 5: // In Großbuchstaben umwandeln
 6: var txt = txt.toUpperCase();
 7:
 8: // Verschiebung des ersten Buchstabens um 1
 9: var charCode = txt.charCodeAt();
10: charCode += 1;
11: var codetxt = String.fromCharCode(charCode);       // Wandelt Code in Buchstaben
12:
13: // Ausgabe ----------------------------------------------------------------
14: alert(codetxt);
15: </script>
```

Das Programm liest den ersten Buchstaben der Eingabe aus, ermittelt den Zeichencode, erhöht diesen um eins und wandelt dann den Zeichencode wieder mit dem Befehl `String.fromCharCode()` in ein Zeichen um.

A 2.2. Testen Sie das Programm. In welcher Situation funktioniert das Programm nicht richtig?

Es reicht nicht aus, alle Buchstaben um einen zu erhöhen. Wenn ein Z eingegeben wird, dann muss wieder zum ersten Buchstaben A gesprungen werden.

Listing 2.5. caesar_2.html

```
 1: <script type="text/javascript">
 2: // Eingabe ------------------------------------------------------------------------
 3: var txt = prompt("Text eingeben");
 4:
 5: // In Großbuchstaben umwandeln
 6: var txt = txt.toUpperCase();
 7:
 8: // Verschiebung des ersten Buchstabens um 1
 9: var charCode = txt.charCodeAt();
10: charCode = (charCode > 89) ? 65 : charCode + 1;
11:
12: var codetxt = String.fromCharCode(charCode);
13:
14: // Ausgabe -------------------------------------------------------------------------
15: alert(codetxt);
16: </script>
```

2.2.2 Verschachtelte Entscheidung

Natürlich ist es auch möglich Entscheidungen zu verschachteln. Wenn z.B. bestimmt werden soll, ob eine Zahl negativ oder positiv ist, dann gibt es auch noch den dritten Fall, dass sie nämlich gleich Null ist. An der Stelle, wo sonst der zurückgegebene Wert steht, kann natürlich wieder der ternäre Operator stehen. Dieser muss aber komplett in Klammern gesetzt werden, wie in Listing 2.6 in Zeile 10 zu sehen.

Listing 2.6. entscheidung_plusminusnull.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------------ //
 3: var zahl = 0;
 4: var ergebnis = "";
 5:
 6: // Eingabe ------------------------------------------------------------------- //
 7: zahl = Number(prompt("Geben Sie eine Zahl ein!", ""));
 8:
 9: // Verarbeitung ------------------------------------------------------------- //
10: ergebnis = (zahl>0) ? "positiv" : ((zahl < 0) ? "negativ" : "null");
11:
12: // Ausgabe ------------------------------------------------------------------- //
13: alert("Die Zahl " + zahl + " ist " + ergebnis + "!");
14: </script>
```

2.3 Entscheidung mit Anweisungsblock

Der Entweder-Oder-Operator ist in seiner Funktionalität auf einfache Wertzuweisungen beschränkt. Für komplexere Aufgaben ist er nicht geeignet. Dafür gibt es die `if`-Entscheidung. Durch den Befehl `if` wird ein Anweisungsblock in Abhängigkeit von dem Wahrheitswert eines Ausdrucks ausgeführt. Ist der Ausdruck wahr, so wird der entsprechende Programmabschnitt abgearbeitet.

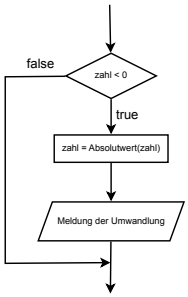

Abbildung 2.2: Struktogramm für die bedingte Ausführung im Skript entscheidung_if.html.

Schauen wir uns das doch mal an einem Programm zur Berechnung der Wurzel aus einer Zahl. Die Wurzel kann nur aus einer positiven Zahl gezogen werden. Deshalb soll das Programm nach der Eingabe überprüfen, ob die Zahl positiv ist.

Listing 2.7. entscheidung_if.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var zahl = 0;
 4: var zahlAlt = 0;
 5: var ergebnis = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: zahl = Number(prompt("Geben Sie eine positive Zahl ein!", ""));
 9:
10: // Verarbeitung ------------------------------------------------------------- //
11: if (zahl < 0) {
12:   zahlAlt = zahl;
13:   zahl = Math.abs(zahl);
14:   alert("Verbotener Wert: Ersetze " + zahlAlt + " durch " + zahl + "!");
15: }
16:
17: ergebnis = Math.sqrt(zahl);
18:
19: // Ausgabe ------------------------------------------------------------- //
20: alert("Die Wurzel aus " + zahl + " ist " + ergebnis + "!");
21: </script>
```

In Zeile 11 wird der Ausdruck `zahl < 0` auf seinen Wahrheitswert geprüft. Sollte der Ausdruck wahr sein, dann wird der folgende Anweisungsblock, der innerhalb der geschweiften Klammern `{ ... }` steht, ausgeführt. Um eine bessere Übersicht zu behalten werden die Zeilen innerhalb des Anweisungsblocks eingerückt. Die Tiefe der Einrückung ist Geschmackssache. Ich bevorzuge, da ich oft die Programmlistings ausdrucke, eine Einrückung von zwei Zeichen. Andere rücken einen ganzen Tabulatorschritt (4 oder 8 Zeichen) ein.

2.3.1 Beispiel: Seitenwechsel mit Bestätigung

Das folgende Listing ermöglicht einen Seitenwechsel zu einer vom Anwender eingegebenen Ziel-Adresse. Vor dem Wechsel wird noch einmal nachgefragt, ob die Aktion auch durchgeführt werden soll. Dazu brauchen wir zwei neue Anweisungen:

confirm() Der Funktion `confirm(`*`Text`*`)` öffnet ein Dialogfenster mit einem Text, und zwei Schaltflächen *OK* und *Abbrechen*. Der Text muß als Parameter übergeben werden. Bei Betätigung der Schaltfläche

OK liefert die Funktion den Wert `true` zurück und bei Betätigung der Schaltfläche *Abbrechen* den Wert `false`.

location.href Das `location`-Objekt steht für die URL der aktuellen HTML-Seite. Die Eigenschaft `href` beschreibt dabei die komplette URL (WWW-Adresse) der Seite. Diese Eigenschaft kann nicht nur ausgelesen werden. Eine Änderung der Variablen führt wie das Betätigen eines Links dazu, dass zu einer neuen Webseite gewechselt wird.

Listing 2.8. entscheidung_confirm.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zielURL = "";
 4:
 5: // Eingabe ---------------------------------------------------------------- //
 6: zielURL = prompt("Geben Sie die Zieladresse an:", "http://www.lernbuffet.de");
 7:
 8: // Eingabe bestätigen und dann wechseln
 9: if (confirm("Sind Sie sicher, dass Sie auf " + zielURL + " wechseln wollen?")) {
10:   location.href = zielURL;
11: }
12: </script>
```

2.3.2 Beispiel: Kontrolle der Eingabe auf Zahlen

Eine der unbeliebtesten aber notwendigen Aufgaben beim Schreiben eines Programms ist die Kontrolle der Eingabe. Die meisten Sicherheitslücken in heutigen Programmen entstehen dadurch, dass von außen kommende Daten nicht richtig überprüft wurden. In Listing 2.7 hast Du schon eine solche Überprüfung kennengelernt. Jetzt wollen wir uns mal mit der Eingabe von Zahlen beschäftigen. Bis jetzt haben wir das Objekt `Number` verwendet um aus der Eingabe eine Zahl zu machen. Allerdings bei der kleinsten Unstimmigkeit liefert `Number()` den Wert `NaN` (Not a Number - Keine Zahl) zurück. Es gibt zwei Funktionen, die ebenfalls einen Text in eine Zahl verwandeln. Die Funktion `parseInt()` wandelt einen Text in eine ganze Zahl um, während `parseFloat()` den Text in eine Fließkommazahl umwandelt.

Listing 2.9. parse.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl = 0;
 4:
 5: // Eingabe ---------------------------------------------------------------- //
 6: zahl = prompt("Geben Sie eine Zahl ein!", "");
 7:
 8: // Verarbeitung und Ausgabe ----------------------------------------------- //
 9: document.writeln(zahl + "<br>");
10: document.writeln("Number: " + Number(zahl) + "<br>");
11: document.writeln("Number: " + parseInt(zahl) + "<br>");
12: document.writeln("Number: " + parseFloat(zahl) + "<br>");
13:
14: </script>
```

A 2.3. Teste das obige Skript mit folgenden Eingaben und notiere die Ergebnisse in der Tabelle.

Eingabe	`Number()`	`parseInt()`	`parseFloat()`
5.3			
5,3			
fünf			
8tausend			
tausend8			
3.75e3			
3.75e-3			
3e3.75			

Ist es keine Zahl?

Die Funktion `isNaN()` testet ob der Parameter den Wert `NaN` (Not a Number) hat. Sie liefert `true` zurück, wenn dies der Fall ist, sonst den Wert `false`.

Das folgende Skript berechnet das Quadrat einer eingegebenen Zahl nur, wenn es sich dabei auch um eine Zahl handelt.

Listing 2.10. isnan.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl = 0;
 4:
 5: // Eingabe ----------------------------------------------------------------- //
 6: zahl = parseFloat(prompt("Geben Sie eine Zahl ein!", ""));
 7:
 8: // Verarbeitung und Ausgabe ------------------------------------------------ //
 9: if (!isNaN(zahl)) {
10:   alert("Das Quadrat von " + zahl + " ist " + (zahl * zahl) + "!");
11: }
12:
13: </script>
```

2.4 Verzweigung

Mit dem Befehl `if` wird nur definiert, ob ein Teil des Programmcodes ausgeführt wird oder nicht. Oft tritt aber der Fall auf, daß entweder der eine oder der andere Code ausgeführt werden soll. Für diesen Fall wird der `if`-Befehl um den Befehl `else` erweitert. Wenn die Aussage wahr ist, dann wird der Anweisungsblock hinter dem `if` ausgeführt. Wenn die Aussage falsch ist, kommt der Anweisungsblock hinter dem `else` zur Ausführung.

Das folgende Skript erwartet eine ganze Zahl und überprüft dann, ob die Zahl gerade – durch zwei teilbar – oder ungerade – nicht durch zwei teilbar – ist.

Listing 2.11. entscheidung_if_else.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl = 0;
 4: var ausgabe = "";
 5:
 6: // Eingabe ----------------------------------------------------------------- //
 7: zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));
 8:
 9: // Verarbeitung ------------------------------------------------------------ //
10: if (zahl % 2 == 0) {
11:   ausgabe = "Die Zahl " + zahl + " ist gerade!";
12: } else {
```

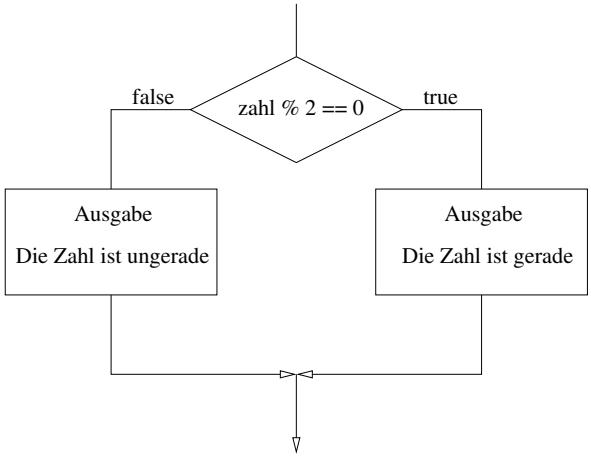

Abbildung 2.3: Die Verzweigung if-else als Flussdiagramm

```
13:   ausgabe = "Die Zahl " + zahl + " ist ungerade!";
14: }
15:
16: // Ausgabe ------------------------------------------------------------------ //
17: alert (ausgabe);
18: </script>
```

Der Befehl `else` bewirkt, daß der folgende Anweisungsblock ausgeführt wird, wenn der Ausdruck falsch gewesen ist. So wird z. B. in Listing 2.11 die Zeile 11 ausgeführt, wenn die Zahl durch zwei teilbar ist, der Ausdruck also wahr ist, und wenn der Ausdruck falsch ist, dann kommt Zeile 13 zum Zuge.

A 2.4. Erweitere Listing 2.10 so, dass bei einer Falscheingabe eine Fehlermeldung erscheint.

Beispiel: Passworteingabe

Das folgende Listing verzweigt auf eine "geheime" Seite erst dann, wenn das richtige Passwort eingegeben wurde.

Listing 2.12. entscheidung_passwort.html

```
 1: <script type="text/javascript">
 2: // Parameter ------------------------------------------------------------- //
 3: var passwort = "Athenaeum";
 4: var urlGeheim = "http://www.lernbuffet.de";
 5: var urlOffen = "http://www.athenaeum-stade.de";
 6:
 7: // Variablen ---------------------------------------------------------------- //
 8: var eingabe = "";
 9:
10: // Eingabe ------------------------------------------------------------------ //
11: alert("Project GRAY PLACE: For your eyes only!");
12: eingabe = prompt("Type the password!", "");
13:
14: // Verarbeitung und Ausgabe ------------------------------------------------- //
15: if (eingabe == passwort) {
16:   alert("Password accepted! Access allowed!");
17:   location.href = urlGeheim;
18: } else {
19:   alert("Wrong Password! Access denied!");
20:   location.href = urlOffen;
21: }
```

```
22: </script>
```

Dieser Passwortschutz ist allerdings (sehr, sehr, sehr) unsicher und kann leicht ausgehebelt werden. Im Prinzip hilft er nur gegen Leute, die keine Ahnung vom World Wide Web besitzen.

2.4.1 Mehrfache Verzweigung mit `if` und `else`

Vielleicht ist Dir aufgefallen, daß bei der Eingabe von Worten das Listing 2.11 die Antwort gibt: "Die Zahl NaN ist ungerade!"

Sinnvoll wäre es, diesen Fall auch auszuschließen. Dazu verwenden wir die bereits bekannte Funktion `isNaN()`.

Wir erweitern jetzt also das Listing 2.11 um eine weitere Entscheidung.

Listing 2.13. entscheidung_if_else_2.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl = 0;
 4: var ausgabe = "";
 5:
 6: // Eingabe ----------------------------------------------------------------- //
 7: zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));
 8:
 9: // Verarbeitung ------------------------------------------------------------ //
10: if (isNaN(zahl)) {
11:   ausgabe = "Sie sollten doch eine ganze Zahl eingeben!";
12: } else if (zahl % 2 == 0) {
13:   ausgabe = "Die Zahl " + zahl + " ist gerade!";
14: } else {
15:   ausgabe = "Die Zahl " + zahl + " ist ungerade!";
16: }
17:
18: // Ausgabe ----------------------------------------------------------------- //
19: alert (ausgabe);
20: </script>
```

Die Entscheidung in Zeile 10 bis Zeile Zeile 16 wird auch als `if ... else if ... else`- Entscheidung bezeichnet.

Dadurch ist es möglich eine ganze Reihe von Befehlsblöcken in Abhängigkeit von vielen Kriterien ausführen zu können. Sobald eine Kontrolle zu einer wahren Aussage führt, wird der entsprechende Block ausgeführt. Die folgenden Kontrollen werden dann nicht mehr abgearbeitet.

2.4.2 Beispiel: Ist das eine ganze Zahl?

Diese Frage stellt sich, wenn ein Programm eine ganze Zahl als Eingabe benötigt. Die Überprüfung ist dabei ganz einfach. Wenn die Zahl gerundet wird und dabei der gleiche Wert herauskommt, dann ist die Zahl eine ganze Zahl. Also braucht das Programm nur den Rundungswert der Eingabe mit der Eingabe vergleichen, wie es z.B. in dem folgenden Listing realisiert ist.

Listing 2.14. entscheidung_ganzezahl.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------ //
3: var zahl = 0;
4:
5: // Eingabe ----------------------------------------------------------------- //
6: zahl = Number(prompt("Geben Sie eine ganze Zahl ein!", ""));
7:
```

```
 8: // Verarbeitung und Ausgabe ------------------------------------------------ //
 9: if (zahl == Math.floor(zahl)) {
10:   alert("Richtig! " + zahl + " ist eine ganze Zahl!");
11: } else {
12:   alert("Falsch! " + zahl + " ist keine ganze Zahl!");
13: }
14: </script>
```

A 2.5. Realisiere das Script 2.6 mit den Befehlen `if` und `else`.

2.4.3 Beispiel: Das 1x1-Training

Manche Sachen muss man einfach stur pauken. Z.B. das kleine Ein-Mal-Eins. Das folgende Programm hilft dabei.

Listing 2.15. trainer_1x1.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var a = 0;
 4: var b = 0;
 5: var ergebnis = 0;
 6:
 7: // Programm ------------------------------------------------------------- //
 8: // Zufallszahlen generieren
 9: a = Math.floor(Math.random()*10)+1;
10: b = Math.floor(Math.random()*10)+1;
11:
12: // Benutzereingabe
13: ergebnis = Number(prompt(a + " * " + b + " = ",""));
14:
15: // Ergebniskontrolle
16: if (ergebnis == a*b) {
17:   alert("Richtig! " + a + " * " + b + " = " + ergebnis);
18: } else {
19:   alert("Falsch! " + a + " * " + b + " = " + (a*b));
20: }
21: </script>
```

Übrigens kannst Du beim Firefox-Browser durch das Drücken der Taste `F5` eine Seite neu laden. Damit wird dann auch wieder das Skript gestartet und eine neue Aufgabe wird gestellt.

A 2.6. Verändere das Skript so, dass es das große 1x1 (bis 20) trainiert.

2.5 Mehrfachverzweigung

Mehrfache Verzweigungen mit `if` und `else` können sehr aufwendig sein. Wenn bei der Verzweigung nur auf Gleichheit getestet wird, dann kann die `switch-case-default`-Struktur verwendet werden.

Listing 2.16. entscheidung_switch.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var eingabe = "";
 4:
 5: // Eingabe ------------------------------------------------------------- //
 6: eingabe = prompt("Geben Sie ein Tier ein!", "Hund");
 7:
```

```
 8: // Verarbeitung und Ausgabe ------------------------------------------------ //
 9: switch (eingabe) {
10:   case "Hund":
11:     alert("Wuff Wuff!");
12:     break;
13:   case "Katze":
14:   case "Kater":
15:     alert("Miau Miau!");
16:     break;
17:   case "Ente":
18:     alert("Quak Quak!");
19:     break;
20:   case "Frosch":
21:     alert("Quaaark Quaaark!");
22:     break;
23:   default:
24:     alert("Dieses Tier kenne ich nicht!");
25:     break;
26: }
27: </script>
```

Die Anweisung `switch` leitet einen Anweisungsblock ein. In diesem Anweisungsblock existieren Sprungmarken durch den Befehl `case`. Passt der bei `switch` eingetragene Wert mit dem Wert einer Sprungmarke überein, dann setzt der Interpreter an genau dieser Stelle das Script fort. Sollte keiner der `case`-Werte stimmen, dann werden die Anweisungen hinter der `default`-Marke ausgeführt.

A 2.7. Entferne alle `break`-Anweisungen aus dem Script 2.16 und teste dann das Programm. Erläutere, was die Anweisung `break` bewirkt.

A 2.8. Erweitere das Script 2.16 um weitere Tiere und ihre Laute.

2.6 Objekt `location`

Der Ort einer Webseite oder auch eines beliebigen anderen Dokuments oder Ressource wird im Internet durch die URI (*Uniform resource identifier*) angegeben. Das Objekt `location` erlaubt es diese Information der Webseite auszulesen oder sogar zu verändern, wie es in Listing 2.8 schon gezeigt wurde.

```
http://de.selfhtml.org:80/javascript/objekte/location.htm?query=athenaeum#href
\___/  \_____________/\_/\_____________________________/\______________/\___/
  |        |           |          |                            |           |
protocol  hostname     port       pathname                   search       hash
       \_____________/
          |
          host
\_____________________________________________________________________________/
                          |
                          href
```

Hier ein Beispiel für ein Programm, das die Informationen über die URI ausgibt. Interessant werden die Angaben dann, wenn die Datei tatsächlich auf einem Webserver liegt.

Listing 2.17. location.html

```
1: <pre>
2: <script type="text/javascript">
3: // Variablen ------------------------------------------------------------- //
4: document.writeln("href    : " + location.href);
5: document.writeln("protocol: " + location.protocol);
6: document.writeln("host    : " + location.host);
```

```
 7: document.writeln("hostname: " + location.hostname);
 8: document.writeln("port    : " + location.port);
 9: document.writeln("pathname: " + location.pathname);
10: document.writeln("hash    : " + location.hash);
11: document.writeln("search  : " + location.search);
12: </script>
13: </pre>
```

Abbildung 2.4 gibt die Ausgabe des Listings für eine URI aus, über die die Gruppenhomepage der AG Programmieren zu erreichen ist.

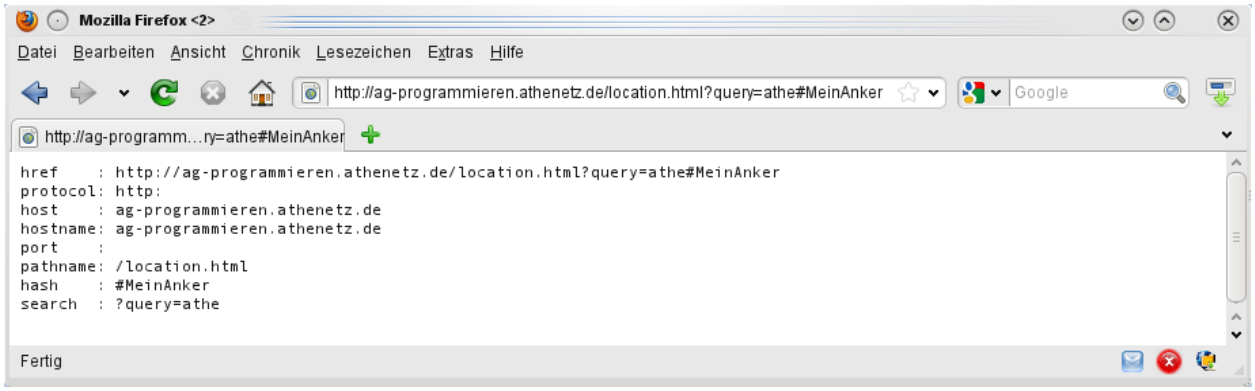

Abbildung 2.4: Informationen über die URI gibt das Objekt `location`.

Kapitel 3

Wiederholungen und Schleifen

Manche Problemlösungen erfordern es, dass ein Vorgang wiederholt wird. Sei es, dass eine Aktion so lange wiederholt wird, bis das gewünschte Ergebnis erzielt wurde, oder dass ein Vorgang z.B. genau elfmal wiederholt werden soll. Die folgenden Formulierungen deuten auf einen solchen Vorgang hin.

- Untersuche alle Zeichen einer Zeichenkette.
- Wiederhole die Eingabe so lange, bis eine ganze Zahl eingegeben wurde.
- Solange die Zahl nicht geraten wurde, wiederhole die Fragerunde.
- Solange der Ball nicht die Kante berührt hat, wiederhole die Verschiebung um einen Punkt nach rechts.
- Zähle von 1 bis 10.
- Zähle von 10 bis 0.
- Zeichne 10 Striche.

Die Lösung von Aufgabe 1.17 ist mit einer Schleife deutlich einfacher, als mit dem Mitteln, die aus Kapitel 1 bekannt sind.

Die Strukturen, die Wiederholungen in Programmen ermöglichen, heißen Schleifen. JavaScript kennt mehrere Varianten von Schleifen.

3.1 Die while-Schleife

Die einfachste Schleife, die JavaScript zur Verfügung stellt, ist `while`. In dieser Schleife wird eine Anweisung wiederholt, so lange der angegebene Prüfausdruck wahr ist. Wird der Ausdruck falsch, dann wird die Schleife beendet.

Das folgende Beispiel addiert so lange aufeinanderfolgende Zahlen auf, bis ein eingegebener Grenzwert erreicht wird.

Listing 3.1. while_summe.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var endsumme = 0;
 4: var summe = 0;
 5: var zahl = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: endsumme = parseInt(prompt("Bis zu welcher Zahl soll addiert werden?",""));
 9:
10: // Verarbeitung
```

```
11: while (summe < endsumme) {
12:   zahl++;
13:   summe += zahl;
14:   document.writeln("... plus " + zahl + " ist " + summe + "<br>");
15: }
16:
17: // Ausgabe
18: alert ("Addiere 1 bis " +zahl+ " um " +endsumme+ " oder mehr zu erreichen.");
19:
20: </script>
```

Eine vorher genau definierte Anzahl von Durchläufen kann auch mit der while-Schleife erreicht werden. Dazu wird nur eine Schleifenvariable zum Zählen benötigt, die im Schleifenrumpf verändert wird. Im folgenden Beispiel führen wir eine Caesar-Verschlüsselung mit einer Verschiebung um 13 Zeichen durch. Diese wird auch als Rot-13-Verschlüsselung bezeichnet.

Listing 3.2. rot13.html

```
 1: <script type="text/javascript">
 2: // Eingabe ------------------------------------------------------------------
 3: var txt = prompt("Text eingeben");
 4:
 5: // In Großbuchstaben umwandeln
 6: txt = txt.toUpperCase();
 7:
 8: // Alle Buchstaben bearbeiten
 9: var strout = "";                                        // String für die Ausgabe
10: var i = 0;                                                 // Schleifenvariable
11:
12: // Arbeite alle Zeichen der Zeichenkette ab
13: while (i < txt.length) {
14:   // Verschiebung des Buchstabens um 13
15:   var charCode = txt.charCodeAt(i);
16:   charCode += 13;
17:   // Über "Z", dann 26 abziehen
18:   if (charCode > 90) {
19:     charCode -= 26;
20:   }
21:   // Zur Ausgabezeichenkette hinzufügen
22:   var strout = strout + String.fromCharCode(charCode);
23:   // Zum nächsten Zeichen gehen
24:   i++;
25: }
26:
27: // Ausgabe ------------------------------------------------------------------
28: alert(strout);
29: </script>
```

A 3.1. Gebe einen Text ein und lasse Dir das Ergebnis anzeigen. Kopiere das Ergebnis und gebe dieses wiederum als Eingabe ein. Beschreibe das Ergebnis und begründe es.

Wir können natürlich auch rückwärts zählen.

Listing 3.3. while_mathemeister.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------ //
3: var mathemeister = 0;
4:
5: // Schleife
```

```
 6: mathemeister = 10;
 7: while (mathemeister > 0) {
 8:   alert(mathemeister + " kleine Mathemeister!");
 9:   mathemeister--;
10: }
11:
12: alert ("... und jetzt sind alle weg.!");
13:
14: </script>
```

Die Zählvariable `mathemeister` wird in Zeile 6 auf den Wert 10 gesetzt. Am Beginn der Schleife (Zeile 7) wird geprüft, ob die Variable `mathemeister` einen Wert größer als `0` besitzt. Solange das der Fall ist, wird die Anweisung in der Schleife wiederholt. Innerhalb des Schleifenblocks muß nun dafür gesorgt werden, daß der Prüfausdruck irgendwann falsch wird. Ist dies nicht der Fall, wird die Schleife nie beendet. Man redet dann von einer Endlosschleife.

A 3.2. Entwerfe ein Programm, dass eine ganze Zahl von 1 bis 10 per Zufall bestimmt. Danach soll der Benutzer so lange nach dieser Zahl gefragt werden, bis er die richtige Zahl eingibt.

Um die Zeichencodierung der Groß- und Kleinbuchstaben sowie des Leerzeichens zu ermitteln, hat Herr Phisigma begonnen ein Programm zu schreiben, dass effektiver sein soll, als das ursprünglich in Aufgabe 1.17 geschriebene.

Listing 3.4. string_char_3.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------
3: var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz ";
4:
5: document.writeln("<pre>");
6:
...
```

A 3.3. Schreibe das Programm unter Verwendung der while-Schleife zu Ende. Verwende auch die Stringeigenschaften bzw. Methoden `.length`, `.charAt()` und `.charCodeAt()`.

3.2 Die do-while-Schleife

In der `while`-Schleife wird immer am Anfang der Schleife geprüft. Es kann daher sein, daß die Anweisung in der Schleife nie ausgeführt wird, wenn der Ausdruck von Anfang an falsch ist. Manchmal soll die Schleife aber mindestens einmal durchlaufen werden. So z. B. bei Eingaben, die bei falschen Werten wiederholt werden sollen. Für diesen Fall gibt es die `do-while`-Schleife. Hier erfolgt die Prüfung am Fuß der Schleife.

Listing 3.5. do_eingabe.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var antwort = false;
 4:
 5: // Eingabe ---------------------------------------------------------------- //
 6: do {
 7:   antwort = confirm("Sollen wir Dein Taschengeld um 10 Euro reduzieren?");
 8: } while (!antwort);
 9:
10: // Ausgabe ---------------------------------------------------------------- //
11: alert ("Dein Taschengeld wurde um 10 Euro reduziert!");
12:
13: </script>
```

Zur Erinnerung: Das Dialogfenster liefert die Werte `true` *oder* `false` *zurück. Das Ausrufungszeichen vor einem Wahrheitswert kehrt die Bedeutung des Wahrheitswerts um.*

Dann lasst euch mal überraschen, was das Programm macht.

Das Programm geht auch noch kürzer, denn innerhalb des Prüfausdrucks können auch Befehle aufgerufen werden.

Listing 3.6. while_eingabe.html

```
1: <script type="text/javascript">
2: // Eingabe ------------------------------------------------------------- //
3: while (!confirm("Sollen wir Dein Taschengeld um 10 Euro reduzieren?"));
4:
5: // Ausgabe ------------------------------------------------------------- //
6: alert ("Dein Taschengeld wurde um 10 Euro reduziert!");
7:
8: </script>
```

A 3.4. Verändern Sie Listing 2.11 so, dass die Eingabe so lange wiederholt wird, bis tatsächlich eine Zahl eingegeben wurde.

3.3 Die Zählschleife

Wie Du schon in Listing 3.3 sehen konnten, benötigt eine Schleife zum Zählen drei wichtige Eigenschaften.

1. Der Anfangswert der Zählvariablen muß gesetzt werden. (Zeile 6)
2. Die Laufbedingung muß geprüft werden. (Zeile 7)
3. Die Laufvariable muß hoch- oder runtergezählt werden. (Zeile 9)

Da Schleifen mit einer vorher bekannten Anzahl von Wiederholung öfter vorkommen, hat man eine Schleife entwickelt, die diese drei Eigenschaften in ihrem Kopf zusammenfasst. Das ist die `for`-Schleife.

```
for (Anfangswert; Laufbedingung; Veränderung) { }
```

3.3.1 Summe der Zahlen von 1 bis 100

Eine kleine Geschichte: Früher auf den Dörfern wurden die Kinder in Dorfschulen unterrichtet. Diese besaßen meistens nur einen Raum, in dem die Schüler aller Altersstufen unterrichtet wurden. Es begab sich aber nun in einer solchen Dorfschule um 1787, daß der Lehrer sich seinen älteren Schülern widmen wollte. Um die jüngeren Schüler zu beschäftigen, stellte der Lehrer ihnen folgende Aufgabe: Wie groß ist die Summe aller Zahlen zwischen 1 und 100?

Zu seinem Erstaunen kam schon nach 10 Minuten ein Schüler zu ihm und sagte ihm die Lösung. Dieser Schüler war Carl Friedrich Gauß, der später als Mathematiker berühmt wurde.

Wir sind keine solchen Genies, aber dafür haben wir Rechner, die uns das mühselige Aufsummieren abnehmen.

Listing 3.7. for_gauss.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------ //
3: var zahl = 0;
4: var summe = 0;
5:
6: // Eingabe ------------------------------------------------------------- //
7: zahl = parseInt(prompt("Summiere die Zahlen von 1 bis ... auf!",100));
8:
9: // Verarbeitung ---------------------------------------------------------- //
```

```
10: for (var i = 1; i <= zahl; i++) {
11:   summe += i;
12: }
13:
14: // Ausgabe ------------------------------------------------------------------ //
15: alert ("Die Summe von 1 bis " + zahl + " ist " + summe + "!");
16:
17: </script>
```

Vielleicht hast Du es schon bemerkt. Dieses Programm macht im Prinzip das Gleiche wie das Programm aus Listing 3.1. Nur wird dort die Summe vorgegeben und hier bis zu welcher Zahl summiert werden soll.

3.3.2 Beispiel: Der Ein-Mal-Eins-Trainer

Einen ersten Ein-Mal-Eins-Trainer gab es schon in Listing 2.15. Dieser hatte aber den Nachteil, dass man für jede Aufgabe auf das Neu-Laden-Symbol klicken musste. Besser wäre es, wenn 10 Fragen hintereinander gestellt werden würden. Solche Programmparameter solltest Du nicht direkt im Programm als Literale einsetzen, sondern sie am Anfang des Programms definieren. Dann sind spätere Änderungen leicht zu erledigen und Du musst nicht das ganze Programm nach den zu verändernden Werten durchsuchen.

Listing 3.8. for_trainer_1x1.html

```
 1: <script type="text/javascript">
 2: // Parameter ------------------------------------------------------------- //
 3: var runden = 10;       // Anzahl der Raterunden
 4: var obereGrenze = 10;  // Zufallszahlen von 1 bis ...
 5:
 6: // Variablen ---------------------------------------------------------------- //
 7: var a = 0;
 8: var b = 0;
 9: var ergebnis = 0;
10: var richtige = 0;
11:
12: // Programm ----------------------------------------------------------------- //
13: for (var i=1; i <= runden; i++) {
14:   // Zufallszahlen generieren
15:   a = Math.floor(Math.random()*obereGrenze)+1;
16:   b = Math.floor(Math.random()*obereGrenze)+1;
17:
18:   // Benutzereingabe
19:   ergebnis = Number(prompt(i + ". Aufgabe: " + a + " * " + b + " = ",""));
20:
21:   // Ergebniskontrolle
22:   if (ergebnis == a*b) {
23:     richtige++;
24:     alert("Richtig! " + a + " * " + b + " = " + ergebnis);
25:   } else {
26:     alert("Falsch! " + a + " * " + b + " = " + (a*b));
27:   }
28: }
29:
30: // Gesamtergebnis ------------------------------------------------------------ //
31: alert(richtige + " Antworten von " + runden + " waren richtig!");
32: </script>
```

Übrigens ist das Programm durch den Einsatz der Dialogfenster sehr hartnäckig und gibt den Browser erst frei, wenn alle Fragen durchgeraten worden sind.

A 3.5. Verändere das obige Programm so, dass die Anzahl der richtigen Ergebnisse am Schluss ausgegeben wird.

A 3.6. Greifen wir doch noch mal Aufgabe 3.3 auf. Löse die Aufgabe unter Verwendung der for-Schleife.

3.4 Spiel: 17 und 4

Mit den Schleifen sind wir in der Lage komplexere Programme zu schreiben, wie z.B. ein einfaches Kartenspiel. Im folgenden Beispiel soll das Kartenspiel *17 und 4* simuliert werden. Hier die vereinfachten Regeln.

- Der Spieler spielt gegen die Bank.
- Ziel des Spielers und der Bank ist es, möglichst nah an 21 Punkte zu kommen, diese Zahl aber nicht zu überschreiten.
- Gezogen werden können die Karten mit den Punktwerten von 1 bis 11.
- Der Spieler beginnt.
- Es werden für den Spieler zwei Karten gezogen.
- Der Spieler kann so viele Karten ziehen, wie er will.
- Hat der Spieler mehr als 21 Punkte, dann verliert er.
- Wenn der Spieler keine Karte mehr will, dann ist die Bank dran.
- Es werden für die Bank zwei Karten gezogen.
- Die Bank muss ziehen bei 16 oder weniger Punkten. Bei mehr als 16 Punkten darf sie nicht mehr ziehen.
- Hat die Bank nicht mehr als 21 Punkte und mehr Punkte als der Spieler, dann gewinnt die Bank.
- Haben beide Spieler die gleiche Punktzahl (Punkte ≤ 21), dann geht das Spiel unentschieden aus.
- Sonst gewinnt der Spieler.

Und hier das Programm nach diesen Regeln mit `for`- und `while`-Schleifen.

Listing 3.9. while_17und4.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------------ //
var gezogen = 0;    // Aktuell gezogene Karte
var spieler1 = 0;   // Punkte fuer Spieler 1
var spieler2 = 0;   // Punkte fuer die Bank
var weiter = true;  // Soll noch weiter gespielt werden?

// Start ---------------------------------------------------------------------- //
// Spieler 1
weiter = true;
// Zwei Karten muessen immer gezogen werden.
for (var i = 0; i<2; i++) {
  gezogen = Math.floor(Math.random()*11)+1;
  spieler1 += gezogen;
  alert("Sie haben " + gezogen + " gezogen: " + spieler1 + " Punkte.");
}

while ((spieler1 <= 21) && weiter) {
  // Frage nach der neuen Karte
  weiter = confirm(spieler1 + " Punkte: Wollen Sie noch eine Karte ziehen?");
  // Wenn ja, geht es weiter
  if (weiter) {
    gezogen = Math.floor(Math.random()*11)+1;
    spieler1 += gezogen;
    alert("Sie haben " + gezogen + " gezogen: " + spieler1 + " Punkte.");
  }
}

// Hat Spieler 1 mehr als 21 Punkte, dann hat er verloren und die Bank braucht
// nicht mehr zu spielen.
if (spieler1 <= 21) {
  // Auch die Bank muss mindestens zwei Karten ziehen
```

```
33:   for (var i = 0; i<2; i++) {
34:     gezogen = Math.floor(Math.random()*11)+1;
35:     spieler2 += gezogen;
36:     alert("Die Bank hat " + gezogen + " gezogen: " + spieler2 + " Punkte.");
37:   }
38:
39:   // Die Bank zieht nur bei 16 oder weniger Punkten.
40:   while (spieler2 <= 16) {
41:     gezogen = Math.floor(Math.random()*11)+1;
42:     spieler2 += gezogen;
43:     alert("Die Bank hat " + gezogen + " gezogen: " + spieler2 + " Punkte.");
44:   }
45: }
46:
47: // Auswertung ------------------------------------------------------------- //
48: if ( (spieler1 > 21) || ( (spieler2 <= 21) && (spieler2 > spieler1) ) ) {
49:   alert ("Die Bank hat gewonnen!");
50: } else if (spieler1 == spieler2) {
51:   alert ("Sie haben unentschieden gespielt!");
52: } else {
53:   alert ("Gratulation. Sie haben gewonnen.");
54: }
55: </script>
```

3.5 Elementschleife

Mit einer weiteren Form der for-Schleife können alle Elemente eines Objekts abgearbeitet werden. Dies kann z.B. genutzt werden, um eine Zeichenkette Zeichen für Zeichen zu bearbeiten.

Listing 3.10. for_in_text.html

```
 1: <script type="text/javascript">
 2: var text = "";
 3: var x = 0;
 4:
 5: text = prompt("Gebe einen Text ein!");
 6:
 7: for (var s in text) {
 8:   x++;
 9:   document.write(s + " " + text[s] + "<br>\n");
10: }
11:
12: alert("Dein Text hat " + x + " Zeichen.");
13: </script>
```

Es ist auch möglich, sich die einzelnen Elemente der eingebauten Objekte anzeigen zu lassen. Das folgende Listing zeigt dies am Beispiel des Objekts `screen`.

Listing 3.11. for_in_screen.html

```
1: <script type="text/javascript">
2: var x = 0;
3:
4: for (var e in screen) {
5:   x++;
6:   document.write("screen." + e + " = " + eval("screen."+e) + "<br>\n");
7: }
8:
```

```
 9: alert("Das Objekt screen hat " + x + " Elemente.");
10: </script>
```

A 3.7. Schreibe ein Programm, dass die Eigenschaften des Objekts `navigator` anzeigt.

3.6 Algorithmus: Primzahlen

Primzahlen sind eine besondere Art von Zahlen. Sie besitzen keine Teiler außer der Zahl 1 und sich selber. Primzahlen spielen eine große Rolle im Bereich der Kryptographie und in der Bruchrechnung. Es stellt sich nun die Frage, wie man eine Primzahl bestimmt.

3.6.1 Der erste Ansatz

Am Anfang wird angenommen, daß die eingegebene Zahl eine Primzahl ist. (`primzahl = true;`) Dann werden alle Zahlen zwischen 2 und der Zahl-1 ausprobiert ob sie Teiler sind. Wenn ja, wird die Variable auf `false` gesetzt.

Listing 3.12. primzahl.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var ergebnis = "";
 4: var primzahl = true;
 5: var zahl = 0;
 6:
 7: // Eingabe ---------------------------------------------------------------- //
 8: zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));
 9:
10: // Verarbeitung ----------------------------------------------------------- //
11: for (var i = 2; i < zahl; i++) {
12:   if (zahl % i == 0) primzahl = false;
13: }
14:
15: // Ausgabe ----------------------------------------------------------------- //
16: ergebnis = (primzahl) ? "eine" : "keine";
17: alert (zahl + " ist " + ergebnis + " Primzahl!");
18:
19: </script>
```

3.6.2 Abbruch einer Schleife: `break`

Die Schwächen beim vorherigen Programm zeigen sich recht schnell. Wenn Du z. B. die Zahl 10 000 000 eingibst, die wir sofort als keine Primzahl identifizieren können, braucht der Rechner doch sehr lange um die Zahl zu bearbeiten. Die Kunst gute Programme zu schreiben liegt u. a. darin, unnötige Arbeit dem Programm abzunehmen.

Wenn ein Teiler für die Zahl gefunden wurde, kann die Schleife abgebrochen werden. Dafür stellt JavaScript den Befehl `break` zur Verfügung. Innerhalb einer Schleife ausgeführt, sorgt er dafür, dass die Schleife sofort abgebrochen wird und die Programmausführung nach dem Schleifenende fortgeführt wird.

Listing 3.13. primzahl_break.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------ //
3: var ergebnis = "";
4: var primzahl = true;
5: var zahl = 0;
6:
```

```
 7: // Eingabe ------------------------------------------------------------- //
 8: zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));
 9:
10: // Verarbeitung ---------------------------------------------------------- //
11: for (var i = 2; i < zahl; i++) {
12:   if (zahl % i == 0) {
13:     primzahl = false;
14:     break;
15:   }
16: }
17:
18: // Ausgabe ---------------------------------------------------------------- //
19: ergebnis = (primzahl) ? "eine" : "keine";
20: alert (zahl + " ist " + ergebnis + " Primzahl!");
21:
22: </script>
```

Allerdings macht der Befehl `break` das Programm unübersichtlich wenn der Schleifenkörper umfangreicher ist. Besser ist es alle Abbruchbedingungen im Schleifenkopf zu sammeln. Im folgenden Fall soll die Schleife nur laufen, wenn die Endzahl noch nicht erreicht wurde und für die Primzahl noch kein Teiler gefunden wurde.

Listing 3.14. primzahl_abbruch.html

```
...
 10: // Verarbeitung ---------------------------------------------------------- //
 11: for (var i = 2; i < zahl && primzahl; i++) {
 12:   if (zahl % i == 0) primzahl = false;
 13: }
...
```

Probiere doch mal den Unterschied zwischen dem nicht optimierten und dem optimiertem Programm mit der Zahl 8 918 097 299 aus. Sie ist übrigens keine Primzahl sondern das Produkt aus den Primzahlen 89 189 und 99 991.

3.6.3 Weitere Optimierung

Wenn wir die Zahl 10 000 019 überprüfen, dann braucht der Rechner schon etwas länger. Dabei ist es gar nicht sinnvoll alle Zahlen von 2 bis 10 000 018 zu testen. Wenn eine Zahl ganzzahlig teilbar ist, gibt es zwei ganze Zahlen n und m, deren Produkt die Zahl ergibt. So ergibt 3 mal 4 die Zahl 12. Testen wir doch mal die 13 und teilen sie.

$$13/2 = 6\frac{1}{2}$$
$$13/3 = 4\frac{1}{3}$$
$$13/4 = 3\frac{1}{4}$$

An dieser Stelle könnten wir bereits abbrechen, da jetzt der Produktpartner ($3\frac{1}{4}$) unseres Teilers (4) kleiner als der Teiler geworden ist. Damit gibt es keine ganzahligen Teiler für die 13 als 1 und 13. Das Programm braucht also nur alle Zahlen bis zur Wurzel der zu untersuchenden Zahl analysieren.

Listing 3.15. primzahl_wurzel.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------- //
 3: var ergebnis = "";
 4: var primzahl = true;
 5: var zahl = 0;
```

```
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));
 9:
10: // Verarbeitung --------------------------------------------------------- //
11: for (var i = 2; i <= Math.sqrt(zahl) && primzahl; i++) {
12:   if (zahl % i == 0) primzahl = false;
13: }
14:
15: // Ausgabe -------------------------------------------------------------- //
16: ergebnis = (primzahl) ? "eine" : "keine";
17: alert (zahl + " ist " + ergebnis + " Primzahl!");
18:
19: </script>
```

Der Vorteil zeigt sich schnell bei einer Zahl wie 10 000 019. Das Programm muss jetzt nicht über 10 Millionen Tests durchführen, sondern kann sich mit etwas mehr als 1000 Tests begnügen.

3.6.4 Liste der Primzahlen

Um eine Liste der Primzahlen zu erstellen, müsste man per Hand alle Zahlen durchtesten. Aber die zu überprüfenden Zahlen können auch durch eine Schleife geliefert werden. Im unteren Beispiel sind zwei Schleifen ineinander verschachtelt. Die äußere Schleife mit der Variablen `j` liefert die zu untersuchende Zahl, während die innere Schleife mit der Variablen `i` den Primzahltest durchführt.

Listing 3.16. primzahlen.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var ergebnis = "";
 4: var primzahl = true;
 5: var zahl = 0;
 6:
 7: // Eingabe ------------------------------------------------------------- //
 8: zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));
 9:
10: // Verarbeitung --------------------------------------------------------- //
11: for (var j = 3; j <= zahl; j++) {
12:   primzahl = true;
13:   for (var i = 2; i < j; i++) {
14:     if (j % i == 0) primzahl = false;
15:   }
16:   // Ausgabe, wenn Primzahl
17:   if (primzahl) document.writeln(j);
18: }
19: </script>
```

Teste es mit der Zahl 30 000 mal aus. Das Programm wird schon etwas Zeit brauchen um die Liste zu schreiben. Wahrscheinlich wird der Browser Dich sogar mehrfach darauf aufmerksam machen, dass das Skript sehr viel Zeit braucht und wird Deine Zustimmung einfordern, das Skript weiter laufen zu lassen.

A 3.8. Füge die Optimierungsschritte aus dem Programm 3.14 und dem Programm 3.15 Stück für Stück in das Listing 3.16 ein und teste jedesmal die Geschwindigkeit des Programms.

3.7 Abbruch eines Schleifendurchgangs: `continue`

Du hast bei der Primzahlberechnung schon die Anweisung `break` kennengelernt, mit der man eine komplette Schleife abbrechen kann. Manchmal ist es aber notwendig, die Ausführung eines Schleifendurchgangs zu beenden und mit dem nächsten Schleifendurchgang zu beginnen. Hier wird die Anweisung `continue` verwendet.

Als Beispiel dient uns die Berechnung der Schaltjahre. Die Aufgabe besteht darin eine Liste von Jahren anzuzeigen und dabei die Schaltjahre zu markieren. Schaltjahre sind alle Jahre, die durch vier teilbar sind. Das folgende Listing bricht in Zeile 9 den Schleifendurchgang vorzeitig ab, wenn das Jahr nicht durch vier teilbar ist.

Listing 3.17. for_continue.html

```
 1: <script type="text/javascript">
 2: var start = 1900;
 3: var stop = 2100;
 4:
 5: document.write("Schaltjahre");
 6:
 7: for (var jahr = start; jahr <= stop; jahr++) {
 8:   document.write("<br>\n" + jahr);
 9:   if ((jahr%4) != 0) continue;
10:   document.write(" ist ein Schaltjahr.");
11: }
12: </script>
```

Übrigens könnte man die Abbruchbedingung in Zeile 9 auch anders formulieren.

```
 9:   if ((jahr%4) > 0) continue;
```

oder auch ganz kurz

```
 9:   if (jahr%4) continue;
```

Die Schaltjahrregel, die in diesem Listing angewandt wurde, ist zu einfach. Julius Caesar hatte die Regel 45 v.Chr. festlegen lassen. Damit war das Sonnenjahr genau 365,25 Tage lang. Das tatsächliche Sonnenjahr ist aber 11 Minuten und 14 Sekunden kürzer. Dies führte dazu, dass etwa alle 130 Jahre sich der Frühlingsanfang um einen Tag verschob. Unter Papst Gregor XIII. wurde festgelegt, dass alle Jahre, die durch 100 teilbar und nicht durch 400 teilbar sind, keine Schaltjahre sind. Daher war 1900 kein Schaltjahr, während 2000 ein Schaltjahr war.

Das folgende Listing berücksichtigt die neuen Regeln.

Listing 3.18. for_continue_2.html

```
 1: <script type="text/javascript">
 2: var start = 1900;
 3: var stop = 2100;
 4:
 5: document.write("Schaltjahre");
 6:
 7: for (var jahr = start; jahr <= stop; jahr++) {
 8:   document.write("<br>\n" + jahr);
 9:   if ((jahr%4) != 0) continue;
10:   if ((jahr%400) == 100) continue;
11:   if ((jahr%400) == 200) continue;
12:   if ((jahr%400) == 300) continue;
13:   document.write(" ist ein Schaltjahr.");
14: }
15: </script>
```

Der Einsatz von `break` und `continue` kann Programme unübersichtlich machen und leicht zu Fehler führen. In den meisten Fällen kann der gleiche Effekt durch geeignete if-Anweisungen erreicht werden. Beim Abfangen von selten auftretenden Fehler können `break` und `continue` sinnvoll eingesetzt werden, um eine zu große Verschachtelungstiefe zu verhindern. Gerade bei `while`- und `do`-Schleifen kann bei fehlerhafter Programmierung die `continue`-Anweisung zu Endlosschleifen führen.

Im folgenden Listing ist das Problem ohne `continue` gelöst worden. Die Formulierung der Schaltjahrbedingung in einer Anweisung ist etwas unübersichtlich. Daher hier erst mal die Lösung im Pseudocode.

Wenn
  das Jahr durch vier teilbar **und** das Jahr nicht durch 100 teilbar ist
oder
  das Jahr durch 400 teilbar ist
dann
  ist es ein Schaltjahr.

Zeile 9 zeigt die Umsetzung im Listing.

Listing 3.19. for_schaltjahre.html

```
 1: <script type="text/javascript">
 2: var start = 1900;
 3: var stop = 2100;
 4:
 5: document.write("Schaltjahre");
 6:
 7: for (var jahr = start; jahr <= stop; jahr++) {
 8:   document.write("<br>\n" + jahr);
 9:   if ((((jahr%4) == 0) && !((jahr%100) == 0)) || ((jahr%400) == 0)) {
10:     document.write(" ist ein Schaltjahr.");
11:   }
12: }
13: </script>
```

3.8 Übungen

A 3.9. Informiere Dich bei Wikipedia über den Julianischen und Gregorianischen Kalender.

A 3.10. Um besser zählen zu können macht man häufig Striche, die meistens zu fünf Strichen gruppiert werden. Z.B. für 7 schreibt man `||||| ||` oder für 13 `||||| ||||| |||`.

a) Schreibe ein Skript, das eine eingegebene Ganzzahl in senkrechten Strichen (`|`) darstellt.
b) Erweitere das Skript so, dass die Striche zu 5er Gruppen zusammengefügt werden. (Leerzeichen nach jeweils 5 Strichen.) Tipp: Modulo-Operator (%) verwenden.
c) Erweitere das Skript so, dass nach 100 Strichen ein Zeilenumbruch erfolgt.

A 3.11. Finde die Fehler in dem folgenden Codefragment.

```
a: war zahl = 0;
b: zahl = parseint( prompt( "Geben Sie eine Zahl &uuml;ber 10 ein:","" );
c: if ( Zahl > 10 ) {
d:   alert( Das ist richtig );
e: } els {
f:   alert( "Das war falsch";
g: }
```

A 3.12. Für die Formel $a^2 + b^2 = c^2$ gibt es Lösungen, in denen alle Variablen a, b und c ganze Zahlen sind. Z.B. gilt $3^2 + 4^2 = 5^2$.

a) Wie berechnet man b, wenn a und c gegeben sind?
b) Wie kann ein Programm überprüfen, ob eine Zahl eine ganze Zahl ist?
c) Schreibe ein Skript, dass nach der Zahl c fragt, und dann nach einem passenden ganzzahligen Paar a b sucht.
d) Erweitere das Skript so, dass alle sinnvollen ganzzahligen Paare gefunden werden.
e) Schreibe ein Skript, dass die Werte von 1 bis 100 für c auf ganzzahlige Paare überprüft.

Kapitel 4

Prozeduren und Funktionen

Ein Programm kann man in kleine logische Einheiten zerlegen. Diese "Miniprogramme" werden auch Unterprogramme, Prozeduren oder Funktionen genannt. Bei JavaScript wird von Funktionen oder genauer von `function` gesprochen. Man könnte auch sagen, dass man mit Funktionen seinen eigenen JavaScript-Befehle schreiben kann.

4.1 Prozedurale Funktionen

Prozedurale Funktionen sind im Prinzip eigenständige Programme. Sie werden eingesetzt, wenn eine Teil des Programmcodes mehrfach verwendet wird. Oft ist es auch günstig den Code einer Teilaufgabe in eine Funktion zu stecken um das Hauptprogramm übersichtlicher zu gestalten. In JavaScript werden prozedurale Funktionen z.B. dann verwendet, wenn das Programm nicht während des Aufbaus der Seite gestartet werden soll, sondern wenn die Seite bereits fertig geladen ist. Später werden wir noch darauf eingehen, wie man solche Funktionen durch Interaktionen des Benutzers, wie z.B. das Klicken auf einen Link oder einen Button auslöst.

Beginnen wir einfach mal mit einer Funktion, die *Hallo Welt!* einmal in den Seitentext schreibt und einmal über ein Dialogfenster anzeigt.

Listing 4.1. func_hallowelt.html

```
1: <script type="text/javascript">
2: // Funktionen ------------------------------------------------------------ //
3: function halloWelt() {
4:   document.writeln("Hallo Welt!");
5:   alert("Hallo Welt!");
6: }
7:
8: halloWelt();
9: </script>
```

In Zeile 3 beginnt die Definition der Funktion mit dem Schlüsselwort `function` gefolgt von dem Namen der Funktion. Hinter dem Namen steht ein paar runder Klammer. Auf die Bedeutung der Klammern gehen wir später ein. Variablen und Funktionen sollten niemals den gleichen Namen haben, denn das kann zu Problemen führen.

Die Anweisungen der Funktionen stehen dann in einem von geschweiften Klammern gebildeten Block, wie er bereits bei Entscheidungen und Schleifen verwendet wurde. Im Anweisungsblock befinden sich zwei bereits bekannte Ausgabebefehle. In Zeile 6 wird dann der Funktionsrumpf geschlossen. Die Funktion wird über ihren Namen mit den folgenden Klammern, wie in Zeile 8 zu sehen, aufgerufen.

Ein weiteres Beispiel informiert den Benutzer über die Größe seines Bildschirms. Dazu wird das Objekt `screen` verwendet.

Listing 4.2. func_screen.html

```
1: <script type="text/javascript">
```

```
 2: function screenInfo() {
 3:   // Variablen deklarieren
 4:   var text = "";         // Der Ausgabetext
 5:
 6:   // Text erstellen
 7:   text = "Ihr Bildschirm ist ";
 8:   text += screen.width + " Pixel breit und ";
 9:   text += screen.height + " Pixel hoch.";
10:
11:   // Textfenster öffnen
12:   alert(text);
13: }
14:
15: screenInfo();
16: </script>
```

A 4.1. Informationen über den Bildschirm

a) Informieren Sie sich über weitere Variablen, die mit dem Bildschirm zusammenhängen (Objekt `screen`).
b) Erweitern Sie das Listing 4.2 so, daß auch die Farbtiefe mit angegeben wird.
c) Schreiben Sie eine Funktion, das die Anzahl der Bildschirmpunkte und den für den Bildschirm benötigten Speicher ausgibt. Der benötigte Speicher ergibt sich aus der Anzahl der Bildpunkte multipliziert mit der Farbtiefe.

4.2 Funktionen mit Rückgabewert

Eine Funktion kann auch einen Wert zurückliefern. Dies erfolgt durch den Befehl `return` gefolgt von dem zurückzugebenden Wert. Der Befehl steht normalerweise als letzte Anweisung im Funktionsrumpf.

Das folgende Beispiel simuliert den Wurf eines sechseitigen Würfels.

Listing 4.3. func_wuerfel.html

```
 1: <script type="text/javascript">
 2: // Funktionen ------------------------------------------------------------ //
 3: function werfe1w6() {
 4:   // Simulation eines sechsseitigen Würfels
 5:   // Rückgabe: Augenzahl
 6:   return Math.floor(Math.random()*6)+1;
 7: }
 8:
 9: // Hauptprogramm ---------------------------------------------------------- //
10: alert("Sie haben " + werfe1w6() + " gewürfelt.");
11: </script>
```

Die Funktion kann nun wie jeder andere Befehl, der einen Wert liefert, verwendet werden. Programmtechnisch ist es nicht immer notwendig Funktionen zu benutzen, gerade wenn es sich um relativ kurze Programmblöcke handelt. Allerdings erhöht die Verwendung von Funktionen, die sogenannte Modularisierung, die Lesbarkeit und damit auch die Wartbarkeit von Programmen. Auch können solchen Funktionen in weiteren Programmen wiederwendet werden. Im folgenden Beispiel wird die bereits vorher erstellte Funktion `werfe1w6` verwendet um den Wurf von zwei Würfeln zu simulieren.

Listing 4.4. func_werfen.html

```
 1: <script type="text/javascript">
 2: // Funktionen ------------------------------------------------------------ //
 3: function werfe1w6() {
 4:   // Simulation eines sechsseitigen Würfels
 5:   // Rückgabe: Augenzahl
```

```
 6:   return Math.floor(Math.random()*6)+1;
 7: }
 8:
 9: // ------------------------------------------------------------------------ //
10: function werfe2w6() {
11:   // Simulation eines Wurfes von zwei sechsseitigen Würfeln
12:   // Rückgabe: Augenzahl
13:   // Variablen
14:   var augenzahl = 0;
15:
16:   // Verarbeitung
17:   augenzahl = werfe1w6() + werfe1w6();
18:
19:   // Rückgabe
20:   return augenzahl;
21: }
22:
23: // Hauptprogramm ---------------------------------------------------------- //
24: alert("Sie haben " + werfe2w6() + " gewürfelt.");
25: </script>
```

A 4.2. Schreiben Sie ein Skript mit einer Funktion `screenMemory()`, das den für die Darstellung des Bildschirms benötigten Speicher als Wert zurückgibt.

4.3 Übergabe von Werten an die Funktion

Daten können beim Aufruf der Funktion mitgegeben werden. Diese Daten bezeichnet man als *Parameter*. Die Funktion hat nicht umsonst runde Klammern hinter ihrem Namen stehen. In die Klammern werden diese Parameter geschrieben, die an die Funktion übergeben werden sollen. Wieviele Werte erwartet werden, wird in der Kopfzeile der Funktionsdefinition angegeben.

```
function hund ( rasse ) { ... }
```

Die Funktion `hund` muss zusammen mit einem Parameter aufgerufen werden. Z. B. `hund( "Collie" )`. Dabei wird der übergebene Wert in der angegebenen Variable gespeichert.

4.3.1 Wahrheitsfunktionen

Manchmal muss nur die Entscheidung gefällt werden, ob ein Sachverhalt stimmt oder nicht. Eine Funktion, die diese Aufgabe übernimmt, muss je nach Ausgang des Tests die Werte `true` (Wahr) oder `false` (Falsch) zurückliefern. Eine solche Funktion ist `isNaN(zahl)`, die Du schon in Listing 2.10 auf Seite 32 kennengelernt hast. Normalerweise fangen die Funktionsnamen für solche Funktionen mit `is` an. Wenn wir den Funktionen deutsche Namen geben, dann sollte der Funktionsname mit `ist` beginnen. Im folgenden Beispiel soll der Gesamtpreis einer Anzahl von Flaschen berechnet werden. Dabei soll der Preis der Flasche und die Anzahl der Flaschen angegeben werden. Dabei ist es natürlich sinnvoll nur ganze Zahlen für die Eingabe der Flaschenzahl zuzulassen. Ob die Zahl eine ganze Zahl ist, dass überprüft die Funktion `istGanzeZahl(zahl)`.

Listing 4.5. func_istganzezahl.html

```
 1: <script type="text/javascript">
 2: // Funktionen ------------------------------------------------------------ //
 3: function istGanzeZahl(zahl) {
 4:  return (zahl == Math.floor(zahl)) ? true : false;
 5: }
 6:
 7: // Variablen ------------------------------------------------------------- //
 8: var anzahl = 0;
 9: var einzelpreis = 0;
10: var gesamtpreis = 0;
```

```
11:
12: // Eingabe ------------------------------------------------------------- //
13: einzelpreis = Number(prompt("Preis pro Flasche: [EUR]", "0.99"));
14: do {
15:  anzahl = Number(prompt("Geben Sie die Flaschenzahl an!", ""));
16: } while ( !istGanzeZahl(anzahl) );
17:
18: // Verarbeitung ---------------------------------------------------------- //
19: gesamtpreis = einzelpreis * anzahl;
20:
21: // Ausgabe ---------------------------------------------------------------- //
22: alert(anzahl + " Flaschen kosten EUR " + gesamtpreis);
23: </script>
```

4.3.2 Funktionen mit mehreren Parametern

Manchen Funktionen brauchen mehrere Parameter um richtig zu funktionieren. In diesem Fall werden die Parametervariablen durch Kommata getrennt aufgeführt wie im unteren Listing in Zeile 7 zu sehen. Aufgerufen wird die Funktion wie sonst auch, nur dass jetzt eine durch Kommata getrennte Liste von Werten in der Parameterklammer (siehe Zeile 30) angegeben werden.

Das folgende Listing ermittelt die ganzzahligen Teiler einer eingegebenen ganzen Zahl. Teste das Programm mal mit der Zahl 720 720.

Listing 4.6. func_istteilervon.html

```
 1: <script type="text/javascript">
 2: // Funktionen ------------------------------------------------------------ //
 3: function istGanzeZahl(zahl) {
 4:  return (zahl == Math.floor(zahl)) ? true : false;
 5: }
 6:
 7: function istTeilerVon(divident,divisor) {
 8:   // Variablen
 9:   var antwort = false;
10:
11:   // Verarbeitung
12:   antwort = (divident%divisor == 0) ? true : false;
13:
14:   // Rueckgabe
15:   return antwort;
16: }
17:
18: // Variablen ------------------------------------------------------------- //
19: var zahl = 0;
20:
21: // Eingabe --------------------------------------------------------------- //
22: do {
23:  zahl = Number(prompt("Geben Sie eine ganze Zahl ein!", ""));
24: } while ( !istGanzeZahl(zahl) );
25:
26: // Verarbeitung und Ausgabe ---------------------------------------------- //
27: for (var i = 2; i < zahl; i++) {
28:   if (istTeilerVon(zahl,i)) {
29:     document.writeln(i + " ist Teiler von " + zahl + "<br>");
30:   }
31: }
32: </script>
```

4.4 Globale und lokale Variablen

JavaScript bietet die Möglichkeit Variablen *global* für alle Scripte einer HTML-Seite zu definieren oder nur *lokal* für eine einzelne Funktion.

4.4.1 Globale Variablen

Globale Variablen nutzt man dann, wenn Werte von verschiedenen Funktionen genutzt werden sollen. Auch bleibt die Variable aktiv, solange die HTML-Seite im Browser dargestellt wird. Im folgenden Beispiel merkt sich die globale Variable `clicks` die Anzahl der Mausklicks auf den Schalter. Gezählt wird mit der Funktion `zaehlen()`, die auch den aktuellen Zählerstand ausgibt.

Listing 4.7. klick_mich.html

```
 1: <html>
 2: <head>
 3:   <title>Klick mich - Beispiel f&uuml;r eine globale Variable</title>
 4:   <script type="text/javascript">
 5:     // gobale Variablen
 6:     var clicks = 0; // Zähler für die Mausklicks auf den Schalter
 7:
 8:     // Zählfunktion
 9:     function zaehlen() {
10:       clicks++; // Inhalt der Zählervariable um eins erhöhen
11:       alert("Du hast schon " + clicks + " mal geklickt.");
12:     }
13:   </script>
14: </head>
15: <body>
16:   <form>
17:     <input type="button" onClick="zaehlen();" value='Klick mich'>
18:   </form>
19: </body>
20: </html>
```

4.4.2 Lokale Variablen

Lokale Variablen haben den Vorteil, dass man sie in Funktionen verwenden kann ohne mit gleichnamigen Variablen in anderen Funktionen in Konflikt zu geraten. So könnte z.B. in der Funktion `hund()` die Variable `geschlecht` verwendet werden und in der Funktion `katze()` ebenfalls, ohne dass sich die beiden Variablen stören.

Als Beispiel für die Funktionsweise von globalen und lokalen Variablen soll das folgende Listing dienen.

Listing 4.8. global_lokal.html

```
 1: <script type="text/javascript">
 2: // Globale Variablen ------------------------------------------------------ //
 3: var a = "global";
 4: var b = "global";
 5:
 6: // Funktionen ---------------------------------------------------------------- //
 7: // Verwendung der globalen Variable a
 8: function setA() {
 9:   document.writeln("<p><b>setA:</b> alt a = " + a + "</p>");
10:   a = prompt("Wert von A:","");
11:   document.writeln("<p><b>setA:</b> neu a = " + a + "</p>");
12: }
13:
```

```
14: // Verwendung der lokalen Variable b
15: function setB() {
16:   var b = "lokal";
17:   document.writeln("<p><b>setB:</b> alt b = " + b + "</p>");
18:   b = prompt("Wert von B:","");
19:   document.writeln("<p><b>setB:</b> neu b = " + b + "</p>");
20: }
21:
22: // Verwendung der globalen Variable c
23: function setC() {
24:   c = "lokal oder global ?";
25:   document.writeln("<p><b>setC:</b> alt c = " + c + "</p>");
26:   c = prompt("Wert von C:","");
27:   document.writeln("<p><b>setC:</b> neu c = " + c + "</p>");
28: }
29:
30: // Hauptprogramm ------------------------------------------------------------ //
31: document.writeln("<p><b>global:</b> a = " + a + "</p>");
32: document.writeln("<p><b>global:</b> b = " + b + "</p>");
33: // document.writeln("<p><b>global:</b> c = " + c + "</p>");
34: setA();
35: document.writeln("<p><b>global:</b> a = " + a + "</p>");
36: setB();
37: document.writeln("<p><b>global:</b> b = " + b + "</p>");
38: setC();
39: document.writeln("<p><b>global:</b> c = " + c + "</p>");
40: </script>
```

Am Anfang werden die globalen Variablen (Zeilen 3+4) `a` und `b` deklariert und mit dem Wert *global* initalisiert. Die Funktionen `setA()`, `setB()` und `setC()` (Zeilen 6-28) geben den alten Wert der Variablen aus und fragen dann nach einem neuen Wert für die Variable. Während `setA()` die globale Variable `a` verwendet, wird in `setB()` eine lokale Variable `b` deklariert, die dann weiter verwendet wird. Funktion `setC()` ist ein Beispiel für einen schlechten Programmierstil. Hier wird der vorher nicht deklarierten Variable `c` ein Wert zugewiesen. Ist sie nun eine globale oder eine lokale Variable?

In den Zeilen 30 bis 39 befindet sich das Hauptscript. Erstmal werden die globalen Variablen `a` und `b` ausgegeben. Die Zeile 33 würde eine Fehlermeldung erzeugen, da der Variablen `c` noch kein Wert zugewiesen wurde. Der Aufruf der Funktion `setA()` (Zeile 34) verändert tatsächlich den Inhalt der globalen Variablen `a`. Der Aufruf der Funktion `setB()` (Zeile 36) hat keinen Einfluß auf die globale Variable `b`. Die Variable `b` ist in der Funktion eine lokale Variable. Lokale Variablen überlagern mit ihrem Namen globale Variablen. Nach dem Ende des Funktionsaufrufs vergisst der JavaScript-Interpreter die lokale Variable `b` wieder und arbeitet mit der globalen Variablen weiter.

Was passiert nun aber beim Aufruf der Funktion `setC()`? Der Interpreter erhält den Befehl der bisher nicht existierenden Variablen `c` einen Wert zuzuordnen. Würde es sich hier um einen sehr strengen Interpreter handeln, dann würde er an dieser Stelle das Programm mit einer Fehlermeldung abbrechen. Bei manchen Programmiersprachen ist solch eine Zuweisung streng verboten. Allerdings wurde JavaScript für einen schnellen und einfachen Einsatz konzipiert. Darum versucht der Interpreter mitzudenken. Er legt nun eigenständig die Variable `c` im Speicher an und da er nicht weiß, welchen Gültigkeitsbereich sie haben soll, nimmt er einfach den größten Bereich. Die Variable `c` ist also global. Allerdings ist diese Verwendung von Variablen ein schlechter Programmierstil. Also immer die Variablen *vor* ihrer Verwendung dort deklarieren, wo sie gebraucht werden.

4.5 Externe JavaScript-Dateien

Ein Grund für den Einsatz von Funktionen ist ihre Wiederverwendbarkeit. So kann z. B. die Funktion `werfe1w6()` aus Listing 4.3 in vielen Programmen eingesetzt werden. Dazu muß sie nur in die jeweilige HTML-Datei hineinkopiert werden.

Was ist aber, wenn viele Seiten das gleiche umfangreiche Skript benötigen? Das Skript müßte in jede dieser

Seiten kopiert werden und auch jedesmal mitgeladen werden. Nicht zu schweigen von den Problemen, wenn etwas an dem Skript geändert werden soll. Deshalb wurde die Möglichkeit geschaffen den JavaScript-Code in eine externe Datei auszulagern.

Das folgende Beispiel soll das Verfahren veranschaulichen. Wir beginnen mit der JavaScript-Datei, die drei Funktionen enthält. Die Funktionen sind mit einfachen Prüfungen versehen, um Fehler im Programm besser festzustellen.

Listing 4.9. bib_random.js

```
 1: /* ----------------------------------------------------------------------------
 2: Bibliothek fuer Zufallsfunktionen
 3: Autor: Ole Vanhoefer                                          Stand: 2010-09-28
 4:
 5: bib_random_bereich(kleinster Wert, groesster Wert)
 6:   Gibt eine zufaellige Zahl mit Nachkommastellen im angegebenen Bereich aus.
 7:
 8: bib_random_werfe(Anzahl der Wuerfel, maximale Augenzahl des Wuerfels)
 9:   Gibt die Augenzahl von mehreren gleichzeitig geworfenen Wuerfeln variabler
10:   Augenzahl wieder.
11:
12: bib_random_zahl(kleinster Wert, groesster Wert)
13:   Gibt eine zufaellige Ganzzahl im angegebenen Bereich aus.
14: ---------------------------------------------------------------------------- */
15:
16: // bib_random_bereich ------------------------------------------------------ //
17: function bib_random_bereich(min,max) {
18:   // min: untere Grenze
19:   // max: obere Grenze
20:
21:   // Parameter ueberpruefen
22:   if (min > max) {
23:     alert("Fehler in bib_random_bereich(): min > max:"+min+";"+max);
24:     return false;
25:   }
26:
27:   // Rueckgabe mit Berechnung
28:   return Math.random()*(max-min) + min;
29: }
30:
31: // bib_random_werfe ------------------------------------------------------- //
32: function bib_random_werfe(anzahl,augen) {
33:   // anzahl: Anzahl der zu werfenden Wuerfel
34:   // augen : Augen der zu werfenden Wuerfel
35:   // Lokale Variablen
36:   var augenzahl = null;
37:
38:   // Parameter ueberpruefen
39:   if (anzahl < 0 || anzahl != Math.floor(anzahl)) {
40:     alert("Fehler in bib_random_werfe(): Falscher Wert anzahl : " + anzahl);
41:     return false;
42:   }
43:
44:   if (augen < 0 || augen != Math.floor(augen)) {
45:     alert("Fehler in bib_random_werfe(): Falscher Wert augen : " + augen);
46:     return false;
47:   }
48:
49:   // Berechnung
50:   for (var i = 0; i < anzahl; i++) {
```

```
51:     augenzahl += Math.floor(Math.random()*augen)+1;
52:   }
53:
54:   // Rückgabe
55:   return augenzahl;
56: }
57:
58: // bib_random_zahl ------------------------------------------------------------ //
59: function bib_random_zahl(min,max) {
60:   // min: untere Grenze
61:   // max: obere Grenze
62:
63:   // Parameter ueberpruefen
64:   if (min > max) {
65:     alert("Fehler in bib_random_zahl(): min > max: "+min+";"+max);
66:     return false;
67:   }
68:
69:   if ((min != Math.floor(min)) || (max != Math.floor(max))) {
70:     alert("Fehler in bib_random_zahl(): Integer erwartet: "+min+";"+max);
71:     return false;
72:   }
73:
74:   // Rueckgabe mit Berechnung
75:   return Math.floor(bib_random_bereich(min,max+1))
76: }
```

Die folgende HTML-Seite lädt in Zeile 2 die externe JavaScript-Datei `bib_random.js` nach. Damit stehen die Funktionen `bib_random_werfe()`, `bib_random_bereich()` und `bib_random_zahl()` in der Datei `bib_random.js` in der HTML-Seite zur Verfügung.

Listing 4.10. bib_random.html

```
1: <pre>
2: <script src="bib_random.js" type="text/javascript"></script>
3: <script type="text/javascript">
4: // Test der Funktionen in der Bibliothek bib_random.js
5: for(var i = 0; i < 10; i++) {
6:   document.writeln("bib_random_werfe(2,6): "+bib_random_werfe(2,6));
7: }
8: for(var i = 0; i < 10; i++) {
9:   document.writeln("bib_random_bereich(1,6): "+bib_random_bereich(1,6));
10: }
11: for(var i = 0; i < 10; i++) {
12:   document.writeln("bib_random_zahl(1,6): "+bib_random_zahl(1,6));
13: }
14: </script>
15: </pre>
```

4.5.1 Beispiel für externe Programmbibliotheken: OpenStreetMap

Es ist nicht immer notwendig das Rad jedesmal neu zu erfinden. Inzwischen gibt es viele fertige Programmbibliotheken, deren Funktionen und Objekte man nutzen kann. Als Beispiel sei hier die OpenLayers-Bibliothek genannt, die es ermöglicht Karten auf der Basis der OpenStreetMap in einer Webseite anzuzeigen.

Das folgende Beispiel zeigt die OpenStreetMap formatfüllend im Browserfenster an.

Listing 4.11. osm_einfache_karte.html

```
1: <!DOCTYPE html>
```

```
 2: <html>
 3: <head>
 4:  <title>Eine einfache OpenStreetMap-Karte</title>
 5:  <style>
 6:   html, body {margin:0; height:100%; padding:0; width:100%;}
 7:   #landkarte {height:100%; width:100%;}
 8:  </style>
 9:  <script type="text/javascript" src="http://openlayers.org/api/2.12/OpenLayers.js"></script>
10:  <script type="text/javascript">
11:   function init() {
12:     // Erstellen der Karte im DIV-Element osm_karte
13:     var karte = new OpenLayers.Map("landkarte");
14:     // Erstellen einer OpenStreetMap-Ebene
15:     var osm = new OpenLayers.Layer.OSM();
16:     // und hinzufuegen zur Karte.
17:     karte.addLayer(osm);
18:     // Zoom auf komplette Kartenansicht einstellen
19:     karte.zoomToMaxExtent();
20:   }
21:  </script>
22: </head>
23: <body onload="init()">
24:  <div id="landkarte"></div>
25: </body>
26: </html>
```

In Zeile 9 wird die OpenLayers-Bibliothek in der Version 2.12 direkt von der Website openlayers.org geladen. Dieses Programm funktioniert also nur, wenn auch eine Internetverbindung besteht.

Die Funktion `init()`, die ab Zeile 11 definiert wird, wird durch den Event-Handler `onload` nach dem Laden der Seite aufgerufen. In dieser Funktion wird in Zeile 13 ein OpenLayers-Karten-Objekt erstellt, dass in dem DIV-Element mit der ID *landkarte* dargestellt wird. In Zeile 15 wird eine neue Rasterebene erzeugt, die die Bilder aus dem OpenStreetMap-Projekt enthält. In Zeile 17 wird die Rasterebene der Karte hinzugefügt. Anschließend wird in Zeile 19 der Zoom auf maximale Ansicht geschaltet.

4.6 Übungen

Hier ein paar Ideen für kleine Programme.

A 4.3. Benzinverbrauch: Der Treibstoffverbrauch wird in Litern pro 100 km angegeben. Als Eingabe wird die gefahrene Strecke und die verbrauchte Treibstoffmenge benötigt. Die Formel lautet:

$$\text{Treibstoffverbrauch} = \frac{\text{Treibstoffmenge}}{\text{Strecke}} \cdot 100 \tag{4.1}$$

Entwickle die Funktion `verbrauch(strecke,menge)` zur Berechnung des Benzinverbrauchs und eine Benutzerschnittstelle.

A 4.4. Umrechnung zwischen Grad Celsius und Kelvin: Temperaturen können in verschiedenen Einheiten gemessen werden. Die Celsius-Skala und die Kelvin-Skala haben die gleiche Schrittweite. Allerdings besitzen sie einen unterschiedlichen Nullpunkt. So liegt der Nullpunkt der Kelvin-Skala bei $-273{,}15\,^{\circ}\text{C}$. Darum meint $0\,^{\circ}\text{C}$ die gleiche Temperatur wie 273,15 Kelvin.

Entwickle die Funktionen `CelsiusInKelvin(celsius)` und `KelvinInCelsius(kelvin)` zur Umrechnung zwischen den verschiedenen Einheiten.

A 4.5. Umrechnung zwischen kW und PS: PS und kW sind Leistungseinheiten, die man z. B. beim Auto findet. Die alte aber noch sehr geläufige Einheit ist die Pferdestärke (PS). Die aktuelle Einheit, die eigentlich nur noch verwendet werden sollte, ist das Kilowatt (kW). Es gilt der Zusammenhang:

$$1\,\text{PS} = 0{,}735\,498\,75\,\text{kW} \qquad 1\,\text{kW} = 1{,}359\,621\,62\,\text{PS} \tag{4.2}$$

Entwickle die Funktionen `psInKw(ps)` und `kwInPs(kw)` zur Umrechnung zwischen den verschiedenen Einheiten.

A 4.6. Umrechnung zwischen Kilojoule und Kilokalorien: kJ und kcal sind Energieeinheiten, die man z. B. bei Lebensmittel auf der Verpackung findet. Die alte aber noch sehr geläufige Einheit ist die Kilokalorie (kcal). Die aktuelle Einheit, die eigentlich nur noch verwendet werden sollte, ist das Kilojoule (kJ). Es gilt der Zusammenhang:

$$1\,\text{kcal} = 4{,}1868\,\text{kJ} \qquad 1\,\text{kJ} = 0{,}239\,\text{kcal} \tag{4.3}$$

Entwickle die Funktionen `kjInKcal(kj)` und `kcalInKj(kcal)` zur Umrechnung zwischen den verschiedenen Einheiten.

Notizen:

Kapitel 5

Lösungen

5.1 Erste Schritte

A 1.1

3-6 Deklaration und Initalisierung der Variablen `nachname`, `vorname` und `name`.

9-10 Öffnen des Eingabefensters und Übergabe der Werte an die Variablen `nachname` und `vorname`.

13 Die Inhalte der Variablen `vorname` und `nachname` werden zu einer Zeichenkette mit einem Leerzeichen getrennt zusammengefügt und in der Variablen `name` gespeichert.

16 Öffnen des Nachrichtenfensters mit einem Begrüßungstext und dem eingegeben Namen. "`\n`" sorgt für einen Zeilenumbruch.

A 1.3 Die Eingabe wird immer als Zeichenkette interpretiert und dieser Typ intern in der Variable gespeichert. Die Funktion `Number()` wandelt die Zeichenkette in eine Zahl um.

A 1.4 In Zeile 4 wird zuerst `2+3` gerechnet und das Ergebnis 5 dann mit der Zeichenkette verbunden: "2+3=5"

In Zeile 5 wird zuerst `2+3=` mit `2` verbunden und dann das Ergebnis mit `3`: "2+3=23"

A 1.5

Listing 5.1. addierezweizahlen_v2.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var zahl_1 = 0;
 4: var zahl_2 = 0;
 5: var ergebnis = 0;
 6: var ausgabe = "";
 7:
 8: // Eingabe ----------------------------------------------------------------- //
 9: zahl_1 = Number(prompt("Die erste Zahl:", ""));
10: zahl_2 = Number(prompt("Die zweite Zahl:", ""));
11:
12: // Verarbeitung ------------------------------------------------------------ //
13:
14: // Addition ---------------------------------------------------------------
15: ergebnis = zahl_1 + zahl_2;
16: ausgabe = "Die Summe aus " + zahl_1 + " und " + zahl_2 + " ist ";
17: ausgabe = ausgabe + ergebnis + "!";
18: alert(ausgabe);
19: // Subtraktion ------------------------------------------------------------
20: ergebnis = zahl_1 - zahl_2;
21: ausgabe = "Die Differenz aus " + zahl_1 + " und " + zahl_2 + " ist ";
22: ausgabe = ausgabe + ergebnis + "!";
23: alert(ausgabe);
24: // Multiplikations --------------------------------------------------------
25: ergebnis = zahl_1 * zahl_2;
26: ausgabe = "Das Produkt aus " + zahl_1 + " und " + zahl_2 + " ist ";
27: ausgabe = ausgabe + ergebnis + "!";
28: alert(ausgabe);
29: // Division ---------------------------------------------------------------
30: ergebnis = zahl_1 / zahl_2;
```

```
31: ausgabe = "Der Quotient aus " + zahl_1 + " und " + zahl_2 + " ist ";
32: ausgabe = ausgabe + ergebnis + "!";
33: alert(ausgabe);
34: // Modulo ------------------------------------------------------------------
35: ergebnis = zahl_1 % zahl_2;
36: ausgabe = "Der Rest der Division aus " + zahl_1 + " und " + zahl_2 + " ist ";
37: ausgabe = ausgabe + ergebnis + "!";
38: alert(ausgabe);
39: </script>
```

A 1.6

++x erhöht den Wert der Variablen `x` um 1 bevor die Zeile ausgeführt wird.

x++ erhöht den Wert der Variablen `x` um 1 nachdem die Zeile ausgeführt wurde.

– –x erniedrigt den Wert der Variablen `x` um 1 bevor die Zeile ausgeführt wird.

x– – erniedrigt den Wert der Variablen `x` um 1 nachdem die Zeile ausgeführt wurde.

A 1.7

Listing 5.2. rechteck.html

```
 1: <script type="text/javascript">
 2: // Variablen ------------------------------------------------------------ //
 3: var a = 0;
 4: var b = 0;
 5: var umfang = 0;
 6: var flaeche = 0;
 7: var ausgabe = "";
 8:
 9: // Eingabe ------------------------------------------------------------- //
10: a = Number(prompt("Länge der ersten Seite:", ""));
11: b = Number(prompt("Länge der zweiten Seite:", ""));
12:
13: // Verarbeitung und Ausgabe ------------------------------------------- //
14:
15: // Umfang -------------------------------------------------------------
16: umfang = 2*(a + b);
17: ausgabe = "Der Umfang eines Rechtecks mit den Seitenlängen " + a + " und "
18:         + b + " ist ";
19: ausgabe = ausgabe + umfang + "!";
20: alert(ausgabe);
21:
22: // Fläche -------------------------------------------------------------
23: flaeche = a * b;
24: ausgabe = "Die Fläche eines Rechtecks mit den Seitenlängen " + a + " und "
25:         + b + " ist ";
26: ausgabe = ausgabe + flaeche + "!";
27: alert(ausgabe);
28: </script>
```

A 1.8 Einfach mal ausprobieren.

A 1.9 Einfach mal ausprobieren.

A 1.10 Einfach mal ausprobieren.

A 1.11

Listing 5.3. kreis.html

```
1: <script type="text/javascript">
2: // Variablen ------------------------------------------------------------ //
3: var radius = 0;
4: var umfang = 0;
5: var flaeche = 0;
6: var ausgabe = "";
7:
8: // Eingabe ------------------------------------------------------------- //
9: radius = Number(prompt("Radius:", ""));
```

```
 10:
 11: // Verarbeitung und Ausgabe ------------------------------------------------ //
 12:
 13: // Umfang ---------------------------------------------------------------------
 14: umfang = 2*Math.PI*radius;
 15: ausgabe = "Der Umfang eines Kreises mit dem Radius " + radius + " ist ";
 16: ausgabe = ausgabe + umfang + "!";
 17: alert(ausgabe);
 18:
 19: // Fläche ---------------------------------------------------------------------
 20: flaeche = Math.PI*radius*radius;
 21: ausgabe = "Die Fläche eines Kreises mit dem Radius " + radius + " ist ";
 22: ausgabe = ausgabe + flaeche + "!";
 23: alert(ausgabe);
 24: </script>
```

A 1.12

```
...
   5: var min = 0;
...
  12: min = Math.min(a, b);
...
  15: alert("Die kleinere Zahl von " + a + " und " + b + " ist " + min + "!");
...
```

A 1.13

```
...
  10: gerundet = Math.round(zahl*1000)/1000;
  11:
  12: // Ausgabe ------------------------------------------------------------- //
  13: alert("Die Zahl " + zahl + " auf 3 Nachkommstellen gerundet: " + gerundet);
...
```

A 1.14 `toFixed()` wandelt die Zahl in eine Zeichenkette mit der genau festgelegten Anzahl von Nachkommastellen um. Daher ist `toFixed()` eher für eine Ausgabe geeignet mit einer genau festgelegten Anzahl von Nachkommastellen z.B. für Geldbeträge.

A 1.15 Die Anzahl der Nachkommastellen muss eine ganze Zahl sein. Also ist es hier notwendig auf eine ganze Zahl abzurunden. Eigentlich müsste jetzt noch auf negative Zahlen getestet werden. Aber Entscheidungen kommen erst im nächsten Kapitel.

A 1.16 Wenn kein Index angegeben wird, wird die 0 verwendet.

A 1.17

Listing 5.4. string_char_2.html

```
  1: <script type="text/javascript">
  2: // Variablen ------------------------------------------------------------------
  3: var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz ";
  4:
  5: document.writeln("<pre>");
  6: document.writeln("Der Text lautet: '" + str + "'");
  7:
  8: // Zeichen an einer bestimmten Position.
  9: document.writeln(str.charAt(0) + ' \t ' + str.charCodeAt(0));
 10: document.writeln(str.charAt(1) + ' \t ' + str.charCodeAt(1));
 11: document.writeln(str.charAt(2) + ' \t ' + str.charCodeAt(2));
 12: document.writeln(str.charAt(3) + ' \t ' + str.charCodeAt(3));
 13: document.writeln(str.charAt(4) + ' \t ' + str.charCodeAt(4));
 14: document.writeln(str.charAt(5) + ' \t ' + str.charCodeAt(5));
 15: document.writeln(str.charAt(6) + ' \t ' + str.charCodeAt(6));
 16: document.writeln(str.charAt(7) + ' \t ' + str.charCodeAt(7));
 17: document.writeln(str.charAt(8) + ' \t ' + str.charCodeAt(8));
 18: document.writeln(str.charAt(9) + ' \t ' + str.charCodeAt(9));
 19: document.writeln(str.charAt(10) + ' \t ' + str.charCodeAt(10));
 20: document.writeln(str.charAt(11) + ' \t ' + str.charCodeAt(11));
 21: document.writeln(str.charAt(12) + ' \t ' + str.charCodeAt(12));
 22: document.writeln(str.charAt(13) + ' \t ' + str.charCodeAt(13));
```

```
 23: document.writeln(str.charAt(14) + ' \t ' + str.charCodeAt(14));
 24: document.writeln(str.charAt(15) + ' \t ' + str.charCodeAt(15));
 25: document.writeln(str.charAt(16) + ' \t ' + str.charCodeAt(16));
 26: document.writeln(str.charAt(17) + ' \t ' + str.charCodeAt(17));
 27: document.writeln(str.charAt(18) + ' \t ' + str.charCodeAt(18));
 28: document.writeln(str.charAt(19) + ' \t ' + str.charCodeAt(19));
 29: document.writeln(str.charAt(20) + ' \t ' + str.charCodeAt(20));
 30: document.writeln(str.charAt(21) + ' \t ' + str.charCodeAt(21));
 31: document.writeln(str.charAt(22) + ' \t ' + str.charCodeAt(22));
 32: document.writeln(str.charAt(23) + ' \t ' + str.charCodeAt(23));
 33: document.writeln(str.charAt(24) + ' \t ' + str.charCodeAt(24));
 34: document.writeln(str.charAt(25) + ' \t ' + str.charCodeAt(25));
 35: document.writeln(str.charAt(26) + ' \t ' + str.charCodeAt(26));
 36: document.writeln(str.charAt(27) + ' \t ' + str.charCodeAt(27));
 37: document.writeln(str.charAt(28) + ' \t ' + str.charCodeAt(28));
 38: document.writeln(str.charAt(29) + ' \t ' + str.charCodeAt(29));
 39: document.writeln(str.charAt(30) + ' \t ' + str.charCodeAt(30));
 40: document.writeln(str.charAt(31) + ' \t ' + str.charCodeAt(31));
 41: document.writeln(str.charAt(32) + ' \t ' + str.charCodeAt(32));
 42: document.writeln(str.charAt(33) + ' \t ' + str.charCodeAt(33));
 43: document.writeln(str.charAt(34) + ' \t ' + str.charCodeAt(34));
 44: document.writeln(str.charAt(35) + ' \t ' + str.charCodeAt(35));
 45: document.writeln(str.charAt(36) + ' \t ' + str.charCodeAt(36));
 46: document.writeln(str.charAt(37) + ' \t ' + str.charCodeAt(37));
 47: document.writeln(str.charAt(38) + ' \t ' + str.charCodeAt(38));
 48: document.writeln(str.charAt(39) + ' \t ' + str.charCodeAt(39));
 49: document.writeln(str.charAt(40) + ' \t ' + str.charCodeAt(40));
 50: document.writeln(str.charAt(41) + ' \t ' + str.charCodeAt(41));
 51: document.writeln(str.charAt(42) + ' \t ' + str.charCodeAt(42));
 52: document.writeln(str.charAt(43) + ' \t ' + str.charCodeAt(43));
 53: document.writeln(str.charAt(44) + ' \t ' + str.charCodeAt(44));
 54: document.writeln(str.charAt(45) + ' \t ' + str.charCodeAt(45));
 55: document.writeln(str.charAt(46) + ' \t ' + str.charCodeAt(46));
 56: document.writeln(str.charAt(47) + ' \t ' + str.charCodeAt(47));
 57: document.writeln(str.charAt(48) + ' \t ' + str.charCodeAt(48));
 58: document.writeln(str.charAt(49) + ' \t ' + str.charCodeAt(49));
 59: document.writeln(str.charAt(50) + ' \t ' + str.charCodeAt(50));
 60: document.writeln(str.charAt(51) + ' \t ' + str.charCodeAt(51));
 61: document.writeln(str.charAt(52) + ' \t ' + str.charCodeAt(52));
 62:
 63: document.writeln("</pre>");
 64: </script>
```

Das Programm ist nicht schön. Wenn es um Schleifen geht, finden wir bestimmt eine bessere Lösung.

A 1.18

```
...
   6: var klein = str.toLowerCase();
   7:
   8: // Ausgabe ------------------------------------------------------------------------
   9: alert(klein);
...
```

5.2 Entscheidungen

A 2.1 Zuerst wird eine Zufallszahl zwischen 0 und 1 erzeugt. Diese wird mit 2 multipliziert und dann abgerundet. Damit sind die Werte 0 und 1 gleich wahrscheinlich. Dann wird getestet, ob der Wert größer als 0 ist. Wenn dies der Fall ist, wird "Kopf" an die alert-Funktion übergeben. Wenn nicht, wird "Zahl" an die alert-Funktion übergeben. Die alert-Funktion erzeugt dann ein Dialogfenster mit dem Text "Kopf" oder "Zahl" im Browser.

A 2.2 Bei der Eingabe von Z müsste eigentlich der Buchstabe A ausgewählt werden. Dies erfolgt aber hier nicht.

A 2.3

Eingabe	`Number()`	`parseInt()`	`parseFloat()`
5.3	5.3	5	5.3
5,3	NaN	5	5
fünf	NaN	NaN	NaN
8tausend	NaN	8	8
tausend8	NaN	NaN	NaN
3.75e3	3750	3	3750
3.75e-3	0.00375	3	0.00375
3e3.75	NaN	3	3000

A 2.4

Listing 5.5. entscheidung_if_else_3.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------- //
var zahl = 0;
var ausgabe = "";

// Eingabe --------------------------------------------------------------- //
zahl = parseInt(prompt("Geben Sie eine ganze Zahl ein!", ""));

// Verarbeitung ---------------------------------------------------------- //
if (isNaN(zahl)) {
    ausgabe = "Sie sollten doch eine ganze Zahl eingeben!";
} else {
  if (zahl % 2 == 0) {
    ausgabe = "Die Zahl " + zahl + " ist gerade!";
  } else {
    ausgabe = "Die Zahl " + zahl + " ist ungerade!";
  }
}

// Ausgabe --------------------------------------------------------------- //
alert (ausgabe);
</script>
```

A 2.5

Listing 5.6. entscheidung_plusminusnull_2.html

```
<script type="text/javascript">
// Variablen ------------------------------------------------------------- //
var zahl = 0;
var ergebnis = "";

// Eingabe --------------------------------------------------------------- //
zahl = Number(prompt("Geben Sie eine Zahl ein!", ""));

// Verarbeitung ---------------------------------------------------------- //
if (zahl > 0) {
  ergebnis = "positiv";
} else if (zahl == 0) {
  ergebnis = "null";
} else {
  ergebnis = "negativ";
}

// Ausgabe --------------------------------------------------------------- //
alert("Die Zahl " + zahl + " ist " + ergebnis + "!");
</script>
```

A 2.6

```
...
a = Math.floor(Math.random()*20)+1;
b = Math.floor(Math.random()*20)+1;
...
```

A 2.7 Wenn die Anweisung `break` weggelassen wird, dann arbeitet das Programm alle Anweisungen ab dieser Stelle ab. Also auch die, die unter anderen `case`-Anweisungen stehen.

A 2.8 Da ich keine Ahnung habe, welche Tiere Du genommen hast, gibt es hier auch keine Musterlösung. Wenn es funktioniert, dann muss es wohl richtig programmiert worden sein.

Index