Unterlagen zum JavaScript-Kurs

Version 0.7 (incl. ECMA-262 und JavaScript 1.2)

Professor Dr. Bernd Blümel

Fachhochschule Bochum

Universitätsstraße 150

44801 Bochum

Tel.: 0234-987-32-10614

email: Bernd.Bluemel@fh-bochum.de

www: http://www.fh-bochum.de/fb6/personen/bluemel/index.html

www: http://wwwtbw.mfh-iserlohn.de/informatik/bluemel/bernd.html

1 Vereinbarungen

Programme sind in der Schriftart Courier Schriftgröße 10. Alle Programme sind getestet und als Referenz in diesen Text kopiert. Reservierte Worte und Java-Sprachkonstanten im laufenden Text sind Courier kursiv.

2 Einleitende Beispiele

Beispiel 2.1 Das Programm *"Hello World"*

```
<!-- Das Programm gibt "Hello World" im Browser aus
  Dateiname: helloWorld.html -->


<HTML>
<BODY>
<SCRIPT LANGUAGE = "JavaScript">
      document.write ("<H2> Hello World </H2>");
</SCRIPT>
</BODY>
</HTML>
```

Beispiel 2.2 *"HelloWorld"* in html

```
<HTML>
<BODY>
      <H2> Hello World </H2>
</BODY>
</HTML>
```

Dennoch wollen wir Beispiel 2.1 kurz durchgehen. Die ersten vier Zeilen sind Standard-html. In der fünften Zeile beginnt JavaScript. Das html-Tag *<SCRIPT LANGUAGE = "JavaScript">* sagt dem die html-Datei darstellenden Browser, daß nun JavaScript-Code folgt. *document.write* ist ein JavaScript Befehl, der html-Text an den Browser übergibt und den Browser veranlaßt, diesen html-Text darzustellen. Der Browser stellt also jetzt *<H2> Hello World </H2>* dar und es erscheint Hello World in der Formatierungsvorschrift H2 im Browser.

Beispiel 2.3 Die Fakultäten von 1 bis 10

```
<!-- Dateiname: fakultaeten.html -->
<HTML>
<BODY>
<SCRIPT LANGUAGE = "JavaScript">
      var fact;
      var i;
      document.write("<H2> Fakult&auml;tentabelle von 1 bis 10 </H2>");
      fact = 1;
      for(i=1; i<=10; i++)
      {
            fact = fact * i;
            document.write(i + "! = " + fact + "<BR>");
      }
</SCRIPT>
</BODY>
</HTML>
```

Beispiel 2.3 ist etwas besser. Es berechnet die ersten 10 Fakultäten und stellt sie im Browser des Benutzers dar. Zunächst werden 2 Variablen deklariert (`var fact; var i`). Danach schreibt das Script in den Browser, was es tun will (nämlich die ersten 10 Fakultäten darstellen). Die Variable `fact` wird mit 1 initialisiert (`fact = 1`). Dann beginnt eine `for`-Schleife. (Die genaue Syntax der for-Schleife wird in Kapitel 5.5 dargestellt). Innerhalb der `for`-Schleife wird die Fakultät berechnet (`fact = fact * i`) und dann in das Browserfenster geschrieben (`document.write(i + "! = " + fact + "<BR>")`. Wir sehen hier, daß wir mit `document.write` mehrere Ausgaben durch ein + verknüpfen können. Hier wird also die Zahl, deren Fakultät gebildet worden ist (`i`), das Zeichen für den Fakultätsoperator, ein Blanc, das Gleichheitszeichen, noch ein Blanc (`! = `), die Fakultät (`fact`), sowie das html-Tag für eine neue Zeile (`<BR>`) in das Browserfenster geschrieben (vgl. Abbildung 2. 1).

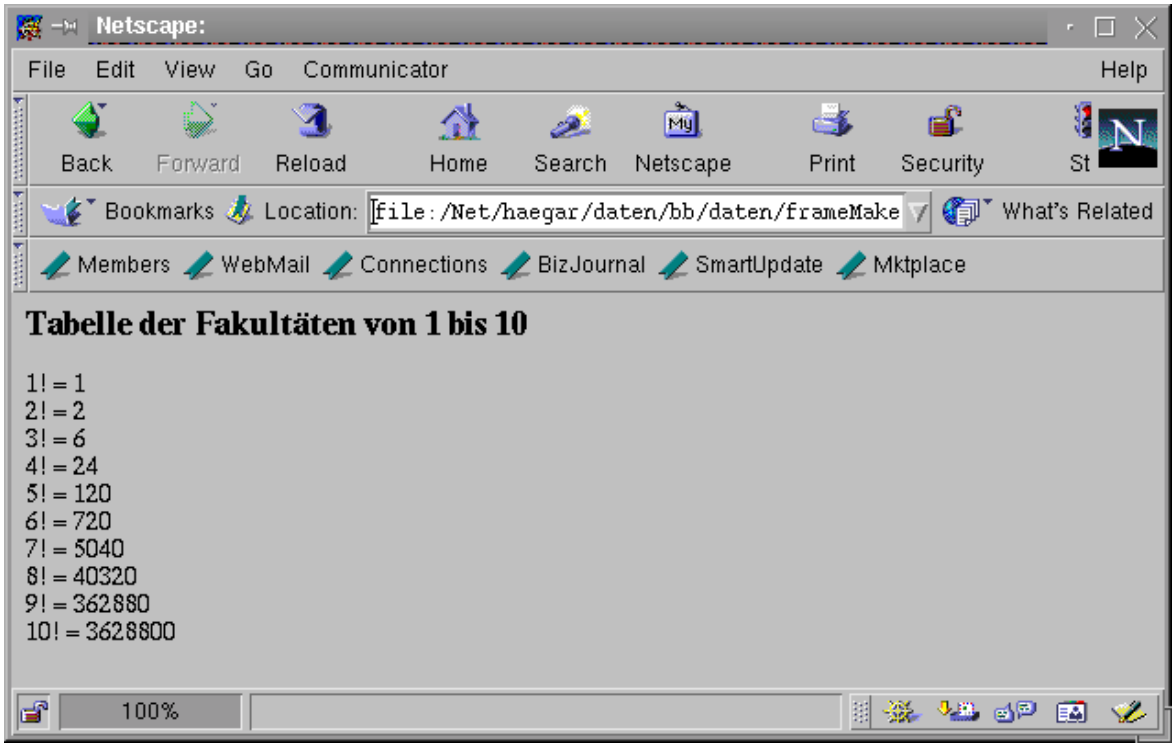


Abbildung 2. 1 Ausgabe des Fakultäten-Beispiels

Dieses kleine Programm benutzt also den Computer des Benutzers (durch seinen Browser), um die Fakultätsberechnung durchzuführen. Diese Berechnung wird jedesmal durchgeführt, wenn der Benutzer die in Beispiel 2.3 dargestellte Seite lädt.

Das ist aber auch der Schwachpunkt von Beispiel 2.3. Der Benutzer der Seite hat keinen Einfluß auf das, was der in der Seite enthaltene Code macht. Immer werden die Fakultäten von 1 bis 10 berechnet und ausgegeben. Sinnvoller wäre also, diese Fakultäten einmal auszurechnen, abzuspeichern und das Ergebnis bei jedem Aufruf der Seite zu übergeben (Beispiel 2.4).

Beispiel 2.4 Die ersten 10 Fakultäten ohne JavaScript

```
<!-- Dateiname: fak2OhneRechnung.html -->
<HTML>
<HEAD>
<TITLE> FAKULT&auml;TEN ohne Rechnung </TITLE>
```

```
</HEAD>
<BODY>
<H2> Tabelle der Fakult&auml;ten von 1 bis 10 </H2>



1! = 1 <BR>
2! = 2 <BR>
3! = 6 <BR>
4! = 24 <BR>
5! = 120 <BR>
6! = 720 <BR>
7! = 5040 <BR>
8! = 40320 <BR>
9! = 362880 <BR>
10! = 362880 <BR>
</BODY>
</HTML>
```

Beispiel 2.3 läßt sich aber leicht zu einer sinnvollen Anwendung erweitern. Stellen wir uns vor, wir wollen den Anwender entscheiden lassen, welche Fakultäten er berechnet haben will. Dann ist eine Vorgehensweise wie in Beispiel 2.4 nicht mehr möglich, da wir im Vorhinein ja nicht wissen, was für den jeweiligen Benutzer ausgegeben werden soll. Wir können also keine statische html-Seite vorhalten, sondern müssen sie bei Bedarf dynamisch erstellen. Da Fakultätsberechnungen (insbesondere bei hohen Zahlen) ziemlich rechenaufwendig sind, ist es auch eine gute Idee, die Berechnung an sich auf dem Computer des Benutzers durchzuführen und nur das Programm zur Verfügung zu stellen.

Beispiel 2.5 Fakultätsberechnung mit Benutzerinteraktion

```
<!-- Dateiname: fakMitEingabe.html -->
<HTML>
<HEAD>
      <TITLE>
            FAKULT&Auml;TEN MIT EINGABE
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm berechnet Fakult&auml;ten.
      </H2>
      <P>
            Geben Sie unten die Zahl ein, bis zu der Sie
            die Fakult&auml;ten berechnet haben m&ouml;chten!
      </P>
      <SCRIPT LANGUAGE = "JavaScript">
            var eingabe;
            var ende;
            var i;
            var fact;
            eingabe = prompt ("Zu berechnende Fakultaet:", "");
```

```
            ende = parseInt (eingabe);
            fact = 1;
            for(i=1; i<= ende; i++)
            {
                  fact = fact * i;
                  document.write(i + "! = " + fact + "<BR>");
            }
      </SCRIPT>
</BODY>
</HTML>
```

Beispiel 2.5 führt eine in JavaScript eingebaute Funktion zur Benutzer-Interaktion ein. *prompt* blendet ein Eingabefenster über dem Browser auf. Das Fenster besteht aus einem Textteil und einem Eingabefeld (vgl. Abbildung 2. 2).

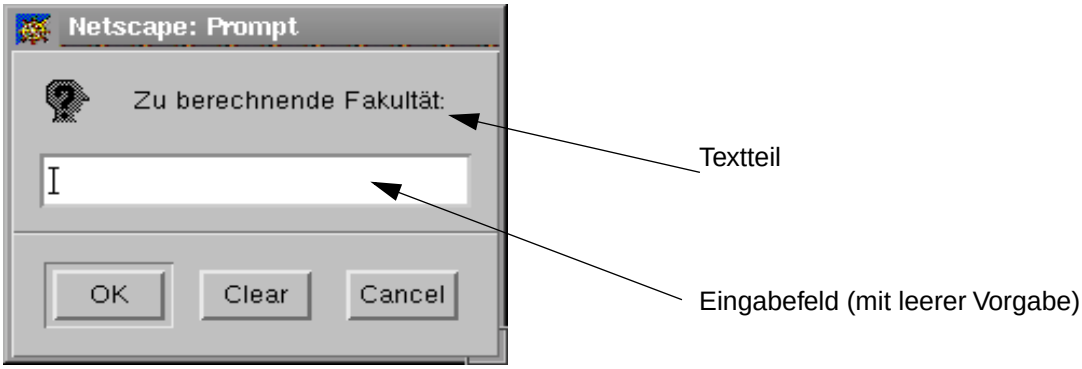


Abbildung 2. 2 Ein mit "prompt" erzeugtes Fenster

Der erste Parameter der Funktion *prompt* (in unserem Fall *"Zu berechnende Fakultät:"*) wird im Textteil dargestellt. Der zweite Parameter (in unserm Fall *""*, also leer) ist der Vorgabewert für das Eingabefeld. *prompt* gibt die Eingabe des Benutzers als *String* zurück.

Da von *Strings* keine Fakultäten gebildet werden können, muß der *String* noch in einen Integer umgewandelt werden. Dies wird von *parseInt* durchgeführt. Wir sehen hier auch schon, daß in JavaScript-Variablendeklarationen zwischen Variablentypen nicht unterschieden wird. *eingabe* und *ende* wurden beide mittels des JavaScript-Schlüsselworts *var* deklariert, obwohl *eingabe* einen *String* aufnimmt, *ende* hingegen einen Integer.

Auf der Variablen *ende* ist jetzt der Wert abgelegt, bis zu dem die Fakultäten berechnet werden sollen. Der weitere Code entspricht Beispiel 2.4 mit der einzigen Ausnahme, daß die *for*-Schleife nicht bis 10, sondern bis *ende* läuft.

Zum Abschluß dieses Kapitels ein weiteres kleines interaktives Programm:

Beispiel 2.6 Ein Additionsprogramm

```
<!-- Dateiname: addition.html-->
<HTML>
<HEAD>
```

```
      <TITLE>
            Additionsprogramm
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm addiert 2 Zahlen.
      </H2>
      <P>
            Geben Sie in den Pop-Up-Fenstern die Zahlen ein, die Sie zu
            addieren wünschen!
      </P>
      <SCRIPT LANGUAGE = "JavaScript">
            var ersterSummand;
            var zweiterSummand;
            var summe;
            var eingabe;
            eingabe = prompt ("Erster Summand", "");
            ersterSummand = parseInt (eingabe);
            eingabe = prompt ("Zweiter Summand", "");
            zweiterSummand = parseInt (eingabe);
            summe = ersterSummand + zweiterSummand;
            document.write("<B> Die Summe ist: " + summe + "</B>");

      </SCRIPT>
</BODY>
</HTML>
```

In Beispiel 2.6 werden 2 Werte mittels *prompt* eingelesen. Die Eingaben werden durch *parseInt* in Integer umgewandelt, addiert und danach im Browser ausgegeben.

3 Einfügen von JavaScript in html-Dateien

3.1 <SCRIPT> und </SCRIPT>

Der einfachste Weg, JavaScript-Code in eine html-Datei einzufügen, besteht darin, JavaScript-Anweisungen zwischen den html-Tag's *<SCRIPT LANGUAGE = "JavaScript">* und *</SCRIPT>* direkt in die html-Datei aufzunehmen. Dies ist auch die in allen Beispielen von Kapitel 2 gewählte Vorgehensweise.

Abkürzend kann hier *LANGUAGE = "JavaScript"* weggelassen werden. Der Sinn in der Angabe der Scriptsprache liegt darin, daß es neben JavaScript in Zukunft andere Scriptsprachen für Internet-Browser geben könnte[1]. Durch die Angabe der Script-Sprache kann der Browser entscheiden, ob er das Script ausführen kann, oder die Anweisungen des Scriptes besser ignorieren sollte.

3.2 Das SRC-Attribut des <SCRIPT>-Tags

Eine schönere (und im weiteren Verlauf dieser Ausarbeitung auch zumeist genutzte) Möglichkeit ist, anstelle des JavaScript-Codes einen Dateinamen innerhalb des *SCRIPT*-Tags anzugeben. Der Browser lädt die referenzierte Datei. Er verhält sich dabei genauso, als ob die in der referenzierten Datei enthaltenen JavaScript-Anweisungen direkt in der html-Datei stehen würden. Ich veranschauliche dies an Beispiel 2.5.

Beispiel 3.1 Fakultätsberechnung mit Benutzerinteraktion und JavaScript-Code in eigener Datei (Beispiel 2.5 abgewandelt)

```
<! Dateiname: fakMitEingabe.html>
<!-- Dateiname: fakMitEingabe.html -->
<HTML>
<HEAD>
      <TITLE>
            FAKULT&Auml;TEN MIT EINGABE
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm berechnet Fakult&auml;ten.
      </H2>
      <P>
            Geben Sie unten die Zahl ein, bis zu der Sie
            die Fakult&auml;ten berechnet haben m&ouml;chten!
      </P>
      <SCRIPT LANGUAGE = "JavaScript" SRC = "./javaScript/fakultaet.js">
      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript - Datei

1.In Wirklichkeit gibt es bereits zwei andere Scriptsprachen: VBScript von Microsoft (wird z.Zt. nur vom Internet Explorer unterstützt); Tcl wird in der html 4.0-Spezifikation als Beispiel für eine Script-Sprache benutzt, z. Zt. aber noch von keinem Browser unterstützt.

```
// Dieses Java-Script-Programm berechnet Fakultaeten
// Dateiname fakultaet.js

var eingabe;
var ende;
var i;
var fact;
eingabe = prompt ("Zu berechnende Fakultaet:", "");
ende = parseInt (eingabe);
fact = 1;
for(i=1; i<= ende; i++)
{
      fact = fact * i;
      document.write(i + "! = " + fact + "<BR>");
}
```

Beispiel 3.1 zeigt, daß die JavaScript-Datei über das *SRC* (Source)-Attribut des *SCRIPT*-Tags in die html-Datei geladen wird. Die JavaScript-Datei heißt *fakultaet.js*. Die Extension *js* ist notwendig, zumindestens für manche Browser. *fakultaet.js* steht im Verzeichnis *javaScript* unterhalb des Verzeichnisses, in dem die html-Datei (*fakMitEingabe.html)* beheimatet ist. Deshalb wird im *SRC*-Attribut der relative Pfad zu *fakultaet.js* angegeben. Pfad- und Dateinamen werden in Anführungszeichen eingeschlossen.

Der Code in *fakultaet.js* entspricht dem JavaScript-Code zwischen den <*SCRIPT LANGUAGE* = *"JavaScript"*> und <*/SCRIPT*>-Tags in Beispiel 2.5. Beispiel 2.5 und Beispiel 3.1 erzeugen dasselbe Ergebnis.

Die Auslagerung der JavaScript-Anweisungen in eine eigene Datei hat viele Vorteile:

- Die html-Datei wird kleiner und besser überschaubar.
- Die Erstellung von html- und JavaScript-Datei kann mit unterschiedlichen Werkzeugen erfolgen.
- Von mehreren html-Seiten genutzter JavaScript-Code wird in einer Datei vorgehalten. Dadurch wird:
    - Die Wartung des Codes (und der html-Seiten) vereinfacht.
    - Festplattenspeicherbedarf vermindert.
    - Die Ladezeit insgesamt verkürzt, da eine JavaScript-Datei, die von mehreren html-Seiten genutzt wird, nur einmal geladen wird.

3.3 Das javascript: Pseudo Protokoll in einer URL

Eine weitere Möglichkeit, JavaScript-Code in eine Seite einzubinden, besteht in der Benutzung des javascript: Pseudo Protokoll in einer URL. Ich erläutere dies an Beispiel 3.2:

Beispiel 3.2 javascript: Pseudo Protokoll in einer URL

```
<!-- Dateiname: helloWorldMitPseudoProtocoll.html -->
<HTML>
<HEAD>
      <TITLE>
            Hello World mit Pseudoprotokoll
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm stellt Hello World in einem &uuml;ber dem Browser
            aufgehenden Fenster dar.
      </H2>
      <P>
            <A HREF=javascript:alert("HelloWorld!")>
                  Klicken Sie hier, um "HelloWorld" in einem
                  eigenen Fenster darzustellen!
            </A>
      </P>
</BODY>
</HTML>
```

Durch *<A HREF=javascript*: wird dem Browser mitgeteilt, daß nun Anweisungen in JavaScript folgen. Es können beliebig viele JavaScript-Anweisungen angeschlossen werden. Die JavaScript-Anweisungen müssen durch *;* getrennt werden. Klickt der Benutzer auf den Link, führt der Browser die Anweisungen durch und stellt das Ergebnis der letzten Anweisung im Browser-Fenster dar. Hat die letzte Anweisung kein darstellbares Ergebnis, ändert sich der Inhalt des Browser-Fensters nicht.

Zugleich lernen wir in Beispiel 3.2 ein neues JavaScript-Kommando[2] kennen. *alert()* blendet ein Fenster über dem Browser-Fenster auf. Der übergebene Parameter, in diesem Fall *"HelloWorld!"* , wird dargestellt. Das Fenster enthält einen "OK-Button", mittels dessen man es "wegklicken" kann (vgl. Abbildung 2. 2).

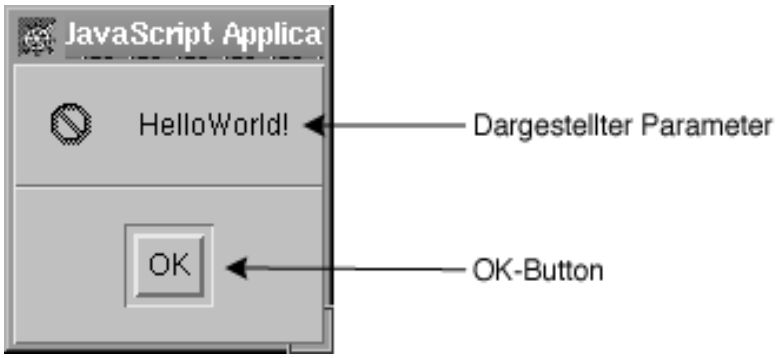


Abbildung 3. 1 Ein mit "alert" erzeugtes Fenster

2.Dies ist nicht ganz richtig. alert() ist strenggenommen kein Kommando, sondern eine Methode des window-Objekts. Das lernen wir aber so richtig erst in Kapitel 8.

Die Implementierung des javascript: Pseudo Protokolls ist zumindest in Netscape 4.7 mit vielen Fehlern behaftet. So ist es z.B. in dieser Version des Browsers nicht möglich, Blancs in die JavaScript-Kommandos aufzunehmen. Das zusammengeschriebene *HelloWorld!* ist also kein Tippfehler, sondern notwendig, da die Ausführung des JavaScript-Kommandos sonst mit einem Fehler abbricht. Das javascript: Pseudo Protokoll sollte daher sehr vorsichtig benutzt werden.

Eine schöne Eigenschaft des javascript: Pseudo Protokolls in Netscape 4.7 möchte ich allerdings noch erwähnen. Gibt man in Netscape 4.7* als URL nur javascript: ein, öffnet der Browser die Netscape Communicator Konsole. Sie besteht aus einer Eingabemöglichkeit und einem Ausgabebereich. Im Eingabefeld können beliebige JavaScript-Kommandos eingegeben werden. Ihre Ausgaben oder eventuelle Fehlermeldungen erscheinen im Ausgabebereich (vgl. Abbildung 2. 2). Dies ist eine einfache Möglichkeit, JavaScript-Kommandos auszuprobieren, ohne erst eine html-Datei schreiben zu müssen. Zugleich dient dieses Fenster zur Fehleranalyse, wenn in JavaScript-Anwendungen Fehler aufgetreten sind. Nach dem Eingeben von javascript: als URL öffnet sich die Javascript-Konsole und stellt die aufgetretenen Fehler und die Zeilen, in der die Fehler auftraten, dar. Microsofts Internet Explorer stellt diese Technik allerdings nicht zur Verfügung.

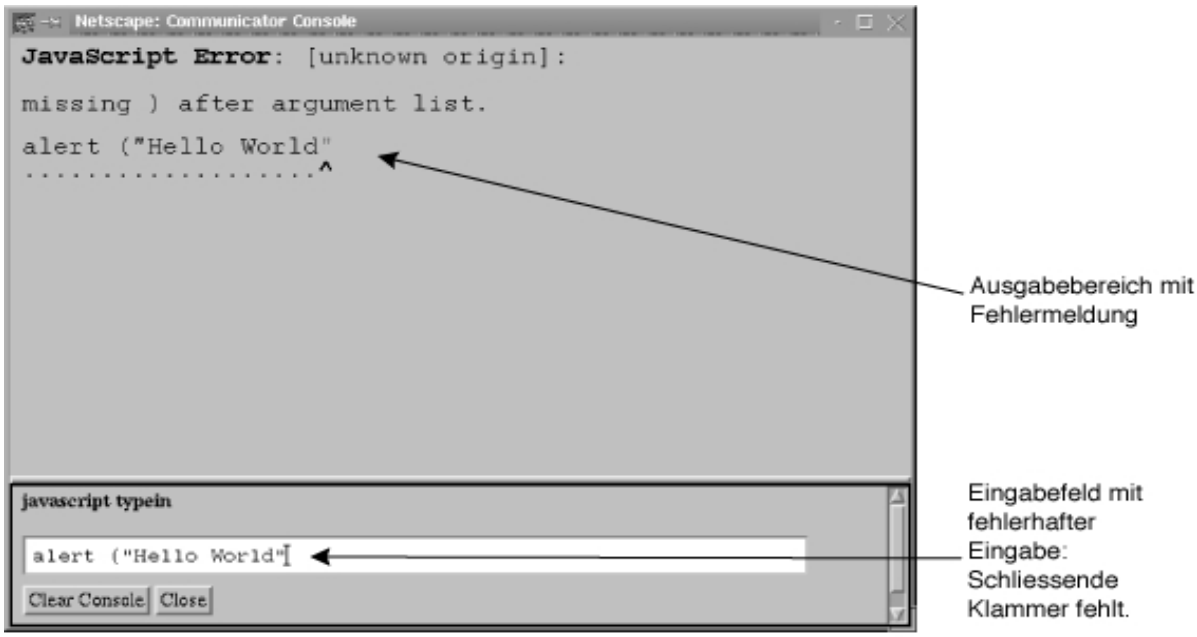


Abbildung 3. 2 Die Netscape Communicator Konsole

3.4 Event Handler und JavaSript in Style Sheets

Es gibt zwei weitere Möglichkeiten, JavaScript in html-Dateien einzubetten:

- **Event Handler:** Alle JavaScript Objekte verfügen über Event Handler. JavaScript Objekte und ihre Event Handler werden ausführlich ab Kapitel 10 besprochen. In diesem Zusammenhang wird dann auch die Einbindung von JavaScript-Code über Event Handler diskutiert.

- **JavaSript in Style Sheets:** Diese Technik wird vom Microsoft Internet Explorer nicht unterstützt. Ich werde sie daher nicht weiter diskutieren.

4 Die JavaScript Syntax

4.1 Variablen, Datentypen und der Zuweisungsoperator

JavaScript ist eine untypisierte Sprache. Variablen werden mit dem Schlüsselwort var deklariert und können danach Werte eines beliebigen Typs annehmen. Der Typ einer Variablen kann im Programmverlauf wechseln.

Beispiel 4.1 Wechsel des Variablentyps

```
var i; // Variable i deklariert
i = 10; // i nimmt Integerwert 10 an, i ist jetzt Integervariable
i = “zehn”; // i nimmt den Stringwert “zehn” an, i ist jetzt ein String
```

Variablen können beliebig im Programmcode deklariert werden. Variablen können bei der Deklaration initialisiert werden. Selbst die Deklaration einer Variablen ist eigentlich unnötig (vgl. Beispiel 4.2). Dennoch ist es guter Programmierstil, Variablen zu deklarieren. Dadurch, daß Variablen nicht deklariert werden müssen, können Schreibfehler recht drastische Auswirkungen haben (vgl. Beispiel 4.3)[3]. Werden Variablen deklariert, ist es leichter möglich, Schreibfehler durch Kontroll-Lesen aufzudecken.

Jede Variablendeklaration kann mit einem `;` abgeschlossen werden.

Merke: Nicht nur jede Variablendeklaration, jede (Ausnahmen bestätigen die Regel) Anweisung kann mit einem `;` abgeschlossen werden. Solange jede Anweisung in einer eigenen Zeile steht, ist dies allerdings nicht notwendig. `;` können dann weggelassen werden.

Der Zuweisungsoperator ist das Gleichheitszeichen =.

Beispiel 4.2 Deklarationen und Zuweisungen

```
<!-- Dateiname: deklarationUndZuweisung.html -->
<HTML>
<HEAD>
        <TITLE>
                Deklarationen und Zuweisungen
        </TITLE>
</HEAD>

<BODY>
        <H2>
                Dies Programm f&uuml;hrt Deklarationen und Zuweisungen aus.
        </H2>
        <P>
                <SCRIPT LANGUAGE = "JavaScript" SRC="./javaScript/deklarationUndZuweisung.js">

                </SCRIPT>
        </P>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: deklarationUndZuweisung.js
```

3.FORTRAN ist eine ältere Programmiersprache, in der Variablen ebenfalls nicht deklariert werden müssen. Der NASA hat ein Schreibfehler in einem FORTRAN-Programm (Programm zur Satellitensteuerung) in den 70er-Jahren 3 Millionen US Dollar (und einen Satelliten) gekostet.

```
var x;              // x deklariert
var y = 19          // y deklariert, Wert 19 zugewiesen
                // y ist Integer-Variable mit Wert 19
document.write(y + "<BR>");  // y wird ausgegeben
                // 19 erscheint im Browser
x = 4.7             // x den Wert 4.7 zugewiesen, x ist jetzt ein Float
document.write(x + "<BR>");  // x wird ausgegeben
                // 4.7 erscheint im Browser
x = "Datum"         // x ist jetzt String mit Wert "Datum"
document.write(x + "<BR>");  // x wird ausgegeben
                // Datum erscheint im Browser

var i;              // i deklariert
i = "Test";         // i ist String mit Wert "Test"
document.write(i + "<BR>");  // i wird ausgegeben
                // Test erscheint im Browser
z = 4;             // z wurde nicht deklariert, dies ist in JavaScript nicht
                //notwendig, z ist jetzt Integer-Variable mit Wert 4
document.write(z + "<BR>");  // z wird ausgegeben
                // 4 erscheint im Browser
```

Beispiel 4.3 Schreibfehler erzeugt fehlerhaftes Programm

```
<!-- Dateiname: schreibfehler.html -->
<HTML>
<HEAD>
      <TITLE>
            Programm mit Schreibfehler
      </TITLE>
</HEAD>
<BODY>
      <H2>
            Dies Programm addiert und multipliziert 2 Zahlen.
      </H2>
      <P>
            Geben Sie in den Pop-Up-Fenstern die Zahlen ein, die Sie zu
            addieren und multiplizieren w&uuml;nschen!
      </P>
      <SCRIPT LANGUAGE = "JavaScript" SRC="./javaScript/schreibfehler.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: schreibfehler.js
// Dateiname: schreibfehler.js
eingabe = prompt ("Erster Summand", "");
ersterSummand = parseInt (eingabe);
eingabe = prompt ("Zweiter Summand", "");
zweiterSummand = parseInt (eingabe);
ergebnis = ersterSummand + zweiterSummand;
```

```
document.write("<B> Die Summe ist: " + ergebnis + "</B><BR>");
ergbnis = ersterSummand * zweiterSummand; //Schreibfehler ergbnis
                                  //statt ergebnis
document.write("<B> Das Produkt ist: " + ergebnis + "</B><BR>");
                  // die Summe wird zum zweiten Mal ausgegeben
                  // das Programm ist fehlerhaft
```

Regeln zu Variablennamen

Merke: Diese Regeln gelten auch für alle anderen Bezeichner.

- Variablennamen beginnen mit einem Buchstaben, dem Dollarzeichen[4] oder einem Unterstrich (_).
- Variablennamen können nach dem ersten Buchstaben Ziffern, Buchstaben,Dollarzeichen und Unterstriche in beliebiger Reihenfolge enthalten.
- Variablennamen können beliebig lang sein.
- Groß- und Kleinschreibung **spielt eine Rolle** (summe ≠ Summe ≠ suMME ≠ SUMME), JavaScript ist “case-sensitive”.

4.2 Arrays

Arrays werden von JavaScript ebenfalls unterstützt. Sie werden wie Variablen deklariert:

```
var meinArray;
```

Vor der Benutzung müssen Arrays jedoch mit dem Schlüsselwort *new* erzeugt werden.

```
meinArray = new Array ();
```

Der Array-Index wird von 0 hochgezählt. Der Zugriff auf die einzelnen Felder eines Arrays erfolgt über eckige Klammern:

```
meinArray[0] = 1;
```

Die Elemente eines Arrays werden bei Benutzung erzeugt. Die Elemente eines Array müssen, im Gegensatz zu Arrays in anderen Programmiersprachen, **nicht** vom gleichen Typ sein.

```
meinArray[1] = “diesesFeldIstEinString”;
```

Die Werte mehrerer Felder eines Arrays können mit einer Anweisung zugewiesen werden. Dies geschieht, indem die Werte in eckigen Klammern eingeschlossen und durch Kommata getrennt dem Arraynamen zugewiesen werden:

```
meinArray = [0,1,2];
```

ist gleichbedeutend mit:

```
meinArray[0] = 0;
```

4.Im Internet-Explorer sind Dollarzeichen nicht erlaubt.

```
meinArray[1] = 1;
meinArray[2] = 2;
```

Arrays sind in Wahrheit Objekte[5] und verfügen daher über Eigenschaften[6] (JavaScript-Wort für Instanzvariablen) und Funktionen. Eine wichtige Eigenschaft ist *length*. *length* gibt die Länge des Arrays aus.

Beispiel 4.4 zeigt ein Beispiel der Benutzung von Arrays.

Beispiel 4.4 Arbeiten mit Arrays

```
<!-- Dateiname: arbeitenMitArrays.html -->
<HTML>
<HEAD>
      <TITLE>
            Arbeiten mit Arrays
      </TITLE>
</HEAD>

<BODY>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/arbeitenMitArrays.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: arbeitenMitArrays.js

var meinArray;
var laenge;
meinArray = new Array();
meinArray[0] = 4;
meinArray[1] = 4;
meinArray[2] = "Unterschied"; // nicht alle Elemente eines Arrays
                        // muessen gleichen Typ haben
laenge = meinArray.length; // laenge ist jetzt 3;
document.write("<B> Die L&auml;ge des Arrays ist : " + laenge + "</B><BR>")
```

4.3 Primitive Datentypen und Referenztypen

JavaScript unterscheidet zwei sehr unterschiedliche Arten von Datentypen, primitive und Referenzdatentypen. Primitive Datentypen sind Zahlen (Numbers), bool'sche Werte und Strings.

5.Der Objektbegriff wird später (Kapitel 7) ausführlich erklärt.
6.Auch Eigenschaften (Instanzvariablen) werden im Zusammenhang mit Objekten in Kapitel 7 erklärt.

Arrays und die noch unbekannten Objekte sind Referenzdatentypen[7]. Was aber ist der Unterschied? Betrachten wir zunächst primitive Datentypen. Variablen primitiver Datentypen enthalten den Wert der Variable an sich. Durch

```
var i = 5;
```

wird die Variable i deklariert und ihr der Wert 5 zugewiesen. i enthält nun den Wert 5.

Durch

```
var j = i;
j = 6;
```

wird die die Variable j deklariert und ihr der Wert der Variablen i zugewiesen. j hat nun zunächst den Wert 5. Die nächste Zeile ändert den Wert der Variablen j auf 6, die Variable i bleibt unberührt und enthält weiter den Wert 5.

Anders verhält es sich bei Referenztypen. Variablen von Referenztypen enthalten eine Referenz auf die Variable, die sie darstellen. Referenzen sind im wesentlichen Hauptspeicheradressen. Das heißt, Variablen von Referenztypen enthalten nicht den Wert der Variable an sich, sondern die Hauptspeicheradresse, wo dieser Wert zu finden ist. Und dies hat Konsequenzen. Durch

```
var meinArray = new Array();
meinArray = [0,1,2];
```

wird ein Array definiert. Die ersten drei Felder des Arrays erhalten die Werte 0, 1, 2.

```
var meinArray1 = new Array();
meinArray1 = meinArray;
```

definiert ein zweites Array. Dem zweiten Array wird die Hauptspeicheradresse des ersten Arrays zugewiesen. Das zweite Array enthält jetzt schon die Werte 0, 1 und 2. *meinArray* und *meinArray1* sind identisch.

```
meinArray1[0] = 7;
```

ändert somit auch meinArray[0] auf den Wert 7.

Wir veranschaulichen uns dies noch einmal an Beispiel 4.5.

Beispiel 4.5 Primitive Typen und Referenztypen

```
<!-- Dateiname: primitivUndReferenz.html -->
<HTML>
<HEAD>
      <TITLE>
            Primitive und Referenzdatentypen
      </TITLE>
</HEAD>

<BODY>
      <SCRIPT LANGUAGE = "JavaScript"
```

7. Auch Funktionen (vgl Kapitel 6) sind Referenzdatentypen. Ich verzichte aber auf die Darstellung von Funktionen als Variablen.

```
                    SRC = "./javaScript/primitivUndReferenz.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// primitivUndReferenz.js
var zahl = 7;
var zahl1;
var string = "test";
var string1;
var array = new Array();
var array1 = new Array();
array = [0,1,2];

zahl1 = zahl;
zahl1 = 8;

string1 = string;
string1 = "anders";

array1 = array;
array1[0] = 12;

document.write("zahl: " + zahl + "   zahl1: " + zahl1 + "<BR>");
document.write("string: " + string + "   string1: " + string1 + "<BR>");
document.write("array: " + array + "   array1: " + array1 + "<BR>");
```

Beispiel 4.5 erzeugt die Abbildung 4. 1 dargestellte Ausgabe im Browser:

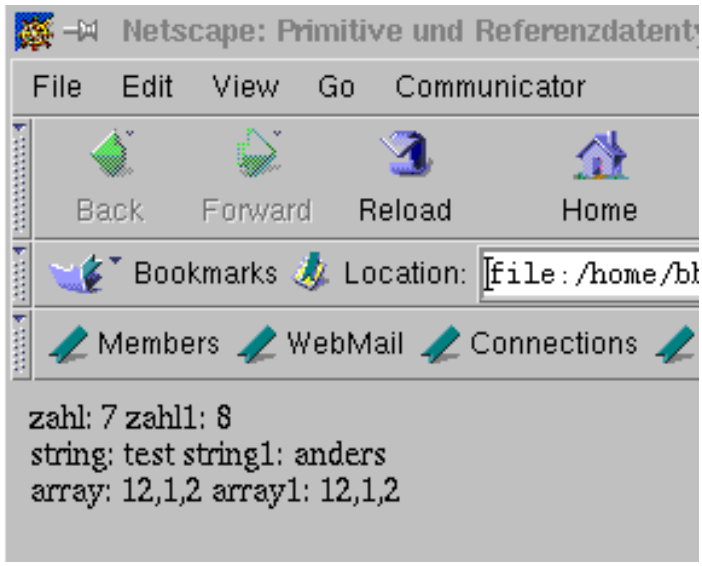


Abbildung 4. 1 Zuweisung von Variablen bei primitiven und Referenztypen

zahl, zahl1, string und *string1* sind primitive Typen. Durch die Zeilen

```
zahl1 = zahl;
```

und

```
string1 = string;
```

werden also die jeweils aktuellen Werte der Variablen *zahl* und *string* nach *zahl1* und *string1* kopiert. *zahl und zahl1, string* und *string1* sind aber unterschiedliche Variablen, die unterschiedliche Speicherplätze im RAM einnehmen. Durch die Zuweisungen:

```
zahl1 = 8;
string1 = "anders";
```

werden also nur *zahl1* respektive *string1* geändert. *zahl* und *string* behalten ihre alten Werte.

array und *array1* hingegen sind Referenztypen. Die Zeile

```
array1 = array;
```

weist die Hauptspeicheradresse von *array* auf *array1* zu. *array* auf *array1* zeigen jetzt auf den gleichen Wert. Auch *array1* hat jetzt schon den Inhalt *[0,1,2]* . Durch

```
array1[0] = 12;
```

wird also, da *array1* und *array* nur unterschiedliche Namen für dieselbe Variable sind, das erste Feld von *array* ebenfalls geändert. *array* und *array1* sind also zwei Namen für die gleiche Variable.

4.4 Operatoren

4.4.1 Zuweisung

Wie schon in Kapitel 4.1 gesehen, ist der Zuweisungsoperator das Gleichheitszeichen:

```
var i;
i = 2;
```

4.4.2 Arithmetische Operatoren

JavaScript stellt die von anderen Programmiersprachen bekannten arithmetischen Operatoren zur Verfügung (vgl. Beispiel 4.6).

Beispiel 4.6 Arithmetische Operatoren angewendet auf Zahlen

```
var i = 2;
var j = 3;
var l;
var r= 2;
var s=3;
var t;
l = i + j; // l = 5;
l = i - j; // l = -1
l = j % i; // l = 1 (Modulo-Bildung)
```

```
l = i * j; // l = 6
t = s/r; // 0.6666666 Division
i++; // bedeutet i = i+1; i ist jetzt 4
i--; // bedeutet i = i - 1; i ist jetzt wieder 3
```

Der Operator + zeigt auf Strings angewendet ein Sonderverhalten. Er fügt zwei Strings zu einem Ergebnisstring zusammen.

Beispiel 4.7 Der + Operator angewendet auf Strings

```
var s1=”ab”;
var s2= “cd”;
s1 = s1 + s2 ; // “abcd” + Operator hat bei Strings Sonderbedeutung
```

Ist nur ein Operand ein String, versucht JavaScript den anderen Operanden in einen String umzuwandeln, um danach die beiden Operanden zusammenzufügen (vgl. Beispiel 4.6).

Beispiel 4.8 Typumwandlung beim + Operator

```
var s1=”ab”;
var s2= 2;
var s3;
var array = new Array();
array = [0,1,2];
s1 = s1 + s2; // s1 hat nun den Wert "ab2"
s3 = s1 + array // s3 hat den Wert "ab20,1,2"
```

Ein *Array* in einen String umwandeln bedeutet also, die Werte des *Array* durch , getrennt, in einen String zu schreiben.

Die anderen arithmetischen Operatoren versuchen ihre Operanden, sofern es sich nicht um bereits Zahlen handelt, in Zahlen zu konvertieren (vgl. Beispiel 4.6).

Beispiel 4.9 Arithmetische Operatoren und Konvertierung

```
var s = "12"; // s ist ein String
var z = 5;
var ergebnis;
ergebnis = z * s; // s wird in 12 konvertiert, dann wird multipliziert
                  // ergebnis erhält den Wert 60
```

4.4.3 Operatoren mit Zuweisung

Wie alle C-ähnlichen Sprachen erlaubt JavaScript in einigen Fällen abkürzende Schreibweisen (vgl. Beispiel 4.10).

Beispiel 4.10 Einige abkürzende Schreibweisen

```
i += j // i = i + j
i -= j // i = i -j
i *= j // i = i * j
i /= j // i = i / j
i %= j // i = i % j
```

4.4.4 Vergleichsoperatoren

Test auf Gleichheit

Der == Operator testet auf Gleichheit, er gibt *true* zurück, wenn die beiden Operanden gleich sind, *false* sonst. Dabei gelten folgende Regeln:

- Die Operanden sind vom gleichen Typ:
  - **Strings:** Wenn beide Operanden gleiche Character an der gleichen Stelle aufweisen.
- o **Zahlen:** Wenn sie den gleichen Wert haben.
- o **Boolean:** Wenn entweder beide true oder beide false sind.
- o **Referenztypen:** Wenn beide Operanden Referenzen auf das gleiche Array oder Objekt sind.

Beispiel 4.11 faßt dies zusammen.

Beispiel 4.11 Test auf Gleichheit bei Operanden vom gleichen Typ

```
var i=5;
var j=6;
var ergebnis;
ergebnis = i == j; // Wert von ergebnis: false
i = 6;
ergebnis = i == j; // Wert von ergebnis: true
i = "test";
j = "test";
ergebnis = i == j; // Wert von ergebnis: true
i = "test1";
ergebnis = i == j; // Wert von ergebnis: false
var array;
var array1;
array = new Array();
array1 = new Array();
array = [1,2,2];
array1 = [1,2,2];
ergebnis = array == array1 // Wert von ergebnis: false, da unterschiedliche
                           // Arrays mit allerdings gleichem Inhalt.
array = array1;
ergebnis = array == array1 // Wert von ergebnis: true, da nun gleiche Arrays
```

- Die Operanden sind nicht vom gleichen Typ: Zunächst wird versucht, beide Operanden in den gleichen Typ zu konvertieren. Dabei gelten die folgenden Regeln:
  - o **Zahl und String:** Es wird versucht, den String in eine Zahl umzuwandeln. Funktioniert dies nicht, wird *false* zurückgegeben, ansonsten wird der Vergleich durchgeführt.
  - o **Zahl und Boolean:** Aus *true* wird 1, aus *false* 0.

- **Alle anderen Kombinationen:** Das Ergebnis ist *false*.

Beispiel 4.11 faßt dies zusammen.

Beispiel 4.12 Test auf Gleichheit bei Operanden verschiedenen Typs

```
var i = 3;
var j = "test"
var ergebnis
ergebnis = i == j; // Wert von ergebnis: false, Konvertierung schlägt fehl
j = "5"
ergebnis = i == j; // Wert von ergebnis: false, Konvertierung funktioniert
                   // zwar, aber Vergleich ergibt false
j = "3";
ergebnis = i == j; // Wert von ergebnis: true
```

Test auf Ungleichheit

Der Test auf Ungleichheit wir durch den *!=* Operator durchgeführt. Der *!=* Operator gibt *true* zurück, wenn der == Operator *false* ergibt und umgekehrt.

Test auf kleiner, kleiner gleich, größer, größer gleich

Die Operatoren für diese Tests sind *<, <=, >* und >=. Sie ergeben *true*, wenn der erste Operand *<, <=, >,* >= als der zweite Operand ist, *false* im anderen Fall.

Operanden können grundsätzlich von jedem Datentyp sein. Der Vergleich erfolgt aber immer auf der Basis von Strings oder Zahlen. Zahlen werden numerisch, Strings lexikalisch verglichen. Kann ein Operand nicht in eine Zahl oder einen String umgewandelt werden, ist das Ergebnis des Vergleichs *false*. Ist ein Operand eine Zahl und der andere ein String, versucht JavaScript den String in eine Zahl umzuwandeln. Ist die Umwandlung erfolgreich, wird ein numerischer Vergleich durchgeführt, andernfalls ist das Ergebnis automatisch *false*.

4.4.5 Logische Operatoren

Logische Operatoren werden hauptsächlich genutzt, um Vergleiche zu verketten oder zu negieren.

Logisches UND

Der *&&* Operator führt die logische und-Verknüpfung durch. Er ergibt *true*, wenn seine beiden Operanden *true* ergeben, ansonsten *false*.

Logisches ODER

Der || Operator führt die logische oder-Verknüpfung durch. Er ergibt *true*, wenn einer seiner beiden Operanden *true* ergibt, ansonsten *false*.

Logische Negation

Der Negations-Operator *!* wird auf nur einen Operanden angewendet. Er ergibt *true*, wenn sein Operand *false* ist und umgekehrt.

Beispiel 4.13 Logische Operatoren

```
var i=5;
var j=6;
var k=6;
var l=5;
ergebnis = (i==j) && (j==k) // Wert von ergebnis: false, da i nicht j
ergebnis = (i==j) || (j==k) // Wert von ergebnis: true, da j gleich k
ergebnis = (i==l) && (j==k) // Wert von ergebnis: true
ergebnis = (i==l) || (j==k) // Wert von ergebnis: true
ergebnis = !(i==j)  // Wert von ergebnis: true
ergebnis = !(i!=j)  // Wert von ergebnis: false
```

4.4.6 Der tenäre Operator

Der tenäre Operator hat die Form ?:. Der tenäre Operator erhält 3 Operanden, der erste kommt vor das Fragezeichen, der zweite zwischen Fragezeichen und Doppelpunkt, der dritte hinter den Doppelpunkt. Der erste Operand muß einen bool'schen Wert ergeben, also *true* oder *false*. Er ist gewöhnlich ein Vergleich. Ergibt der erste Operand true, wird das Ergebnis des zweiten Operanden zurückgegeben, ansonsten das des dritten Operanden (vgl. Beispiel 4.14).

Beispiel 4.14 Der tenäre Operator

```
int i = 1;
int j = 2;
int k;
(i == j)? k = i : k = 6; // wenn i gleich j, dann k=i, sonst k=6
                         // in unseren Fall k = 6
```

5 Steuerungsstrukturen (Kontrollstrukturen)

5.1 Die if - Anweisung

Die *if*-Anweisung hat die Form:

```
if (logischer Ausdruck)
{
      anweisung1;
            :
            :
      anweisungN;
}
else
{
      anweisung1nachElse;
            :
            :
      anweisungNnachElse;
}
```

"Logischer Ausdruck" ist ein Ausdruck, der ein "logisches Ergebnis" erzeugt.

Ein logisches Ergebnis ist:

true (wahr)

false (falsch)

Logische Ausdrücke sind daher Vergleiche oder bool'sche Ausdrücke:

```
var nenner;
var x;
      :
if (nenner == 0)
      :
if (x > 3)
```

Ergibt der logische Ausdruck den Wert *true* werden die Anweisungen im

if - Teil (von nun an *if*-Block genannt),

anderenfalls (der logische Ausdruck ergibt *false)* werden die Anweisungen im

else - Teil (von nun an *else*-Block genannt) ausgeführt.

Regeln:

Die Zeile mit *if ...* darf nicht mit einem *;* abschließen.

Vor *else* darf kein*;* stehen.

Der *else*-Block ist optional .

Gibt es den *else*-Block nicht, ist dies gleichbedeutend mit: ansonsten tue nichts.

Wenn im *if*-Block nur eine Anweisung steht, können die geschweiften Klammern weggelassen werden.

Wenn im *else*-Block nur eine Anweisung steht, können die geschweiften Klammern weggelassen werden.

Merke: Dies ist extrem gefährlich!!!! Man sollte die geschweiften Klammern immer aufnehmen. Grund: Programmänderungen.

Merke: Einrücken ist wichtig. Einrücken erhöht die Lesbarkeit der Programme. Komplizierte Entscheidungsstrukturen sind ohne Einrückungen absolut unverständlich.

Das folgende Programm liest 2 reelle Zahlen ein, dividiert die erste eingelesene Zahl durch die zweite und gibt das Ergebnis aus. Wird als zweite Zahl 0 eingegeben, so wird die Fehlermeldung “Versuch durch 0 zu teilen” ausgegeben.

Beispiel 5.1 Ein Divisionsprogramm

```
<!-- Dateiname: division.html -->
<HTML>
<HEAD>
      <TITLE>
            Divisionsprogramm
      </TITLE>

</HEAD>

<BODY>
      <H2>
            Dies Programm dividiert die erste durch die
            zweite eingegebene Zahl.
      </H2>
      <P>
            Geben Sie in den Pop-Up-Fenstern die Zahlen ein, die Sie zu
            dividieren wünschen!
      </P>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/division.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: division.js

var zaehler;
var nenner;
var ergebnis;
var eingabe;
eingabe = prompt ("Z‰hler: ", "");
zaehler = parseFloat (eingabe);
```

```
eingabe = prompt ("Nenner: ", "");
nenner = parseFloat (eingabe);
if (nenner == 0)
{
      document.write ("Versuch durch 0 zu teilen!");
}
else
{
      ergebnis = zaehler / nenner;
      document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
}
```

In Beispiel 5.1 werden nach der Variablendeklaration (*var zaehler; var nenner; var ergebnis; var eingabe;*) vermittels *prompt* (*eingabe = prompt ("Zähler: ", ""); zaehler = parseFloat (eingabe); eingabe = prompt ("Nenner: ", ""); nenner = parseFloat (eingabe);)* Werte für den Zähler und den Nenner eingelesen und durch *parseFloat* in Zahlen umgewandelt. Dann wird mittels der *if*-Anweisung getestet, ob der Benutzer als Nenner 0 eingegeben hat (*if (nenner == 0)*). Beachten Sie, daß, wie in Kapitel 4.4 besprochen, Tests auf Gleichheit vermittels == erfolgen. In Abhängigkeit vom Ergebnis des Tests wird entweder "Versuch durch 0 zu teilen!" (der Test hat *true* ergeben) oder das berechnete Ergebnis (der Test hat *false* ergeben) durch *document.write* in das Browser-Fenster geschrieben. Die Anweisungen im *if*-Block und im *else*-Block der *if*-Anweisung werden in geschweifte Klammern *{}* eingeschlossen.

Die *if*-Anweisung und alle folgenden JavaScript Anweisungen werde ich (soweit sinnvoll) an zwei durchgängigen Beispielen veranschaulichen. Das erste dieser Beispiele (beginnend mit Beispiel 5.2) ist ein einfaches Taschenrechnerprogramm. Hier wird dem Benutzer ein Taschenrechner für die vier Grundrechenarten zur Verfügung gestellt. Zwei Fehler (Eingabe eines falschen Operators und Versuch durch 0 zu teilen) werden abgefangen.

Das zweite Beispiel (beginnend mit Beispiel 5.3) stellt ein stellt ein Umrechnungsprogramm zwischen Euro und DM zur Verfügung. Dies könnte z.B. Teil eines Internetauftritts einer Bank sein.

Beispiel 5.2 Ein kleines Taschenrechnerprogramm

```
<!-- Dateiname: taschenrechner.html -->

<HTML>
<HEAD>
      <TITLE>
            Taschenrechner
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm stellt Taschenrechner-Funktionalit&auml;t
            zur Verf&uuml;gung!.
      </H2>
      <P>
```

```
            Geben Sie in den Pop-Up-Fenstern die Zahlen und den
            Operator ein, mit denen Sie zu rechnen w&uuml;nschen!
      </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/taschenrechner.js">
      </SCRIPT>
      <P>
            Ich hoffe, dieses Programm hat Ihnen gefallen!
      </P>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: taschenrechner.js

var ersteEingabe;
var zweiteEingabe;
var ergebnis;
var eingabe;
var operator;
var richtigerOperator;

eingabe = prompt ("Erste Eingabe: ", "");
ersteEingabe = parseFloat (eingabe);
operator = prompt ("Operator: ", "");
eingabe = prompt ("Zweite Eingabe: ", "");
zweiteEingabe = parseFloat (eingabe);
richtigerOperator = false;

if (operator == "+")
{
      ergebnis = ersteEingabe + zweiteEingabe;
      richtigerOperator = true;
      document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
}

if (operator == "-")
{
      ergebnis = ersteEingabe - zweiteEingabe;
      richtigerOperator = true;
      document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
}

if (operator == "*")
{
      ergebnis = ersteEingabe * zweiteEingabe;
      richtigerOperator = true;
      document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
}

if (operator == "/")
{
      if (zweiteEingabe == 0)
```

```
        {
                richtigerOperator = true;
                document.write ("<B> <I> Versuch durch 0 zu teilen! </B </I>");
        }
        else
        {
                ergebnis = ersteEingabe / zweiteEingabe;
                richtigerOperator = true;
                document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
        }
}

if (!richtigerOperator)
{
        document.write ("<B> <I> Sie haben einen falschen Operator eingegeben
</B </I><BR>");
}
```

In Beispiel 5.2 werden zunächst die notwendigen Variablen deklariert. Die Variable *richtigerOperator* wird mit *false* initialisiert und dient dazu, festzustellen, ob die Benutzerin einen zugelassenen Operator eingegeben hat. Über *prompt*-Fenster werden sodann die Operanden und der Operator eingelesen. Es folgen vier *if*-Anweisungen. Die *if*-Anweisungen bestehen jeweils nur aus einem *if*-Block. Der jeweilige *if*-Block wird ausgeführt, wenn der Benutzer den zugehörigen Operator eingegeben hat (z.B. *if (operator == "+")*). In den Anweisungen des *if*-Blocks wird die Berechnung durchgeführt, das Ergebnis mittels *document.write* in den Browser geschrieben und der Wert der logischen Variable *richtigerOperator* auf *true* geändert. Das Programm schließt mit einer *if*-Anweisung, in der der Wert der logischen Variable *richtigerOperator* getestet wird. Ist der Wert von *richtigerOperator false* (*!richtigerOperator* mithin *true*) wurde der Wert von *richtigerOperator* im Programmverlauf nicht geändert. Dies bedeutet, daß keine der vorherigen *if*-Blöcke durchgeführt wurde. Der Benutzer hat also einen von unserem Programm nicht unterstützten Operator eingegeben. Dies wird ihm im Browser mitgeteilt.

Beispiel 5.3 Euro-DM Umrechnung

```
<!-- Dateiname: euroDM.html -->

<HTML>
<HEAD>
        <TITLE>
                Umrechnung von DM in Euro oder von Euro in DM
        </TITLE>
</HEAD>

<BODY>
        <H2>
                Diese Internet-Anwendung ist ein Service der Internet-Bank.
                Sie erlaubt Ihnen, DM-Betr&auml;ge in Euro oder
                Euro-Betr&auml;ge in DM umzurechnen.
        </H2>
        <P>
                Geben Sie dazu im ersten Pop-Up-Fenster die Zielw&auml;rung an!
                Im zweiten Pop-Up-Fenster geben Sie sodann den Betrag an, der
                umgerechnet werden soll. Das Ergebnis wird Ihnen im Browser-Fenster
```

```
            angezeigt.
      </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/euroDM.js">
      </SCRIPT>
      <P>
            Die Internet-Bank hofft, Ihnen mit dieser Dienstleistung
            geholfen zu haben.
      </P>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: euroDM.js

var zielWaehrung;
var quellWaehrung;
var umrechnungsbetrag;
var ergebnis;
var umrechnungsKurs = 1.996;

zielWaehrung = prompt("Zielw‰hrung: ", "");
if ((zielWaehrung == "DM") || (zielWaehrung == "dm"))
{
      umrechnungsbetrag = prompt("Bitte geben Sie den \n" +
                              "umzurechnenden Euro-Betrag ein!", "");
      ergebnis = parseFloat(umrechnungsbetrag) * umrechnungsKurs;
      document.write(umrechnungsbetrag + " Euro ergibt: "
                        + ergebnis + " DM! <BR>");
}
else
{
      if ((zielWaehrung == "Euro") || (zielWaehrung == "euro"))
      {
            umrechnungsbetrag = prompt("Bitte geben Sie den \n" +
                              "umzurechnenden DM-Betrag ein!", "");
            ergebnis = parseFloat(umrechnungsbetrag) *
                              (1/umrechnungsKurs);
            document.write(umrechnungsbetrag + " DM ergibt: "
                        + ergebnis + " Euro! <BR>");
      }
      else
      {
            document.write("Sie haben eine falsche Zielw&auml;rung" +
                        " angegeben. Erlaubt sind DM und Euro!");
      }
}
```

In Beispiel 5.3 wird nach den Variablendeklarationen zunächst die Währung in die umgerechnet werden soll abgefragt. In der Bedingung der ersten *if*-Anweisung wird getestet, ob es sich um DM handelt. Hier sind die Schreibweisen DM und dm zugelassen. Ist dies der Fall, wird der *if*-Block durchgeführt. Hier fragen wir den Betrag ab, rechnen- von DM in Euros um und geben das Ergebnis im Browserfenster aus.

Ist dies nicht der Fall, wird der *else*-Block durchgeführt. Als erstes fragen wir in einer weiteren *if*-Anweisung ab, ob Euro als Zielwährung angegeben wurde. Hier sind die Schreibweisen Euro und euro erlaubt. Ist dies der Fall wird der *if*-Block der zweiten *if*-Anweisung ausgeführt. Wir fragen den Umrechnungsbetrag ab, rechnen von DM in Euro um und geben das Ergebnis im Browser aus. Ist dies nicht der Fall, hat die Benutzerin weder Euro noch DM als Zielwährung eingegeben. Durch den *else*-Teil der zweiten if-Anweisung wird eine diesbezügliche Fehlermeldung in das Browser-Fenster geschrieben.

5.2 Der Switch

Die switch-Anweisung hat die Form:

```
switch (selector)
{
      case marke1 :
                  anweisung1.1;
                        :
                  anweisung1.N;
                  break;
      case marke2 :
                  anweisung2.1;
                        :
                  anweisung2.N;
                  break;
      case markeN :
                  anweisungN.1;
                        :
                  anweisungN.N;
                  break;
      default :   anweisungD.1;
                        :
                  anweisungD.N;
                  break;
};
```

Hierbei ist *selektor* ein Ausdruck, der einen String, eine Zahl oder einen Boolean ergibt. *marke1* bis *markeN* sind konstante Ausdrücke eines der oben angeführten Typen. Dabei kann *marke1* durchaus ein String sein und *marke2* eine Zahl. Ein konstanter Ausdruck zeichnet sich dadurch aus, daß er, wenn der JavaScript-Code in den Browser geladen wird, bestimmt werden kann und danach seinen Wert nicht mehr ändert. Variablen sind daher als Marken nicht zugelassen (sehr wohl natürlich als *selektor*). Die Anweisungen, die einer Marke folgen, werden ausgeführt, wenn der Wert des Selektors der Wert der Marke ist. Danach werden alle Anweisungen der Marken unterhalb der gefundenen Marke durchgeführt. Dies wird selten gewünscht. Die letzte Anweisungen hinter einer Marke ist daher `break`. `break` bewirkt, daß die Programmausführung hinter dem Switch fortgesetzt wird.

Ist der Wert des selektors in den Konstanten keiner Marke enthalten, wird der `default`-Zweig ausgeführt. Fehlt der `default`-Zweig, wird die `switch`-Anweisung ignoriert.

Die Zeile, die das Schlüsselwort *switch* enthält, wird nicht mit einem *;* abgeschlossen.

Hinter einer Marke folgt ein *:*.

Beispiel 5.4 zeigt das Taschenrechner-Programm mit einem Switch realisiert.

Beispiel 5.4 Ein Switch

```
<!-- Dateiname: taschenrechnerMitSwitch.html -->

<HTML>
<HEAD>
      <TITLE>
            Taschenrechner
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm stellt Taschenrechner-Funktionalit&auml;t
            zur Verf&uuml;gung!.
      </H2>
      <P>
            Geben Sie in den Pop-Up-Fenstern die Zahlen und den
            Operator ein, mit denen Sie zu rechnen w&uuml;nschen!
      </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/taschenrechnerMitSwitch.js">
      </SCRIPT>
      <P>
            Ich hoffe, dieses Programm hat Ihnen gefallen!
      </P>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: taschenrechnerMitSwitch.js

var ersteEingabe;
var zweiteEingabe;
var ergebnis;
var eingabe;
var operator;

eingabe = prompt ("Erste Eingabe: ", "");
ersteEingabe = parseFloat (eingabe);
operator = prompt ("Operator: ", "");
eingabe = prompt ("Zweite Eingabe: ", "");
zweiteEingabe = parseFloat (eingabe);

switch (operator)
{
      case "+":
            ergebnis = ersteEingabe + zweiteEingabe;
            document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
            break;
```

```
        case "-":
                ergebnis = ersteEingabe - zweiteEingabe;
                document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
                break;

        case "*":
                ergebnis = ersteEingabe * zweiteEingabe;
                document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
                break;

        case "/":
                if (zweiteEingabe == 0)
                {
                        document.write ("<B> <I> Versuch durch 0 zu teilen! </B </I>");
                }
                else
                {
                        ergebnis = ersteEingabe / zweiteEingabe;
                        document.write("<B> Das Ergebnis ist: " + ergebnis + "</B>");
                }
                break;
        default:
                document.write ("<B> <I> Sie haben einen falschen Operator eingegeben </B </
I><BR>");

}
```

5.3 Die while-Anweisung

Die *`while`*-Schleife hat die Form

```
while (logischer Ausdruck)
{
        Anweisung1;
                :
                :
        AnweisungN;
}
```

Wirkung:

Wenn das Programm auf die *`while`*-Schleife trifft, wird zunächst *`logischer Ausdruck`* ausgewertet. Ergibt *`logischer Ausdruck`* den Wert *`false`* wird die gesamte Schleife ignoriert (nicht ausgeführt). Ergibt *`logischer Ausdruck`* den Wert *`true`* werden die Anweisungen zwischen den geschweiften Klammern ausgeführt (Eintritt in die Schleife).

Immer dann, wenn das Programm auf die schließende geschweifte Klammer der Schleife trifft, wird *`logischer Ausdruck`* erneut überprüft. Ergibt *logischer Ausdruck* *`true`* wird die Schleife erneut ausgeführt. Ergibt *logischer Ausdruck* *`false`* wird die Schleife abgebrochen (das bedeutet, das Programm setzt mit der auf die schließende geschweifte Klammer der *`while`*-Anweisung folgenden Anweisung fort.

Folgerungen:

Ist `logischer Ausdruck` bei der ersten Auswertung `false` werden die Anweisungen der Schleife nie ausgeführt (abweisende Schleife).

Wird *logischer Ausdruck* nie `false` wird die Schleife nie abgebrochen (Endlosschleife). Die Anweisungen der Schleife werden unendlich oft wiederholt (dies ist selten vom Programmierer so beabsichtigt).

Der Wert von *logischer Ausdruck* muß also innerhalb der Schleife geändert werden (wenn man keine Endlosschleife zu programmieren beabsichtigt).

Regeln:

Hinter der Zeile `while` (*logischerAusdruck*) darf kein `;` stehen.

Jede Anweisung in der Schleife kann mit einem `;` abgeschlossen werden.

Anmerkung

Besteht die Schleife nur aus einer Anweisung, können die geschweiften Klammern weggelassen werden.

Merke: Dies ist extrem gefährlich und kann leicht zu logischen Fehlern führen. (vgl. Anmerkung zur *if*-Anweisung).

Unser Taschenrechnerprogramm soll so geändert werden, daß das Programm erst dann abbricht, wenn der Benutzer als Operator ein "b" eingibt. Die Eingabe zweier reeller Zahlen ist dann zwar immer noch notwendig, wird aber ignoriert.

Beispiel 5.5 Taschenrechner mit `while`

```
<! Dateiname: taschenrechnerMitSwitchUndWhile.html>
<HTML>
<HEAD>
      <TITLE>
            Taschenrechner
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Dies Programm stellt Taschenrechner-Funktionalit&auml;t
            zur Verf&uuml;gung!.
      </H2>
      <P>
            Geben Sie in den Pop-Up-Fenstern die Zahlen und den
            Operator ein, mit denen Sie zu rechnen w&uuml;nschen! Geben Sie
            als Operator b ein, um das Programm zu beenden!
      </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/taschenrechnerMitSwitchUndWhile.js">
      </SCRIPT>
      <P>
            Ich hoffe, dieses Programm hat Ihnen gefallen!
      </P>
</BODY>
```

```
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: taschenrechnerMitSwitchUndWhile.js

var ersteEingabe;
var zweiteEingabe;
var ergebnis;
var eingabe;
var operator;

eingabe = prompt ("Erste Eingabe: ", "");
ersteEingabe = parseFloat (eingabe);
operator = prompt ("Operator: ", "");
eingabe = prompt ("Zweite Eingabe: ", "");
zweiteEingabe = parseFloat (eingabe);
while (operator != "b")
{
      switch (operator)
      {
            case "+":
                  ergebnis = ersteEingabe + zweiteEingabe;
                  document.write("<B> Das Ergebnis ist: " +
                                    ergebnis + "</B><BR>");
                  break;

            case "-":
                  ergebnis = ersteEingabe - zweiteEingabe;
                  document.write("<B> Das Ergebnis ist: " +
                                    ergebnis + "</B><BR>");
                  break;

            case "*":
                  ergebnis = ersteEingabe * zweiteEingabe;
                  document.write("<B> Das Ergebnis ist: " +
                                    ergebnis + "</B><BR>");
                  break;

            case "/":
                  if (zweiteEingabe == 0)
                  {
                        document.write ("<B> <I> Versuch durch 0" +
                              " zu teilen! </B </I><BR>");
                  }
                  else
                  {
                        ergebnis = ersteEingabe / zweiteEingabe;
                        document.write("<B> Das Ergebnis ist: " +
                                    ergebnis + "</B><BR>");
                  }
                  break;
            default:
                  document.write ("<B> <I> Sie haben einen " +
                              "falschen Operator eingegeben </B </I><BR>");
```

```
        }
        eingabe = prompt ("Erste Eingabe: ", "");
        ersteEingabe = parseFloat (eingabe);
        operator = prompt ("Operator: ", "");
        eingabe = prompt ("Zweite Eingabe: ", "");
        zweiteEingabe = parseFloat (eingabe);
}
```

5.4 Die do-while-Schleife

Die `do-while`-Schleife hat die Form:

```
do
{
        anweisung1;
                :
                :
        anweisungN;
} while (logischer Ausdruck);
```

Wirkung:

Wenn das Programm auf die *`while`*-Anweisung trifft, wird `logischer Ausdruck` ausgewertet.

Wenn *logischer Ausdruck* *`false`* ist, passiert nichts weiter, das Programm fährt einfach mit der auf *`while`* folgenden Anweisung fort, als hätte es *`do ... while`* nie gegeben.

Ist *logischer Ausdruck* jedoch *`true`* werden die zwischen *`do`* und *`while`* stehenden Anweisungen erneut durchgeführt. Jedesmal, wenn das Programm im weiteren Verlauf auf *`while`* trifft, wird logischer Ausdruck erneut ausgewertet. Ist logischer Ausdruck *`true`* wird die Schleife erneut ausgeführt, andernfalls (*`false`*) verlassen.

Folgerungen:

Die *`do ... while`*-Schleife wird mindestens einmal durchlaufen (logischer Ausdruck wird am Ende der Schleife überprüft). Dies ist der große Unterschied zur *`while`*-Schleife.

Wird *logischer Ausdruck* nie *false* haben wir wieder eine Endlosschleife. Will man das vermeiden muß der Wert von *logischer Ausdruck* in der *do ... while*-Schleife geändert werden.

Regeln:

Hinter *`do`* darf kein *`;`* stehen.

Hinter *`while`* `logischer Ausdruck` darf ein *`;`* stehen.

Hinter jeder Anweisung in der *`do ... while`*-Schleife darf ein *`;`* stehen.

5.5 Die for-Schleife

Die for-Schleife hat die Form:

```
for (Initialisierungsanweisung; Bedingung; Änderungsanweisung)
{
      anweisung1;
      ...
      anweisungN
}
```

Dies wird anhand des Fakultätenbeispiels aus der Einleitung erklärt.

Beispiel 5.6 Die *for*-Schleife (das Fakultätenprogramm erklärt).

```
<!-- Dateiname: fakultaeten2.html -->
<HTML>
<HEAD>
      <TITLE>
            Fakult&auml;ten
      </TITLE>

</HEAD>

<BODY>
      <H2>
            Dies Programm berechnet Fakult&auml;ten.
      </H2>
      <P>
            Geben Sie unten die Zahl ein, bis zu der Sie die
            Fakult&auml;ten berechnet haben m&ouml;chten!
      </P>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/fakultaet.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
/// Dieses Java-Script-Programm berechnet Fakultaeten
// Dateiname fakultaet.js

var eingabe;
var ende;
var i;
var fact;
eingabe = prompt ("Zu berechnende Fakultaet:", "");
ende = parseInt (eingabe);
fact = 1;
for(i=1; i<= ende; i++)
{
      fact = fact * i;
```

```
      document.write(i + "! = " + fact + "<BR>");
}
```

Zunächst lesen wir mit *prompt* die Zahl ein, bis zu der die Fakultäten berechnet werden sollen, wandeln den von *prompt* eingelesenen String in einen Integer um und besetzen das Ergebnis der Fakultätenberechnung mit eins vor. Dann startet die *for*-Schleife.

Die Integer Variable i wird zu Beginn der Schleife auf 1 gesetzt (Initialisierungsanweisung, *i=1*). Danach wird geprüft, ob der Wert der Variable i kleiner gleich der eingebenen Zahl ist (Bedingung, *i*<= *ende*) . Wir nehmen jetzt an, der Benutzer hat 4 eingegeben. Dann ist dies beim Start der Schleife der Fall. Die Änderungsanweisung *(i*++) wird beim Eintritt in die Schleife nicht ausgeführt.

Daraufhin wird die `for`-Schleife durchlaufen. Nach dem Durchlauf wird die Änderungsanweisung durchgeführt (*i*++). Nun wird wieder geprüft, ob die Bedingung immer noch erfüllt ist (*i*<= *ende*). Ist sie erfüllt, wird die Schleife ein weiteres Mal durchlaufen.

Die Initialisierungsanweisung wird nun nicht mehr durchgeführt, sie wird ein einziges Mal beim Eintritt in die Schleife durchgeführt. Jeweils nach Durchlauf der Schleife wird die Änderungsanweisung ausgeführt.

Die Schleife endet, wenn die Bedingung nach Durchführung der Änderungsanweisung nicht mehr erfüllt ist. In unserem Fall wird die `for`-Schleife also 4-mal (wir hatten angenommen, daß der Benutzer 4 eingegeben hat) durchlaufen.

Regeln:

Nach der *for*-Anweisung darf kein Semikolon stehen.

Initialisierungsanweisung, Bedingung und Änderungsanweisung werden durch Semikolons getrennt.

Es kann mehrere Initialisierungsanweisungen und Änderungsanweisungen in einer *for*-Schleife geben. Sie werden dann jeweils durch Kommata getrennt.

6 Funktionen

Funktionen werden durch das Schlüsselwort *function* definiert. Danach folgt der Name der Funktion. Die Übergabeparameter einer Funktion folgen in runden Klammern. Mehrere Übergabeparameter werden durch Kommata getrennt. Funktionen ohne Übergabeparameter erhalten in der Deklaration leere runde Klammern.

Der Rumpf der Funktion schließt sich (eingeschlossen in geschweifte Klammern) an.

Beispiel 6.1 Ausgabefunktion mit Zeilenvorschub

```
function schreibeInBrowser(text)
{
      document.write("<B>" + text + "</B><BR>");
}
```

Funktionen können Rückgabewerte haben. Als Rückgabewert ist jeder Daten- oder Objekttyp möglich. Der Wert, den eine Funktion zurückgibt, folgt nach dem Schlüsselwort *return*. Nach *return* kehrt die Funktion zurück, Code-Zeilen nach *return* werden nicht mehr ausgeführt.

Der Code

```
function getAlter()
{
      return alter;
      alter = alter + 2;
}
```

ist also Unsinn, da die Funktion *getAlter()* nach der Zeile *return alter* zurückkehrt. Die Zeile *alter = alter + 2* wird daher nie ausgeführt. Funktionen, die einen Wert zurückgeben, müssen mindestens[8] eine *return*-Anweisung erhalten.

Beispiel 6.2 Fakultätenberechnung mit Funktion

```
<!-- Dateiname: fakultaetenMitFunktion.html -->
<HTML>
<HEAD>
      <TITLE>
            Fakult&auml;ten
      </TITLE>

</HEAD>

<BODY>
      <H2>
            Dies Programm berechnet Fakult&auml;ten.
      </H2>
      <P>
            Geben Sie unten die Zahl ein, bis zu der Sie die
            Fakult&auml;ten berechnet haben m&ouml;chten!
      </P>
```

8.Überlegen Sie sich als Übungsaufgabe eine sinnvolle Methode mit 2 return-Anweisungen!

```
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/fakultaet.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: fakultaetenMitFunktion.js

function fakultaet(ende)
{
      if (ende <= 1)
      {
            return 1;
      }
      var fact;
      var i;
      fact = 1;
      for(i=1; i<= ende; i++)
      {
            fact = fact * i;
      }
      return fact;
}

function schreibeInBrowser(text)
{
      document.write("<B>" + text + "<BR>");
}

//Hauptprogramm

var eingabe;
var ende;
var fact;
eingabe = prompt ("Zu berechnende Fakult‰t:", "");
ende = parseInt (eingabe);
fact = fakultaet(ende);
schreibeInBrowser("Die Fakult&auml;t zu " + ende + " ist: " + fact);
```

Für Übergabeparameter gilt das in Kapitel 4.3 Gesagte analog. Werden primitive Datentypen an eine Funktion übergeben, so wirken sich in der Funktion durchgeführte Änderungen der übergebenen Variablen nicht auf die Variablen des Hauptprogramms aus. In der Funktion wird eine Kopie der Variable erzeugt. Diese wird geändert.

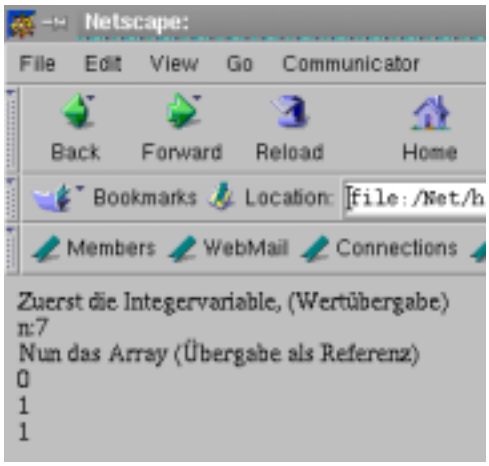
Wird hingegen eine an eine Funktion übergebene Variable eines Referenztyps geändert, so ist diese Änderung im Hauptprogramm sichtbar (vgl. Beispiel 6.3).

Beispiel 6.3 Variablenübergabe

```
<!-- Dateiname: uebergabe.html -->
```

```
<HTML>
<BODY>
<SCRIPT LANGUAGE = "JavaScript">
function aendere (m, n)
{
      m[2] = m[1];
      n = 9;
}


var n = 7;
var array = new Array();
array = [0,1,2];
aendere (array, n);
document.write("Zuerst die Integervariable, (Wert&uuml;bergabe)  <BR>");
document.write("n:" + n + "<BR>");
document.write("Nun das Array (&Uuml;bergabe als Referenz)  <BR>");
for (i = 0;i<=2;i++)
{
      document.write(array[i] + "<BR>");

}
</SCRIPT>
</BODY>
</HTML>
```

Beispiel 6.3 erzeugt die in Abbildung 6. 1 dargestellte Ausgabe:



bbildung 6. 1 Ausgabe von Beispiel 6.3

Es findet keinerlei Überprüfung statt, ob die Anzahl Parameter im Funktionsaufruf und die Anzahl Parameter, mit der die Funktion definiert wurde, übereinstimmen. Werden mehr Parameter als definiert übergeben, werden die überzähligen Parameter einfach ignoriert, werden weniger Parameter als erwartet übergeben, bleiben die restlichen Variablen undefiniert (vgl. Beispiel 6.4 und Beispiel 6.5).

Beispiel 6.4 Fakultätenberechnung mit Funktion und zuviel Parametern

```
<!-- Dateiname: fakultaetMitFunktion2Parameter.html -->
<HTML>
<HEAD>
<TITLE> FAKULT&Auml;TEN MIT FUNKTION und zwei Parametern</TITLE>
<SCRIPT LANGUAGE = "JavaScript">
	function fakultaet(ende)
	{
		if (ende <= 1)
		{
			return 1;
		}
		var fact;
		var i;
		fact = 1;
		for(i=1; i<= ende; i++)
		{
			fact = fact * i;
		}
		return fact;
	}

	function schreibeInBrowser(text)
	{
		document.write("<B>" + text + "<BR>");
	}

</SCRIPT>
</HEAD>

<BODY>
<H2>
	Dies Programm berechnet Fakult&auml;ten.
</H2>
	Geben Sie unten die Zahl ein, bis zu der Sie die
	Fakult&auml;ten berechnet haben m&ouml;chten!
<P>
<SCRIPT LANGUAGE = "JavaScript">
	var eingabe;
	var ende;
	var fact;
	eingabe = prompt ("Zu berechnende Fakultät:", "");
	ende = parseInt (eingabe);
	fact = fakultaet(ende, fact); // fact wird in der Funktion einfach ignoriert
	schreibeInBrowser("Die Fakult&auml;t zu " + ende + " ist: " + fact)
</SCRIPT>
</BODY>
</HTML>
```

Beispiel 6.5 Fakultätenberechnung mit Funktion und zuwenig Parametern

```
<!-- Dateiname: fakultaetMitFunktion0Parameter.html -->
<HTML>
<HEAD>
<TITLE> FAKULT&Auml;TEN MIT FUNKTION und null Parametern</TITLE>
<SCRIPT LANGUAGE = "JavaScript">
	function fakultaet(ende)
	{
		if (ende <= 1)
```

```
            {
                  return 1;
            }
            var fact;
            var i;
            fact = 1;
            for(i=1; i<= ende; i++)
            {
                  fact = fact * i;
            }
            return fact;
      }

      function schreibeInBrowser(text)
      {
            document.write("<B>" + text + "<BR>");
      }

</SCRIPT>
</HEAD>

<BODY>
<H2>
      Dies Programm berechnet Fakult&auml;ten.
</H2>
      Geben Sie unten die Zahl ein, bis zu der Sie die
      Fakult&auml;ten berechnet haben m&ouml;chten!
<P>
<SCRIPT LANGUAGE = "JavaScript">
      var eingabe;
      var ende;
      var fact;
      eingabe = prompt ("Zu berechnende Fakultät:", "");
      ende = parseInt (eingabe);
      fact = fakultaet(); // fuehrt zu einem Fehler fakultaet gibt 1
                          // zurueck, egal was der Benutzer eingegeben
                          // hat.
      schreibeInBrowser("Die Fakult&auml;t zu " + ende + " ist: " + fact)
</SCRIPT>
</BODY>
</HTML>
```

In Beispiel 6.4 und Beispiel 6.5 zeigt sich auch, daß Funktionen auch im <HEAD>-Teil eines html-Dokuments definiert werden können. Sie sind im <BODY> zugreifbar. Der Namensraum von JavaScript-Bezeichnern ist das gesamte Dokument.

Neben Variablen können Funktionen an Funktionen übergeben werden. Dies erlaubt es, sehr allgemeine und damit sehr mächtige Funktionen zu schreiben. Wir betrachten als Beispiel die *`sort()`*-Methode des Array-Objekts[9]. Wird diese Methode ohne Pa-

9.Funktionen heißen Methoden, wenn sie im Zusammenhang mit Objekten diskutiert werden. Das Arrays Objekte sind, wurde bereit in Kapitel 4.2 behauptet. Der Objekt-Begriff an sich wird in Kapitel 7 geklärt.

rameter aufgerufen, werden die Elemente des Arrays alphabetisch sortiert. Will man aber Zahlen ordnen, ist dies nicht so gut, weil 245 z.B. nach alphabetischer Ordnung kleiner ist als 7.

sort() akzeptiert aber als Parameter eine Vergleichs-Funktion. Die Vergleichs-Funktion ihrerseits muß 2 Parameter akzeptieren. Sie muß eine negative Zahl zurückgeben, wenn der erste Parameter nach ihren Kriterien kleiner ist als der zweite; 0, wenn beide gleich sind und eine positive Zahl, wenn der erste Parameter größer als der zweite ist. Beispiel 6.6 zeigt ein dies veranschaulichendes JavaScript-Beispiel.

Beispiel 6.6 Das sort()-Beispiel

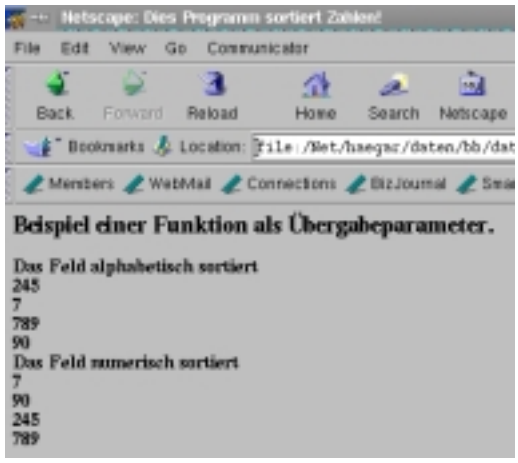
```
<!-- Dateiname: ordneZahlen.html -->
<HTML>
<HEAD>
      <TITLE>
            Dies Programm sortiert Zahlen!
      </TITLE>

</HEAD>

<BODY>
      <H2>
            Beispiel einer Funktion als &Uuml;bergabeparameter.
      </H2>


      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/ordneZahlen.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: ordneZahlen.js
// Dateiname: ordneZahlen.js
// Funktionsdeklarationen

function ordne (a, b)
{
      return (a - b);
}

function schreibeInBrowser(text)
{
      document.write("<B>" + text + "</B><BR>");
}


// Hauptprogramm

var meinFeld;
```

```
meinFeld = new Array (4);
meinFeld[0] = 245;
meinFeld[1] = 90;
meinFeld[2] = 789;
meinFeld[3] = 7;

meinFeld.sort();
schreibeInBrowser("Das Feld alphabetisch sortiert");
for (i=0 ; i <= meinFeld.length - 1; i++)
{
      schreibeInBrowser(meinFeld [i]);
}

meinFeld.sort(ordne);
schreibeInBrowser("Das Feld numerisch sortiert");
for (i=0 ; i <= meinFeld.length - 1; i++)
{
      schreibeInBrowser(meinFeld [i]);
}
```

Beispiel 6.6 erzeugt die in Abbildung 6. 2 dargestellte Ausgabe:



bbildung 6. 2 Ausgabe von Beispiel 6.6

Wie kann die in Beispiel 6.6 benutzte `sort()`-Funktion erkennen, ob sie mit einem Argument oder überhaupt ohne Argument aufgerufen wurde. Hierfür gibt es in JavaScript eine sehr interessante Möglichkeit. Funktionen sind in Wirklichkeit auch Objekte (der Objektbegriff wird in Kapitel 7 besprochen) und haben als solche Eigenschaften (properties).

Eine Eigenschaft einer Funktion ist das Array `arguments[]`. Es enthält alle Parameter, mit der die Funktion aufgerufen wurde. Da `arguments[]` selber wiederum ein Array ist, besitzt `arguments[]` (vgl. Kapitel 4.2) die Eigenschaft `length`. `length` ist die Anzahl der Elemente des Array und damit die Anzahl der Parameter mit der die Funktion aufgerufen wurde. Beispiel 6.7 und Beispiel 6.8 zeigen beispielhaft die Nutzung der `length`-Eigenschaft des Arrays `arguments[]` einer Funktion.

Beispiel 6.7 Fakultätenberechnung mit Funktion, zuwenig Parametern und Einsatz der *arguments[].length*-Eigenschaft

```
<!-- Dateiname: fakultaetUndFehlerabfang -->

<HTML>
<HEAD>
<TITLE>
      FAKULT&Auml;TEN MIT FUNKTION und null Parametern Fehler abgefangen
</TITLE>
<SCRIPT LANGUAGE = "JavaScript">

      function schreibeInBrowser(text)
      {
            document.write("<B>" + text + "<BR>");
      }

      function fakultaet(ende)
      {
            if (fakultaet.arguments.length != 1)
            {
                  schreibeInBrowser ("Funktion wurde mit falscher Anzahl" +
                                     "Argumente aufgerufen!");
                  return 0;
            }

            if (ende <= 1)
            {
                  return 1;
            }

            var fact;
            var i;
            fact = 1;
            for(i=1; i<= ende; i++)
            {
                  fact = fact * i;
            }
            return fact;
      }



</SCRIPT>
</HEAD>

<BODY>
<H2>
      Dies Programm berechnet Fakult&auml;ten.
</H2>
      Geben Sie unten die Zahl ein, bis zu der Sie die
      Fakult&auml;ten berechnet haben m&ouml;chten!
<P>
<SCRIPT LANGUAGE = "JavaScript">
      var eingabe;
      var ende;
      var fact = 0;
      eingabe = prompt ("Zu berechnende Fakultaet:", "");
```

```
      ende = parseInt (eingabe);
      fact = fakultaet(); // Fehler wird abgefangen, im Browser
                          // erscheint die Fehlermeldung der Funktion
      if (fact != 0)
      {
            schreibeInBrowser("Die Fakult&auml;t zu " + ende +
                              " ist: " + fact);
      }

</SCRIPT>
</BODY>
</HTML>
```

Beispiel 6.8 zeigt eine weitere schöne Anwendung der Möglichkeit, Funktionen mit einer variablen Anzahl Parameter schreiben zu können. Zugleich sehen wir, daß in eine html-Datei beliebig viele (in Beispiel 6.8 allerdings nur zwei) JavaScript-Code-Dateien eingebunden werden können.

Beispiel 6.8 zeigt eine ein Minimum beliebig vieler Zahlen berechnende Funktion. Die das Minimum berechnende Funktion laden wir im <HEAD>-Teil des html-Dokuments, den JavaScript-Code, der sie aufruft, im <BODY>.

Beispiel 6.8 Eine allgemeine Minimum-Funktion

```
<!-- Dateiname: minimum.html -->
<HTML>
<HEAD>
      <TITLE>
            Minimumberechnung
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/minimum.js">

      </SCRIPT>
</BODY>

</HEAD>

<BODY>
      <H2>
            Hier werden Minima beliebig vieler Parameter
            berechnet!
      </H2>
      <P>
            Geben Sie in den Pop-Up-Fenstern die Zahlen ein,
            derer Minima Sie ermittelt haben m&ouml;chten.
      </P>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/minimumAufruf.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörigen JavaScript-Dateien

```
// Dateiname: minimumAufruf.js

var eingabe1;
var eingabe2;
var eingabe3;
var min;

eingabe1 = prompt ("Erste Zahl:", "");
eingabe1 = parseInt (eingabe1);
eingabe2 = prompt ("Zweite Zahl:", "");
eingabe2 = parseInt (eingabe2);
min = minimum (eingabe1, eingabe2);
document.write ("Das Minimum ist  " + min + "<BR>");

eingabe1 = prompt ("Erste Zahl:", "");
eingabe1 = parseInt (eingabe1);
eingabe2 = prompt ("Zweite Zahl:", "");
eingabe2 = parseInt (eingabe2);
eingabe3 = prompt ("Zweite Zahl:", "");
eingabe3 = parseInt (eingabe3);
min = minimum (eingabe1, eingabe2, eingabe3);
document.write ("Das Minimum ist  " + min + "<BR>");

// Dateiname: minimum.js
// Dateiname: minimum.js

function minimum()
{
      if (minimum.arguments.length == 0)
      {
            document.write ("Funktion wurde mit falscher Anzahl" +
                              "Argumente aufgerufen!");
            return 0;
      }
      var i;
      var min;

      min = minimum.arguments[0];
      for (i = 1; i <= arguments.length - 1; i++)
      {
            if (minimum.arguments[i] < min)
            {
                  min = minimum.arguments[i];
            }
      }
      return min;
}
```

7 Der Objekt-Begriff in JavaScript

Objekte in JavaScript bestehen wie in jeder anderen objektorientierten Programmiersprache auch aus Daten und Funktionen. Die Daten können beliebigen Typs sein (auch wieder selbst Objekte). Die Daten werden Eigenschaften (Properties) des Objekts genannt, die Funktionen eines Objekts heißen Methoden.

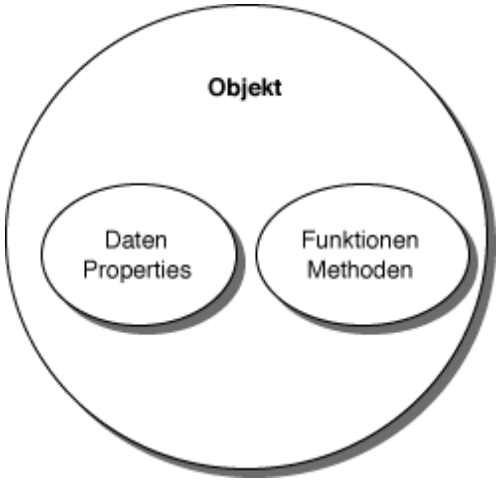


Abbildung 7. 1 Ein Objekt

7.1 Von JavaScript zur Verfügung gestellte Objekte

Das für den JavaScript-Entwickler mit Abstand wichtigste im Umgang mit JavaScript-Objekten ist die Nutzung der von JavaScript zur Verfügung gestellten Objekte. Hierbei handelt es sich im Einzelnen um:

- *Objekte, die das Entwickeln von JavaScript-Anwendungen erleichtern:* In JavaScript existiert beispielsweise ein Date-Objekt zum Arbeiten mit dem Datum oder ein String-Objekt zur Stringbearbeitung.
- *Interaktion mit dem Browser:* Jedes wichtige html-Element wird in JavaScript durch ein Objekt repräsentiert. So ist z.B. jedes html-Formular (<FORM>-Tag) mit einem JavaScript Objekt verbunden. Über die Methoden und Eigenschaften der den html-Elementen zugeordneten Objekte lassen sich html-Seiten dynamisch verändern. Eines dieser Objekte haben wir bereits kennengelernt. Das *document*-Objekt repräsentiert den Inhalt eines Browser-Fensters (oder Frames). Mittels der Methode *write()* des *document*-Objekt können wir von JavaScript aus in das Browser-Fenster schreiben. Das *document*-Objekt verfügt darüberhinaus über Properties wie *URL* (die URL der dargestellten Seite) oder *lastModified* (das Datum der letzten Änderung des im Browser-Fenster dargestellten Dokuments), die bei der html-Programmierung sehr sinnvoll einzusetzen sind.

Diese Objekte und ihr Einsatz in der praktischen JavaScript-Entwicklung werden in den folgenden Kapiteln eingehend besprochen.

7.2 Aufruf von Objekt-Methoden und Ansprechen von Objekt-Properties

Durch das bereits erlernte Arbeiten mit dem *document*-Objekt wissen wir auch, wie Methoden der Objekte aufgerufen werden. Dies geschieht durch den Objekt-Namen, einen Punkt (.) und den Namen der Methode, wie z.B. in

```
document.write ("Das Minimum ist " + min + "<BR>");
```

aus Beispiel 6.8.

Objekt-Properties werden auf identische Weise angesprochen. Auch hier wird der Name der Property mittels eines Punkts (.) an den Objektnamen angeschlossen. Auch dies ist uns bereits bekannt, da Arrays Objekte sind und wir bereits mit der *length*-Property von Arrays gearbeitet haben, wie in

```
for (i = 1; i <= arguments.length - 1; i++)
```

aus Beispiel 6.8.

7.3 Erstellung eigener Objekte

Die Programmierung eigener Objekte hat in Client-Side JavaScript meines Erachtens nicht die große Bedeutung, da in html-Dateien eingebettete Scripts meistens doch nicht so große Anwendungen sind, daß sie einer wohlüberlegten Objektstruktur bedürfen. Anders kann dies sein, wenn JavaScript serverseitig zur Implementierung größerer Anwendungen genutzt wird.

Serverseitiges JavaScript wird jedoch von den wenigsten http-Servern unterstützt und ist hier auch nicht Thema.

Daher werde ich dies Kapitel relativ kurz fassen. Auch sind die Beispiele dieses Kapitels nicht besonders praxisbezogen.

Um eigendefinierte Objekte zu erzeugen, müssen wir einen Konstruktor schreiben. Der Name des Konstruktors ist der Name der Klasse[10] unserer Objekte.

Im Konstruktor werden zunächst die Properties des Objekts definiert und mit Werten initialisiert (vgl. Beispiel 7.1).

Objekte werden mit dem Schlüsselwort *new* erzeugt. Darauf folgt der Konstruktor des Objekts (vgl. die Erzeugung von Arrays in Kapitel 4.2).

Beispiel 7.1 zeigt die Definition und den Umgang mit Objekten:

Beispiel 7.1 Das Objekt-Mensch-Beispiel (Beginn)

```
<! Dateiname: arbeiteMitMensch.html>
<HTML>
<HEAD>
      <TITLE>
            Objekte 1
```

10.Klassen sind Vorlagen nach denen Objekte erzeugt werden.

```
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/Mensch.js">

      </SCRIPT>
</HEAD>

<BODY>
      <H2>
            Properties von Mensch Objekten!
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/arbeiteMitMensch.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörigen JavaScript-Dateien

```
// Code um mit der Klasse Mensch zu arbeiten
// Dateiname: arbeiteMitMensch.js

ersterMensch = new Mensch(); // Default-Konstruktor
zweiterMensch = new Mensch("Bluemel", "Bernd"); //Konstruktor mit Wertangabe


document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.name + "<BR>");
document.write(ersterMensch.vorname + "<BR>");

document.write("Eigenschaften des zweiten Menschen <BR>");
document.write(zweiterMensch.vorname + "<BR>");
document.write(zweiterMensch.name + "<BR>");

// Objekt Mensch Implementierung
// Dateiname: Mensch.js

function Mensch()
{
      if (Mensch.arguments.length == 0)
      // keine Parameter (Default-Konstruktor)
      {
            this.name = "WasWeissIch";
            this.vorname = "WeissIchAuchNich";

      }

      if (Mensch.arguments.length == 2)
      // Alle Parameter angegeben
      {
            this.name = Mensch.arguments[0];
            this.vorname = Mensch.arguments[1];
      }
```

```
}
```

In Beispiel 7.1 laden wir die Implementierung der Klasse Mensch im <HEAD>-Teil der html-Datei. Diese Implementierung (in der Datei Mensch.js) beginnt mit der Zeile

```
function Mensch()
```

Hier beginnt der Konstruktor des Objekts. Dieser Konstruktor wird immer dann aufgerufen, wenn ein Objekt vom Typ Mensch erzeugt werden soll[11]. Wird der Konstruktor ohne Parameter aufgerufen, werden 2 Eigenschaften (Variablen, properties) des Objekts mit default-Werten belegt:

```
this.name = “WasWeissIch”;
this.vorname = “WeissIchAuchNich”;
```

Neu an der Funktion *Mensch()* ist das Schlüsselworts *this*. *this* ist einfach der Name des Objekts selber, dessen Properties *name* und *vorname* schließlich werden sollen. *this* wird innerhalb des Objekts für das Objekt selber verwendet. *this* zeigt der JavaScript-Laufzeit-Umgebung, daß es sich bei *function Mensch()* um einen Konstruktor handelt.

Die Verwendung von *this* innerhalb des Konstruktors zeigt auch sofort, daß *name* und *vorname* nur im Kontext eines Objekts benutzt werden können, da sie ansonsten nicht ansprechbar sind.

Wird der Konstruktor mit 2 Parametern aufgerufen, werden die Properties des Objekts mit den übergebenen Werten belegt:

```
this.name = Mensch.arguments[0];
this.vorname = Mensch.arguments[1];
```

Properties von Objekten müssen im Konstruktor mit Hilfe von *this* deklariert werden.

Unsere Klasse Mensch enthält nach dieser (ersten) Implementierung nur Daten. Methoden haben wir noch keine definiert.

Im <BODY>-Teil der html-Datei aus Beispiel 7.1 wird der JavaScript-Code, der Objekte der Mensch-Klasse benutzt, geladen (Datei arbeiteMitMensch.js).

Dort werden zunächst zwei Objekte vom Typ Mensch mit der Syntax:

```
ersterMensch = new Mensch(); // Konstruktor ohne Parameterübergabe
zweiterMensch = new Mensch("Bernd", "Bluemel");
                        //Konstruktor mit Wertangabe
```

erzeugt. Die Objekte haben die Namen *ersterMensch* und *zweiterMensch*. Jedes dieser Objekte verfügt über zwei Eigenschaften (*name* und *vorname*).

Die Eigenschaften der Objekts werden nun mit der Syntax:

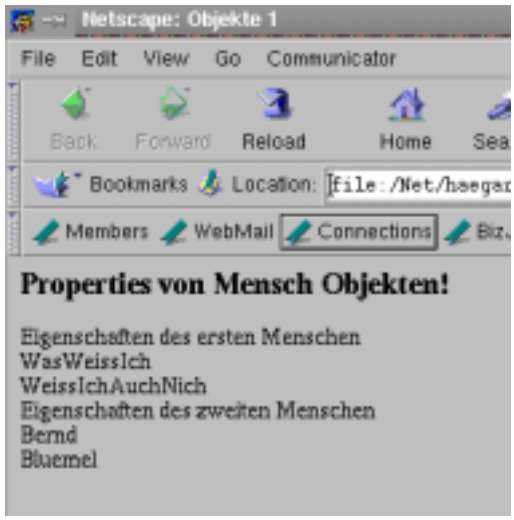
```
nameDesObjekts.nameDerEigenschaft
```

11.Wie dies geschieht (ein Objekt erzeugen) wird später besprochen.

angesprochen. Dies zeigt der weitere Code in Beispiel 7.1.

```
document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.name + "<BR>");
document.write(ersterMensch.vorname + "<BR>");

document.write("Eigenschaften des zweiten Menschen <BR>");
document.write(zweiterMensch.vorname + "<BR>");
document.write(zweiterMensch.name + "<BR>");
```

Beispiel 7.1 erzeugt die in Abbildung 7. 2 dargestellte Ausgabe:



bbildung 7. 2 Ausgabe von Beispiel 7.1

Nun wollen wir Methoden in unsere Mensch-Objekte aufnehmen. Beispiel 7.2 illustriert dies.

Beispiel 7.2 Das Objekt-Mensch-Beispiel mit Methoden

```
<! Dateiname: arbeiteMitMensch2.html>
<HTML>
<HEAD>
      <TITLE>
            Objekte 2
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/Mensch2.js">

      </SCRIPT>
</HEAD>
<BODY>
      <H2>
            Properties von Mensch Objekten!
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/arbeiteMitMensch2.js">
```

```
      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörigen JavaScript-Dateien

```
// Code um mit der Klasse Mensch zu arbeiten
// Dateiname: arbeiteMitMensch2.js

ersterMensch = new Mensch(); // Default-Konstruktor
zweiterMensch = new Mensch( "Bluemel", "Bernd"); //Konstruktor mit Wertangabe


document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");


document.write("Eigenschaften des zweiten Menschen <BR>");
document.write(zweiterMensch.getName() + "<BR>");
document.write(zweiterMensch.getVorname() + "<BR>");
document.write(zweiterMensch.getFullName() + "<BR>");

// Objekt Mensch Implementierung
// Dateiname: Mensch2.js

function Mensch()
{
      if (Mensch.arguments.length == 0)
      // keine Parameter (Default-Konstruktor)
      {
            this.name = "WasWeissIch";
            this.vorname = "WeissIchAuchNich";

      }

      if (Mensch.arguments.length == 2)
      // Alle Parameter angegeben
      {
            this.name = Mensch.arguments[0];
            this.vorname = Mensch.arguments[1];
      }
}

function Mensch_getName ()
{
      return this.name;
}
function Mensch_getVorname ()
{
      return this.vorname;
}
```

```
function Mensch_getFullName()
{
      return (this.vorname + " " + this.name);
}

function Mensch_setName (neuName)
{
      this.name = neuName;
}
function Mensch_setVorname (neuVorname)
{
      this.vorname = neuVorname;
}

Mensch.prototype.getName = Mensch_getName;
Mensch.prototype.getVorname = Mensch_getVorname;
Mensch.prototype.getFullName = Mensch_getFullName;
Mensch.prototype.setName = Mensch_setName;
Mensch.prototype.setVorname = Mensch_setVorname;
```

Die Erzeugung von Objekt-Methoden geschieht in zwei Schritten:

- Zunnächst definieren wir die Funktionen, die die Methoden des Objektes werden sollen. Dies sind die Code-Zeilen:

  ```
  function Mensch_getName ()
  {
        return this.name;
  }
  function Mensch_getVorname ()
  {
        return this.vorname;
  }

  function Mensch_getFullName()
  {
        return (this.vorname + " " + this.name);
  }

  function Mensch_setName (neuName)
  {
        this.name = neuName;
  }
  function Mensch_setVorname (neuVorname)
  {
        this.vorname = neuVorname;
  }
  ```

  Diese Funktionen unterscheiden sich von unseren bisher benutzten Funktionen dadurch, daß sie das Schlüsselwort *this* benutzen können und damit auf die Variablen des Objekts Zugriff haben.

- Danach werden die so definierten Funktionen zu Methoden des Objekts gemacht:

```
Mensch.prototype.getName = Mensch_getName;
Mensch.prototype.getVorname = Mensch_getVorname;
Mensch.prototype.getFullName = Mensch_getFullName;
Mensch.prototype.setName = Mensch_setName;
Mensch.prototype.setVorname = Mensch_setVorname;
```

Das Schlüsselwort *prototype* sagt der JavaScript-Umgebung, daß für alle Objekte vom Typ Mensch nur eine Kopie der Anweisungen der jeweiligen Funktion in den Hauptspeicher geladen werden muß.

Unsere Objekte können die so definierten Methoden dann mittels der uns ja schon bekannten Syntax

```
nameDesObjekts.nameDerMethode()
```

aufrufen. Dies zeigt der weitere Code in Beispiel 7.2.

```
document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");

document.write("Eigenschaften des zweiten Menschen <BR>");
document.write(zweiterMensch.getName() + "<BR>");
document.write(zweiterMensch.getVorname() + "<BR>");
document.write(zweiterMensch.getFullName() + "<BR>");
```

Beispiel 7.2 erzeugt die in Abbildung 7. 3 dargestellte Ausgabe:

bbildung 7. 3 Ausgabe von Beispiel 7.2

In JavaScript ist es leider nicht möglich, private Variablen in Objekten anzulegen (Variablen, die nicht von außerhalb des Objekts zugreifbar sind).

Dies bedeutet, daß die Definition der Zugriffsmethoden auf die Eigenschaften des Objekts in der “Klasse” Mensch eigentlich nicht notwendig sind. Wir können die Eigenschaften eines jeden Objekts (wie ja auch in Beispiel 7.1 geschehen) immer über ihren Namen ansprechen. Zugriffsmethoden sind aber dennoch in vielen Fällen sinnvoll, da

sie Fehlersituationen vermeiden helfen. Dies zeigt folgende Erweiterung von Beispiel 7.1:[12]

Beispiel 7.3 Änderung der Eigenschaften des Mensch-Objektes durch Direktzugriff

```
<! Dateiname: arbeiteMitMensch3.html>
<HTML>
<HEAD>
      <TITLE>
            Objekte 2
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/Mensch.js">

      </SCRIPT>
</BODY>

</HEAD>

<BODY>
      <H2>
            Properties von Mensch Objekten!
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/arbeiteMitMensch3.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige (geändert gegenüber Beispiel 7.1) JavaScript-Datei

```
// Code um mit der Klasse Mensch zu arbeiten
// Dateiname: arbeiteMitMensch3.js

ersterMensch = new Mensch(); // Default-Konstruktor


document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.name + "<BR>");
document.write(ersterMensch.vorname + "<BR>");

ersterMensch.nane = "Bluemel";
ersterMensch.vorname= "Bernd";

document.write("Der erste Mensch geändert: <BR>");
document.write(ersterMensch.name + "<BR>");
document.write(ersterMensch.vorname + "<BR>");
```

Beispiel 7.3 erzeugt die in Abbildung 7. 4 dargestellte Ausgabe:

12.Geändert wurden nur arbeiteMitMensch.js und arbeiteMitMensch.html, sodaß auch nur diese Datei neu aufgeführt werden.

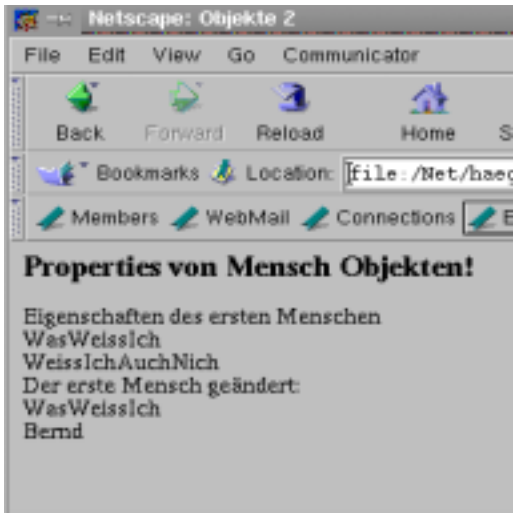


Abbildung 7. 4 Ausgabe von Beispiel 7.3

Wie erkennbar, ist der Wert der Eigenschaft *name* des Objektes Mensch nicht auf "Bluemel" geändert, sondern immer noch WasWeissIch. Dies ist auch nicht verwunderlich, da im Programm-Code ein Schreibfehler ist (`ersterMensch.nane = Bluemel`). Andererseits erfolgt auch keine Fehlermeldung. Dies liegt einfach daran, daß JavaScript annimmt, wir wollten eine weitere Eigenschaft des Objektes `meinMensch` deklarieren mit dem Namen `nane`. `ersterMensch` hat danach3 Eigenschaften (name, vorname und nane).

Dies ist übrigens eine generelle Eigenschaft von Objekten in JavaScript. Neue Objekt-Eigenschaften (properties) können wir einfach dadurch deklarieren, daß wir Ihnen irgendwo im Programmcode einen Namen geben und einen Wert zuweisen.

Die Fehlermöglichkeit von Beispiel 7.3 können wir allerdings durch die Benutzung von Zugriffsfunktionen eliminieren, wie Beispiel 7.4 zeigt:

Beispiel 7.4 Fehlervermeidung durch Zugriffsfunktionen

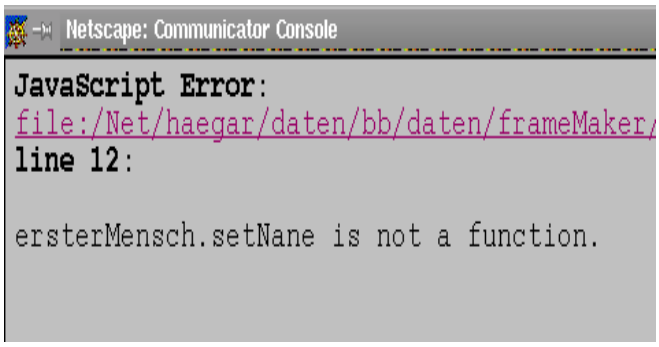
```
<! Dateiname: arbeiteMitMensch4.html>
<HTML>
<HEAD>
      <TITLE>
            Objekte 4
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/Mensch2.js">

      </SCRIPT>
</HEAD>
<BODY>
      <H2>
            Properties von Mensch Objekten!
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
```

```
                SRC="./javaScript/arbeiteMitMensch4.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige (geändert gegenüber Beispiel 7.2) JavaScript-Datei

```
// Code um mit der Klasse Mensch zu arbeiten
// Dateiname: arbeiteMitMensch4.js

ersterMensch = new Mensch(); // Default-Konstruktor


document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");

ersterMensch.setNane ("Bluemel");
ersterMensch.vorName= ("Bernd");

document.write("Der erste Mensch ge&auml;ndert: <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");
```

Diese Variante funktioniert zwar auch nicht (wir haben uns ja wieder vertippt, *setNane()* anstelle von *setName()*), erzeugt aber folgende Fehlerausgabe:



bbildung 7. 5 Ausgabe von Beispiel 7.4

Objekte und Klassen in JavaScript haben eine weitere schöne Eigenschaft. Über das *prototype*-Schlüsselwort können den Objekten weitere Methoden "on the fly" hinzugefügt werden. Diese Methoden stehen dann auch Objekten zur Verfügung, die vor der Definition der Methode erzeugt wurden. Dies wird in Beispiel 7.5 erläutert:

Beispiel 7.5 Hinzufügen weiterer Methoden zur Objektdefinition

```
<! Dateiname: arbeiteMitMensch5.html>
<HTML>
<HEAD>
      <TITLE>
```

```
            Objekte 5
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/Mensch2.js">

      </SCRIPT>
</BODY>

</HEAD>

<BODY>
      <H2>
            Properties von Mensch Objekten!
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/arbeiteMitMensch5.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige (geändert gegenüber Beispiel 7.2) JavaScript-Datei

```
// Code um mit der Klasse Mensch zu arbeiten
// Dateiname: arbeiteMitMensch4.js

ersterMensch = new Mensch(); // Default-Konstruktor


document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");

function Mensch_setFullName(name, vorname)
{
      this.name=name;
      this.vorname=vorname;
}

Mensch.prototype.setFullName = Mensch_setFullName;

ersterMensch.setFullName("Bluemel", "Bernd");

document.write("Der erste Mensch geîndert: <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");
```

Beispiel 7.5 erzeugt die in Abbildung 7. 6 dargestellte Ausgabe:

Wir sehen hier, daß die Methode *setFullName* erst nach der Erstellung des Objektes *ersterMensch* der Klassendefinition hinzugefügt wurde (und das noch außerhalb der Datei, in der die Klasse definiert wurde). Dies spielt aber in JavaScript keine Rolle. Me-

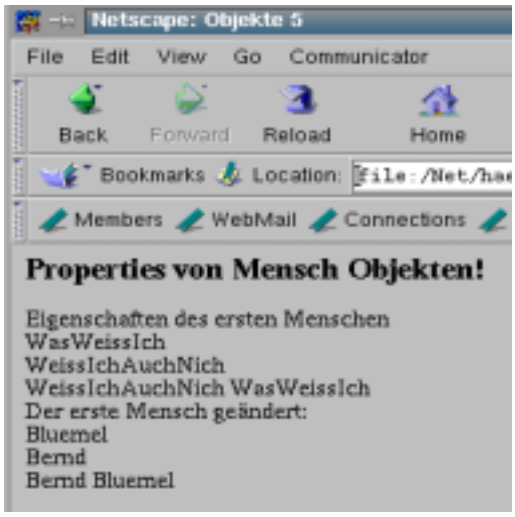


Abbildung 7. 6 Ausgabe von Beispiel 7.5

thoden können den Klassen an jeder beliebigen Stelle hinzugefügt werden, Klassen können an jeder beliebigen Stelle deklariert werden.

7.4 Erweiterung JavaScript interner Klassen

Das Prototyp-Konzept von JavaScript beschränkt das nachträgliche Hinzufügen von Methoden nicht auf selbstdefinierte Objekte und Klassen (vgl Beispiel 7.5). Auch den internen JavaScript-Klassen können Methoden hinzugefügt werden. Ich zeige dies beispielhaft an der String-Klasse. Strings sind, wie wir ja bereits festgestellt haben, Zwitter zwischen primitiven Datentypen und Objekten. Wenn Strings als Objekte benutzt werden, können sie alle Methoden, die für die String-Klasse definiert sind, ausführen.

In unserem Beispiel wollen wir nun der String-Klasse eine Methode *startsWith* hinzufügen, deren Sinn es ist zu überprüfen, ob ein String mit einem bestimmten Buchstaben beginnt.

Beispiel 7.6 Hinzufügen der startWith-Methode zur String-Klasse

```
<! Dateiname: stringErweiterung.html>
<HTML>
<HEAD>
      <TITLE>
            String Erweiterung
      </TITLE>

</HEAD>

<BODY>
      <H2>
            Nutzung des prototype-Konzepts zur Erweiterung von
            internen Klassen
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
```

```
                    SRC="./javaScript/stringErweiterung.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: stringErweiterung.js//

var string1 = "abc";
function String_startsWith(a)
{
      if(this.charAt(0) == a)
      {
            return true;
      }
      else
      {
            return false;
      }
}

String.prototype.startsWith = String_startsWith;

var string2="cde";

document.write(string1.startsWith("a") + "<BR>");
document.write(string2.startsWith("a") + "<BR>");
document.write(string1.startsWith("c") + "<BR>");
document.write(string2.startsWith("c") + "<BR>");
```

Beispiel 7.6 erzeugt die in Abbildung 7. 7 dargestellte Ausgabe.

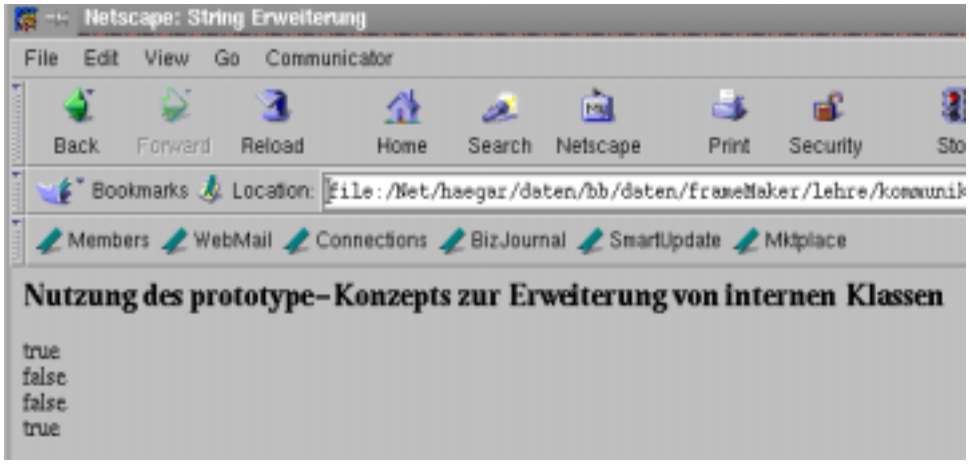


Abbildung 7. 7 Ausgabe von Beispiel 7.6

Wir sehen, daß wir JavaScript internen Klassen auf die gleiche Art und Weise wie selbstdefinierten Klassen Methoden hinzufügen können. Auch hier ist es so, daß Ob-

jekte der geänderten Klasse, selbst wenn sie vor Änderung der Klasse erzeugt wurden, über die neuen Methoden verfügen. In Beispiel 7.6 nutzen wir eine interne Funktion der String-Klasse (*charAt*).

7.5 Statische Variablen und Methoden

7.5.1 Statische Methoden

In der bisherigen Diskussion benötigten wir immer ein Objekt, um Methoden einer Klasse auszuführen (im obigen Beispiel mußten wir erst das Objekt `ersterMensch` mit dem Schlüsselwort *new* erzeugen, erst daraufhin konnten wir die Methoden der Klasse nutzen). Es gibt aber Fälle, wo wir die Methoden einer Klasse nutzen wollen, aber ein Objekt der Klasse nicht unbedingt brauchen (dies ist z.B. immer dann der Fall, wenn wir keine Properties benötigen).

Betrachten wir als Beispiel die JavaScript interne Klasse `Math`. In `Math` sind die mathematischen Funktionen, die JavaScript unterstützt, zusammengefaßt. Nehmen wir nun an, wir wollten die Wurzel einer Zahl berechnen. Ohne statische Methoden würde der Code ungefähr so aussehen:

Beispiel 7.7 Wurzel einer Zahl ohne statische Methoden:

```
<! Dateiname: matheFalsch.html>
<HTML>
<HEAD>
      <TITLE>
             Mathe Falsch
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Nicht funktionierende Mathematik
      </H2>
      <P>
            Bitte geben Sie in das aufgehende Fenster die Zahl ein,
            aus der Sie die Wurzel ziehen wollen.

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/matheFalsch.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: matheFalsch.js

var matheObjekt;
matheObjekt = new Math(); // Default-Konstruktor
var eingabe;
var ergebnis;

eingabe = parseFloat(prompt("Bitte geben Sie die Zahl ein, "+
```

```
                  "deren Wurzel Sie wissen wollen!"));
ergebnis = matheObjekt.sqrt(eingabe);

document.write("Die Wurzel ist: " + ergebnis + "<BR>");
```

Der Browser führt dies aber nicht aus, sondern bricht mit der Fehlermeldung "Math is not a constructor" ab.

Wir wollen hier ein Objekt der Klasse *Math* erzeugen, obwohl wir lediglich eine Funktion auf eine eingebene Zahl anwenden wollen. Variablen existieren in der Klasse *Math* nicht. Um solche Dinge zu vereinfachen, existieren in JavaScript statische Methode. Statische Methoden werden in der JavaScript-Dokumentation als *staticfunctions* bezeichnet. Solche Methoden können mit der Syntax:

```
Klassenname.Methodenname([Parameter])
```

aufgerufen werden. D.h., unser Beispiel muß wie folgt implementiert werden:

Beispiel 7.8 Benutzung der Klasse Math, richtige Implementierung

```
<! Dateiname: matheRichtig.html>
<HTML>
<HEAD>
      <TITLE>
             Mathe Richtig
      </TITLE>
</HEAD>

<BODY>
      <H2>
            Funktionierende Mathematik
      </H2>
      <P>
            Bitte geben Sie in das aufgehende Fenster die Zahl ein,
            aus der Sie die Wurzel ziehen wollen.
      </P>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/matheRichtig.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: matheRichtig.js

var matheObjekt;
var eingabe;
var ergebnis;

eingabe = parseFloat(prompt("Bitte geben Sie die Zahl ein, "+
                  "deren Wurzel Sie wissen wollen!", ""));
ergebnis = Math.sqrt(eingabe);
```

```
document.write("Die Wurzel ist: " + ergebnis + "<BR>");
```

7.5.2 Statische Properties (Variablen)

Properties, wie wir sie bis jetzt kennengelernt haben, werden bei der Erzeugung von Objekten angelegt. Speicherplatz wird für sie reserviert, sie werden initialisiert. Dies bedeutet, jedes Objekt hat eine eigene Kopie seiner Properties. Änderungen, die ein Objekt an seinen Properties vornimmt, sind in anderen Objekten nicht zu sehen. Dies ist in den allermeisten Anwendungsfällen auch sinnvoll.

In manchen Fällen können Properties sinnvoll sein, für die es eine Kopie pro Klasse gibt. Nehmen wir folgendes Beispiel:

Wir möchten mitzählen, wieviel Objekte von der Klasse Mensch erzeugt werden. Dies kann keine Property eines Objekts sein, denn einzelne Objekte der Klasse Mensch können nicht wissen, wieviele andere Mensch-Objekte nach ihnen erzeugt werden. Für diese Zwecke bietet JavaScript die Möglichkeit, statische Properties zu definieren. Beispiel 7.9 zeigt, wie so etwas realisiert wird:

Beispiel 7.9 Statische Properties der Klasse Mensch

```
<! Dateiname: arbeiteMitMensch6.html>
<HTML>
<HEAD>
      <TITLE>
            Objekte 6
      </TITLE>
            <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/Mensch3.js">

      </SCRIPT>
</HEAD>
<BODY>
      <H2>
            Properties von Mensch Objekten!
      </H2>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/arbeiteMitMensch6.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörigen JavaScript-Dateien

```
// Code um mit der Klasse Mensch zu arbeiten
// Dateiname: arbeiteMitMensch6.js

ersterMensch = new Mensch(); // Default-Konstruktor

document.write("Eigenschaften des ersten Menschen <BR>");
document.write(ersterMensch.getName() + "<BR>");
document.write(ersterMensch.getVorname() + "<BR>");
document.write(ersterMensch.getFullName() + "<BR>");
```

```
document.write(Mensch.anzahl + "<BR>");

zweiterMensch = new Mensch( "Bluemel", "Bernd"); //Konstruktor mit Wertangabe

document.write("Eigenschaften des zweiten Menschen <BR>");
document.write(zweiterMensch.getName() + "<BR>");
document.write(zweiterMensch.getVorname() + "<BR>");
document.write(zweiterMensch.getFullName() + "<BR>");
document.write(Mensch.anzahl + "<BR>");

// Objekt Mensch Implementierung
// Dateiname: Mensch3.js

Mensch.anzahl = 0;
function Mensch()
{
      if (Mensch.arguments.length == 0)
      // keine Parameter (Default-Konstruktor)
      {
            this.name = "WasWeissIch";
            this.vorname = "WeissIchAuchNich";
            Mensch.anzahl +=1;
      }

      if (Mensch.arguments.length == 2)
      // Alle Parameter angegeben
      {
            this.name = Mensch.arguments[0];
            this.vorname = Mensch.arguments[1];
            Mensch.anzahl +=1;
      }
}

function Mensch_getName ()
{
      return this.name;
}
function Mensch_getVorname ()
{
      return this.vorname;
}

function Mensch_getFullName()
{
      return (this.vorname + " " + this.name);
}

function Mensch_setName (neuName)
{
      this.name = neuName;
}
function Mensch_setVorname (neuVorname)
{
```

```
      this.vorname = neuVorname;
}

Mensch.prototype.getName = Mensch_getName;
Mensch.prototype.getVorname = Mensch_getVorname;
Mensch.prototype.getFullName = Mensch_getFullName;
Mensch.prototype.setName = Mensch_setName;
Mensch.prototype.setVorname = Mensch_setVorname;
```

In Beispiel 7.9 wird die statische Property *anzahl* durch die Zeile:

```
Mensch.anzahl = 0;
```

deklariert und mit 0 initialisiert. Wir sehen, daß statische Properties mit der Syntax:

```
Klassenname.Propertyname
```

deklariert und später auch benutzt werden.

Im Konstruktor der Klasse Mensch wird bei Erzeugung eines neuen Objektes durch

```
Mensch.anzahl +=1;
```

diese Property um eins hochgezählt. Das dies auch richtig funktioniert zeigt Abbildung 7. 7.

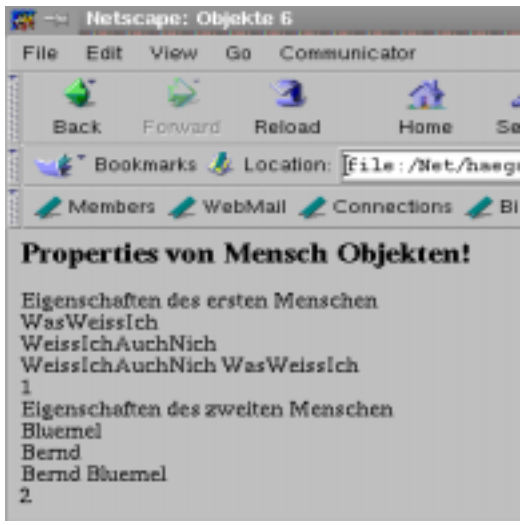


Abbildung 7. 8 Ausgabe von Beispiel 7.9

Dies beendet die Diskussion der JavaScript-Syntax. Im weiteren werden wir die Nutzung der JavaScript-API diskutieren. Dabei handelt es sich um Objekte, die standardmäßig in jedem JavaScript unterstützenden Browser zur Verfügung stehen. Wir haben solches im Verlauf des Kurses schon getan, `document` ist beispielsweise ein Objekt der API und `write()` eine Methode von `document`.

8 Das window-Objekt

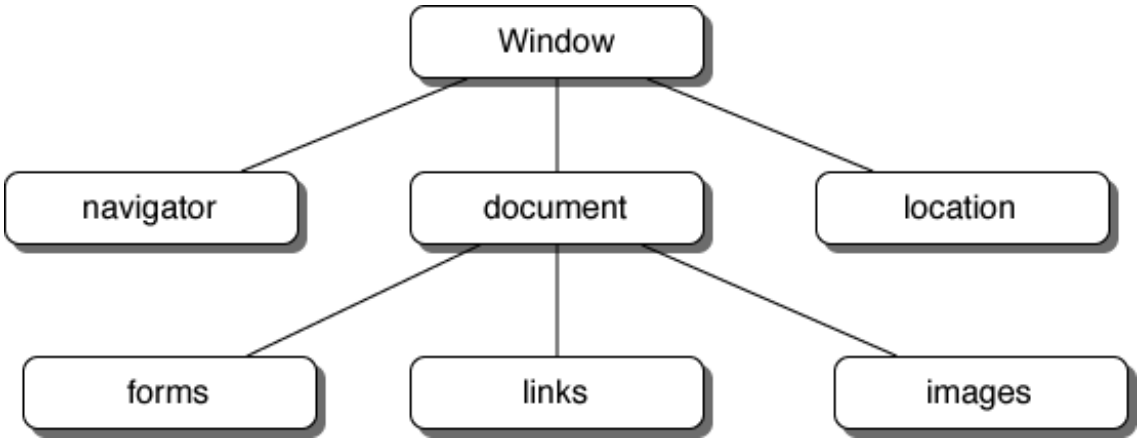
Das Browser-Fenster, in dem die html-Seite, die den JavaScript-Code enthält, dargestellt wird, wird durch ein window-Objekt repräsentiert.

Das window-Objekt verfügt über Methoden wie `prompt()` (haben wir ja bereits kennengelernt) oder `alert()` und `confirm()` (werden im folgenden besprochen).

Darüberhinaus hat das window-Objekt Eigenschaften, die sich von JavaScript aus steuern lassen. Dazu gehört die Statuszeile (`status`-Eigenschaft), die URL, die gerade dargestellt wird (`location`) oder die URL's der Seiten, die bis jetzt dargestellt wurden (`history`).

In Wirklichkeit sind sind sogar alle anderen JavaScript-html-Objekte entweder selbst Eigenschaften des window-Objekts oder Eigenschaften[13] von Eigenschaften des window-Objekts (auch das document-Objekt, welches wir bereits öfter zum Schreiben in das Browser-Fenster benutzt haben).

Das window-Objekt ist also die Wurzel (root) der JavaScript-Objekt-Hierarchie. Abbildung 8. 1 zeigt einen Auszug aus der JavaScript-Objekt-Hierarchie. Die dort dargestellten Objekte werden in den folgenden Kapiteln besprochen.



bbildung 8. 1 Auszug aus der JavaScript-Objekt-Hierarchie

8.1 Benutzer-Interaktion mittels des window-Objekts

Das window-Objekt verfügt über 3 Methoden, mittels derer es mit dem Benutzer interagieren kann. Die erste (`prompt()`) haben wir bereits kennengelernt. `prompt()` öffnet ein eigenes Fenster oberhalb des Browser-Fensters. Das Fenster enthält 2 Bereiche: Im ersten Bereich wird eine Meldung dargestellt (erster Parameter von `prompt()`), im zweiten Bereich kann der Benutzer Eingaben tätigen (der zweite Parameter von `prompt()` ist eine Vorgabe im Eingabefeld).

`confirm()` erwartet vom Benutzer eine Bestätigung. Der einzige Parameter von `confirm()` ist eine Meldung, die im von `confirm()` aufgespannten Fenster darge-

13.In Kapitel 7 haben wir ja gesehen, daß auch Objekte Eigenschaften von Objekten sein können.

stellt wird. Der Benutzer kann mit einem OK-Button bestätigen (*confirm()* liefert dann *true* zurück) oder mit einem »Cancel-Button« abbrechen (*confirm()* liefert *false* zurück).

alert() gibt ein Fenster mit einem Hinweis aus. Das Fenster kann mittels eines OK-Buttons geschlossen werden.

Die von *prompt()* und *confirm()* geöffneten Fenster sind modal. Dies bedeutet, der laufende JavaScript-Code wird angehalten, bis der Benutzer seine Eingabe getätigt hat, das Programm ist geblockt. *alert()* hingegen blockiert das Programm nicht.

Beispiel 8.1 zeigt eine Änderung des Fakultäten-Beispiels mit Einsatz von *confirm()*.

Beispiel 8.1 Fakultäten-Programm mit confirm, alert und prompt

```
<!-- Dateiname:fakMitConfirm.html -->
<HTML>
<HEAD>
      <TITLE>
            FAKULT&Auml;TEN MIT EINGABE und Fenstern
      </TITLE>
</HEAD>
<BODY>
      <H2>
            Dies Programm berechnet Fakult&auml;ten.
      </H2>
      <P>
            Geben Sie unten die Zahl ein, bis zu der Sie die
            Fakult&auml;ten berechnet haben m&ouml;chten!
      </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/fakMitConfirm.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname fakMitConfirm.js

var eingabe;
var ende;
var i;
var fact;
var wirklich = true;
var confirmMessage="Dies ist eine sehr hohe Zahl!!" +
      "\n Die Fakult‰ts-Berechnung kann sehr lange dauern!!" +
      "\n Sie kˆnnen vermutlich erstmal Kaffe trinken gehen!";
var alertMessage = "Berechnung l‰uft!";

eingabe = prompt ("Zu berechnende Fakult‰t:", "");
ende = parseInt (eingabe);
if (ende > 150)
```

```
{
      wirklich = confirm(confirmMessage);
}
if (wirklich)
{
      alert (alertMessage);
      fact = 1;
      for(i=1; i<= ende; i++)
      {
            fact = fact * i;
            document.write(i + "! = " + fact + "<BR>");
      }
}
else
{
      document.write("Berechnung durch Benutzer abgebrochen!!" + "<BR>");
}
```

Die einzige Änderung zu unserem ursprünglichen Fakultätenprogramm ist die Abfrage einer Benutzerbestätigung, wenn die Zahl, zu der die Fakultät berechnet werden soll, größer als 150 ist. In diesem Fall wird ein *confirm*-Fenster aufgeblendet. Wählt die Benutzerin "OK", gibt *confirm true* zurück. Der bool'schen Variablen *wirklich* wird damit der Wert *true* zugewiesen und die Fakultäten werden berechnet. Klickt die Benutzerin hingegen *Cancel*, wird von *confirm false* zurückgegeben, *wirklich* wird damit *false* zugewiesen und die Berechnung findet nicht statt. Abbildung 8. 1 zeigt Abbildungen der drei Interaktionsfenster des *window*-Objekts.

Erstaunlich an diesem Beispiel (und eigentlich auch an den vorhergegangenen Nutzungen von `prompt(), confirm(), alert ()` und `document.write()` ist, daß wir den Namen des window-Objektes nicht voranstellen müssen. Sie haben ja in Kapitel 7 gelernt, daß der Aufruf einer Methode eines Objekts mittels Objektname, Punkt, Methodenname erfolgt. Dies ist hier ja nicht der Fall. Der Grund, daß der Code aus Beispiel 8.1 dennoch funktioniert ist einfach, daß JavaScript automatisch, sofern kein Name eines window-Objekts angegeben ist, den Namen des aktuellen Fensters voranstellt. Der Name des aktuellen Fensters ist einfach `window`. Wir können dies in unseren Programmen aber auch selber tun. Beispiel 8.1 ist äquivalent zu:

Beispiel 8.2 Beispiel 8.1 mit Referenz des window-Objekts

```
<!-- Dateiname:fakMitConfirmWindow.html -->
<HTML>
<HEAD>
      <TITLE>
            FAKULT&Auml;TEN MIT EINGABE und Fenstern
      </TITLE>
</HEAD>
<BODY>
      <H2>
            Dies Programm berechnet Fakult&auml;ten.
      </H2>
```

Abbildung 8. 2 alert-, prompt- und confirm-Fenster

```
      <P>
            Geben Sie unten die Zahl ein, bis zu der Sie die
            Fakult&auml;ten berechnet haben m&ouml;chten!
      </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/fakMitConfirmWindow.js">

      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname fakMitConfirmWindow.js

var eingabe;
var ende;
var i;
var fact;
var wirklich = true;
var confirmMessage="Dies ist eine sehr hohe Zahl!!" +
      "\n Die Fakult‰ts-Berechnung kann sehr lange dauern!!" +
      "\n Sie k^nnen vermutlich erstmal Kaffe trinken gehen!";
var alertMessage = "Berechnung l‰uft!";

eingabe = window.prompt ("Zu berechnende Fakult‰t:", "");
ende = window.parseInt (eingabe);
```

```
if (ende > 150)
{
      wirklich = window.confirm(confirmMessage);
}
if (wirklich)
{
      window.alert (alertMessage);
      fact = 1;
      for(i=1; i<= ende; i++)
      {
            fact = fact * i;
            window.document.write(i + "! = " + fact + "<BR>");
      }
}
else
{
      window.document.write("Berechnung durch Benutzer abgebrochen!!" +
"<BR>");
}
```

In Beispiel 8.2 haben wir allen Methodenaufrufen eine explizite Referenz des aktuellen window-Objekts hinzugefügt. JavaScript erkennt, das in allen Methoden-Aufrufen Referenzen auf ein window-Objekt existieren und fügt daher selbst keine mehr hinzu.

Wir sehen an Beispiel 8.2 auch, daß auch *parseInt()* eine Methode des window-Objekts ist. Darüberhinaus erkennen wir, wie wir Methoden von Objekt-Eigenschaften eines Objekts ansprechen können. Dies geschieht einfach mittels der Syntax:

```
Objektname.Objektname.Methodenname()
```

In Beispiel 8.2 sehen wir dies anhand der Nutzung der write-Methode des document-Objekts des Windows-Objekts *(window.document.write(i + "! = " + fact + "<BR>")*. Dies erlaubt uns z.B. auch in andere Fenster zu schreiben. Wir müssen nur den Namen des Fensters wissen (eine Referenz zum Fenster haben) und können dann vermittels:

```
Fenstername.document.write()
```

in dieses Fenster schreiben.

Beispiel 8.3 illustriert diese Technik. Sinn von Beispiel 8.3 ist es, sich für eine Bestellung zu bedanken. Wir stellen uns dazu vor, daß der Benutzer gerade über ein html-Formular eine Bestellung abgeschickt hat. Wir möchten uns beim Benutzer bedanken und ihm zu diesem Zweck eine von uns schön gestylte html-Seite aufblenden.

Bestell-Formulare werden allerdings erst in Kapitel 12 besprochen, so daß wir uns das Bestellformular in der Seite denken müssen.

Beispiel 8.3 Öffnen eines weiteren Fensters

```
<!-- Dateiname: dankeFuerBestellung.html -->
<HTML>
<HEAD>
```

```
      <TITLE>
             Bestellformular
      </TITLE>

</HEAD>

<BODY>
      <H2>
            Bestellfenster
      </H2>

            <P>
                  Hier soll ein Bestellformular stehen.
                  Wie dies geht sehen wir sp&auml;ter.
            </P>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC="./javaScript/dankeFuerBestellung.js">
      </SCRIPT>

</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: dankeFuerBestellung.js
// oeffnen eines neuen Fensters

var dankeFenster = window.open("","Danke");

// Der Schreibbereich des Fensters

var dankeDokument = dankeFenster.document;
dankeDokument.write("<HTML> \n");
dankeDokument.write("<HEAD> \n");
dankeDokument.write("<TITLE> \n");
dankeDokument.write("Vielen Dank!!");
dankeDokument.write("</TITLE> \n");
dankeDokument.write("</HEAD> \n");

dankeDokument.write("<BODY> \n" );
dankeDokument.write("<H1> Vielen Dank f&uuml;r Ihre Bestellung! </H1>");
dankeDokument.write("<P> Sie erhalten Ihre Ware in 2 Wochen </P>");
dankeDokument.write("</BODY> \n");
dankeDokument.write("</HTML> \n");
```

Das zweite Fenster erhält den Namen *dankeFenster*. Über diesen Namen ist das neue Browser-Fenster jetzt ansprechbar. Danach definieren wir eine Variable *dankeDokument*. Sie enthält nun den Schreibbereich des Fensters. Dies geschieht, um Schreibaufwand zu sparen. *dankeDokument.write* ist identisch mit *dankeFenster.document.write*. Danach schreiben wir den gewünschten html-Code in das neue Fenster. Als Ergebnis öffnet sich ein neues Browser-Fenster, indem der Browser diesen Code darstellt (vgl. Abbildung 8. 1).

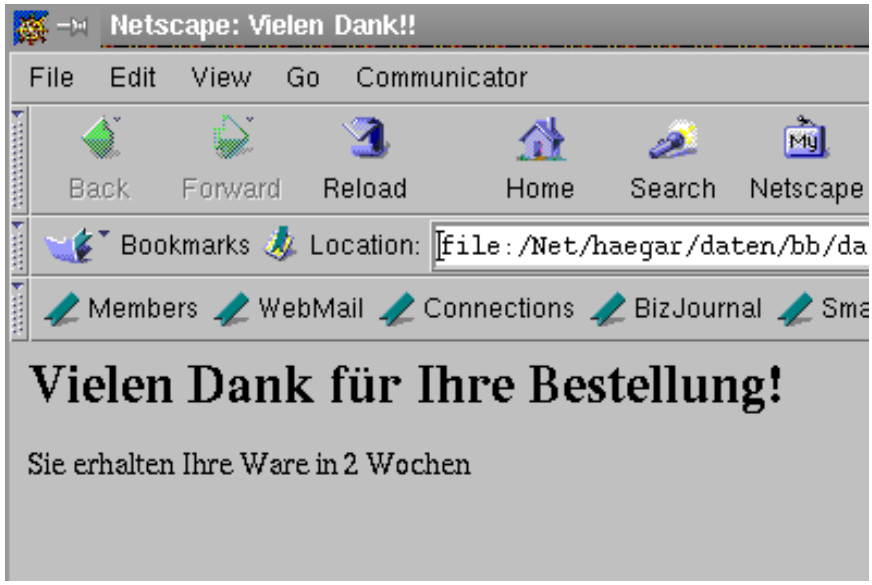


Abbildung 8. 3 Von JavaScript dynamisch geöffnetes Fenster

Dieses "Feature" sollte man meiner Ansicht nach sehr vorsichtig behandeln. Jedes neue Fenster nimmt Platz auf dem Workspace der Benutzerin weg. Dies macht den Bildschirminhalt für die Benutzerin schnell sehr unübersichtlich.

Neben den bisher behandelten gibt es weitere Eigenschaften und Methoden des *window*-Objekts. Teilweise werden sie in folgenden Kapiteln im Zusammenhang mit den anderen den Browser steuernden JavaScript-Objekten besprochen. Für die restlichen verweise ich auf die Dokumentation.

9 Das document-Objekt

Das document-Objekt repräsentiert den Schreibbereich des Browser-Fensters. Seine wichtigste Methode ist die *write()*-Methode. Wir haben die *write()*-Methode schon sehr häufig angewendet, so daß sich eine weitere Diskussion hier erübrigt.

Neben der *write()*-Methode existiert eine *writeln()*-Methode. *writeln()* fügt einen Zeilenvorschub an. Dies ist, wenn wir html schreiben, nur bedingt sinnvoll, da Zeilenvorschübe von Browsern ja ignoriert werden. Man kann in JavaScript allerdings auch nicht html-Dokumente (z.B. Ascii-Dokumente) öffnen[14]. Dann wird die *writeln()*-Methode wieder sinnvoll.

Daneben enthält das document-Objekt einige ganz nützliche Eigenschaften, z.B:

- *lastModified*: Enthält das Datum der letzten Änderung.
- *URL*: Enthält den URL des Dokuments.
- *title*: Enthält den Titel des Dokuments (Text zwischen den <TITLE> und </TITLE> Tags.
- *referrer*: Der URL der Seite von der der Benutzer kam.
- *bgColor, fgColor, linkColor, alinkColor*: Für das Dokument eingestellte Farben.

Beispiel 9.1 zeigt den sinnvollen Einsatz einer dieser Eigenschaften.

Beispiel 9.1 Einsatz der lastModified Eigenschaft des document-Objekts im Fakultät-Beispiel

```
<!-- Dateiname: fakMitEingabe.html -->
<HTML>
<HEAD>
	<TITLE>
		FAKULT&Auml;TEN MIT EINGABE und Datum lezte &Auml;derung
	</TITLE>
</HEAD>

<BODY>
	<H2>
		Dies Programm berechnet Fakult&auml;ten.
	</H2>
	<P>
		Geben Sie unten die Zahl ein, bis zu der Sie
		die Fakult&auml;ten berechnet haben m&ouml;chten!
	</P>
	<SCRIPT LANGUAGE = "JavaScript" SRC = "./javaScript/fakultaet.js">
	</SCRIPT>

	<SCRIPT LANGUAGE = "JavaScript" SRC = "./javaScript/dateOfLastChange.js">
	</SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname dateOfLastChange.js
```

14.Dies behandeln wir allerdings nicht.

```
var dateChanged=new Date(document.lastModified);
var day = dateChanged.getDate();
var month = germanMonth(dateChanged.getMonth());
var year = dateChanged.getFullYear();
document.write("Datum der letzten &Auml;nderung: ");
document.write(day + " " + month + ". " + year);

function germanMonth(monthAsInteger)
{
      if (monthAsInteger == 0)
      {
            return "Jan";
      }
      if (monthAsInteger == 1)
      {
            return "Feb";
      }
      if (monthAsInteger == 2)
      {
            return "M&auml;r";
      }
      if (monthAsInteger == 3)
      {
            return "Apr";
      }
      if (monthAsInteger == 4)
      {
            return "Mai";
      }
      if (monthAsInteger == 5)
      {
            return "Jun";
      }
      if (monthAsInteger == 6)
      {
            return "Jul";
      }
      if (monthAsInteger == 7)
      {
            return "Aug";
      }
      if (monthAsInteger == 8)
      {
            return "Sep";
      }
      if (monthAsInteger == 9)
      {
            return "Okt";
      }
      if (monthAsInteger == 10)
      {
            return "Nov";
```

```
        }
        if (monthAsInteger == 11)
        {
                return "Dez";
        }
}
```

Auf die Darstellung von *fakultaet.js* habe ich verzichtet.

In der Datei *dateOfLastChange.js* wird aus dem String, den *document.lastModified* zurückgibt, ein Date-Objekt erzeugt. Wie alle anderen Objekte auch wird das Date-Objekt mit dem Schlüsselwort *new()* erzeugt. Dem Konstruktor wird das Datum, zu dem das Objekt erzeugt werden soll, als String übergeben. Nun können die Methoden der Date-Klasse eingesetzt werden. *getDate()*[15] gibt den Tag der letzten Änderung zurück, *getMonth()* den Monat und *getFullYear()* das Jahr (4-stellig). Wie immer werden die Methoden der Objekte vermittels eines Punktes an den Objektnamen angeschlossen.

Zur "schönen" Darstellung des Monats dient die Funktion germanMonth. *getMonth()* liefert den Monat als Zahl zwischen 0 und 11 zurück, *getMonth()* wandelt dies in den Namen des Monats um (vgl. Abbildung 9. 1).

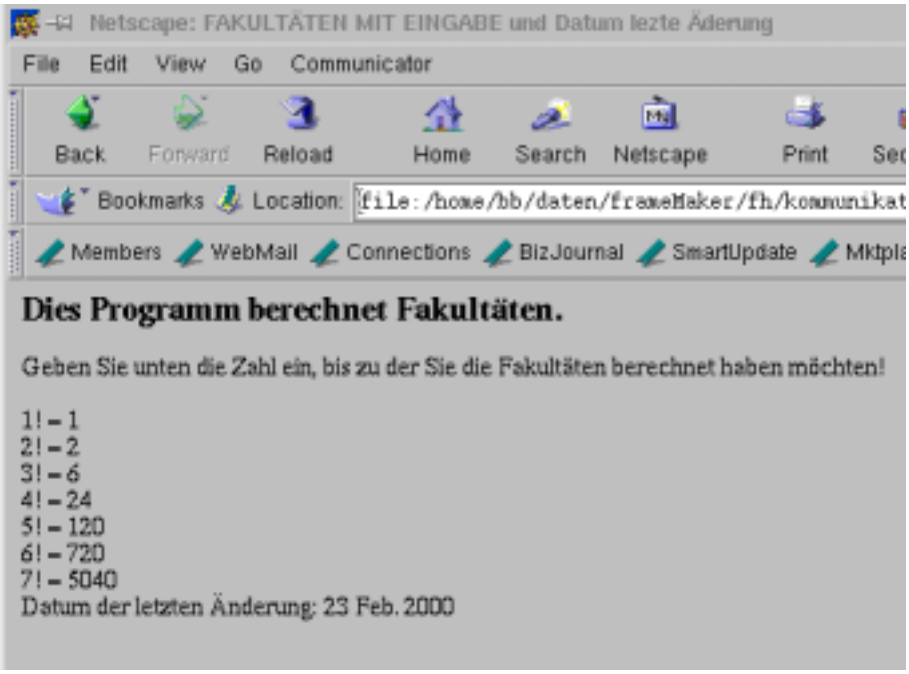


Abbildung 9. 1 Ausgabe der Datums-Änderungsfunktion

15.Finde ich selbst nicht die beste Wahl, getDay() wäre passender gewesen.

10 Event-Handler

Bislang wurde JavaScript-Code in html-Seiten genau dann ausgeführt, wenn der Browser im <BODY>-Teil auf den <JavaScript>-Tag gestoßen ist. Der JavaScript-Code startet also ohne Interaktion mit dem Benutzer. Der Benutzer kann auch nicht entscheiden, welcher Teil des Codes ausgeführt wird.

Verglichen mit heutigen Oberflächen ist dies "Technik von Gestern". Vergegenwärtigen Sie sich Ihre Umgehensweise mit Rechnern. Wenn ein Computer mit einer grafischen Oberfläche startet, wartet das Betriebssystem nach Ihrer Anmeldung auf eine Aktion Ihrerseits. Sie starten z.B. ein Textverarbeitungsprogramm. Dies tun Sie z.B. dadurch, daß Sie auf das Icon des Programms klicken. Nach dem Start der Textverarbeitung wartet das Programm auf weitere Eingaben von Ihnen. Sie klicken z.B. auf ein Bildchen, um eine Datei zu laden. Daraufhin führt die Textverarbeitung den zugehörigen Code aus und präsentiert Ihnen eine Datei-Öffnungsbox.

Bei jeder Aktion Ihrerseits wird ein Event (Ereignis) ausgelöst. Dann wird der zu diesem Event gehörige Code durchgeführt. Dieselbe Möglichkeit bietet auch die Clientseitige html-JavaScript-Kombination.

html stellt grafische Interaktionselemente bereit. Dies sind z.B. Links, auf die die Benutzer klicken können und damit den Browser veranlassen, eine neue Seite zu laden. Dies sind aber auch Eingabefelder oder Buttons in Formularen. Abbildung 8. 1 zeigt eine Auswahl grafischer html-Interaktionselemente.

Die Benutzerin kann hier in den Textfeldern Eingaben machen und Buttons klicken, um Berechnungen zu starten. Bei der Programmierung der html-Seite aus Abbildung 8. 1 würde sich eine JavaScript-Lösung anbieten. Wir haben ja im Prinzip eine zu unserem Fakultätenbeispiel identische Problematik. Allerdings gibt es ein Problem:

Mit unserem bisherigen Wissen können wir nicht feststellen, ob und wann der Benutzer die grafischen Interaktionselemente bedient.

Alle grafischen html-Interaktionselemente bieten Event-Handler, aus denen heraus JavaScript-Code ausgeführt werden kann. Wir werden dies jetzt anhand eines (zugegebenermaßen nicht besonders sinnvollen) Beispiels eines Event-Handlers des Link-Objekts betrachten.

Jeder Link in einer html-Datei ist ein Objekt, welches von JavaScript angeprochen werden kann (das Link-Objekt wird ausführlich in Kapitel 11 betrachtet). Für das Link-Objekt ist der onMouseOver-Event-Handler definiert. JavaScript-Code, der dem onMouseOver-Event-Handler folgt, wird ausgeführt, wenn der Benutzer den Maus-Zeiger auf den Link führt. Beispiel 10.1 verdeutlicht dies.

Beispiel 10.1 Benutzung des onMouseOver-Event-Handler des Link-Objekts

```
<!-- Dateiname: linkSchwachsinn.html -->
<HTML>
<HEAD>
      <TITLE>
            Link Abfrage
      </TITLE>
</HEAD>
```

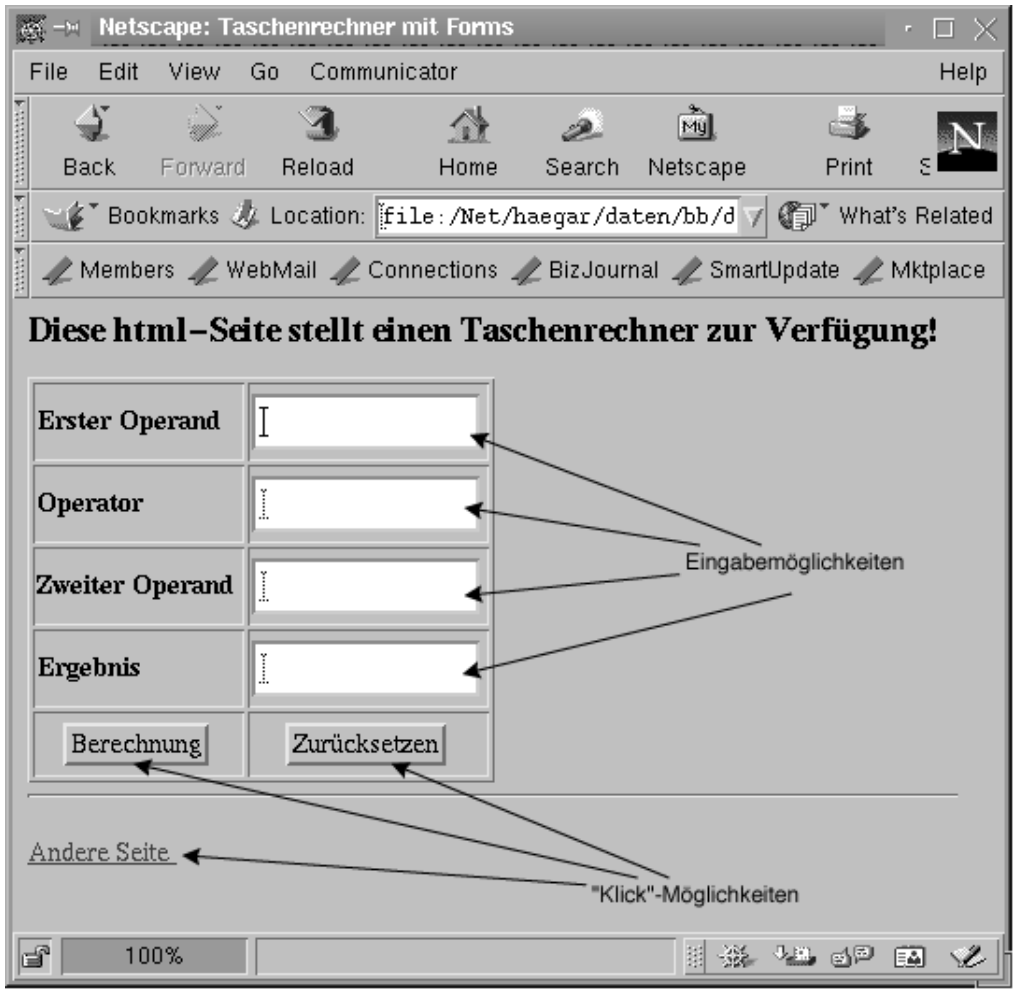


Abbildung 10. 1 Einige grafische html-Interaktionselemente

```
<BODY>
      <H2>
            Dies Programm macht Unsinn!
      </H2>

      <P>
            <A HREF= "http://www.mfh-iserlohn.de"
                  onMouseOver = "prompt ('Geben Sie Irgendwas ein!', '');">
                  MFH
            </A>
      </P>
</BODY>
</HTML>
```

Die Aktionen, die ausgeführt werden sollen, wenn das Ereignis (engl. Event) des Event-Handlers eintritt, werden an diesen nach einem Gleichheitszeichen angeschlossen. Sie werden in Hochkommata eingeschlossen. In Beispiel 10.1 wird also jedesmal, wenn der Benutzer die Maus auf den Link führt, ein Eingabefenster geöffnet, in dem “Geben Sie irgendwas ein!” steht.

Die Event-Handler von Java-Script-Objekten werden jeweils bei der Betrachtung der zugehörigen Objekte besprochen.

Aus Event-Handlern darf `document.write()` **nicht benutzt werden.**

11 Das Link-Objekt

Jeder Link in einer html-Datei wird von JavaScript als Objekt behandelt und kann von JavaScript angesprochen werden. Für das Link-Objekt sind u.a. 3 Event-Handler definiert:

- *onMouseOver*
- *onMouseOut*
- *onClick*.

onMouseOver wird ausgelöst, wenn der Benutzer die Maus auf den Link bewegt, *onMouseOut*, wenn die Maus vom Link wegbewegt wird, *onClick*, wenn der Benutzer auf den Link klickt. Die folgenden Beispiele veranschaulichen die Nutzung dieser Event-Handler.

Beispiel 11.1 Server-Verlassen mit Nachfrage

```
<!-- Dateiname: linkAbfrage.html -->
<HTML>
<HEAD>
	<TITLE>
			Link Abfrage
	</TITLE>
	<SCRIPT LANGUAGE = "JavaScript"
			SRC = "./javaScript/linkAbfrage.js">

	</SCRIPT>
</HEAD>

<BODY>
	<H2>
		Dies Programm fragt beim Verlassen des Servers nach!
	</H2>
	<P>
		<A HREF= "http://www.mfh-iserlohn.de" onClick = "return(frage())";>
			MFH
		</A>
	</P>


</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: linkAbfrage.js
function frage()
{
	if (confirm ("Sie verlassen diesen Server!"))
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

Beispiel 11.1 nutzt eine Eigenschaft des *onClick*-Event-Handlers aus. Gibt der *onClick*-Event-Handler *true* zurück, wird der Link ausgeführt (die im Link genannte Seite wird geladen), gibt er *false* zurück, wird der Link nicht ausgeführt. In Beispiel 11.1 wird auf die Leitseite der MFH verwiesen. Klickt der Benutzer auf den Link, wird die Funktion *frage()* ausgeführt. *frage()* bringt ein *confirm*-Fenster auf den Bildschirm, welches nachfragt, ob der Server wirklich verlassen werden soll (wir nehmen an, wir befinden uns auf einer Seite der Fachhochschule Bochum). Klickt der Benutzer auf den *OK*-Button des *confirm*-Fensters, gibt *frage() true* zurück und der Link wird ausgeführt, klickt er *Cancel* gibt *frage() false* zurück und die Seite wird nicht verlassen.[16]

Beispiel 11.2 FTP-Server-Ermittlung

```
<!-- Dateiname: linkZusammemsetzen.html -->
<HTML>
<HEAD>
      <TITLE>
                  Link Abfrage
      </TITLE>
      <SCRIPT LANGUAGE = "JavaScript"
            SRC = "./javaScript/linkZusammensetzen.js">

      </SCRIPT>
</HEAD>

<BODY>
      <H2>
            Dies Programm verzweigt zum richtigen FTP-Server!
      </H2>
      <P>
      <A HREF= "" onClick = "this.href = frage();"
                  onMouseOver = "status = 'Ermittlung des richtigen FTP-
Servers';"
                  onMouseOut = "status = ''">
                  Zum FTP-Server
            </A>

      </P>


</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: linkZusammemsetzen.js

function frage()
```

16.Die Zeile onClick = 'return(confirm("Sie verlassen diesen Server!"))' hätte übrigens denselben Effekt. Beachten Sie die Nutzung der Apostrophes ', da Anführungszeichen wegen ihrer Nutzung in der String-Konstante in confirm nicht mehr eingesetzt werden können.

```
{
      var endung;
      var meinLink = "http://www.firma";
      endung = prompt("Geben Sie Ihre L&auml;nderkennung an!", "de");
      switch (endung)
      {
            case "com" : meinLink = meinLink + ".com";
            case "edu" : meinLink = meinLink + ".edu";
            case "gov" : meinLink = meinLink + ".com";
            case "uk"  : meinLink = meinLink + ".uk";
            default    : meinLink = meinLink + ".de";
      }
      return meinLink;
}
```

Beispiel 11.2 zeigt die Nutzung von 3 Event-Handler des Link-Objektes. Führt der Benutzer die Maus auf den Link, wird in der Statuszeile des Browsers der String *"Ermittlung des richtigen FTP-Servers"* eingeblendet. *status* ist eine Eigenschaft des window-Objektes (*status* = ist damit äquivalent zu *window.status*=) und kann von einem JavaScript-Programm geändert werden.[17] Verläßt der Mauszeiger den Link, wird die Anzeige im Statusfeld gelöscht (durch den leeren String ersetzt).

Klickt der Benutzer auf den Link, wird die Funktion *frage()* ausgeführt. *frage()* setzt zunächst den Beginn des Links, auf den später verzweigt werden soll (*meinLink = "ftp://www.firma").* Danach wird ein Eingabefenster aufgeblendet, in dem der Benutzer seine Länderkennung eingeben soll. Abhängig von der vom Benutzer eingegebenen Länderkennung, wird dann die URL des dem Benutzer nächstgelegenen FTP-Servers ermittelt.

Die ermittelte URL wird der Eigenschaft *href* des Linkobjekts zugewiesen. *href* enthält die Sprungadresse des Links. Das Schlüsselwort *this* bedeutet, daß der gerade aktuelle Link (der Link dessen *onClick* Event-Handler ausgeführt wurde) gemeint ist. Der Link wird nun ausgeführt und die erste Seite des FTP-Servers in den Browser geladen.

Ein weiteres bekanntes Beispiel zur Nutzung des Link-Objekts (Mouse-Rollovers) wird im Zusammenhang mit Bildern (image-Objekt) in Kapitel 14 diskutiert.

17. Defaultmäßig wird bei Netscape im Statusfeld die Adresse angezeigt, auf die der Link zeigt. Die ist in unserem Beispiel aber noch nicht bekannt.

12 Das Form-Objekt

Eine der interessantesten Eigenschaften von JavaScript ist die Zusammenarbeit mit Formularen. JavaScript kann die Werte von Texteingabefeldern von Formularen auslesen und ändern.

Formulare werden in html durch das Form-Tag repräsentiert. Das Form-Tag wird in JavaScript auf ein Form-Objekt abgebildet. Form-Objekte, sowie alle ihre Elemente (Textfelder, Buttons, etc) verfügen über Event-Handler, von denen JavaScript-Programme aufgerufen werden können. Diese Eigenschaften wollen wir zunächst an unserem Taschenrechnerbeispiel erläutern. Ein Taschenrechner, wie in Abbildung 8. 1 dargestellt, soll programmiert werden:

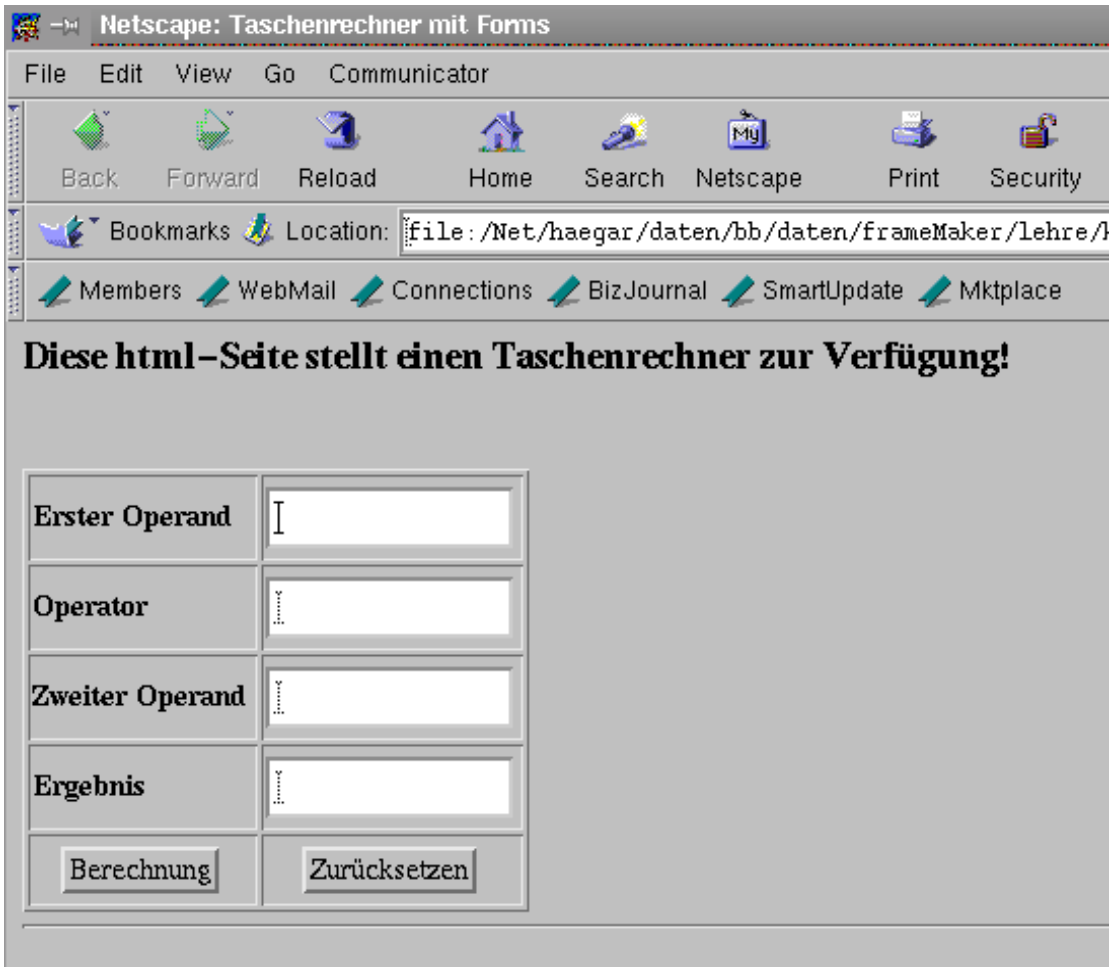


Abbildung 12. 1 Das Taschenrechnerbeispiel

Beispiel 12.1 Der Taschenrechner mit einem Formular

```
<!-- Dateiname: rechnerMitForm.html -->
<HTML>
<HEAD>
      <TITLE>
            Taschenrechner mit Forms
```

```
      </TITLE>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/rechnerMitForm.js">

      </SCRIPT>
</HEAD>
<BODY>
<H2>
      Diese html-Seite stellt einen Taschenrechner
      zur Verf&uuml;gung!
</H2>

<P>
       
</P>

<FORM NAME = "rechner">
      <TABLE BORDER>
            <TR>
                  <TD>
                        <B> Erster Operand </B>
                  </TD>
                  <TD>
                        <INPUT TYPE = text NAME = "opEins" SIZE = 12 >
                  </TD>
            </TR>
            <TR>
                  <TD>
                        <B> Operator </B>
                  </TD>
                  <TD>
                        <INPUT TYPE = text NAME = "operator"  SIZE = 12">
                  </TD>
            </TR>
            <TR>
                  <TD>
                        <B> Zweiter Operand </B>
                  </TD>
                  <TD>
                        <INPUT TYPE = text NAME = "opZwei" SIZE = 12>
                  </TD>
            </TR>
            <TR>
                  <TD>
                        <B> Ergebnis </B>
                  </TD>
                  <TD>
                        <INPUT TYPE = text NAME = "ergebnis" SIZE = 12>
                  </TD>
            </TR>
            <TR>
                  <TD ALIGN = CENTER>
```

```
                        <INPUT TYPE = Button VALUE = "Berechnung" NAME=
"button1" onClick = rechne()></TD>
                  </TD>
                  <TD ALIGN = CENTER>
                        <INPUT TYPE = Button VALUE = "Zur&uuml;cksetzen"
NAME= "button2" onClick = stop()>
                  </TD>
            </TR>
      </TABLE>
      <HR>
</FORM>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: rechnerMitForm.js
function taschenrechner(operand1, operator, operand2)
{
      if (operator == "+")
      {
            return (parseFloat(operand1) + parseFloat(operand2));
      }
      if (operator == "-")
      {
            return (parseFloat(operand1) - parseFloat(operand2));
      }
      if (operator == "*")
      {
            return (parseFloat(operand1) * parseFloat(operand2));
      }
      if (operator == "/")
      {
            if (parseFloat(operand2) != 0)
            {
                  return (parseFloat(operand1) / parseFloat(operand2));
            }
            else
            {
                  return "TEILEN_DURCH_NULL";
            }
      }
      return "FALSCHER_OPERATOR"
}

function rechne()
{
      if(isNaN (document.rechner.opEins.value))
      {
            alert("Erster Operand ist keine Zahl");
            return;
      }
      if(isNaN (document.rechner.opZwei.value))
      {
```

```
            alert("Zweiter Operand ist keine Zahl");
            return;
      }
      var ergebnis;
      ergebnis = taschenrechner (document.rechner.opEins.value,
                              document.rechner.operator.value,
                              document.rechner.opZwei.value);
      if(ergebnis == "TEILEN_DURCH_NULL")
      {
            alert("Versuch durch 0 zu teilen");
            return;

      }
      if(ergebnis == "FALSCHER_OPERATOR")
      {
            alert("Falschen Operator eingegeben!");
            return;
      }
      document.rechner.ergebnis.value = ergebnis;
}

function stop()
{
      document.rechner.opEins.value = "";
      document.rechner.operator.value = "";
      document.rechner.opZwei.value = "";
      document.rechner.ergebnis.value = "";
}
```

In Beispiel 12.1 wird im *BODY* der html-Datei zunächst ein Formular definiert. Das Formular erhält über das *NAME*-Tag den Namen “rechner”. Das form-Objekt ist eine Eigenschaft des document-Objekts (wie z.B. auch location oder URL vgl. Kapitel 9). Das in der html-Datei enthaltene Formular ist jetzt vermittels:

```
document.rechner
```

von JavaScript aus ansprechbar. In dem Formular selber werden Eingabetextfelder (Elemente des Formulars) definiert. Auch diese Felder erhalten über das *NAME*-Tag Namen. Alle Formular-Elemente (also auch die in Beispiel 12.1 definierten Eingabetextfelder) sind Eigenschaften des form-Objekts (welches selber wieder eine Eigenschaft des document-Objekts war). Daher ist also das Eingabetextfeld mit dem Namen “ergebnis” vermittels:

```
document.rechner.ergebnis
```

von JavaScript aus ansprechbar. Die Eingabetextelemente ihrerseits besitzen wieder Eigenschaften (es sind schließlich Objekte). Die wichtigste Eigenschaft der Eingabetextfelder ist die Eigenschaft *value*. *value* enthält den derzeitigen Wert des Eingabetextfeldes. Mit der Syntax:

```
meineVariable = document.rechner.ergebnis.value;
```

kann also der Wert des Eingabetextelementes “ergebnis” gelesen und vermittels

```
document.rechner.ergebnis.value = meinWert;
```

geschrieben werden.

Darüberhinaus enthält das Formular 2 Buttons. Buttons besitzen den Event-Handler *onClick*, der (natürlich) dann ausgelöst wird, wenn der Benutzer auf den Button klickt. In Beispiel 12.1 ruft der *onClick*-Event-Handler die Funktion *rechne()* auf. *rechne()* überprüft zunächst, ob die in den Eingabetextfeldern eingegebenen Werte Zahlen sind. Dazu wird die Methode *isNaN* des window-Objekts benutzt (*isNaN*[18] *(document.rechner.opEins.value)*).

Ist dies sichergestellt, wird die Funktion *taschenrechner()* mit den in die Eingabetextelemente eingegebenene Werten aufgerufen. *taschenrechner()* führt die Berechnung durch und gibt das Ergebnis zurück. Das Ergebnis wird dann in das dafür vorgesehene Formular-Element geschrieben:

```
document.rechner.ergebnis.value = ergebnis;
```

Klickt der Benutzer auf “Zurücksetzen”, wird über den *onClick*-Event-Handler die Funktion *stop()* aufgerufen. *stop()* setzt einfach alle Eingabetextelemente auf den Leerstring[19].

JavaScript ist aber auch im Zusammenhang mit Formularen, die an den Server übertragen werden sollen, interessant. Ebenso, wie die in Beispiel 12.1 genutzten einfachen Buttons, besitzen auch die *Submit*-Buttons den Event-Handler *onClick*. Gibt *onClick true* zurück, wird das Formular übertragen, gibt *onClick false* zurück, wird das Klicken auf den *Submit*-Button ignoriert. Dadurch können auf einfache Weise Eingabeüberprüfungen vorgenommen werden. Gibt ein Benutzer z.B. den 45.14.2089 in ein Datumsfeld eines Formulars ein und will anschließend das Formular an den Server übertragen, so ist es wesentlich sinnvoller, wenn dieser Fehler vor der Übertragung auf dem Client entdeckt wird, als wenn das Formular übertragen wird und erst der Server den Fehler entdeckt.

Beispiel 12.2 Fehlerkontrolle vor der Übertragung eines Formulars

```
<!-- Dateiname formTest.html -->
<HTML>
<HEAD>
        <TITLE> Form mit &Uuml;berpr&uuml;fung </TITLE>
        <SCRIPT LANGUAGE = "JavaScript"
                SRC = "./javaScript/feldKontrolle.js">

        </SCRIPT>
        <SCRIPT LANGUAGE = "JavaScript"
                SRC = "./javaScript/formTest.js">

        </SCRIPT>
</HEAD>

<BODY>
```

18.isNaN = is not a number

19.Dasselbe würde man natürlich auch erreichen, indem man den Button in html als Rest-Button definiert.

```
<H2>
       Dies Programm &uuml;berpr&uuml;ft eine Form!
</H2>

<P>
<FORM NAME = "tester" action="mailto:Bernd.Bluemel@fh-bochum.de"
                      method=post>
       <TABLE BORDER>
              <TR>
                     <TD>
                            <B> Artikelname: </B>
                     </TD>
                     <TD>
                            <INPUT TYPE = text NAME = "artikel" SIZE = 12 >
                     </TD>
              </TR>
              <TR>
                     <TD>
                            <B> Anzahl: </B>
                     </TD>
                     <TD>
                            <INPUT TYPE = text NAME = "anzahl" SIZE = 12 >
                     </TD>
              </TR>
              <TR>
                     <TD ALIGN = CENTER>
                            <INPUT TYPE = Submit VALUE = "&Uuml;bersenden"
                            NAME= "button1" onClick = "return teste()">
                     </TD>
                     <TD ALIGN = CENTER>
                            <INPUT TYPE = Reset VALUE = "Zur&uuml;cksetzen"
                            NAME= "button2"
                            onClick = "return confirm('Wirklich alles löschen?')">
                     </TD>
              </TR>
       </TABLE>
       <HR>
</FORM>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Dateien

```
// Dateiname : formTest.js

function teste()
{
      var fehlermeldungAnzahl= "Anzahl falsch eingegeben (keine Zahl)!";
      var fehlermeldungArtikel= "Kein Artikel eingegeben!";
      if (istKeineZahl (document.tester.anzahl, fehlermeldungAnzahl))
      {
            return false;
      }
      if ( istLeer(document.tester.artikel, fehlermeldungArtikel))
      {
            return false;
      }
      return true;
}

// Dateiname : feldKontrolle.js
```

```
function istKeineZahl (feld, fehlermeldung)
{
      if (isNaN(feld.value) ||  (feld.value == ""))
      {
            feld.focus();
            alert(fehlermeldung);
            return true;
      }
      return false;
}

function istLeer (feld, fehlermeldung)
{
      if (feld.value == "")
      {
            feld.focus();
            alert(fehlermeldung);
            return true;
      }
      return false;
}
```

In Beispiel 12.2 wird im *BODY* ein Formular definiert, über die Artikel bestellt werden sollen. Der Benutzer muß Artikelname und Anzahl eingeben und kann danach vermittels Klicken auf den *Submit*-Button den Inhalt des Formulars an den Server übersenden, welcher es dann per email an mich weiterleitet. Nun ist die Übersendung dieses Formulars aber nur dann sinnvoll, wenn die eingegebene Anzahl auch eine Zahl ist (daß ein Bestellformular ohne Adresse auch nur bedingt sinnvoll ist, ignorieren wir einfach). Bevor das Formular also übertragen wird, wird vermittels des *onClick*-Event-Handlers der Rückgabewert der Funktion *teste()* an das form-Objekt übergeben. *teste()* überprüft, ob im Feld für Anzahl eine Zahl steht und ob das Feld für Artikelname nicht leer ist.

Dazu bedient sich teste() der Hilfsfunktionen *istKeineZahl()* und *istLeer()*. *istKeineZahl()* überprüft, ob der Inhalt des ihm als ersten Parameter übergebenen Feldes keine Zahl oder leer[20] ist. Ist dies der Fall (man beachte hier die umgedrehte Logik: "der Fall sein" bedeutet keine Zahl), erhält das Eingabefeld den Fokus (*feld.focus()*), die als zweiter Parameter übergebene Fehlermeldung wird ausgegeben (hier "Anzahl falsch eingeben: (keine Zahl)!) und die Funktion gibt *true* zurück. Ist dies nicht der Fall, gibt *istKeineZahl() false* zurück.

Kommt von *istKeineZahl() true*, gibt *teste() false* zurück, das Formular wird nicht übertragen, der Benutzer sieht die Fehlermeldung. Das Feld Anzahl hat bereits den Fokus (durch *istKeineZahl()*) und der Benutzer kann seine Eingaben korrigieren.

Antwortet *istKeineZahl()* mit *false*, ruft *teste() istLeer()* auf, um den Inhalt des Artikelfeldes zu überprüfen. *istLeer()* überprüft nur, ob das ihm übergebene

20.Erstaunlicherweise ist für JavaScript ein leeres Feld ein Feld mit einer Zahl.

Feld leer ist. Ist dies der Fall, wird die als zweiter Parameter übergebene Fehlermeldung ausgegeben, das überprüfte Feld erhält den Fokus und die Funktion gibt `true` zurück. Ansonsten gibt `istLeer()` `false` zurück.

Kommt von `istLeer()` `true`, gibt `teste()` `false` zurück, das Formular wird nicht übertragen, der Benutzer sieht die Fehlermeldung. Das Feld Artikel hat bereits den Fokus und der Benutzer kann seine Eingaben korrigieren.

Antwortet `istLeer()` mit `false`, gibt `teste()` `true` zurück und das Formular wird übertragen.

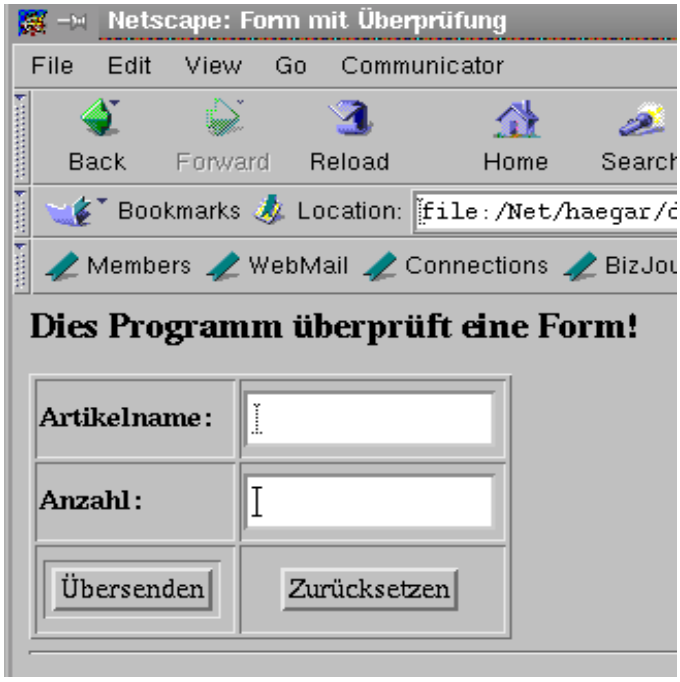


Abbildung 12. 2 Das zu überprüfende Formular am Bildschirm

Darüberhinaus sehen wir hier die Aufsplittung der Logik in zwei allgemeine Funktionen (`istKeineZahl()` und `istLeer()`), die in einer eigenen Datei abgespeichert werden und so für jedes Formular zur Verfügung stehen und der Funktion `teste()`. `teste()` macht ohne das zu spezielle zu überprüfende Formular keinen Sinn, da `teste()` ja den Namen des Formulars und der Eingabefelder nutzt.

Beispiel 12.2 zeigt darüberhinaus eine Nutzung des `onClick`-Event-Handlers des `Reset`-Buttons. Klicken des `Reset`-Buttons setzt alle Elemente des Formulars auf den Leerstring. Insbesondere bei großen Formularen macht es Sinn, noch einmal nachzufragen, ob der Benutzer auch wirklich alle seine Eingaben löschen will. Der `onClick`-Event-Handler des `Reset`-Buttons funktioniert völlig analog zum `onClick`-Event-Handler des `Submit`-Buttons. Gibt er `true` zurück, wird die Aktion durchgeführt, ansonsten nicht. In Beispiel 12.2 blenden wir nach dem Klicken auf den `Reset`-Button ein `Confirm`-Fenster auf. Klickt der Benutzer hier auf `OK`, gibt `confirm()` `true` zurück, die Inhalte der Felder werden gelöscht. Klickt der Benutzer hingegen auf `Cancel`, wird die Aktion abgebrochen und die Eingaben bleiben erhalten.

13 Zusammenfassendes Beispiel: Formulare, Stringbehandlung, Events

In diesem Kapitel wollen wir das bisher Gelernte an einem größeren Beispiel verdeutlichen. In diesem Beispiel benutzen wir:

- Methoden der String-Klasse für Ein- und Ausgabeaufbereitung.
- Ein Formular und Event-Handler für die Benutzerinteraktion.

Ziel dieses Beispiels ist es, eine html-Seite zur Verfügung zu stellen, die DM-Beträge in Dollar oder Dollar-Beträge in DM umrechnet. Dabei soll jeweils der entsprechende Euro-Betrag mitausgegeben werden. Die zu erstellende html-Seite ist in Abbildung 8. 1 dargestellt:

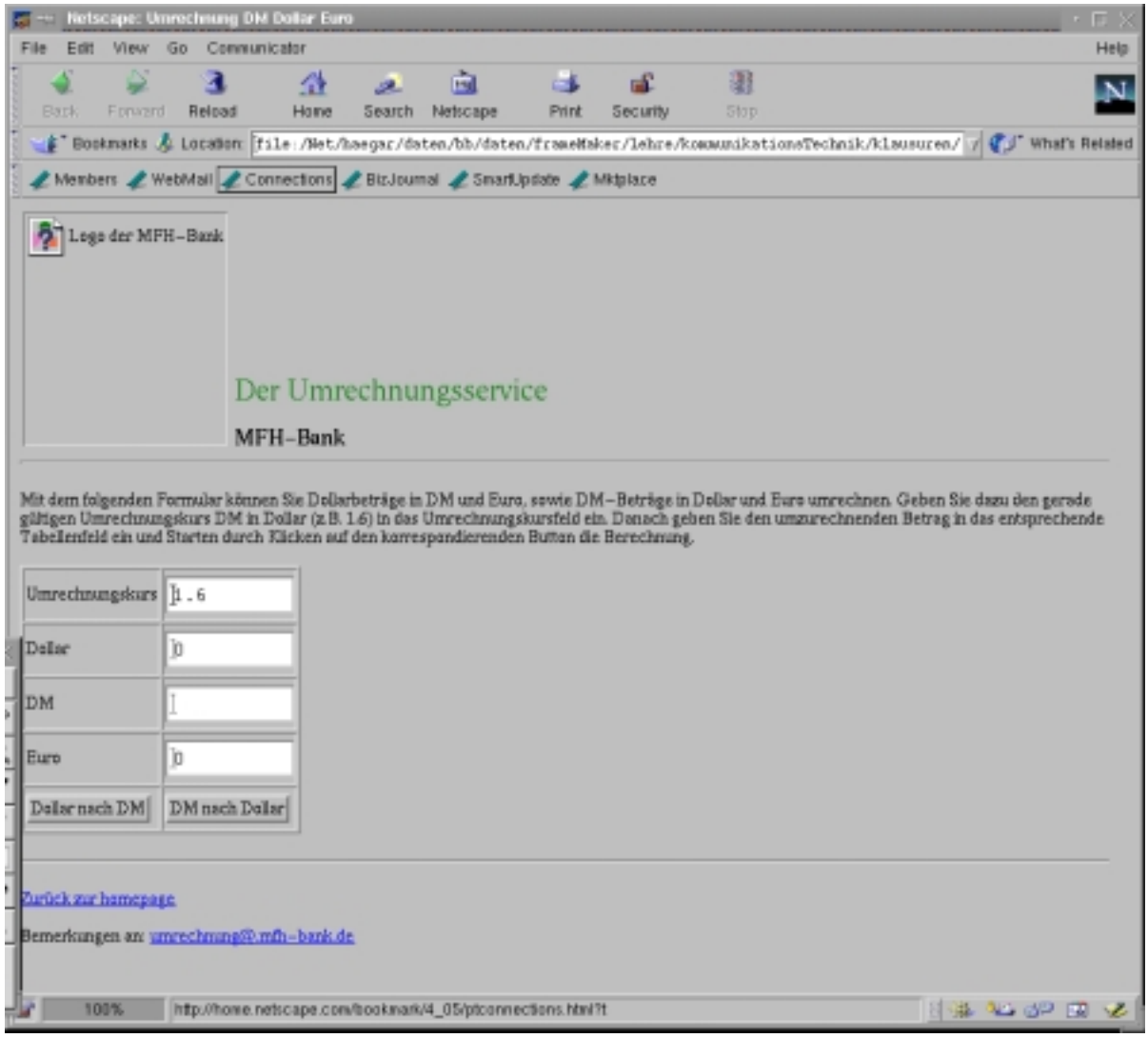


Abbildung 13. 1 html-Seite zur DM Dollar-Umrechnung

Folgende Logik soll implementiert werden:

Klickt der Benutzer auf den Button “Dollar nach DM” soll der in das Dollar-Feld eingetragene Betrag mit dem in das Feld Umrechnungskurs eingetragenen Umrechnungskurs nach DM umgerechnet werden. Der korrespondierende Euro-Betrag soll ebenfalls berechnet werden. Die Ergebnisse sollen in den jeweiligen Feldern der Tabelle eingeblendet werden. Ist die Eingabe im Umrechnungskurs-Feld oder im Dollar-Feld keine Zahl, soll ein Fehlerfenster über dem Browser geöffnet werden. Berechnungen sollen dann selbstverständlich nicht durchgeführt werden.

Klickt der Benutzer auf den Button “ DM nach Dollar” soll der in das DM-Feld eingetragene Betrag mit dem in das Feld Umrechnungskurs eingetragenen Umrechnungskurs nach Dollar umgerechnet werden. Der korrespondierende Euro-Betrag soll ebenfalls berechnet werden. Die Ergebnisse sollen in den jeweiligen Feldern der Tabelle eingeblendet werden. Ist die Eingabe im Umrechnungskurs-Feld keine Zahl, soll ein Fehlerfenster über dem Browser geöffnet werden. Berechnungen sollen dann nicht durchgeführt werden. Ist die Eingabe im Feld DM keine Zahl, soll zunächst geprüft werden, ob im Euro-Feld eine Zahl eingegeben wurde. Ist dies der Fall, wird der Euro-Betrag zunächst in DM und dann vermittels des eingegebenen Umrechnungskurses in Dollar umgerechnet. Die Ergebnisse werden in den jeweiligen Feldern der Tabelle eingeblendet werden. Ist dies nicht der Fall, soll ein Fehlerfenster über dem Browser geöffnet werden. Berechnungen sollen dann nicht durchgeführt werden.

Zahleneingaben sollen sowohl mt einem Punkt als Dezimaltrennzeichen (der Voreinstellung in JavaScript) als auch mit einem Komma als Trennzeichen möglich sein. Alle Ausgaben sollen ein Komma als Dezimalternnzeichen beinhalten, Eingaben mit Dezimalpunkt werden nach der Berechnung mit einem Dezimalkomma überschrieben.

Beispiel 13.1 DM Dollar-Umrechnung mit Euro-Ausgabe

```
<!-- Dateiname: euroDMForms.html -->
<HTML>
<HEAD>
       <TITLE>
              Umrechnung DM Dollar Euro
       </TITLE>

       <SCRIPT LANGUAGE = "JavaScript"
                     SRC = "./javaScript/einAusgabeBearbeitung.js">
       </SCRIPT>
       <SCRIPT LANGUAGE = "JavaScript"
                     SRC = "./javaScript/euroDMForms.js">
       </SCRIPT>

</HEAD>

<BODY >
    <TABLE BORDER="0">
      <TR>
        <TD WIDTH="15%"><IMG SRC="./bilder/MFH-BankLogo.gif"
        ALT="Logo der MFH-Bank" WIDTH="164" HEIGHT="189">
        </TD>
        <TD WIDTH="70%" ALIGN="LEFT" VALIGN="BOTTOM">
        <FONT COLOR="#228B22" SIZE="+4">
                       Der Umrechnungsservice
              </FONT>

        <P><FONT COLOR="#0000FF" SIZE="+2"><FONT COLOR="#000000">
              MFH-Bank
```

```
        </FONT></P>
        </TD>
      </TR>
    </TABLE>
    <HR >


    <P>
       Mit dem folgenden Formular können Sie Dollarbeträge in DM und Euro, sowie
       DM-Beträge in Dollar und Euro umrechnen. Geben Sie dazu den gerade g&uuml;ltigen
       Umrechnungskurs DM in Dollar (z.B. 1.6) in das Umrechnungskursfeld ein.
       Danach geben Sie den umzurechnenden Betrag in das entsprechende
       Tabellenfeld ein und Starten durch Klicken auf den korrespondierenden
       Button die Berechnung.
    </P>
    <FORM NAME = "umrechnung">
         <TABLE BORDER="1">

              <TR>
                     <TD>Umrechnungskurs</TD>
                     <TD><INPUT TYPE = "text" NAME = "dollarkurs" SIZE = 10></TD>

              </TR>
              <TR>
                     <TD>Dollar</TD>
                     <TD><INPUT TYPE ="text" NAME = "dollars"
                                   onChange = dollar2dm()  SIZE = 10 ></TD>
              </TR>
              <TR>
                     <TD>DM</TD>

                     <TD><INPUT TYPE = "text" NAME = "dms"
                                   onChange = dm2dollar()  SIZE = 10></TD>
              </TR>
              <TR>
                     <TD>Euro</TD>

                     <TD><INPUT TYPE = "text" NAME = "euros"  SIZE = 10></TD>
              </TR>
              <TR>
                     <TD><INPUT TYPE = "button" NAME = "button1"
                                   VALUE = "Dollar nach DM"
                                   onClick = dollar2dm()>
                     <TD><INPUT TYPE = "button" NAME = "button2"
                                   VALUE = "DM nach Dollar"
                                   onClick = dm2dollar()>
              </TR>
         </TABLE>
    </FORM>

    <HR>
    <P>
       <A HREF="./welcome.html"> Zur&uuml;ck zur homepage</A>
    </P>
    <P>
       Bemerkungen an:
              <A HREF="mailto:umrechnung@.mfh-bank.de">
                                          umrechnung@.mfh-bank.de</A>
    </P>

</BODY>
</HTML>
```

Die zugehörigen JavaScript-Dateien

```
// Dateiname: einAusgabeBearbeitung.js

function schoeneAusgabe (ausgabeWert)
{
	if (schoeneAusgabe.arguments.length != 1)
	{
		alert ("Funktion wurde mit falscher Anzahl Argumente aufgerufen");
		return 0
	}
	var ausgabe;
	var b;
	ausgabe = new String (ausgabeWert);
	b = ausgabe.indexOf (".");
	if (b == -1)
	{
		ausgabe = (ausgabe+ ",00"); // keine Nachkommastellen
	}
	else
	{
		if ((ausgabe.substring(b+1,ausgabe.length)).length == 1)
		{
			ausgabe =
(ausgabe.substring(0,b)+","+ausgabe.substring(b+1,b+2)+"0");
		}
		else
		{
			ausgabe =(ausgabe.substring(0,b)+","+ausgabe.substring(b+1,b+3));
		}

	}
	return ausgabe;
}


function komma2Punkt (umwandelWert)
{
	if (komma2Punkt.arguments.length != 1)
	{
		alert ("Funktion wurde mit falscher Anzahl Argumente aufgerufen");
		return 0
	}
	var punkt;
	var a;
	punkt = new String (umwandelWert);
	a = punkt.indexOf (",");
	if (a != -1)
	{
		punkt = punkt.substring (0,a)+"."+punkt.substring (a+1,punkt.length);
	}
	return punkt;
}

// Dateiname: euroDMForms.js

function ausgeben(dollarKurs, dollars, dms, euros)
{
	document.umrechnung.dollarkurs.value = schoeneAusgabe(dollarKurs)
	document.umrechnung.dollars.value = schoeneAusgabe (dollars)
	document.umrechnung.dms.value = schoeneAusgabe(dms);
	document.umrechnung.euros.value = schoeneAusgabe(euros);
```

```
}


function dollar2dm()
{
      var euroKurs = 1.96;
      var dollarKurs;
      var euros;
      var dollars;
      var dms;
      dollarKurs = komma2Punkt(document.umrechnung.dollarkurs.value);
      dollars = komma2Punkt(document.umrechnung.dollars.value);

      if (isNaN(dollarKurs) || isNaN(dollars) ||
            (dollarKurs == "") ||(dollars == ""))
      {
            alert("Umrechnungskurs und Dollars  " +
                              "muessen Zahlen sein!!");
      }

      else
      {
            dms = dollars * dollarKurs;
            euros = dms / euroKurs ;
            ausgeben(dollarKurs, dollars, dms, euros);
      }
}


function dm2dollar()
{
      var euroKurs = 1.96;
      var dollarKurs;
      var euros;
      var dollars;
      var dms;
      dollarKurs = komma2Punkt(document.umrechnung.dollarkurs.value);
      if (isNaN(dollarKurs) || (dollarKurs == ""))
      {
            alert("Umrechnungskurs  " +
                              "muss eine Zahl sein!!");
      }

      else
      {
            if (isNaN(komma2Punkt(document.umrechnung.dms.value)) ||
                        (document.umrechnung.dms.value == ""))
            {
                  if(isNaN(komma2Punkt(document.umrechnung.euros.value)) ||
                        (document.umrechnung.euros.value == ""))
                  {

                        alert("DMs oder Euros muessen eine Zahl sein!");
```

```
                }
                else
                {
                        euros =
komma2Punkt(document.umrechnung.euros.value);
                        dms = euros * euroKurs;
                        dollars = dms / dollarKurs ;
                        ausgeben(dollarKurs, dollars, dms, euros)
                }
            }
            else
            {
                dms = komma2Punkt(document.umrechnung.dms.value);
                euros = dms / euroKurs;
                dollars = dms / dollarKurs ;
                ausgeben(dollarKurs, dollars, dms, euros);


            }
        }
}
```

In Abbildung 8. 1 bauen wir zunächst, wie ja bereits in Kapitel 12 behandelt, ein Formular in einer html-Seite auf. Eingebettet in das Formular sind Eingabe-Felder und Buttons. Die Buttons sind über den Event-Handler *onClick* mit JavaScript-Funktionen verbunden. Neu ist der Event-Handler *onChange*, über den ebenfalls JavaScript-Funktionen aufgerufen werden können. *onChange* wird ausgelöst, wenn der Benutzer die Eingabe eines Feldes beendet hat.

Wird ein Event-Handler ausgelöst, muß zuerst überprüft werden, ob die Eingabewerte Zahlen sind. Da JavaScript den Dezimalpunkt erwartet, wir aber ausdrücklich auch ein Dezimalkomma gestatten wollen, muß zunächst ein etwaiges Dezimalkomma in einen Dezimalpunkt umgewandelt werden. Dafür ist die Funktion *komma2Punkt* verantwortlich. Sie wandelt das erste Komma im ihr übergebenen Argument in einen Punkt um.

Dabei wenden wir Methoden der Stringklasse an:

```
punkt.indexOf (",")
```

gibt als Ergebnis die Stelle zurück, an der das erste Komma im String *punkt* steht. Enthält der String kein Komma, wird -1 zurückgegeben. In diesem Fall macht unsere Funktion gar nichts. Gibt es ein Komma, wird dies in einen Punkt umgewandelt:

```
punkt = punkt.substring (0,a)+"."+ punkt.substring (a+1,punkt.length);
```

Hierbei wird die String-Methode *substring* genutzt. *substring* extrahiert einen Teilstring aus dem String-Objekt. Der Teilstring beinhaltet den Character am ersten *substring* übergebenen Parameter, sowie alle Character zwischen dem ersten und dem zweiten Parameter, den Character am zweiten übergebenen Parameter allerdings **nicht**.

Danach werden die notwendigen Berechnungen durchgeführt. Da dies hier analog zu Beispiel 12.1 geschieht, werde ich darauf nicht weiter eingehen. Vor der Ausgabe in das Formular (dies geschieht hier durch die Funktion *ausgeben* und ist ebenfalls ana-

log zu Beispiel 12.1) müssen etwaige Dezimapunkte in Dezimalkommas umgewandelt werden und überflüssige Stellen abgeschnitten werden. Dafür ist die Funktion *schoeneAusgabe* verantwortlich. Diese Funktion benutzt zunächst *indexOf (".")*, um festzustellen, ob ein Dezimalpunkt vorhanden ist. Ist dieser nicht vorhanden, gibt es keine Nachkommastellen und ,00 wird angehängt[21].

Gibt es einen Dezimalpunkt wir dieser mittels der *substring*-Methode ersetzt. Hierbei wird noch unterschieden, ob es eine oder mehr Nachkommastellen gibt. Bei einer Nachkommastelle wird eine 0 angefügt. Sind es 2 oder mehr, werden die über 2 hinausgehenden Stellen abgeschnitten.

Wir sehen auch hier wieder die Trennung der Logik in allgemeine Funktionen und Funktionen, die eng an die html-Seite gekoppelt sind (indem sie die Namen der Eingabefelder benutzen).

21.Dezimalkommas können bei der Ausgabe nicht vorkommen, da etwaige eingegebene Dezimalkommas in Dezimalpunkte umgewandelt wurden und Ergebnisse von Berechnungen sowieso nur Dezimalpunkte enthalten.

14 Das Image-Objekt

Über das Image-Objekt können wir Bilder im Browserfenster manipulieren. Im einfachsten Fall können wir, wenn der Benutzer die Maus über ein Bild führt, das Bild gegen ein anderes Bild austauschen. Wir können aber auch Animationen programmieren. Dies wird in den folgenden Beispielen veranschaulicht. Wir starten mit dem schwierigeren Fall und schauen uns an, wie Animationen in JavaScript programmiert werden.

Beispiel 14.1 Eine Animation

```
<!-- Dateiname: animation.html -->
<HTML>
<HEAD>
      <TITLE>
            Bilder austauschen
      </TITLE>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/animation.js">
      </SCRIPT>
</HEAD>

<BODY>
      <H2>
            Dies Programm animiert Bilder!
      </H2>

      <IMG SRC = "./bilder/1.gif" NAME = anim>



      <FORM>
            <INPUT TYPE = button VALUE = "Start"
                  onClick = "start()">

            <INPUT TYPE = button VALUE = "Stop"
            onClick = "stop()">
      </FORM>
      <SCRIPT LANGUAGE = "JavaScript">
            animate();
      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname: animation.js
var i;
var j;
bilder = new Array (2);
for(i = 0; i <= 1; i++)
{
      bilder[i] = new Image();
      j = i + 1;
```

```
      bilder[i].src = "./bilder/" + j + ".gif";
}


var zaehler = 0;
var timeout_id = null;

function start()
{
      if (timeout_id == null)
      {
            animate()
      }
}

function stop()
{
      if (timeout_id)
      {
            clearTimeout(timeout_id)
      }
      timeout_id = null;
}

function animate()
{
      document.anim.src = bilder[zaehler].src;
      zaehler = (zaehler + 1) % 2;
      timeout_id = setTimeout("animate()", 500);
}
```

Wenn die html-Seite geladen wird läuft zunächst ein kleines Script ab. Zwei Variablen werden deklariert und initialisiert (*zaehler* und *timeout_id*, beide auf 0 bzw. *null*). Dann wird ein Array mit zwei Elementen deklariert (*bilder*). In der nun folgenden Schleife werden 2 Objekte vom Typ Image erzeugt und auf den Elementen des Arrays abgelegt *(bilder[i] = new Image()*). Danach wird die *src*-Eigenschaft der Bilder gesetzt. Die *src*-Eigenschaft gibt an, welches Bild geladen werden soll (der Dateiname des Bildes). Die Bilder, die hier dargestellt werden sollen, heißen 1.gif, bzw 2.gif und stehen im Verzeichnis *bilder* unterhalb der html-Datei (*bilder[i].src = "./bilder/" + j + ".gif"*). In dem Moment, wo die *src*-Eigenschaft gesetzt wird, wird das zugehörige Bild über das Netz kopiert und in den Cache des Browsers geladen. Nach Beendigung der *for*-Schleife sind also beide Bilder vom Server herunterkopiert und lokal im Cache vorhanden (werden aber noch nicht dargestellt).

Danach wird die Funktion *animate()* definiert. Sie wird später besprochen, wenn sie von der Animation aufgerufen wird.

Im <*BODY*>-Teil der html-Datei wird zunächst (mit dem *IMG*-Tag) ein Bild geladen und dargestellt (in Wahrheit wird das Bild nicht mehr geladen, denn es ist das Bild 1.gif, welches ja bereits im Cache des Browsers ist). Durch das *IMG*-Tag geladene Bilder sind ebenfalls JavaScript-Objekte (nämlich Image-Objekte). Sie sind Eigenschaften des document-Objekts. Da wir dem Image-Objekt mit dem *NAME*-Tag einen Namen ge-

geben haben (völlig analog dem Formular-Beispiel Beispiel 12.1) ist das Image-Objekt jetzt von JavaScript-Code zugreifbar und zwar unter dem Namen:

```
document.anim
```

Dann wird eine ganz einfaches Formular erstellt, die nur zwei Elemente enthält, 2 Buttons, deren einer die Aufschrift “start” und deren anderer die Aufschrift “stop” erhält. Für beide ist ein *onClick*-Event-Handler definiert, der die Animation startet, bzw. anhält.

Klickt der Benutzer auf den Start-Button, wird zunächst überprüft, ob die Variable *timeout_id* den Wert *null* hat. Beim ersten Start des Programms ist das so. Daher wird die Funktion *animate()* aufgerufen. Diese Funktion führt die eigentliche Animation durch.

animate() setzt zunächst die *src*-Eigenschaft des im Browser dargestellten Bildes (*document.anim.src = bilder[zaehler].src*). Wird die *src*-Eigenschaft eines im Browserfenster dargestellten Bildes verändert, so wird sofort das im Browser dargestellte Bild gegen das Bild aus der neuen Datei ausgetauscht. Nun hat *zaehler* aber den Wert 0 und im ersten Element des Bilder-Arrays ist 1.gif abgelegt. Dieses Bild wird aber sowieso gerade dargestellt, so daß sich nichts ändert.

In der nächsten Zeile wird *zaehler* um 1 modulo 2 hochgezählt (*zaehler = (zaehler + 1) % 2*). *zaehler* hat jetzt also den Wert 1 (*zaehler* nimmt im weiteren Verlauf abwechselnd die Werte 0 und 1 an).

Danach kommt die trickreichste Anweisung. Es wird die Methode *setTimeout* des window-Objekts aufgerufen. *setTimeout* gibt einen Identifier (*timeout_id*) zurück mit der die ganze Animation später gestoppt werden kann (wir wollen ja keine Endlos-Sachen programmieren). *setTimeout* erwartet zwei Parameter. Der zweite gibt an, wielange die Ausführung des JavaScript-Programms angehalten werden soll (in Millisekunden), der erste, welche Funktion dann aufgerufen wird. Nach der Zeile

```
timeout_id = setTimeout(“animate()”, 500)
```

wird die Ausführung von *animate()* also 500 Millisekunden angehalten. Danach wird *animate()* erneut aufgerufen. Jetzt ist zaehler allerdings 1, so daß *document.anim.src* jetzt den Wert 2.gif zugewiesen bekommt. Dies bedeutet, daß das Bild, welches im Browser dargestellt wurde (1.gif), gegen 2.gif ausgetauscht wird. Danach wird *zaehler* wieder um 1 modulo 2 erhöht (ist damit wieder 0) und die Anwendung wartet wieder 500 Millisekunden, ruft wieder *animate()* auf, wodurch 2.gif wieder gegen 1.gif ausgetauscht wird. Dies läuft jetzt endlos so weiter.

Abhilfe kann der genervte Benutzer nur schaffen, wenn er auf den Button mit der Aufschrift “Stop” klickt. Hier wird nämlich durch den *onClick*-Event-Handler dieses Buttons zunächst überprüft, ob die Animation läuft (ob *timeout_id* einen Wert hat). Läuft die Animation wird sie mit der Methode *clearTimeout()* (des window-Objekts) gestoppt. Danach wird *timeout_id* wieder auf *null* gesetzt, damit der Benutzer, wenn er denn will, die Animation wieder starten kann.

Im nächsten Beispiel verwenden wir unsere bereits programmierte Animation in einem anderen Kontext:

Beispiel 14.2 Animation in einem Link

```
<!-- Dateiname: bildAustausch.html -->
<HTML>
      <HEAD>
      <TITLE> Bilder austauschen </TITLE>
      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/animation.js">
      </SCRIPT>
</HEAD>
<BODY>
      <H2>
            Dies Programm animiert Bilder!
      </H2>
      <A HREF= "http://www.mfh-iserlohn.de"
                  onMouseOver = "start()"
                  onMouseOut = "stop()">
            <IMG SRC = "./bilder/1.gif" NAME = anim>
      </A>


</BODY>
</HTML>
```

Im *BODY*-Teil ist ein Link definiert, der ausgelöst wird, wenn man auf ein Bild klickt. Dieses Bild ist ein Image-Objekt und damit wie in Beispiel 14.1 aus JavaScript ansprechbar. Führt der Benutzer die Maus über den Link startet die Animation durch den *onMouseOver*-Event-Handler (gleicher Code wie in Beispiel 14.1). Verläßt der Mauszeiger den Link, wird die Animation angehalten.

Beispiel 14.3 (Mouse-Rollover) vereinfacht Beispiel 14.2 (und verbessert es). Wenn die html-Seite geladen wird, wird das Bild 1.gif dargestellt. Führt der Benutzer die Maus über das Bild (identisch mit dem Link) wird das Bild gegen 2.gif ausgetauscht. Verläßt der Mauszeiger das Bild wieder (und damit den Link) wird wieder 1.gif dargestellt.

Beispiel 14.3 Austausch eines Bildes

```
<!-- Dateiname: bildAenderung.html -->
<HTML>
<HEAD>
      <TITLE>
            Bilder austauschen
      </TITLE>

      <SCRIPT LANGUAGE = "JavaScript"
                  SRC = "./javaScript/animation.js">
      </SCRIPT>
</HEAD>
<BODY>
      <H2>
            Dies Programm &auml;ndert das Aussehen eines grafischen Links!
      </H2>
```

```
      <A HREF= "http://www.mfh-iserlohn.de"
                  onMouseOver = "document.anim.src = bilder[1].src;"
                  onMouseOut = "document.anim.src = bilder[0].src;">
            <IMG SRC = "./bilder/1.gif" NAME = anim>
      </A>


</BODY>
</HTML>
```

In diesem Beispiel verwenden wir die Datei animation.js, obwohl wir weder *animate()* noch *start()* oder *stop()* verwenden. Wir benötigen aber den Code, der die Bilder in den Cache vom Browser lädt. Da wir dies aber in animation.js schon realisiert haben, verwenden wir es auch. Wir nehmen dabei geringfügig größere Ladezeiten in Kauf. Die Funktionen sind reiner ASCII-Code und daher wesentlich kleiner als die Bilder, die geladen werden müssen.

15 Das Navigator-Objekt

Das navigator-Objekt erlaubt es uns, Informationen über den Browser, mit dem unsere Seite betrachtet wird, zu erhalten. Dies beinhaltet Name des Browsers (z.B. Navigator, Explorer, Omniweb), Versionsnummer und genutzte Plattform (z.B. Macintosh, Linux, Nextstep, Solaris, Windows 95/NT). Dies ist z.B. sinnvoll, wenn man Konstrukte einsetzt, von denen man weiß, daß sie von manchen Browsern nicht unterstützt werden.

Solche Dinge kann man dann in *if*-Strukturen einbauen, um sie nicht von Browsern ausführen zu lassen, die damit nichts anfangen können.

Auch bei der Fehlerbehandlung ist dies sinnvoll. Wenn bei einem Benutzer eines unserer Programme Fehler auftreten, ist es ganz sinnig, wenn wir nicht nur die Fehlerbeschreibung und die Stelle im Code, wo der Fehler passierte, wissen, sondern auch welcher Browser, welche Version und welche Plattform. Denn manche Fehler passieren nicht auf jeder Plattform und auf jedem Browser, und manche Browser unterstützen sowieso nicht alle JavaScript-Funktionalitäten.

Beispiel 15.1 Fehlermeldung mit Fehlerursache, Plattform, Browser und Version

```
<!-- fehlerUebermittler.html -->
<HTML>
<HEAD>
      <TITLE>
            Programm mit Fehlerfenster
      </TITLE>
      <SCRIPT cLANGUAGE = "JavaScript"
                  SRC = "./javaScript/fehlerUebermittler.js">
      </SCRIPT>
</HEAD>

<BODY>
      <P>
            Gleich wird ein Fehlerfenster aufgehen!
      </P>
      <SCRIPT LANGUAGE = "JavaScript">
            { //Fehler oeffnende Klammer, keine schliessende
      </SCRIPT>
</BODY>
</HTML>
```

Die zugehörige JavaScript-Datei

```
// Dateiname fehlerUebermittler.js

function fehlerUebermittler(fehlermeldung, url, zeile)
{
// oeffnen eines neuen Fensters

      var fehlerFenster = window.open("","Fehler")
//                            "resizable, status, width=300, height=20");

// Der Schreibbereich des Fensters
```

```
      var fehlerDokument = fehlerFenster.document;

      fehlerDokument.write("<HTML> \n");
      fehlerDokument.write("<BODY> \n" );

      fehlerDokument.write("In der gerade geladenen Seite ist ein Fehler
aufgetreten!");
      fehlerDokument.write("Bitte Klicken Sie auf Absenden, damit wir den
Fehler beheben k&ouml;nnen!");

      // Einfuegen einer Form

      fehlerDokument.write('<FORM ACTION = "mailto:is@helga.mfh-iserlohn.de"
>');
      fehlerDokument.write('<TABLE BORDER>');
             fehlerDokument.write("<TR>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write("Ihre Mail-Adresse (optional)");
                fehlerDokument.write("</TD>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write('<INPUT Size = 50, Name =
"MailAdresse">');
                fehlerDokument.write("</TD>");
              fehlerDokument.write("</TR>");
              fehlerDokument.write("<TR>");
                fehlerDokument.write("<TD>");
                 fehlerDokument.write("Plattform- und Browser-Information");
                fehlerDokument.write("</TD>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write('<INPUT Size = 50, Name = "Version"
VALUE = "'+navigator.userAgent+'">');
                fehlerDokument.write("</TD>");
              fehlerDokument.write("</TR>");
              fehlerDokument.write("<TR>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write("html-Seite");
                fehlerDokument.write("</TD>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write('<INPUT Size = 50, Name = "Version"
VALUE = "'+url+'">');
                fehlerDokument.write("</TD>");
              fehlerDokument.write("</TR>");
              fehlerDokument.write("<TR>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write("Fehlermeldung");
                fehlerDokument.write("</TD>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write('<INPUT Size = 50, Name =
"Fehlermeldung" VALUE = "'+fehlermeldung+'">')
                fehlerDokument.write("</TD>");
              fehlerDokument.write("</TR>");
              fehlerDokument.write("<TR>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write("Zeile");
```

```
                fehlerDokument.write("</TD>");
                fehlerDokument.write("<TD>");
                 fehlerDokument.write('<INPUT Size = 50, Name = "Zeile" VALUE
= "'+zeile+'">');
                fehlerDokument.write("</TD>");
              fehlerDokument.write("</TR>");
              fehlerDokument.write("<TR>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write('<INPUT TYPE = "submit" VALUE =
"Absenden">');
                fehlerDokument.write("</TD>");
                fehlerDokument.write("<TD>");
                  fehlerDokument.write('<INPUT TYPE = "button" VALUE =
"Schlieflen" onClick = self.close()>');
                fehlerDokument.write("</TD>");
              fehlerDokument.write("</TR>");
      fehlerDokument.write("</TABLE>");
      fehlerDokument.write("</FORM>");
      fehlerFenster.defaultStatus = "Dies ist ein Fehlerfenster";
      fehlerDokument.write("</BODY> \n");
      fehlerDokument.write("</HTML> \n");

      return true;
}

self.onerror = fehlerUebermittler;
```

In Beispiel 15.1 wird zunächst die Funktion *fehlerUebermittler* definiert. *fehlerUebermittler* erstellt zunächst mit der Methode *window.open* ein neues Browser-Fenster.Dieses Fenster erhält den Namen *fehlerFenster*. Über diesen Namen ist das neue Browser-Fenster jetzt ansprechbar. Dann definieren wir eine Variable *fehlerDokument*. Sie enthält nun den Schreibbereich des Fensters. Dies geschieht, um Schreibaufwand zu sparen. *fehlerDokument.write* ist identisch mit *fehlerFenster.document.write*.

Sodann erzeugen wir im neuen Browser-Fenster ein Formular, das bei Klicken des *Submit*-Buttons automatisch an unsere e-Mail-Adresse geschickt wird (der Benutzer muß da selber nicht mehr aktiv werden).

Nun müssen wir noch sicherstellen, daß *fehlerUebermittler* im Fehlerfall auch aufgerufen wird. Hierzu gibt es den Event-Handler *onerror*. *onerror* ist ein Event-Handler des window-Objekts und wird aufgerufen, wenn im window-Objekt ein nicht behebbarer Fehler auftritt. Normalerweise wird dann die Default-Fehlerprozedur aufgerufen, die uns meldet, an welcher Stelle im Code welcher Fehler aufgetreten ist. Dies ist zur Fehlersuche beim Programmieren sinnvoll, aber nicht, wenn ein Benutzer unserer Internet-Seiten auf einen Fehler trifft. Dieser Benutzer hat ja keinen Zugriff auf unser Programm und kann den Fehler (selbst wenn er ein JavaScript-Programmierer ist) nicht beheben. Der *onerror* Event-Handler ist aber eine Eigenschaft des window-Objekts. Daher können wir ihm einen anderen Wert zuweisen. Dies geschieht in der Zeile:

```
window.onerror = fehlerUebermittler;
```

Der Event-Handler *onerror* wird mit den 3 Parametern Fehlermeldung, URL der Datei, in der der Fehler auftrat, und Zeile, in der der Fehler auftrat, aufgerufen. Diese Parameter tragen wir in unser Formular ein. Damit wissen wir, in welcher Zeile welcher html-Seite welcher Fehler auftrat.

Um nun die Browser--Information mitzuübertragen, verwenden wir die Property *navigator.userAgent. navigator.userAgent* ist eine Eigenschaft des navigator-Objekts und enthält genau die Informationen, die wir zusätzlich noch wissen wollen (Browser-Name, Version, Plattform, Betriebssystem-Version) als String.

Unsere Fehlerfunktion gibt zum Schluß true zurück. Dies sagt der JavaScript-Umgebung des Browsers, daß wir den Fehler behandelt haben.

Zum Schluß führen wir absichtlich einen Fehler herbei.

Beispiel 15.1 führt bei meinem Browser folgende Ausgabe herbei:

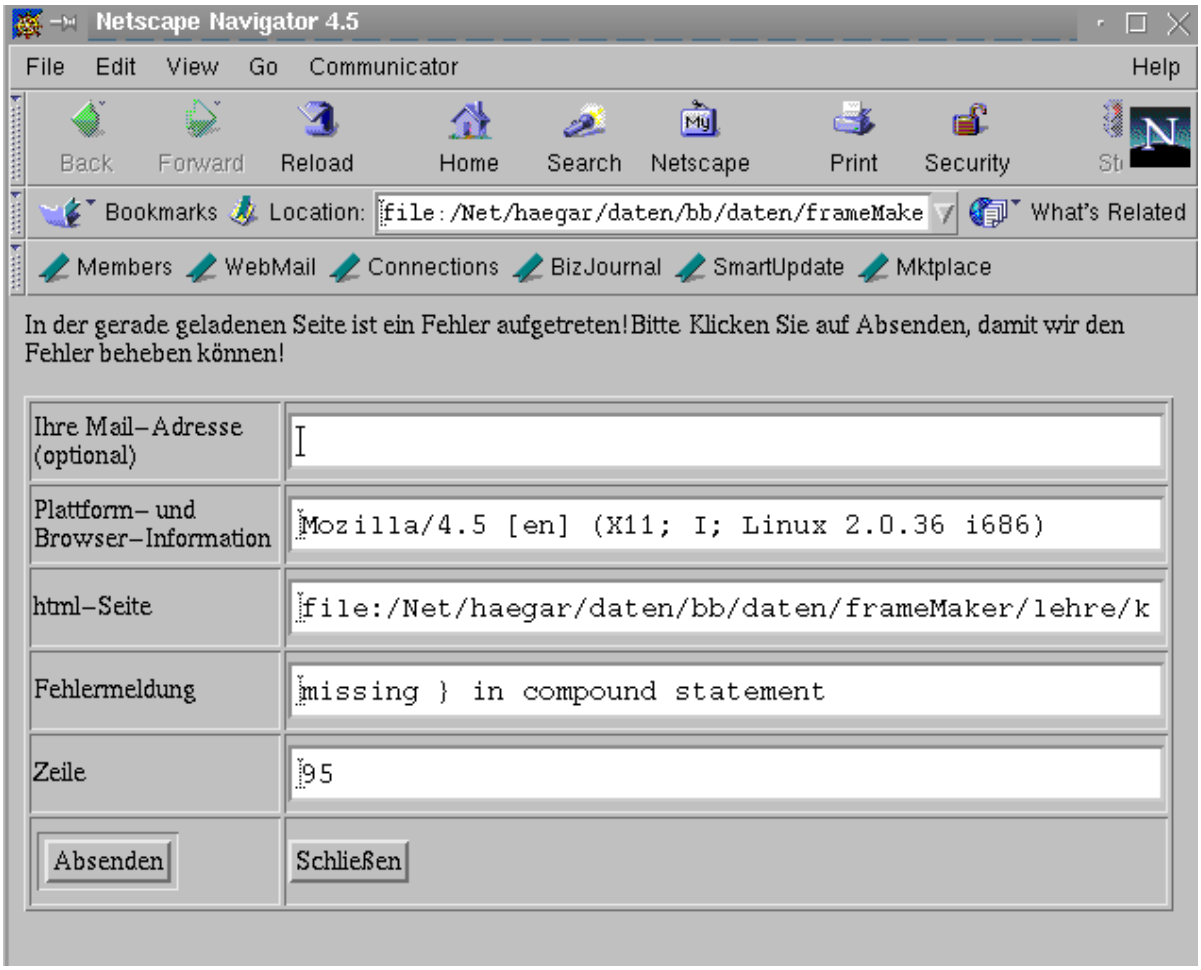


Abbildung 15. 1 Ausgabe von Beispiel 15.1

16 Frames

Wenn in einem Fenster Frames enthalten sind, kann man über die *frames[]*-Property des *window*-Objekts auf die Frames zugreifen. Man kann dann mittels *document.write()* genauso in die Frames schreiben wie in andere windows (vgl. Kapitel 15, Beispiel 15.1). *frames[0]* ist also das erste Frame eines windows. Gibt es in einem *window* keine Frames, so ist *frames[]* leer. Beinhaltet ein Frame weitere Frames, so ist über die frames[] property des Frames selber auf die Tochter-Frames zuzugreifen. *frames[1].frames[0]* ist also das zweite Tochterframe des ersten Frames des Fensters.

Jedes Frame wiederum besitzt die parent-Property, die auf das obergelagerte Frame oder window zeigt.

Mit JavaScript kann man dafür sorgen, daß Frames immer im Zusammenhang am Bildschirm erscheinen. Dies veranschauliche ich an einem Beispiel. Problematisch bei Frames sind unter anderem Bookmarks. Es kann vorkommen, daß Benutzer ein Bookmark auf ein Frame, statt auf das das Frame beinhaltende Fenster setzen.

Wird dann die Seite über das Bookmark geladen, sieht der Benutzer nur ein Frame, nicht, wie beabsichtigt, das ganze Frameset. Dies kann dazu führen, daß der Benutzer keine Navigation in unseren Seiten mehr hat (die Navigation ist bei Frames meist in einem eigenen Frame untergebracht. Mittels JavaScript kann man testen, ob solch ein Fall eingetreten ist und, wenn dies der Fall ist, das ganze Frame-Set laden.

Beispiel 16.1 Laden eines Frame-Sets

```
// Dateiname: frameRecognizer.js
var derzeitigeURL;
var letzterPunkt;
var bisHTML;
var neueURL;

if (parent  == self)
{
      derzeitigeURL = location.href;
      letzterPunkt = derzeitigeURL.lastIndexOf(".");
      bisHTML = derzeitigeURL.substring(0, letzterPunkt);
      neueURL= bisHTML + "_index.html";
      location.replace (neueURL);
}
```

Hier wird zunächst getestet ob die geladene Seite als Frame oder als window geladen wurde.

```
if (parent == self)
```

Ergibt dies false, ist alles klar, es gibt ein übergeordnetes Fenster, das Frame-Set wurde korrekt geladen. Ergibt dies jedoch true, ist die geladene Datei selbst das übergeordnete Fenster und damit kein Frame. Dann wird im if-Teil die URL des Frame-Sets zusammengesetzt und dann über *location.replace()* geladen. Wird diese JavaScript-Datei in den <HEAD>-Teil eines Frames eingebettet, passiert dies, bevor die Seite zu

laden beginnt. Der Benutzer merkt von der Überleitung nichts. Durch *location.replace()* wird die alte URL darüberhinaus aus der History gelöscht und durch die neue ersetzt.

17 Cookies (Beispiel eines Internet-Vertriebssystems)

Beispiel 17.1 Die html-Datei der Bestellseite

```
<!-- Dateiname: to_order.html -->
<HTML>
<HEAD>
<TITLE>net-m@n Bestellungen</TITLE>
<SCRIPT LANGUAGE = "JavaScript" SRC = "./javascript/stringSplitten.js">
</SCRIPT>
<SCRIPT LANGUAGE = "JavaScript" SRC = "./javascript/zahlAusgeben.js">
</SCRIPT>
<SCRIPT LANGUAGE = "JavaScript" SRC = "./javascript/bearbeiteCookie.js">
</SCRIPT>
</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#0000A0"
            ALINK="#0000A0" onUnload="if (!(parent == self)) setzeCookie()">
<SCRIPT LANGUAGE = "JavaScript" SRC = "./javascript/frameRecognizer.js">
</SCRIPT>
<H2>
<IMG SRC="./bilder/net-Bestellung.jpg" HEIGHT=51 WIDTH=342></H2>
<B>Kontrolle ist besser!</B>
<BR>Sie m&ouml;chten die in der Liste aufgef&uuml;hrten Mengen der einzelnen
Produkte bestellen? Pr&uuml;fen Sie genau, ob die aufgef&uuml;hrten Mengen
Ihrem Wunsche entsprechen.

<P><B>Darf's ein Bischen mehr sein?</B>
<BR>Sie k&ouml;nnen Ihre Order auch noch korrigieren. Klicken Sie einfach
in das Mengenfeld, und &auml;ndern Sie die Anzahl!  Achtung: Beim
Bundle sparen Sie 14 Mark
<BR><FORM ACTION = 'mailto:is@helga.mfh-iserlohn.de' METHOD=post
                  NAME = 'Bestellungen' onSubmit = 'return(teste())' >
<TABLE BORDER >

<TR>
<TD>Baustein </TD>

<TD>Produkt </TD>

<TD>Menge </TD>

<TD>Einz.-Preis </TD>

<TD>Summe DM </TD>
</TR>

<TR>
<TD>1 </TD>

<TD>4skin Cleanser (e 100 ml)</TD>

<TD><INPUT TYPE = text NAME =  waterQuantity  onChange = 'berechne()'
                              VALUE = '0' SIZE = 4 ></TD>
```

```
<TD ALIGN=LEFT>DM 18,00 </TD>

<TD ALIGN=CENTER><INPUT TYPE = text NAME = waterSum  VALUE = '0'
                        onChange = 'berechneNeu()'  SIZE = 10 ></TD>
</TR>

<TR>
<TD>2 </TD>

<TD>4skin Moisturizer (e 100 ml)</TD>

<TD><INPUT TYPE = text NAME =  gelQuantity
                        onChange = 'berechne()' VALUE = '0' SIZE = 4 ></TD>

<TD ALIGN=LEFT>DM 39,00 </TD>

<TD ALIGN=CENTER><INPUT TYPE = text NAME = gelSum + VALUE = '0'
                        onChange = 'berechneNeu()' SIZE = 10 ></TD>
</TR>

<TR>
<TD>3 </TD>

<TD>4skin Vitalizer  (e 100 ml)</TD>

<TD><INPUT TYPE = text NAME =  creamQuantity
                  onChange = 'berechne()' VALUE = '0' SIZE = 4 ></TD>

<TD ALIGN=LEFT>DM 46,00 </TD>

<TD ALIGN=CENTER><INPUT TYPE = text NAME = creamSum + VALUE = '0'
                  onChange = 'berechneNeu()' SIZE = 10 ></TD>
</TR>

<TR>
<TD>4 </TD>

<TD>4skin Starter Kit </TD>

<TD><INPUT TYPE = text NAME =  bundleQuantity
                  onChange = 'berechne()' VALUE = '0' SIZE = 4 ></TD>

<TD ALIGN=LEFT>DM 89,00 </TD>

<TD ALIGN=CENTER><INPUT TYPE = text NAME = bundleSum + VALUE = '0'
                  onChange = 'berechneNeu()' SIZE = 10 ></TD>
</TR>

<TR>
<TD> </TD>

<TD> </TD>
```

```
<TD> </TD>

<TD> </TD>

<TD> </TD>
</TR>

<TR>
<TD> </TD>

<TD><B>Versand &amp; Verpackung</B> </TD>

<TD>1 </TD>

<TD>DM 5,00 </TD>

<TD ALIGN=LEFT>DM 5,00 </TD>
</TR>

<TR>
<TD> </TD>

<TD><B>Gesamtkosten</B> </TD>

<TD> </TD>

<TD> </TD>

<TD ALIGN=CENTER><INPUT TYPE = text NAME = Gesamtkosten VALUE = '0'
                        onChange = 'berechneNeu()' SIZE = 10 ></TD>
</TR>
</TABLE>
 
<H3>
Geben Sie nun bitte Ihre Adresse an!</H3>

<TABLE BORDER >
<TR>
<TD>Name </TD>

<TD><INPUT TYPE = text NAME = name  SIZE = 20 ></TD>
</TR>

<TR>
<TD>Vorname </TD>

<TD><INPUT TYPE = text NAME = vorname  SIZE = 20 ></TD>
</TR>

<TR>
<TD>Wohnort </TD>
```

```
<TD><INPUT TYPE = text NAME = wohnort  SIZE = 20 ></TD>
</TR>

<TR>
<TD>PLZ </TD>

<TD><INPUT TYPE = text NAME = plz  SIZE = 20 ></TD>
</TR>

<TR>
<TD>Stra&szlig;e </TD>

<TD><INPUT TYPE = text NAME = strasse  SIZE = 20 ></TD>
</TR>

<TR>
<TD>e-mail </TD>

<TD><INPUT TYPE = text NAME = email  SIZE = 20 ></TD>
</TR>
</TABLE>


<P> 
<TABLE BORDER >
<TR>
<TD ALIGN=CENTER><INPUT TYPE = "submit" VALUE = "Absenden" Size = 30></TD>
</TR>
</TABLE>
</FORM><SCRIPT LANGUAGE = "JavaScript" >
      aktualisiereMengen();
      berechne();
</SCRIPT>
 
<B>.. und ab die Post!</B><B></B>

<P><B>Mit diesem Klick senden Sie Ihre Bestellung ab.</B>

<TD>
<BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE><IMG SRC="./bilder/net-m@nedition.jpg" HEIGHT=18 WIDTH=160></
BLOCKQUOTE>
</BLOCKQUOTE>
</BLOCKQUOTE>
</BLOCKQUOTE>
</BLOCKQUOTE>
</BLOCKQUOTE>
</TD>
</TR>
```

```
</TABLE>
<B></B> 
</BODY>
</HTML>
```

Beispiel 17.2 Die zugehörigen JavaScript-Dateien

```
//Dateiname: bearbeiteCookie.js

function ausgeben(ausgabeString)
{
// Achtung: Diese Funktion benoetigt zahlAusgeben.js.
// Diese Datei muss auch in
// html-Seite importiert werden!!!!

      var ausgabe;
      ausgabe = zahlAusgeben(ausgabeString);
      return ("DM " + ausgabe);
}


function berechne()
{
      var summe = 0;
      var produkt;
      var waterPreis = 18;
      var gelPreis = 39;
      var creamPreis = 46;
      var bundlePreis = 89;
      var versandPreis = 5;

      produkt = parseFloat(document.Bestellungen.waterQuantity.value)
                        * waterPreis;
      summe += produkt;
      document.Bestellungen.waterSum.value = ausgeben(produkt);

      produkt = parseFloat(document.Bestellungen.gelQuantity.value)
                        * gelPreis;
      summe += produkt;
      document.Bestellungen.gelSum.value = ausgeben(produkt);

      produkt = parseFloat(document.Bestellungen.creamQuantity.value)
                        * creamPreis;
      summe += produkt;
      document.Bestellungen.creamSum.value = ausgeben(produkt);

      produkt = parseFloat(document.Bestellungen.bundleQuantity.value)
                        * bundlePreis;
      summe += produkt;
      document.Bestellungen.bundleSum.value = ausgeben(produkt);

      summe += versandPreis;
      document.Bestellungen.Gesamtkosten.value = ausgeben(summe);
```

```
}

function berechneNeu()
{
      alert("Sie haben einen Wert in ein Feld eingegeben. Der Wert dieses
Feldes" +
            "\n wird aber durch die von Ihnen eingebene Artikelmenge und den"
+
            "\n Artikelpreis bestimmt. Der Wert des Feldes wird daher erneut"
+
            "\n berechnet." +
            "\n"+
            "\n Wenn Sie mehr bestellen wollen, klicken Sie einfach in das
Mengen-" +
            "\n feld und geben dort die neue Artikelmenge ein!");
      berechne();
}



function teste()
{
      if ((document.Bestellungen.waterQuantity.value == 0) &&
            (document.Bestellungen.gelQuantity.value == 0) &&
            (document.Bestellungen.creamQuantity.value == 0) &&
            (document.Bestellungen.bundleQuantity.value == 0))
      {
            alert ("Sie haben kein Produkt bestellt! Alle" +
                  "\n Bestellmengen sind 0!");
            return false;
      }
      if (document.Bestellungen.name.value == "")
      {
            alert("Sie haben keinen Namen angegeben!\n " +
                        "Das Formular kann daher nicht ¸bertragen werden!");
            return false;
      }
      if (document.Bestellungen.vorname.value == "")
      {
            alert("Sie haben keinen Vornamen angegeben!\n" +
                        "Das Formular kann daher nicht ¸bertragen werden!");
            return false;
      }
      if (document.Bestellungen.wohnort.value == "")
      {
            alert("Sie haben keinen Wohnort angegeben!\n" +
                        "Das Formular kann daher nicht ¸bertragen werden!");
            return false;
      }
      if (document.Bestellungen.plz.value == "")
      {
            alert("Sie haben keine Postleitzahl angegeben!\n" +
                        "Das Formular kann daher nicht ¸bertragen werden!");
            return false;
```

```
      }
      if (document.Bestellungen.strasse.value == "")
      {
            alert("Sie haben keine Strafle angegeben!\n" +
                        "Das Formular kann daher nicht ¸bertragen werden!");
            return false;
      }
      alert("Ihre Bestellung wird nun per e-mail an uns ¸bertragen!"+
            "\n Das net-m@n Team bedankt sich f¸r Ihre Bestellung!");
      return true;
}

function aktualisiereMengen()
{
// Achtung: Diese Funktion benoetigt stringSplitten.js. Diese datei muss auch
in
// html-Seite importiert werden!!!!
      var zaehler;
      var bestellString;
      var artikelArray;
      var meinCookie;
      meinCookie = document.cookie;
      zaehler = meinCookie.indexOf("=");
      bestellString = meinCookie.substring (zaehler + 1, meinCookie.length);
      artikelArray = splitte("&", bestellString);
      anzahlZeilen = artikelArray.length - 1;

      for (i = 0; i <= anzahlZeilen; i++)
      {
            artikelName = splitte ("!" , artikelArray [i]);
            if (artikelName[0] == "Water")
            {
                  document.Bestellungen.waterQuantity.value =
                              artikelName[1];
            }
            if (artikelName[0] == "Gel")
            {
                  document.Bestellungen.gelQuantity.value =
                              artikelName[1];
            }
            if (artikelName[0] == "cream")
            {
                  document.Bestellungen.creamQuantity.value =
                              artikelName[1];
            }
            if (artikelName[0] == "bundle")
            {
                  document.Bestellungen.bundleQuantity.value =
                              artikelName[1];
            }
      }
}
```

```
function setzeCookie()
{
      var meinCookie;
      var bestellung = false;
      meinCookie = "Bestellung=" +
                  "Water!" + document.Bestellungen.waterQuantity.value +
                  "&Gel!" + document.Bestellungen.gelQuantity.value +
                  "&cream!" + document.Bestellungen.creamQuantity.value +
                  "&bundle!" + document.Bestellungen.bundleQuantity.value;
      document.cookie = meinCookie;
}


// Dateiname: frameRecognizer.js

var derzeitigeURL;
var letzterPunkt;
var bisHTML;
var neueURL;

if (parent  == self)
{
      derzeitigeURL = location.href;
      letzterPunkt = derzeitigeURL.lastIndexOf(".");
      bisHTML = derzeitigeURL.substring(0, letzterPunkt);
      neueURL= bisHTML + "_index.html";
      location.replace (neueURL);
}


// Dateiname: stringSplitten.js

function splitte (splitCharacter, bestellString)
{
      var bestellArray = new Array();
      var zaehler;
      var i;
      var arrayZaehler = 0;
      var startPosition = 0;
      var endPosition = 0;
      endPosition = bestellString.indexOf(splitCharacter);
      if (endPosition == -1)
      {
            bestellArray[0] = bestellString;
            return bestellArray;
      }

      i = 1;
      while ((endPosition != -1) && (i <= 6))
      {
            i += 1;
            bestellArray [arrayZaehler] =
                  bestellString.substring(startPosition, endPosition);
```

```
            startPosition = endPosition + 1;
            arrayZaehler += 1;
            endPosition =
                  bestellString.indexOf(splitCharacter, startPosition);
      }
      startPosition = bestellString.lastIndexOf(splitCharacter) + 1;
      bestellArray [arrayZaehler] =
            bestellString.substring(startPosition, bestellString.length);
      return bestellArray;
}
```