Unterrichtsgang JavaScript

A. Burges und T. Regenbrecht

12. November 2003

Über Korrekturen und konstruktive Kommentare bin ich immer erfreut. Sie finden die aktuelle Version dieses Dokumentes, eine Möglichkeit zur Kontaktaufnahme sowie die längeren Quelltexte unter http://www.trcoding.de/informatik/javascript/index.htm.

Inhaltsverzeichnis

1 Vorbemerkungen **3**
1.1 Verwendete Konventionen in diesem Dokument . . . 3
1.2 Voraussetzungen . . . 3
1.3 Literatur zum Nachschlagen . . . 3
1.4 Wie arbeitet man mit diesem Kurs? . . . 3

2 Das erste JavaScript **3**
2.1 Hallo Welt! . . . 4

3 Variablen **4**
3.1 Inhalte von Variablen . . . 5

4 Events **6**

5 Ein MouseOver-Effekt **6**
5.1 Events . . . 6
5.1.1 Anmerkungen . . . 6
5.1.2 Der Quelltext . . . 7
5.1.3 Erläuterung . . . 7
5.2 Strings . . . 7
5.2.1 Ausgabe von Text auf der Seite . . . 8
5.2.2 Variablen in Strings . . . 8

6 Funktionen **8**
6.1 Ein einfacher Taschenrechner . . . 9
6.2 Welche Gründe gibt es, Funktionen zu benützen? . . . 10

7 Objekte **11**
7.1 Ansprechen der Bilder in der Seite . . . 11
7.2 Richtige Objekte - das Array Objekt . . . 12
7.3 For-Schleifen . . . 13
7.3.1 Geschachtelte Schleifen . . . 13
7.4 Richtige Objekte - das Date Objekt . . . 14
7.5 Variablentypen / Konversion String zu Zahl . . . 14
7.6 Ausgabe des Monatsnamens / Kombination von Array und Date Objekt . . . 14

8 Das erste richtige Script / Ändern vorhandener Elemente auf der Seite **14**
8.1 Kontinuierliche Änderung der Anzeige . . . 15
8.2 Anzeige der auf der Seite verbrachten Zeit . . . 15

9 Fallunterscheidungen: If-else/Wahrheitswerte **16**
9.1 Ein Quiz . . . 16
9.2 Analyse des Quizes . . . 18
9.2.1 Lokale und globale Variablen . . . 18
9.3 Erweiterung und Verbesserung des Quizes . . . 18

10 Weiteres zu Schleifen **19**
10.1 while-Schleife . . . 19
10.2 Rückgabewerte von Funktionen . . . 21

11 Mehrfachauswahlen **22**
11.1 **switch** . . . 22
11.2 Weiterleitung . . . 23

12 Eigene Objekte **24**
12.1 Definition eines eigenen Objektes . . . 24
12.2 Wozu sind Objekte eigentlich gut? . . . 27
12.3 Vorbelegung von Objekteigenschaften . . . 27
12.4 Vererbung . . . 28
12.5 Verstecken von Methoden . . . 29

13 Das Document Object Model (DOM) **30**
13.1 Eine Webseite nachträglich verändern . . . 30
13.2 Bewegen im DOM . . . 32
13.3 Wie man Überschriften numeriert . . . 32

14 Ein Graphikprogramm in JavaScript **33**
14.1 Das Grundgerüst der Webseite . . . 33
14.2 Wie zeichnet man schräge Linien? (1. Versuch) . . . 36
14.3 Wie zeichnet man schräge Linien? (2. Versuch) . . . 37
14.4 Kreise . . . 38

1 Vorbemerkungen

1.1 Verwendete Konventionen in diesem Dokument

Der JavaScript-Source Code ist innerhalb dieses Dokumentes formatiert angegeben, dabei werden Schlüsselwörter fett gedruckt und die Kommentare kursiv dargestellt. String-Literale sind in Schreibmaschinenschrift wiedergegeben.

1.2 Voraussetzungen

Um diesen Kurs erfolgreich durcharbeiten zu können, mußt du ein wenig einfaches HTML können. Bei Bedarf kannst du aber benötigte Elemente auch nachschlagen, insbesondere an der Stelle, wo es um Formulare in Webseiten geht.

Alle Beispiele wurden mit Mozilla 1.3 getestet, sie sollten auch ohne Probleme mit dem Internet Explorer 6 laufen. Fast alle Teile funktionieren auch mit dem Internet Explorer ab Version 5.0.

1.3 Literatur zum Nachschlagen

Einige Aufgaben in diesem Script erfordern es, dass du die Syntax (also die genaue Schreibweise sowie die Parameter) eines Befehls nachschlägst. Dazu suchst du am besten die Webseite http://selfhtml.teamone.de/javascript/index.htm auf. Das die Befehle hier nicht alle besprochen werden hat einen einfachen Grund.

Um ernsthaft zu programmieren, wirst du häufig nachschlagen müssen. Du solltest also lernen, wie und an welchen Stellen man die nötigen Informationen finden kann.

1.4 Wie arbeitet man mit diesem Kurs?

Dieser Text ist nur ein Fragment, er kann nur einen kurzen Abriß über die Eigenschaften und Möglichkeiten von JavaScript geben. Deshalb wird der Lehrer vermutlich weitere Beispiele, Aufgaben und Erläuterungen geben. Eine Sache muß man allerdings von Anfang an beherzigen:

Um Programmieren zu lernen, muß man vor allen Dingen eins tun: programmieren!

Deshalb müssen alle im Text angegebenen Beispiele eingegeben werden, ihr müßt diese jeweils in eine Web-Seite einbetten und sie „zum Laufen“ bringen. Alle Übungen müssen natürlich bearbeitet werden. Die Ergebnisse müssen schriftlich festgehalten werden, z.B. in Form eines Ausdrucks. Der Ausdruck ist natürlich nur bei umfangreichen Eingaben nötig, häufig reicht die Beantwortung der Frage oder einige wenige Zeilen, die man von Hand schreiben kann.

2 Das erste JavaScript

Zunächst mußt du wissen, wie man überhaupt ein JavaScript schreibt. JavaScript besteht aus einfachem Text, der in eine HTML-Seite hineingeschrieben wird. Man kann ein Script im Head oder im Body der Seite unterbringen, das kommt immer auf den Anwendungszweck an. Damit der Browser auch erkennt, dass jetzt JavaScript kommt, schreibst du:

```
1 <script language="javascript" type="text/javascript">
2 <!--
3         Hier steht dann das Script.
4 //-->
5 </script>
```

Die Zeilen <!-- und //--> müssen nicht unbedingt sein. Es handelt sich dabei um die Zeichen zur Begrenzung eines HTML-Kommentares. Sie dienen dazu das Script vor älteren Browsern, die damit nicht umgehen können, zu verstecken. In den folgenden Abschnitten wird immer der obige Block benützt, um das JavaScript aufzuschreiben. Das mußt du, um die Beispiele auszuprobieren, natürlich selbst ergänzen.

2.1 Hallo Welt!

Das typische Beispiel zur Einführung einer neuen Programmiersprache ist die Ausgabe von „Hallo Welt“ auf dem Bildschirm. In JavaScript geht das folgendermaßen (hier füge ich ausnahmsweise ein vollständiges Beispiel ein, die darfst in den anderen Beispielen die umgebenden Script-Tags – dort wo sie nötig sind – nicht vergessen):

```
<html>
<head>
</head>
<body>
<script language="javascript" type="text/javascript">
<!--

        window.alert("Hallo Welt!");

//-->
</script>
</body>
</html>
```

Anstatt window.alert darf man auch nur alert() schreiben, das window wird dann vom Browser ergänzt. Die Hochkommata geben an, dass der Text nicht zum Programm gehört, sondern so wie er ist am Bildschirm ausgegeben werden soll (ein sogenannter String). Beachte auch das Semikolon am Ende der Zeile. Ein Semikolon schließt einen einzelnen Befehl in JavaScript ab.

Übung 2.1

1. Verändere das obige Script so, dass es "Hallo <dein Name>" ausgibt!

3 Variablen

Du kannst das obige Script natürlich jedesmal verändern, wenn du einen anderen Namen ausgeben willst, und immer in das window.alert hineinschreiben. Aber es gibt eine viel bessere Variante:

```
        var name;     // hier wird eine Variable ins Leben gerufen
        name = "Tommy";  // Variable mit dem Inhalt "Tommy" belegen
        window.alert("Hallo " + name + "!");  // addiere den Namen
                                              //und den Text
```

Variablen sind Kästen, in die man etwas hineintun kann. Damit man den Kasten wiederfindet, bekommt er einen Namen. Zunächst teilen wir also mit, dass wir einen Kasten benötigen, dessen Name "name" ist. Die Variable mit Namen "Name" ist übrigens etwas ganz anderes, da Groß- und Kleinschreibung unterschieden werden!

Wir könnten den Kasten (die Variable) auch "Hugo" oder "q89d_djkfuiQQdk87" nennen, aber das wäre ziemlich umständlich. In den Kasten tun wir in der zweiten Zeile etwas hinein, nämlich die Zeichenkette "Tommy". In der dritten Zeile addieren wir den String "Hallo" mit *dem Inhalt* der Variable name, und dem aus einem Zeichen bestehenden String "!".

Das addieren sieht zunächst einmal komisch aus, da weder "Hallo" noch "Tommy" noch "!" Zahlen sind. Die Zeichenketten werden durch addieren einfach hintereinander gehängt. Die anderen mathematischen Operationen lassen sich auf Zeichenketten übrigens nicht anwenden.

Übung 3.0

1. Verzichte auf das Semikolon hinter alert, probiere es auch mit dem Semikolon hinter "Tommy", was stellst du fest?
2. Schreibe jetzt den Variablennamen in der dritten Zeile bei alert() groß (Name). Was für eine Ausgabe bekommst du?
3. Wenn wir schon dabei sind: schreibe doch auch den Funktionsnamen alert() groß, was erwartest du und welche Ausgabe erhälst du?

3.1 Inhalte von Variablen

Neben den oben bereits benutzten Strings kann eine Variable auch andere Inhalte, wie etwa Zahlen speichern. Welche Bedeutung die anderen Inhalte haben wird später besprochen. Beachte bitte, dass nur Zeichenketten in Hochkommata eingefasst werden, alle anderen Werte nicht.

Zur Vereinfachung können Variablen gleichzeitig deklariert und mit einem Wert gefüllt werden.

```
    var name = "Peter"; // Eine Zeichenkette
    var einkommen = 3000.283; // Speicherung der Zahl 3000.283
    var heute_ist_montag = false; // Speicherung eines Wahrheitswertes
    var a = new Array();
        // Speicherung eines neu erzeugten Objektes,
        // hier einer Liste
```

Kommazahlen werden mit einem Punkt und nicht mit einem Komma geschrieben, also 3000.489 und *nicht* 3000,489.

Der Inhalt einer Variablen kann während des Scriptablaufes beliebig oft geändert werden. Dafür gibt es auch gleich ein Beispiel.

Mit der Methode window.prompt wird eine Benutzereingabe erfragt. Der erste Parameter ist ein Aufforderungstext. Er beschreibt, was der Anwender eingeben soll. Der zweite Parameter ist eine Feldvorbelegung. Mit diesem Text wird das Eingabefeld vorbelegt. Wenn du ein leeres Eingabefeld willst, benütze eine leere Zeichenkette "" oder eine leere Variable.

Der Rückgabewert der Methode kann dann in einer Variablen gespeichert werden. Wie das geht, zeigt das nächste Script.

```
<html>
<head>
</head>
<body>
<script language="javascript" type="text/javascript">
<!--

    var name = "Ihr Name";
    name = window.prompt("Geben Sie Ihren Namen ein: ",name);
    var alter = window.prompt("Wie alt sind Sie?","");
    window.alert(name + " ist " + alter + " Jahre alt." );

//-->
</script>
</body>
</html>
```

Übung 3.1

1. Schreibe ein Script welches zwei Zahlen abfragt, diese addiert und dann mit einem Alert-Fenster wieder ausgibt. Du wirst dabei feststellen, dass das Addieren nicht so funktioniert wie erwartet. In Abschnitt 6.1 kannst du nachschauen, wie man zwei Zahlen einließt.

4 Events[1]

Bisher mußte die Seite jedesmal neu geladen werden, wenn das Script ausgeführt werden sollte. Außerdem wurde es automatisch beim Laden der Seite ausgeführt. Meistens will man aber auch Benutzeraktionen reagieren. Diese Benutzeraktionen sind Ereignisse = Events, die vom Browser registriert werden können. Was kann auf (mit) einer Web-Seite also alles passieren?

onLoad Die Seite wird geladen.

onUnload Die Seite wird verlassen.

onClick Es wird auf etwas geklickt.

onMouseOver Der Mauszeiger überfährt ein Element.

onMouseOut Der Mauszeiger verläßt ein Element.

onBlur Ein Fenster oder Formularfeld verliert den Focus (z.B. wird der Cursor in ein anderes Formularfeld gesetzt oder ein anderes Fenster in den Vordergrund geholt).

onFocus Ein Formularfeld oder ein Fenster erhält den Focus.

onChange Der Wert eines Formularfeldes wird geändert.

Wir kommen jetzt auch zur *zweiten Möglichkeit* JavaScript ausführen zu lassen. Man kann das gesamte Script in den EventHandler hineinschreiben, dann entfallen auch die sonst notwendigen Script-Tags.

```
1 <a href="#" onClick="document.bgColor = '#00CC00';">Klick mich!</a>
```

Zunächst erzeugen wir einen Link, das Ziel des Links ist das aktuelle Dokument (#). Innerhalb der doppelten Hochkommata hinter onClick steht das gesamte JavaScript. Innerhalb des Scriptes wird document.bgColor ein neuer Wert zugewiesen. Dabei handelt es sich um einen String (#00CC00), der seinerseits natürlich wie alle Strings in Hochkommata eingeschlossen sein muß.

5 Ein MouseOver-Effekt

5.1 Events

5.1.1 Anmerkungen

Im folgenden Beispiel benützen wir wohl die einfachste Form des MouseOver-Effektes, der HTML-Source Code ist der Übersichtlichkeit wegen bis auf das Notwendigste verkürzt (verändert nach [8], S. 320). [2]

Wie oben schon erwähnt, ist die zweite Möglichkeit Scripte auszuführen, die Benutzung von Events. Sehr häufig werden onMouseOver und onMouseOut benutzt. Das eigentliche Script steht dann *nach* dem Gleichheitszeichen in den Hochkommata.

[1] Nein, es geht hier nicht um Partys!

[2] Insbesondere das Bild-Objekt ist stark verkürzt angegeben. Das Script funktioniert so nur deswegen, weil dieser MouseOver-Effekt häufig gebraucht wird. Deshalb wurde eine abkürzende Schreibweise eingeführt.

Um Teile der Web-Seite zu manipulieren, d.h. zu verändern, muß man sie irgendwie ansprechen können. Bei einem Bild ist das recht einfach, man gibt dem Bild einen Namen und kann es dann über diesen Namen ansprechen. In folgendem Beispiel erzeugen wir auf der WebSeite ein Bild mit dem <**img** />-Tag und geben ihm den Namen "bild". Damit das Beispiel funktioniert, müss die Bilddateien natürlich vorhanden sein. Notfalls müßt ihr Bilder erstellen.

Ein Bild hat viele verschiedene Eigenschaften, z.B. die Eigenschaft `src` (Source=Quelle, in diesem Fall eine Datei). Um die Eigenschaft `src` zu verändern, weist man mit Hilfe eines Gleichheitszeichens eine neue Datei zu. Damit klar ist, welches Bild geändert werden soll, wird dieses natürlich auch angegeben. Die Eigenschaft, welche geändert werden soll, wird durch einen Punkt abgetrennt. Ein Beispiel:

```
1 bild.src="bild2.jpg";
```

Und zur Erinnerung: eine Anweisung in JavaScript wird mit einem Semikolon abgeschlossen, hier könnte statt einer Anweisung natürlich eine; und noch eine; ...; stehen.

5.1.2 Der Quelltext

Und nun der Quelltext zu dem MouseOver bzw. MouseOut Event:

```
1 <a href="#"
2         onMouseOver="bild.src='bild2.jpg';"
3         onMouseOut="bild.src='bild1.jpg';">
4         <img src="bild1.jpg" name="bild" alt="Image" width="104"/>
5 </a>
```

5.1.3 Erläuterung

In dem Link werden zwei Event-Handler definiert, onMouseOver und onMouseOut. Den Event-Handlern werden zwei sehr kurze Scripte zugewiesen. Bei onMouseOver wird bei etwas, was sich `bild` nennt, die `src` (Quelle) auf `bild2.jpg` gewechselt, d.h. das vorhandene Bild durch `bild2.jpg` ausgetauscht. Bei onMouseOut wird dieser Wechsel wieder rückgängig gemacht.

Neben der Eigenschaft `src` hat ein Bild weitere Eigenschaften, darunter die Eigenschaften `width` (Breite) und `height` (Höhe).

Übung 5.1

1. Wie liest man auf die Eigenschaften `width` und `height` aus? Schreibe dazu ein JavaScript!
2. Wenn der Mauszeiger die Graphik überfährt, soll die Graphik auch noch größer werden, verläßt der Mauszeiger die Graphik wieder, soll die Graphik wieder ihre ursprüngliche Größe annehmen.
3. Zwei Graphiken sollen mit MouseOver-Effekten versehen werden.

5.2 Strings

In Webseiten treten immer wieder zwei Arten von Hochkommata auf: einmal das doppelte Hochkomma " und das einfache '. Wozu sind sie gut?

Die einfachen Hochkommata umschließen einen Text. Damit weiß der Browser: Hier kommt ein zusammengehöriger Bereich, der einfach so stehen bleiben soll wie er da steht. Man nennt so etwas einen String (eine Folge von Textzeichen). Strings haben für den Browser keine besondere Bedeutung, er muß nicht versuchen einen Sinn zu erkennen. Eine Anweisung bild2.jpg kann der Browser nicht ausführen, da er sie nicht kennt. Ganz im Gegensatz zu bild.src, dabei weiß der Browser ganz genau worum es sich handelt.

Man muß sich leider daran gewöhnen, dass ein doppeltes Hochkomma innerhalb der Web-Seite und innerhalb eines Scriptes unterschiedliche Bedeutung haben kann. Innerhalb der Web-Seite können sie z.B. Scripte einfassen, wie im obigen Beispiel. Innerhalb von Scripten definieren sie Strings (die aber einige Besonderheiten haben, wie wir noch sehen werden).

Um mit Strings arbeiten zu können, lernen wir gleich ein besonders wichtiges Ding kennen, das document. Das document ist der Inhalt des Fensters. Der Fensterinhalt hat natürlich verschiedene Eigenschaften, z.B. den Titel oder die Schriftfarbe. Außerdem gibt es Tätigkeiten, die das document für uns erledigen kann, z.B. Text auf der Seite ausgeben. Text, das bedeutet Strings.

5.2.1 Ausgabe von Text auf der Seite

Der Aufruf von document.write("Text") gibt Text auf der Seite aus, so als ob der Text direkt eingegeben worden wäre. Es können dabei auch HTML-Tags auf der Seite ausgegeben werden. Eine Variante des Befehls document.write() ist document.writeln(). Zusätzlich zu dem übergebenen Text gibt er noch einen Zeilenumbruch aus (der allerdings wie alle Zeilenumbrüche auf der Webseite nicht sichtbar ist, nur im Quelltext der Seite!).

Das Zusammenfügen von Strings nennt man „Concatenation“. In den meisten Programmiersprachen wird das mit einem Punkt erledigt, JavaScript ist da anders. In JavaScript werden Strings addiert. Das sieht dann so aus:
document.write("Text"+ "
"+ "noch mehr Text");

5.2.2 Variablen in Strings

Im folgenden Beispiel wird gezeigt wie eine Variable benutzt wird, um einen bestimmten festen Ausdruck in einen String einzufügen. Jetzt ist die Variable br fast genauso lang wie die Zeichenkette
, aber bei einer komplizierteren Formatierung spart das natürlich viel Tipparbeit.

```
var br="<br />";
window.document.write("Text" + br +
        "zweite Zeile" + br + "dritte Zeile");
        // Der Zeilenumbruch ist hier nur drin,
        // damit die Zeile nicht zu lang wird.
```

Übung 5.2

1. Die Adresse der Seite soll auf der Seite selbst ausgegeben werden. Ihr müßt zunächst heraussuchen, welches Objekt in welcher Eigenschaft die Adresse bereithält. Geht daher zu Selfhtml, sucht das passende Objekt und die richtige Eigenschaft in der Objektreferenz. Fragt diese Eigenschaft ab und gebt nun über document.write(); die Adresse auf der Seite aus.

2. Sucht euch anschließend drei Dokument-Eigenschaften und ändert diese. Benutzt zur Änderung ein Mouse-Over Event!

---

6 Funktionen

Bis jetzt haben wir das gesamte Script bei Bedarf immer in einen Link hineingeschrieben, das ist doch ziemlich umständlich. Auch wenn man ein Script mehrmals auf der Seite ausführen will, benötigt man eine bessere Lösung. Diese Lösung heißt: Funktionen! Eine Funktion hat folgende Syntax:

```
function funktionsname(parameter1, parameter2, ...)
{
        //Programmblock
        return wert;
}
```

Die Anzahl der Parameter ist beliebig - der Funktionsname auch, solange kein reserviertes Wort benutzt wird. Der Wert der hinter dem **return** steht, wird als Rückgabewert der Funktion zurückgegeben. Häufig soll die Funktion aber gar keinen Wert zurückgeben, sondern nur z.B. einen Text ausgeben. Dann kann das **return** auch weggelassen werden. Als Beispiel eine Funktion, die Text auf der Seite fett formatiert ausgibt.

```
<html>
<head>
<script language="javascript" type="text/javascript">

function gibfettentextaus(text)
{
        document.write("<b>"+text+"</b>");
}

</script>
</head>
<body>
<script language="javascript" type="text/javascript">

        gibfettentextaus('Dies ist etwas fetter Text');

</script>
</body>
</html>
```

Das Beispiel ist etwas witzlos, genausogut könnten wir natürlich direkt formatieren - zeigt aber das wesentliche. Interessanter wird es natürlich, wenn wir die Funktion benutzen um mehr als ein Attribut zu setzen, dann sparen wir Schreibarbeit.

Übung 6.0

1. Gib den Quelltext ein und ändere ihn soweit ab, dass der Text nicht nur fett formatiert, sondern gleichzeitig kursiv und in Rot ausgegeben wird.

6.1 Ein einfacher Taschenrechner

Funktionen können natürlich wiederum andere Funktionen aufrufen. Das ist eigentlich selbstverständlich, denn selbstgeschriebene Funktionen unterscheiden sich nicht von den vordefinierten Funktionen. Das sehen wir im folgenden Beispiel.

```
<html>
<head>
  <title>
    Der Taschenrechner, der eigentlich keiner ist.
  </title>
  <script language="JavaScript">
```

```
        var zahl1,zahl2,antwort,ergebnis;
        // Diese Variablen brauchen wir gleich im Script
        function zahlen_eingabe()
        {
            zahl1=window.prompt("Gib die erste Zahl ein","Zahl");
            zahl2=window.prompt("Gib die zweite Zahl ein","Zahl");
            zahl1=parseFloat(zahl1);
            zahl2=parseFloat(zahl2);
        }

        function addieren()
        {
                zahlen_eingabe();
                ergebnis=zahl1 + zahl2; // hier wird gerechnet!
                antwort="Die Summe aus " + zahl1 +
                        zahl2 + " ist " + ergebnis + ".";
                window.alert(antwort);
        }

  </script>
</head>
<body>
  <h3>Rechen-Script</h3>
  <form name="formular">
    <table border="0">
        <tr><td>Addition: </td>
            <td><input type="button" value="(+)"
                name="plus" onClick="addieren()"></td>
        </tr>
        </table>
  </form>
</body>
</html>
```

Übung 6.1

1. Wozu braucht man **parseFloat**()?
2. Neben **parseFloat**() gibt es noch die Funktion **parseInt**(). Welchen Unterschied gibt es zwischen diesen beiden Funktionen?
3. Erweitere den „Taschenrechner“ um die anderen drei Grundrechenarten.

6.2 Welche Gründe gibt es, Funktionen zu benützen?

Das Einsparen von Tipparbeit ist eigentlich der unbedeutendste Grund Funktionen zu benützen. Funktionen wurden vor allen Dingen deswegen eingeführt, um ein Programm in mehrere übersichtliche Teilstücke aufteilen zu können.

Ein Programm soll ja ein bestimmtes Problem lösen. Häufig ist das Problem ziemlich schwierig, und ein Lösungsweg umfasst viele Schritte. Man dröselt das Problem daher in (hoffentlich einfacher zu lösende) Teilprobleme bzw. Teilschritte auf, bis man einen Teilschritt genau festlegen kann. Für diesen

Teilschritt schreibt man eine Funktion. Der Funktion werden bestimmte Werte übergeben, die Funktion gibt dann ein Ergebnis zurück bzw. führt eine Aktion aus.

Insgesamt wird ein Programm dadurch viel übersichtlicher, leichter zu bearbeiten, zu verstehen und zu korrigieren (debuggen). Funktionen definieren *wie* etwas getan wird, also die *Methode*.

Das zweite wesentliche Element von Programmen sind die Daten, es wäre natürlich schön, die auch irgendwie schön verpackt und in Teilelemente gegliedert bearbeiten zu können. Bei den Daten geht es um das *was*, und im folgenden Abschnitt wird gezeigt wie man das Ziel der Gliederung und Einkapselung erreichen kann.

7 Objekte

Nachdem im Kapitel 5 immer ein bißchen drumherum geredet wurde, soll doch jetzt die offizielle Sprachweise eingeführt werden.

Ein Objekt ist ein Ding, das Eigenschaften hat und irgendetwas anstellen kann[3]. Nehmen wir als Beispiel ein Auto. Ein Auto hat z.B. eine Farbe und es kann fahren. Die Farbe ist also eine *Eigenschaft*, das Fahren ist eine *Methode*. Eigenschaften entsprechen den Variablen, Methoden den Funktionen eines Objektes. Man könnte auch genausogut von Objektvariablen und Objektfunktionen reden.

Schauen wir uns ein zweites Fortbewegungsmittel an, das Fahrrad. Auch ein Fahrrad hat eine Farbe, auch das Fahrrad kann fahren. Offensichtlich gibt es also Gemeinsamkeiten zwischen den beiden Fortbewegungsmitteln, Unterschiede liegen in der erreichbaren Geschwindigkeit, der Transportkapazität, der Antriebsart und so weiter. Diese Gemeinsamkeiten kann man in einer Programmiersprache nachbilden.

Haben zwei Dinge den gleichen Satz von Eigenschaften (Farbe, Höchstgeschwindigkeit, ...) und Methoden (Fahren), gehören sie zur gleichen *Klasse*. Insbesondere zur gleichen Klasse gehören natürlich zwei Autos vom selben Typ, aber unterschiedlicher Farbe. Zur gleichen Klasse (der Fortbewegungsmittel) können auch Autos und Fahrräder gehören.

Was hat das mit JavaScript zu tun? Das Browserfenster ist ein Objekt. Das Dokument (die WebSeite) ist auch ein Objekt. Jedes Bild auf der Seite ist ein Objekt. Jedes Formular ist ein Objekt. Es gibt offensichtlich Objekte, die andere Objekte enthalten können!

Um ein bestimmtes Objekt ansprechen zu können, muß man den vollständigen Namen eines Objektes wissen. Nehmen wir an, wir wollten ein Bild auf der Seite verändern. Der vollständige Pfad bis zu dem Bild heißt window.document.images, jetzt muß man nur noch angeben, welches der Bilder man meint.

Das übergeordnete Objekt ist das window, das ist das Browserfenster. In dem Browserfenster kann eine Webseite angezeigt werden, das document. Innerhalb der Webseite kann es Bilder (images) geben.[4]

Wir haben ja schon window.document.write, eine Methode, und window.document.URL, ein Attribut, benutzt. Manche Attribute kann man lesen und schreiben, wie z.B. die Hintergrundfarbe des document. Die Adresse der Seite kann man nur lesen. Welche Attribute und welche Methoden ein Objekt hat, muss man in der Referenz nachschlagen.

7.1 Ansprechen der Bilder in der Seite

Kommen wir zu einer weiteren Übung. Die Ausgabe von Bildattributen mittels document.write geht leider nicht so einfach wie die Methode aus Abschnitt 5.1, wo ein Bild ausgetauscht wurde. Ein erster Versuch, die Breite eines Bildes auf der Seite auszugeben, könnte so aussehen:

```
<a href="#"
        onMouseOver="bild.src='bild2.jpg';"
        onMouseOut="bild.src='bild1.jpg';">
<img src="bild1.jpg" name="bild" alt="Image" width="104" />
</a>
<script language="javascript" type="text/javascript">
```

[3] Siehe [9] S. 83. Übersicht in [4], S. 43.

[4] Das document wird vom Browser auch ohne das window verstanden, window wird dann automatisch ergänzt.

```
7           document.writeln(bild.width);
8   </script>
```

Dies funktioniert nicht, da das Objekt bild, und damit auch bild.width im JavaScript nicht mehr definiert ist. Es ist während der Programmausführung nicht klar, was bild sein soll. Man muß dem Browser also auf die Sprünge helfen und ihm genau sagen, was bild eigentlich ist. Um die richtige Syntax hinzubekommen, hilft folgende Überlegung weiter: Bilder sind Teile der Seite, und die Seite ist ein Teil des Fensters. Es gibt das Objekt: window.document.images, in dem muß jetzt noch das Bild angesprochen werden. Die richtige Syntax lautet also:

```
1   document.writeln(window.document.images["bild"].width);
```

Die Schreibweise mit den eckigen Klammern dürfte im nächsten Kapitel klar werden.

Übung 7.1

1. Bring das obige (fehlerhafte) Beispiel zum Funktionieren. Es soll auf der Webseite die Breite des Bildes ausgegeben werden.
2. Schreibe eine Web-Seite auf der ein Bild angezeigt wird (per <**img** />-Tag). Unter dem Bild sollen mit JavaScript alle über das Bild verfügbaren Informationen ausgegeben werden.

7.2 Richtige Objekte - das Array Objekt

Bisher haben wir immer schon fertige Objekte benutzt, also Objekte die schon vorhanden sind, wenn der Besucher die Seite aufruft (z.B. das Browserfenster). Andere Objekte sind noch nicht da, sondern müssen erst erzeugt werden (sie werden nicht auf allen Seiten benötigt). Ein Beispiel sind sogenannte Arrays. Ein Array ist nichts weiter als eine Liste, die freundlicherweise beliebig lang werden kann (so weit der Hauptspeicher reicht). Woran man sich etwas gewöhnen muss, ist das in jeder richtigen Programmiersprache die Zählung der Liste bei 0 und nicht bei 1 beginnt.

```
1   var a = new Array();
2   a[0] = "Januar";
3   a[1] = "Februar";
4   a[2] = "März";
5   a[3] = "April";
6   a[4] = "Mai";
7   a[5] = "Juni";
8   a[6] = "Juli";
9   //usw.
```

Mit der ersten Zeile **var** a = **new** Array() wird eine Array-Variable deklariert, und ein Array-Objekt erzeugt. Die vorhandene Liste wird dann mit den Monatsnamen gefüllt, und zwar in der Reihenfolge der Monate. Um das Array zu füllen gibt es noch eine kürzere Methode.

```
1   var a = new Array("Januar","Februar","März","April","Mai","Juni",
2                     ...,"Dezember");
```

Auf die Elemente des Arrays kann man ganz einfach zugreifen, indem man den Index in der eckigen Klammer angibt.

```
1           document.write(a[0]);
```


Übung 7.2

1. Ergänze das Array um die fehlenden Monate und schreibe dann eine Seite, in der das Array in umgekehrter Reihenfolge wieder ausgegeben wird (Dezember, November usw.)

7.3 For-Schleifen

Besser ist in der obigen Übung die Benutzung der for-Schleife.

```
1           for (Initialisierung; Bedingung ; Befehl ){
2                   Anweisungen;
3           }
```

Dazu das Beispiel mit der Ausgabe der Monate:

```
1           for ( var i=0 ; i<12 ; i++ ){
2                   document.write(a[i]+"<br />");
3           }
```

Das doppelte Plus ist der sogenannte Inkrement-Operator, er bedeutet das gleiche wie i=i+1;. Außer dem Inkrement-Operator gibt es auch den Dekrement-Operator i−−;, das bedeutet das gleiche wie i=i−1;.

Eine Schleife wie die **for**-Schleife eignet sich immer dann hervorragend, wenn eine bestimmte Anzahl von Elementen ausgegeben werden soll, z.B. eine Zahlentabelle.

Übung 7.3

1. Löse die Übung 7.2 noch einmal, aber dabei mit einer for-Schleife. Benutze dazu den Dekrement-Operator −−.
2. Gib die Quadratzahlen von 1 bis 25 aus!

7.3.1 Geschachtelte Schleifen

Schleifen können natürlich auch ineinandergeschachtelt werden. Ein typisches Anwendungsbeispiel ist die Ausgabe einer Tabelle. Die Ausgabe des Zeilenumbruchs erfolgt dann in der äußeren Schleife, die Ausgabe der Tabellenzellen in der inneren Schleife. Wenn z.B. eine HTML-Tabelle erzeugt werden soll, benötigt man zunächst das Tag <**table**>. In der äußeren Schleife wird dann das einleitende <tr> geschrieben, anschließend in der inneren Schleife die Felder <td></td>. Nach Abschluß der inneren Schleife folgt das abschließende </tr>. Wenn beide Schleifen schließlich durchlaufen wurden, wird das abschließende </**table**> ausgegeben.

Übung 7.3

1. Gib das kleine 1*1 in tabellenform aus!

7.4 Richtige Objekte - das Date Objekt

Häufig sieht man auf Web-Seiten das aktuelle Datum und/oder die aktuelle Uhrzeit. Dabei handelt es sich meistens nicht um die Uhrzeit auf dem Server, sondern um die Zeit auf dem Client-Rechner – also dem Rechner des Besuchers. Auch dafür benötigt man ein Objekt, das Date-Objekt. Als erstes benötigen wir eine Instanz des Objektes:

```
1   var d = new Date();
```

Übung 7.4

1. Schlage nach und schreibe auf, welche Methoden das Date-Objekt besitzt (okay, 6 Methoden reichen), und welchen Rückgabewert sie haben!
2. Gib das aktuelle Datum und die aktuelle Uhrzeit im Format "Es ist hh:mm am dd des jetzigen Monats jjjj!"

Man kann insbesondere mit dem Rückgabewert getTime rechnen, d.h. Zeiten voneinander subtrahieren oder sie addieren. start.getTime − jetzt.getTime liefert die Zeitdifferenz in Millisekunden zwischen einer Startzeit und Jetzt.

7.5 Variablentypen / Konversion String zu Zahl

Das obige Beispiel krankt natürlich an der fehlenden Ausgabe des Monatsnamens. Was passiert denn, wenn du d.getMonth() ausgibst?

Der naheliegende Versuch d.getMonth()+1 liefert zumindest numerisch den richtigen Monat. Aber was passiert, wenn du `"Monat "`+ d.getMonth()+1 ausgibst? Das Problem ist hier, wie JavaScript den Wert d.getMonth() interpretiert. Steht er alleine gilt er als Zahl, mit der man rechnen kann. Im zweiten Beispiel wird er zunächst in eine Zeichenkette umgewandelt, und da man mit Zeichenketten nicht rechnen kann, wird die 1 auch zur Zeichenkette gemacht (das nennt man automatische Typkonvertierung).

Man kann JavaScript zwingen einen String als Zahl zu behandeln. Dazu multipliziert man ihn mit 1 oder benutzt die bereits bekannte Funktion **parseInt**() (bzw. **parseFloat**()).

7.6 Ausgabe des Monatsnamens / Kombination von Array und Date Objekt

Den Monatsnamen haben wir aber immer noch nicht. Aber mit der Kombination des Monatsarrays und des Date-Objektes kannst du jetzt das Datum in folgendem Format ausgeben:

Übung 7.6

1. "Es ist hh:mm am dd. Monatsname jjjj!"

8 Das erste richtige Script / Ändern vorhandener Elemente auf der Seite

In unserem ersten richtigen Programm müssen wir die bisher erlernten Dinge kombinieren. Das Ziel soll ein, die vom Besucher auf der Seite verbrachte Zeit anzuzeigen. Dafür brauchen wir noch ein paar Dinge.

window.setTimeout(`"auszuführende Funktion"`,<Zeit>);
Diese Methode führt eine Funktion nach einer bestimmten Verzögerungszeit aus.

Zur Anzeige benutzen wir ein Formular. Das stellt ein Eingabefeld zur Verfügung, dessen Eingabe wir aber gar nicht auswerten. Das Eingabefeld kann mit einem Wert vorbelegt werden, und das nutzen wir aus. Die Namen sind wichtig, weil wir mit Hilfe der Namen das Eingabefeld ansprechen könnnen.

```
1  <form action="" name="Anzeige">
2  <input size="40" name="Zeit" value="Beim Laden der Seite war es ...">
3  </form>
```

window.document.Anzeige.Zeit.value=`"Wert"`; gibt den Wert im Formular aus. Wir fangen ganz einfach an.

Übung 8.0

1. Schreibe ein HTML-Dokument, das 10 Sekunden nach dem Laden der Seite den Ladezeitpunkt in dem Formular ausgibt (in einem vernünftigen Format versteht sich - also hh:mm:ss).

8.1 Kontinuierliche Änderung der Anzeige

Wie kann diese Anzeige jetzt kontinuierlich geändert werden? Man müßte die Funktion periodisch wieder aufrufen. Das kann man dadurch erreichen, dass die Funktion sich selbst mit Verzögerung aufruft! Am Ende der Funktion muss also wieder
window.setTimeout(`'Funktionsname()'`,Zeit); aufgerufen werden.

Übung 8.1

1. Gib die Uhrzeit im Format hh:mm:ss auf der Seite in einem Formularfeld aus. Die Uhrzeit soll natürlich ständig aktualisiert werden, also mit laufenden Sekunden.

8.2 Anzeige der auf der Seite verbrachten Zeit

Zur endgültigen Lösung fehlt nicht mehr viel. Das Vorgehen muss etwa folgendermaßen sein.

1. Zeit bei Aufruf der Seite merken (das muß nur einmal geschehen)
2. Von der aktuellen Zeit die Startzeit abziehen und das Ergebnis durch 1000 teilen.
3. Das Ergebnis von Schritt 2 schön formatieren und in Minuten und Sekunden aufteilen.
4. Das Ergebnis von Schritt 3 ausgeben.
5. Nach einer Sekunde bei Schritt 2 weitermachen.

Nach Schritt 2 haben wir die Zeitdauer in Sekunden, die der Besucher auf der Seite verbracht hat. Um die Sekundenzahl in Sekunden und Minuten aufzuteilen und die Ausgabe zu formatieren, sind zwei Funktionen ausgesprochen nützlich.

Math.round(Zahl) Rundet die übergebene Zahl wie in der Mathematik üblich.

zahl % zahl Modulo-Operator (Rest-Operator). Dieser ermittelt den ganzzahligen Rest einer Division. 1000 % 60 liefert also die Zahl 40.

Um daraus die bisher abgelaufenen Minuten zu ermitteln, braucht man etwa folgende Berechnungsvorschrift *(Algorithmus)*:
Minuten = (AbsoluteSekundenzahl − RelativeSekundenZahl) / 60;

Übung 8.2

1. Die Zeit, die ein Besucher auf einer Web-Seite verbracht hat, soll in einem Formularfeld angezeigt werden. Das Format für die Zeitausgabe soll mm:ss sein, allerdings darf eine führende Null ausgelassen werden.

9 Fallunterscheidungen: If-else/Wahrheitswerte

Eins der wichtigsten Sprachelemente einer Programmiersprache ist die Fallunterscheidung Wenn/Dann. Praktisch kein Programm kommt ohne diese aus. In JavaScript sieht diese Fallunterscheidung so aus:

```
if (Bedingung) {
        Anweisungen wenn Bedingung erfüllt;
} else {
        Anweisungen wenn Bedingung nicht erfüllt;
}
```

Wenn die Bedingung erfüllt ist, dann wird der erste Teil ausgeführt, ist sie nicht erfüllt, der zweite. Der **else**-Teil kann weggelassen werden. Bedingungen werden so definiert:

```
(true) // Immer wahr
(false) // Immer falsch
(1==2) // Vergleich, offensichtlich falsch
(1!=2) // Sind die Werte ungleich? Richtig!
((1!=2)&&(1!=3)) // Und-Verknüpfung,
        //      beide Seiten müssen wahr sein.
((1!=2)||(1==2)) // Oder-Verknüpfung,
        //      nur eine Seite muss wahr sein.
(1>2) // Eins ist größer als zwei?
(1<2) // Eins ist kleinder als zwei!
```

Um die verschiedenen Varianten durchzuprobieren, benützt ein JavaScript mit der Ausgabe von alert-Fenstern.

Übung 9.0

Ermittle den Wahrheitswert folgender Bedingungen:
(1==2); (1==2)&&(3==4); (1==2)&&(3==3); ("Ich"=="Du"); ("Ich"!="Du");
(**true**); ((**true**)||(**false**)); ((13>12)||(13<12)); ((13>12)&&(13<12));
((1!=2)||(1==2)); ((7!=8)||(6==7)); (("A"=="A")&&("A"=="a"));

9.1 Ein Quiz

Als nächstes Programm sollt ihr ein Quiz programmieren. Dazu brauchen wir natürlich die Fragen selbst, dann muss die Antwort vom Besucher der Web-Seite abgefragt werden und schließlich wird ihm angezeigt, ob seine Antwort richtig oder falsch war. Ganz am Ende des Testes bekommt er noch das Gesamtergebnis.

```
<html>
<head>
<title>Ein Quiz</title>
<script language="javascript" type="text/javascript"><!--

  var frage1="Wie heißt die Hauptstadt von Frankreich?";
  var antwort1="PARIS";
  var zaehler=0;

  function befrage(frage,antwort)
  {
    var eingabe=window.prompt(frage,"Gib hier die Lösung ein");
    if(eingabe==null) {
      window.alert("Bis zum nächsten Mal.");
    } else {
      eingabe=eingabe.toUpperCase();
      if (antwort==eingabe) {
        window.alert("Richtige Antwort. Bravo!");
        zaehler +=1;
      } else {
        window.alert("Leider die falsche Antwort, sie Nulpe!");
      }
    }
  }

  function quiz_start()
  {
    befrage(frage1,antwort1);
    if(zaehler==1) {
      window.alert("Sie haben alles gewußt, prima!");
    }
    if (zaehler==0) {
      window.alert("Sie haben ja gar nichts gewußt.");
    }
    zaehler=0;
  }

//--></script>
</head>
<body>
  <table>
    <tr><th>Teste dein Wissen!</th></tr>
    <tr>
    <td align="center">
      <form name="formular">
        <input type="button" value="Quiz starten"
                onClick="quiz_start()">
      </form>
    </td></tr>
  </table>
</body>
</html>
```

9.2 Analyse des Quizes

Das sieht ja schon ganz schön kompliziert aus, ist es aber gar nicht. Zunächst mal solltest du versuchen, das Programm von Hand durchzuführen. Dazu mußt du noch einige Dinge wissen.[5]

Der Aufruf window.prompt liefert beim Klick auf „Abbrechen“ den Rückgabewert **null**. Das ist keine Zeichenkette und auch keine Zahl, es ist ein leerer Wert. Die Variable eingabe ist dann leer. Daher wird das Script an der Stelle abgebrochen. Leere Werte braucht man immer dann, wenn ein Nutzer auch einen leeren String `""` oder eine 0 eingeben könnte. Der String `"null"` ist natürlich etwas völlig anderes als ein leerer Wert!

9.2.1 Lokale und globale Variablen

An dieser Stelle muss ich unbedingt noch auf den Unterschied zwischen *lokalen* und *globalen* Variablen hinweisen. In Zeile 7 wird die Variable zaehler definiert.[6] Diese Variable bezeichnet man als global, das bedeutet, sie ist überall im Script gültig. In der Funktion befrage() wird zaehler um eins heraufgesetzt, wenn die Antwort richtig war. In der Funktion quiz_start wird die Variable dann abgefragt, und am Spielende wieder auf 0 zurückgesetzt.

Die Benützung einer globalen Variablen lässt sich an dieser Stelle auch nicht vermeiden, brauchen wir doch einen Zähler, der von Beginn bis zum Ende des Spieles gültig ist. Würden wir die Variable erst in der Funktion befrage() definieren, wäre sie in der Funktion quiz_start() nicht definiert!

Es liegt jedoch eine große Gefahr in der Benutzung globaler Variablen. Der Programmierer muß zu jedem Zeitpunkt wissen welche Werte die Variable annehmen kann, und an welchen Stellen sie noch benützt wird. Wird das Programm auch nur etwas größer, sind Programmfehler nur schwer zu vermeiden. Aus genau diesem Grund wurde die objektorientierte Programmierweise entwickelt.

Übung 9.2

1. Was macht die Funktion toUpperCase?
2. Jetzt führe das Programm auf Papier aus. Nummeriere die Zeilen in der Reinfolge des Programmablaufes. Du fängst an, wenn der Besucher der Seite auf den Button klickt. Schreibe hinter die entsprechende Zeile die Zahl 1. Du musst dich natürlich entscheiden, ob der Besucher die richtige oder die falsche Antwort eingibt. Gehe jetzt Schritt für Schritt durch das Programm.
3. Warum wird toUpperCase hier eingesetzt?
4. Es gibt eine Funktion, die das Gegenteil von toUpperCase macht. Wie heißt sie?

9.3 Erweiterung und Verbesserung des Quizes

Das Quiz hat in der obigen Form ganz wesentliche Nachteile. Es ist nur schwer zu erweitern, an mehreren Stellen sind die Ausgaben des Programmes von der Zahl der Fragen abhängig. Du sollst das Quiz verbessern. Die erste Verbesserung ist die Benutzung eines Arrays zur Speicherung der Fragen und Antworten.

```
1  var fragen=new Array();
2    fragen[0]="Wie heißt die Hauptstadt von Frankreich?";
3  var antworten=new Array();
4    antworten[0]="PARIS";
5  var zaehler=0;
```

[5] Übrigens enthält das Script einen logischen Fehler, findest du ihn?

[6] Der Variablen zaehler wird auch gleich ein Wert zugewiesen.

Durch diese Konstruktion können wir beliebig viele Fragen und Antworten hinzufügen, ohne dass wir ständig neue Variablen erzeugen müßten. Natürlich muss der Rest des Programmes auch angepasst werden. Wir ändern in Zeile 27 den Aufruf der Funktion:
befrage(frage1,antwort1) in: befrage(fragen[0], antworten[0]).

Dann müssen die Ergebnisse anders bewertet werden. Die Gesamtzahl der definierten Fragen und Antworten finden wir in der Eigenschaft length des Arrays. fragen.length gibt an, wieviele Einträge in dem Array stecken. Wir unterscheiden zwischen den Bedingungen: Alles gewußt, mehr als oder genausoviel wie die Hälfte gewußt, weniger als die Hälfte gewußt, gar nichts gewußt.

```
1  function quiz_start()
2  {
3    befrage(fragen[0],antworten[0]);
4    if(zaehler==fragen.length) {
5      window.alert("Sie haben alles gewußt, prima!");
6    }
7    if (zaehler>=Math.round(fragen.length/2)) {
8      window.alert("Sie haben immerhin mehr als die Hälfte gewußt!");
9    }
10   if (zaehler<Math.round(fragen.length/2)) {
11     window.alert("Sie haben noch nicht einmal die Hälfte gewußt!");
12   }
13   if (zaehler==0) {
14     window.alert("Sie haben ja gar nichts gewußt.");
15   }
16   zaehler=0;
17 }
```

Dieses Script enthält natürlich direkt zwei Fehler im Programmablauf, man erhält nämlich zwei Alert-Fenster. Das darfst du in der folgenden Übung verbessern.

Zunächst aber muß die Funktion befrage() so oft aufgerufen werden, wie Fragen da sind. Dazu benützen wir wieder eine Schleife:

```
1    for (var i=0; i<fragen.length; i++){
2        befrage(fragen[i],antworten[i]);
3    }
```

Übung 9.3

1. Schreibe das Quiz so um, dass du beliebig viele Frage/Antwort-Paare abfragen kannst. Verbessere die Ausgabe des Ergebnisses, so dass in jedem Fall nur ein Alert-Fenster mit einer Bewertung erscheint.

10 Weiteres zu Schleifen

In diesem Kapitel wird dir ein weiterer wichtiger Schleifentyp vorgestellt. Außerdem erfährst du etwas über Rückgabewerte von Funktionen. Zum Schluß werden wir uns mit der Steuerung der Schleifenausführung beschäftigen.

10.1 while-Schleife

Häufig ist die Dauer der Schleifenausführung nicht im voraus bekannt. Das gilt z. B. immer dann, wenn ein Benutzer beliebig viele Eingaben machen darf, oder eine vorher unbekannte Zahl von Einträgen aus

einer Datei ausgelesen werden sollen. Im Grunde wird auch die Programmausführung jedes Programms mit einer Benutzeroberfläche durch eine Schleife gesteuert. Für solche Fälle gibt es in JavaScript die **while**-Schleifen.

```
while (Bedingung) {
        Anweisungen;
}
```

Die **while**-Schleife ist also eigentlich viel einfacher als die **for**-Schleife. Man *muss* aber in den Anweisungen dafür sorgen, dass die Bedingung irgendwann einmal falsch wird. Als Beispiel ein Script, dass mit einer **while**-Schleife eine **for**-Schleife nachbaut.

```
var i=0;
while (i<10) {
        document.write("Der Wert von i beträgt: " + i + "<br />");
        i++;
}
// Zur Kontrolle des Variablen-Wertes noch die Ausgabe von i am Ende:
document.write("Hier ist die Schleife zuende.<br />");
document.write("Der Wert von i beträgt nun: " + i + "<br />");
```

Zum Vergleich noch mal die **for**-Schleife.

```
for (var i=0;i<10;i++) {
        document.write("Der Wert von i beträgt: " + i + "<br />");
}
// Zur Kontrolle des Variablen-Wertes noch die Ausgabe von i am Ende:
document.write("Hier ist die Schleife zuende.<br />");
document.write("Der Wert von i beträgt nun: " + i + "<br />");
```

Ein wesentlicher Unterschied zwischen den Schleifen ist der: bei der **while**-Schleife muss die Bedingung beeinflusst werden, in unserem Beispiel durch Ändern der Variable i. Bei der **for**-Schleife darf die Variable durch die Anweisungen *auf keinen Fall* geändert werden.

Die **for**-Schleife ist daher erheblich übersichtlicher, wohingegen eine **while**-Schleife beliebig kompliziert werden kann. Einer der beliebtesten Fehler ist es, die Abbruchbedingung der **while**-Schleife „zu versaubeuteln“, dann läuft die Schleife ewig und das Programm reagiert nicht mehr.

Übung 10.1

1. Schreibe ein Programm, das den Besucher der Web-Seite zur Eingabe eines Passwortes auffordert. Ist das Passwort richtig, dann gib eine entsprechende Meldung aus. Ist das Passwort falsch, gib erneut eine Passwortabfrage aus. Gib dem Besucher außerdem die Möglichkeit abzubrechen!

Eine Variante der **while**-Schleife ist die **do-while**-Schleife. Wie immer zuerst die Syntax:

```
do {
        Anweisungen;
} while (Bedingung);
```

Im Gegensatz zur **while**-Schleife wird hier die Schleife immer erst einmal durchlaufen.

Die Schleifen lassen sich natürlich alle ineinander überführen, welche man letztlich benutzt ist auch eine Frage des persönlichen Stils, allerdings erhöht es die Lesbarkeit des Programmes erheblich, wenn die am besten geeignete Schleife benutzt wird.

10.2 Rückgabewerte von Funktionen

In Abschnitt 9.2.1 habe ich die Unterschiede zwischen globalen und lokalen Variablen erläutert, und außerdem die Nachteile von globalen Variablen herausgestellt. Eine Möglichkeit globale Variablen zu vermeiden, ist die Benutzung von Rückgabewerten.

In dem folgenden Skript werden wir keine globalen Variablen benutzen, sondern nur Rückgabewerte. Das Math-Objekt kennst du ja bereits. Hier brauchen wir eine Zufallszahl (*engl.* random= zufällig). Math.random erzeugt allerdings eine Zufallszahl zwischen 0 und 1 (die 1 ausschliessend), so dass wir zunächst mit 11 multiplizieren und hinterher mit Math.floor die Nachkommastellen abschneiden, um Zahlen zwischen 0 und 10 zu erhalten.

Dir wird außerdem der Befehl **break** auffallen. Mit **break** kann eine Schleife beendet werden, die Programmausführung wird dann hinter der Schleife fortgesetzt.

```
<html>
<head>
  <title>Zahlen erraten</title>
  <script language="JavaScript" type="text/javascript">

function zufallszahl()
{
        var zufall=Math.random() * 11;
        zufall=Math.floor(zufall);
        return(zufall);
}
function neues_spiel()
{
        var ratezahl=zufallszahl();
        for(var c=1; c<=10;c++){
                var geraten=window.prompt("Dies ist dein " +
                                c + ". Versuch!");
                if(geraten==ratezahl){
                        window.alert(c + ". Versuch! Gut geraten!");
                        break;
                }
                if (geraten==null){
                        window.alert("... Good Bye ...");
                        break;
                }
                if(c==10){
                        window.alert("Die gesuchte Zahl war "
                                + ratezahl);
                }
        }
}

</script>
</head>
<body>
  <form name="formular">
  Erraten Sie die gesuchte Zahl! Sie liegt zwischen 0 und 10. <br />
  <input type="button" name="neu" value="Neues Spiel, neues Glück!"
          onClick="neues_spiel()">
  </form>
</body>
```

```
</html>
```

Die Verwendung von **break** ist aus programmiertechnischer Sicht sehr problematisch. Wenn ich eine Bedingung dafür formulieren kann, die Schleife zu verlassen, warum setzte ich sie dann nicht in der Schleifenbedingung ein? Wer **break** benutzt zeigt nur, dass er es nicht besser kann.[7] Daher werden wir uns im nächsten Abschnitt darum kümmern, wie man das **break** vermeiden kann. Aber zunächst wieder eine

Übung 10.2

1. Schreibe das Programm so um, dass der Benutzer selbst die Obergrenze für die Zufallszahlen wählen kann.
2. Gib dem Besucher die Möglichkeit, so lange zu raten bis er entweder abbricht, oder das richtige Ergebnis erhalten hat. Du darfst hier kein **break** benutzen, benutze am besten die **do−while**-Schleife!

11 Mehrfachauswahlen

In diesem Kapitel wirst du erfahren, wie man eine Entscheidung nicht nur zwischen zwei, sondern zwischen vielen Möglichkeiten treffen kann. In einem Beispiel wird gezeigt, wie man die Besucher der Web-Seite in Abhängigkeit von einer bestimmten Bedingung auf eine andere Web-Seite weiterleiten kann.

11.1 switch

Häufig ist es so, dass es nicht nur zwei Auswahlmöglichkeiten gibt, sondern viele verschiedene. Das kann man natürlich auch mit **if** bzw. **else** lösen, wenn man diese Abfragen nur tief genug ineinander schachtelt. Aber es gibt eine viel bessere Möglichkeit, **switch**[8].

```
switch(variable) {
 case "wert1": Anweisungen 1;
 break;
 case "wert2": Anweisungen 2;
 break;
  .
  .
  .
 default: Anweisungen sonst;
 break;
}
```

Wenn die Variable also `"wert1"` enthält, werden die Anweisungen 1 ausgeführt, bei `"wert2"` die Anweisungen 2, bei **default** sonstige Dinge. Wichtig ist dabei auch das **break** am Ende jedes Falls. Wenn **break** weglassen wird, werden nämlich alle nachfolgenden Fälle auch ausgeführt, aber das will man ja in der Regel nicht.

Für den Fall, dass keiner der definierten Fälle zutrifft, *kann* am Ende der Fallunterscheidung der Fall **default**: definiert werden. Die darunter stehenden Anweisungen werden ausgeführt, wenn keiner der anderen Fälle zutrifft. **default** ist aber optional.

Ein Beispiel (nach [10]):

[7] Zugegeben: Es gibt Fälle wo ein **break** angezeigt ist. Deshalb habe ich es auch hier aufgenommen. Aber sinnvolle Anwendungen gibt es nur wenig.

[8] In einigen anderen Programmiersprachen heist das **case**


```
//Unterschiedliche Begrüssung je nach Wochentag,
//Beachte Sonntag=0, Montag=1, Dienstag=2, etc.

var datum=new Date();
var tag=datum.getDay();
switch (tag)
{
case 5:
  document.write("Endlich Freitag");
  break;
case 6:
  document.write("Super es ist Samstag");
  break;
case 0:
  document.write("Ausschlafen, es ist Sonntag");
  break;
default:
  document.write("Ich freue mich auf das Wochenende");
  break;
}
```

11.2 Weiterleitung

Eine solche **switch**-Anweisung wird (leider) recht häufig verwendet, um die Besucher der Seite je nach ihrem verwendeten Browser auf eine speziell angepasste Seite weiterzuleiten. Dazu wird zunächst der Browsername abgefragt. Diese Abfrage ist mittlerweile bei den vielen verschiedenen Browserversionen kompliziert, und auf keinen Fall empfohlen [3]. Wir werden daher hier den Besucher je nach Wochentag auf eine andere Seite weiterleiten.

Alle Angaben zur URL der aktuellen Seite sind im location-Object gespeichert, das wiederum ein Unterobjekt des window-Objectes ist. Auf das Objekt greift man also mit window.location zu[9]. Durch einfaches Zuweisen zur Eigenschaft window.location.href wird eine neue Seite geladen.

```
    var datum = new Date();
    var tag = datum.getDay();
        switch (tag)
        {
        case 5:
          window.location.href="freitag.html";
          break;
        case 6:
          window.location.href="samstag.html";
          break;
        case 0:
          window.location.href="sonntag.html";
          break;
        }
```

Übung 11.2

1. Suche dir vier verschiedene beliebige Seiten im Internet aus. Leite jeweils auf eine dieser Seiten weiter. Erweitere das Script, so dass an einem Werktag der Besucher ebenfalls umgeleitet wird.

[9] Da das window-Objekt allerdings das Standard-Objekt ist, kann man auch nur location benutzen.

2. Schreibe das Script nun so um, dass du die URLs in einem Array speicherst und aus dem Array abrufst.

3. Erweitere dein Script, so dass an einem Werktag eine von mehreren zufälligen Adressen abgerufen wird, auf keinen Fall aber eine von den Adressen für Freitag, Samstag und Sonntag. Für die Generierung von ganzzahligen Zufallszahlen kannst du Math.random und Math.floor benutzen (Siehe auch Seite 21).

---

12 Eigene Objekte

In diesem Abschnitt wirst du lernen, wie du eigene Objekte definieren kannst. Du wirst lernen wofür eigene Objekte gut sind und wie Objekte aufeinander aufbauen können. Du musst unbedingt diese Scripte selbst eingeben, die notwendigen Graphiken kannst du dir selbst erzeugen.

12.1 Definition eines eigenen Objektes

Die Definition eines Objekt (sein Prototyp) wird in JavaScript durch eine Funktion realisiert. Um das Objekt dann benutzen zu können ist eine Instanz dieses Objektes notwendig. Eine Instanz wird durch Zuweisen zu einer Variable mit dem Schlüsselwort **new** erzeugt. Du kennst das bereits z.B. von den Arrays. Im folgenden werden wir ein Objekt definieren, welches den Namen (die Quelle der Bilddatei) und die Eigenschaften eines Bildes speichern kann.

```
  function pimage(graphik, id, height,width) {
    this.graphik = graphik;
    this.id = id;
    this.height = heigth;
    this.width = width;
  }
  var meinimage = new pimage("3dball.gif","img1","20","20");
  var mein_grosses_image = new pimage("3dball.gif","img2","40","40");
```

Die Funktion kann beliebig benannt werden, ich habe die Funktion pimage genannt, da es sich dabei um einen Prototypen eines Objektes handelt. Durch das Schlüsselwort **this** werden die Eigenschaften des Objektes definiert. Eigenschaften entsprechen also Variablen, die nur innerhalb eines Objektes gültig sind. Die Zuweisungen innerhalb der Funktion werden bei Erzeugen der Instanz des Objektes ausgeführt. Man nennt dies auch einen Konstruktor.

Nach der Objektdefinition werden zwei Instanzen des Objektes erzeugt. Diese Variablen habe ich mit „mein“ beginnen lassen, um anzuzeigen dass es sich dabei um Objektinstanzen handelt. Jetzt müssen wir noch zwei Bilder in HTML erzeugen, die die tatsächliche Anzeige der Bilder übernehmen. Diese Bilder erhalten eine sogenannte ID, über die sie dann identifiziert werden können. Die Funktion document.getElementById() sorgt dann dafür, dass die Bilder über JavaScript verändert werden können.

```
<body>
<img scr="" id="img1" />
<img scr="" id="img2" />

<script language="javascript" type="text/javascript"><!--

  function pimage(graphik, id, height,width) {
    this.graphik = graphik;
```



```
    this.id = id;
    this.height = height;
    this.width = width;
  }
  var meinimage = new pimage("3dball.gif","img1","20","20");
  var meingimage = new pimage("3dball.gif","img2","40","40");

  // Hier werden die Bilder jetzt angezeigt
  var bild1=document.getElementById("img1");
    bild1.src=meinimage.graphik;
    bild1.heigth=meinimage.heigth;
    bild1.width=meinimage.width;
  var bild2=document.getElementById("img2");
    bild2.src=meingimage.graphik;
    bild2.heigth=meingimage.heigth;
    bild2.width=meingimage.width;

//--></script>
</body>
```

Die Variable meinimage nimmt das erste Bild auf, die Variable meingimage das gleiche Bild, aber doppelt so groß.

Es wäre allerdings viel besser, wenn die Bilder sich selbst anzeigen könnten. Dazu brauchen wir eine Methode, nennen wir sie zeige().

Methoden eines Objektes sind Funktionsaufrufe innerhalb der Objektdefinition, die wieder über das Schlüssewort **this** an das Objekt gebunden werden.

Zusätzlich zur Methodendefinition werden wir unser Ojekt noch weiter verändern, es soll nämlich gleich einen Verweis auf das Bild aufnehmen. Dann müssen wir die Eigenschaften width und heigth auch nicht mehr separat speichern, schließlich hat das Bild ja bereits diese Eigenschaften.

```
<body>
<img scr="" id="img1" />
    <!-- Hier wird das erste Bildobjekt erzeugt -->
<img scr="" id="img2" />
    <!-- Hier wird das zweite Bildobjekt erzeugt -->

<script language="javascript" type="text/javascript"><!--

 function _zeige() {
    this.bild.src=this.graphik;
 }

 function pimage(graphik, id, height,width) {
    this.graphik = graphik;
    this.id = id;
    this.bild=document.getElementById(this.id);
    this.bild.height=height;
    this.bild.width=width;
    this.zeige = _zeige; // Methodendefinition von pimage.zeige
 }
  var meinimage = new pimage("3dball.gif","img1","20","20");
  var meingimage = new pimage("3dball.gif","img2","40","40");

  // Hier werden die Bilder jetzt angezeigt
```

```
    meinimage.zeige();
    meingimage.zeige();

//--></script>
</body>
```

Beachte, dass bei der *Methodendefinition* die Funktion _zeige ohne Klammern aufgerufen wird. Die Funktion _zeige sollte niemals direkt aufgerufen werden, sondern immer nur von dem Objekt. Allerdings kann man den direkten Aufruf nicht verhindern, er ergibt dann aber keinen Sinn, denn **this**.graphik usw. wären dann nicht definiert.

Als nächstes wollen wir das Bild wieder verstecken. Dazu benutzen wir ein transparentes GIF, welches das Bild ersetzt. Wir definieren uns also eine Methode verstecke, benötigen aber noch eine weitere Eigenschaft, nämlich sichtbar. Diese Eigenschaft muss sowohl in zeige als auch in verstecke gesetzt werden.

```
function _verstecke(){
    if (this.sichtbar==true){
      this.bild.src="transparent.gif";
      this.sichtbar=false;
    }
  }
  function _zeige() {
    if (this.sichtbar==false) {
      this.bild.src=this.graphik;
      this.sichtbar = true;
    }
  }

  function pimage(graphik, id, height,width) {
    this.graphik = graphik;
    this.id = id;
    this.sichtbar = false;
    this.bild=document.getElementById(this.id);
    this.bild.height=height;
    this.bild.width=width;
    this.zeige = _zeige;
    this.verstecke = _verstecke;
  }

  var meinimage = new pimage("3dball.gif","img1","20","20");
  var meingimage = new pimage("3dball.gif","img2","40","40");
  meinimage.zeige();
  meingimage.zeige();
  window.setTimeout("meinimage.verstecke()", 2000);
   // Damit wir das Verstecken auch sehen können erst nach 2 Sekunden
  window.setTimeout("meinimage.zeige()", 4000);
   // Und wieder sichtbar
```

Übrigens: Vorsicht vor setTimeout(). Das ist keine Warte-Funktion, nach dem Aufruf wird das übrige Script direkt fortgesetzt. Würden beide Timeouts auf z.B. 2000 ms gesetzt werden, wäre es ziemlicher Zufall welche der beiden Funktionen zuerst ausgeführt würde. Wir könnten wegen unserer Objekt-Definition jetzt natürlich auch das andere Bild verstecken.

Nun zu unserer ersten Übung:

Übung 12.1

1. Füge eine Methode swap() hinzu, die die Sichtbarkeit ändert. Benutze in swap() **this**.zeige() und **this**.verstecke().

12.2 Wozu sind Objekte eigentlich gut?

Einige Vorteile der objektorientierten Programmierung sollten dir jetzt schon aufgefallen sein.

- Objekte kapseln ihre Variablen ein. Die Zahl globaler Variablen sinkt.
- Das Objekt kennt seinen eigenen Status, der Programmierer muss sich um den Zustand des Objektes nicht kümmern.
- Wie das Objekt bestimmte Operationen realisiert, ist bei der Benutzung des Objektes egal. Man kann das Objekt intern ändern, ohne dass man das gesamte übrige Programm ändern muß.
- Objekte können in anderen Programmteilen oder Programmen leicht weiterverwertet werden.
- Objekte können sich selbst initialisieren, also ihre internen Eigenschaften mit sinnvollen Werten vorbelegen.
- Objekte können voneinander erben.
- Objekte können ihre Methoden sogar vollständig kapseln, d.h. diese so definieren, dass sie nicht direkt aufgerufen werden können.

Die letzten drei Punkte sollen hier nacheinander vorgeführt werden.

12.3 Vorbelegung von Objekteigenschaften

Als Beispiel soll der – hier verkürzt dargestellte – Konstruktor von pimage dienen.

```
function pimage(graphik, id, height,width,src) {
  this.graphik = graphik;
  this.id = id;
  this.sichtbar = false; // Vorbelegung mit einem festen Wert
  this.bild=document.getElementById(this.id);
  this.bild.width = width;
  this.bild.height = height;
  this.bild.src = src || "transparent.gif";
    // Vorbelegung alternativ
}
var meinimage = new pimage("3dball.gif","img1","20","20");
    // Instantiierung ohne uebergebene Graphik
var meingimage = new pimage("3dball.gif","img1","20","20","blau.gif");
    // Instantiierung mit uebergebener Graphik
```

Wir sehen hier zwei Fälle der Vorbelegung. **this**.sichtbar wird mit einem festen Wert belegt, der grundsätzlich gilt. **this**.bild.src erhält entweder den Wert der Variablen src, sofern eine solche Variable beim Aufruf übergeben wird, oder den Wert `"transparent.gif"`. Das funktioniert so:

Das Zeichen || bedeutet „logisches oder“. Ist die Variable src nicht definiert ist die linke Seite von || falsch, dann wird also die rechte Seite genommen und **this**.bild.src zugewiesen.

Der Unterschied von meinimage und meingimage ist also, dass das kleine Bild nach dem Aufruf von **new** nicht angezeigt wird (bzw. das Bild `"transparent.gif"` wird angezeigt), statt des größeren Bildes wird das Bild `"blau.gif"` für meingimage angezeigt.

Übung 12.3

1. Erweitere pimage um eine Methode skaliere(). Die Methode soll einen Parameter haben, den Skalierungsfaktor. Wird die Methode ohne Skalierungsfaktor aufgerufen, wird die Bildgroesse verdoppelt. Die Bildgroesse muss ganzzahlig sein, dafür kannst du aber einfach runden.

12.4 Vererbung

Bisher können wir unser Bild zeigen, verstecken und skalieren. Nun wollen wir einige Bilder auch noch über den Bildschirm bewegen. Dazu könnten wir unser Objekt erweitern, oder wir schaffen uns ein neues Objekt, das zusätzlich zu den bisher definierten Eigenschaften und Methoden noch die Methode bewege() enthält. Dieses Script funktioniert *nicht* mit dem IE 5.0, aber mit Mozilla und dem IE 6.0.

```
function _bewege(x,y){
    this.bild.style.left = x+"px";
    this.bild.style.top = y+"px";
}

function pmoveimage (graphik,id,height,width,src){
  this.base = pimage; // Hier wird pimage Elter von pmoveimage
  this.base(graphik, id, height, width,src);
                // Und hier wird der Konstruktor
                // von pimage aufgerufen
  this.bild.style.position = "absolute";
  this.bewege = _bewege;
}

var meinimage = new pimage("3dball.gif","img1","20","20");
var meinmoveimage = new pmoveimage("3dball.gif","img2","40","40");
meinimage.zeige();
meinmoveimage.zeige();
window.setTimeout("meinmoveimage.bewege(100,100)",2000);
```

Jetzt wollen wir mal zwei Graphiken annimieren. Ausgerechnet das geht in JavaScript leider nicht objektorientiert (bzw. nur mit [1]), sondern mit Hilfe einer Funktion.

```
function rennen(aufrufe) {
   if (aufrufe<250) {
      aufrufe++;
      meinmoveimage1.bewege(aufrufe*2,aufrufe*2);
      meinmoveimage2.bewege(aufrufe,aufrufe);
      window.setTimeout("rennen("+aufrufe+")",1);
   }
}

rennen(0);
```

Du solltest das obige Beispiel unbedingt ausprobieren, nachher kommt dazu noch eine kompliziertere Übung.

Übung 12.4

1. Ergänze pmoveimage um eine Methode bewegerel, die das Bild um eine relative Strecke von ihrem Ausgangspunkt aus bewegt. Du benötigst dazu auch noch eine Eigenschaft ursprung, die den Bezugspunkt für das Bild enthält. ursprung muss in _bewege gesetzt werden.

2. Denke dir eine kompliziertere Bewegung aus. Ergänze damit die Animation aus dem Beispiel. Ein Tip: für periodische Bewegungen eignet sich die Sinus-Funktion hervorragend! Wenn du ein ziemlich kompliziertes Beispiel sehen willst, schau dir mal [6] an.

12.5 Verstecken von Methoden

Um Methoden in einem Objekt zu verstecken, kann man sie anonym, also ohne Funktionsnamen definieren.

```
function objekt (){
   this.anonymous = function () {}
}
```

this.anonymous kann nur über das Objekt aufgerufen werden, da die Funktion nirgendwo sonst definiert ist. Jetzt ein Beispiel mit einer anonymen und einer benannten Funktion.

```
function _mynamed (num) {
  return this.attribut+num;
}

function objekt () {
  this.anonymous = function (a,b,c) {return (a+b+this.named(c));}
  this.named = _mynamed;
  this.attribut = 5;
}

var meins = new objekt();
document.write(meins.anonymous(1,2,3));
```

Man muss dabei ein bißchen Vorsicht walten lassen, dazu ein nicht funktionierendes Beispiel. Das Problem ist das **this** in _mynamed. In disem Beispiel ist die Funkion _mynamed nicht an das Objekt gebunden, daher hat **this** nicht die gewünschte Bedeutung (nämlich gar keine). **this**.attribut ist hier in der Funktion _mynamed nicht definiert.

```
function _mynamed (num) {
  return this.attribut+num;  // Hier krachts!
}

function objekt () {
  this.anonymous = function (a,b,c) {return (a+b+_mynamed(c));}
  this.attribut = 5;
}

var meins = new objekt();
document.write(meins.anonymous(1,2,3));
```

Das Ergebnis ist NaN, d.h. „Not a Number“. Logisch, da **this**.attribut in _mynamed keine Zahl ist.

13 Das Document Object Model (DOM)

Du hast jetzt schon viele verschiedene Spracheigenschaften von JavaScript kennengelernt. In diesem und dem nächsten Kapitel geht es nicht so sehr um die Programmiersprache selbst, sondern um die moderne Schnittstelle zwischen der WebSeite und JavaScript. Ansonsten werden wir das (hoffentlich) Gelernte anwenden.

13.1 Eine Webseite nachträglich verändern

Bisher konnten wir nur Objekte auf einer Webseite verändern, die bereits im Quelltext der Webseite vorhanden waren. Oder wir benutzten document.write, um Text auf der Seite auszugeben. Jetzt werden wir eine Methode kennenlernen, um nachträglich *jedes* Stück der Seite anzusprechen, auszulesen, zu verändern, zu erstellen und zu löschen.

Eine Webseite besteht vollständig aus Knoten (Nodes).

Es wird zwischen ElementNodes – z.B. body, div, p – AttributNodes und TextNodes unterschieden. TextNodes sind zwischen HTML-Elementen stehende Textbereiche, also das, was nachher auf der Webseite angezeigt wird.

Die Nodes sind baumartig angeordnet[10]. Der WurzelNode aller anderen Nodes ist html. An dem WurzelNode sitzt ein Kindknoten (childNode), z.B. body. Am ChildNode body sitzten viele weitere Nodes, z.B. Text oder andere HTML-Elemente. In den Elementen können wieder andere Elemente oder Text als childNodes stehen usw.

Nodes (außer html) haben stets einen Vaterknoten, den parentNode. Sind Nodes parallel zueinander, heißen sie sibling[11]. An einem Beispiel sollen diese doch etwas schwierigen Zusammenhänge erläutert werden.

```
1  <body id="alles">
2  <h1>Überschrift a</h1>
3     Dies ist ein Text.
4  <h2>Auch ne Überschrift</h2>
5
6  <h1>Überschrift b</h1>
7
8  <table>
9  <tr>
10    <td><h1>Überschrift in einer Tabelle</h1></td>
11 </tr>
12 </table>
13
14 <div><h1>Überschrift in einem Div</h1></div>
15 </body>
```

In Abbildung 1 auf Seite 31 ist die Web-Seite aus Sicht des DOM dargestellt. Allerdings ist die Darstellung noch nicht ganz vollständig, denn einige TextNodes habe ich weggelassen. Schließlich ist auch ein Leerzeichen z.B. nach dem <**body**> und vor dem ersten <**h1**> in der Webseite ein TextNode. Dann fällt im DOM noch in der Tabelle das <tbody> auf, dieses wird vom Browser automatisch eingefügt. Wir schauen uns jetzt an, wie man die einzelnen Nodes im DOM erreichen und manipulieren kann.

[10] Mozilla besitzt in der Sidebar den DOM-Inspektor. Dort kann man sich die Struktur der Seite wunderbar anzeigen lassen. Man kann das Hinzufügen und Löschen von Nodes beobachten, man kann Nodes sogar aus dem DOM-Inspektor heraus löschen. Der DOM-Inspektor kann unter `Bearbeiten/Einstellungen.../Erweitert/DOM Inspektor` installiert werden.

[11] Sibling bedeutet Geschwister.

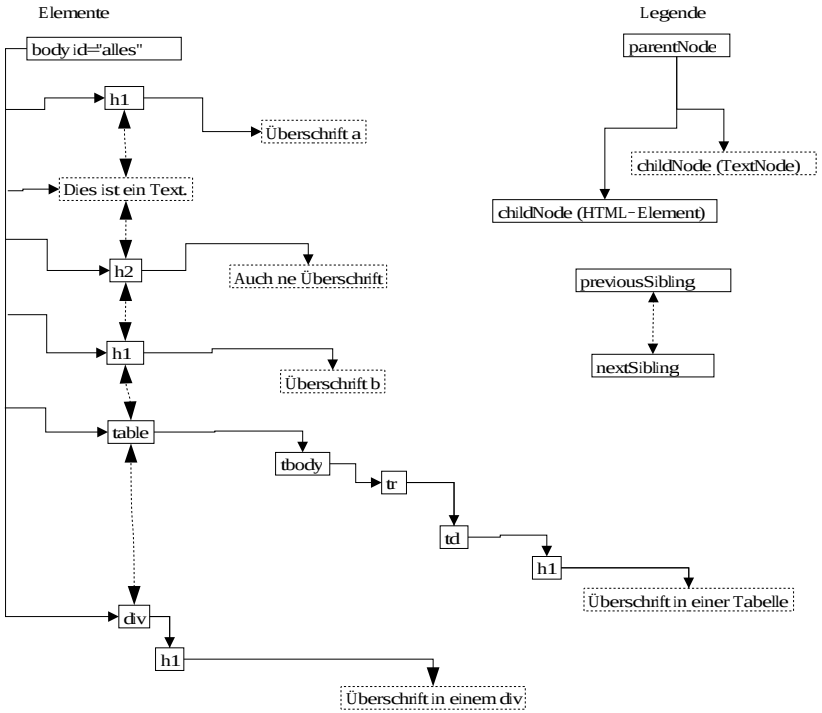

Abbildung 1: Einige HTML-Elemente, etwas Text und die Stellung im Document-Object-Model

13.2 Bewegen im DOM

Ich liste einige zum Bewegen im DOM-Baum nötige Methoden und Eigenschaften auf, soweit ihr sie im folgenden Beispiel benötigt. Ansonsten schaut bitte bei [5] nach.

document.getElementById(*"ID"*) Diese Funktion liefert den Node für das HTML-Element mit der ID *ID* zurück. Mit diesem Node können wir jetzt weiterarbeiten.

***Node*.childNodes.length** Wenn man sich über getElementByID() einen Verweis auf den entsprechenden *Node* besorgt hat, erhält man so die Anzahl der ChildNodes. Diese ChildNodes kann man nun z.B. in einer **for**-Schleife durchlaufen.

***Node*.nodeName** Enthält bei HTML-Elementen das HTML-Tag als Zeichenkette (z.B. `H1`). Bei TextNodes enthält es die Zeichenkette `#text`. Es ist allerdings total bescheuert, dass an dieser Stelle die HTML-Elemente *groß* geschrieben werden. Überall sonst Kleinbuchstaben, nur hier nicht. grrr...

***Node*.nodeValue** Bei HTML-Elementen enthälte es stets `null`, bei TextNodes den tatsächlichen Text.

***Node*.hasChildNodes** Ein Wahrheitswert der angibt, ob der *Node* ChildNodes hat.

***Node*.childNodes[i]** Ein Verweis auf den i-ten ChildNode.

***Node*.firstChild** Ein Verweis auf den erstenChildNode des *Node*.

Jeder Node kann auf mehrere Wege erreicht werden. Welche Methode die günstigste ist, muss der Programmierer entscheiden.

13.3 Wie man Überschriften numeriert

Um mit dem DOM umzugehen, wollen wir eine Aufgabe angehen, die bisher in HTML nicht möglich war. Das Ziel wird sein, alle Überschriften eines Dokumentes in der richtigen Reihenfolge durchzunummerieren. Wir wollen zunächst das Beispiel von Seite 30 bearbeiten. Zunächst sollen die Überschriften in der Tabelle bzw dem nicht numeriert werden. Das Vorgehen wird etwa so sein:

1. Wir suchen den übergeordneten Node im Dokument, in diesem Fall ist das body.
   var body = document.getElementById(`"alles"`);
2. Mehrere Zähler für die Überschriften müssen initialisiert werden. Noch eleganter könnte man dies mit einem Array realisieren.
3. In einer **for**-Schleife wird body.childNodes.length durchlaufen.
4. Wenn eine Überschrift gefunden wird, z.B. folgendermaßen:
   body.childNodes[i].nodeName.toLowerCase() == `"h1"`
   oder noch besser mit einer **switch**-Anweisung, muss
5. der Überschrifttext ausgelesen, um den Zähler ergänzt und wieder eingefügt werden. Der Überschrifttext ist ChildNode zum Überschriftelement.
   body.childNodes[i]. firstChild .nodeValue = j + `""` + body.childNodes[i]. firstChild .nodeValue;
   Diese Anweisung gilt z.B. für Überschriften der ersten Ebene, wenn j der Zähler für die erste Ebene ist.
6. Nun muß der Zähler (die Zähler) entsprechend heraufgezählt oder heruntergesetzt werden, das kommt auf den Überschrifttyp an.

Übung 13.3

1. Schreibe ein Programm, welches die Überschriften der ersten, zweiten und dritten Stufe numeriert. Die erste Stufe soll mit 1, 2 usw. numeriert werden, die zweite Stufe mit 1.1, 1.2, 2.1, 2.2 usw. Ergänze dazu die Webseite um einige weitere Überschriften der ersten, zweiten und dritten Stufe.
2. Jetzt schreibe das Programm so um, dass du zum Numerieren eine Funktion benutzt. Diese Funktion (nenn sie enumerate()) soll einen Aufrufparameter haben, nämlich den Startknoten. Wenn du am Ende der Funktion abfragst ob es einen ChildNode gibt, kannst du die Funktion sich wieder selbst aufrufen lassen, diesmal mit dem ChildNode als Startknoten. Auf diese Weise kannst du alle Nodes der Seite durchlaufen, und alle Überschriften numerieren.
3. *Für Fortgeschrittene:* Mit den Funktionen aus Kapitel 14 kannst du neue Nodes in die Seite einfügen. Am Anfang der Seite soll ein Inhaltsverzeichnis eingefügt werden. Die Einträge im Inhaltsverzeichnis sollen dem Text der dazugehörigen Überschriften entsprechen. *Hinweis: Halte beim Umbenennen der Überschriften deren Text in einem Array fest. Erzeuge das Inhaltsverzeichnis zum Schluß aus dem Array.*[12]

14 Ein Graphikprogramm in JavaScript

Wir wollen nun ein Graphikprogramm erstellen. Natürlich wird dieses Graphikprogramm objektorientiert programmiert. Zur Steuerung benützen wir ein Formular, welches nach der Eingabe der notwendigen Parameter – z.B. Breite und Höhe, Farbe usw. – verschiedene Objekte wie Punkte, Linien, Rechtecke und Kreise auf den Bildschirm zeichnet.

Das beste HTML-Element welches wir dafür verwenden können, ist ein div. Dem div kann über Styles eine Hintergundfarbe zugewiesen werden, es kann in Höhe und Breite verändert werden und man kann es an einer beliebigen Stelle positionieren. Mit divs können wir also gefüllte Rechtecke zeichnen, alle anderen Elemente müssen wir aus mehreren divs zusammensetzen.

Jetzt werden wir neue Nodes erzeugen, sie in den Baum „einhängen“ und wieder löschen. Dazu benötigen wir einige Funktionen.

document.createElement(*"TagName"*) Der ElementNode wird erzeugt, z.B. ein div. Rückgabewert der Funktion ist ein Verweis auf den Knoten. Ein Beispiel:
var mein_knoten = document.createElement(`"div"`);

document.createTextNode(*"Mein Text"*) Ein TextNode wird erzeugt. Rückgabewert der Funktion ist ein Verweis auf den Knoten.

***Node*.appendChild(*"Verweis auf neuen Node"*)** Jetzt wird der neue *Node* in den Dokument-Baum eingefügt. Ist *Node* der Verweis auf den body, wird das Element so angezeigt, als ob wir es direkt in HTML in den body hineingeschrieben hätten.

***Node*.removeChild(*"Verweis auf ChildNode"*)** *Node* ist der Verweis auf den ParentNode, der ChildNode wird dann gelöscht und ist am Bildschirm auch nicht mehr zu sehen.

14.1 Das Grundgerüst der Webseite

```
1 <head>
2 <title>Graphikprogramm in JavaScript</title>
3 <script type="text/javascript">
```

[12] Diese Aufgabe und die vorige stammt aus [4, S. 197]. Dort findet man noch mehr Hinweise zur Lösung und weitere – auch schwierigere – Aufgaben.

```
/*
  In diesem Scriptabschnitt definieren wir
  unsere eigentlichen Zeichenfunktionen
*/

function _Point(x,y,width,height){
/* Erzeugt einen "Punkt" mit übergebener  x- und y-Position
   sowie entsprechender Breite und Höhe.
*/
  var Node = document.createElement("div");
  /* Wir erzeugen ein div, welches die entsprechenden
  Attribute zugewiesen bekommt. */
  Node.style.backgroundColor =this.Color;
  Node.style.position = "absolute";
  Node.style.top = y + "px";
  Node.style.left = x + "px";
  Node.style.width = width + "px";
  Node.style.height = height + "px";
  this.Canvas.appendChild(Node); // Jetzt wird das div sichtbar
  this.Nodes.push(Node); // Wir speichern den Node,
                         // um das div später noch bearbeiten zu können

}

function _removeNodes() {
    for (var i=0;i<this.Nodes.length;i++){
      this.Canvas.removeChild(this.Nodes[i]);
    }
    delete(this.Nodes); /* Jetzt werfen wir das ganze
           Array weg, damit wir nicht jedes Element
                  einzeln löschen müssen */
    this.Nodes = new Array(); /* und erzeugen es neu,
           um wieder neue Elemente einsetzen zu könnnen. */
}

function Graphics(cv,defaultColor){
/* Das Graphics-Objekt
   Im Parameter cv wird die ID der Zeichenfläche übergeben.
   Im Parameter defaultColor die zu benutzende Farbe.
*/
    this.Nodes = new Array(); // Hier werden alle erzeugten
                              // Zeichenobjekte gespeichert
    this.Point = _Point;
    this.remove = _removeNodes;
    this.Color = defaultColor;
    this.Canvas = document.getElementById(cv);
}

</script>
</head>

<body>
<form action="" ID="formular">
```



```
    <table cellspacing="0">
    <tr align="right">
      <td><input type="button" value="drawPoint()"
                 onclick="DRAW(this.value)"></td>
      <td>x1<input name="dp_x1" type="text" size="5" maxlength="4"
                 value="2"> </td>
      <td>y1<input name="dp_y1" type="text" size="5" maxlength="4"
                 value="2"> </td>
      <td>Breite: <input name="dp_w1" type="text" size="5" maxlength="4"
                 value="2"> </td>
      <td>Höhe: <input name="dp_h1" type="text" size="5" maxlength="4"
                 value="2"> </td>
    </tr>
    <tr align="right">
      <td><input type="button" value="remove()"
                 onClick="DRAW(this.value)"></td>
    </tr>
    </table>
</form>

<div id="Canvas" style="position:relative;height:250px;width:100%;"></div>

<script type="text/javascript">

var myGraph=new Graphics("Canvas","#0000ff");
     // Hier wird das Graphik-Objekt erzeugt.

function DRAW(anweisung) {
  var f=document.forms[0]; // Eine weitere Möglichkeit einen Verweis auf
                           // ein Element zu bekommen, hier auf das erste
                                                    // Formular der Seite.
  switch (anweisung) {
      case "drawPoint()" : myGraph.Point(parseInt(f.dp_x1.value),
            parseInt(f.dp_y1.value),parseInt(f.dp_w1.value),
                parseInt(f.dp_h1.value));
    break;
      case "remove()": myGraph.remove();
    break;
  }
}

</script>

</body>
</html>
```

Übung 14.1

1. Ergänze das Formular um ein Eingabefeld und einen Button, um die Farbe der Objekte zu ändern. Schreibe nun eine Methode setColor(), die die Farbe aller gezeichneten Objekte verändert. *Hinweis: Dazu muss in einer Schleife* ***this****.Nodes durchlaufen werden, so ähnlich wie in _removeNodes().*

2. Wie kann man eine horizontale/vertikale Linie beliebiger Breite erstellen? Wie ein leeres Rechteck mit beliebiger Liniendicke? Welche Eingabefelder sind dafür notwendig?

3. Erweitere das Programm (und das Formular) um vertikale und horizontale Linien. Aus diesen Linien kannst du dann leere Rechtecke erstellen. (jeweils mit beliebiger Liniendicke). *Hinweis: Beachte bitte auch die Fälle, in denen $x_2 \leq x_1$ und/oder $y_2 \leq y_1$. Was passiert bei $x_2 \neq x_1$ und $y_2 \neq y_1$, also einer falschen Eingabe?*

4. Erweitere das Programm um die Möglichkeit, Text an beliebigen Stellen des Bildschirms anzuzeigen. Dafür muss in dem zur Positionierung ein TextNode erzeugt werden.

14.2 Wie zeichnet man schräge Linien? (1. Versuch)

Was zunächst ganz einfach erscheint – das Zeichnen von schrägen Linien – beweist doch seine Tücke im Detail. Wenn wir uns an den Mathe-Unterricht erinnern, konnte man doch Geraden durch die Geradengleichung bestimmen.[13]

$$y = m * x + b$$

Dabei war m die Steigung und b der y-Achsenabschnitt. Die Steigung ist leicht zu berechnen:

$$m = (x_2 - x_1)/(y_2 - y_1)$$

Der y-Achsenabschnitt ist einfach y_1.

Jetzt könnte man so vorgehen:

1. Berechne die Steigung.
2. Durchlaufe eine Schleife mit dem Zähler i von 0 bis $x2 - x1$. In der Schleife berechne:
   a) $x = i + x_1$
   b) $y = m * i + y_1$
3. Setze einen Punkt.

Das geht auch verhältnismäßig gut, solange $x_2 > x_1$. Wenn das nicht der Fall ist, können wir aber die Koordinaten tauschen. Es spielt ja für unser Programm keine Rolle ob wir die Linie von rechts nach links oder von links nach rechts zeichnen.

Übung 14.2

1. Erweitere das Objekt um eine Methode SingleLine(x1,y1,x2,y2), die schräge Linien mit einem Pixel Dicke zeichnet. Dabei soll es egal sein, ob die Linie von links nach rechts oder von rechts nach links läuft. *Hinweis: Schreibe zunächst eine Methode die funktioniert, wenn x2 > x1. Kümmere dich dann erst um das Tauschen der Laufrichtung.*

---

[13] Eine elegantere Methode zum Zeichnen von Linien als hier vorgeschlagen gibt es mit dem Bresenham-Algorithmus, der ist an vielen Stellen im Internet zu finden.

14.3 Wie zeichnet man schräge Linien? (2. Versuch)

Jetzt gibt es noch ein Problem. Die Linien werden nicht mehr durchgängig gezeichnet, sobald der Betrag der Steigung zu groß wird. Am schlimmsten wäre es eine vertikale Linie zu zeichnen, von der erscheint gleich gar nichts. Wenn man mal genau hinsieht, kann man fünf Fälle unterscheiden.

1. $x_2 = x_1$. Da funktioniert unsere Funktion noch nicht! Der Nenner bei der Berechnung von m kann nicht berechnet werden (Teilung durch 0). Außerdem ist das eine vertikale Linie, die wir viel leichter zeichnen könnten. Diesen Fall müssen wir vor der Berechnung von m aussortieren.
2. $|m| \leq 1$, $x_2 > x_1$. Diesen Fall haben wir schon gelöst.
3. $|m| \leq 1$, $x_2 < x_1$. Auch diesen Fall haben wir bereits (bzw. du in der Beantwortung der Aufgabe) schon gelöst durch Vertauschung der Koordinaten.
4. $|m| > 1$, $x_2 > x_1$. Die Linie wird pixelig. Wir müssen unsere Funktion ergänzen.
5. $|m| > 1$, $x_2 < x_1$. Die Linie wird pixelig. Wir müssen unsere Funktion ergänzen.

Am schlimmsten ist eigentlich Fall 1. Wir müssen also zunächst diesen Fall aussortieren. Aus Performancegründen sollte man auch den Fall $y_2 = y_1$ gesondert behandeln, da wir diese Linie leichter zeichnen können als durch das Setzen einzelner divs. Das Vorgehen muss ungefähr so sein:

1. Wenn $x_2 = x_1$ oder $y_2 = y_1$ wende die Methode zum Zeichnen der geraden Linien an. Sonst:
2. Berechne m.
3. Wenn $|m| \leq 1$
   a) Überprüfe auf $x_2 > x_1$.
   b) Tausche wenn nötig die Koordinaten.
   c) Zeichne die Linie.
4. Wenn $|m| > 1$
   a) Überprüfe auf $y_2 > y_1$.
   b) Tausche wenn nötig die Koordinaten.
   c) Zeichne die Linie, allerdings gilt jetzt
      - $b = x_1$
      - die Schleife läuft von 0 bis $y_2 - y_1$
      - $x = i/m + b$
      - $y = i + y_1$

Übung 14.3

1. Schreibe eine entsprechende Funktion zum Zeichnen schräger Linien. Das Formular musst du kaum ändern. Du kannst aber jetzt die Eingabe beliebiger Werte zulassen.

14.4 Kreise

Nachdem wir uns mit den schrägen Linien etwas länger beschäftigt haben, kommt nun eine einfachere Aufgabe, das Zeichnen von Kreisen. Es wird deshalb einfacher, weil wir jetzt immer einzelne Punkte zeichnen und uns nicht so lange mit den verschiedenen Fallunterscheidungen aufhalten müssen. Allerdings dauert es recht lange, bis ein Kreis mit einem Radius von z.B. 400 Pixeln gezeichnet ist (und der Browser reagiert dann nur noch sehr träge).

Das größte Problem ist natürlich, die Anzahl an Punkten zu ermitteln, die zum Zeichnen des Kreises notwendig sind. Nehmen wir zu viele Punkte, dauert das Zeichnen – das sowieso schon lange dauert – noch länger. Wenn zu wenig Punkte gezeichnet werden ist der Kreis unterbrochen. Die Mathematik liefert hier aber schnell die Antwort. Der Umfang des Kreises ist gerade die Größe die wir brauchen!

$$U = 2 * \pi * r$$

Wird der Radius des Kreises in Pixeln angegeben, haben wir bereits die nötige Pixelanzahl gefunden. Jeder einzelne ganzzahlige Wert gibt einen Winkel ω an, unter dem ein Punkt gezeichnet werden muss. Unsere **for**-Schleife muss also von 0 bis $U - 1$ laufen. Jetzt müssen wir „nur“ noch die x- und y-Koordinaten der Punkte ausrechnen. Das geschieht mit den Sinus- und Cosinus-Funktionen:

$$x = \sin(\omega) * r + x_0$$

$$y = \cos(\omega) * r + y_0$$

Schade nur, dass die Sinus- und Cosinus-Funktionen ihren Wertebereich zwischen $0 - 2 * \pi$ erwarten. Unseren Winkel ω, der irgendeinen Wert zwischen 0 und U annehmen wird, müssen wir entsprechend umrechnen (normieren). Die Laufvariable der **for**-Schleife wird dafür zunächst durch die Zahl der Punkte geteilt und anschließend mit $2 * \pi$ multipliziert. Und jetzt noch mal im Zusammenhang:

1. Definiere die notwendigen Variablen in der Funktion.
2. Berechne die Anzahl der notwendigen Punkte.
3. Durchlaufe in einer **for**-Schleife die Schritte:
   a) Normiere die Laufvariable der **for**-Schleife.
   b) Berechne die x-Koordinate.
   c) Berechne die y-Koordinate.
   d) Zeichne den Punkt.

Übung 14.4

1. Erweitere Formular und Programm um die Möglichkeit, Kreise zu zeichnen.
2. *Für Fortgeschrittene:* Gib die Möglichkeit, auch gefüllte Kreise zu zeichnen. *Hinweis: ich würde das mit horizontalen Linien lösen, deren Anfangs- und Endpunkte zu berechnen sind.*

Index

++ (Inkrement-Operator), 13
%, 15

alert, 4
appendChild(), 33
Array: Größe (length), 19
Arrays (Array-Objekt), 12

break, 21

childNode, 30
childNodes, 32
closures, 29
createElement(), 33
createTextNode(), 33

Date-Objekt, 14
do-while-Schleife, 20
document-Objekt, 8
document.create.TextNode(), 33
document.createElement(), 33
document.getElementByID(), 32
document.write, 8
DOM, 30

Events, 6

firstChild, 32
for-Schleife, 13
Formular: input type=button, 10
Formular: input type=text, 15
Funktion deklarieren, 8
Funktion, anonyme, 29
Funktion: Rückgabewert, 21
Funktionen: break, 21

getElementByID(), 32
getTime, 14

hasChildNodes, 32
Hochkomma, 7

if-else, 16
Inkrement-Operator, 13

Klasse (von Objekten), 11
Kommentar (HTML), 3
Konstruktor, 24

location, 23

Math-Objekt, 15
Math.floor, 21
Math.random, 21
Mehrfachauswahl, 22
Modulo-Operator, 15

nodeName, 32
nodeValue, 32

Objekt Definition, 24
Objekt, eigenes, 24
Objekt, Eigenschaften, 24
Objekt, Eigenschaften initialisieren, 27
Objekt, Instanz, 24
Objekt, Konstruktor, 24
Objekt, Methode, 25
Objekt, Vererbung, 28
Objekte, 11

parentNode, 30
parseFloat, 10
parseInt, 10
prompt, 5

removeChild(), 33
return (bei Funktionen), 9

Schleifen (do-while), 20
Schleifen (for), 13
Schleifen (while), 20
Script - Tags, 3
setTimeout, 15
SiblingNode, 30
String als Zahl, 14
Strings, 7
switch, 22

this, 24
toLowerCase, 18
toUpperCase(), 18

Variable: global, 18
Variable: lokal, 18
Variablen deklarieren, 4
Verzögerung, 15

Wahrheitswerte, 16
Weiterleitung, 23
while-Schleife, 20
window.alert, 4
window.document.images, 11, 12
window.location, 23

window.prompt, 5

Zusammenfügen von Strings, 8

Literatur

[1] Clary, Bob: *A JavaScript Wrapper for Making Asynchronous Function and Object Method Calls*, http://devedge.netscape.com/toolbox/examples/2003/CCallWrapper/, 28.09.2003

[2] Netscape: *Core JavaScript Guide 1.5, Chapter 8: Details of the Object Model*, http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide/obj2.html, 28.09.2000

[3] Netscape: *Browser Detection and Cross Browser Support*, http://devedge.netscape.com/viewsource/2002/browser-detection/, 21.09.2003

[4] Mintert, S., Kühnel, C. (Hrsg.): *Workshop JavaScript* München: Addison-Wesley, 2000

[5] Münz, Stefan: *SELFHTML: Version 8.0 vom 27.10.2001 JavaScript / Objekt-Referenz / node*, http://www.selfhtml.teamone.de/javascript/objekte/node.htm, 09.11.2003

[6] petern@paddy.nu: *3D-Cube*, http://www.world-direct.com/mozilla/dhtml/funo/3dcube/index.htm, 23.02.2003

[7] Schrödel, Thomas: *Der Mittelpunktlinien-Algorithmus von Bresenham*, http://www.lehrer-online.de/url/bresenham, 13.08.2003

[8] Team Ideenreich (Hrsg.): *Coole Websites*, Düsseldorf : DATA BECKER, 2000

[9] Vogler, R.: *Java Script für Kids*, Bonn : MITP, 2001

[10] Unbekannt: *JavaScript Bedingungen*, http://www.jakober.ch/js/js_bedingungen.php, 26.01.2003