JAVASCRIPT

[ruediger.meyer@online.de]

Inhaltsverzeichnis

1	**Einleitung und Vorbereitung**	**1**
1.1	html und CSS	
1.2	Editoren	
2	**Das erste Programm**	**2**
2.1	Einbinden von Javascripts	
2.2	Einbinden von Scripts von externen Dateien	
2.3	Standardausgabe **`document.write()`**	
2.4	Kommentare	
2.5	objektorientierte Programmierung	
3	**Variablen**	**4**
3.1	Numerische Variablen	
3.2	String-Variablen (Zeichenketten)	
3.3	Boolean (wahr-falsch)	
4	**Operatoren**	**5**
4.1	Arithmetische Operatoren	
4.2	Logische Operatoren	
5	**Verzweigungen**	**6**
5.1	**`if/else`**	
5.2	**`switch/case`**	
6	**Schleifen**	**7**
6.1	**`for`**-Schleife	
6.2	**`while`**-Schleife	
6.3	**`do-while`**-Schleife	
7	**Arrays**	**8**
7.1	Deklaration und Initialisierung	
7.2	mehrdimensionale Arrays	
7.3	Attribute und Methoden	
7.3.1	**`length`**	
7.3.2	**`concat()`**	
7.3.3	**`join()`**	
7.3.4	**`pop()`**	
7.3.5	**`push()`**	
7.3.6	**`reverse()`**	
7.3.7	**`sort()`**	
8	**Funktionen**	**11**
8.1	Objekt unabhängige Funktionen	
8.1.1	**`eval()`**	
8.1.2	**`IsFinite()`**	
8.1.3	**`parseInt()`**	
8.1.4	**`Number`**	
8.1.5	**`String()`**	
8.1.6	**`IsNaN()`**	
9	**Wichtige Objekte**	**12**
9.1	**`String`**	
9.1.1	**`length`**	
9.1.2	**`charAt(n)`**	
9.1.3	**`concat()`**	
9.1.4	**`split()`**	
9.1.5	**`substring(von,bis)`**	
9.1.6	**`toLowerCase / toUpperCase`**	
9.2	**`Math`**	
9.2.1	Zufallszahlen	
9.2.2	Runden von Zahlen - **`round()`**, **`ceil()`** und **`floor()`**	
9.2.3	diverse mathematische Funktionen und Konstanten	
9.3	**`Date`**	
9.3.1	Initialisierung	
9.3.2	Anwendungsbeispiel	
9.3.3	Methoden des **`Date`**-Objekts	
9.3.4	Rechnen mit dem **`Date`**-Objekt	
9.3.5	Timeouts und Intervalle	

1 Einleitung und Vorbereitung

JavaScript ist wie der Name schon sagt eine Script-Sprache. Im Gegensatz zu vollwertigen Programmiersprachen liefert sie keine eigenständig lauffähigen Programme, die durch einen Compiler in Maschinencode bzw. Runtime-kompatible Konstrukte übersetzt werden, sondern Programmfragmente, die vom Interpreter des jeweiligen Browsers direkt verwendet werden. JavaScript ist somit eine clientseitige Sprache, d.h. dass die Konfiguration des lokalen Rechners über die Funktion entscheidet. Dies hat primär den Vorteil, dass keine unnötigen Daten zwischen Server und Client transportiert werden müssen und somit die Geschwindigkeit des „Programmes“ auch nicht von der Leistungsfähigkeit der Netzwerkverbindung abhängt. Allerdings hängt das Aussehen und die Funktion stark vom gewählten Betriebssystem (der gewählten virtuellen Maschine, in der das Skript letztendlich läuft) und vor allem vom gewählten Browser ab.
Es gibt zwar gewisse Vereinbarungen über den Aufbau und die einzelnen Elemente eines Javascripts, allerdings werden diese von den einzelnen Browsern verschieden streng beachtet und die führt immer wieder zu überraschenden Ausgaben.
Daher sollte man stets die Browserkompatibilität des Scripts testen und bei Bedarf auch Anpassungen an die einzelnen (eigentlich unzähligen) verschiedenen Browser durchführen.
Alle notwendigen Hilfsmittel (Runtime, Interpreter,....) für die Programmierung werden heute von Betriebssystem und Browser bereitgestellt. Bei Problemen hilft wie so oft das Internet.

1.1 html und CSS

Da JavaScript im Browser im Browser ausgeführt wird sind natürlich gewisse Ausdrücke in der „Sprache der Browser“ erforderlich, sprich ohne html und CSS ist es sehr schwierig, in den Programmen Ausrücke zu erzeugen, die dann auch ordentlich dargestellt werden. Allerdings gibt es im Internet zahlreiche Tutorials und Referenzbibliotheken, die einem die Arbeit deutlich erleichtern(selfhtml,...).
Die Verwendung von CSS-Klassen und -Deklarationen in html, sowie Tabellen, Formulare und die Grundstrukturen in html werden hier vorausgesetzt.

1.2 Editoren

Grundsätzlich kann man mit jedem Texteditor programmieren, allerdings bringt das einige Nachteile mit sich: 1. hat man keine Vorschaumöglichkeit (nur umständlich abspeichern und dann im Browser öffnen) und 2. kein Highlighting (erkannte Befehle und Strukturen werden farblich hervorgehoben). Daher empfiehlt es sich, spezielle Editoren zu verwenden. Diese gibt es wie Sand am Meer (Webcraft, Bluefish,....).
Natürlich gibt es auch Programme, die automatische Codes erzeugen. Man sollte allerdings beachten, dass diese Codes nicht immer sehr effizient sind und teilweise zu Problemen führen können. Dabei gestaltet sich dann die Fehlersuche äußerst schwierig, weshalb ich hier die Verwendung von den oben erwähnten Editoren sowie gute alte Handarbeit vor allem für komplexere Projekte empfehlen würde.
Die hier erwähnten Beispiele werden mit Bluefish und Firefox getestet.

2 Das erste Programm

Wie in jeder Programmiersprache werden wir auch hier so schnell wie möglich versuchen, dem System eine sinnvolle Ausgabe zu entlocken. Und da wir mit der Tradition nicht brechen wollen, folgt hier unser **„Hello World!“**

```
<html>
  <head>
  </head>
  <body>
    <script type="text/javascript" >
      document.write("Hello World!");
    </script>
   </body>
</html>
```

Streng genommen versteckt sich in diesem Script schon sehr viel Theorie, auf die wir dann teilweise später noch genauer zu sprechen kommen.

2.1 Einbinden von Javascripts

Da es mehrere Arten von Scripts gibt, muss man bei der Eröffnung eines Script-Tags auch gleich die entsprechende Sprache mitliefern. Hierbei gibt es zwei Möglichkeiten:

```
…
<script type="text/javascript">
…
</script>
// oder
<script language="javascript">
…
</script>
```

Je nach Dokumentation wir die eine oder die andere Art bevorzugt. Grundsätzlich funktionieren beide und können so auch verwendet werden.

2.2 Einbinden von Scripts von externen Dateien

Man kann JavaScripts auch als externe Datei einbinden. Dazu muss das Script in einer Datei mit der Endung **`*.js`** gespeichert werden.

```
<script language="JavaScript" src="baden.js">
```

Man sollte allerdings bedenken, dass **`src`** verhindert, dass weitere Befehle innerhalb des **`<script>`**-Tags ausgeführt werden. Dies kann man ändern, wenn man für die Befehle einen eigenen **`<script>`**-Tag öffnet:

```
<script language="JavaScript" src="baden.js"></script>
<script language="JavaScript"> …</script>
```

2.3 Standardausgabe document.write()

Mit **document.write("Hello World!")** wir der Ausdruck zwischen den Anführungszeichen an **html** übergeben. Somit ist es auch möglich, **html**-Tags in diese Ausdrücke einzubauen und die Formatierungsoptionen von **html** und **CSS** zu nutzen. Bzgl. Anführungszeichen sollte man beachten, dass einfache und doppelte Anführungszeichen völlig gleichwertig verwenden werden können. Somit kann man auch **html**-Attribute mit ihren Werten übergeben. Allerdings muss man dabei auf korrekte Strukturen achten.

```
document.write("<h1 align='center'>Hello World!</h1>");
document.write("Zitat:\"....\"");
```

Man kann die Ausdrücke mit Werten von Variablen verketten. Sollte man ein Zeichen, das Teil der Syntax ist (wie hier z.B. die Anführungszeichen), ausgeben wollen, muss man dieses mit **** maskieren und somit „entwerten".

2.4 Kommentare

Bei umfangreicheren Programmen ist es sinnvoll und oft auch erforderlich, dass man die einzelnen Programmteile mit Kommentaren versieht. Vor allem die Wartung und die Nachvollziehbarkeit für andere wird dadurch deutlich gesteigert.

```
// einzeiliger Kommentar
/* mehrzeiliger
Kommentar */
```

2.5 objektorientierte Programmierung

In Javascript sind einige Elemente der OOP eingebaut worden, zB. haben wir in **"Hello World"** mit **document.write("Hello World");** die Methode **write** der Klasse **document** aufgerufen.
Das Erstellen von eigenen Klassen mit Attributen und Methoden ist allerdings hier relativ umständlich(vor allem im Vergleich zu Java, C#,....). Allerdings haben wir es ja auch nicht mit einer vollwertigen Programmiersprache zu tun.
Wir werden noch einige implementierte Klassen verwenden. Einige Unterschiede zur vollwertigen Programmiersprache werden sich recht bald zeigen.

3 Variablen

Den Begriff „Variable" kennt man aus der Mathematik vor allem als „Platzhalter". So ähnlich kann man das auch hier interpretieren: bei der Deklaration einer Variablen wird Speicherplatz reserviert, auf dem dann ein Wert gespeichert werden kann.
Anders als bei vollwertigen Programmiersprachen muss der Datentyp bei der Deklaration nicht angegeben werden. Streng genommen wird dieser auch erst bei der Initialisierung (Belegen der Variable mit einem Wert) festgelegt. Zusätzlich ist dieser Variablentyp dann auch nicht in Stein gemeißelt, sondern jede Variable kann mit irgendeinem Wert überschrieben werden und dabei wird der Variablentyp neu festgelegt.
Die Deklaration und Initialisierung erfolgt mit **var**.

```
var x=15;
var y="Hallo!"
```

3.1 Numerische Variablen

Javascript unterscheidet nicht zwischen Dezimalzahlen und ganzzahligen Werten. Es wird immer eine 64-Bit-Gleitkommazahl angelegt. Man sollte beachten, dass als „Komma“ der aus dem englischen stammende Dezimal-`"."` dient.

```
var a=1999;
var b=1.92923923923;
var z;
var hex= 0xff; // entspricht 255
```

Man kann ganze Zahlen auch hexadezimal schreiben, beginnend mit **`0x`**.
Sollte eine Operation einen Fehler oder ein sinnloses Ergebnis zur Folge haben (z.B. Wurzel aus einer negativen Zahl), so liefert sie den Spezialwert **`NaN`** (not a number).
Mit der Funktion **`isNaN()`** kann man auf diesen Wert prüfen.

3.2 String-Variablen (Zeichenketten)

-

3.3 Boolean (Wahr-falsch)

-

4 Operatoren

4.1 Arithmetische Operatoren

Verschiedene Rechenoperatoren zur Verwendung mit Zahlenvariablen:

Operator	Beschreibung	Beispiel
`+`	Addition	`a = 7 + 4`
`-`	Subtraktion	`a = 7 - 4`
`*`	Multiplikation	`a = 7 * 4`
`/`	Division	`a = 7 / 4`
`%`	Modulo ("Restrechnung")	`a = 7 % 4`
`-`	Negation	`a = -b`
`++`	um `1` erhöhen("Inkrement")	
`--`	um `1` vermindern ("Dekrement")	

Wenn man den Wert einer Variable nicht um `1` sondern um einen beliebigen Wert erhöhen möchte, verwendet man folgende Operatoren:

Operator	Beschreibung	Langform	Kurzform
`+=`	Addition	`a = a + b`	`a += b`
`-=`	Subtraktion	`a = a - b`	`a -= b`
`*=`	Multiplikation	`a = a * b`	`a *= b`
`/=`	Division	`a = a / b`	`a /= b`
`%=`	Modulo	`a = a % b`	`a %= b`

4.2 Logische Operatoren

Diese **Logikoperatoren** dienen zum Verknüpfen von Wahrheitswerten.

Operator	Beschreibung
`&&`	Logisches `"AND"; "true"` nur wenn beide `"true"`
`||`	Logisches `"OR"; "true"` wenn eine von zwei Variablen oder beide `"true", "false"` nur wenn beide `"false"`
`!`	Logisches `"NOT";` macht aus `"true" "false"` und umgekehrt

Vergleichsoperatoren:

Operator	Beschreibung	Beispiel	Ergebnis (für `a`)
`==`	gleich	`a = (3 == 4)`	`false`
`!=`	ungleich	`a = (5 != 4)`	`true`
`>`	grösser als	`a = (3 > 4)`	`false`
`<`	kleiner als	`a = (3 < 4)`	`true`
`>=`	grösser oder gleich	`a = ( 3 >= 4)`	`false`
`<=`	kleiner oder gleich	`a = (3 <= 4 )`	`true`

5 Verzweigungen

Oft muss man gewisse Fälle ausschließen, d.h. überprüfen, ob Bedingungen erfüllt sind.

5.1 if/else

Wenn eine Bedingung erfüllt ist, wird der entsprechende Anweisungsteil ausgeführt, ansonsten der Anweisungsteil des **`else`**-Zweiges (falls vorhanden)

```
if (Bedingung)        // wenn
{
  Anweisungsteil      // dann
}
else
{
  Anweisungsteil      // ansonsten
}
```

Falls man weitere Verzweigungen benötigt, kann man **`else if`** verwenden.

5.2 switch/case

„**`switch/case`**“ benötigt man für Mehrfachverzweigungen. **`switch`** berechnet einen Ausdruck und durchläuft dann die Liste von **`case`**-Klauseln bis ein Wert gefunden wird, der zum Wert des Ausdrucks passt.
Grundlegende Syntax:

```
switch (Ausdruck)
{
  case wert1: Anweisungen;
              break;
  case wert2: Anweisungen;
              break;
...
}
```

Man muss jede **`case`**-Klausel mit **`break`** oder **`return`** abschließen, da er sonst in alle weiteren **`case`**-Klauseln ab dem passenden Wert springt.

Beispiel: Ausgabe der Monatsnamen nach Eingabe einer Monatszahl

```
switch (monatszahl)
{
  case 1: monat = "Januar";
          break;

  case 2: monat = "Februar";
          break;

  case 3: monat = "März";
          break;            // Vorsicht bei Umlauten!
…
}
```

6 Schleifen

Schleifen dienen in der Programmierung zum mehrmaligen Ausführen von Anweisungen. Dadurch kann man sich als Programmierer einiges an Programmcode ersparen. Will man z.B. ein Array mit Zufallszahlen füllen, so braucht man nicht für jede Stelle des Arrays eine Anweisung, sondern versucht das Ganze in einer Schleife in ein paar Zeilen zu erledigen und dabei ist es dann egal, ob ich `10` Stellen zuweisen möchte oder `100000` oder mehr.

6.1 for-Schleife

Diese Schleife wird primär für die Vorgabe einer bestimmten Anzahl an Schleifendurchläufen verwendet. Allerdings ist sie eigentlich die mächtigste unter den Schleifen, da man mehrere Parameter auch auf relativ komplexe Bedingungen überprüfen kann.
Grundlegende Sytax:

```
for (Startbedingung;Überprüfung;Schrittweite)
{
  Anweisungsteil ….
}
```

Konkretes Beispiel: Wir wollen die Zahlen von `1` bis `10` ausgeben:

```
for (var i=1;i <= 10;i++)
{
  document.write(i);
}
```

Die Laufvariable (hier `i`) muss deklariert werden. Dies kann allerdings wie hier auch direkt in der Schleife erfolgen, die Variable steht dann allerdings auch nur für diese Schleife zur Verfügung.
Je nach Bedingung kann der Anweisungsteil auch nicht ausgeführt werden, was man allerdings als eher sinnlos betrachten müsste.

6.2 while-Schleife

Bei dieser Schleife wird eine Bedingung überprüft und dann der Anweisungsteil ausgeführt, allerdings gibt es keine Laufvariable, womit die Anzahl der Durchläufe rein von der Bedingung abhängt.
Grundlegende Systax:

```
while (Bedingung)
{
  Anweisungsteil ….
}
```

Konkretes Beispiel: Wir wollen zwei Zahlen auf Gleichheit überprüfen und die zweite Zahl dann solange um **1** erhöhen, bis sie gleich sind:

```
while (a!=b)
{
  document.write(“nicht gleich“);
  b++;
}
```

Das Beispiel scheint im ersten Moment wahrscheinlich nicht als sehr sinnvoll, allerdings erkennt man sehr gut die Verwandtschaft zur **for**-Schleife. Man kann meistens jede Schleife durch die andere ersetzen, allerdings zumeist auf Kosten des Komforts.

6.3 do-while-Schleife

Hier handelt es sich um eine while-Schleife, bei der die Bedingungen erst nach dem ersten Durchlauf des Anweisungsteils überprüft wird. Somit kann ich einen Durchlauf sicherstellen.
Grundlegende Systax:

```
do
{
  Anweisungsteil ….
} while(Bedingung)
```

Wenn ich nun den Benutzer z.B. eine Zahl erraten lasse, muss ich zumindest eine Chance auf eine Eingabe geben und dann erst diese überprüfen.
Dies würde dann so aussehen:

```
do
{
  var a = 100;                      // zu erratende Zahl
  var b = prompt("Zahl eingeben"); // Benutzereingabe

  if (a == b)
  {
    document.write("SIEGER");
  }
} while (a != b)
```

7 Arrays

Arrays sind Datencontainer, in denen eine beliebige Anzahl an Daten gesammelt werden können. Im Gegensatz zu vielen Programmiersprachen werden in JavaScript Arrays dynamisch angelegt, d.h. die Anzahl der Elemente muss bei der Definition nicht fixiert werden. Es können also beliebig Elemente hinzugefügt werden. Außerdem kann man auf die einzelnen Elemente über ihren Index (beginnend bei **0**) zugreifen.

7.1 Deklaration und Initialisierung

Neue Arrays können mit dem **Array()**-Konstruktor erzeugt werden:

```
var a = new Array();      // leeres Array
var b = new Array(10);    // 10 Elemente
var c = new Array(1,2,3); // Elemente 1,2,3
```

Eine weitere Möglichkeit ist die **Arrayliteral**-Schreibweise, wobei die Werte als Liste durch Komma getrennt in eckigen Klammern angegeben werden:

```
var a = [1,2,3];
```

Man kann dann auf jedes **Array**-Element über den Index zugreifen.

```
Document.write(a[2]);     // liefert 3
```

7.2 mehrdimensionale Arrays

Es können auch mehrdimensionale Arrays erzeugt werden. Mehrdimensionale Arrays sind immer dann von Vorteil, wenn man mehrere Sätze zusammengehöriger Daten in einem Array speichern will (z.B. Personendatenbank).

```
var person = new Array(); // Array für Personen

person[0]  = new Array(); // Jede Person als Array von Informationen
person[0]["Vorname"]  = "Rüdiger";
person[0]["Nachname"] = "Meyer";
person[0]["E-Mail"]   = "firma@fwm.com";
person[1] = new Array();
person[1]["Vorname"]  = "Karl";
person[1]["Nachname"] = "Fisch";
person[1]["E-Mail"]   = "kutter@nordsee.com";
```

7.3 Attribute und Methoden

Einige Eigenschaften und Prozeduren stehen für **Array**-Objekte zur Verfügung.

7.3.1 length

Mit diesem Attribut kann man die Länge eines Arrays abfragen, also wie viele Elemente in diesem Array gespeichert sind. Allerdings sollte man beachten, dass z.B. bei Länge **13** das letzte Element den Index **12** hat. Dieses Attribut wird oft in einer **for**-Schleife für die Ausgabe aller Array-Elemente verwendet.

```
var speicher = new Array(1,2,3);

for (var i=0;i < speicher.length;i++)
{
  document.write(speicher[i]
};
```

7.3.2 concat()

Mit dieser Methode kann man an ein **Array** Elemente anhängen

```
var neuspeicher = speicher.concat(4,5,6); // 4,5,6 wird an speicher
                                          // angehängt, womit das
                                          // neue Array jetzt
                                          // Länge 6 besitzt
```

Wenn die Argumente von **concat** selbst Arrays sind, werden deren Elemente angehängt, nicht die Arrays selbst.

7.3.3 join()

Mit **join (Trennzeichen)** wird ein String zurückgegeben, der aus allen Arrayelementen mit dem angegebenen Trennzeichen gebildet wird.

7.3.4 pop()

Entfernt das letzte Element und gibt es zurück.

7.3.5 push()

Hängt die als Parameter angegebenen Werte an das bestehende Array an und liefert die Länge des Arrays zurück.

7.3.6 reverse()

Kehrt die Reihenfolge der Arrayelemente um.

7.3.7 sort()

Mit **sort** wird ein Array geordnet. Wenn die **Array**-Elemente Strings sind, dann erfolgt die Sortierung lexikalisch aufsteigend. Will man Zahlenarrays sortieren, benötigt man eine Vergleichsfunktion.
Man kann z.B. eine Funktion deklarieren, deren Rückgabewert regelt, wie zwei Elemente sortiert werden. Beim Sortieren wird die Funktion mit jeweils zwei Elementen **a**, **b** aufgerufen, deren Differenz dann als Grundlage für die Sortierung dient: ist sie **>0** dann hat die erste Zahl **a** einen höheren Index, **<0** wird **b** mit höherem Index versehen und wenn die Differenz **0** ist, so sind die Zahlen gleich und müssen nicht umsortiert werden.

```
<script type="text/javascript">
  function Numsort(a, b) // Hilfsfunktion zum Sortieren
  {
    return(a - b);
  }
  var UnterrichtsFach = new Array("Mathe", "Informatik",
                                  "Deutsch", "Sport");

  UnterrichtsFach.sort();

  var anzahl = new Array(1, 7, 9, 7,5);

  Zahlen.sort(Numsort);

  var Stringausgabe = Namen.join(",");
  var Zahlenausgabe = Zahlen.join(",");

  document.write("sortierte Unterrichtsfächer: " +
  Stringausgabe + "<br>");
  document.write("sortierte Anzahl: " + Zahlenausgabe);
</script>
```

8 Funktionen

Als Funktion in JavaScript bezeichnet man ein Stück Programmcode, das nur einmal definiert aber von einem Programm mehrfach ausgeführt werden kann. Man unterscheidet Funktionen mit und ohne Parameter.

```
function summe(x, y) // Funktion mit Parametern x, y
{
  return(x + y);
}
```

`return` liefert einen Wert zurück. Den kann ich dann beliebig weiterverwenden(z.B. einer Variable zuweisen, ausgeben,...).
Der Aufruf erfolgt dann einfach über den Funktionsnamen mit Übergabe der entsprechenden Parameter. JavaScript ist es eigentlich völlig egal, ob zu viele oder zu wenige Parameter übergeben werden, auch ob der richtige Datentyp übergeben wird ist völlig irrelevant. Werden zu wenige Parameter übergeben, so bleiben die letzten Parameter **`undefined`**, werden zu viele Parameter übergeben, so werden die überzähligen Parameter ignoriert.

```
var z= summe(3,5); // z wird mit dem Wert 8 initialisiert
```

8.1 Objekt unabhängige Funktionen

8.1.1 eval()

eval() interpretiert ein zu übergebendes Argument und gibt das Ergebnis zurück.
Wenn das übergebene Argument als Rechenoperation interpretierbar ist, wird die Operation berechnet und das Ergebnis zurückgegeben. Dabei sind auch komplexe Rechenausdrücke mit Klammerung möglich.

```
var kugel1 = 99;
var kugel2 = 299;
var kugel3 = 2387;
…
for (var i=1;i <= 3;i++) // Ausgabe
{
  document.write(eval("kugel" + i) + "<br />");
}
```

So kann man gewährleisten, dass alle Variablenwerte **kugel1**, **kugel2**, **...** ausgegeben werden, ohne jede einzelne Variable separat aufzurufen.

8.1.2 IsFinite()

Mit dieser Funktion kann man verhindern, dass man aus dem vorgegebenen Wertebereich hinaus schießt.

```
var Zahl = Number.MAX_VALUE;

if (!isFinite(Zahl * 2))
  alert("Die Zahl ist nicht zu verarbeiten.");
```

8.1.3 parseInt()

Wandelt eine zu übergebende Zeichenkette in eine Ganzzahl um und gibt diese als Ergebnis zurück. Sinnvoll, um z.B. Anwendereingaben in Zahlen umzuwandeln, mit denen man anschließend rechnen kann.
Gibt **NaN** (Not a Number) zurück, wenn die Zeichenkette mit Zeichen beginnt, die sich nicht als Teil einer Zahl interpretieren lassen.

```
for (var i=0;i < Elemente.length;i++)
  document.write(Elemente[i]           +
                 " = <b> "             +
                 parseInt(Elemente[i]) +
                 "<\/b><br>");
```

8.1.4 Number

Wandelt den Inhalt eines Objektes in eine Zahl um und gibt diese zurück.

8.1.5 String()

Wandelt den Inhalt eines Objekts in eine Zeichenkette um und gibt diese zurück.

8.1.6 IsNaN()

Ermittelt, ob ein Wert keine gültige Zahl ist.

9 Wichtige Objekte

9.1 String

Auch Zeichenketten werden als Objekte der String-Klasse erzeugt, daher kann man z.B. bei der Erzeugung einer Stringvariable auch den Konstruktor für String-Objekte verwenden.

```
var a= new String("Hallo");
```

Auch für diese Objekte stehen wieder einige Attribute und Methoden zur Verfügung.

9.1.1 length

Die Anzahl der Zeichen im String wird zurückgegeben.

```
Document.write(a.length); // liefert 5
```

9.1.2 charAt(n)

Gibt das Zeichen an der Stelle n zurück.

```
a.charAt(2) // liefert "l" (beginnend bei 0)
```

9.1.3 concat()

Verkettet die übergebenen Strings mit dem bestehenden zu einem neuen String und liefert diesen zurück.

```
var eins = "Heute";
var ziel = eins.concat("ist", "ein", "schöner", "Tag");
// liefert "Heute ist ein schöner Tag";
```

9.1.4 split()

Teilt einen String in Teilstrings auf und legt diese in ein Array ab.

```
var test        = "Heute ist ein schöner Tag";
var testInArray = test.split(" ");
// Somit wird jedes Wort als einzelnes Array-Element abgelegt
```

9.1.5 substring(von,bis)

Liefert einen Teilstring von Zeichen an der Stelle **"von"** bis zum Zeichen an der Stelle **"bis - 1"**.

9.1.6 toLowerCase / toUpperCase

Liefert jeweils eine Kopie der Zeichenkette in jeweils nur Groß- bzw. Kleinbuchstaben.
Es gibt noch zahlreiche Methoden, die wahrscheinlich für unsere Programme nicht relevant sind und die man bei entsprechender Recherche sicher leicht im Internet findet.

9.2 Math

Die Klasse **Math** stellt einige wichtige Methoden für mathematische Berechnungen zur Verfügung, darunter auch wichtige Funktionen, wie **Sinus**, **Cosinus**,...

9.2.1 Zufallszahlen

Zum Erzeugen von Zufallszahlen verwendet man die Methode **Math.random()**. Hier wird eine zufällige Dezimalzahl zwischen **0** und **1** erzeugt. Durch Multiplikation mit einem entsprechenden Faktor erzeugt man dann Dezimalzahlen in einem beliebigen Intervall.

```
var a = Math.random() * 1000; // erzeugt eine zufällige Dezimalzahl
                              // zwischen 0 und 1000
```

Für ganzzahlige Zufallszahlen muss man die Dezimalzahlen dann runden.

9.2.2 Runden von Zahlen- round(), ceil() und floor()

Das Runden nach den bekannten Regeln erledigt **round()**, **ceil()** rundet auf jeden Fall auf, **floor()** auf jeden Fall ab.

```
var a = Math.ceil(Math.random()*49); // Lottozahlen zwischen 1 und 49
```

9.2.3 diverse mathematische Funktionen und Konstanten

```
document.write(Math.E);             // liefert die eulersche Zahl
document.write(Math.LN2);           // ln 2
document.write(Math.LOG2E);         // Logarithmus der Eulerschen
                                    // Konstante zur Basis 2
document.write(Math.PI);            // liefert PI
document.write(Math.SQRT2);         // liefert die Quadratwurzel aus 2
document.write(Math.sin(Math.PI)); //Sinus von PI
// cos(), tan(),...
```

9.3 Date

Das Objekt **Date** ist für alle Berechnungen mit Datum und Zeit verantwortlich.

9.3.1 Initialisierung

```
var jetzt = new Date(); // Timestamp jetzt
```

Hier wird das aktuelle Datum mit der aktuellen Zeit (Quelle ist der lokale Rechner!) zum Zeitpunkt der Initialisierung der Variablen gespeichert.
Es gibt nun auch die Möglichkeiten, ein **Date**-Objekt mit einem bestimmten Datum oder auch mit Datum und Zeit zu belegen:

```
var datum     = new Date(Jahr, Monat, Tag);
var datumzeit = new Date(Jahr, Monat, Tag, Stunde, Min, Sek);
```

9.3.2 Anwendungsbeispiel

Im folgenden Beispiel werden die Sekunden bis zum Jahr **2050** ausgegeben:

```
function sekBis2050()
{
  var jetzt   = new Date();
  var Zeit    = jetzt.getTime() / 1000;        // aktuelle Zeit
  var ziel    = new Date(2050, 0, 1, 0, 0, 0); // Jahr 2050
  var Endzeit = ziel.getTime() / 1000;
  var Rest    = Math.floor(Endzeit - Zeit);    // Differenz

  alert("Es sind noch " + Rest + " Sekunden bis zum Jahr 2050");
}
```

9.3.3 Methoden des **Date**-Objekts

Methode	Beschreibung
getDate()	liefert den Monatstag, z.B. „**1**“ für 1. Januar
getDay()	liefert Wochentag, **0** ... Sonntag – **6** ...Samstag
getFullYear() / getYear()	liefert volles Jahr bzw. die letzten beiden Ziffern, z.B. „**1998**“ bzw. „**98**“ (analog **getMonth()**)
getHours()	Stunden z.B. **23:59** liefert „**23**“ (analog **getMinutes()** und **getSeconds()**)
getTime()	rechnet das Datum in Millisekunden (seit **1. Januar 1970, 0:00:00 Uhr UTC**) um
set...()	analog zum Auslesen mit **get....()** kann man die entsprechenden Parameter mit **set....()** setzen

9.3.4 Rechnen mit dem **Date**-Objekt

```
<script type="text/javascript">
  var jetzt    = new Date();
  var ziel     = new Date(2010, 6, 2, 10, 0, 0);     // Schulschluss
  var diff     = ziel - jetzt;                       // ms
  var sekunden = Math.floor(diff / 1000);
  var tage     = Math.floor(diff / 1000 / 60 / 60 / 24);

  diff = diff - (tage * 1000 * 60 * 60 * 24);

  var stunden = Math.floor(diff / 1000 / 60 / 60);

  diff = diff-(stunden * 1000 * 60 * 60);

  var minuten = Math.floor(diff / 1000 / 60);

  document.write('Noch ' + tage + ' Tage, ' +
                 stunden + ' Stunden und ' +
                 minuten + ' Minuten bis Schulschluss!');
</script>
```

Wie man an diesem Beispiel sieht, bereitet zumeist nicht das **Date**-Objekt selbst Probleme, sondern das Umrechnen von Sekunden in Stunden, Jahre,....

9.3.5 Timeouts und Intervalle

In JavaScript ist es möglich verschiedene Methoden zeitversetzt oder mehrfach auszulösen. Man muss dazu nur ein **Timeout** bzw. **Intervall** setzen.

Beispiel für Intervall:

```
<script type="text/javascript">
  var aktiv = window.setInterval("Farbe()", 1000);
  var i     = 0,
      farbe = 1;

  function Farbe()
  {
    if (farbe == 1)
    {
      document.bgColor = "yellow";
      farbe = 2;
    }
    else
    {
      document.bgColor = "aqua";
      farbe = 1;
    }
    i = i + 1;
    if (i >= 10)
      window.clearInterval(aktiv);
  }
</script>
```

Beispiel für Timeout:

```
<script type="text/javascript">
  function Hinweis()
  {
    var x = confirm("Sie sind jetzt schon 10 Sekunden auf
                     dieser Seite. Fortfahren?");

    if (x == false)
      top.close();
  }
  window.setTimeout("Hinweis()", 10000);
</script>
```