Programmieren lernen mit JavaScript

(Version 25.9.2021)

Inhalt:

Hilfen:

Self-HTML-Ersatz: http://wiki.selfhtml.org/wiki/Startseite

V1: Interaktive Videotutorial Serie:

Die Videoserie war dafür gemacht, JavaScript wirklich zu lernen - sie stellt die Sprache nicht bloß vor, sondern fordert zum Mitmachen auf und erklärt Programmieren durchweg mit Hilfe von Beispielen. Programme müssen keine installiert werden, da als Entwicklungsumgebung die Webkonsole des Browsers genutzt wird.

V2: Einführung:

Anwendung: Webseiten, Windows 8 Apps, Smartphone Apps
HTML5 und CSS3

Wichtig: **JAVA ≠ JAVASCRIPT**!!

Konsole des Browsers erlaubt Eingaben und Übungen mit JavaScript

Öffnen der Webkonsole: Firefox -> Extras > Web-Entwickler -> Browser-Konsole (ctrl + shift + K)

V3a: Strings:

-> Strings sind Zeichenketten die mit doppelten oder einfachen Anführungszeichen eingerahmt sind.

-> **Hallo Welt** gibt eine Fehlermeldung, da für JavaScript ein nicht verständlicher Code

-> **„Hallo Welt“** als String gibt keine Fehlermeldung, da JavaScript innerhalb Strings nicht nach Code sucht!

-> Wichtig zu wissen: Innerhalb Strings prüft der JavaScript Code nichts

-> Um Leseprobleme mit Strings zu eliminieren kann ein Backslash vor nicht zu lesende Anführungszeichen gesetzt werden:
   -> 'hier gibt's ein Problem‘ führt zu einer Syntaxfehlermeldung
   -> ‘hier gibt\'s ein Problem' wird richtig gelesen, da der Backslash das Apostroph im String versteckt bzw. maskiert (sog. Maskierungs- Zeichen oder Escape-Character).

V3b: Zusammenfügen von Strings

-> Strings können zusammengesetzt sein aus Textstücken, Zahlen und Variablenresultate.

Für das Zusammenfügen gelten folgende Regeln:

- Textstücke werden immer innerhalb Anführungszeichen geschrieben (z.B. “Text Text Text“)
- Um Leseprobleme mit Strings zu eliminieren kann ein Backslash vor nicht zu lesende Anführungszeichen gesetzt werden:

-> 'hier gibt's ein Problem‘ führt zu einer Syntaxfehlermeldung wegen Apostroph
-> ‘hier gibt\'s ein Problem' wird richtig gelesen, da der Backslash das Apostroph im String versteckt bzw. maskiert (sog. Maskierungs- Zeichen oder Escape-Character).

- Zahlen (z.B. 123) werden ohne Anführungszeichen geschrieben
  (Grund: weil JavaScript keinen Code in Form von Zahlen hat)

- Variablenausgaben (z.B. VarName) benötigen keine Anführungszeichen
  (Grund: da Variablendeklarationen immer in Anführungszeichen sind)

- String-Stücke werden mittels + zusammengehängt (“AAA“ + Zahl + “BBB“)

- Zusammenfügen von Textstrings durch Elimination der mittleren Anführungszeichen
  (z.B. “AAAA“ “BBBB“ -> “AAAABBBB“)

- Zusammenfügen von Textstrings durch Einfügen eines Pluszeichens zwischen den Teilstrings
  (z.B. “AAAA“ + “BBBB“ -> “AAAABBBB“)

- Zusammenfügen von Textstrings mit Variablenausgaben erfolgt durch Einfügen eines Pluszeichens vor der Variablenausgabe.
  (z.B. “AAAA“ + VarName + “BBBB“ -> “AAAAVarWertBBBB“)
  Bem. Befindet sich die Variablenausgabe am Schluss des Strings darf kein + angefügt werden.

- Beim Zusammenfügen von Textstrings mit Variablenausgaben die addiert werden, muss bei Summenbildung mit Plus jede Variablenausgaben vorher in „parseInt“ umgewandelt werden. Ansonsten werden die Variablenausgaben wie zwei hintereinandergeschaltete Zahlenstrings (d.h. nicht addiert!) angezeigt.

V4: Rechnen:

-> Zahlen brauchen keine Stringzeichen, Abstand zwischen Zahl und weiteren Zeichen ist frei

-> Dezimalkommas werden als Punkte geschrieben

-> Zum Rechnen werden die normalen mathematischen Symbole (**+ - / ***) verwendet

-> Es ist völlig unwichtig ob zwischen Zahl und mathematischem Symbol eine Lücke ist oder nicht

-> verschachtelte Ausdrücke werden automatisch in richtiger Reihenfolge berechnet:
-> 10 + 4 / 2 wird als 10 + (4/2) gelesen und behandelt. Sonst muss (10 + 4)/2 geschrieben werden

-> höchstmögliche Zahl ist 1.79769..e+3308 / niedrigste Zahl ist 5e-324 (e bedeutet Exponentialzahl von 10 und die kann positiv oder negativ sein). Es gibt auch +infinity und –infinity

V5: Boole‘sche Werte und Vergleichsoperatoren:

-> 1 ist Ja bzw. true / 0 ist Nein bzw. false

-> **Datentypen**:

Boolean	true oder false bzw. 1 oder 0
Number	42
Strings	„Text“
Function	function() {}
Object	{}
Undefined	Variable ohne definierten Wert
Null	

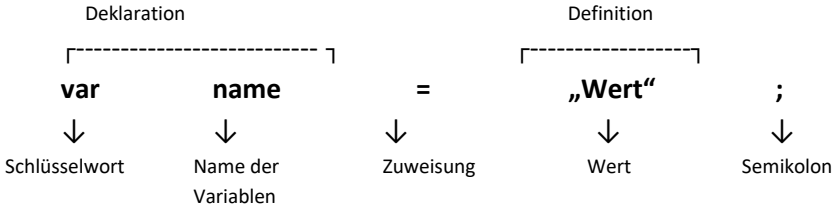
-> **Vergleichsoperatoren**:

<	Kleiner als
>	Grösser als
<=	Kleiner gleich als
>=	Grösser gleich als
==	Gleich wie (Wertevergleich / siehe Bem *)
===	striktes Gleich (Werte- & Datentypenvergleich)
!=	Ungleich (nur Werte)
!==	Striktes Ungleich (Werte & Datentypen)

Bem* Dieser Vergleich geht bei JavaScript auch zwischen verschiedenen Datentypen! Es werden immer mindestens zwei Gleichheitszeichen verwendet, da ein einzelnes Gleichheitszeichen als Zuweisung für Variablen verwendet wird!

V6: Variablen:

-> Variablen sind Stellvertreter für bestimmte Werte. Variablen-Bezeichnung in JavaScript:



-> In einem var-Statement lassen sich mehrere Variablen (z.B. var i = 1, res = [];) unterbringen.

-> Auslesen einer Variablen durch Eingabe des nackten Variablennamens (z.B. name)

-> Beispiele für JavaScript:
```
var x = 5;
var domain = „css3-html5.de“;
var juenger = true;
```

-> analoge Beispiele in PHP:
```
$x = 5;
$domain = „css3-html5.de“;
$juenger = true;
```

-> Beim Ändern von gesetzten Variablen muss bei JavaScript „var“ nicht mehr eingegeben werden (z.B. x = 6). Var ist nur beim erstmaligen Anlegen einer Variablen nötig.

-> x = x + 2 führt in jedem Schritt zu einem neuem x (für Zähler wichtig)

-> Schreibweise x += 2 führt zu gleichem Resultat (für Zähler wichtig)

-> Diese Schreibweise lässt sich anwenden für:

Addition:	x += y	x = x + y (auch zum Anfügen von Teilstrings)
Subtraktion:	x -= y	x = x - y
Multiplikation:	x *= y	x = x * y
und Division:	x /= y	x = x / y
oft verwendet	x++ oder ++x	x = x + 1 (Bem. [1])
oft verwendet	x-- oder --x	x = x - 1 (Bem. [2])

Bem. [1]: x++ ist nicht gleich ++x. Bei x++ wird zuerst addiert und dann ausgegeben und bei ++x wird zuerst ausgegeben und dann addiert

Bem. [2]: x-- ist nicht gleich --x. Bei x-- wird zuerst subtrahiert und dann ausgegeben und bei --x wird zuerst ausgegeben und dann subtrahiert

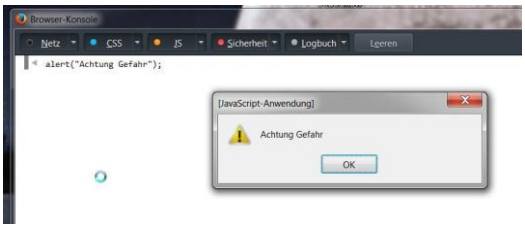
Bem. 3: Es gibt lokale (innerhalb Funktionen gültige) und globale (ausserhalb & innerhalb Funktionen gültige) Variablen.

V7: Funktionen aufrufen

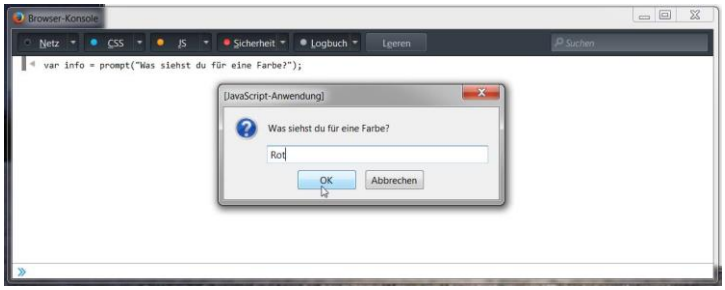
-> Funktion besteht aus:

functionName(param1, param2, param3, ..)
↓ ↓
Funktionsname Argumente

-> **Anzeigebox**: alert("Achtung Gefahr");
Nach Start mit „Enter" erscheint eine JavaScript Warnmeldung mit diesem Text



-> **Dialogfeld**: prompt("Was siehst du für eine Farbe?",""); bzw prompt("Was siehst du für eine Farbe?","Rot");
Nach Enter erscheint eine JavaScript Dialogfeld mit Eingabemöglickeit, zweites Atribut zeigt Voranzeige im Eingabefeld die überschrieben werden kann.



Nach Eingabe und OK erscheint die Eingabe im Script nun als **Rückgabewert**; speicherbar in einer Variablen. Bei Abbrechen wird ‚null' zurückgegeben.

V8: Wie verarbeitet man Strings?

-> Strings werden durch Anfürungszeichen zusammengehalten. Zwei Strings können durch ein Plus vernüpft werden. Zahlen und Variablen können mit Plus mit Strings verknüpft werden.

-> Die in der Variablen „info" abgelegte, eingegebene Information (Rot) wird nun auf verschiedene Kriterien untersucht. In diesem Übungsstück zum Beispiel

Durch einen Punkt nach der Eingabe von info ins Konsolenfeld erscheint ein Auswahlmenü in dem „length" ausgewählt wird. Damit kann die Länge der Variablen „info" bestimmt werden.

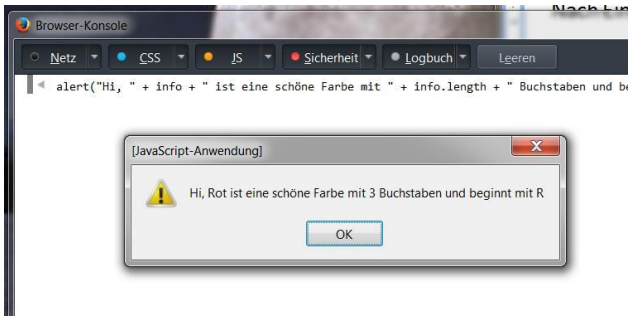
Durch Anfügen einer 0 zwischen eckigen Klammern nach der Variablen „info" wird der erste Buchstabe des info-Strings angezeigt.

-> Die Länge eines Strings kann mittels anfügen von .lenth an den String bestimmt werden. Z.B. "AAAA".length

-> Nun wird ein Antwortstring zusammengesetzt und in einer Alert-Box ausgegeben: alert("Hi, " + info + " ist eine schöne Farbe mit " + info.length + " Buchstaben und beginnt mit " + info[0]);

-> Zum Herauslesen eines bestimmten Zeichens aus einem String wird nach dem String eine Eckige Klammer mit der Position angehängt. Z.B. "ABCD"[0] ist A.

-> Zahlen innerhalb eines Strings können nicht einfach zusammengezählt werden, da dabei nur die einzelnen Ziffern hintereinander gestellt werden! In diesem Fall muss jede Zahl vorher in ParseInt umgewandelt werden.



… und es erscheint die zusammengesetzte Antwort.

Hier das vollständige JavaScript mit einem vollständigen Auswertungstext:

```
alert("Achtung Gefahr");
var info = prompt("Was siehst du für eine Farbe?");
alert("Hi, " + info + " ist eine schöne Farbe mit " + info.length + " Buchstaben und beginnt mit " + info[0]);
```

Hier dasselbe JavaScript mit zusammengesetzten Auswertungstext-Teilen:

```
alert("Achtung Gefahr");
var info = prompt("Was siehst du für eine Farbe?");
var output = "Hi, " + info;
output += " ist eine schöne Farbe mit " + info.length;
output += " Buchstaben und beginnt mit " + info[0];
alert(output);
```

-> **Bem**. Der Wert in der Alert-Klammer hat keine Stringzeichen, da die Variable bereits einen String-Wert hat!

V9: Das EVA-Prinzip

-> Eingabe -> Verarbeitung -> Ausgabe ist das Grundprinzip der Computerverarbeitung

Beispiel mit Quadratzahl-Erzeugung:

```
var eingabe = prompt("Gib eine Zahl ein ");
var output = "Die Quadratzahl von " + eingabe;
output += " ist " + eingabe * eingabe;
alert(output);
```

Beispiel mit Summen-, Differenz-, Produkt- und Quotient-Erzeugung:

```
var eingabe1 = prompt("Gib die erste Zahl ein");
var eingabe2 = prompt("Gib die zweite Zahl ein");
alert("Die erste Zahl lautet " + eingabe1 + " und die zweite lautet " + eingabe2 + "\nDie Summe ist " +
    (parseInt(eingabe1) + parseInt(eingabe2)) + "\nDie Differenz ist " + (eingabe1 - eingabe2) + "\nDas
    Produkt ist " + eingabe1 * eingabe2 + "\nDer Quotient ist " + eingabe1 / eingabe2 )
```

Bem: \n bedeutet neue Zeile

Gleiches Beispiel mit gestaffelter Ausgabe:

```
var eingabe1 = prompt("Gib die erste Zahl ein");
var eingabe2 = prompt("Gib die zweite Zahl ein");
var differenz = (eingabe1 - eingabe2);
var produkt = eingabe1 * eingabe2;
var quotient = eingabe1 / eingabe2;
var summe = (parseInt(eingabe1) + parseInt(eingabe2));
var ausgabe = "Die erste Zahl ist " + eingabe1;
ausgabe += " und die zweite Zahl" + " lautet " + eingabe2;
ausgabe += "\n Die Summe ist " + summe;
ausgabe += "\n Die Differenz ist " + differenz;
ausgabe += "\n Das Produkt lautet " + produkt;
ausgabe += "\n und der Quotient beträgt " + quotient;
alert(ausgabe);
```

V10: Was sind if und else

Normales Conditional Statement bei mehreren Anweisungen:

```
if (Bedingung) {
Anweisungscodes 1
} else {
Anweisungscodes 2
}
```

Die geschweifte Klammer wird zur Zusammenfassung mehrer Anweisungen benötigt. Bei nur einer einzigen Anweisung kann darauf verzichtet werden (gilt übrigens auch für else-Teil). Sollte aber wenn möglich nicht gemacht werden, da das nur eine Eigenheit von JavaScript ist! Bei else welches gerade in ein if leitet kann direkt mit else if gearbeitet werden.

Vereinfachtes Conditional Statement ohne else mit einer einzigen Anweisung:

```
if (Bedingung) Anweisungscode
```

Beispiel mit Vergleich der Längen zweier Namen:

```
var name1 = prompt("Gib den Vornamen Oli ein");
var name2 = prompt("Gib deinen Vornamen ein");
if (name1.length < name2.length) {
alert("Mein Name ist kürzer als dein Name");
} else {
alert("Mein Name ist länger");
}
```

Bem. geht nur, wenn die Variablen vorher einzeln eingegeben werden / Darum Prompt- Abfrage eingefügt!
Längenbestimmung mittels Beifügen von .length an Variablenname: z.B. name1.length
Vergleich name1.length < name2.length führt zu true und zu Anweisungscode 1

Beispiel mit Begrüssungstext:

```
var lang = prompt("Gib deine Mutersprache z.B. de oder en an");
var gender = prompt("Gib dein Geschlecht z.B. m oder w an");
if (lang == "de") {
    if (gender == "m") {
    alert("Hallo mein Herr");
    } else {
    alert("Hallo, die Frau");
    }
} else {
    if (gender == "m") {
    alert("Hello, sir");
    } else {
    alert("Hello, madam");
    }
}
```

Bem. geht nur, wenn die Variablen vorher einzeln eingegeben werden / Darum Prompt- Abfrage eingefügt!

Beispiel mit Zahlenuntersuchung auf gerade / ungerade:

```
var nr = prompt("Gib eine Zahl ein");
If (Analyse auf ungerade Zahl) {
alert("Die Zahl " + nr  +" ist ungerade");
}else{
alert(„Die Zahl " + nr  +" ist gerade“);
}
```

Komplizierte Methode zur Analyse:

```
if ((""+nr/2)[(""+nr/2).length - 2] == "." ){
alert("Die Zahl " + nr  +" ist ungerade");
}else{
alert("Die Zahl " + nr +" ist gerade");
}
```

V11: Was ist Modulo

„Modulo“ ist der Rest einer Division.
Modulo 2 zeigt den Rest bei Division durch 2! Modulo 3 zeigt den Rest bei Division durch 3!
Bei ungeraden Zahlen heisst der Rest einer Division durch 2 immer 1

Daher ist (nr % 2) = 1 bei ungeraden Zahlen und die Funktion if(nr % 2) gibt 1 bzw true, womit die nachfolgende Anweisung durchgeführt wird.

Die Analyse ob Zahl gerade ist kann daher mit Modulo viel einfacher gemacht werden und lautet::
Script dazu:

```
var nr = prompt("Gib eine Zahl ein");
if (nr % 2) {
alert("Die Zahl " + nr  +" ist ungerade");
```

```
}else{
alert("Die Zahl " + nr +" ist gerade");
}
```

V12: Truthy & Falsy

Im JavaScript gilt das „Truthy & Falsy“- Konzept, das heisst Werte werden wie folgt als True oder False angenommen:

	Truthy	Falsy
String	alle nicht-leeren Strings	alle leeren Strings
Number	alle Zahlen ausser 0	0, NaN *)
Undefined		undefined

*) NaN bedeutet -> not a number

Anwendung: if (String) { Code} -> Wenn String vorhanden wird Klammer Truth und {Code} abgearbeitet.

Aufgabe: erstelle JavaScript für folgende Abfrage:

Test ob eine Eingabe eine Zahl oder keine Zahl ist kann mit der Funktion isNaN() (isNaN heisst „is not a number“) erfolgen.

isNAN(123) -> false (d.h. Zahl) isNAN(ABC) -> true (d.h. keine Zahl!)

Script dazu:

```
var eingabe = prompt("Geben Sie eine Zahl ein");
if (isNaN(eingabe)) {
alert("Bitte geben Sie eine Zahl ein!");
}else{
if (eingabe % 2) {
alert("Die Zahl " +eingabe +" ist ungerade");
}else{
alert("Die Zahl " +eingabe +" ist gerade");
}
}
```

Aufgabe: Herausfinden ob eine Zahl im gewünschten Bereich von 0 … 100 liegt.

Script dazu:

```
var eingabe = prompt("Geben Sie eine Zahl ein");
```

```
if (isNaN(eingabe)) {
alert(eingabe +" Bitte geben Sie eine Zahl ein!");
}else{
	if (eingabe >= 0) {
		if (eingabe <= 100) {
		alert("Die Zahl " + eingabe +" liegt im gewünschten Bereich");
		}else{
		alert("Die Zahl " + eingabe +" liegt ausserhalb des Bereiches");
		}
	}else{
	alert("Die Zahl " + eingabe +" liegt ausserhalb des Bereiches");
	}
}
```

V13: Was sind logische Verknüpfungen (&&, ||, !1 bzw. !0)

Mit && (AND), || (OR) und !0 bzw. !1 (NOT) lassen sich Bedingungen von if-Abfragen logisch verknüpfen und dadurch die Anzahl Abfragen reduzieren.

Aufgabe: Vereinfachen des vorherigen Scripts, so dass weniger if-Abfragen nötig sind.

```
var eingabe = prompt("Geben Sie eine Zahl ein");
if (isNaN(eingabe)) {
alert(" Bitte geben Sie eine Zahl ein!");
}else{
	if (eingabe >= 0 && eingabe <= 100 ) {		// bzw. if(!(eingabe < 0) && !(eingabe > 100)) {
	alert("Die Zahl " + eingabe +" liegt im gewünschten Bereich");
	}else{
	alert("Die Zahl " + eingabe +" liegt ausserhalb des Bereiches");
	}
}
```

Aufgabe: Zuordnen von Namen zu einem Bedienschalter

```
var eingabe = prompt("Geben Sie ihren Namen ein");
if (!(isNaN(eingabe)))  {
alert(" Bitte geben Sie ein Name ein!");
}else{
If (eingabe[0] === "A" || eingabe[0]=== "B") {
alert("gehen sie zu Schalter 1");
If (eingabe[0] === "C" || eingabe[0]=== "D") {
alert("gehen sie zu Schalter 2");
}
```

V14: Was sind Kommentare?

Kommentare dienen zum besseren Begreifen des Programms und werden durch Bezeichnen mittels // oder /* ..*/ eingefügt.

Nach // wird alles ignoriert was auf dieser Zeile steht. Bei mehreren Zeilen am Anfang vor jeder Zeile // setzen
Nach /* wird alles ignoriert bis zum Erscheinen von */; geht über mehrere Zeilen ohne spezielles Zeichen!
Bem: <!-- Bedeutet ausblenden bzw. verstecken des Scripts über mehrere Zeilen auf html-Seite -->

V15: Wie schreibt man Funktionen?

Mit Funktionen kann man Anweisungen automatisiert hintereinander ausführen.

Schreibweise 1 (als Funktion):

Schlüsselwort	Funktionsname	runde Klammern	Ausführungscode in geschweiften Klammern	
function	**functionName**	**()**	**{**	**Anweisung }**

Schreibweise 2 (als Variable):

Variable mit Funktionsname		Funktion	runde Klammern	Ausführungscode in geschweiften Klammern	
var functionName	=	**function**	**()**	**{**	**Anweisung }**

Die Funktion ist eine Art ein Unterprogramm, das mit dem Funktionsnamen aufgerufen werden kann.

Beispiel 1: Funktion mit Verwendung eines Scripts

Script:

```
var name = prompt("Wie heisst du?");
alert("Hey, "+name +"! Schön dich zu sehen!");
```

Funktion mit Verwendung dieses Scripts:

```
function hallo() {
    var name = prompt("Wie heisst du?");
    alert("Hey, "+name +"! Schön dich zu sehen!");
}
```

Beispiel 2: Funktion mit mehreren Alertboxen, die der Reihe nach angezeigt werden

```
function abc() {
    alert("a");
    alert("b");
    alert("c");
}
```

Beispiel 3: Funktion abcdef bestehend aus zwei Einzelfunktionen

```
function abc() {
```

```
        alert("a");
        alert("b");
        alert("c");
}
function def() {
        alert("d");
        alert("e");
        alert("f");
}
function abcdef() {
        abc();
        def();
}
```

Innerhalb von Funktionen können weitere Funktionen aufgerufen werden. Solche Funktionszusammenfassungen nennt man Callstack / Aufrufstapel.

Funktionen mit Parametern:

function functionName (param1, paramx) { Anweisung }

Beispiel 1: Zeigt erstes Zeichen eines als Parameter angegebenen Strings an

```
function zeigeZeichen(str) {
        if (str) {     // prüft ob ein String vorhanden ist
        alert("Das erste Zeichen ist ein " + str[0]);
        }
}
```

Beispiel 2: Zeigt gewünschtes Zeichen eines als Parameter angegebenen Strings an

```
function zeigeZeichen(str,i) {
        if (isNaN(i)) {       // prüft ob der Parameter i vorhanden ist
                i = 0;
                }
                if (str) {       // prüft ob ein String vorhanden ist
                alert("Das " + (i+1) ". Zeichen ist ein " + str[i]);
                }
}
```

Abfrage mittels „zeigeZeichen(„Ukulele“,5); “ ergibt 5. Zeichen ist ein L

Beispiel 3: Zeigt Zeichen eines als Parameter angegebenen Strings der Reihe nach an.

```
function zeigeZeichen(str, i) {
        if (isNaN(i)) {
                i = 0;
                }
                if (str) {
                alert("Das " + (i+1) ". Zeichen ist ein " + str[i]);
                zeigeZeichen(str, i+1);
}
```

}
Das ***ist extrem gefährlich*** und endet in einem unendlichen Loop mit der Meldung: „Maximal Call Stack Size is exeeded“. Eine Funktion die sich selber wieder aufruft nennt sich Rekursion.

Variablen innerhalb von Funktionen können nicht von aussen abgerufen werden (Function scope). Rückgabewert mittels Schlüsselwort „Return“ ist aber immer möglich.

V16: Was sind Rekursionen?

Rekursion ist eine Funktion die sich selber wieder aufruft! Sie kann zu einem unendlichen Loop führen und muss daher immer mit einer **Abbruchbedingung** versehen werden.

Möglicher Countdown mit Abbruchbedingung:

```
function countdown (nr)  {
        if (nr >= 0) {
        alert(nr);
        countdown(nr - 1);
        }
        }
```

Aufgabe: Wie kann Beispiel 3 vor Rekursion geschützt werden? ***Bem***: Im folgenden sind Str (String) und i (Stelle im String) Parameter der Funktion. Einfügen einer Abbruchfunktion!

```
function zeigeZeichen(str, i) {
        if (isNaN(i)) {
                i = 0;
        }
if (i === str.length) {      // Einleitung Abbruchfunktion!
return;
}
if (str) {
        alert("Das " + (i+1) ". Zeichen ist ein " + str[i]);
        zeigeZeichen(str, i+1);
        }
}
```

Weitere mögliche Abbruchfunktionen:
Bedingung if (str) {   erweitern auf if (str && i < str.length) {   -> geht ohne Return!
oder
Bedingung if (str) {   erweitern auf if (str && str[i]) {   -> raffinierter, geht auch ohne Return!

V17: Arrays

Arrays sind im Grunde genommen Listen. Sie haben ein Length-Attribut angehängt. Diese Attribut lässt sich elegant mittels ArrayName.length anzeigen. Hier eine Variable mit einem Array:

var cast = ["Jodie", "Lawrence", "Kasi", "etc"];

Auslesen der Länge des Arrays bzw. Bestimmen des Length-Attributs:
cast.length -> 4 (Bem. Letzte Position ist aber [3] wegen Zählfolge ab 0)

Auslesen einer Variablen an 1. Stelle ergibt (Zählfolge ab 0!!)
cast [0] -> "Jodie"

Überschreiben von Stellen im Array:
cast [0] = "Walo" überschreibt "Jodie" mit "Walo"

Aufgabe: Erstellen einer Funktion, die einen neuen Eintrag zuhinterst an ein Array anhängt.
Script:

```
function push (arr, element) {
var eintrag = "element"+" , ";
arr [arr.length] = eintrag;
}
```

Bem: Die Funktion push ist eine existierende, normalisierte Funktion!

V18: Schleifen (while)

Schleifen sind bei der Abfrage von Arrays nötig. Damit werden Iterationen (Näherungsschritte) gemacht, wobei das Programm gewisse Aufgaben immer wieder wiederholt.

Beispiel 1: String zusammensetzen aus den Elementen eines Arrays

```
var array = ["Jodie", "Lawrence", "Kasi", "etc"];
var resString = "";
function iterator (i) {
        if (i < array.length) {
        resString += array[i] + ", ";
        iterator (++i);
        }
}
```

Auslösung mit -> iterator (0); ergibt Ausgabe: "Jodie, Lawrence, Kasi, etc, "

Beispiel 2: String zusammensetzen aus den Elementen eines Arrays mittels einer Funktion:

```
var array = ["Jodie", "Lawrence", "Kasi", "etc"];
function join (arr, delimiter) {
        var resString = "";
        function iterator (i) {
            if (i < arr.length) {
                resString += arr[i];
                if (i < arr.length -1) {
                resString  += delimiter;
                }
```

```
            iterator (++i);
        }
    }
    iterator(0);
    return resString;
}
```

Auslösung mittels: -> join(array, "|") ergibt Ausgabe: "Jodie|Lawrence|Kasi|etc| "

Anstelle dieser Code-Konstruktionen verwendet man meist **Schleifen**.

Die **while-Schleife** führt den Code aus solange die Bedingung wahr ist. Der Code wird nur ausgeführt, solange die Bedingung erfüllt ist. Wird die Bedingung von Anfang an nicht erfüllt, so wird der Code überhaupt nie ausgeführt.!

while (Bedingung) { Anweisung }

Lösung des vorangehenden Beispiels 2 mit einer while-Schleife:

```
var array = ["Jodie", "Lawrence", "Kasi", "etc"];
function join (arr, delimiter) {
    var resString = "";
    var i = 0;
        while (i < arr.length) {
        resString += arr[i];
        if (i < arr.length -1) {
        resString  += delimiter;
        }
        ++i;
    }
    return resString;
    }
```

Auslösung mittels: -> join(array, " $$ ") ergibt Ausgabe: "Jodie $$ Lawrence $$ Kasi $$ etc "

V19: Schleifen (for / do while / for in)

Schleifen sind bei der Abfrage von Arrays nötig. Damit werden Iterationen (Näherungsschritte) gemacht, wobei das Programm gewisse Aufgaben immer wieder wiederholt.

Die **for-Schleife** **führt den Code aus solange die Bedingung wahr ist**. Sie hat die Eigenheit, dass sie die Initialisierung, Endbedingung und die Zählervariable im Kopf der Schleife enthält. Der Code wird daher nur solange ausgeführt wie die Bedingung erfüllt ist!

Einmal, bevor Schleife beginnt	Jedes Mal, vor Durchlauf	Jedes Mal nach Durchlauf
↓	↓	↓

for (Initialisierung; Bedingung; Zählanweisung) { Anweisung }

Beispiel mit for-Schleife:

aufwärtszählend:

```
for (var i = 0; i < 3, i++) {
alert(i);
}
```

abwärtszählend:

```
for (var i = 3; i > 0, i--) {
alert(i);
}
```

Umwandlung des while-Schleifen Beispiels in eine for-Schleife

```
var array = ["Jodie", "Lawrence", "Kasi", "etc"];
function join (arr, delimiter) {
    var resString = "";
    var i = 0;
        while (i < arr.length) {
            resString += arr[i];
            if (i < arr.length -1) {
            resString  += delimiter;
            }
            ++i;
        }
    return resString;
}
```

```
var resString = "";

for(var i = 0; i < arr.length; ++i) {
    resString += arr[i];
    if (i < arr.length -1) {
    resString  += delimiter;
    }

}
    return resString;
    }
```

Auslösung mittels: -> join(array, " $$ ") fürt zu Ausgabe: "Jodie $$ Lawrence $$ Kasi $$ etc "

Die **do / while-Schleife** führt den Code vor der Bedingung durch und prüft nachher ob die Bedingung wahr ist. Der Code wird daher immer mindestens einmal ausgeführt!

do { Anweisung } while (Bedingung)

Beispiel mit do / while-Schleife:

```
do {
alert("Hi");
} while (false)
```

Die **for / in-Schleife** führt den Code nur in einem bestimmten Objekt aus. Sie ist die langsamste Schleife und sollte daher nicht für Arrays benützt werden.

for (Iterator in Objekt) { Anweisung }

Notbremse bei Schleifenapplikationen:

```
var counter = 0;
while (true) {
counter++;
if (counter === 3000) {
break;
}
```

break; -> springt aus der gesamten Schleife heraus
continue; -> überspringt den Rest des aktuellen Schleifendurchlaufs und fährt mit dem nächsten fort.

V20: Wie findet man Primzahlen

Aufgabe: Programm schreiben, das alle Primzahlen in einem Zahlenbereich erkennt (z.B. findPrimes(50)).

Auflösung Teil 1 (Anzeige aller Primzahlen im Bereich der höchsten Zahl):

```
function findFactors (nr) {          // Funktionsname mit höchster Zahl
    var res = [];                    // Variable wird als Array eröffnet
    for ( var i = 1; i <= nr; i++)  {    // Setzen des Schleifenkopfes (Zählvariable i, Stoppbed., Zählschritt
        if (!(nr % i)) {             // wenn Modulo i der Eingabe null ist wird if-Schleife freigegeben
        res[res.length] = i;         // Wert der Zählvariablen i wird in das nächst leere Feld des Arrays gesetzt
    }
    }
    return res                       // Array wird zurückgegeben
}
```

Bem. Modulo i der Eingabe heisst „**Rest der Eingabe durch i**“. Wenn Rest null ist Eingabe durch i teilbar. Primzahlen sind nur durch sich und durch eins teilbar; d.h. es gibt nur 2 Einträge im Array.

Auflösung Teil 2 (Anzeige ob eingegebene Zahl eine Primzahl ist oder nicht (Ausgabe true oder false):

```
function isPrime (nr) {                  // Funktionsname mit interessierender Zahl
    var factors = findFactors (nr);      // gefundener Array wird gesetzt
        if (factors.length === 2) {      // Wenn Array-Länge 2 ist (Primzahl!) wird if-Schleife freigegeben
        return true;
    }else{                               // Wenn Array-Länge ungleich 2 ist, wird auf else-Zweig geschaltet
    return false;
    }
```

Dasselbe etwas vereinfacht:

```
function isPrime (nr) {                  // Funktionsname mit interessierender Zahl
    var factors = findFactors (nr);      // gefundener Array wird gesetzt
    ~~if (factors.length === 2) {~~      // überflüssiges Statement: gibt true zurück wenn factor.length = true ist
    ~~return true;~~
    ~~}else{~~                           // überflüssiges Statement: gibt false zurück wenn factor.length = false ist
    ~~return false;~~
    ~~}~~
    return factors.length === 2;         // Ersatz des überflüssigen Statements
}
```

Und nun nochmal einfacher durch direktes Einsetzen der Variablen in den Returnbefehl:

```
function isPrime (nr) {
    ~~var factors = findFactors (nr);~~
    return ~~factor~~ findFactors (nr).length === 2;
}
```

Vollständige Auflösung der Aufgabe:

```
function findFactors (nr) {                 // Funktionsname mit höchster Zahl
    for ( var i = 1, res = []; i <= nr; i++)  {   // Setzen des Schleifenkopfes (Zählvariable i, Array-Variable,
                                                  // Stoppbedingung, Zählschritt
        if (!(nr % i)) {                          // wenn Modulo i der Eingabe null ist wird if-Schleife freigegeben
        res[res.length] = i;                      // Wert der Zählvariablen i wird in das nächst leere Feld des Arrays
                                                  // gesetzt
    }
    }
    return res;                                   // Array wird zurückgegeben
}
function isPrime (nr) {                           // Funktion zum Bestimmen ob nr eine Primmzahl ist
    return findFactors(nr).length === 2;          // Gibt bei nr = Primmzahl ein true zurück
}
function findPrimes (nr, from) {                  // Funktion zum finden aller Primzahlen von from bis zur höchsten
                                                  // Zahl nr
if (isNaN(from)) {                                // Kontrolle der from-Eingabe
from =1;                                          // wenn keine Zahl, wird 1 gestzt
}
    for ( var i = from, res = []; i <= nr; i++) {     // Setzen des Schleifenkopfes (Zähl-Variable i, Array-Variable,
                                                      // Stoppbedingung, Zählschritt
        if (isPrime(i)) {                             // wenn Funktion isPrime true ist wird if Schleife freigegeben
        res[res.length] = i;                          // Wert der Zählvariablen i wird in das nächst leere Feld des
                                                      // Arrays gesetzt
    }
    }
    return res;                                       // Array wird zurückgegeben
}
```

V21: Wie spaltet man Strings?

-> Die Funktion split() ist das Umgekehrte der Funktion join() von V18. Hier wird der String anhand des Trennzeichens (deliminator) in einen Array aufgeteilt.

Entwicklunsresultat mit eine Funktion:

```
var string = "abc|de|fghi|klasdf|sdkfjhsdf|asd";
var delimiter = "|";
function split (str, delimiter) {
    for ( var i = 0, res = [], tmpStr "", lastDelimiter = 0; i < str.length; i++) {
        if (str(i) === delimiter) {
            for ( var j = lastDelimiter; j < i; j++) {
            tmpStr += str[j];
            }
            res[res.length] = tmpStr;
            tmpStr = "";
            lastDelimiter = i + 1;
```

```
}
}
Return res;
}
```

Aufgeteilt in 2 Funktionen: eine für innere Schleife des Teilstrings und eine für Stringfunktion:

```
var string = "abc|de|fghi|klasdf|sdkfjhsdf|asd";
var delimiter = "|";
function substring (str, start, end) {
    for ( var j = start, tmpStr ""; j < end; j++) {
        tmpStr += str[j];
    }
    Return tmpStr;
}
function split (str, delimiter) {
    for ( var i = 0, res = [], lastDelimiter = 0; i < str.length; i++) {
        if (str(i) === delimiter) {
            res[res.length] = substring(str, lastDelimiter, i);
            lastDelimiter = i + 1;
            }
        }
        Return res;
    }
```

Aufrüstung der Funktion für mehrzeichige Delimiter wie z.B. " $$ ":

```
var string = "abc $$ de $$ fghi $$ klasdf $$ sdkfjhsdf $$ asd";
var delimiter = " $$ ";

function substring (str, start, end) {
    for ( var j = start, tmpStr ""; j < end; j++) {
        tmpStr += str[j];
    }
    Return tmpStr;
}
function split (str, delimiter) {
    for ( var i = 0, res = [], lastDelimiter = 0; i < str.length; i++) {
        if (str(i) === delimiter) {    Vergleich eines einzigen Zeichens
        if (substring(str, i, i + delimiter.length) === delimiter {    Vergleicht nun mehrere Zeichen
            res[res.length] = substring(str, lastDelimiter, i);
            lastDelimiter = i + delimiter.length;    zählt nun mehrere Zeichen dazu
            }
        }
        Return res;
    }
```

Bem. Die eben entwickelte Split-Funktion gibt es schon, genau gleich wie die Join- und die Push-Funktionen. Sie können im Auswahlmenü nach der Eingabe eines Punktes nach der Variablen ausgewählt werden (wie bei der .length-Funktion).

V22: Objekt-Orientierte Programmierung

Bis jetzt wurde die prozedurale Programmierung gemacht bei welcher die Funktionen und Prozeduren nacheinander ausgeführt wurden. Bei der projekt-orientierten Programmierung stehen die Funktionen und Prozeduren nun neu auch noch in Zusammenhang mit den Objekten. Hier besitzen die Objekte verschiedene objekteigene Eigenschaften (Funktionen oder Werte) die auf das Objekt zugeschnitten sind (z.B. robot.walk() bzw. duck.walk()). JavaScript ist für das projektorientierte Programmieren zugeschnitten ganz im Gegensatz zu PHP. Es ruft zuerst das Objekt und anschliessend hinter einem Punkt bzw. in angefügten eckigen Klammern die Funktion auf:

Programmierung	JavaScript	PHP
Allg. Struktur	Objekt.property Objekt[property]	Property(Objekt)
Beispiel String-Länge	"string".length "string" [0]	strlen("string");

V23a: Primitive und Reverence Types

Vergleich von gleichen Strings ergibt true, aber Vergleich von gleichen Arrays ergibt false. Warum?
Der Grund liegt darin, dass in JavaScript je nach Datentyp verschieden abgespeichert wird. Bei den primitive types werden die Werte direkt abgespeichert, bei den reference types werden nur die Referenzen zum Ort der abgelegten Werte (Array oder Objekt) abgespeichert.

Primitive types	Undefined Boolean Number String	Hier werden Werte direkt abgespeichert
Reference types	Function Object	Hier werden nur die Referenzen abgespeichert

Ein Vergleich von refenzierten Daten (Array, Objekte) führt immer zu false, da die Referenzen auch bei identischen Daten nie gleich sein können. Das ist eine Eigenheit des JavaScripts!

V23b: Wichtigste Array-Methoden

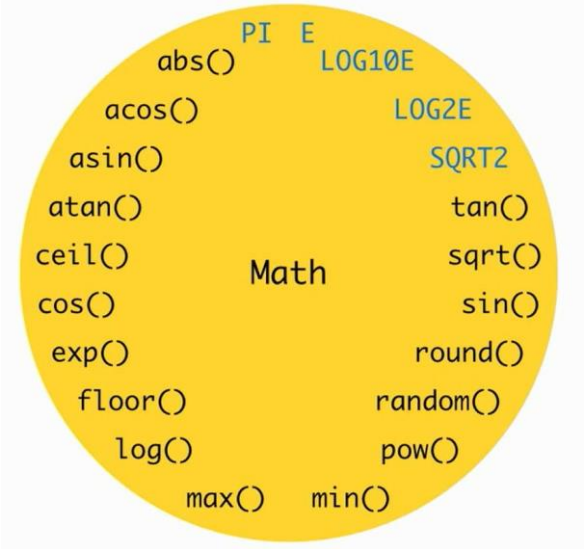
Hier eine Zusammenstellung der möglichen Bearbeitungsfunktionen für Arrays :

var arr = "a b c d e f" -> setzt Array
arr -> "a b c d e f" –> liest Array
arr.split("") -> ["a", "b", "c", "d", "e", "f"] -> splittet nach jedem Leerschlag

arr.push(1) -> ["a", "b", "c", "d", "e", "f", 1] -> setzt 1 ans Ende des Strings
arr.pop() -> ["a", "b", "c", "d", "e", "f"] -> entfernt letztes Element des Strings
arr.reverse() -> ["f", "e", "d", "c", "b", "a"] -> kehrt String um
arr.shift(f) -> ["e", "d", "c", "b", "a"] -> entfernt f von der ersten Position
arr.reverse() -> ["a", "b", "c", "d", e"]
arr.unshift(f) -> ["f", "a", "b", "c", "d", e"] -> setzt f als erstes Element
arr.slice(3) -> ["c", "d", e"] -> kopiert die Elemente ab 3. Position heraus
arr.slice(1, 3) -> ["a", "b"] -> kopiert die Elemente ab 1. bis vor 3. Position heraus
arr.slice(1, 3).concat(arr.slice(3)) -> ["a", "b", "c", "d", e"] -> fügt slice(3) wieder an arr.slice(1, 3) an.
arr.sort() -> ["a", "b", "c", "d", "e", "f"] -> sortiert alphabetisch
arr.splice(2, 1) -> ["c"] -> schneidet 1 Zeichen nach 2. Stelle heraus

V24: Das Math-Objekt

Das Math-Objekt besitzt alle grundlegenden Funktionen und Werte die man für fortgeschrittene Rechnungen braucht. Aufruf durch „Math“ und anschliessender Punkt eröffnet das Properties-Wahlmenü.



Das Math-Objekt bildet einen „NameSpace“ d.h. einen Raum für global zusammengehörende Math-Funktionen und Werte. Konstanten werden in der Programmierung per Konvention immer grossgeschrieben!

V25: Wie erstellt man eigene Objekte?

Arrays und viele andere Sachen können Objekte sein. Beispiel für ein „Objekt“-Objekt:

var o = {} mit geschweiften Klammern. Bei Abfrage mit o wird hier object {} angezeigt, was heisst, dass das Objekt zwischen den Klammern ist.

Nach einem Punkt hinter o wird das Propertyfeld mit den Möglichkeiten zur Auswahl geöffnet:

> o.
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
construktor

hasOwnProperty
isPrototypeOf
…

Nach Wahl von „myProperty = my value“ kommt folgendes heraus:

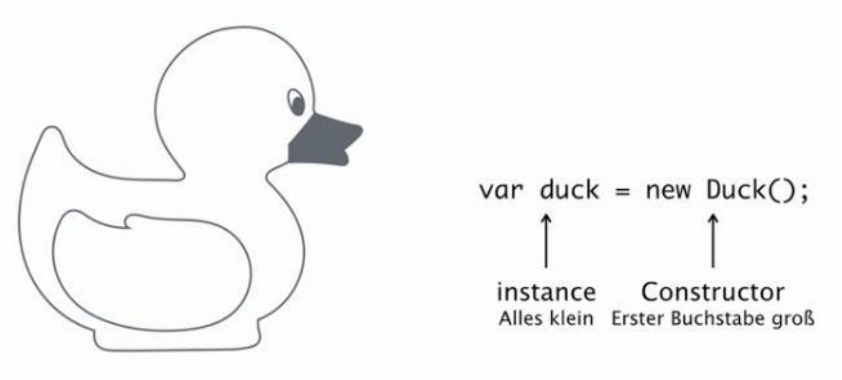
```
> o
  Object {}
> o.myProperty = "my value";
  "my value"
> o
  Object {myProperty: "my value"}
> o = {"myProperty": "my value"};
  Object    {myProperty: "my value"}
> o = {myProperty: "my value"};
  Object    {myProperty: "my value"}
```

Beispiel durch Erstellen eines Objekts „actress“ mit verschiedenen gesetzten Properties:

```
> var actress = {name; "Jodie Foster", gender: "female", birthday: "19.11.1962"}
  undefined
> actress
  Object    {name; "Jodie Foster", gender: "female", birthday: "19.11.1962"}
> actress.name
  "Jodie Foster"
> actress.gender
  "female"
> actress.birthday
  "19.11.1962"
```

Nun können die Werte der Properties elegant herausgelesen werden.

Objekte werden oft durch Ableiten aus einer Vorlage erstellt. Hier im Beispiel wird ein neues Enten-Objekt erstellt, indem aus dem alten, bestehenden Objekt (instance) „duck“ ein neues Objekt (Constructor) „new Duck ()“ abgeleitet wird:



V26: Das Date-Objekt in JavaScript

In JavaScript wird für Zeiten und Datum das Date-Object verwendet. Es erlaubt „neue“ Date Variablen zu schaffen:

Eingabe von „new Date()“ gibt aktuelles Datum und Uhrzeit in folgender Form an:
Thu Mar 06 2014 11:21:02 GMT+0100 (CET)
und beinhaltet die Zeit zu der es geschaffen wurde!

Als Variable gespeichert var d = new Date(); gibt es daher auch bei späteren Abfragen immer den Wert der Entstehung der Variablen.

Soll das Date Objekt andere Ausgaben geben, so müssen wir die Argumente in der Klammer setzen:

Bei Eingabe von „**new Date(2014, 5, 1)**“ wird **Sun Jun 01 2014 00:00:00 GMT+0200 (CEST)** angezeigt.
Argumente in Klammern gehen immer von Gross nach klein / Vorsicht: Argument Monat beginnt bei 0!

Auslesen von d.getTime gibt z.B. eine riesige Zahl. Was bedeutet das? Das sind die Milisekunden, die seit dem 1.1.1970 abgelaufen sind. Für was ist diese Zahl? Computer zählen die Zeit, und der 1.1.1970 um 01.00 Uhr ist der 0-Ausgangswert für alle Computer (The Epoch).

Das 2038-Problem: Der Computer kann nur bis zu einer maximalen Zahl zählen und diese Zahl wird am 19 Januar 2038 erreicht sein. Hier machen einige Computer einen Fehler und werden in eine viel frühere Zeit (13. Dez. 1901) verschoben! Das ist ein riesiges Problem für ältere Computer!

Weitere Auslesemöglichkeiten:
date.getDate -> gibt Tageszahl in Form von 1 – 31 heraus
date.getMonth -> gibt Monat in Form von 0 – 11 heraus
date.getFullYear -> gibt Jahrzahl heraus
date.getDay -> gibt Wochentag in Form von 0 – 6 heraus

V27: Das Switch-Konstrukt in JavaScript

Beispiel einer normalen Auswahl-Konstruktion:

```
> function getMonthName(date) {
  var monthIndex = date.getMonth();
  if (monthIndex === 0) {
  return „Januar“;
  } else if (monthIndex === 1) {
  return „Februar“;
  } else if (monthIndex === 2) {
  return „März“;
  } else if (monthIndex === 3) {
  return „April“;
  usw
  usw
  } else if (monthIndex === 11) {
```

```
    return „Dezember“;
    }
```

Dasselbe als Switch-Konstrukt bringt eine grosse Vereinfachung. Um das hier verwendete „Fall Trough“-Prinzip zu überlisten, wird nach jedem case noch ein break eingefügt, wodurch der Rest der Fälle übersprungen und direkt zum Schleifenende gesprungen werden kann!

```
>   function getMonthName(date) {
    var monthIndex = date.getMonth();
    var monthName;
    switch (monthIndex) {
    case 0: monthName = „Januar“; break;
    case 1: monthName = „Februar“; break;
    case 2: monthName = „März“; break;
    case 3: monthName = „April“; break;
    usw
    usw
    case 11: monthName = „Dezember“; break;
    }
    return monthName
}
```

V28: Wie ermittelt man den Wochentag?

Übung: Erstellen einer Funktion tellDayName() die zu einem in einem Promptfeld eingegebenen Datum den Wochentag in einem Alarmfeld ausgibt (benötigt daher kein Return!).

Neu: - Eingabefenster mit vorgesetztem Datum und -> zweites Argument im Prompt!
- Wochentag aus einem Datum herauslesen -> date.getDay

Dabei Hilfen unter Google, SELFHTML/JavaScript, Selfhtml.wiki, etc. suchen.

Lösung:

```
function tellDayName() {
var dateStr = prompt("Bitte gib ein Datum mit folgendem Format ein: ","jjjj-mm-tt");
var date = parseDateString(dateStr);
var dayName = getDayName(date.getDay());
var monthName = getMonthName(date.getMonth());
alert("Der "+ date.getDate() + ". "+ monthName + " " + date.getFullYear() + " ist ein " + dayName);

function parseDateString(dateStr) {    // dateStr ist Argument das übergeben wird (eff: dateString)
var dateParts = dateString.split("-");    // ergibt ein Array mit Index 0 = Jahr / Index 1 = Monat / Index 2 = Tag
return new Date(dateParts[0], dateParts[1] - 1, dateParts[2]);   // Argumente: Jahr / Monat / Tag
}

function getMonthName(monthIndex) {  // monthIndex ist Argument das übergeben wird (eff: date.getMonth)
return "Januar Februar März April Mai Juni Juli August September Oktober November Dezember".split(" ")
[monthIndex];
}
```

```
function getDayName(dayIndex) {    // dayIndex ist Argument das übergeben wird (eff: date.getDay)
return "Sonntag Montag Dienstag Mittwoch Donnerstag Freitag Samstag".split(" ") [dayIndex];
}

}
```

Bemerkungen:
-> Mit parseDateString() wird der DateString in eine Zahl bzw. hier in ein Array umgewandelt
-> Funktion tellDayName läuft in html-Dok, wenn mit onload aufgerufen

V29: multidimensionale Datenstrukturen

Var cast = ["Anthony Hopkins", "Jodie Foster", "Kasi Lemmons"];

Daraus wird nun ein zweidimensionales Array mit vorerst einem Objekt erstellt:

```
cast = [{
        name: "Anthony Hopkins",
        gender: "male",
        birthday: new Date(1937, 11, 31)
        }];
```

und nun werden 2 weitere Objekte in das zweidimensionale Array „hineingepuscht“:

```
cast.push({
        name: „Jodie Foster“,
        gender: „female“,
        birthday: new Date(1962,10,19)
        })

cast.push({
        name: „Kasi Lemmons“,
        gender: „female“,
        birthday: new Date(1962,1,24)
        })
```

Aus diesem zweidimensionalen Array lassen sich nun interessante Kombinationen heraussuchen:

```
function women(arr) {     // zeigt alle weiblichen Objekte
        for (var resArr = [], i = 0; i< arr.length; i++) {
                if (arr[i].gender === „female“) {
                resArr.push(arr[i]);
                }
         return resArr;
         }
}

function men(arr) {      // zeigt alle männlichen Objekte
      for (var resArr = [], i = 0; i< arr.length; i++) {
                if (arr[i].gender === „male“) {
```

```
                resArr.push(arr[i]);
                }
         return resArr;
        }
}

function oldest(arr) {      // zeigt das älteste Objekt
        for (var oldest = arr[0], i = 1; i < arr.length; i++) {
                if (arr[i].birthday> oldest.birthday) {
                oldest = arr[i];
                }
        return oldest;
        }
}
```

Verallgemeinerung der Suchfunktion für Maximum einer Property:

```
function maximum (arr, prop) {      // zeigt das Maximum der angegebenen Property (z.B. birthday) an
        for (var maximum = arr[0], i = 1; i < arr.length; i++) {
                if (arr[i][prop] > maximum[prop]) {
                maximum = arr[i];
                }
        return maximum;
        }
}
```

Suchfunktion für bestimmten Wert einer Property:

```
function where (arr, prop, val) {      // zeigt den Ort des gesuchten Wertes an
        for (var resArr = [], i = 0; i < arr.length; i++) {
                if (arr[i][prop] === val) {
                resArr.push(arr[i];
                }
        return resArr;
        }
}
```

V30: Erstellen einer Datenbank

Zum Erstellen eines neuen Objekts schreibt man:

```
        new Object ();
```

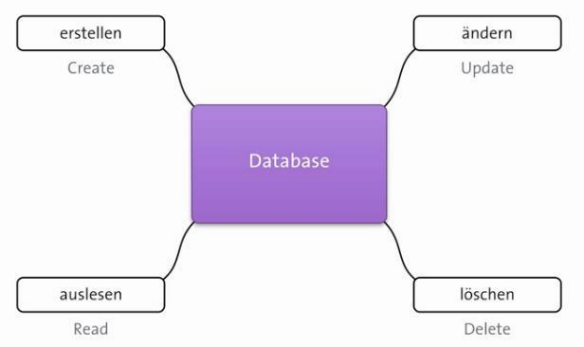
Mittels dem **new**-Operator wird eine Objektvorlage ohne Inhalt erstellt und mit dem Klammerwert versehen (instanziert).

Wir haben überall Vorlagen dazu - Array, Object oder auch Date – wie hier, wenn man ein neues Datum haben will. Da schreibt man auch new Date und mit diesem new-Operator oder mit diesem new-Schlüsselwort wird quasi ein Objekt aus dieser Vorlage heraus erstellt.

```
        new Database();     ->geht natürlich nur wenn bereits eine Database()-Vorlage existiert!
```

var o = **new** Object (); -> Abfrage mit o bzw. o = {} gibt Object{}

Die Funktion Database ist eine solche Vorlage. Was soll unsere Funktion Database nun alles können?



Durch die ersten Buchstaben der grundlegenden englischen Database-Operationen entsteht das sog. CRUD. Die effektiven Funktionen heissen bei Datenbanken insert, where, update und delete. Auf die Ableitung der einzelnen Befehle wird verzichtet, da sehr kompliziert.

var db = **new** Database (); -> Abfrage mit db ergibt *Database {}*

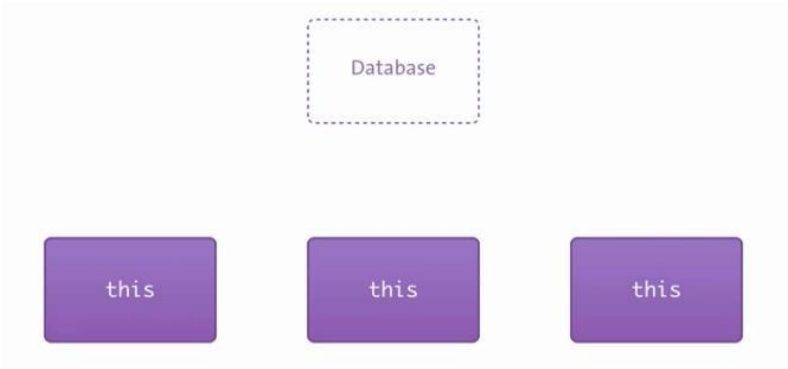
Durch Anfügen von „Methoden“ nach einem Punkt kann die Datenbank mit Funktionen ergänzt werden.

```
db.array = [];   //setzt Array

db.insert = function (obj) {    // fügt die Funktion „Insert“ ein
        // obj ist das Objekt mit seinen Datenbankeintragungen
        db.array.push(obj);   // Funktion zum Einfügen der Daten
} ;
```

this:

this heißt im Deutschen so viel wie “dies” oder “das hier” und damit spricht man die jeweilige Instanz (bzw. das aktuelle Objekt) an.
Wenn Instanzen aus einer Vorlage (Database) erstellt werden, dann ist jedes einzelne Objekt (hier Boxen „this“), das innerhalb des Objektes erstellt wird, mit this ansprechbar:



Database ist Vorlage / this-Boxen sind Instanzen bzw. einzelne Objekte dieser Vorlage.

Innerhalb von Funktionen kann das Objekt, an der die Funktion hängt, per this erreicht werden.
Mit -> this spricht man daher die jeweilige, eigene Instanz (hier Boxen „this“) an. Hier ein Beispiel dazu:

```
function Database () {
	this.database = [];   //“Array“ wird der Instanz resp. Funktion „Database“ beigefügt
	this.insert = function (obj) {  //“Funktion (obj)“ wird der Instanz resp. Funktion „Database“ beigefügt
		this.database.push(obj);
	};
	this.where = function (prop, val) {    //“Funktion (prop, val)“ wird der Instanz „Database“ beigefügt
		for (var resArr = [], i = 0; i < this.database.length; i++ {
			if (this.database[i][prop] === val) {
				resArr.push(this.database[i]);
			}
		}
		return resArr;
	}
	}
```

Abfrage mit db = new Database() zeigt

Database {database: Array[0], insert: function, where: function}

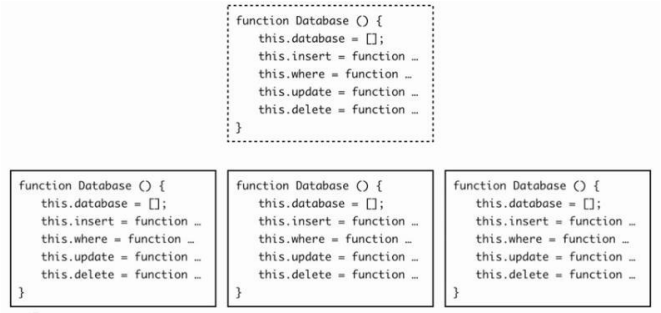
Erweiterte Datenbank mit zusätzlichen Fähigkeiten (Kontrolle, Update,Delete):

```
function Database () {
	this.database = [];
	this.insert = function (obj) {
		this.database.push(obj);
	};
	this.where = function (prop, val) {
		if (!prop) {
			// wenn kein Property eingeführt wurde, wird der
			// Inhalt der gesamten Datenbank zurück gegeben
			return this.database
		}
		for (var resArr = [], i = 0; i < this.database.length; i++ {
			if (this.database[i][prop] === val) {
				resArr.push(this.database[i]);
			}
		}
		return resArr;
	};
	this.update = function (whereProp, whereVal, updateProp, updateVal) {  // Update Funktion
	for (var i = 0; i < this.database.length; i++ {
			if (this.database[i][whereProp] === whereVal) {
				// Objekt gefunden!
				this.database[i][updateProp] = updateVal;
			}
		}
	};
	this.delete = function (prop, val) {    // Lösch Funktion
		if (!prop) {
			// lösche alles
```

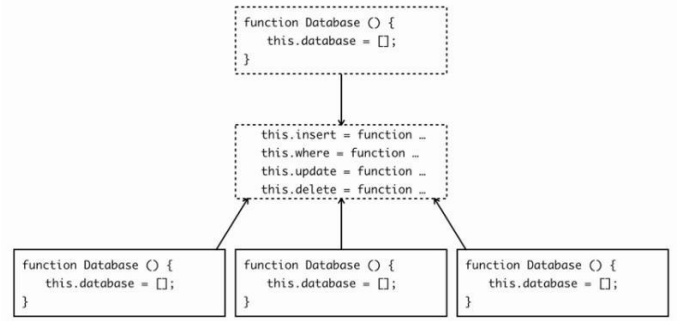
```
                var currentDatabase = this.database;
        this.database = [];
        return currentDatabase;
}
for (var i = 0; i < this.database.length; i++ {
                if (this.database[i][prop] === val) {
                        // Objekt gefunden!
                        return this.database.splice(i, 1);
                }
};
}
```

V31: Prototypen in JavaScript

Beim jedem Ausführen der Konstruktorfunktion (oben) werden alle Objekt-Properties neu erstellt.



Es wäre aber optimaler, wenn die Funktionen ausgelagert werden könnten.



Das geht aber nur bei prototypenbasierten Programmiersprachen wie JavaScript.

Bei einer Prototypenbasierten Programmiersprache gibt es Prototype-Objekte. Wenn man hier ein Objekt aus einer Konstruktorfunktion erzeugt, dann referenziert dieses Objekt das Prototyp-Objekt und all die Funktionen und anderen Werte, die im Prototypeobjekt sind. Sie können nun auch über das erzeugte Objekt erreicht werden. Das tolle daran ist, das man das Prototyp-Objekt jetzt auch noch verändern kann und zum Beispiel eine Funktion hinzufügen kann, wobei diese Funktion dann natürlich auch über das erzeugte Objekt erreichbar wird.

Wie komme ich an das Prototypeobjekt ran? Es geht ganz einfach:
Man nimmt die Konstruktorfunktion - und schreibt dahinter Prototype.

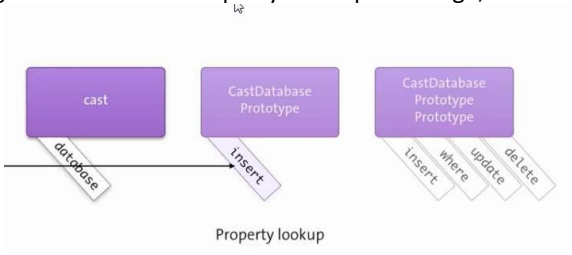
```
Function Database () {  // Konstruktorfunktion
    this.database = [];  // Property für Array
}
Database.prototype = {   // Prototypefunktion für auszulagernde Funktionen
    insert: function … // Prototypeproperty für Insert
    where: function …. // Prototypeproperty für where
    update: function …. // Prototypeproperty für update
    delete: function …. // Prototypeproperty für delete
};
```

Jede Konstruktorfunktion, bzw. jede Funktion hat dadurch eine prototype-Property. Übrigens: Daran dass auch Funktionen properties haben können, sieht man, dass selbst Funktionen in JavaScript Objekte sind.
Also in JavaScript ist alles ein Objekt, was nicht primitive type ist.

Aus dem Konstruktor kreierte neue Objekte erhalten jedesmal neue Properties, da sie kopiert werden. Aus einem Prototype erstellte Objekte erhalten hingegen nicht jedesmal eine neue Property, da sie auf den Prototype referenzieren.

```
function Database () {
  this.database = [];
}
Database.prototype = {
  insert: function …
  where: function …
  update: function …
  delete: function …
};
```

Das Database-Property bleibt eine eigene unveränderte Instanz und die Funktionen werden in ein Database.Prototypen Objekt ausgelagert wo sie gemeinsam benutzt werden können. Dabei können sogar gleiche Funktionen (siehe Insert-Funktion unten) verwendet werden. Dabei kommt die sog. Prototype-Vererbungs-Kette mit dem Property-Lookup zum Zuge, wo immer von Oben nach Unten gesucht wird:



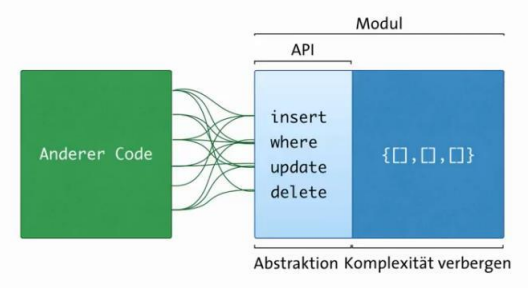
Hier nun als Beispiel das mit Prototype modifizierte Datenbankmodul der letzten Lektion:

```
function Database () {    // Array bleibt weiterhin in Konstruktorfunktion die für alle gleich ist
        this.database = [];
        var index = 0;    //Variable index und neue erweiterte Insert-Funktion / alte bleibt!!
        this.insert = function (obj) {
        obj[„id“] = index++;
        this.database.push(obj);
}
Database.prototype = {    // Funktionen werden neu im Prototype-Objekt erzeugt
        insert: function (obj) {
        this.database.push(obj);
},
        where : function (prop, val) {
```

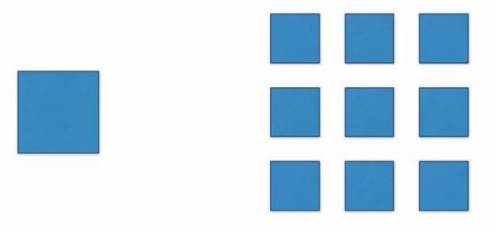
```
        if (!prop) {
                // wenn kein Property eingeführt wurde, wird der
                // Inhalt der gesamten Datenbank zurück gegeben
                return this.database;
        }
        for (var resArr = [], i = 0; i < this.database.length; i++ {
                if (this.database[i][prop] === val) {
                        resArr.push(this.database[i]);
                }
        }
        return resArr;
},
update : function (whereProp, whereVal, updateProp, updateVal) {  // Update Funktion
for (var i = 0; i < this.database.length; i++ {
                if (this.database[i][whereProp] === whereVal) {
                        // Objekt gefunden!
                        this.database[i][updateProp] = updateVal;
                }
        }
},
delete : function (prop, val) {    // Lösch Funktion
        if (!prop) {
                // lösche alles
                var currentDatabase = this.database;
        this.database = [];
        return currentDatabase;
}
for (var i = 0; i < this.database.length; i++ {
                if (this.database[i][prop] === val) {
                        // Objekt gefunden!
                        return this.database.splice(i, 1);
                }
}
}
```

V32: Skalierbarkeit, API und Kapselung

Skalierbarer Code verhält sich möglichst gleich, egal wie viel Daten da sind.



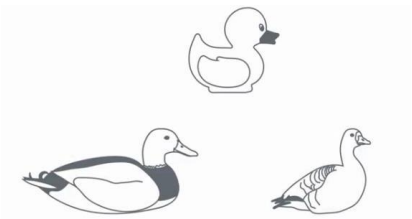
Das Programmierinterface (API -> Application Programming Interface) deckt die Komplexität der Datenbank ab (Kapselung) und kann dabei unter Umständen auch auf mehrere parallele Datenbanken zugreifen.



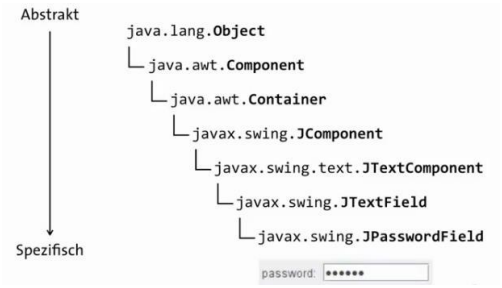
Man sollte daher beim Schreiben von Software solche Aspekte berücksichtigen.

V33: Vererbung in JavaScript

Vererbung ist eine essentielle Sache in der Objektorientierung, wenn wir nochmal unsere Ente als Beispiel nehmen, von der könnten wir erben und dann durch verschiedene Veränderungen andere Enten draus machen.



Die Richtung ist dabei immer von Abstrakt zu Spezifisch: Man fängt bei einer sehr abstrakten Grundlage an und von da aus kann man in jede beliebige Richtung gehen und vererben und vererben und dabei immer spezifischer und spezifischer werden.



Das spart nicht nur Schreibarbeit sondern erleichter zukünftige Veränderungen, es lagert auch Komplexität aus. Deswegen ist Vererbung so essentiell.

V34: typeof, instanceof & Null

Mit dem **typeof-Operator** kann man den Datentyp eines Wertes herausfinden. typeof und dann irgendein beliebiger Wert ergibt den Datentyp in Stringform.
D.h. zum Beispiel typeof von irgendeiner Zahl, wie z.B. 42, typeof 42 würde den String “number” zurückgeben. So ist das bei allen Datentypen, bei allen Datentypen wird der Name des Datentyps als String kleingeschrieben zurückgegeben, außer bei einem Datentyp und das ist Null.

```
typeof  true            => "boolean"
typeof  42              => "number"
typeof  "Text"          => "string"
typeof  function () {}  => "function"
typeof  {}              => "object"
typeof  undefined       => "undefined"
typeof  null            => "object"
```

Der **Datentyp Null** hat, genau wie „undefined“ auch, nur einen einzigen Wert und zwar: null. Wie undefined auch, ist auch null ein falsy-Wert. Bei Zahlen ist ja nur 0 falsy, bei Strings sind nur leere Strings falsy, aber bei undefined und null haben die Datentypen ja nur einen einzigen Wert und dieser Wert ist falsy.

Mit **instanceof** kann man überprüfen, ob ein Objekt die Instanz von irgendwas ist. Also zum Beispiel: Wenn ich eine Instanz von CastDatabase mache, also mit new CastDatabase das instanziiere und das dann in der Variable cast speichere, dann kann ich fragen: Ist dieses Objekt in der Variablen cast eine Instanz von CastDatabase? Ja, das ist es, true.

```
var cast = new CastDatabase();
cast instanceof CastDatabase  => true
cast instanceof Object        => true
var obj = {}; // dasselbe wie new Object()
obj instanceof Object         => true
obj = []; // typeof [] => "object"
obj instanceof Object         => true
```

CastDatabase erbt allerdings von Database. Würde ich jetzt fragen “Ist cast instanceof Database?”, würde auch da true rauskommen. Und Database wiederum erbt letztendlich von Object, weil alle Objekte von Object erben, Object ist quasi immer ganz zum Schluss in der Prototype-Chain, und deswegen kommt auch true raus, wenn ich frage: Ist cast instanceof Object? Ja, das ist es.
Dasselbe passiert, wenn ich so ein einfaches Objekt erstellen würde, das ja dasselbe ist wie new Object, also eine Instanz von Object. Wenn ich dann frage “Ist obj (also diese Variable obj mit dem einfachen Objekt) instanceof Object?” kommt auch da true raus.
Dasselbe passiert auch mit Arrays. Denn Arrays sind ja auch kein eigener Datentyp, sondern Arrays sind Objekte. Und wenn wir da fragen “Ist das Array instanceof Object?”, dann kommt auch da true raus. Das findet man bei allen Objekten. Alle Objekte erben irgendwann von Object und deswegen sind alle Objekte instanceof Object.

Jetzt schauen wir uns mal null an. Wo kommt null in JavaScript vor? Im Grunde nie, es sei denn, es wird explizit etwas auf null gesetzt. Die einzige Ausnahme ist, dass null auch das Ende einer jeden Prototyp-Kette markiert, und das ergibt auch Sinn, weil die Prototyp-Objekte alle Objekte sind und dann wäre das letzte Objekt quasi ein Objekt laut Typ, das aber gleichzeitig kein Objekt ist. Aber das ist vorrangig Definitionssache und ich weiß nicht, ob das wirklich die Präsenz eines eigenen Datentyps legitimiert.

undefined	null
• Beim Zugriff auf eine deklarierte, aber nicht definierte Variable • Der implizierte Rückgabewert, wenn eine Funktion nichts zurückgibt • Lookups nicht-existierender Properties • Parameter, die einer Funktion nicht übergeben wurden • Alles, was auf undefined gesetzt wurde	• Alles, was auf null gesetzt wurde • Das Ende einer Prototyp-Kette

Unterm Strich muss ich sagen: undefined ist total nützlich, das kommt überall drin vor, das ist ein wichtiger Teil in JavaScript, aber null ist in JavaScript völlig überflüssig. In anderen Sprachen macht das Sinn. In anderen

Sprachen hat man kein undefined und da gibt es aber auch ein null und da übernimmt dieses null dann die Aufgaben, die in JavaScript das undefined übernimmt. Aber in JavaScript ist null meiner Meinung nach und meiner Erfahrung nach völlig überflüssig und man braucht es überhaupt nicht.

V35: arguments, apply & call und der Debugger

Arguments heißen Parameter, die man einer Funktion übergibt.

```
functionName(param1, param2, param3, …);
             ^^^^^^^^^^^^^^^^^^^^^^^^^
                      arguments
```

Arguments stehen auch innerhalb einer Funktion als Objekt zur Verfügung. Man schreibt einfach arguments und da hat man alle Argumente, die einer Funktion übergeben wurden.

```
function functionName(param1, param2, param3, …) {
   arguments // Argumente der Funktion
}
```

Arguments ist ein Objekt und das hat auch Properties. arguments hat zum Beispiel das length-Property, um die Anzahl der übergebenen Argumente auszulesen.

```
function functionName(param1, param2, param3, …) {
   arguments // Argumente der Funktion
   arguments.length // Anzahl der Argumente
   arguments[0] // Zugriff auf einzelne Argumente
   arguments instanceof Array // => false
}
```

Um auf ein bestimmtes Argument zuzugreifen, schreibt man einfach in eckigen Klammern den Index dahinter.

Und nun zu den **Funktionen apply und call**. apply heißt auf Deutsch “anwenden” oder “anwenden auf” und mit apply kann man eine Funktion auf ein beliebiges Objekt anwenden. Dann ist this innerhalb der Funktion auf dieses andere Objekt gesetzt:

```
functionName(arg1, arg2, …);
functionName.apply(obj, [arg1, arg2, …]);
functionName.call(obj, arg1, arg2, …);
```

Und der Grund, warum ich euch das erzähle, ist folgendes: Man übergibt der apply-Methode mit dem Objekt ja schon einen Parameter. Aber was, wenn ich eine Funktion auch mit anderen Parametern aufrufen will? Wie übergebe ich diese Parameter, wenn ich apply nutze? Ich übergebe diese Parameter in Form einer Liste! Also zum Beispiel in Form eines Arrays oder in Form eines arguments-Objekts!

Auch bei call, wie bei apply, ist der erste Parameter das Objekt, auf das sich this beziehen soll. Aber im Gegensatz zu apply übergibt man hier keine Liste, sondern die Parameter selbst, so wie man die auch übergeben würde, wenn man die Funktion normal aufrufen würde.

“Debug” heißt zu Deutsch so viel wie “Fehler beseitigen” und ein “**debugger**” ist ein Modus, der einem dabei hilft, diese Fehler aufzuspüren. Und wenn ein Browser einen debugger-Modus besitzt, dann kann man den in JavaScript aufrufen, indem man das debugger-Statement benutzt.
Der debugger-Modus ist unglaublich nützlich, aber auch unglaublich komplex und unterscheidet sich von Browser zu Browser

V36: Wofür ist die for …in-Schleife?

Die for…in-Schleife scheint perfekt zu sein für Arrays. Man braucht nicht viel machen und sie geht über jedes einzelne Element des Arrays. Aber es ist auch mit Abstand die langsamste von allen Schleifen!

Die for…in-Schleife iteriert über jedes Property und guckt dann aber erstmal, ob es überhaupt über dieses Property iterieren darf oder nicht. Außerdem guckt euch mal „cast“ an: Hier haben wir das Property database, und wenn wir dann uns das prototype angucken, haben wir hier das Property database und das Property insert, und wenn wir uns dann das Prototype angucken, haben wir hier die Properties delete, update und where. Die for…in-Schleife hat aber all diese Properties ausgegeben, d.h. die for…in-Schleife geht sogar dann auch noch durch die Prototype-chain durch und iteriert dann auch noch über sämtliche geerbten Properties.
Das alles fällt bei den anderen Schleifen weg! Also kein Wunder, dass die for…in-Schleife die langsamste Schleife überhaupt ist.

Spez 1: Ergänzungen

1.1: Images:

Allgemeines zur Verwendung

Mit dem Objekt `images`, das in der JavaScript-Objekthierarchie unterhalb des `document`-Objekts liegt, haben Sie Zugriff auf alle `Grafiken`, die in einer HTML-Datei definiert sind. Dabei können Sie auch vorhandene Grafiken dynamisch durch andere ersetzen.

Ein neues Grafik-Objekt wird automatisch erzeugt, wenn der Web-Browser eine `Grafik` in der HTML-Datei vorfindet.

Es stehen folgende Arten zur Verfügung um mit JavaScript eine bestimmte Grafik anzusprechen:

*mit einer **Indexnummer (#):***

```
document.images[#].Eigenschaft
document.images[#].Methode()

Höhe des ersten Bildes = document.images[0].height;
```

Bei Verwendung von Indexnummern geben Sie `document.images` an und dahinter in eckigen Klammern, die wievielte Grafik in der Datei Sie meinen. Jede Grafik, die in HTML mit dem `<img>`-Tag notiert wurde, zählt. Beachten Sie, dass der Zähler bei 0 beginnt, d.h. die erste Grafik sprechen Sie mit `images[0]` an, die zweite Grafik mit `images[1]` usw. Beim Zählen gilt die Reihenfolge, in der die `<img>`-Tags in der Datei notiert sind.

*mit dem **Namen der Grafik:***

```
document.images.Bildname.Eigenschaft
document.images.Bildname.Methode()

Höhe des Bildes "Portrait"= document.images["Portrait"].height;
```

Dabei wird die Grafik als Unterobjekt von `document.images` angesprochen. Notieren Sie `document.images`, dahinter einen Punkt zum Ansprechen des Unterobjekts und schließlich den Namen, den Sie bei der Definition der Grafik im einleitenden `<img>`-Tag im Attribut `name` angegeben haben.

*mit dem **Namen der Grafik als Indexnamen:***

```
document.images["Bildname"].Eigenschaft
document.images["Bildname"].Methode()

Höhe = document.images["Portrait"].height;
```

Diese Art ist eine zu Schema 2 äquivalente Schreibweise, denn in JavaScript ist das Ansprechen von Unterobjekten über `objekt.unterobjekt` gleichwertig zu `objekt["unterobjekt"]`. Dabei notieren Sie wie beim Ansprechen mit Indexnummer hinter `document.images` eckige Klammern. Innerhalb der eckigen Klammern notieren Sie in Anführungszeichen den Namen, den Sie bei der Definition der Grafik im einleitenden `<img>`-Tag im Attribut `name` angegeben haben. Diese Schreibweise ist vor allem zum Zugriff auf Grafiken nützlich, deren Namen Sonderzeichen enthalten, welche den Zugriff nach Schema 2 unmöglich machen. Sie können zwischen den eckigen Klammern auch eine String-Variable notieren, die den Grafiknamen enthält.

*mit dem **Namen der Grafik direkt:***

```
document.Bildname.Eigenschaft
document.Bildname.Methode()

Höhe = document.Portrait.height;
```

Dabei geben Sie mit document.Bildname den Namen an, den Sie bei der Definition der Grafik im einleitenden <img>-Tag im Attribut name angegeben haben.

1.2: Element „innerHTML“:

Allgemeines zur Verwendung

Speichert den Inhalt inkl. interne HTML-Tags eines HTML-Elements (z.B. h1#Test bei Beispiel 2). Wenn Sie beim dynamischen Ändern des gespeicherten Inhalts HTML-Tags notieren, werden diese bei der Aktualisierung des Elementinhalts interpretiert.

Die Eigenschaft innerHTML sollten Sie nicht direkt beim Einlesen der HTML-Datei anwenden, sondern immer erst abhängig von Aktionen wie Verweisklicks oder Button-Klicks oder mit einem Seite setTimeout() von einigen Sekunden davor. Bei Anwendung direkt beim Einlesen der Datei meldet der Internet Explorer 4.0 einen Laufzeitfehler.

Der Internet Explorer ist aus irgendeinem Grund nicht in der Lage, innerHTML auf die HTML-Elemente table;, thead, tbody, tfoot und tr schreibend anzuwenden. Man kann Tabellen also nicht über innerHTML verändern, sondern ist gezwungen, entweder ein die Tabelle einschließendes Element mit einer kompletten Tabelle neu zu schreiben, oder z.B. mit den DOM-Methoden zu arbeiten.

Beispiel 1 (reine HTML-Anwendung)

```
<html><head><title>Test</title>
</head><body>
<h1 id="Test"
onmouseover="this.innerHTML = 'Sehen Sie?' "
onmouseout="this.innerHTML = 'Ich bin dynamisch' ">Ich bin dynamisch</h1>
</body></html>
```

In dem Beispiel wird eine Überschrift erster Ordnung definiert. Innerhalb der Überschrift sind die Event-Handler onmouseover und onmouseout notiert. Der Event-Handler onmouseover tritt in

Aktion, wenn der Anwender die Maus in den Anzeigebereich der Überschrift bewegt, und onmouseout wird aktiv, wenn die Maus wieder ausserhalb dem Anzeigebereich liegt. Mit Hilfe von this nehmen Sie Bezug auf das aktuelle Objekt (h1#Test) und können mittels der Eigenschaft innerHTML mit jedem Aktivwerden eines der beiden Event-Handler den Text der Überschrift dynamisch austauschen. Bei onmouseover wird ein anderer Text angezeigt, bei onmouseout wieder der ursprüngliche Text.

Das Beispiel zeigt, wie Event-Handler auch in HTML-Tags funktionieren, bei denen das bislang nicht möglich war. Das Beispiel funktioniert mit dem Internet Explorer ab Version 4.x und im Netscape Navigator ab Version 6, welche die Event-Handler nach HTML 4.0 weitgehend interpretieren.

Auch Opera ab Version 5 interpretiert den Event-Handler nach HTML 4.0. Er kennt jedoch nicht die Eigenschaft innerHTML.

Beispiel 2 (mit JavaScript)

```
<html><head><title>Test</title>
<script type="text/javascript">
var Neu = "neuer <b>fetter<\/b> Text";    // -> neuer Text mit HTML-Tag
function Aendern () {
  document.all.meinAbsatz.innerHTML = Neu;
}
</script>
</head><body>
<p id="meinAbsatz">Text</p>     // -> Ursprungstext
<a href="javascript:Aendern()">Anderer Text</a>
</body></html>
```

Das Beispiel enthält einen Textabsatz und einen Verweis. Beim Anklicken des Verweises wird die Funktion Aendern() aufgerufen. Diese Funktion weist dem Absatz mit der id="meinAbsatz" für die Eigenschaft innerHTML den Wert der zuvor definierten Variablen Neu zu. Der Inhalt des Absatzes ändert sich dann dynamisch und berücksichtigt dabei auch die HTML-Formatierung ... beim neuen Inhalt des Elements.

Spez 2: Einfügen von JavaScripts auf Webseiten

JavaScript Code direkt im HTML-Body:

```
<body>
<script  type="text/javascript">
<!—versteckt Script in ältere Browsern
Code
// den Rest ab Ende dieser Zeile nicht mehr verstecken -->
</script>
</body>
```

Einfügen des JavaScript Codes im HTML-Head:

```
<head>
<script type="text/javascript">Code1</script>
<script type="text/javascript">Code2</script>
etc.
</head>
```

Auslagern des JavaScripts Codes auf externen Speicherort:

In HTML-Head: <script type="text/javascript" src="myscripts.js"></script> auf Ebene html abgelegt
<script type="text/javascript" src="myscripts.js" charset="UTF-8"></script> für UTF8-Scripte
<script type="text/javascript" src="scripts/myscripts.js"></script> in Verzeichnis „scripts“

Ablegen des Scripts: Die Datei ="myscripts.js" kann durch Umbenennen einer Textdatei erstellt werden in der das JavaScript (ohne Script Tags!) eingefügt wurde. ***In diesem Script können mehrere Funktionen hintereinander aufgelistet sein. Es können aber auch alle Funktionen in einzelnen separaten Scripts ausgelagert werden***.

Spez 3: Aktivieren von JS im Body einzelner Seiten

Beim Öffnen der Seite: onload="myFunction1(),myFunction2(),myFunctionX()";

Durch Einschalten einer Funktion:
am Ende des JavaScripts: window.onload = myFunction1,myFunction2,myFunctionX;
Die Funktion "myFunction()" muss im JavaScript abgelegt sein!

Durch Einschalten einer Funktion mit Knopf: <button onclick="myFunction()">Try it</button>
Die Funktion "myFunction()" muss im JavaScript abgelegt sein!

Durch Einschalten einer Funktion mit Klick: Try it
Die Funktion "myFunction()" muss im JavaScript abgelegt sein!

Durch Einschalten einer Funktion mit Bild: <img src="bild.gif" onClick="show ()"
Die Funktion "show ()" muss im JavaScript abgelegt sein!

Durch Einschalten einer Funktion mit MouseOver: <img src="bild.gif" onMouseover="show ()"
Die Funktion "show ()" muss im JavaScript abgelegt sein!

Durch Einschalten einer Funktion mit MouseOut: <img src="bild.gif" onMouseout="halt()"
Die Funktion "halt()" muss im JavaScript abgelegt sein!

Spez 4: JavaScript-Einfügungen auf CMBasic-Seiten

Es gibt folgende **Möglichkeiten zum Einfügen eines JavaScript** gesteuerten Programms auf CMBasic Seiten:

Methode 1: Das **JavaScript wird unmittelbar vor dem Programm** im body-Teil der entsprechenden Seite mittels eines Skript-Einschubes eingesetzt und das dazugehörige html-Programm mit dem Funktionsaufruf wird unmittelbar darunter eingetragen. Start der Programme durch Aufruf der Funktion mittels einem Button oder einem Bild im Eventhandler möglich.

Bem: Die einzeln eingefügten Scripts werden alle der Reihe nach in einen gemeinsamen Script-Behälter abgelegt. Es muss daher unbedingt darauf geachtet werden, dass **für Variablen und Funktionen unterschiedliche Bezeichnungen verwendet** werden! Kleine Programmfehler wirken sich auf alle anderen Programme aus!

CMBasic-Datenbankeinträge erlauben leider nur **maximal 2400 Zeilen**. Grössere Scripts (wie z.B. Logo-Slideshow) haben aber mehr als das Dreifache davon und können daher nicht eingefügt werden. (unklar woher ich das habe)

Methode 2: Die einzelnen JavaScripts werden in einzelne oder eine gemeinsame JavaScript-Datei verschoben welche (z.B. nach Skript/js/ xyz.js) ausgelagert werden. Im Header der index.php wird diese Datei wieder eingebunden, so dass sie von allen Seiten her abrufbar wird. Die dazugehörigen html-Programme mit dem Funktionsaufruf werden auf der entsprechenden Seite im body-Teil eingetragen. Start der Programme durch Aufruf der Funktion mittels einem Button oder einem Bild im Eventhandler möglich.

Bem: Die einzeln eingefügten Scripts werden alle der Reihe nach in einen gemeinsamen Script-Behälter abgelegt. Es muss daher unbedingt darauf geachtet werden, dass **für Variablen und Funktionen unterschiedliche Bezeichnungen verwendet** werden! Kleine Programmfehler wirken sich auf alle anderen Programme aus!

Methode 3: Die nötigen **JavaScript-Funktionen** werden **zusammen mit dem Programm in einem Symbolbefehl {Funktion} verpackt**. Die dazu zu entwickelnde Datei bot_xyz.php wird im Verzeichnis Minibot abgelegt und die Ansteuerung muss im config.inc.php und im contentshow.inc.php eingerichtet werden. In diesem Fall gibt es keine Restriktionen bez. Länge des JavaScripts dafür aber eine Neuentwicklung!

Wichtige Besonderheiten bei JavaScript – Einfügungen im CMBasic:

- Start mit <body onload=…> ist bei CMBasic nicht realisierbar. Im CMBasic müssen Programme, die selbständig ablaufen (z.B. Digitaluhr) am Ende des jeweiligen JavaScripts mittels window.onload = function1,function2,functionX; oder mit einem Button gestartet werden.
- Der CMB-html-Editor optimiert und verändert die Html Eingaben vor dem Einfügen in die Datenbank teilweise so stark, dass Programme nicht mehr funktionsfähig sind. So wird z.B. im Img-Tags wird das ***Attribut name="xyz"*** entfernt / das Tag ****** wird durch ***<b>*** ersetzt / das Tag ***
*** wird zeitweise zu <p> umgewandelt / die Script-Öffnungsausdrücke und Script-Abschlussausdrücke bekommen Zusätze / fehlende Abschlussausdrücke werden eingefügt / Klammern werden verändert / Attribute werden weggelassen, etc. -> **Hier ist besondere Vorsicht geboten!**

- **Abhilfe**: Seiten mit **Scripte sollten daher nur direkt im Content bearbeitet** werden oder nur ausnahmsweise mit einem Häklein bei der rechts von "Bearbeiten" liegenden "HTML-Box" geöffnet und bearbeitet werden (nur so kann die Editorfunktion umgangen werden). Vorsicht: Die Anzeige im Editorfeld kann bereits vor dem Bearbeiten ohne HTML-Box massiv verändert angezeigt worden sein, da der fehlerbehaftete Editor in diesem Fall auch zur Erstellung der Anzeige verwendet wird!
- Seiten mit mehreren JS-Scripts können beim CMB nicht "body onload" gestartet werden! **Auf Seiten mit mehreren JS-Scripts muss ein gemeinsamer Script-Behälter (Eventhandler) verwendet werden, der die verschiedenen Scrips zusammenfasst und gemeinsam mit "windows.onload" startet**. Auch Programme die selbständig (autonom) ablaufen müssen in diesen Evenhandler eingefügt werden, ausser sie werden mit einem CMB-Systembefehl eingefügt.
- Die einzeln eingefügten Scripts können auch alle der Reihe nach in einen gemeinsamen Script-Behälter abgelegt und von dort gestartet werden. Hier muss **unbedingt darauf geachtet werden, dass die Variablen und Funktionen unterschiedliche Bezeichnungen haben!**
- Bilder und Knöpfe können in den Scripten mit einer ID (z.B. Knopf ID=xy) versehen werden. Das Verwenden von Bildernamen ist nicht empfohlen, da sie im Editor oft herausgelöscht werden!
- Ansteuerung bzw. Bildaufrufe mit document.images auf Seiten mit mehr als einem Bild gehen nur, wenn sich die Bilder bez. Namen oder ID unterscheiden. Beispiele:
  document.images[„one“].src (one ist Bildbezeichnung) oder
  document.one.src (one ist Bildname) oder
  document.images[0] (0 ist Bildnummer)
  Mit Bildnummern ist‘s heikel einzurichten, mit unterschiedlichen Bildnamen geht’s einfacher und mit Bild-IDs ist sicherer.

Spez 5: Spezielle Funktionen

5.1: Firefox / Einstellungen:

Anzeigen von Firefox-Einstellungen: Eingabe von “about:config” in der Adressleiste.

5.2: Firefox / Webkonsole:

Verwenden der Web-Konsole (Extras> Web-Entwickler > Web-Konsole) im Firefox zur Eingabe von JavaScript-Codes bzw. Befehlen und zur Untersuchung einer geöffneten Website.

Spez 6: Einfache JavaScript- Beispiele

6.1 Pseudopasswort mit JavaScript

Funktions-Prinzip:

Wandelt ein in ein Formularfeld eingegebenes Passwort in Passwort.html um und springt auf diese Passwort.html Datei welche auf der selben Hierarchie-Ebene liegen muss. Muss um eine Hierarchie-stufe zurückgesprungen werden können, so ist die Weiterleitanweisung wie folgt (blauer Eintrag) zu ändern:

```
document.forms[0].action="../"+document.forms[0].element[0].value +".html" //->Weiterleitung
```

JavaScript im Head:

```
<script type="text/javascript">
<!--                    //-> wird bei alten Browsern bis // -->  ignoriert
function codeTyp()      //-> Funktionsname
{                                                   //-> Funktionsbeginn
if (document.forms[0].elements[0].value=="")        //-> Bedingung wenn Wert leer
      {
return false;                                       //-> Reaktion: es wird nichts ausgelöst
      }
document.forms[0].action = document.forms[0].elements[0].value +".html"      //-> Adresserstellung
return true;                                        //->springt zu eben errechneter Adresse
}
//-->                   //-> Aufhebung des Ignorieren für ältere Browser am Ende der Zeile
</script>
```

Aufruf des JavaScripts vom Body:

```
<form name="Passwort" action ="" onSubmit ="return codeTyp();" target="_top"><b>Insider:</b>
      //-> erstellt leeres Formular „Passwort“ und schaltet bei Senden die Funktion "return codeTyp();" ein
<br><input type=password size=6 name="Eingabe"></form>
      //-> Definiert Passwortfeld und schliesst das Formular
```

6.2 Einfügen von SPAM-geschützten Mailadressen mit JS

Funktions-Prinzip:

Setzt html-TAG (a href=“ ….“.>) mittels JavaScript zusammen und kann daher von aussen nicht abgefragt werden. Zeigt aber Mailadresse auf Seite an.

JavaScript:

```
<span style="font-size: 11px">
<script type="text/javascript">

<!--
        var name = "praesident";
```

```
        var domain = "h-boot.ch";
        document.write('<a href=\"mailto:' + name + '@' + domain + '\">');
        document.write(name + '@' + domain + '</a>');
        // -->
</script>
</span>
```

Bem. Die erste Write-Anweisung enthält ein HTML-TAG (Link) das auf der Seite nicht angezeigt wird. Der letzte Teilstring + '\" kann nicht weggelassen werden, sonst gäbe es Probleme mit dem ersten \". Mit " " oder ' ' wird ein String abgegrenzt; die beiden Zeichen haben eine identische Funktion. Damit der Browser keine Probleme bekommt, werden die Zeichen paarweise von aussen nach innen verwendet. Wenn innerhalb des STRING (wie im Beispiel) noch einmal zwei Zeichen " enthalten sind die ebenfalls einen String abschliessen bekommt der Write-Befehl ein Problem. Damit write den STRING richtig interpretieren kann, wird vor den beiden " jeweils ein \ eingefügt. Diese Backslashes und die html-Tags werden im Resultat nicht angezeigt.

Die zweite Write-Anweisung wird hingegen auf der Seite angezeigt.

Aufruf des JavaScripts im Body:

Dieses Script wird normalerweise direkt im Body verwendet, da Variablen immer anders sind.

6.3 Einfügen einer „Zurück-Schaltfläche“ mit JavaScript

Unter body einfügen:

```
<form>                                                          //-> Form öffnen
<input type="button" value="zurück" onClick="parent.history.back()  //-> Definition des Rücksprungs
</form>                                                         //->schliessen des Formulars
```

6.4 Einfügen einer einfachen „Datumanzeige“ mit JavaScript

Funktions-Prinzip:

Setzt das aktuelle Datum auf die Seite.

JavaScript:

```
<!doctype html>
<html>
```

```
<head>
<meta charset="utf-8">
<title>Datumanzeige it noscript-Anzeige</title>
<script> function Zeit() {
var Jetzt = new Date();
var Tag = Jetzt.getDate();
var Monat = Jetzt.getMonth() + 1;
var Jahr = Jetzt.getYear(); if (Jahr < 999) {Jahr += 1900};
var Vortag = ((Tag < 10) ? "0" : "");
var Vormon = ((Monat < 10) ? ".0" : ".");
var Datum = Vortag + Tag + Vormon + Monat + "." + Jahr;
document.write("<h1>" + Datum + "<\/h1>") }
</script>
</head>

<body>
<script> Zeit(); </script>
<noscript> <h1>In Ihrem Browser ist JavaScript deaktiviert.</h1>
<p>Im SELFHTML-Wiki erfahren Sie, <a
href="http://wiki.selfhtml.org/wiki/FAQ/JavaScript_aktivieren"> wie Sie
JavaScript in Ihrem Browser aktivieren können.</a></p> </noscript> <p>Aktuelle Nachrichten …</p>
</body>

</html>
```

6.5 Einfügen einer digitalen Uhr mit JavaScript

Funktions-Prinzip:

Zeigt eine 6-stellige Digitaluhr welche sich entweder durch „onLoad“ oder durch ein „Start-Button“ starten lässt.

JavaScript:

```
<script type="text/javascript">
<!--
// Digitaluhr
img = new Array()
for(var i=0; i <= 14; i++) {
      img[i] = new Image()
}
img[1].src = "images3/dg0.gif"
img[2].src = "images3/dg1.gif"
```

```
img[3].src = "images3/dg2.gif"
img[4].src = "images3/dg3.gif"
img[5].src = "images3/dg4.gif"
img[6].src = "images3/dg5.gif"
img[7].src = "images3/dg6.gif"
img[8].src = "images3/dg7.gif"
img[9].src = "images3/dg8.gif"
img[10].src = "images3/dg9.gif"
img[11].src = "images3/dgon.gif"
img[12].src = "images3/dgoff.gif"
img[13].src = "images3/dgam.gif"
img[14].src = "images3/dgpm.gif"
var base = "dg";
var stellung = false;
function stop() {
    document.one.src = "images3/space2.gif";
    document.two.src = "images3/space2.gif";
    document.three.src = "images3/space2.gif";
    document.four.src = "images3/space2.gif";
    document.five.src = "images3/space2.gif";
    document.six.src = "images3/space2.gif";
    stellung = false;
}
function pause() {
    stellung = false
}
function go() {
    stellung = true
    start()
}
function start() {
    if(stellung == true) {
        var now = new Date();
        var hours = now.getHours();
        var ampm = (hours < 12) ? "am" : "pm";
        hours = (hours > 12) ? (hours - 12) + "" : hours + ""
        hours = (hours == "0") ? "12" : hours
        hours = (hours < 10) ? "0" + hours : hours + ""
        var minutes = now.getMinutes();
        minutes = (minutes < 10) ? "0" + minutes : minutes + ""
        var seconds = now.getSeconds();
        seconds = (seconds < 10) ? "0" + seconds : seconds + ""
        document.one.src = (hours.charAt(0)=="0") ? "images3/space2.gif" : add(hours.charAt(0))
        document.two.src = add(hours.charAt(1))
```

```
            document.three.src = (now.getSeconds() % 2) ? add("on") : add("off")
            document.four.src = add(minutes.charAt(0))
            document.five.src = add(minutes.charAt(1))
            document.six.src = add(ampm)
            setTimeout("start()",1000)
        }
}
function add(it) {
        return "images3/dg" + it + ".gif";
}
// -->
</script>
<h1>Uhr-Objekt</h1>
<p>
<img src="images3/space2.gif" alt="one" width="16" height="21" name="one" />
<img src="images3/space2.gif" alt="two" width="16" height="21" name="two" />
<img src="images3/space2.gif" alt="three" width="16" height="21" name="three"  />
<img src="images3/space2.gif" alt="four" width="16" height="21" name="four"  />
<img src="images3/space2.gif" alt="five" width="16" height="21" name="five"  />
<img src="images3/space2.gif" alt="six" width="16" height="21" name="six"  />
</p>
<p><button onclick="go()">Start</button><button onclick="stop()">Stopp</button><br />
```

(Bzw. bei Start der Seite: <body onLoad="go()" > geht aber nur, wenn als einziges Programm auf der Seite)

Bemerkung: Die Verwendung von Bilder-Id's wäre besser und Sicherer

6.6 Einfügen eines „Begrüssungstextes" mit JavaScript

Funktions-Prinzip:

Fragt nach Name. Wenn Name richtig ist, wird ein guter Tag gewünscht und das Programm läuft weiter. Sonst kommt eine Folge von Komplimenten (wie z.B. xy ist rrossartig).

JavaScript im Head:

```
<script type="text/javascript">
<!-- Verstecken für ältere Browser          //-> wird bei alten Browsern bis //-->  ignoriert
function Test () {                          //-> Funktionsbeginn
var name;
name=prompt("wie ist dein Name?","");       //-> eingegebener Text wird var name zugeordnet
if (name = "Walo Zach" || name = "Walo" || name = "Zach")  //-> verschiedene if-Bedingungen
{ document.write("<BR> Häb e guete Tag!");   //-> Nach wagrechter Linie wird Text geschrieben
}
```

```
else {
 alert ("Was für ein schöner Name, " + name);  //-> erste alert-box
 alert ("Du bist eine tolle Person, " + name);                                    //-> zweite alert-box
 alert ("Du bist =überhaupt die grossartigste Person, " + name);                  //-> dritte alert-box
 alert ("Wir alle lieben Dich, " + name);                                         //-> vierte alert-box
 alert ("Dein Charme und deine Klugheit sind einfach unerreicht, " + name); //-> fünfte alert-box
 alert (name + ", jeder auf diesem Planeten schätzt dich!");                      //-> sechste alert-box
}
// -- Rest nicht mehr verstecken -->
</script>
```

Aufruf des JavaScripts im Body:

Aktivierung mit einem Button <button onclick="Test ()">
(Bzw. bei Start der Seite: < body onLoad="Test ()") > geht aber nur, wenn als einziges Programm auf der Seite)

Aufruf beim Laden der Seite mit <body onLoad="Test ()"> oder

6.7 Einfügen einer „Scherzfrage“ mit JavaScript

Funktions-Prinzip:

Stellt eine Frage und beurteilt die gegebene Antwort. Gibt antwortabhängigen Kommentar.

JavaScript im Head:

```
<script type="text/javascript">              //-> öffnen des JavaScripts
<!-- Verstecken für ältere Browser          //-> wird bei alten Browsern bis // -->  ignoriert
function Scherzfrage () {                    //-> Funktionsbeginn
Antwort = prompt ("Wie viele Räder hat ein Auto?");  //-> eingeg. Text wird Antwort zugeordnet
if (Antwort == 4) {   //-> erste Ueberprüfung der Antwort
        alert ("Das ist leider falsch, denn Sie haben" +      //-> Falschmeldung mit alert-box
        " das Lenkrad vergessen.");
        }
if (Antwort == 5) {      //-> zweite Ueberprüfung der Antwort
        alert ("Das ist genau richtig. \n" +                  //-> Richtigmeldung mit alert-box
        "Herzlichen Glückwunsch.")
        }
if ((Antwort !=4) && (Antwort !=5)) {       //-> wenn beide Antworten falsch sind
        confirm ("Leider nein. klicken Sie\n" + "auf [ OK ]. wenn Sie es erneut probieren" +
        "wollen.")  //-> Anfrage erneuter Versuch mit confirm-box
        {
Scherzfrage ();              //-> Rücksprung zur Funktion Scherzfrage
}
```

```
}
}
// -- Rest nicht mehr verstecken -->     //-> wird bei alten Browsern bis Ende Zeile ignoriert
</script>                                 //-> schliesst das JavaScript
```

Aufruf des JavaScripts im Body:

Mit einem Button <button onclick="Scherzfrage ()">
(Bzw. bei Start der Seite: <body onLoad=“ Scherzfrage () > geht aber nur, wenn als einziges Programm auf der Seite)

6.8 Mausgesteuerte Lupenfunktion mit JavaScript

Funktions-Prinzip:

Vergrössert ein Bild, wenn man mit der Maus darüberfährt.

JavaScript im Head:

```
<script type="text/javascript">
var mBa1 = new Image();
mBa1.src = "images3/b5.jpg";
var i = 0;

function ania() {
i++;
   document.getElementById("ba").src = mBa1.src;
   document.getElementById("ba").style.height = (150 + i * 3) + "px";
   document.getElementById("ba").style.width = (200 + i * 4) + "px";
   document.getElementById("ausgabe").innerHTML = "Breite: " +
document.getElementById("ba").style.width + ", Höhe: " +
document.getElementById("ba").style.height;
   if (i < 100)                           // schliesst  Funktion sofort ab (überspringt alles folgende!)
   window.setTimeout("ania()",1);         // startet Funktion nach 2ms erneut
}

function zurueck() {
i--;
   document.getElementById("ba").src = mBa1.src;
   document.getElementById("ba").style.height = (150 + i * 3) + "px";
   document.getElementById("ba").style.width = (200 + i * 4) + "px";
   document.getElementById("ausgabe").innerHTML = "";
```

```
  if (i > 0)                                  // schliesst  Funktion sofort ab (überspringt alles folgende!)
  window.setTimeout("zurueck()",1);   // startet Funktion nach 2ms erneut
}

function inita() {
document.getElementById("ba").onmouseover = ania;        // startet Aufblasen mit Maus
document.getElementById("ba").onmouseout = zurueck;  // startet Verkleinern mit Maus
}
window.onload = inita;
</script>
```

Aufruf des JavaScripts im Body:

```
<body>
<h1 align="center">Vergrösserungsglas</h1>
<div align="center"><img src="images3/b5.jpg" id="ba" width="300"></div>
<div id="ausgabe" align="center"></div>
</body>
```

Spez 7: Zusammenfassung zum Thema Eventhandling

Allgemeines

- Ein Eventhandler ist der Mechanismus zur Reaktion auf Ereignisse
- Eventhandler sind das wichtigstes Bindeglied zwischen HTML und JavaScript
- Es muss unterschieden werden: Eventhandler-Ereignis und Eventhandler-Reaktion

Eventhandler-Ereignisse

- Eventhandler-Ereignisse sind: onclick, onload, onmouseover, onmouseout, etc
- Es gibt HTML- und JavaScript-Eventhandler
- windows.onload =“init()“ ist z.B. ein Java-Script Eventhandler am Ende eines Scripts
- <body onload=“Funktion()“> ist z.B ein HTML-Eventhandler in einem HTML-Tags

zum Eventhandler selber

- Es gibt Eventhandler für die verschiedensten Ereignisse
- Der Eventhandler löst Funktionen oder Objektmethoden aus
- Es gibt im HTML einige festgelegte HTML-Eventhandler die als Attribute in Tags zu finden sind z.B. <body onclick=“Funktion()“

- Ein HTML Eventhandler ist nur ein Block von JavaScript-Anweisungen in einem Tag.
  z.B. <body onload="Funktion()"> HTML-Tags </body>
- In einem HTML-Tag können mehrere Eventhandler eingefügt sein (eher selten!)

Abläufe

Der HTML-String wird gleichzeitig mit dem eingebetteten oder ausgelagerten JavaScript in den Browser geladen. Der Parser des Browsers macht daraus einen DOM-Baum und legt die JavaScript Funktionen in einem gemeinsamen Speicher ab. Hier untersucht der JavaScriptInterpreter die JavaScrips und erstellt die nötigen Eventhandler.

Das Laden und Parsen des HTML-Dokumentes und das Laden des JavaScript erfolgen parallel. Der Start eines JavaScript-gesteuerten Ablaufes kann daher erst am Ende des Ladens erfolgen.

Erkenntnisse vom JavaScript-Worshop meiner Website

JavaScript-Onload-Eventhandler können sich beeinflussen, wenn sie auf derselben Seite benutzt werden

Ein JavaScript-Onload-Eventhandler überschreibt alle vorgelagerten auf der gleichen Seite

Animationen bei JavaScripts werden bei Contentmanagement-Seiten am besten mit ID's bezeichnet und mit dem Befehl getElementById(„x") behandelt. Grund: Übersichtlichkeit bei Bildern ist sonst fast nicht möglich.

Ein onload-Eventhandler weist auf eine „init"-Funktion hin welche ihrerseits verschiedene Unter-Events beherbergen kann. Beispiel:

```
window.onload=init     // Aufruf der Funktion init()

funktion init() {                                  // öffnet Funktion
document.getElementById("x").onclick = aaa;        // Aktiviert Funktion "aaa" bei Mausklick auf x
document.getElementById("y").onclick = bbb;        // Aktiviert Funktion "bbb" bei Mausklick auf y
document.getElementById("z").onclick = ccc;        // Aktiviert Funktion "ccc" bei Mausklick auf y
}
```

Bei aufeinaderfolgen von zwei onload-Eventhandlern wird die erste Funktion (init) mit der zweiten Funktion (start) überschrieben wobei alle Unter-Events der ersten Funktion verloren gehen. Der resultierende onload Befehl sieht dann effektiv wie folgt aus:

```
window.onload=start     // Aufruf der Funktion start()

funktion start() {                                 // öffnet Funktion start
document.getElementById("a").onclick = rrr;        // Aktiviert Funktion "rrr" bei Mausklick auf a
document.getElementById("b").onclick = sss;        // Aktiviert Funktion "sss" bei Mausklick auf b
```

document.getElementById(“z“).onclick = ttt; // Aktiviert Funktion “ttt“ bei Mausklick auf z
}

Das Beispiel zeigt, dass die Animationen der Funktion init() nicht mehr ablaufen! Zudem kommt in der neuen Funktion auch ein Mausklick auf “z“ vor, der aber eine andere Funktion auslöst!

Um dem Problem auszuweichen, verzichtet man am besten auf die erste onload-Anweisung und fügt alle Unter-Events der ersten onload-Funktion in die zweite onload-Funktion ein. Auf diese Weise erreicht man, dass alle Unter-Events am selben Ort abgespeichert sind und auch ausgeführt werden. Dabei muss aber unbedingt darauf geachtet werden, dass in den Unterevents nirgends für ein und dieselbe ID unterschiedliche Funktionen aufgerufen werden. Dies würde natürlich nicht funktionieren. Hier muss zwangsläufig eine der beiden Animationen verändert werden.

Mögliches Beispiel dazu:

~~window.onload=init // Aufruf der Funktion init()~~ // wurde gelöscht!

window.onload=start // Aufruf der Funktion start() // wurde erweitert

```
funktion start() {                                  // öffnet Funktion start
document.getElementById(“x“).onclick = aaa;         // Aktiviert Funktion “aaa“ bei Mausklick auf x
document.getElementById(“y“).onclick = bbb;         // Aktiviert Funktion “bbb“ bei Mausklick auf y
document.getElementById(“z“).onclick = ccc;         // Aktiviert Funktion “ccc“ bei Mausklick auf y
document.getElementById(“a“).onclick = rrr;         // Aktiviert Funktion “rrr“ bei Mausklick auf a
document.getElementById(“b“).onclick = sss;         // Aktiviert Funktion “sss“ bei Mausklick auf b
document.getElementById(“z“).onmouseover = ttt;  // Aktiviert Funktion “ttt“ bei Mausover auf z
}
```

Diese Lösung geht aber nicht, wenn die erste onload-Anweisung in einem ausgelagerten Programm versteckt ist. In meinem Fall ist das bei der **Fadeshow** der Fall. Wenn eine weitere Animation mit window.onload nach der Fadeshow kommt läuft die Fadeschow nicht mehr!

Dank dem dass **Bishow** kein window.onload hat stören sich die beiden Bildershows nicht. Die **Fadeshow** hat den JavaScript Eventhandler “window.onload=Faderframe() und startet aus dieser Funktion heraus. Die **Bishow** hat kein window.onload. Der Start erfolgt hier durch den HTML-Eventhandler onload=“los()“ im Bildtag und steuert die Funktion „los“ direkt d.h. ohne Event-Unterhandler an.

Spez 8: Erkenntnisse aus dem Programmieren

setTimeout("ani()",2);

Diese recht wichtige Funktion führt zu Problemen, wenn mehrere in einer gleichen Schleife verwendet werden. Besonderes: Alle starten zur gleichen Zeit!

Verschiedene Arten von if-Schleifen:

if (i < 100) window.setTimeout(); — ohne geschweifte Klammern. / Bei Nichterfüllen der Bedingung erfolgt Ausstieg aus der Funktionsschleife

if (String) { Code} — Wenn der String vorhanden ist wird die Klammer Truth und der nachfolgende {Code} abgearbeitet.

if (i < 100) { — mit geschweiften Klammern und else-Zweig

```
        window.setTimeout();
        } else {
            Anweisung 2
            }
```

if (stellg ==false) — ohne irgend etwas / als Ausstieg aus einer Funktionsscheife

Setzten und Vergleichen von true & false:

var stellg = false; — Setzen einer false-Variablen (mit einem Gleichheitszeichen!)

if (stellg ==false) — Vergleichen ob vorhanden. Immer mindestens zwei Gleichheitszeichen, da ein einzelnes Gleichheitszeichen als Zuweisung für Variablen verwendet wird!

if (String) { Code} — Wenn der String vorhanden ist wird die Klammer „Truth“gesetzt und der nachfolgende {Code} abgearbeitet.

DOM-Zuordnung

Bei CMBasic-Anwendungen verwendet man mit Vorteil Elelemente mit ID's, da die Bestimmung sonst sehr kompliziert und unpräzis ist.