Thomas Wenzlaff

Microsoft JScript

für den

Hobby-Programmierer

Stand 20.03.2007

Hinweise zum Dokument

Wer strukturiert und objektorientiert programmieren möchte, sollte sich z.B. mit Javascript/JScript beschäftigen.

Schwerpunkte dieser Dokumentation sind:

Programmierung mit Javascript/JScript unter dem Betriebssystem "Microsoft Windows 32-Bit" und mit JScript verbundene **programmtechnische Verwendungen** der Produkte

"Microsoft Internet Explorer"	unter und ab der Version 6.x. (vor allem ab 6.x),
"Microsoft Windows Media Player"	exemplarisch Version 7.1
"Netscape"-Browser	unter und ab der Version 6.x.

Dieses vorliegende Dokument dient nicht als Lehrmaterial, sondern ist eine stark strukturierte Systematik. Sämtliche Beispiele dienen ausschließlich dem Verständnis und als Anregungen zur Programmierung. Da sich das Dokument an den Hobby-Programmierer richtet, befindet sich in diesem Dokument nur eine eingeschränkte, aber ausreichende Syntaxbeschreibung.

Autor: Thomas Wenzlaff
www.twseite.de

Stand des Dokumentes: 20.03.2007

Rechtewahrung, Haftung und Verbesserungsvorschläge:
Das vorliegende Dokument richtet sich ausschließlich an den Nutzerkreis der Hobby-Programmierer.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz verwendet. Warennamen, Hardware- und Softwarebezeichnungen, Softwarekomponenten sowie Algorithmen werden ohne Gewährleistung der freien Verwendbarkeit benutzt, gehören dem jeweiligen Eigentümer, oder sollten als solche betrachtet werden.

Der Autor kann für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Der missbräuchliche Einsatz dieser Dokumentation - vor allem bei sicherheitsrelevanter Programmierung - wird vom Autor grundsätzlich abgelehnt.

Für Verbesserungsvorschläge und Hinweise ist der Autor dankbar.

Warnung zur Zweckentfremdung von Javascript/JScript:
Es sei darauf hingewiesen, dass mit Javascript/JScript sicherheitsrelevante Aktionen programmierbar sind, die das Sicherheitsbedürfnis des Users betreffen und/oder rechtlich relevant werden können. Ein Problem kann die Unkenntnis des Benutzers sein, wenn o.g. Aktionen nicht für den User transparent gestaltet wurden.
Eine radikale Maßnahme gegen das Risiko ist z.B. die Abschaltung von Scripten (wie Javascript), ActiveX und Cookies (z.B. im Browser selbst oder anhand einer Firewall-Software). Die Abschaltung von Javascript durch den User macht die Anwendung dieser Dokumentation durch einen Programmierer hinfällig. Dem User sind daher dringend eine Antivirus- **und** eine Firewall-Software zu empfehlen, die **beide** auch während einer Netzwerksitzung im Internet aktiv sind. Z.B. erfolgen die Blockierung von nervenden Portscans durch Mitbenutzer jeder Art im Netzwerk und die Abwehr von immer häufiger eingesetzten Trojanern und Scriptviren (Scriptviren innerhalb eines HTML-Dokumentes oder einer HTML-Email werden durch die o.g. empfohlenen Softwaren gefiltert).

Hinweise zur Kodierung von Quelltext in diesem Dokument:
Die im Dokument verwendete Rechtschreibung und Grammatik sowie das Layout des Dokumentes entsprechen nicht den aktuellen Rechtsnormen. In diesem Dokument werden auch aus Gründen der Transparenz DV-Fachbegriffe sowie Sprachstile verwendet, die nicht nur in der Sprachpflege umstritten sind. Der Ersatz deutscher Begriffe durch die aus der DV-Fachsprache ist nicht zu vermeiden. Konsequent gesehen, hätte nur die englische Sprache verwendet werden dürfen, da die meisten Softwareentwicklungen und tendenziell neuen Produkte nicht aus dem deutschsprachigen Raum stammen.

Die in diesem Dokument verwendeten Einrückungen und Einschachtelungen von HTML-Elementen wie Tags, Scripten usw. dienen vorrangig der Transparenz z.B. von Quelltexten, Kommentaren und Syntax-Beschreibungen. Tippfehler lassen sich nicht vermeiden.

Diese Dokumentation wurde u.a. erstellt mit

Microsoft Office 97:

Leider hat sich die eingesetzte Version für solche Arten von Dokumentationen als wenig geeignet erwiesen. Mit der kurz vor der tausendsten Seite erfolgten Meldung, dass die Textdatei komplett defekt war und der Autor auf Rat der Meldung den gesamten Text-Bestand über die Zwischenablage (!) in ein neues, leeres Dokument eingefügt hatte, wurde dieses klar: Auch das neue Dokument blieb defekt - Totalverlust der Daten. Die Indexerstellung über 1600 Seiten ließ die Software abstürzen (inklusive dokumenteigener Format-Vorlage). Fußzeilen funktionierten danach nicht mehr dokumentweit. Die Pflege eines Wörterbuches für die Rechtschreibungsanalyse erwies sich als nicht machbar, da Text-Formate nicht ausklammerbar waren.

Microsoft Office XP:

Die Leistungsfähigkeit der Software (auf Intel P 4 mit 1 GB Rambus, Win XP Home SP2) war am Ende, als sie ein Inhaltsverzeichnis formatiert erstellen sollte. Nach mehreren Programmabstürzen (inkl. während der Wiederherstellung des Dokumentes) sind wenigstens die Seitennummern-Felder richtig erhalten geblieben. Die zerstörte Formatierung musste manuell vereinheitlicht werden. Die Indexerstellung verlief tadellos.

Inhaltsverzeichnis

1. Dialekte und Versionen von Javascript

Ein Dialekt entspricht einer Version der Scriptmaschine des jeweiligen Browser-Herstellers.

Herstellerspezifische Versionen von Javascript - Ursachen, Vor- und Nachteile, Risiken für Web-Designer

Die Scriptmaschine ist eine Softwarekomponente (z.T. auch eine Komponente des Betriebssystems), die den Quelltext liest, interpretiert (parst) und in durch den Browser ausführbare Befehle umwandelt (z.B. Anzeige von HTML-Elementen). Die Scriptmaschine erzeugt keinen Maschinencode wie ein Compiler, sondern benutzt vordefinierte Run-Time-Bibliotheken (Objekte).

Die Browser-Version ist immer an die Version der Scriptmaschine gebunden. Implementationen in einer jüngeren Version einer Scriptmaschine müssen vom Browser älteren Datums nicht unbedingt nutzbar sein. Dafür kann die Scriptmaschine zu älteren Browsern abwärtskompatibel sein, muss es aber nicht sein. Browserinterne Elemente (Objekte) hängen also von der Scriptmaschine ab, die diese Objekte implementiert hat. Was die Scriptmaschine nicht kennt, kann der Browser erst recht nicht wissen und können.

Grundsätzlich gilt: Browser-Hersteller implementieren JavaScript immer mit browser-spezifischen Eigenschaften, die bewusst von anderen Browsern abgrenzen sollen, auch wenn ansonsten auf Kompatibilität geschworen wird. Die Abgrenzung nennt sich dann u.a. Browsermodernisierung im Rahmen der "innovativen" Standardpflege per Konsortium aus Browserherstellern. Dass also Standardpflege bewusst vom Standard wegmutiert, ist so möglich, indem z.B. der Standard über zig Jahre nicht mehr weitergepflegt wird und durch inzwischen andere heranschleichende, ev. innovative Hersteller-"Standards" ersetzt wird - XHTML ist ein Beispiel dafür. - Letztendlich geht es nur um Geld aus Rendite und Marktanteilen (wozu auch gern mal Open Source benutzt wird).
Noch cleverer macht es Microsoft: Der Browser ist konzeptionell eine abgeleitete Instanz des Windows Explorers als Instanz des Arbeitsplatzes also der GUI-Shell (Pendant in Linux z.B. KDE)). Microsoft lässt z.B. JScript durch JScript.Net innovativ ersetzen (also auch die Art der Komponenten des Browsers bzw. von Windows und deren Zugriff über JScript) und schaltet zusätzlich Komponenten von Windows wie Active-X-Control, die bisher per älterem JScript aufrufbar waren, per Definition (z.B. wegen Sicherheitsproblemen oder Patenrechtwahrung) z.T. ersatzlos ab, oder implementiert den Zugriff auf neue Komponenten nicht mehr in JScript außerhalb der Net-Umgebung: Programmierer werden dann regelrecht über den Tisch gezogen, wenn ein bisher funktionierendes Script mit Referenz z.B. auf ein Active-X-Control wegen abgeschalteten Bibliotheken des Control nicht mehr funktioniert, oder wenn z..B. eine JScript-Komponente im HTML-DOM unter neuerem Browser Fehler verursacht, obwohl sich der Syntax im HTML-DOM nicht verändert hat (konkrekte Beispiele siehe unten). Man beachte zusätzlich, das Veränderungen und Abschaltungen von vom Windows-Nutzer nicht oder doch eingespielten Patches abhängen können, wobei fehlerhafte Patches und deren eventuelle Bereinigung durch Nachfolgepatch eingeschlossen sind (konkretes Beispiel siehe unten Abänderungen wegen Sicherheitspatches der jeweiligen Windows-Versionen).

Die Schnittmenge der JavaScript-Derivate sollte also die Kompatibilität umfassen, tut es aber nicht.
Zusätzlich erschwerend ist der Umstand, dass namensgleiche JavaScript-eigene Komponenten differenziert implementiert sein können.

Es gibt aber eine Schnittmenge, die zwar unbequem (weil mit Programmierungs- und Pflegeaufwand verbunden) ist, dafür den Scriptcode portierbar hält: Man benutze Standard-Komponenten von Javascript, die allen Browsern bekannt sind. Für Browserspezifische Komponenten benutze man Bibliotheken (JS-Dateien), die je nach Browser ausgewechselt werden und deren Schnittstellen aus Standard-Komponenten bestehen.
Man **muss** also **dynamisch programmieren**, auch um Kompatibilitätsprobleme lösen zu können.

Analog zu den Standard-Script-Komponenten gibt es HTML-DOM-Funktionen, die von verschiedenen Browsern beherrscht werden - natürlich auch hier z.T. divergent z.B. der Zwang zur Nutzung der .getElementByXXXX-Funktionen in Nicht-Microsoft-Browsern, wobei der MS Internet
Explorer diese Funktionen auch kennt, aber bereits den Wert des ID-Attributes im HTML-Tag als Zeiger im Script zulässt (deswegen hat man ja im HTML-Code das ID-Attribut hinterlegt, zumal diverse HTML-Tags das ID-Attribut besitzen). Also benutze man auch hier JS-Dateien als Bibliotheken je nach Browser-Version (z.B. eine Funktion für ein im Body vorhandenes Objekt, die beim IE eval('var Zeiger='+ID_Attribut_Wert+';'); return Zeiger; und bei anderen Browsern return document.body.getElementbyId(ID_Attribut_Wert); benutzt, wobei ID_Attribut_Wert aus dem Funktionsargument stammt. Würde nur für den IE programmiert werden, würde kein Code anfallen, da ID sofort einsetzbar ist. Nur wegen anderer Browser und Code-Reduzierung ist auch im IE .getElementById() zu verwenden (Tipp: Zeiger in (globale) Variable (z.B. als Zeigerfeld-Element) speichern, oder grundsätzlich createElement() verwenden und den dadurch gelieferten Zeiger speichern müssend, wobei dann ein ID-Attribut im HTML-Code des Argumentes von createElement() keinesfalls kodiert sein darf. (doppelte Zeiger unzulässig))).

Beispiel: Netscape und Internet Explorer

Javascript-Versionen und -Dialekte zeigen z.T. sehr deutlich die unlösbaren Divergenzen zwischen den Browsern von Netscape und Microsoft. Beide Hersteller verändern z.T. inkompatibel ihre Script-Versionen gegenüber dem Javascript-Standard.

Es ist möglich, für beide Browser per Javascript-Standard zu programmieren. Dabei muss verzichtet werden

bezüglich des Internet Explorer ab 5.5	auf Erweiterungen zum Browser
bezüglich des Netscape ab 6.x	auf die Java-Klassen-Einbindung (z.B. von Sun und Netscape) für Plugins

Es ist unmöglich, ein und denselben Javascriptcode gleichzeitig für beide Browser **ohne** eine Browserunterscheidung zu verwenden. Es muss vor allem dann doppelt programmiert werden, wenn nicht der Javascript-Standard verwendet werden soll.

Näheres zu Javascript, Skriptmaschine und Objekten ist in den Beschreibungen zu den Objekten in Javascript/JScript und des Browsers zu finden. Dort werden auch ein Ansatz zur objektorientierten Programmierung mit Javascript geliefert und die Ursachen für die Existenz von Javascript-Dialekten bzw. -Versionen erklärt.

SCRIPT-Tags sind **nicht** verschachtelbar (auch nicht innerhalb eines Javascript-Codes). Man kann also **nicht** mit Javascript einen **neuen** Script-Block **innerhalb eines vorhandenen** Scriptblockes erzeugen, wenn dieser Block, der den Javascriptcode zur Script-Blockerzeugung enthält, gerade geparst wird (auch eval() funktioniert nicht). Dabei ist es egal, ob der Javascript-Code im HEAD oder BODY liegt. Sollte ein Browser doch auf o.g. Art einen inneren Scriptblock erzeugen können, so ist der Browser in diesem Fall nicht kompatibel. Außerdem kann es sein, dass ein Script-Tag, der innerhalb eines Javascriptcodes (z.B. innerhalb von document.write()) oder nach dem Kommentarzeichen // kodiert wurde, vom Browser als HTML-Tag erkannt und als solches eigenständig und unabhängig vom Kodierungskontext ausgeführt wird.

Für Programmierer unter Windows XP mit dem Internet Explorer bitte **unbedingt** beachten: Die Scriptmaschine unter Windows XP ist wesentlich **weniger fehlertolerant** als die Scriptmaschine unter Windows 98 bei identischer Browserversion und identischem Patch-Stand der Browsersoftware. Unter Windows XP ist der Internet Explorer 6.x implementiert. JScript-Code, der unter Windows 98 einwandfrei funktioniert, muss es unter Windows XP **nicht** ! Javascript-Code, der unter Windows XP funktioniert, wird es auch unter Windows 98 tun. Mit anderen Worten: Die Scriptmaschine unter Windows XP ist bezüglich Fehler **nicht** abwärtskompatibel ! Deswegen bitte **unbedingt** den Scriptcode unter Windows XP testen !

Beispiel: Microsoft ändert fortlaufend die Active-X-Eigenschaften von Windows und somit auch des Internet Explorers

Diese fortlaufenden Änderungen muss der Programmierer in Erfahrung bringen.
Der Programmierer kann sich definitiv nicht auf Verfügbarkeit von Active-X-Controls verlassen und muss damit rechnen, dass seine Webseiten schlagartig nicht mehr komplett laufen weil u.a. Programmcode noch nicht angepasst ist. Ebenfalls muss der Programmierer Varianten von Windows und Patchzustände beachten, die prinzipiell Kostenprobleme verursachen können.
Mit anderen Worten: Wer Microsoft-Komponenten nutzt, muss wissen, was ihm blüht ... siehe nachfolgende Beispiel für Risiken.

Prinzipielle Lizenzprobleme für den Programmierer

Microsoft verlangt Linzensierung von Windows. Bezüglich Windows-Versionen gibt es die Updatestufen z.B. per Servicepacks
Ein Windows mit Servicepack fällt unter die Lizenz des geupdateten Windows.
Ein Windows mit Vorversion zum Servicepack bedarf einer anderen Lizenz.
Will man z.B. den Internet Explorer 7 und 6 parallel testen, benötigt man 2 Windowslizenzen, da beide Versionen nicht parallel installierbar. Dazu kommt, dass es den IE 6 in 2 Versionen gibt: Win SP1 und SP2 (IE 7 nur ab Win SP2).
Für 3 Browserversionen benötigt man 3 Windowslizenzen, will man parallel testen.

Ein Blick auf Browser-Konkurrenzprodukte klärt die Sachlage unschlagbar: Opera ist z.B. parallel installierbar.

Hinweis: Man suche doch mal im Internet nach einem kostenlosen HTTP-Server vom Microsoft, um IE-Seite testen zu können, die JScript nutzen (inklusive Debugger). Denn sollte kein kostenloses Angebot findbar sein, kommen die Kosten von Entwicklungssoftware zum IE hinzu. Ein Blick auf Konkurrenz-HTTP-Server klärt die Sachlage: Apache-HTTP-Server ist kostenlos, allerdings nicht einfach einzurichten (Hinweis: Der HTTP-Server sollte virtuelle Hosts einrichten können und korrekt mit der Firewall des Users zusammenarbeiten können).

Abänderungen wegen Sicherheitspatches der jeweiligen Windows-Versionen

Abschaltungen von Active-X-Controls erfolgen auch im Rahmen der Sicherheitspatches zu Windows-Versionen.
Es ist auch möglich, dass wegen Sicherheitslücken abgeschaltet wird und somit Komponenten einer Webseite je nach Windowsversion nicht mehr laufen.

Im Rahmen der Sicherheitspatches ist es Microsoft sogar gelungen, Webseiten, die den MS-Encoder zur Komprimierung von
HTML- und JScript-Code nutzen, schlagartig unnutzbar zu machen: Ein Bug in einem Patch zu Windows XP - Q918899
Das Patch verusacht IE-Browser-Absturz bei per MS ScriptEncoder gepacktem JScript unter SP1 und 2 wenn HTTP 1.1 mit
Kompression genutzt wird z.B. bei
onlick-Handler auf IMG
klick ins Fenster per aktivem Popup

Der Absturz ist "read" -Fehler von immer ein und derselben Speicherstelle.
User, die dieses Patch installiert haben, können ab sofort keine IE-Seiten mit codiertem Script mehr ansehen.
Microsoft stellt Abhilfe nach geraumer Zeit zur Verfügung, jedoch spezifisch nach Windows XP-Version:
Patch Q918899 für
Windows XP SP1Download für jedermann bereitgestellt
SP2 nur auf kostenpflichte telefonische Anfrage des Users per Downloadlink bereitgestellt, da
Microsoft explizit die User registriert haben will, bei denen das
Patchproblem auftritt (User muss sich Telefonnummer besorgen)
Solange also das Patch zum fehlerhatem Patch vom User nicht installiert wird,
z.B. weil der User keine Ahnung hat, dass und wo er sich die Telefonnummer
von Microsoft besorgen muss bzw. zu besorgen hat, wird der User
IE-Seiten mit komprimierten Code dauerhaft nicht nutzen können.
(Microsoft-Support ist z.T. nur in Englisch).

Abänderungen wegen Browser-Inkompatibilität

Popupblocker-Fehler

Die Microsoft Browser-Version IE 7 ist nicht abwärtskompatibel bezüglich Popup per window.createPopup()
Popup per window-Objekt ist ein Markenzeichen des IE, das im IE 7 nicht mehr fehlerfrei nutzbar ist.
Der Fehler liegt in der Popup-Blockerverwaltung des IE und wurde mit dem IE 7 implementiert.
Der Fehler tritt nicht auf, wenn ein Fenster per window.open() erzeugt wurde.
Bedingung:
Scripfehleranzeige ist erlaubt im IE 7
Popupblocker ist im IE abgeschaltet
ein aktives Fenster (Register) mit Dokument, dass fortlaufend (rekursiv) genau 1 window.popup per .show()erzeugt.
ein weiteres Fenster (Register) z.B. leere Seite (about:blank)
beide (Register) liegen in einer gemeinsamen IE-Instanz
Ablauf: Wird Focus auf Register der leeren Seite gehalten und wird parallel das Popup per .show() erzeugt,
bricht der Browser das Dokument mit .show() ab (Scriptfehler).
Der Popupblocker für die leere Seite verursacht den Programmfehler im Dokument mit .show(). Es wird folgende
Meldung angezeigt (in der Informationsleiste):
'Ein Popup wurde geblockt. Klicken Sie hier, um das Popup bzw. weitere Optionen anzuzeigen.'
Die Bedeutung der Meldung laut Microsoft-Hilfe im IE 7:
Der Popupblocker hat ein Popupfenster geblockt. Sie können den Popupblocker deaktivieren
oder Popups temporär zulassen, indem Sie auf die Informationsleiste klicken.
Die Realität zur obigen Meldung ist völlig anders:
Linke oder rechte Maus auf die Meldung liefert z.B. Einstellungen darunter
Popupblocker einschalten
weitere Informationen
jedoch keine Möglichkeit wie laut Bedeutung
Damit gilt: Der abgeschaltete Popupblocker ist in Wirklichkeit aktiv.
Pikant: Ein Popup erscheint normalerweise auch über fremde Fenster, die nicht das Popup erzeugt haben (z.B. Fenster
einer Windowsanwendung z.B. einer anderen IE-Instanz)
Der Popupblocker des IE bemeckert aber NUR Webseite, die das Popup erzeugt.
Durch das Abwürgen von Popup wird das Popup natürlich auf und für anderen Seiten nicht relevant; im Falle einer
anderen IE-Instanz also auch für diese nicht relevant, obwohl diese Instanz per Popupblocker verwaltet wird.
Der Popupblocker beschneidet die Popup-Reichweite an der Wurzel, ist aber nicht objektorientiert zu den anderen
Webseiten (die nicht das Popup erzeugt haben).
Der Popupblocker ist nicht als Filter aufgesetzt sondern reingestrickt worden.
Der Popupblockerfehler verändert die Eventverwaltung:
Es werden u.a. ignoriert
onfocus
onblur
onfocusin
onfocusout
und viele andere, so dass trotz Events z.B. des Body der Popupblockerfehler entsteht.

```
// nachfolgender Code setzt focus nicht neu: Fenstereintrag in Taskleiste blinkt eventuell
window.focus();
window.document.focus();
if(document.body!=null)
{if(document.body.style!='hidden')          // wenn  hidden so focus() nicht möglich (Scriptfehler erzeugt)
 {document.body.focus();}
}
// wenn paralleles Fenster offen (on oder offline), so Scriptfehler erzeugt
popupzeiger.show(...);
```

Hinweis: Der Popupfehler ist so elementar, dass die vielen Beta-Testphasen des IE mehr als fragwürdig erscheinen, wie die Angabe von Microsoft, dass Code neu programmiert wurde, um den IE sicherer zu machen.

focus-Methode beim IE 7

windows.focus() document.focus() und body.focus() funktionieren NICHT
zwischen Register in einem IE-Fenster
zwischen Fensters z.B. in Taskleiste

Hinweis:
.focus() setzt Element aktiv, gibt dem Element den Focus und feuert dann onfocus
.setActive() ist Teilmenge von .focus(): nur das aktiv setzen
funktioniert nicht mit allen Elementen, mit denen .focus() funktioniert

animierte Gif (mit Timer)

Animierte Gifs (mit Timer), die unter IE 6 korrekt laufen, müssen unter IE 7 im Timer nicht mehr laufen:
z.B. garnicht mehr sichtbar, oder Timer nicht verwendet.
Dann müssen animierte Gif-Bilder nach IE-Version bereitgestellt werden.

Abänderungen wegen Rechtstreitigkeiten von Microsoft mit Fremdanbietern

Ein sehr bekanntes Beispiel ist die nachträglich eingeführte Einschränkung von Active-X-Controls wegen Patentwahrung durch Microsoft, wobei für den JScript-Programmierer massive Änderungen eintreten.

Wegen Patentwahrung hat Microsoft ein zunächst freiwilliges Patch herausgegeben, dass bei ActiveX-Control per APPLET, EMBED oder OBJECT, die auf dem Bildschirm rendern (mit oder ohne Userschnittstelle), dafür sorgt, dass bei mouseover über das Control eine Sprechblase erscheint, die darauf hinweist, dass das Objekt als ActiveX-Control klickbar ist.
Diese Sprechblase erscheint auch, wenn das Control keine Userschnittstelle hat, also diese gar nicht klickbar ist.

Es wurde das Eventmodell gleichzeitig geändert:

Es werden alle Events solange unterdrückt, bis der User die Sprechblase geklickt hat.
Das Klicken muss auf das Objekt im Sprechblasenrahmen erfolgen, der so groß ist, wie die Dimension, in der gerendert wurde.
Es muss also ERST per Mausklick das Control aktiviert werden, ehe das Control klickbar und damit die Eventsteuerung aktiviert ist.
Ein Control, dass programmtechnisch zwar was rendert, aber ansonsten ohne sichtbare programmtechnisch startet, muss ebenfalls geklickt werden, obwohl es bereits läuft und es nichts zu klicken gäbe (wenn keine Eventsteuerung eingebaut wurde).
Wegen blockierter Eventsteuerung ist also die Sprechblase z.B. nicht automatisch klickbar.
Die Eventauslösung per nicht-objekteigenen Eventhandler, der für das Objekt per fireEvent() ein Event auslöst, ist solange blockiert, bis der User die Sprechblase geklickt hat.

style.visibility='hidden' wird ignoriert

Die Sprechblase erscheint auch dann, wenn das Control mit style.visibility='hidden' belegt ist, also sich unsichtbar rendert:
Der Sprechblasenrahmen hat genau die Dimension wie die des unsichtbaren Controls. Der Sprechblasenrahmen erscheint also Zusammenhangslos, und der User weiß nicht, warum er klicken soll, wenn er nichts sieht. Vor allem weiß er nicht, WAS er klickt ... ideale Basis für Schadsoftware per Script.

Diese Sprechblase erscheint nur DANN NICHT, wenn die Userschnittstelle mit Breite == Höhe == 0 gerendert wird. Sollte die Userschnittstelle in einem Container liegen, z.B. DIV, dann wird der Container, wenn er in der Dimension kleiner ist, also die Userschnittstelle, angepasst. Daher muss der Container ebenfalls mit Breite == Höhe == 0 gerendert werden. Wegen Dimensionierung auf 0 sollte style.visivility="hidden" sein. Im Falle eines Containers reicht es, den style des Containers zu ändern, da visibility normalerweise vererbt wird an Kinder, also auch an das Control.

Abänderung wegen Abschaltungen

DirectX ist wegen Abschaltung von Active-X--Controls nicht mehr abwärtskompatibel:
Z.B. wurde bei Win XP SP2 Direct Animation aus DirectX schlagartig durch Abschaltung von Bibliotheken dezimiert, die es bei Win XP SP1 aber noch gibt.

Hier ein Beispiel aus dem Jahr 2004: Abschaltungen von Active-X-Controls

ActiveX-Controls und Unterstützung/Verbot 20041215

erlaubt sind noch

Tabular Data-Steuerelement{333C7BC4-460F-11D0-BC04-0080C7055A83} Das TDC (Tablular Data-Steuerelement) ermöglicht die Weiterverarbeitung von Daten, die nur im Textformatvorliegen, beispielsweise durch Darstellung in einer Tabelle oder Sortierung. Weitere Informationen:• http://msdn.microsoft.com/workshop/database/tdc/tabular_data_control_node_entry.asp(http://msdn.microsoft.com/workshop/database/tdc/tabular_data_control_node_entry.asp)

Microsoft Agent Control - Version 2.0 {D45FD31B-5C6E-11D1-9EC1-00C04FD7081F} Microsoft Agent repräsentiert die neue Generation des ursprünglichen Office-Assistenten. Anstatt den Assistenten jedoch innerhalb eines Rahmens darzustellen wird hier lediglich der Charakter bzw. Agent selbst dargestellt und kannauch in Webseiten verwendet werden. Weitere Informationen:• http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm(http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm)

Microsoft MSChat-Steuerelement-Objekt 2.0 - 2.5 {D6526FE0-E651-11CF-99CB-00C04FD64497} Dieses Steuerelement wird von Webautoren verwendet, um text- und graphisch basierte Chatgemeinden für Echtzeitkonversationen im Web zu erstellen.

Microsoft ActiveX Upload-Steuerelement, Version 1.5 {886e7bf0-c867-11cf-b1ae-00aa00a3f2c3} Dieses Steuerelement kann auf vielerlei Art genutzt werden, um auf einfache Weise Webinhalte via Drag and Drop zuveröffentlichen. Weitere Informationen:• 230298 (http://support.microsoft.com/kb/230298/DE/) - Posting Acceptor Release Notes
• http://msdn.microsoft.com/workshop/management/tools/reference/file_upload_control.asp (http://msdn.microsoft.com/workshop/management/tools/reference/file_upload_control.asp)

verboten sind

Datenbindung RDS {BD96C556-65A3-11D0-983A-00C04FC29E36} {BD96C556-65A3-11D0-983A-00C04FC29E33} Die RDS (Remote Data Service) Steuerelemente ermöglichen dem Browser, client-basierte SQL Abfragen an einen Webserver zu stellen. Inzwischen wurde RDS jedoch durch neuere Standards wie SOAP abgelöst, von einer weiteren Verwendung von RDS wird daher abgeraten. Weitere Informationen:• 184375 (http://support.microsoft.com/kb/184375/DE/) - Sicherheitsaspekte bei RDS 1.5, IIS 3.0 oder 4.0 und ODBC
http://msdn.microsoft.com/library/en-us/iissdk/iis/remotedatabindingwithremotedataservice.asp (http://msdn.microsoft.com/library/en-us/iissdk/iis/remotedatabindingwithremotedataservice.asp)
http://msdn.microsoft.com/library/en-us/dnmdac/html/data_mdacroadmap.asp (http://msdn.microsoft.com/library/enus/dnmdac/html/data_mdacroadmap.asp)

XMLDSO, XMLDocument, DOMDocument, und XMLIslandPeer {550dda30-0541-11d2-9ca9-0060b0ec3d39} {CFC399AF-D876-11d0-9C10-00C04FC99C8E} {e54941b2-7756-11d1-bc2a-00c04fb925f3}{7108ECB4-AFDC-11D1-ADC1-00805FC752D8}XMLDSO, XMLDocument, DOMDocument, und XMLIslandPeer ermöglichen die Verarbeitung von XML Daten,etwa die Bindung von HTML Elementen an einen XML Datensatz, oder das Einlesen, Manipulieren, und Zurückschreiben von XML Daten.

Die Steuerelemente DOMDocument und XMLIslandPeer bzw. die dazugehörigen ClassIDs sind nicht mehr aktuell, so dass von einer generellen Freigabe dieser Steuerelementgruppe abgeraten wird. Weitere Informationen:• http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_concepts2_7ook.asp(http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_concepts2_7ook.asp)

Internet Explorer

Active Setup / IE Active Setup-Steuerelement {F72A7B0E-0DD8-11D1-BD6E-00AA00B92AF1}Dieses Steuerelement enthält die in Microsoft Security Bulletin MS99-037 beschriebene Sicherheitsanfälligkeit. Umeine weitere Ausführung zu verhindern wurde im Rahmen dieses Security Bulletins ein Kill-Bit gesetzt, so dass selbst bei einer Freigabe dieses Controls eine Ausführung blockiert wird. Weitere Informationen:• http://www.microsoft.com/technet/security/bulletin/ms99-037.mspx(http://www.microsoft.com/technet/security/bulletin/ms99-037.mspx)
http://www.microsoft.com/technet/security/bulletin/fq99-037.mspx (http://www.microsoft.com/technet/security/bulletin/fq99-037.mspx)
240797 (http://support.microsoft.com/kb/240797/DE/) - So verhindern Sie die Ausführung von ActiveX-Steuerelementen in Internet Explorer

Media Player / Active Movie Runtime {A4001DE0-7075-11d0-89AB-00A0C9054129} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das Active Movie Runtime Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wirdabgeraten.

Media Player / ActiveMovie-Steuerelement{05589FA1-C356-11CE-BF01-00AA0055595A} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das Active Movie Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wird abgeraten.

Media Player / Microsoft NetShow Player {2179C5D3-EBFF-11CF-B6FD-00AA00B4E220} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das NetShow Player Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wird abgeraten.

Media Player / Windows Media Player {22D6F312-B0F6-11D0-94AB-0080C74C7E95} Dies ist das Steuerelement für Windows Media Player version 6.4 und war Installationsbestandteil bis einschließlich Windows Media Player Version 8. Ab Windows Media Player 9 wurde diese ClassID durch die neue ClassID {6BF52A52-394A-11D3-B153-00C04F79FAA6} abgelöst, deren Verwendung stattdessen empfohlen wird. Ab Windows Media Player Version 9 wirdferner die alte ClassID anhand eines Wrappers automatisch auf die neue ClassID umgeleitet. Die ClassID für Windows Media Player Version 9 ist jedoch nicht in der Liste der vom Administrator genehmigten Steuerelemente enthalten, undmuss bei Bedarf manuell hinzugefügt werden.

Animierte Schaltflächen{0482B100-739C-11CF-A3A9-00A0C9034920} Dieses Steuerelement erlaubte in frühen Versionen des Internet Explorer die Verwendung animierter Schaltfächen auf Webseiten. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von derFreigabe des Steuerelements wird daher abgeraten.

IE Label-Steuerelement
{99B42120-6EC7-11CF-A6C7-00AA00A47DD2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 auch kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

IE Menu-Steuerelement {74701400-9DD9-11CF-A662-00AA00C066D2} Dieses Steuerelement ermöglicht die Handhabung von Menüstrukturen in Webseiten, wird jedoch nicht mehr unterstützt und dürfte nur noch selten Verwendung finden. Von einer Freigabe des Steuerelements wird daher abgeraten.

IE Preloader-Steuerelement {16E349E0-702C-11CF-A3A9-00A0C9034920} Dieses Steuerelement ermöglichte das Vorladen von Webseiten, ist jedoch inzwischen nicht mehr aktuell, wird nicht mehr unterstützt und dürfte nicht mehr im Einsatz sein. Aufgrund einer potentiellen Sicherheitsanfälligkeit in diesem Steuerelement wird von einer Freigabe abgeraten. Weitere Informationen:• 231452 (http://support.microsoft.com/kb/231452/DE/) - Update Available for "Legacy ActiveX Control" Issue

IE Timer-Steuerelement {59CCB4A0-727D-11CF-AC36-00AA00A47DD2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

MCSiMenü {275E2FE0-7486-11D0-89D6-00A0C90C9B67} Dieses Steuerelement dient der Anpassung von Popupmenüs, ist jedoch nicht mehr aktuell und wurde nach Windows 98 nicht mehr ausgeliefert. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten.

Popupmenüobjekt {7823A620-9DD9-11CF-A662-00AA00C066D2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

Microsoft Agent Control - Version 1.5 {F5BE8BD2-7DE6-11D0-91FE-00C04FD701A5} Microsoft Agent repräsentiert die neue Generation des ursprünglichen Office-Assistenten. Anstatt den Assistenten jedoch innerhalb eines Rahmens darzustellen wird hier lediglich der Charakter bzw. Agent selbst dargestellt und kann auch in Webseiten verwendet werden. Diese Version des Steuerelements ist jedoch nicht mehr aktuell und wird nicht mehr unterstützt. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm (http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm)

Aktive Inhalte im Internet Explorer

Ab IE 6.0 ist das Blockieren aktiver Inhalte möglich, z.B. als Standardeinstellung. Es wird also dem IE verboten, JScript zu nutzen.
Daher muss mit Start der Webseite auf das Blockieren von Inhalten der Webseite, die auf JScript basieren, aufmerksam gemacht werden.
Bleibt die Blockierung aktiv, so muss die Webseite ALLE Elemente, die per Script angesteuert werden, inaktiv machen: Am besten garnicht erst anzeigen. Oder es wird eine scriptfreie Version der Webseite per <NOSCRIPT> aktiviert, wobei dann Browser vorzuziehbar sind, die z.B. CSS
exakter rendern als der IE (will man keine IE-spezifischen HTML-Elemente verwenden).
Wenn der IE 6.x aktive Inhalte blockiert, wird NOSCRIPT-Tag aktivert, Ausnahme: Frameset
FRAMESET ist ein aktiver Inhalt:
Da der Frameset anstelle <BODY> kodiert sein muss, gilt:
Alle Tags, die für BODY zulässig sind, werden ignoriert, auch NOSCRIPT.
Wird neben Frameset noch BODY kodiert, so wird Frameset ingoriert.
Die Freigabe der Scriptblockierung erzeugt Ausführung aller Script-Teile inklusive der Eventauslösungen

Bsp.: Folgendes funktioniert vom Dokument, das window.open() hat im geöffneten Dokument (Quelltext im Dokument das window.open() verwendet):

```
function Y_unload(X00){X85[X00].close();}

var X85=new Array();var X86=new Array();

X85[0]=window.open(...);

var X87='parent.Y_unload(0);'; X86[0]=new Function('',X87);

X85[0].document.body.onunload=X86[0];
```

Wird die Scriptblockierung im geööfneten Fenster abgeschaltet,
so wird das Fenster geschlossen, weil onunload ausgelöst wird.

Achtung: document.body.onunload funktioniert ev. nicht mehr
wenn z.B. mit attachevent() aktiviert wurde

Folgende Metatags sind für den IE 6.x aktiver Inhalt:

```
<META HTTP-EQUIV="imagetoolbar" CONTENT="no">
```
unterdrückt NICHT IE-Kontextmenü rechte Maus auf Bild

```
<META HTTP-EQUIV="site-enter" CONTENT="revealtrans(duration=0.3, transition=12) ">
<META HTTP-EQUIV="site-exit" CONTENT="revealtrans(duration=0.3, transition=12) ">
```

Achtung: Für das Hinzufügen von Elementen in den BODY (document.body) per DOM-Funktion createElement() MUSS der Body komplett geparst sein (document.body.readyState == 'complete').
Grund: Es wird standargemäß immer am Ende des BODY angefügt.

Für das Hinzufügen nicht an das Ende des BODY muss im HTML-Code ein Platzhalter z.B. DIV kodiert sein, innerhalb
dessen dann die neuen HTML-Elemente erzeugt werden.

Hinweis zum Einrichten eines virtuellen Hosts per Apache-HTTP-Server:

Anstelle des HTTP-Webservers vom Internetprovider kann der eigene PC als Test-Server fungieren, um z.B. die Webseite so zu testen, als wäre sie gerade online auf dem Webserver.

Der Hobbyprogrammierer will u.a. kostengünstig testen, also bieten sich die kostenlosen HTTP-Server an, die einen virtuellen Host anhand eines beliebigen Ordners auf der Festplatte erzeugen können, wobei Local Host (127.0.0.1) als virtueller Host einrichtbar sein muss (der nicht von der Firewall-Software des PC allein verwaltet sein darf) und die zu testende Webseite im Browser per Domainnamen aktivierbar sein muss (anstelle der Eingabe von 127.0.0.1). Man google, um festzustellen, welche Produkte diesen Kriterien entsprechen. - Abkürzend fällt die Wahl nicht zufällig auf den HTTP-Server von Apache (www.apache.org), welcher ziemlich schwierig zu konfigurieren ist, wenn mehr als nur localhost genutzt werden soll (Für Hobbyzwecke reicht localhost aus).

Der Hobbyprogrammierer, welcher für den Microsoft Internet Explorer (ab IE 7 heißt der Windows Internet Explorer) in seinen Varianten je nach Windows-Version programmieren will, wird definitiv folgende Probleme bekommen:

Microsoft lässt u.a. die Installation des Internet Explorers in parallelen Versionen nicht zu, obwohl Browserversionen nachweislich nicht kompatibel sind und Microsoft Browserversionen innerhalb Windowsversionen supportet werden. Daher benötigt man pro Version des Internet Explorers eine Windows-Installation. Pro Windows-Installation wird eine Windows-Lizenz fällig, auch wenn auf anderer Festplatte am ansonsten identischen PC installiert wird (Windows-Online-Update erkennt solche Doppelversionen und verweigert den Support). Auch wer Windows unter VM emulieren will (oder auf Apple mit Intel-Technik Windows parallel mit Apple nutzen will) , benötigt eine Lizenz. Mit anderen Worten: Auch wenn die Mehrfachinstallation nicht parallel nutzbar wäre, sondern immer nur genau 1, will Microsoft Geld haben - nicht umsonst ist Microsoft-Chef so beliebt wie reich und nicht umsonst migrieren immer mehr IT-Anwender zu Linux-Derivaten, die fast identische Browser bezüglich Windows haben.
Ergo, der Hobbyprogrammierer wird wohl sämtliche Bekannte mit Testwünschen nerven, oder illegal testen, oder auf andere Browser-Hersteller und deren HTML- sowie Script-Versionen ausweichen, die nicht nur Parallelinstallationen des Browsers zulassen (z.B. Opera unter Windows), sondern auch noch ziemlich gut kompatibel sind (weil identische Scriptmaschine nutzend).
Dass Microsoft eventuell keinen kostenlosen HTTP-Server anbietet (der ansonsten auch noch Javascript, Active-X-Control- Aufrufe per Script, DirectX-Zugriff kennen müsste), ist das kleinere - vorallem lösbarere Problem: Eben ohne Microsoft.

Pfade für Dateien im Script einer Webseite, die per HTTP-Server oder lokal von Festplatte gestartet wird:

Relative Pfade per '..' sind unabhängig davon, wie die Wurzel (Root) der Webseite heißt.
Absolute Pfade sind abhängig davon, wie die Wurzel (Root) der Webseite heißt.

Wird die Webseite (z.B. www.twseite.de) von einem HTTP-Server gestartet, dann sind absolute Pfade bezüglich http://www.twseite.de möglich, wobei http://www.twseite.de die Root der Webseite darstellt.

(Achtung: Auf einem HTTP-Server des Internetproviders liegt die Webseite natürlich auch auf einer Festplatte, also dort in einem Ordner. Dieser Ordner muss also mit www.twseite.de logisch verbunden sein - wie, das teilt der Internet-Provider mit.

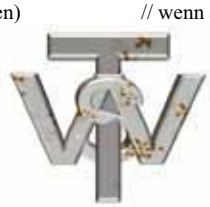
Tipp: Namen des Ordners auf der Festplatte des Internet-Providers kann man genauso nennen wie den des Ordners auf der lokalen Festplatte des PC. Ordnernamen mit Pfadzeichen wie ':' oder '//' sind natürlich nicht erlaubt)

Wird die Webseite (z.B. www.twseite.de) von lokaler Festplatte aus gestartet, dann sind absolute Pfade bezüglich http://www.twseite.de nicht möglich, da es solchen Ordnernamen nicht geben kann. Festplattenordner möglich z.B. c:\twseite\

Will man die Webseite identisch verwalten, egal ob man die Webseite von einem HTTP-Server oder von lokaler Festplatte aus startet, dann schaue man sich folgendes Beispiel für www.twseite.de an, das allerdings mit JavaScript oder JScript realisiert wird. Der Festplattenordner ist c:\twseite.

```
var BrowserAufOnlinePruefen=true;          // false für Browser nicht auf online prüfen
                                           // online: Webseite wurde auf HTTP-Server aktiviert
var DomainOhneHTTP='www.twseite.de';       // Host der Webseite ohne http:// und ohne Port
var BrowserIstOnline=false;                // Annahme: Browser ist nicht online
var PfadDerDateien='';                     // Annahme: Browser ist nicht online also
                                           //          alle Pfade unterhalb von c:\twseite\
                                           //                   wobei die Startdatei index.html der
                                           //                   Webseite eben in diesem Ordner liegt

if(BrowserAufOnlinePruefen)                // wenn auf online geprüft werden soll
```

```
{if(window.location.hostname!=null);        // aktuell gefundener Host
 BrowserIstOnline=(window.location.hostname== DomainOhneHTTP);}
                                            // Host www.xxx.yyy prüfen auf aktuelle gefundenen Host
                                            // true so Browser online
}

if(BrowserIstOnline) {PfadDerDateien='http://' + DomainOhneHTTP;}
                                            // Pfad wenn Browser online ist: Alle Pfade unterhalb von
                                            //          http://www.twseite.de
                                            //  Für den HTTP-Server liegt die Webseite natürlich
                                            //          auf einer Festplatte, also dort in einem Ordner.
                                            //          Jeder Pfad in der Webseite wird in den Ordnerpfad
                                            //          der Festplatte automatisch konvertiert (HTTP zu
                                            //          Festplatte per Dienst: Daher der Name HTTP-
                                            //          Server)
```

Nachfolgend die beispielhafte Einrichtung einer Webseite per Apache-HTTP-Server 2.2.2 bis 2.2.4. unter Windows XP ab SP 1:

Webseitendomain heißt www.twseite.de mit index.html als Startdatei
Festplattenordner der Domain, die als virtueller Host über localhost (also 127.0.0.1) laufen soll
c:\twseite\
Apache wurde installiert unter e:\wxp\apache\
wobei gelten muss
DNS-Dienst von Windows muss aktiv sein (in der Regel ist der Dienst automatisch aktiv)
Installationstyp All Users, on Port 80, as a Service - Recommended
Domain und Servername 127.0.0.1 (nicht localhost kodieren)
Email beliebig@localhost
Mit der Installation wird der Dienst-Monitor von Apache bei Windowsstart ebenfalls starten.
Es wird Apache als permanenter Dienst eingerichtet, der über den Dienst-Monitor von Apache aktivierbar / deaktivierbar ist. Es sind zwar mehrere Apache-Dienste einrichtbar, aber genau 1 kann nur immer aktiv sein.

Apache-Software einrichten:
Dienst-Monitor von Apache: siehe oben
aber: Da in Windows für jeden Aufruf der Webseite www.twseite.de unter Apache eine Einstellung getroffen werden muss, empfiehlt es sich, den Apache-Dienst grundsätzlich manuell zu starten und zu stoppen anhand nachfolgend vorgestellter BAT-Dateien.
Dafür muss aber einmalig folgendes eingestellt werden:
Der Apache-Dienst-Name ist per Dienst-Monitor zu sehen (und der Status ob aktiv oder deaktiv).
Dienste sind unter Windows per Systemsteuerung-Verwaltung-Dienste verwaltbar (auch per Apache-Monitor ist die Dienstverwaltung aktivierbar).
Der Apache-Dienst muss zuerst deaktiviert werden, dann auf Starttyp manuell gesetzt werden: Mit Windows-Start startet der Dienst nicht automatisch.

Windows anpassen an den Virtuellen Host

Die Anpassung erfolgt so, dass Apache per Batch-File, die unten erklärt werden, gestartet und deaktiviert wird (Batch-Files sind passend zur Anpassung von Windows).

Unter c:\windows\system32\drivers\etc\ liegt die Datei hosts

```
Copyright (c) 1993-1999 Microsoft Corp.
#
# Dies ist eine HOSTS-Beispieldatei, die von Microsoft TCP/IP
# für Windows 2000 verwendet wird.
#
# Diese Datei enthält die Zuordnungen der IP-Adressen zu Hostnamen.
# Jeder Eintrag muss in einer eigenen Zeile stehen. Die IP-
# Adresse sollte in der ersten Spalte gefolgt vom zugehörigen
# Hostnamen stehen.
# Die IP-Adresse und der Hostname müssen durch mindestens ein
# Leerzeichen getrennt sein.
#
# Zusätzliche Kommentare (so wie in dieser Datei) können in
# einzelnen Zeilen oder hinter dem Computernamen eingefügt werden,
```


```
# aber müssen mit dem Zeichen '#' eingegeben werden.
#
# Zum Beispiel:
#
#      102.54.94.97     rhino.acme.com          # Quellserver
#       38.25.63.10     x.acme.com              # x-Clienthost

127.0.0.1       localhost
```

Diese Datei verwaltet -wie man sieht - auch den localhost.

Standardgemäß ist localhost auf 127.0.0.1 gelegt (nur deswegen sind localhost und 127.0.0.1 synonym)
Soll aber die Webseite www.twseite.de über 127.0.0.1 gestestet werden,
muss also die Zeile

```
127.0.0.1       localhost
```

ersetzt werden durch

```
127.0.0.1       www.twseite.de
```

Nach dem Test muss localhost wieder für 127.0.01 verfügbar gemacht werden, also der Standard gesetzt werden.

Genau dieses Ersetzen machen die Batch-Files siehe unten. Dafür benötigen sie nur 2 neue Ordner, die einmalig mit Inhalt manuell angelegt werden müssen in c:\windows\system32\drivers\etc\

Schritt 1: Ordner c:\windows\system32\drivers\etc_hosts_standard\

manuell erzeugen, und dorthin die bisher unveränderte, also originale hosts-Datei kopieren
(Datei enthält 127.0.0.1 localhost)

```
# Die IP-Adresse und der Hostname müssen durch mindestens ein
# Leerzeichen getrennt sein.
#
# Zusätzliche Kommentare (so wie in dieser Datei) können in
# einzelnen Zeilen oder hinter dem Computernamen eingefügt werden,
# aber müssen mit dem Zeichen '#' eingegeben werden.
#
# Zum Beispiel:
#
#      102.54.94.97     rhino.acme.com          # Quellserver
#       38.25.63.10     x.acme.com              # x-Clienthost

127.0.0.1       localhost
```

Schritt 2 Ordner c:\windows\system32\drivers\etc_hosts_mit_www_twseite_de\
manuell erzeugen, und dorthin die bisher unveränderte, also originale hosts-Datei kopieren
(Datei enthält 127.0.0.1 localhost)
dann diese Datei in diesem Ordner per notepad.exe (nicht Word etc.) auf

```
127.0.0.1  www.twseite.de
```

```
# Copyright (c) 1993-1999 Microsoft Corp.
#
# Dies ist eine HOSTS-Beispieldatei, die von Microsoft TCP/IP
# für Windows 2000 verwendet wird.
#
# Diese Datei enthält die Zuordnungen der IP-Adressen zu Hostnamen.
# Jeder Eintrag muss in einer eigenen Zeile stehen. Die IP-
# Adresse sollte in der ersten Spalte gefolgt vom zugehörigen
# Hostnamen stehen.
# Die IP-Adresse und der Hostname müssen durch mindestens ein
# Leerzeichen getrennt sein.
#
```


```
# Zusätzliche Kommentare (so wie in dieser Datei) können in
# einzelnen Zeilen oder hinter dem Computernamen eingefügt werden,
# aber müssen mit dem Zeichen '#' eingegeben werden.
#
# Zum Beispiel:
#
#      102.54.94.97     rhino.acme.com          # Quellserver
#       38.25.63.10     x.acme.com              # x-Clienthost

127.0.0.1       www.twseite.de
```

Virtual-Host genau 1x einrichten in e:\wxp\apache\conf\ dort in der Datei httpd.conf

erst httpd.conf kopieren in einen Sicherungsordner freier Wahl

dann per notepad.exe (nicht per Word etc.) am Ende der httpd.conf folgenden Text einfügen

Achtung: Es muss natürlich dann

`www.twseite.de` ersetzt werden durch zu testende Domain

`c:/twseite` ersetzt werden durch den wirkliche Pfad

```
# Virtual Host für www.twseite.de auf 127.0.0.1:80 gehostet

# ERST Virtual Host für den Server, der damit alle Servernamen ungleich
www.twseite.de abfängt
#          also Angaben aus Serverinstalltion verwendet
NameVirtualHost 127.0.0.1
<VirtualHost 127.0.0.1>
  ServerName localhost
  ServerAlias 127.0.0.1
  DocumentRoot e:/wxp/apache/htdocs
  ErrorLog e:/wxp/apache/logs/error.log
  TransferLog e:/wxp/apache/logs/access.log
  ScriptAlias /cgi-bin/ e:/wxp/apache/cgi-bin/
</VirtualHost>

# Virtual Host für www.twseite.de
<VirtualHost 127.0.0.1>
  ServerName www.twseite.de
  ServerAlias 127.0.0.1
  DocumentRoot c:/twseite
  DirectoryIndex index.html
  ErrorLog c:/twseite\apache_error.log
  TransferLog c:/twseite\apache_access.log
  <Directory c:/twseite>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

Apache-Start und Stop per folgender BAT-Dateien, die z.B. in C:\ liegen (egal wo auf Festplatte):

Achtung: Falls der Start von Apache wegen nicht vorhandenem Dienst nicht erfolgen kann, dann gilt:

Mit der Apache-Installation wurde ein Dienst eingerichtet: siehe oben. Nur leider, der Apache-Start verlangt einen weiteren Dienst mit anderem Namen. Da bekannt ist, dass aber nur 1 Dienst zu jedem Zeitpunkt aktiv sein kann, ist es seht verwunderlich, wieso Apache-Start den bereits vorhandenen Dienst nicht nimmt.

Anstelle von wundern bitte folgendes genau 1 mal ausführen:

e:\wxp\apache\bin\ httpd.exe -k install

im bin-Verzeichnis liegen die ausführbaren Dateien von Apache.
httpd.exe ist die Hauptkomponente von Apache.
-k install installiert einen weiteren Apache-Dienst.

Logischerweise ist dieser Dienst analog zum Dienst, der bei Apache-Installation erzeugt wurde, ebenfalls per Dienste-Verwaltung zu deaktivieren und auf Starttyp manuell zu setzen.
Dieser neue Dienst hat aber den passenden Namen, der vom Start von Apache akzeptiert wird.

Hier ein Beispiel für den Dienste-Wirrwarr:

Apache-Installation erzeugt Dienst 'Apache2.2'.
Apache-Start will aber den Dienst 'Apache 2'

Batch-File 'ApacheStarten.bat' für Start des Apache, wobei der Dienst 'Apache2' aktiviert wird

```
@echo off
cls
echo Apache als Anwendung starten bei inaktivem Apache-Dienst 'Apache2'
echo             (Apache-Dienst darf nicht bei Windows-Start aktiv sein)
echo             (Apache stoppen immer per ApacheStoppen.bat)
echo hosts-Datei mit www.twseite.de bereitstellen
echo.
copy /V /Y C:\WINDOWS\system32\drivers\etc\_hosts_mit_www_twseite_de\hosts
C:\WINDOWS\system32\drivers\etc\hosts > NUL
e:\wxp\apache\bin\httpd.exe -k start
echo.
echo.
echo      ..... Apache-Anwendung wurde nur gestartet, wenn oben keine
echo                Fehlermeldung angezeigt wurde !
echo            Falls Fehlermeldung dann im Monitor Apache Servers
echo                zu allen Eintraege STOP-Button klicken (falls klickbar)
echo              und Batch-File neu starten
echo.
echo.
echo      Apache kann keine ActiveX durchreichen (z.B. Logox WebSpeech)
echo                  per JScript-erzeugte Popups nicht immer korrekt
rendern
echo.
echo.
pause
echo.
echo.
echo aktuelle hosts-Datei lautet
echo.
type C:\WINDOWS\system32\drivers\etc\hosts
echo.
pause
```

Batch-File 'ApacheStoppen.bat' für Start des Apache, wobei der Dienst 'Apache2' de-aktiviert wird

```
@echo off
cls
echo Apache als Anwendung stoppen bei aktiven Apache-Dienst 'Apache2'
echo             (Apache-Dienst aktiviert per ApacheStarten.bat)
echo hosts-Datei ohne www.twseite.de bereitstellen
echo.
copy /V /Y C:\WINDOWS\system32\drivers\etc\_hosts_standard\hosts
C:\WINDOWS\system32\drivers\etc\hosts >NUL
e:\wxp\apache\bin\httpd.exe -k stop
echo.
echo.
echo      ..... Apache-Anwendung wurde nur gestoppt, wenn oben keine
```


```
echo                Fehlermeldung angezeigt wurde !
echo            Falls Fehlermeldung dann im Monitor Apache Servers
echo                dann war kein Appache-Server aktiv.
echo.
echo.
pause
echo.
echo.
echo aktuelle hosts-Datei lautet
echo.
type C:\WINDOWS\system32\drivers\etc\hosts
echo.
pause
```

Batch-File 'ApacheHostsDateiAnzeigen.bat' für Anzeige der hosts-Datei

```
@echo off
cls
echo aktuelle hosts-Datei anzeigen
echo.
type C:\WINDOWS\system32\drivers\etc\hosts
echo.
echo.
echo.
pause
```

Webseite aktivieren per Apache:

Da die Webseite nur per o.g. Batch-Dateien gestartet bzw. gestoppt werden kann, kann der Apache-Dienst-Monitor nur noch zur Kontrolle benutzt werden, ob eine Apache-Dienst aktiv ist.

Schritt 1: Vor dem Start der Webseite darf kein Apache-Dienst aktiv sein

Schritt 2 ApacheHostsDateiAnzeigen.bat aktivieren. Die hosts-Datei muss enthalten

127.0.0.1 localhost

Schritt 3: ApacheStarten.bat aktivieren. Apache startet.

Wer will kann jetzt Schritt 2 wiederholen und sieht dann

127.0.01. www.twseite.de

Wenn jetzt die Firewall-Software sich meldet, dann Apache erlauben am Port 80 (also HTTP) von 127.0.0.1

Achtung: Nutzt die zu testende Webseite andere Ports z.B. die von Plugins, dann können die Plugins selbst einen Firewalleintrag verlangen, aber Apache kann natürlich nur HTTP (Port 80) und muss nur zwischen Webseite und den Plugins vermitteln (eben über Port 80)

Schritt 4: Browser der Wahl aktivieren
Dort die Domain www.twseite.de eintippen .... die Webseite öffnet sich anhand c:\twseite\index.html

Tipp: Sollte die Webseite bereits auf einem Webserver online sein, dann würde ohne Apache-Start natürlich die online-Variante aktiviert werden und nicht die Webseite von der Festplatte aus dem Ordner c:\twseite !

Wichtig: Sollte die Webseite im Ordner c:\twseite\ geändert worden sein, so muss der Browser-Cache gelöscht werden, bevor die die Webseite erneut aktiviert wird. Apache lässt den Browser genauso agieren, als ob der online die Webseite laden würde, also über den Browser-Cache. Manche Browser oder Webseiten unterbinden dann das Neuladen einer Datei, die im Browser-Cache dem Namen nach vorhanden ist, aber in c:\twseite verändert vorliegt also zwingend neu geladen werden muss.... Ergo Browser-Cache vorher löschen.

Browser-Cache im löschen im
Internet Explorer per Internetoptionen, die
im Browser per Menüpunkt Extra oder per Systemsteuerung aktivierbar sind (ev. Verknüpfung auf Desktop manuell erzeugen)
Opera per Menüpunkt Extra-Einstellungen-Erweitert

Schritt 5: Webseite getestet (man hat jetzt ein Smily-Gesicht oder graue Haare mehr, oder den Chef, Kunden, Frau am Hals *g)
also Zeit zum Deaktivieren von Apache per ApacheStoppen.bat

Schritt 6: Webseitendaten aus c:\twseite sichern.

1.1. Javascript-Versionen beim Netscape

Die Version der Scriptmaschine wird durch HTML-Angabe des unterstützten Javascript-Standards bekannt gegeben.

HTML-Kodierung der Scriptversion	Javascript-Standard	NS-Version
<SCRIPT LANGUAGE="JavaScript1.0">	Version 1.0	ab NS 2.x
<SCRIPT LANGUAGE="JavaScript1.1">	Version 1.1	ab NS 3.x
<SCRIPT LANGUAGE="JavaScript1.2">	Version 1.2	NS 4.0 bis 4.05
<SCRIPT LANGUAGE="JavaScript1.3">	Version 1.3	NS 4.06 bis 4.70
<SCRIPT LANGUAGE="JavaScript1.4">	Version 1.4	NS 4.7x
<SCRIPT LANGUAGE="JavaScript1.5">	Version 1.5	ab NS 6.x

Wird nur <SCRIPT> kodiert wurde, so gilt
JScript wenn das HTML-Dokument unter dem IE läuft
Javascript in der Version laut Browser, wenn das HTML-Dokument unter dem NS läuft

Javascript wird gegenüber dem Javascript-Standard z.B. unter NS ab 6.x browserspezifisch erweitert .

Beispiele: Java-Klassen-Einbindung für Plugins
Verwaltung signierter Scripts

Netscape kann Erweiterungen des Browsers durch Plugins realisieren, die
auch über Javascript programmierbar sein **können**, aber nicht müssen
bezüglich dem IE 6.x **inkompatibel** sind, da dieser keine Plugins mehr unterstützt.

1.2. Javascript-Versionen von Microsoft

Die Version der Scriptmaschine wird durch **keine** HTML-Angabe bekannt gegeben.
JScript kennt auch keine HTML-Angaben für die Sprachversionen von JScript. Es muss **nur** kodiert werden:

<SCRIPT LANGUAGE="JScript">

Wird nur <SCRIPT> kodiert wurde, so gilt
JScript wenn das HTML-Dokument unter dem IE läuft
Javascript in der Version laut Browser, wenn das HTML-Dokument unter dem NS äuft

Hinweis: Wenn unter dem IE <SCRIPT LANGUAGE="JavaScript1.x> (x ist zu ersetzen mit der Versionsnummer) **und** zugleich JScript-spezifische Elemente kodiert wurden, dann erfolgt Verwendung von JScript.

JScript ist nur z.T. kompatibel zum Javascript-Standard:

Javascript-Standard	IE-Version
JavaScript 1.0	ab IE 3.0
JavaScript 1.1	ab IE 3.02 teilweise
JavaScript 1.2	ab IE 4.0
JavaScript 1.3	ab IE 5.0
JavaScript 1.4	ab IE 6.0
JavaScript 1.5	unklar ab wann und ob überhaupt noch

JScript erweitert browserspezifisch den Sprachumfang gegenüber dem Javascript-Standard
integriert optional Teile des Betriebssystems und Addons von Windows z.B.
Windows Media Player
Direct Animation
Active-X-Controls (ab IE 6.x als Totalersatz für Plugins)
ist eine Schnittstelle zur Erweiterung des Internet Explorers, der in Windows z.T. integriert ist.
unterstützt ab dem IE 6.x Plugins als Browsererweiterungen **nicht** mehr

Für Programmierer unter Windows XP mit dem Internet Explorer bitte **unbedingt** beachten: Die Scriptmaschine unter Windows XP ist wesentlich **weniger fehlertolerant** als die Scriptmaschine unter Windows 98 bei identischer Browserversion und identischem Patch-Stand der Browsersoftware. Unter Windows XP ist der Internet Explorer 6.x implementiert. JScript-Code, der unter Windows 98 einwandfrei funktioniert, muss es unter Windows XP **nicht** ! Javascript-Code, der unter Windows XP funktioniert, wird es auch unter Windows 98 tun. Mit anderen Worten: Die Scriptmaschine unter Windows XP ist bezüglich Fehler **nicht** abwärtskompatibel ! Deswegen bitte **unbedingt** den Scriptcode unter Windows XP testen !

Microsoft ändert fortlaufend die Active-X-Eigenschaften von Windows und somit auch des Internet Explorers

Diese fortlaufenden Änderungen muss der Programmierer in Erfahrung bringen.
Der Programmierer kann sich definitiv nicht auf Verfügbarkeit von Active-X-Controls verlassen und muss damit rechnen, dass seine Webseiten schlagartig nicht mehr komplett laufen weil u.a. Programmcode noch nicht angepasst ist. Ebenfalls muss der Programmierer Varianten von Windows und Patchzustände beachten, die prinzipiell Kostenprobleme verursachen können.
Mit anderen Worten: Wer Microsoft-Komponenten nutzt, muss wissen, was ihm blüht ... siehe nachfolgende Beispiel für Risiken.

Prinzipielle Lizenzprobleme für den Programmierer

Microsoft verlangt Linzensierung von Windows. Bezüglich Windows-Versionen gibt es die Updatestufen z.B. per Servicepacks
- Ein Windows mit Servicepack fällt unter die Lizenz des geupdateten Windows.
- Ein Windows mit Vorversion zum Servicepack bedarf einer anderen Lizenz.
- Will man z.B. den Internet Explorer 7 und 6 parallel testen, benötigt man 2 Windowslizenzen, da beide Versionen nicht parallel installierbar. Dazu kommt, dass es den IE 6 in 2 Versionen gibt: Win SP1 und SP2 (IE 7 nur ab Win SP2).
- Für 3 Browserversionen benötigt man 3 Windowslizenzen, will man parallel testen.
- Ein Blick auf Browser-Konkurrenzprodukte klärt die Sachlage unschlagbar: Opera ist z.B. parallel installierbar.

Hinweis: Man suche doch mal im Internet nach einem kostenlosen HTTP-Server vom Microsoft, um IE-Seite testen zu können, die JScript nutzen (inklusive Debugger). Denn sollte kein kostenloses Angebot findbar sein, kommen die Kosten von Entwicklungssoftware zum IE hinzu. Ein Blick auf Konkurrenz-HTTP-Server klärt die Sachlage: Apache-HTTP-Server ist kostenlos, allerdings nicht einfach einzurichten (Hinweis: Der HTTP-Server sollte virtuelle Hosts einrichten können und korrekt mit der Firewall des Users zusammenarbeiten können).

Abänderungen wegen Sicherheitspatches der jeweiligen Windows-Versionen

Abschaltungen von Active-X-Controls erfolgen auch im Rahmen der Sicherheitspatches zu Windows-Versionen.
Es ist auch möglich, dass wegen Sicherheitslücken abgeschaltet wird und somit Komponenten einer Webseite je nach Windowsversion nicht mehr laufen.

Im Rahmen der Sicherheitspatches ist es Microsoft sogar gelungen, Webseiten, die den MS-Encoder zur Komprimierung von HTML- und JScript-Code nutzen, schlagartig unnutzbar zu machen: Ein Bug in einem Patch zu Windows XP - Q918899
- Das Patch verusacht IE-Browser-Absturz bei per MS ScriptEncoder gepacktem JScript unter SP1 und 2 wenn HTTP 1.1 mit Kompression genutzt wird z.B. bei
  - onlick-Handler auf IMG
  - klick ins Fenster per aktivem Popup

- Der Absturz ist "read" -Fehler von immer ein und derselben Speicherstelle.
- User, die dieses Patch installiert haben, können ab sofort keine IE-Seiten mit codiertem Script mehr ansehen.
- Microsoft stellt Abhilfe nach geraumer Zeit zur Verfügung, jedoch spezifisch nach Windows XP-Version:
  - Patch Q918899 für
    - Windows XP SP1Download für jedermann bereitgestellt
    - SP2 nur auf kostenpflichte telefonische Anfrage des Users per Downloadlink bereitgestellt, da Microsoft explizit die User registriert haben will, bei denen das Patchproblem auftritt (User muss sich Telefonnummer besorgen)
      Solange also das Patch zum fehlerhatem Patch vom User nicht installiert wird, z.B. weil der User keine Ahnung hat, dass und wo er sich die Telefonnummer von Microsoft besorgen muss bzw. zu besorgen hat, wird der User IE-Seiten mit komprimierten Code dauerhaft nicht nutzen können. (Microsoft-Support ist z.T. nur in Englisch).

Abänderungen wegen Browser-Inkompatibilität

Popupblocker-Fehler

Die Microsoft Browser-Version IE 7 ist nicht abwärtskompatibel bezüglich Popup per window.createPopup()
Popup per window-Objekt ist ein Markenzeichen des IE, das im IE 7 nicht mehr fehlerfrei nutzbar ist.
Der Fehler liegt in der Popup-Blockerverwaltung des IE und wurde mit dem IE 7 implementiert.
Der Fehler tritt nicht auf, wenn ein Fenster per window.open() erzeugt wurde.
Bedingung:
- Scripfehleranzeige ist erlaubt im IE 7
- Popupblocker ist im IE abgeschaltet
- ein aktives Fenster (Register) mit Dokument, dass fortlaufend (rekursiv) genau 1 window.popup per .show()erzeugt.

ein weiteres Fenster (Register) z.B. leere Seite (about:blank)
beide (Register) liegen in einer gemeinsamen IE-Instanz
Ablauf: Wird Focus auf Register der leeren Seite gehalten und wird parallel das Popup per .show() erzeugt,
bricht der Browser das Dokument mit .show() ab (Scriptfehler).
Der Popupblocker für die leere Seite verursacht den Programmfehler im Dokument mit .show(). Es wird folgende
Meldung angezeigt (in der Informationsleiste):
'Ein Popup wurde geblockt. Klicken Sie hier, um das Popup bzw. weitere Optionen anzuzeigen.'
Die Bedeutung der Meldung laut Microsoft-Hilfe im IE 7:
Der Popupblocker hat ein Popupfenster geblockt. Sie können den Popupblocker deaktivieren
oder Popups temporär zulassen, indem Sie auf die Informationsleiste klicken.
Die Realität zur obigen Meldung ist völlig anders:
Linke oder rechte Maus auf die Meldung liefert z.B. Einstellungen darunter
Popupblocker einschalten
weitere Informationen
jedoch keine Möglichkeit wie laut Bedeutung
Damit gilt: Der abgeschaltete Popupblocker ist in Wirklichkeit aktiv.
Pikant: Ein Popup erscheint normalerweise auch über fremde Fenster, die nicht das Popup erzeugt haben (z.B. Fenster
einer Windowsanwendung z.B. einer anderen IE-Instanz)
Der Popupblocker des IE bemeckert aber NUR Webseite, die das Popup erzeugt.
Durch das Abwürgen von Popup wird das Popup natürlich auf und für anderen Seiten nicht relevant; im Falle einer
anderen IE-Instanz also auch für diese nicht relevant, obwohl diese Instanz per Popupblocker verwaltet wird.
Der Popupblocker beschneidet die Popup-Reichweite an der Wurzel, ist aber nicht objektorientiert zu den anderen
Webseiten (die nicht das Popup erzeugt haben).
Der Popupblocker ist nicht als Filter aufgesetzt sondern reingestrickt worden.
Der Popupblockerfehler verändert die Eventverwaltung:
Es werden u.a. ignoriert
onfocus
onblur
onfocusin
onfocusout
und viele andere, so dass trotz Events z.B. des Body der Popupblockerfehler entsteht.

```
// nachfolgender Code setzt focus nicht neu: Fenstereintrag in Taskleiste blinkt eventuell
window.focus();
window.document.focus();
if(document.body!=null)
{if(document.body.style!='hidden')          // wenn  hidden so focus() nicht möglich (Scriptfehler erzeugt)
 {document.body.focus();}
}
// wenn paralleles Fenster offen (on oder offline), so Scriptfehler erzeugt
popupzeiger.show(...);
```

Hinweis: Der Popupfehler ist so elementar, dass die vielen Beta-Testphasen des IE mehr als fragwürdig erscheinen, wie die Angabe von Microsoft, dass Code neu programmiert wurde, um den IE sicherer zu machen.

focus-Methode beim IE 7

windows.focus() document.focus() und body.focus() funktionieren NICHT
zwischen Register in einem IE-Fenster
zwischen Fensters z.B. in Taskleiste

Hinweis:
.focus() setzt Element aktiv, gibt dem Element den Focus und feuert dann onfocus
.setActive() ist Teilmenge von .focus(): nur das aktiv setzen
funktioniert nicht mit allen Elementen, mit denen .focus() funktioniert

animierte Gif (mit Timer)

Animierte Gifs (mit Timer), die unter IE 6 korrekt laufen, müssen unter IE 7 im Timer nicht mehr laufen:
z.B. garnicht mehr sichtbar, oder Timer nicht verwendet.
Dann müssen animierte Gif-Bilder nach IE-Version bereitgestellt werden.

Abänderungen wegen Rechtstreitigkeiten von Microsoft mit Fremdanbietern

Ein sehr bekanntes Beispiel ist die nachträglich eingeführte Einschränkung von Active-X-Controls wegen Patentwahrung durch Microsoft, wobei für den JScript-Programmierer massive Änderungen eintreten.

Wegen Patentwahrung hat Microsoft ein zunächst freiwilliges Patch herausgegeben, dass bei ActiveX-Control per APPLET, EMBED oder OBJECT, die auf dem Bildschirm rendern (mit oder ohne Userschnittstelle), dafür sorgt, dass bei mouseover über das Control eine Sprechblase erscheint, die darauf hinweist, dass das Objekt als ActiveX-Control klickbar ist.
Diese Sprechblase erscheint auch, wenn das Control keine Userschnittstelle hat, also diese gar nicht klickbar ist.

Es wurde das Eventmodell gleichzeitig geändert:

Es werden alle Events solange unterdrückt, bis der User die Sprechblase geklickt hat.
Das Klicken muss auf das Objekt im Sprechblasenrahmen erfolgen, der so groß ist, wie die Dimension, in der gerendert wurde.
Es muss also ERST per Mausklick das Control aktiviert werden, ehe das Control klickbar und damit die Eventsteuerung aktiviert ist.
Ein Control, dass programmtechnisch zwar was rendert, aber ansonsten ohne sichtbare programmtechnisch startet, muss ebenfalls geklickt werden, obwohl es bereits läuft und es nichts zu klicken gäbe (wenn keine Eventsteuerung eingebaut wurde).
Wegen blockierter Eventsteuerung ist also die Sprechblase z.B. nicht automatisch klickbar.
Die Eventauslösung per nicht-objekteigenen Eventhandler, der für das Objekt per fireEvent() ein Event auslöst, ist solange blockiert, bis der User die Sprechblase geklickt hat.

style.visibility='hidden' wird ignoriert

Die Sprechblase erscheint auch dann, wenn das Control mit style.visibility='hidden' belegt ist, also sich unsichtbar rendert:
Der Sprechblasenrahmen hat genau die Dimension wie die des unsichtbaren Controls. Der Sprechblasenrahmen erscheint also Zusammenhangslos, und der User weiß nicht, warum er klicken soll, wenn er nichts sieht. Vor allem weiß er nicht, WAS er klickt ... ideale Basis für Schadsoftware per Script.

Diese Sprechblase erscheint nur DANN NICHT, wenn die Userschnittstelle mit Breite == Höhe == 0 gerendert wird. Sollte die Userschnittstelle in einem Container liegen, z.B. DIV, dann wird der Container, wenn er in der Dimension kleiner ist, also die Userschnittstelle, angepasst. Daher muss der Container ebenfalls mit Breite == Höhe == 0 gerendert werden. Wegen Dimensionierung auf 0 sollte style.visivility="hidden" sein. Im Falle eines Containers reicht es, den style des Containers zu ändern, da visibility normalerweise vererbt wird an Kinder, also auch an das Control.

Abänderung wegen Abschaltungen

DirectX ist wegen Abschaltung von Active-X--Controls nicht mehr abwärtskompatibel:
Z.B. wurde bei Win XP SP2 Direct Animation aus DirectX schlagartig durch Abschaltung von Bibliotheken dezimiert, die es bei Win XP SP1 aber noch gibt.

Hier ein Beispiel aus dem Jahr 2004: Abschaltungen von Active-X-Controls

ActiveX-Controls und Unterstützung/Verbot 20041215

erlaubt sind noch

Tabular Data-Steuerelement{333C7BC4-460F-11D0-BC04-0080C7055A83} Das TDC (Tablular Data-Steuerelement) ermöglicht die Weiterverarbeitung von Daten, die nur im Textformatvorliegen, beispielsweise durch Darstellung in einer Tabelle oder Sortierung. Weitere Informationen:• http://msdn.microsoft.com/workshop/database/tdc/tabular_data_control_node_entry.asp(http://msdn.microsoft.com/workshop/database/tdc/tabular_data_control_node_entry.asp)

Microsoft Agent Control - Version 2.0 {D45FD31B-5C6E-11D1-9EC1-00C04FD7081F} Microsoft Agent repräsentiert die neue Generation des ursprünglichen Office-Assistenten. Anstatt den Assistenten jedoch innerhalb eines Rahmens darzustellen wird hier lediglich der Charakter bzw. Agent selbst dargestellt und kannauch in Webseiten verwendet werden. Weitere Informationen:• http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm(http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm)

Microsoft MSChat-Steuerelement-Objekt 2.0 - 2.5 {D6526FE0-E651-11CF-99CB-00C04FD64497} Dieses Steuerelement wird von Webautoren verwendet, um text- und graphisch basierte Chatgemeinden für Echtzeitkonversationen im Web zu erstellen.

Microsoft ActiveX Upload-Steuerelement, Version 1.5 {886e7bf0-c867-11cf-b1ae-00aa00a3f2c3} Dieses Steuerelement kann auf vielerlei Art genutzt werden, um auf einfache Weise Webinhalte via Drag and Drop zuveröffentlichen. Weitere Informationen:• 230298 (http://support.microsoft.com/kb/230298/DE/) - Posting Acceptor Release Notes

• http://msdn.microsoft.com/workshop/management/tools/reference/file_upload_control.asp (http://msdn.microsoft.com/workshop/management/tools/reference/file_upload_control.asp)

verboten sind

Datenbindung RDS {BD96C556-65A3-11D0-983A-00C04FC29E36} {BD96C556-65A3-11D0-983A-00C04FC29E33} Die RDS (Remote Data Service) Steuerelemente ermöglichen dem Browser, client-basierte SQL Abfragen an einen Webserver zu stellen. Inzwischen wurde RDS jedoch durch neuere Standards wie SOAP abgelöst, von einer weiteren Verwendung von RDS wird daher abgeraten. Weitere Informationen:• 184375 (http://support.microsoft.com/kb/184375/DE/) - Sicherheitsaspekte bei RDS 1.5, IIS 3.0 oder 4.0 und ODBC

http://msdn.microsoft.com/library/en-us/iissdk/iis/remotedatabindingwithremotedataservice.asp (http://msdn.microsoft.com/library/en-us/iissdk/iis/remotedatabindingwithremotedataservice.asp)

http://msdn.microsoft.com/library/en-us/dnmdac/html/data_mdacroadmap.asp (http://msdn.microsoft.com/library/enus/dnmdac/html/data_mdacroadmap.asp)

XMLDSO, XMLDocument, DOMDocument, und XMLIslandPeer {550dda30-0541-11d2-9ca9-0060b0ec3d39} {CFC399AF-D876-11d0-9C10-00C04FC99C8E} {e54941b2-7756-11d1-bc2a-00c04fb925f3}{7108ECB4-AFDC-11D1-ADC1-00805FC752D8}XMLDSO, XMLDocument, DOMDocument, und XMLIslandPeer ermöglichen die Verarbeitung von XML Daten,etwa die Bindung von HTML Elementen an einen XML Datensatz, oder das Einlesen, Manipulieren, und Zurückschreiben von XML Daten.

Die Steuerelemente DOMDocument und XMLIslandPeer bzw. die dazugehörigen ClassIDs sind nicht mehr aktuell, so dass von einer generellen Freigabe dieser Steuerelementgruppe abgeraten wird. Weitere Informationen:• http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_concepts2_7ook.asp(http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_concepts2_7ook.asp)

Internet Explorer

Active Setup / IE Active Setup-Steuerelement {F72A7B0E-0DD8-11D1-BD6E-00AA00B92AF1}Dieses Steuerelement enthält die in Microsoft Security Bulletin MS99-037 beschriebene Sicherheitsanfälligkeit. Umeine weitere Ausführung zu verhindern wurde im Rahmen dieses Security Bulletins ein Kill-Bit gesetzt, so dass selbst bei einer Freigabe dieses Controls eine Ausführung blockiert wird. Weitere Informationen:• http://www.microsoft.com/technet/security/bulletin/ms99-037.mspx(http://www.microsoft.com/technet/security/bulletin/ms99-037.mspx)

http://www.microsoft.com/technet/security/bulletin/fq99-037.mspx (http://www.microsoft.com/technet/security/bulletin/fq99-037.mspx)

240797 (http://support.microsoft.com/kb/240797/DE/) - So verhindern Sie die Ausführung von ActiveX-Steuerelementen in Internet Explorer

Media Player / Active Movie Runtime {A4001DE0-7075-11d0-89AB-00A0C9054129} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das Active Movie Runtime Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wirdabgeraten.

Media Player / ActiveMovie-Steuerelement{05589FA1-C356-11CE-BF01-00AA0055595A} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das Active Movie Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wird abgeraten.

Media Player / Microsoft NetShow Player {2179C5D3-EBFF-11CF-B6FD-00AA00B4E220} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement

abgedeckt. Das NetShow Player Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wird abgeraten.

Media Player / Windows Media Player {22D6F312-B0F6-11D0-94AB-0080C74C7E95} Dies ist das Steuerelement für Windows Media Player version 6.4 und war Installationsbestandteil bis einschließlich Windows Media Player Version 8. Ab Windows Media Player 9 wurde diese ClassID durch die neue ClassID {6BF52A52-394A-11D3-B153-00C04F79FAA6} abgelöst, deren Verwendung stattdessen empfohlen wird. Ab Windows Media Player Version 9 wirdferner die alte ClassID anhand eines Wrappers automatisch auf die neue ClassID umgeleitet. Die ClassID für Windows Media Player Version 9 ist jedoch nicht in der Liste der vom Administrator genehmigten Steuerelemente enthalten, undmuss bei Bedarf manuell hinzugefügt werden.

Animierte Schaltflächen{0482B100-739C-11CF-A3A9-00A0C9034920} Dieses Steuerelement erlaubte in frühen Versionen des Internet Explorer die Verwendung animierter Schaltfächen auf Webseiten. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von derFreigabe des Steuerelements wird daher abgeraten.

IE Label-Steuerelement
{99B42120-6EC7-11CF-A6C7-00AA00A47DD2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 auch kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

IE Menu-Steuerelement {74701400-9DD9-11CF-A662-00AA00C066D2} Dieses Steuerelement ermöglicht die Handhabung von Menüstrukturen in Webseiten, wird jedoch nicht mehr unterstützt und dürfte nur noch selten Verwendung finden. Von einer Freigabe des Steuerelements wird daher abgeraten.

IE Preloader-Steuerelement {16E349E0-702C-11CF-A3A9-00A0C9034920} Dieses Steuerelement ermöglichte das Vorladen von Webseiten, ist jedoch inzwischen nicht mehr aktuell, wird nicht mehr unterstützt und dürfte nicht mehr im Einsatz sein. Aufgrund einer potentiellen Sicherheitsanfälligkeit in diesem Steuerelement wird von einer Freigabe abgeraten. Weitere Informationen:• 231452 (http://support.microsoft.com/kb/231452/DE/) - Update Available for "Legacy ActiveX Control" Issue

IE Timer-Steuerelement {59CCB4A0-727D-11CF-AC36-00AA00A47DD2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

MCSiMenü {275E2FE0-7486-11D0-89D6-00A0C90C9B67} Dieses Steuerelement dient der Anpassung von Popupmenüs, ist jedoch nicht mehr aktuell und wurde nach Windows 98 nicht mehr ausgeliefert. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten.

Popupmenüobjekt {7823A620-9DD9-11CF-A662-00AA00C066D2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 kein Bestandteil der Installation mehr. Das Steuerelement wird nicht

mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

Microsoft Agent Control - Version 1.5 {F5BE8BD2-7DE6-11D0-91FE-00C04FD701A5} Microsoft Agent repräsentiert die neue Generation des ursprünglichen Office-Assistenten. Anstatt den Assistenten jedoch innerhalb eines Rahmens darzustellen wird hier lediglich der Charakter bzw. Agent selbst dargestellt und kann auch in Webseiten verwendet werden. Diese Version des Steuerelements ist jedoch nicht mehr aktuell und wird nicht mehr unterstützt. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm (http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm)

Aktive Inhalte im Internet Explorer

Ab IE 6.0 ist das Blockieren aktiver Inhalte möglich, z.B. als Standardeinstellung. Es wird also dem IE verboten, JScript zu nutzen.
Daher muss mit Start der Webseite auf das Blockieren von Inhalten der Webseite, die auf JScript basieren, aufmerksam gemacht werden.
Bleibt die Blockierung aktiv, so muss die Webseite ALLE Elemente, die per Script angesteuert werden, inaktiv machen: Am besten garnicht erst anzeigen. Oder es wird eine scriptfreie Version der Webseite per <NOSCRIPT> aktiviert, wobei dann Browser vorzuziehbar sind, die z.B. CSS
exakter rendern als der IE (will man keine IE-spezifischen HTML-Elemente verwenden).
Wenn der IE 6.x aktive Inhalte blockiert, wird NOSCRIPT-Tag aktivert, Ausnahme: Frameset
FRAMESET ist ein aktiver Inhalt:
Da der Frameset anstelle <BODY> kodiert sein muss, gilt:
Alle Tags, die für BODY zulässig sind, werden ignoriert, auch NOSCRIPT.
Wird neben Frameset noch BODY kodiert, so wird Frameset ingoriert.
Die Freigabe der Scriptblockierung erzeugt Ausführung aller Script-Teile inklusive der Eventauslösungen

Bsp.: Folgendes funktioniert vom Dokument, das window.open() hat im geöffneten Dokument (Quelltext im Dokument das window.open() verwendet):

```
function Y_unload(X00){X85[X00].close();}

var X85=new Array();var X86=new Array();

X85[0]=window.open(...);

var X87='parent.Y_unload(0);'; X86[0]=new Function('',X87);

X85[0].document.body.onunload=X86[0];
```

Wird die Scriptblockierung im geööfneten Fenster abgeschaltet,
so wird das Fenster geschlossen, weil onunload ausgelöst wird.

Achtung: document.body.onunload funktioniert ev. nicht mehr
wenn z.B. mit attachevent() aktiviert wurde

Folgende Metatags sind für den IE 6.x aktiver Inhalt:

```
<META HTTP-EQUIV="imagetoolbar" CONTENT="no">
```
unterdrückt NICHT IE-Kontextmenü rechte Maus auf Bild

```
<META HTTP-EQUIV="site-enter" CONTENT="revealtrans(duration=0.3, transition=12) ">
<META HTTP-EQUIV="site-exit" CONTENT="revealtrans(duration=0.3, transition=12) ">
```

Achtung: Für das Hinzufügen von Elementen in den BODY (document.body) per DOM-Funktion createElement() MUSS der Body komplett geparst sein (document.body.readyState == 'complete').
Grund: Es wird standargemäß immer am Ende des BODY angefügt.
Für das Hinzufügen nicht an das Ende des BODY muss im HTML-Code ein Platzhalter z.B. DIV kodiert sein, innerhalb
dessen dann die neuen HTML-Elemente erzeugt werden.

1.3. Feststellung des Dialektes von Javascript

1.3.1. Schritt 1: Erkennung des Browserherstellers

1.3.1.1. Browsererkennung anhand des Browsernamen (navigator.appName)

Diese Art der Browsererkennung ist nicht zuverlässig.

Es wird das Objekt navigator.appName benutzt, dessen Implementation sich aber bezüglich der Werte ändern kann (je nach Version der Scriptmaschine).

Beispiel:

```
<SCRIPT LANGUAGE="JavaScript">
<!-      var ie4=0;                                                    /* Annahme: Netscape gefunden */
         if (     (navigator.appName.indexOf("Explorer")>=0)           /* auf Explorer */
            &&    (navigator.appVersion.indexOf('4.0')    >=0)         /* der Version 4.0 prüfen */
           )
         ie4=1;
// -->
</SCRIPT>
```

für Netscape bitte "Netscape" verwenden und entsprechende Version

1.3.1.2. Browsererkennung anhand der Unterscheidung browserinterner Objekte

Diese Art der Browsererkennung ist zuverlässig und verwendet nicht die Browserunterscheidung anhand der Browsernamen.

Internet Explorer und Netscape haben z.T. divergierende Objekte, die vom jeweiligen Konkurrenzprodukt nicht erkannt werden. Das erleichtert eine Browsererkennung per Javascript/JScript.

Andererseits ist im Falle ab Netscape 6.x eine Änderung gegenüber dessen Vorgängern bis 4.7x eingetreten (Netscape-Versionen 5.x existieren nicht). Es wurde mit z.T. Fähigkeiten des verbreiteten Netscape 4.7x konsequent gebrochen, was so mancher Internet-User nicht weiß, der zu dem eventuell noch hartnäckig auf "Netscape schwört". Netscape ab 6.x kann endlich Objekte abbilden, die der Internet Explorer ebenfalls kennt. Basis beider Browser ist die genormte Implementierung von HTML und Script anhand des "Dokument Objekt Modells" (DOM).

Beispiel:

```
// interne Variablen initialisieren
var Browser_Version_Haupt = 0;
var Browser_Version_Unter  = 0;

// Informationen zum Useragent holen
//          Objekt navigator.userAgent kennen IE und NS
var UserAgent  = navigator.userAgent;

// prüfen ob UserAgent keine Leerkette ist
if (UserAgent != '')
{
        // UserAgent ist keine Leerkette

        // Hauptversion des Browsers holen
        for (var i=0; i < 10; i++)          // Annahme: Eine Hauptversion höher 9 existiert noch nicht
        {
                eval(       'if (UserAgent.indexOf('"'      // Version steht vor dem Punkt
                        + i                                 //  i stellt die zu suchende Hauptversion dar
                        + '.") != -1)'                      //  gesucht wird Ziffernfolge mit **nachfolgendem** Punkt
                                                            //  i wird automatisch in Ziffernfolge umgewandelt
                        + '{Browser_Version_Haupt = '       //  wenn gefunden, so numerischen Wert von i
                                                            //          der Variablen zuweisen
                        + i
                        + ';}'
                    );
        }

        // Unterversion des Browsers holen
        for (var i=0; i < 100; i++)         // Annahme: Eine Unterversion höher 99 kann nicht existieren
        {
                eval(       'if (UserAgent.indexOf('.'      //  Unterversion steht nach dem Punkt
                        + i                                 //  i stellt die zu suchende Unterversion dar
                        + '") != -1)'                       //  gesucht wird Ziffernfolge mit **vorausgehendem** Punkt
                                                            //  i wird automatisch in Ziffernfolge umgewandelt
                        + '{Browser_Version_Unter = '       //  wenn gefunden, so numerischen Wert von i
                                                            //          der Variablen zuweisen
                        + i
                        + ';}'
                    );
        }

        // wenn Unterversion des Browsers am Ende eine Null hat, so diese eliminieren z.B. 50 zu 5
        if ((Browser_Version_Unter  % 10) == 0)
```


```
            {Browser_Version_Unter = Browser_Version_Unter / 10;}        // Bsp.: 50 % 10 = 5 Rest 0, also 50 / 10 = 5
}

// NS selektieren

var NSunter6=   (       (!document.all)                 // kein NS kennt document.all
                    || (document.layers)                // document.layers  nur bis NS unter 6.x
                );

var NSab6=      (           (!document.all)             // kein NS kennt document.all
                    && (document.getElementById)        // ab NS 6.x ist getElementByID implementiert

                    // theoretisch wäre zusätzlich    (!document.layers)    kodierbar,
                    //           aber mit Implementation von getElementByID wurde gleichzeitig
                    //           document.layers abgeschafft
                );

var NS      =   (NSunter6  ||  NSab6);

var NS4x    =   (           (NSunter6)
                    &&  (Browser_Version_Haupt >= 4)
                    &&  (Browser_Version_Haupt <   5)
                );

var NS6x    =   NSab6;

                // theoretisch kann noch auf die Hauptversionsnummer geprüft werden
                //      var NS6x =  (           (NSab6)
                //                          &&  (Browser_Version_Haupt >= 6)
                //                  );

// IE selektieren
var IE      =   (document.all) ? true : false;      // if-Anweisung ist wichtig, sonst erfolgt Zeigerzuweisung
                                                    // Objekt document.all ist in allen IE-Versionen verfügbar

var IEab5 =     (           (IE)
                    && (document.getElementById )       // ab IE 5.x ist getElementByID implementiert
                );

var IE4x    =   (           (IE)                        //  NS-Browser setzt IE       auf false
                    && (!IEab5 )                        //    damit setzt NS-Browser   IEab5 auf false,
                                                        //                             also !IEab5 auf true
                                                        // Es muss (IE) && (!IEab5) abgefragt werden !
                                                        //          Würde nur  (!IEab5) kodiert sein,
                                                        //          dann würde der NS-Browser
                                                        //          IE4x auf true setzen.
                    && (Browser_Version_Haupt >= 4)
                    && (Browser_Version_Haupt <   5)
                );

var IE5x    =   (           (IEab5 )
                    && (Browser_Version_Haupt >= 5)
                    && (Browser_Version_Haupt <   6)
                );

var IE55 =      (           (IEab5)
                    && (Browser_Version_Haupt == 5)
                    && (Browser_Version_Unter  == 5)
                );

var IE6x =      (           (IEab5)
                    && (Browser_Version_Haupt >= 5)
                );
```

1.3.1.3. Browsererkennung beim Internet Explorer ab IE 5.x

Es werden Tags benutzt, die nur der IE ab 5.x erkennt.
Beispiel 1:

```
<!-- [if IE 5] //-->
        javascript_ oder_html_code_fuer_den_IE5
<!-- [endif] // -->
```


```
<!-- [if ! IE 5] //-->
        javascript_ oder_html_code_fuer_andere_Browser      // auch alle IE-Versionen ungleich 5
<!-- [endif] //-->
```

Beispiel 2:

```
<!-- [if IE] //-->
        javascript_ oder_html_code_fuer_den_IE
<!-- [endif] // -->

<!-- [if ! IE] //-->
        javascript_ oder_html_code_fuer_andere_Browser      // aber keine IE-Versionen
<!-- [endif] //-->
```

Scriptcode muss innerhalb von <SCRIPT ...> ... </SCRIPT> kodiert werden.
HTML-Code kann nur kodiert werden, wenn obige Prüfung innerhalb von BODY liegt.

1.3.2. Schritt 2: Erkennung der Javascript-Version (browserhersteller-spezifisch)

Javascriptversionen -Versionen sind abwärtskompatibel.
JScript-Versionen sind abwärtskompatibel.
Javascript und JScript sind nur z.T. kompatibel.
Bestimmte Features des Browsers lassen sich nur mit bestimmter Script-Version bzw. beim IE manchmal nur mit bestimmter Scriptmaschinen-Version realisieren.

JavaScript 1.0	ab NS 2.x	ab IE 3.0
JavaScript 1.1	ab NS 3.x	ab IE 3.02 teilweise
JavaScript 1.2	NS 4.0 bis 4.05	ab IE 4.0
JavaScript 1.3	NS 4.06 bis 4.70	ab IE 5.0
JavaScript 1.4	NS 4.7x	ab IE 6.0
JavaScript 1.5	ab NS 6.x	steht noch aus

Beispiel:

Hinweise: Bitte keine var-Kodierung von Variablen, da in älteren Versionen eine var-Anweisung unbekannt ist.
Achtung: Kodierung ohne var bewirkt, dass die Variable global ist !

Für jede Scriptversionsprüfung muss ein eigenes <SCRIPT LANGUAGE ...></SCRIPT> kodiert werden.

```
<HEAD>
<SCRIPT>
        var JavscriptVersion = "unbekannt";
        var JScript = false;                    // Annahme: kein Microsoft

        function Anzeigen()
        {
                var Kette = "JScript: ";

                Kette += "Aktuelle Maschine = " + ScriptEngine();
                Kette += " mit Hauptversion "    + ScriptEngineMajorVersion();
                Kette += " mit Unterversion "    + ScriptEngineMinorVersion();
                Kette += " mit Buildnummer "     + ScriptEngineBuildVersion();

                alert(Kette);
        }
</SCRIPT>

<SCRIPT LANGUAGE="JavaScript">
        JavscriptVersion = "1.0";          // bitte keine var-Kodierung, da in älteren Versionen eine var-Anweisung unbekannt
ist
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript1.1">
        JavscriptVersion = "1.1";
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript1.2">
        JavscriptVersion = "1.2";
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
        JScript = true;                    // JScript kennt keine Sprachversion, dafür aber Versionen der Scriptmaschine,
                                           //          was Sinn macht, denn die Scriptmaschine bestimmt den Umfang
                                           //          von JScript und der Kompatibilität zum Javascript-Standard.
</SCRIPT>

<SCRIPT>
```


```
        if (JScript)                    // if-Anweisung     ist in allen Scriptversionen bekannt
        {Anzeigen();}                   // Funktionsaufruf ist in allen Scriptversionen möglich
        else
        {alert(JavscriptVersion);}      // alert()          ist in allen Scriptversionen bekannt
</SCRIPT>
</HEAD>
```

1.4. Feststellung der aktuellen JScript-Maschine

Es werden folgende Methoden verwendet, die **keine Objektreferenz** benötigen:

.ScriptEngine()	Sprache der gerade benutzten Scriptmaschine im Internet Explorer
	"JScript" Microsoft JScript
	"VBA" Microsoft Visual Basic for Applications
	"VBScript" Microsoft Visual Basic Scripting Edition
.ScriptEngineBuildVersion()	Buildnummer der gerade benutzten Scriptmaschine im Internet Explorer
.ScriptEngineMajorVersion()	Hauptversion der gerade benutzten Scriptmaschine im Internet Explorer
.ScriptEngineMinorVersion()	Unterversion der gerade benutzten Scriptmaschine im Internet Explorer

Beispiel:

```
function Anzeigen()
{
        var Kette = "";

        Kette += "Aktuelle Maschine = " + ScriptEngine();
        Kette += " mit Hauptversion "    +  ScriptEngineMajorVersion();
        Kette += " mit Unterversion "    + ScriptEngineMinorVersion();
        Kette += " mit Buildnummer "     + ScriptEngineBuildVersion();

        alert(Kette);
}
```

2. Javascript in HTML einbinden

2.1. Javascript-Direktkodierung in das HTML-Dokument

2.1.1. Javascript-Kodierung per HTML-Tag

Interpretation des HTML-Dokumentes von oben nach unten

Mit der Interpretation wird der Javascriptcode ausgeführt:

Javascript-Funktionen werden nur dann ausgeführt, wenn sie mit einer Argumentenliste "()" kodiert sind (egal ob volle oder leere Argumentenliste)

Kodierung einer Javascript-Funktion ohne "()" bewirkt Lieferung des **Zeigers** auf die Funktion und kein Funktionsaufruf

Javascriptcode innerhalb <HEAD> ... <HEAD> wird vor dem Javascriptcode innerhalb <BODY> ... </BODY> ausgeführt, da der HEAD-Abschnitt vor dem BODY-Abschnitt interpretiert wird.

Beispiel:

```
<HTML>
        <HEAD>
                .....
                <TITLE> ..... </TITLE>
                <SCRIPT>
                        // hier die Scriptanweisungen
                </SCRIPT>
        </HEAD>
        <BODY>
                <SCRIPT>
                ....
                </SCRIPT>
                <NOSCRIPT>  Text fuer Browser, die Script nicht kennen </NOSCRIPT>
                ....
        </BODY>
</HTML>
```

Empfehlung: LANGUAGE-Attribut in <SCRIPT ...> mitkodieren, wenn bestimmte Version verlangt wird

für Browser, der kein Script kennt, muss kodiert werden:

```
<SCRIPT ....>
<!--
```


```
        // hier die Scriptanweisungen
-->
</SCRIPT>
```

wobei <!-- ....--> der mehrzeilige Kommentarbegrenzer ist

```
<NOSCRIPT>
text
</NOSCRIPT>
```

wobei Text nur dann angezeigt wird, wenn der Browser das <NOSCRIPT>-Tag kennt

2.1.2. Javascript-Kodierung als Wert eines HTML-Attributes im HTML-Tag (Javascript-Entity)

jedes beliebiges HTML-Attribut
Syntax:

attribut="&{javascript_ausdruck};"

javascript_ausdruck muss den Werte des Attributes liefern (Typengleichheit !!)

Beispie für automatische Bildanpassung auf 25% der aktuellen Fensterbreite bei Netscape:

<IMG WIDTH="&{Math.ceil(self.innerWidth*0,25)};" ...>

2.1.3. Javascript-Kodierung als Aktion eines Eventhandlers im HTML-Tag (Javascript-Entity)

Syntax:

onXXX=**" javascript:**javascript_code**"**
onXXX=**' javascript:**javascript_code**'**

" " bzw **' '**	muss kodiert werden
on	Suffix für Eventhandler muss kodiert werden
javascript:	optional kodierbar
XXX	für Bezeichner des Events Gross-Klein egal
javascript_code	beliebig

Beispiel 1:

<BODY onload="**javascript:**alert('Hallo!');">

Beispiel 2:

```
<HTML>
<HEAD>
<SCRIPT>
        funktion Anzeige()
        {alert("Das Laden von BODY, also des Dokumentes, wurde soeben beendet !");}
</SCRIPT>
<BODY onload="javascript:Anzeige();">
</BODY>
</HTML>
```

2.1.4. Javascript-Kodierung als Aktion eines Eventhandlers im SCRIPT-Tag

ab IE 4.x
Das Script-Tag wird um die Attribute FOR und EVENT erweitert:

```
<SCRIPT  FOR=objekt_bezeichner  EVENT=event_bezeichner .....>
.....
</SCRIPT>
```

objekt_bezeichner	Standard-Objekt laut DOM z.B. window ID des instanzierten Objekt z.B. laut ID-Attribut Kodierung wahlweise mit oder ohne " " bzw. ' '
event_bezeichner	alle Events **mit** Präfix on z.B. onload siehe Objekt event Kodierung wahlweise mit oder ohne " " bzw. ' '

Ansonsten gilt das Übliche zum Script-Tag, also auch die Lage im HEAD und/oder BODY.

Beispiel:

```
<HTML>
<HEAD>
<SCRIPT  FOR=window
        EVENT=onload
>
        var Filter0 = ID_Div.filters[0];
        Filter0.Apply();
        ID_Div.innerHTML= "<IMG SRC='test.jpg'  WIDTH=300 HEIGHT=300>";
        Filter0.Play();
</SCRIPT>
</HEAD>
<BODY>
        <DIV    ID= "ID_Div"
                STYLE=  "position:absolute;width:300;height:300;top:20;left:20;
                         filter:revealTrans(transition=12,duration=8)
                        "
        >
                Dieser Text wird ueberblendet per Filter revealTrans (nur IE).
                <BR>
                Der Filter wird aktiv mit Laden des Dokumentes in das Fensters.
        </DIV>
</BODY>
</HTML>
```

2.1.5. Javascript-Kodierung zur Laufzeit des HTML-Dokumentes

Sämtlicher Scriptkode, der im Dokument abgearbeitet werden soll, muss **spätestens** zur Laufzeit und nicht unbedingt bereits beim **Laden** des HTML-Dokumentes vorliegen.

Mit Javascript ist der Programmierer in der Lage, Script mit speziellen Javascript-Anweisungen zu erzeugen, welche den erzeugten Scriptcode auch **sofort parsen und ausführen** und das auch **nach dem Laden** des HTML-Dokumentes, also zu seiner Laufzeit. Diese speziellen Anweisungen müssen allerdings bereits dann im Dokument vorliegen, wenn das Dokument geladen wird. Sie müssen also im Dokument kodiert und beim Laden bekannt sein. Der Aufruf dieser Methoden kann während oder nach dem Laden erfolgen.

Der Umfang des per Script zu erzeugenden Skriptcodes ist beliebig. Natürlich kann der zu erzeugende Scriptcode mit zu erzeugendem HTML-Code gemischt werden.

Im Falle der Erzeugung von HTML-Code ist es wichtig, mit welchen Methoden dieser erzeugt wird: Ob mit Methoden des HTML-DOM oder mit Methoden, die nicht HTML-DOM-konform sind (siehe weiter unten). Für die Anwendung letzterer Methoden im Internet Explorer ist unbedingt zu beachten, dass der HTML-Code bereits **während des Ladens** des HTML-Dokumentes, also **vor** der Auslösung des Ereignisses onload erzeugt werden muss (Begründung: siehe weiter unten).

Wenn der Programmierer den gesamten BODY-Teil des HTML-Dokumentes (inklusive der Tags <BODY> und </BODY>) per Script erzeugen will, so kann er dieses tun: Der Scriptcode muss dazu komplett im HEAD kodiert sein und wird mit dem Parsen des HEAD abgearbeitet, also vor dem Erreichen des Tags </HTML>. Natürlich wird ein hinter dem HEAD per BODY-Tag kodierter Body auch abgearbeitet, aber eben erst **nach** dem HEAD. Daher ist es empfehlenswert, entweder den Body nicht oder als leer zu kodieren (<BODY></BODY>).

Falls es der Browser zulässt, kann das **gesamte** HTML-Dokument **dynamisch erzeugt werden** und zwar inklusive der Werte zu Attributen von HTML-Elementen. Achtung: Ein User, der Javascript deaktiviert hat, wird dann nichts zusehen bekommen ! Im Übrigen ist das Feature, ein HTML-Dokument aus purem Script erstellen zu können, ein Analogon zu PHP, mit dem HTML-Code per PHP-Code erzeugt werden kann.

Skriptcode kann also

dynamisch durch Script **verwaltet** werden
sich auf Variablen, Funktionen und Objekte beziehen, die erst zur Laufzeit in Art und Umfang erzeugt werden sollen
kann HTML-Code dynamisch verwalten.

Es gibt diverse und z.T. browserspezifische Anweisungen, die Scriptcode zur Laufzeit erzeugen. Nachfolgend werden 3 wichtige und oft benutzte Anweisungen beschrieben, die **weder** browser- und nicht objektspezifisch **noch** Methoden des DOM sind. Diese 3 Methoden erzeugen einen Datenstrom, der im Falle von zu erzeugendem HTML-Code wie das Laden einer HTML-Datei wirkt (siehe weiter unten). Hinweis: Das Laden eines Dokumentes aus einer HTML-Datei kann nur über einen Datenstrom durch Auslesen der Datei realisiert werden.

Für Programmierer unter Windows XP mit dem Internet Explorer bitte **unbedingt** beachten: Die Scriptmaschine unter Windows XP ist wesentlich **weniger fehlertolerant** als die Scriptmaschine unter Windows 98 bei identischer Browserversion und identischem Patch-Stand der Browsersoftware. Unter Windows XP ist der Internet Explorer 6.x implementiert. JScript-Code, der unter Windows 98 einwandfrei funktioniert, muss es unter Windows XP **nicht** ! Javascript-Code, der unter Windows XP funktioniert, wird es auch unter Windows 98 tun.

Mit anderen Worten: Die Scriptmaschine unter Windows XP ist bezüglich Fehler **nicht** abwärtskompatibel ! Deswegen bitte **unbedingt** den Scriptcode unter Windows XP testen !

2.1.5.1. Methode eval()

Diese Methode parst und führt den **Javascriptcode** in der Zeichenkette (String oder Stringliteral) sofort aus.
Der Javascriptcode darf **keinen** HTML-Code enthalten.
wird im Kontext des kodierten eval() ausgeführt.

Hinweis: Die Methoden document.write() und document.writeln() können Script- **und** HTML-Code ausführen.

Syntax:

eval(zeichenkette_mit_beliebigem_javascript_inhalt);

Die Zeichenkette kann zuvor mit den üblichen Kettenoperationen gebildet werden.
Es ist möglich, auch Ausdrücke zu verwenden, die aber immer String liefern müssen.

Beispiel:

```
eval("var Feld = new Array();"); // Die Variable Feld wird instanziert !
```

2.1.5.2. Methoden document.write() und document.writeln()

Diese Methoden parsen und führen den **Javascriptcode** in der Zeichenkette (String oder Stringliteral) sofort aus.
Der Javascriptcode darf HTML-Code enthalten.
wird im Kontext des BODY-Teils vom **aktuellen** HTML-Dokument ausgeführt und zwar unabhängig davon,
wo die Anweisungen document.write() und document.writeln() kodiert sind
(für den Internet Explorer bitte **unbedingt** weiter unten lesen !).

Hinweis: Die Methode eval() kann nur Script- und **keinen** HTML-Code ausführen.

Syntax:

```
document.write(zeichenkette_mit_beliebigem_javascript_und_oder_html_inhalt);
document.writeln(zeichenkette_mit_beliebigem_javascript_und_oder_html_inhalt);
```

Die Zeichenkette kann zuvor mit den üblichen Kettenoperationen gebildet werden.
Es ist möglich, auch Ausdrücke zu verwenden, die aber immer String liefern müssen.

document.writeln() erzeugt automatisch ein
 (Zeilenumbruch)
Ein <BR> kann auch durch "\n" kodiert werden z.B. bei document.write()

Beispiel 1 für dynamisches Zeigerfeld auf OBJECT-Elemente:

```
var ObjektZeigerFeld = new Array();                    // dynamisches Feld

function ObjektErzeugen(ObjektNummer)
//  Die Objektnummer muss >=0 sein
//  eine lückenlose Vergabe ist nicht nötig
{
        //      externes Objekt einbinden, wobei der Name unbedingt mit der Objektnummer versehen wird
        document.write(     '<OBJECT ID="OBJECT_ID' + ObjektNummer.toString() + '"></OBJECT>');

        //      Zeiger vom externen Objekt im ObjektZeigerFeld[] merken
        document.write(     '<SCRIPT LANGUAGE="JavaScript1.2">'
                          + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                          + ']=document.OBJECT_ID' + ObjektNummer.toString() + ';\n'
                          + '</SCRIPT>'
                      );
}

for ( var i=0; i<5; i++)
{
        // Lücke lassen bei Index 2 bzw. 3 bzw. 4
        if (   (i==0)
            || (i==1)
            || (i==5)
          )
        {ObjektErzeugen(i);}            // ID-Attribut bekommt den Wert      "OBJECT_ID0"
                                        //                                   "OBJECT_ID01"
                                        //                                   "OBJECT_ID5"
}

Die Zeiger sind per   ObjektZeigerFeld[0]  bzw. OBJECT_ID0
                      ObjektZeigerFeld[1]  bzw. OBJECT_ID1
                      ObjektZeigerFeld[5]  bzw. OBJECT_ID5
```


ansprechbar.

Die Attribute der OBJECT-Tags sind noch anzupassen.

Beispiel 2 für dynamisches Zeigerfeld auf OBJECT-Elemente mit eigenen Script-Funktionen per Prototyping:

```
// Funktion als String
var RumpfCodeDerFunktion1 =         'if (Wert1 != 0)\n'
                                +   '{alert("Wert1 ist " + Wert1);}\n'
                                +   'else\n'
                                +   '{alert("Wert1 ist Null !");}\n';

// internes Zeigerfeld für Objekte
var ObjektZeigerFeld = new Array();                 // dynamisches Feld
                                                    // Es wird die Eigenschaft .prototype erzeugt

function ObjektErzeugen(ObjektNummer)
//  Die Objektnummer muss >=0 sein
//  eine lückenlose Vergabe ist nicht nötig
{
        //      externes Objekt einbinden, wobei der Name unbedingt mit der Objektnummer versehen wird
        document.write(     '<OBJECT ID="OBJECT_ID' + ObjektNummer.toString() + '"></OBJECT>');

        //      Zeiger vom externen Objekt im ObjektZeigerFeld[] merken
        document.write(     '<SCRIPT LANGUAGE="JavaScript1.2">'
                        + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                        + ']=document.OBJECT_ID' + ObjektNummer.toString() + ';\n'
                        + '</SCRIPT>'
                    );

        //      globale Kodierung der objekteigenen Methode mit einem objektinternen Bezeichner, der
        //              eindeutig die Objektzugehörigkeit anzeigt
        //      die Funktion wird zugleich instanziert und ist somit per Zeiger adressierbar
        document.write(     '<SCRIPT LANGUAGE="JavaScript1.2">'
                        + 'function OBJECT_ID' + ObjektNummer.toString() + '_Funktion1()\n'
                        + RumpfCodeDerFunktion1
                        + '</SCRIPT>'
                    );

        //  Erweiterung des Objektes per Prototyping um    die Funktion1 durch deren Zeiger
        //                                                 den Wert1, der zugleich instanziert und initialisiert wird
        document.write(     '<SCRIPT LANGUAGE="JavaScript1.2">'
                        + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                        + '].prototype.Funktion1= OBJECT_ID' + ObjektNummer.toString() + '_Funktion1\n'

                        + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                        + '].prototype.Wert1= 0;    // Initialisierung\n'
                        + '</SCRIPT>'
                    );
}
```

Beim Prototyping von Funktion1 bitte nicht () kodieren, da ja ein Zeiger übergeben werden soll !

Der Zugriff auf die Daten und Methoden des Objektes erfolgt über ObjektZeigerFeld[ObjektNummer].

```
z.B.    ObjektZeigerFeld[ObjektNummer].Funktion1();
        ObjektZeigerFeld[ObjektNummer].Wert1=33;
```

Die objekteigene Funktion1 kann natürlich auch Parameter haben.

Die Doppelbezeichnung ein und desselben Funktionscodes mit

```
        "Funktion1"
und     "OBJECT_IDx_Funktion1"          x ist die jeweilige Objektnummer
```

macht Sinn, sobald mehrere Instanzen des externen Objektes gebildet werden sollen, da somit alle Objekte den selben Funktionsbezeichner haben, aber intern namentlich verschiedene. **Und**: Die Objekte haben je ihre **eigene** Funktion, die von keinem anderen Objekt benutzt werden kann. Das Prinzip der Kapselung wurde somit erfüllt. Der Funktionsname ist für den Programmierer einheitlich. Durch den Bezug per

ObjektZeigerFeld[ObjektNummer].Funktion1
ObjektZeigerFeld[ObjektNummer]Wert1

wird die Kapselung aufrechterhalten.

Diese Methode kostet Ressourcen und bläht die HTML-Datei auf ! Daher ist es natürlich auch möglich, nur 1 Methode zu deklarieren und dann den Objekten den identischen Zeiger zuzuweisen. Wichtig ist dabei, dass die Objekte dann **nicht parallel** auf diese Funktion zugreifen dürfen, da die Funktion in keinem Objekt gekapselt ist !!!

Durch die Auslagerung des Code der Funktion1 in die Stringvariable

RumpfCodeDerFunktion1

wird Übersicht erzeugt. Man beachte, dass
" und ' sich nicht inkorrekt mischen !
die maximal mögliche Gesamtlänge des Strings beachtet werden muss, also eventuell mehrere Variablen zu kodieren sind.

Durch die Auslagerung der Variable

RumpfCode DerFunktion1

in eine eigene JS-Datei und deren Einbindung in das HTML-Dokument, kann der Aufbau der Funktion1, also des Objektes, von außen gesteuert werden, ohne das HTML-Dokument editieren zu müssen.

Empfehlenswert ist es, per \n am Zeilenende einen Zeilenumbruch zu erzeugen, damit die jeweilige erzeugte Zeile mit Ausführung von document.write() nicht unnötig zu lang wird, da Interpreter von Browsern nicht unbedingt überlange Zeilen ausführen können.

Beim Internet Explorer ist unbedingt zu beachten:

Die Ausführung von document.write() bzw. document.writeln(), das **HTML-Tags** erzeugt, hat 2 Konsequenzen und zwar genau dann, wenn diese Anweisung **nach dem kompletten Laden des Dokumentes, also nach Auslösung des Ereignisses onload** aktiviert wird:

Fakt 1:

Die **ERSTE** Erzeugung eines HTML-Tags bewirkt das **automatische Öffnen eines neuen** HTML-Dokumentes, wenn das aktuelle Dokument **bereits komplett geladen**, also das Ereignis onload **bereits** ausgelöst wurde. Letzteres ist immer dann der Fall, wenn der BODY-Teil des Dokumentes komplett geparst wurde. Grund: Ein komplett geparstes Dokument kann per write() bzw. writeln() nicht um HTML-Elemente verändert werden, da diese Methoden keine des HTML-DOM sind. Mit anderen Worten: Nur die Verwendung von Methoden des HTML-DOM lassen eine **nachträgliche** HTML-Elemente-Veränderung des Dokumentes zu.

Die Methoden write() und writeln() erzeugen einen Datenstrom aus HTML-Elementen und das neue Dokument empfängt diesen so, als würde er aus einer HTML-Datei stammen. Mit Ende des Datenstromes wird quasi ein Dateiende erkannt und das neue Dokument löst das Ereignis onload aus. Damit wird aber das neue Dokument zum aktuellen Dokument, also im Fenster über dem des alten Dokumentes angezeigt. Da das Fenster des neuen Dokumentes automatisch erzeugt wurde (nicht per Methode open()), sind also die Standards für eine Fenstererzeugung verwendet worden. Damit hat das neue Dokument einen History-Eintrag. Der User kann nun mit diesem zwischen dem alten und neuen Dokument umschalten.

Fakt 2:

Im Falle der o.g. nachträglichen Veränderung des Dokumentes um HTML-Elemente per wirte() bzw. writeln() kennt das neue, automatisch geöffnete Dokument das alte Dokument nur **als Eltern**. Es muss also im neuen Dokument mit dem Zeiger auf die Eltern gearbeitet werden, wenn Daten und Routinen der Eltern benutzt werden sollen (siehe Objekt window bzw. Objekt document). Mit anderen Worten: Das neue Dokument muss dann komplett per Script erzeugt werden, denn dieser Zeiger lässt sich nur über Script ansprechen. Jedes geladene Dokument hat ansonsten seine eigenständige Umgebung.

*2.2. Javascript-Kodierung in externer Datei *.js*

(Javascript-Code mehrfach verwenden)

ab Javascript 1.1
ab JScript, das Javascript 1.1 unterstützt

Dateiname: möglichst 8 Zeichen
alles klein
frei wählbar
Suffix *.js

Datei: ASCII
darf **nur** Javascript-Code und keinen HTML-Code enthalten, also auch kein <SCRIPT> </SCRIPT>

Mimetyp "text/javascript" muss mit dem Suffix von *.js verknüpft sein

Einbinden im HEAD des HTML-Dokument per eigenem <SCRIPT></SCRIPT>

```
<SCRIPT  LANGUAGE="....."
         TYPE="text/javascript"
         SRC="script_datei.js"
>
// alle andersartigen Javascript-Anweisungen werden hier vom Browser ignoriert !
</SCRIPT>
```

SRC: Dateiname auch mit Pfad möglich
mehrere SRC sind möglich --> Empfehlung: pro JS-Datei das eigene <SCRIPT> ..... </SCRIPT>

<SCRIPT> ... </SCRIPT> innerhalb </HEAD> .. </HEAD> und <BODY> ... </BODY> kodierbar

Bei Frames: Trotz gemeinsamer Quelle (js-Datei) wird der in allen Frames identische Bezug auf ein Javascript-Element immer im Kontext des Frame ausgeführt (pro Frame eigenständig),

Beispiel:
Inhalt von TEST.HTM:

```
<SCRIPT  LANGUAGE="....."
         TYPE="text/javascript"
         SRC="test.js"
>
         // alle andersartigen Javascript-Anweisungen werden hier vom Browser ignoriert !
         alert(TestWert); // wird nicht ausgeführt
</SCRIPT>

<SCRIPT>
         var Kette = "TestWert";
         alert(Kette + " " + TestWert);      // TestWert 10 wird angezeigt
</SCRIPT>
```

Inhalt von TEST.JS:

```
var TestWert=10;
```

Beispiel für Einbindung von Datensätze als Teil einer Javascript Datei (*.js):

daten.js

Pro Satz sind 3 Felder vorhanden, die mit Zeichenkettenwerten belegt werden.
Die Sätze werden in eine Array-Variablen abgelegt → Array-Indizierung ab 0 !!

Aufbau von daten.js:

```
var anzahl_saetze=3;                                   /* Anzahl stets ab 0, sonst beliebig */

function erzeuge_satz(satz_teil0, satz_teil1, satz_teil2)
                                                        /* Konstruktor zur satzweisen Feldbelegung */
{        this. satz_teil0=wert0;
         this. satz_teil1=wert1;
          this. satz_teil2=wert3;
}

 datensaetze_feld=new Array(anzahl_saetze);
datensaetze_feld[0]=new erzeuge_satz("0_1","0_2","0_3");
datensaetze_feld[1]=new erzeuge_satz("1_1","1_2","1_3");
datensaetze_feld[2]=new erzeuge_satz("2_1","2_2","2_3");
```

Einbindung von daten.js

Mit Einbinden der *.js-Datei wird der Javascript-Code SOFORT interpretiert. Damit werden alle Anweisungen außerhalb von Funktionen sofort ausgeführt.

```
<HTML>
<HEAD>
```


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

<SCRIPT LANGUAGE="JavaScript" >
<!--
        // Es wird im satz_teil0 eines jeden Satzes gesucht.
        // Das Stichwort ist der Suchbegriff.
        // Im Satz und im Suchbegriff können Gross -und kleinbuchstaben auftauchen.
        // Der Suchbegriff kann im satz_teil0 als Teilkette auftauchen, aber auch das gesamte Feld umfassen.

        function vergleich(satznr,suchwort)
        {
                suchwort_laenge=suchwort.length;                        /* Suchwort akt. Länge */
                datensatz_laenge=datensaetze[satznr].satz_teil0.length;
                                                                        /* Feld 0 im Satz: akt. Länge */
                if (datensatz_laenge >= suchwort_laenge)
                {
                        anzahl_pos= datensatz_laenge - suchwort_laenge;

                        for ( var pos=0; pos <= anzahl_pos; pos++)
                        {
                                teilkette =
                        datensaetze[satznr].satz_teil0.substring(pos,suchwort_laenge).toLowerCase();

                                if  (teilkette == suchwort.toLowerCase())
                                {return true;}          // gefunden  im satz_teil0 des Datensatzes
                        }
                }
                else {return false;}                            // nicht gefunden im satz_teil0 des Datensatzes
        }

        function suche(suchwort)
        {
                var gefunden=false;

                if (anzahl_saetze > 0)                                  // anzahl_saetze laut js-Datei !!
                {
                        var satzzahler=-1;
                        while   (       (!gefunden)
                                &&  (satznr <= anzahl_saetze)
                                )
                        {
                                satzzahler++;                   // um 1 erhöhen
                                gefunden= vergleich(satzzahler,suchwort);
                        }
                }

                return gefunden;
        }

        function starten()
        {
                suchwort=document.form_name.input_name.value;

                if (suchwort == "")
                {alert("Bitte Suchwort eingeben !");}
                else
                {
                        if (suche(suchwort))
                        {alert(" gefunden !");}
                        else
                        {alert("nicht gefunden!");}
                }
        }
//-->
</SCRIPT>
</HEAD>
<BODY>
        <FORM NAME="form_name" onSubmit="starten()">
                Bitte Suchbegriff eingeben:
                <INPUT NAME="input_name" TYPE=TEXT>             </INPUT>
                <INPUT TYPE=submit              VALUE="Suche starten"> </INPUT>
```


```
        </FORM>
</BODY>
</HTML>
```

2.3. Javascript und Browserperformance

Die Browser-Performance zur Darstellung des HTML-Dokumentes oder eines Objektes kann wie folgt erhöht werden:

In HTML: Wenn das ID-Attribut zum Tag zulässig ist, dann immer kodieren.
Wenn das NAME-Attribut zum Tag zulässig ist, dann immer kodieren.
Wenn NAME- **und** ID-Attribut zum Tag zulässig sind, dann ID-Attribut kodieren,
es sei denn, es wird eine Referenz über das NAME-Attribut benötigt.

In Script:

Bei Referenz auf eine Methode oder Eigenschaften eines Objektes sollte immer **zuerst** ein Zeiger auf das Objekt erzeugt werden, um **dann** mit diesem Zeiger die Methode bzw. Eigenschaft anzusprechen. Das gilt **vor allem** für Objekte, die Kinder haben, welche aufgerufen werden sollen. Je l**änger** die Punktnotation ist, um so mehr Aufwand hat der Browser beim Interpretieren, denn für jede Ebene in der Notation muss der Zeiger der Ebene berechnet werden.

Die Zeigerbildung schafft auch Übersichtlichkeit im Quellcode.

Bsp.:
```
var BodyObjekt = document.body;
BodyObjekt.eigenschaft = ....;

    // Die langsamere Variante ist
    //      document.body.eigenschaft =  ...;
```

Bsp. für IE:
```
var FeldDerZeigerAllerHTMLElementeImDokument = document.all;

var ZeigerAufHTMLObjekt =
        FeldDerZeigerAllerHTMLElementeImDokument.IDdesHTMLElementes;

ZeigerAufHTMLObjekt.eigenschaft  = ....;

    // ID des HTML-Elementes      z.B. laut ID-Attribut im HTML-Tag
```

Die Zeigerbildung ist auch dann sinnvoll, wenn das betroffene Objekt mehrmals im Scriptcode referenziert werden soll.

Bsp. für IE:
```
var FeldDerZeigerAllerHTMLElementeImDokument = document.all;

var FeldDerH1TagsImDokument = FeldDerZeigerAllerHTMLElementeImDokument.tags("H1");

var AnzahlDerH1TagsImDokument = FeldDerH1TagsImDokument.length;

for (var Zahler = 0; Zahler < AnzahlDerH1TagsImDokument; Zahler++)
{.............}

    // Die langsamere Variante ist
    //      for (var Zahler = 0; Zahler < document.all.tags("H1").length; Zahler++)
    //      {.............}

    //      pro Vergleich mit Zahler muss document.all.tags("H1").length immer neu
    //              berechnet werden
```

Ein Objekt sollte immer mit **absolutem Pfad** referenziert werden, der bereits vorher per Teilzeiger referenziert sein sollte, falls der Teilzeiger in Mehrfachkodierungen verwendet werden kann.

Bsp.:
```
var Kette = window.navigator.userAgent;   // Zeigers des Fensters explizit kodieren,
                                          //      also window als Pfadursprung

    // Die langsamere Variante ist
    //      var Kette = navigator.userAgent;
    //                  Zeiger auf window muss impliziert werden
```

2.4. Javascript- und HTML-Dateien auf dem Server

Javascript-Dateien sollten mit ihren zugehörigen Dokumenten auf dem Server in systematischer Struktur abgelegt sein, die dem Aufbau des HTML-Projektes (Web-Projektes) als Gesamtheit aller HTML-Komponenten entspricht. Die Struktur auf dem Server entspricht auch einer Baumstruktur wie im Dateisystem von Windows (nur dass möglichst Dateinamen in der 8.3-Kodierung analog zu DOS verwendet werden sollten z.B. t2345678.htm und nicht 1t345678.htm und nicht T2345678.htm und nicht TestDateiMitHTMLCode.htm).

2.4.1. robots.txt und Suchmaschinen

Die Scanvorgänge von Suchmaschinen lassen sich beeinflussen über

META-Tag
ASCII-Datei robots.txt

Der Nutzung von Meta-Tags ist die Anwendung von robots.txt vorzuziehen.

2.4.1.1. robots.txt und ihre Lage auf dem Server

NUR im Hauptverzeichnis des Webs auf dem Server

Bsp. www.test.de/robots.txt

Dateinamen klein schreiben
als Pfadtrenner nur / und nicht \ kodieren
sobald sich diese Datei auf dem Server befindet, ist sie wirksam

2.4.1.2. robots.txt die nicht auf dem Server vorhanden ist (Scan-Standard)

Existiert keine robots.txt im Hauptverzeichnis, so gilt folgender Scan-Standard:

JEDE Suchmaschine darf alles (auch Dateien etc.) abscannen.

2.4.1.3. robots.txt und Aufbau

2.4.1.3.1. robots.txt als Zeilenfolge-Script erstellen

als reine ASCII-Text, also z.B. mit Notepad
keine Formate etc. wie z.B. unter WORD !!

jede Zeile per Return-Taste-Eingabe beenden

Kodierung ist in ihrer nachfolgend gezeigten Groß-Kleinschreibung einzuhalten

2.4.1.3.2. robots.txt und ihre Blöcke

Block: Folge von Zeilen, die genau eine Scan-Variante beschreiben

beliebige Anzahl von Blöcken sind in robots.txt möglich

Trennung von Blöcken erfolgt nicht, da jeder neue Block an seiner 1. Zeile erkannt wird

2.4.1.3.2.1. robots.txt-Block: Aufbau Zeile 1 (Zulassen bzw. Sperren von Suchmaschinen)

Diese Zeile dient zum Ein- bzw. Ausschließen von Suchmaschinen für den Scan des HTML-Dokumentes

2.4.1.3.2.1.1. robots.txt und Zeile 1: ALLE Suchmaschinen dürfen scannen

user-agent: *

Hinweis: zwischen Doppelpunkt und * bzw. Name der Suchmaschine ist ein Leerzeichen zu kodieren

2.4.1.3.2.1.2. robots.txt und Zeile 1: KEINE Suchmaschine darf scannen

Diese Variante ist nicht kodierbar.

2.4.1.3.2.1.3. robots.txt und Zeile 1: Eine bestimmte Suchmaschinen darf scannen

user-agent: WebCrawler

WebCrawler darf scannen

Sollen mehrere Suchmaschinen ausgewählt werden, so muss pro Suchmaschine ein eigener Block mit Zeile 1 wie im Beispiel kodiert werden. Der Name der Suchmaschine muss dafür exakt bekannt sein.

Hinweis: zwischen Doppelpunkt und * bzw. Name der Suchmaschine ist ein Leerzeichen zu kodieren

2.4.1.3.2.1.4. robots.txt und Zeile 1: Eine bestimmte Suchmaschinen darf NICHT scannen

user-agent: WebCrawler

WebCrawler darf scannen, aber **ab Zeile 2** sind alle Inhalte als gesperrt zu markieren

Sollen mehrere Suchmaschinen ausgewählt werden, so muss pro Suchmaschine ein eigener Block mit Zeile 1 wie im Beispiel kodiert werden. Der Name der Suchmaschine muss dafür exakt bekannt sein.

Hinweis: zwischen Doppelpunkt und * bzw. Name der Suchmaschine ist ein Leerzeichen zu kodieren

2.4.1.3.2.2. robots.txt-Block: Aufbau ab Zeile 2 (Zulassen bzw. Sperren von Inhalten auf dem Server für Suchmaschinen, die scannen)

2.4.1.3.2.2.1. robots.txt und ab Zeile 2: Alle Daten auf dem Server dürfen gescannt werden

Disallow:

es muss **nur noch diese Zeile** kodiert werden

Daten: Haupt- und alle Unterverzeichnisse und alle HTML-Dateien mit deren Bildern etc.

Für folgende Fälle ist diese Variante nicht zulässig:

Verzeichnisse, die mit User-Passwort geschützt seien sollen
Verzeichnisse, die private Dinge des Web-Programmierers enthalten
ausgewählte Dateien wie HTML-Dateien die niemanden was angehen

2.4.1.3.2.2.2. robots.txt und ab Zeile 2: Alle Daten auf dem Server dürfen NICHT gescannt werden

Disallow: /

sperrt das Hauptverzeichnis und damit alle Unterverzeichnisse

Hinweis: zwischen Doppelpunkt und / ist ein Leerzeichen zu kodieren

2.4.1.3.2.2.3. robots.txt und ab Zeile 2: Bestimmte Daten auf dem Server dürfen NICHT gescannt werden

2.4.1.3.2.2.3.1. robots.txt und ab Zeile 2: Bestimmte Verzeichnisse auf dem Server dürfen NICHT gescannt werden

Disallow: /pfad_angabe/

pfad_abgabe: Pfadkodierung mit / für dasjenige Verzeichnis, das nicht gescannt werden darf
abschließendes / nicht vergessen !

pro Verzeichnis ist jeweils eine eigene Zeile zu kodieren

Hinweis: zwischen Doppelpunkt und dem ersten / ist ein Leerzeichen zu kodieren

2.4.1.3.2.2.3.2. robots.txt und ab Zeile 2: Bestimmte HTML-Dateien auf dem Server dürfen NICHT gescannt werden

Disallow: /pfad_angabe/datei_angabe

pfad_abgabe/dateiangabe: Pfadkodierung mit / für diejenige Datei, die nicht gescannt werden darf

pro Datei ist jeweils eine eigene Zeile zu kodieren

Hinweis: zwischen Doppelpunkt und dem ersten / ist ein Leerzeichen zu kodieren
Diese Kodierungsvariante ist wichtig für PHP-Scripte und Dateien, die Passwörter verwalten bzw. enthalten.
Sollte die Passwortverwaltung durch den Provider selbst angeboten werden, so prüfen, ob das vorgegebene Verzeichnis mit seinen Dateien vor dem Scannen geschützt sind !

2.4.1.3.2.2.4. robots.txt und ab Zeile 2: NUR bestimmte Daten auf dem Server dürfen gescannt werden

Diese Variante ist nicht kodierbar.

2.4.1.3.3. robots.txt und Beispiele

Beispiel: WebCrawler darf ALLES scannen
user-agent: WebCrawler
Disallow:

Beispiel: WebCrawler darf NICHTS scannen
user-agent: WebCrawler
Disallow: /

Beispiel: WebCrawler darf ALLES außer /test/ scannen
user-agent: WebCrawler
Disallow: /test/

Beispiel: WebCrawler darf ALLES außer /test/test.htm scannen
user-agent: WebCrawler
Disallow: /test/test.htm

Beispiel: ALLE Suchmaschinen dürfen ALLES scannen (robots.txt kann entfallen)
user-agent: *
Disallow:

Beispiel: ALLE Suchmaschinen dürfen NICHTS scannen
user-agent: *
Disallow: /

Beispiel: ALLE Suchmaschinen dürfen ALLES außer /temp/ scannen
user-agent: *
Disallow: /temp/

Beispiel: ALLE Suchmaschinen dürfen ALLES außer /test/test.htm scannen
user-agent: *
Disallow: /test/test.htm

Beispiel: ALLE Suchmaschinen dürfen ALLES außer /test/test.htm UND /temp/ scannen
user-agent: *
Disallow: /test/test.htm
Disallow: /temp/

2.4.2. .htaccess und .htpasswd und Passwortschutz

Der Schutz von Dateien in einem Ordner auf dem Server ist Sache des Inhabers des Web-Projektes **und nicht Sache** des Serverinhabers. Es steht dem Inhaber des Web-Projektes frei, **seine** Daten zu schützen.
FTP-Zugriffschutz:

Browser sind in der Lage, auch FTP-Ordner anzuzeigen und auf diese zuzugreifen. Manche Web-Anbieter nutzen Internet-Adressen mit ftp://.... **anstelle von** http://..... als öffentliche Systeme, die standardgemäß per FTP-Protokoll les- und schreibbar sind, wenn der Inhaber der FTP-Ordner keinen FTP-Schutz installiert hat. Existiert ein Zugriffschutz, so erscheint mit dem Versuch eines Zugriffes im Browser ein Login-Fenster, das diverse Kennungen verlangt. Diese Kennungen müssen zwischen dem User, der per FTP zugreifen will, und dem Inhaber der FTP-Ordner **vereinbart** sein.

Das Web-Projekt ist standardgemäß per FTP auch schreibbar (FTP-Upload auf den Server), wenn nicht ein FTP-Zugriffschutz vereinbart wurde: Für den Schutz des FTP-Upload auf den Server besitzt der Inhaber des Web-Projektes ein Passwort, dass dem Serverinhaber-Inhaber bekannt ist. Dieses Passwort liegt in einer Datenbank des Serverinhabers und wird mit der Passwort-Eingabe des Inhaber des Web-Projektes abgeglichen, sobald ein FTP-Zugriff versucht wird. Die Datenbank des Servers ist durch den Inhaber des Web-Projektes nicht einsehbar, da der Serverinhaber Komponenten benutzt, die zugleich ein Login-Interface demjenigen bereitstellt, der FTP aktivieren will. Diese Komponenten werden server-lokal abgearbeitet und gelangen nie auf den PC des FTP-Users (außer das Login-Interface).

Der FTP-Schutz hat nichts mit dem Passwortschutz per .htaccess und .htpasswd zu tun.

Passwort-Schutz per .htaccess und .htpasswd:

Es können Ordner des Web-Projektes derart geschützt werden, dass ein User nur einen **erlaubten Zugriff** auf Webseiten bekommt. Standardgemäß sind Webseiten immer lesbar, um auf den Client (User-PC) übertragbar zu sein, damit sie der User auch im Browser
anschauen kann. Mit Aktivierung des Passwortschutzes ist ein User-individueller Zugriff installiert, der auch für den Inhaber des Web-Projektes gilt. Der Inhaber kann pro User ein Passwort vergeben. Ein User **ohne Passwort** bekommt keinen Lesezugriff und kann damit den Inhalt des Ordners im Browser nicht anzeigen lassen.

Schreibzugriff auf Ordner kann nur per FTP erfolgen (siehe oben).

Um Dateien im Ordner auf dem Server vor unberechtigtem Zugriff zu schützen, müssen **pro zu schützenden Ordner** die Dateien.htaccess **und** .htpasswd mit ordnerspezifischen Inhalten hinterlegt werden., wobei der jeweilige Dateiname in Kleinbuchstaben **und mit führendem Punkt** kodiert sein muss. Des weiteren muss der Serverinhaber die Verarbeitung dieser Dateien durch den Server unterstützen. Sollte das der Fall sein **und** existieren beide Dateien im jeweiligen zu schützenden Ordner, so ist der Passwortschutz für diese Ordner sofort aktiv.

Die Dateien **.htaccess** und **.htpasswd** sind reine ASCII-Dateien und sollten z.B. nur mit NOTEPAD.EXE erstellt werden, wobei Formatierungen wie bei Word oder Wordpad und deutsche Umlaute und Sonderzeichen nicht verwendet werden dürfen.

Aufbau der Datei .htaccess

AuthType Basic
AuthName "freier Text**"**
AuthUserFile absoluter_pfad/Ordner/**.htpasswd**
<Limit GET POST PUT>
require user freier_username_1
....
require user freier_username_n
</Limit>

freier Text	als Hinweis muss in " " kodiert sein
absoluter_pfad	auf dem Server immer von der Root aus Bsp.:

/root/xx/yy/Ordner/**.htpasswd**

mit root als Platzhalter für die Wurzel der Baumstruktur, welche frei benannt sein kann
xx und yy als Platzhalter für frei benennbare Ordner im Pfad
Ordner als Platzhalter für den frei benennbaren und zu schützenden Ordner

Achtung: Gross-Klein wird unterschieden, also die Namen der Pfadelemente korrekt kodieren !

freier_username_1 bis freier_username_n pro Username genau 1 Zeile mit **require user**

z.B. otto_1 bis otto_n

.... als symbolischer und nicht zu kodierender Platzhalter für die User ab 2 bis n

Aufbau der Datei .htpasswd

freier_username_1**:**verschluesseltes_password
....

freier_username_n**:**verschluesseltes_password

pro User laut Zeilen **require user** freier_username_1
....
require user freier_username_n

in der Datei **.htaccess**

genau 1 Zeile in der Datei **.htpasswd**

z.B. otto1**:**Ho/87YN6YqP89

Doppelpunkt muss kodiert werden

.... als symbolischer und nicht zu kodierender Platzhalter für die User ab 2 bis n

Die Passwortverschlüsselung muss anhand einer standardisierten Formel erfolgen, die diverse Anbieter als online-bedienbares Programms anbieten (eventuell der Serverinhaber selbst):

z.B. http://www.linux-profis.de/php4
http://www.inch.com/info/tech/HOWTOS/htaccess/htpasswd.html

Es ist Sache des Inhabers des Web-Projektes, sich die Verschlüsselung zu besorgen.

3. Javascript-Elemente (Auswahl)

Kodierungsregeln:

Unterscheidung Groß von Klein für Anweisungen
Variablen
Objekte etc.

Es gibt KEINE vereinheitlichte Kodierungsregeln wie bei einer compilerbedingten Programmiersprache. Die Scriptmaschine als Javascript-Interpreter ist mehr oder weniger Fehlertolerant (je nach Hersteller).

Empfehlung: **erst** mit Kleinschreibung probieren, **dann** mit gemischter Kodierung versuchen.
Genaue Syntaxkenntnisse

Hinweis für Kodierung eines HTML-Tags:

keine Unterscheidung zwischen Groß und Klein
Empfehlung: TAG-Bezeichner groß
TAG-Attribute groß
vordefinierte Werte für Attribute klein

3.1. Javascript-Bezeichner

Kodierungsregeln:

max. 32 Zeichen
nur Buchstabe, Ziffer, Unterstrich
Groß und Klein werden unterschieden
wenn durch Programmierer erzeugt, so kann ein reservierter Bezeichner **mit** enthalten sein

3.2. Javascript-Kommentar

im HTML-Code:

<!-- .... --> bewirkt **nur**, dass Browser, der Script nicht kennt (also auch das SCRIPT-Tag nicht kennt), den Scriptcode als Kommentar überliest
mehrzeilig möglich

Beispiel:
```
<SCRIPT>
<!--
        function Test()
        {alert("Test");}
-->
</SCRIPT>
```

im Scriptcode:

```
// das ist ein einzeiliger Kommentar

/*   das ist
     ein ein- oder mehrzeiliger
     Kommentar
*/
```

Empfehlung: /* und */ immer untereinander kodieren
möglichst // verwenden anstelle /* */ also zeilenweise kommentieren, da jede Scriptanweisung möglichst ihre eigene Zeile bekommen sollte

Beispiele für Kombination von /* */ mit //

Beispiel 1:
```
<SCRIPT>
<!--
        function Test()      // Eine Funktion
        {alert("Test");}     /* Die Funktion
                                zeigt in der Alert-Box
                                den Text an */
-->
</SCRIPT>
```

Beispiel 2:
```
<SCRIPT>
<!--
        function Test()      // Eine Funktion
        {alert("Test")       //  ;}  Dieser Kommentar erzeugt Syntaxfehler
                             /* Die Funktion
                                zeigt in der Alert-Box
                                den Text an */
-->
</SCRIPT>
```

Beispiel 3:
```
<SCRIPT>
<!--
        function Test()      // Eine Funktion
        {alert("Test")       //  ;}  Dieser Kommentar erzeugt keinen Syntaxfehler
        ;}                   /* Die Funktion
                                zeigt in der Alert-Box
                                den Text an */
                             //  /* wird nicht erkannt
                             */  // erzeugt Syntaxfehler, da /* fehlt
-->
</SCRIPT>
```

Beispiel 4:
```
<SCRIPT>
<!--
/*
        function Test()      // Eine Funktion
        {alert("Test");}     /* Die Funktion
                                zeigt in der Alert-Box
                                den Text an */
                             */  // die Funktion wird nicht erkannt
-->
</SCRIPT>
```

siehe auch bedingtes Parsen nur beim Internet Explorer z.B. per @if
siehe auch Objekt comment

3.3. Datentypen

Der Datentyp legt fest, wie der Browser und der Programmierer
eine einer Instanz
den Wert der Instanz (z.B. Variable)
zu behandeln haben.

Es gibt Basis-Datentypen und Basis-Datenstrukturen, die in der Scriptmaschine implementiert sind.
Der Programmierer kann anhand dieser Basisgrößen neue Datentypen kreieren.

Hinweis: Script-Objekte, die Datentypen definieren, werden in den Beschreibungen zu den Objekten erklärt.

Die Deklaration einer Instanz ohne Datentyp kann erfolgen per var Instanz;
wobei mit der nächsten Wertzuweisung ein Daten-Typ automatisch zugewiesen und damit eine Adresse der Instanz erzeugt werden.

Eine Instanz ohne Datentyp
kann nur für eine Wertzuweisung auf die Instanz verwendet werden,
kann vom Browser nur als Platzhalter reserviert werden,
hat den Zeigerwert null (nicht numerisch 0).

Der Datentyp kann auch durch Kontextanalyse des Scriptcodes ermittelt z.B. beim Literal.

Ermittlung eines Basis-Datentyps bzw. einer Basis-Datenstruktur per typeof:

Syntax: typeof operand;

liefert String z.B.
"undefined"
"function"
"object"
"number"
"boolean"
"string"

Beispiel: typeof 42 ergibt "number"

Die Datentyp-Konvertierung erfolgt .B. während der Wertzuweisung :
z.T. automatisch anhand von objektübergreifenden Methoden, die allen Objekten innewohnen,
per Methoden zum Basis-Datentyp bzw. Basis-Datenstruktur,
bei vom Programmierer kreierten Datentypen nur durch passend-programmierte Routinen.

Die Datentyp-Konvertierung kann folgende Werte liefern:

NaN	Not a Number Wert wird geliefert bei numerischer Operation mit nicht-numerischen Wert z.B. Zeichenkette bei Konvertierung von String nach numerisch, wenn String nicht Zeichen laut number Datentyp enthält
Infinity	Wert wird geliefert bei numerischer Operation, dessen Wertebereich den von Script überschreitet
-Infinity	Wert wird geliefert bei numerischer Operation, dessen Wertebereich den von Script überschreitet

Datentypen die gleichzeitig numerisch sind:

Typ	entspricht numerisch numerischem Wert
null-Zeiger bzw. undefined	jeder (egal welcher)
true	> 0
false	0
Leerkette UND Kette nur aus Leerzeichen	0
// aber jede andere Kette	> 0

Hinweis: Browser meldet undefined, wenn null-Zeiger erkannt.

3.3.1. array Datentyp (Basis-Datenstruktur)

basiert auf dem Script-Objekt Array
z.B. verwendet für Feld aus Literalen

Syntax für Feldfüllung eines instanzierten Feldes:

Zeiger = [[liste_1] [..... , [liste_n]]	für mehrdimensionales Feld
Zeiger = [liste]	für eindimensionales Feld
Zeiger	**instanziertes** Objekt der Objektklasse Array, also Ableitung vom Script-Objekt Array per var Zeiger = new Array();

[ ] bedeutet hier **nicht** optional !!

[liste_1] bis [liste_n]	Liste aus kommagetrennten Elementen Liste stellt ein symbolisches Feld da
liste	Liste aus kommagetrennten Elementen
Listenelement:	kann Ausdruck sein, der Wert liefern muss kann entfallen, aber das trennende Komma muss kodiert werden Folge der Listenelemente entspricht Initialisierungsfolge des Feldes pro Komma eine automatische Indexerhöhung wenn Element nicht kodiert, so wird das Feldelement zum Index nicht initialisiert ---> Feldelement hat keinen Datentyp

Hinweise: Feldindex beginnt ab 0
Anzahl der Feldelemente: 0 so Feld leer
sonst ab 1

Beispiele für ein instanziertes und leeres Feld mit dem Bezeichner "Feld":

```
Feld = [1,2,3];      Element an Index 0 hat Wert 1
                     Element an Index 1 hat Wert 2
                     Element an Index 2 hat Wert 3

                     Anzahl der Elemente 3

Feld = [1,,,,5];     Initialisiert wird nur  mit Index 0   auf Wert 1
                                             mit Index 4   auf Wert 5
                     Anzahl der Elemente 2
```

Feld = [["Name", "Tom", "Tim", "Teo"], ["Alter", 6, 5, 4]];
Ein Feld, das aus genau 1 numerischen Wert besteht
darf nicht wie folgt deklariert werden:

```
var t new Array(33);  // 33 als Wert ist falsch, sondern hier als Anzahl der Feldelemente

var t new Array()
t[0]=33;              // ist korrekt
```

```
var t=new Array();

alert(t)      zeigt nichts an
alert(!= null) zeigt true an
```

Beispiele für lokaler Feldzeiger

```
var X_Feld=new Array();
...

function test();
{var X00=X_Feld;          // erzeugt Zeiger als KOPIE
 var X01=X00[0];          // lesen aus Kopie
 var X01=12;                    // Schreiben in Kopie und NICHT in X_Feld
X_Feld[0]=0;              // Schreiben in Feld  und NICHT in Kopie
}
```

Beispiele für Mehrfachindex eines Feldes

```
var X_Feld=new Array();
X_Feld[0]=new Array();          // Feldelement 0 ist selbst Feld (Unterfeld)
X_Feld[0][0]=10;                // Feldelement 0 als Unterfeld: Unterfeld-Element 0 belegen
```

Beispiele für null-Zuweisung

Variable deklariert ohne Wert: Wert ist undefiniert
Zuweisung dieser Variablen auf eine andere: Diese andere wird ebenfalls "undefined".

```
var t;              // undefined
var s=new Array();
```


```
s[0]=1;          // mit Wert
s[0]=t;          // undefined --> ES WIRD null-Zeiger gebildet so dass  s[0] != null    false ergibt
                 // das Feldelement bleibt aber erhalten, so dass .length nicht verändert wird
```

3.3.2. boolean Datentyp (Basis-Datentyp)

basiert auf dem Script-Objekt Boolean
hat Werte mit ausschließlich folgender Schreibweise **true**
false

also nicht 0
nicht 1
nicht TRUE
nicht True
nicht "true" etc. (siehe Script-Objekt Boolean)

Beispiel: if (10 > 20) ....

3.3.3. date Datentyp (Basis-Datenstruktur)

basiert auf dem Script-Objekt Date
siehe dort

3.3.4. function Datentyp (Basis-Datenstruktur)

basiert auf dem Script-Objekt Function
siehe Script-Objekt und Funktion

3.3.5. Literal Datentyp (Basis-Datentyp)

Wertkonstante, die **keiner** Variablendeklaration bedarf

Literal immer in ' ' bzw. " " kodieren:

\'	Entwertung des Zeichen '	z.B. '\''	bedeutet das Literal '
\"	Entwertung des Zeichen "	z.B. "\""	bedeutet das Literal "

" bzw. "" als leeres Literal möglich

Zeichen kann 16-Bit-Unicode sein (also auch Zeichensatz ab 256)

ASCII: pro Zeichen 7 Bit verwendet, also 128 Zeichen möglich (0-127)
enthalten im Unicode
Unicode: pro Zeichen 16 Bit verwendet, also 65535 Zeichen möglich
Zeichen 0-127 identisch mit ASCII-Code

Kodierung als ESC-Sequenz \uxxxx mit xxxx als Hexaziffern

Beispiele:

\u0008	Rückschritt
\u0009	horizontaler Tabulator
\u000A	Zeilenvorschub (Line Feed) neue Zeile
\u000B	vertikaler Tabulator
\u000C	Seitenvorschub (Form Feed)
\u000D	für Wagenrücklauf
\u0020	Blank, Leerzeichen
\u0022	"
\u0027	'
\u005C	\

Escape Sequenzen werden immer in Literal konvertiert:

\b	Backspace, Rückschritt
\f	Form feed, Blattvorschub
\n	Line feed (newline), Zeilenvorschub
\r	Carriage return , Wagenrücklauf (nicht Enter !!)
\t	Horizontal tab (Ctrl-I), horizontaler Tabulator
\'	Entwertung des Zeichen '
\"	Entwertung des Zeichen "
\\	Backslash z.B. für lokale Pfadangabe wie C:\\test\\bilder\\test.gif.
\\\\	Entwertung Backslash
\xhh	hexadezimal aber nur von 00 bis FF
\uhhhh	Unicode

Beispiele:

\u0008	Rückschritt
\u0009	horizontaler Tabulator
\u000A	Zeilenvorschub (Line Feed) neue Zeile
\u000B	vertikaler Tabulator
\u000C	Seitenvorschub (Form Feed)
\u000D	für Wagenrücklauf

\u0020	Blank, Leerzeichen
\u0022	"
\u0027	'
\u005C	\

Ganzzahl: dezimal, hexa, oktal

Bsp.:

'18'	dezimal
'0x12'	hexa
'022'	oktal

ab Javascript 1.5 im Netscape 6.x ist oktal deprecated

Fließkomma: IMMER mit Dezimalpunkt

Bsp.: '.12e34'
'5E-5.'

boolean: 'true' oder 'false'

null ist der null-Zeiger also nicht 0

3.3.6. number Datentyp (Basis-Datentyp)

basiert auf dem Script-Objekt Number

Integer: Ganzzahl immer 32 Bit breit
Zahlenbasis dezimal
oktal
hexadezimal

oktal nur Ziffern 0 bis 7
erste Ziffer muss 0 sein
auch negativ

hexadezimal Ziffern 0 bis 9 sowie Buchstaben A bis F (Gross-klein egal)
ersten zwei Zeichen müssen sein 0X oder 0x
auch negativ

Floating point: Gleitkommazahl, immer 64 Bit breit
Exponential-Darstellung möglich
Tausender-Trenner ist Komma
Dezimalkomma ist Punkt

NaN Not a Number
Wert wird geliefert bei numerischer Operation mit nicht-numerischen Wert z.B. Zeichenkette

Infinity Wert wird geliefert bei numerischer Operation, dessen Wertebereich den von Script überschreitet

-Infinity Wert wird geliefert bei numerischer Operation, dessen Wertebereich den von Script überschreitet

Beispiele:

oktal 0377

hexadezimal
0x37CF
0x37cF
0x37Cf
0x37cf
0X37CF
0X37cF
0X37Cf
0X37cf

Floating point

0.0001 kann auch kodiert werden als 00.0001
.0001,
1e-4,
1.0e-4

3.45e2

3.3.7. string Datentyp (Basis-Datenstruktur)

basiert auf den Script-Objekt String

Zeichen kann 16-Bit-Unicode sein (also auch Zeichensatz ab 256)
ASCII: pro Zeichen 7 Bit verwendet, also 128 Zeichen möglich (0-127)

enthalten im Unicode
Unicode: pro Zeichen 16 Bit verwendet, also 65535 Zeichen möglich
Zeichen 0-127 identisch mit ASCII-Code

Kettenlänge nur begrenzt aufgrund Vorgaben des Betriebssystems/Browsers
Bsp.: Netscape-Browser hat max. 256 Zeichen pro Zeile

Eine Zeichenkette kann im Quellcode **nicht** über mehrere Zeilen gehen !
Es sind die Methoden des Stringobjektes oder der Stringoperator + zu verwenden.

Leerkette ist "" oder '' (Zeichen " bzw. ' zweimal kodiert)

Escape Sequenzen nicht zulässig, sondern immer als Literal kodieren per Verkettungs-Operation

\b	Backspace, Rückschritt
\f	Form feed, Blattvorschub
\n	Line feed (newline), Zeilenvorschub
\r	Carriage return , Wagenrücklauf (nicht Enter !!)
\t	Horizontal tab (Ctrl-I), horizontaler Tabulator
\'	Entwertung des Zeichen '
\"	Entwertung des Zeichen "
\\	Backslash z.B. für lokale Pfadangabe wie C:\\test\\bilder\\test.gif.
\\\\	Entwertung Backslash
\xhh	hexadezimal aber nur von 00 bis FF
\uhhhh	Unicode

Beispiele:

\u0008	Rückschritt
\u0009	horizontaler Tabulator
\u000A	Zeilenvorschub (Line Feed) neue Zeile
\u000B	vertikaler Tabulator
\u000C	Seitenvorschub (Form Feed)
\u000D	für Wagenrücklauf
\u0020	Blank, Leerzeichen
\u0022	"
\u0027	'
\u005C	\

3.3.8. Objekttyp oder Objektklasse (Basis-Datenstruktur)

vordefiniert per Script-Objekt Object
vom Programmierer frei definiert als Konstruktor zum Erzeugen eines Objektes per new Anweisung:
Konstruktor ist der Bezeichner z.B. des Script-Objektes (auch Script-Objekt Objekt)

3.3.9. Zeigertyp (Basis-Datentyp)

Zeigerverwendung

Sämtliche Objekte und Instanzen werden über Zeiger referenziert. Selbst ein Literal, das Ergebnis eines Ausdruckes, ein Funktionsargument oder eine Methode bzw. Eigenschaft eines Objektes erhalten je einen Zeiger.

Nur über Zeiger sind auch Adressbereiche im Hauptspeicher ansprechbar, wobei die Zeiger natürlich in Zeiger laut Speicherverwaltung zum Betriebssystem automatisch umgewandelt werden.

Beispiel für Verwaltung der Zeiger bei dynamischen HTML-Objekten:

Dynamische HTML-Objekte werden zur Laufzeit gebildet. Dazu muss die Methode document.write() verwendet werden. Des weiteren ist es möglich, bei mehreren Objekten gleicher Art ein Feld als Sammlung aller Zeiger zu verwenden.

```
// internes Zeigerfeld für Objekte
var ObjektZeigerFeld = new Array();                    // dynamisches Feld

function ObjektErzeugen(ObjektNummer)
//  Die Objektnummer muss >=0 sein
//  eine lückenlose Vergabe ist nicht nötig.
{
        //          Objekt erzeugen, wobei der Name unbedingt mit der Objektnummer versehen werden muss
        document.write(         '<OBJECT ID="OBJECT_ID' + ObjektNummer.toString() + '" '></OBJECT>);

        //          und dessen Zeiger merken in ObjektZeigerFeld[]
        document.write(         '<SCRIPT LANGUAGE="JavaScript1.2">'
                                    + 'ObjektZeigerFeld[' + ObjektNummer
                                    + ']=document.OBJECT_ID' + Index.toString() + ';'
                                + '</SCRIPT>'
                        );
}
```


Zeigererzeugung im Browser (interner Zeiger)

Der Browser erzeugt Zeiger nicht immer automatisch:

Z.B. ist ein Funktionsargument im Kopf der Funktion ein Platzhalter für den Parameter laut Funktionsaufruf.

Aber **ohne** kodierten Platzhalter weiß der Browser nicht, wie er den Parameter zuordnen soll, weil kein interner Zeiger gebildet werden konnte.

Zeiger, die vom Browser erzeugt werden, stehen immer im Kontext zum per Zeiger referenzierten Objekt bzw. zur Instanz:

Die Gültigkeit der Zeiger ist die der Gültigkeit des Objektes bzw. der Instanz.

Es besteht die Möglichkeit, über Script auf vom Browser erzeugte (interne) Zeiger zuzugreifen.

Zeigerzuweisung

Beispiel 1

```
<HTML>
<HEAD>
<script>
    function test(X0)
    // X0 Zeiger auf ein Objekt
    {
                var X1=X0.style;
                var X2='color';
                var X3;                         // Zeiger auf X0.style.color
                var X4;                         // Nullzeiger
                var X5='white';

                alert('Eigenschaft X0.style.' + X2 + ' vorhanden ? ' + (X0.style.color !=null));          // true
                alert('Eigenschaft X1.'       + X2 + ' vorhanden ? ' + (X1.color !=null));              // true

                X1.color='black';
                alert('Colorwert == black ? ' + (X1.color=='black'));

                eval('X3=X1.' + X2 + ';'); // liefert Zeiger von X0.style.color
                alert('eval von X3 erfolgt, also X3==X0.style.color ? ' + (X3==X0.style.color));          // true

                if (X3!=null)
                {
                        X4=X3;
                        alert('Zuweisung X4=X3 erfolgt, also X4==X0.style.color ? ' + (X4==X0.style.color));    // true
                        alert('Zuweisung X4=X3 erfolgt, also X4==X1.color ? ' + (X4==X1.color));                // true
                        alert('X4 == black ? ' + (X4 == 'black'));                                              // true

                }

                // ABER:
                //      Wird ein Zeiger mit einem NICHT-Zeiger überschrieben, so wird der Zeiger ebenfalls zum Nicht-Zeiger.
                //              Grund: Javacript passt das Ziel der Quelle an auch mit Typwechsel der Zeiger-Variablen
                //                      von Zeiger auf Nicht-Zeiger
                //      X4=X5; ist eben NICHT das belegen von X0.style.color  SOMDERN die Umwandlung vom Zeiger X4 in
einen
                //                              String !!!!
    }
</script>
</HEAD>
<BODY>
<DIV ID="TEST">
Test-DIV
</DIV>
<script>
test(TEST);
</script>
<BODY>
```

Beispiel 2

Es gibt diverse Ausnahmen: Objekt wie z.B. Style.

Da das Style-Element z.B. visibility über einen Zeiger verfügen muss, damit es einen Wert ablegen kann ab der Adresse laut Zeiger,

darf die Wertzuweisung nicht den Zeiger löschen ! Dafür sorgt das Objekt selbst.

```
function test(X00,X01,X02)
// X00 Zeiger auf Feld, das Objektzeiger enhält
// X01 Style-Eigenschaft z.B. 'visibility', kein ':' kodieren, darf NICHT Leerkette sein
// X02 Stylewert der Eigenschaft laut X01
//                      muss synatxgerecht sein
//                      Bsp.: X01 ist 'visibility'
//                                  X02 ist 'hidden'
//                      wenn Leerkette so Eigenschaft im Wert gelöscht:
//                                                  Da Eigenschaften öfters einen Standardwert haben, muss getestet werden,
//                                                  was mit dem Löschen passiert, also ob Standardwert wieder aktiv wird !
//          kein ':' kodieren
{
        var X03=0;              // Länge
        var X04=0;              // Zähler
        var X05;                // Zeiger auf Style

        // Länge von Feld der Objektzeiger
        X03=X00.length;
        if (X03>0)
        {
                // prüfen ob Style-Eigenschaft nicht leer ist
                if (X01 !='')
                {
                        // Felder der Objektzeiger abklappern
                        for (X04=0; X04 < X03; X04++)
                        {
                                // Stylezeiger holen
                                X05=X00[X04].style;
                                // style ist nicht komplett belegbar mit Stringwerten:
                                //      wenn X05=X00[X04].style dann bewirkt X05=X02; die
                                //              Umwandlung von X05 ist String wie X02
                                //              und nicht X00[04].style=X2
                                //      wenn X05=X00[X04] dann bewirkt X05.style=X02; die
                                //              Umwandlung der Eigenschaft Style
                                //              von Zeiger nach String wie X02
                                //              o dass alert(X05.style); einen Scriptfehler bringt,
da
                                //              tyle nicht existent
                                //              (Mitglied nicht gefunden: style muss Zeiger sein,
                                //              kein String)
                                // Hinweis: style ist nur durch style eines vorhandenen Objektes ersetzbar, also
                                //      durch kopieren der style-Zeiger, wobei das vorhandene Objekt
                                //      NICHT gelöscht  werden darf: Vorhandenes Objekt als
permanentes
                                //      unsichtbares Vorlageobjekt ist möglich, da trotz Zeigerindentität
im
                                //      style sich das ID vom  Vorlageobjekt und das ID vom Objekt mit
                                //      identischem style-Zeiger unterscheiden
                                // Zeiger der Syle-Eigenschaft holen und Wert zu weisen:
                                //      Das Style-Objekt sorgt dafür, dass mit Zuweisung des Wertes
                                //      NICHT die Style-Eigenschaft von Zeiger zu String wird, sondern
der
                                //      String als Wert ab Adresse laut Zeiger abgelegt wird.
                                //      Ein alert auf die Style-Eigenschaft zeigt NICHT [object] an
sondern
                                //      immer den Wert der Style-Eigenschaft.
                                eval('X05.' + X01 + '=' + X02 + ';');
                                // Variablen, deren Inhalte kein String sind, müssen als 'variablen_bezeichner'
                                //      kodiert werden.
                                // Variablen, deren Inhalt String sind, dürfen NICHT als 'variablen_bezeichner'
                                //      kodiert werden, sondern nur als variablen_bezeichner.
                        }
                }
        }
}
```

Zeigerkopie

Jeder Zeiger kann in eine Zeigervariable kopiert werden. Ist er gespeichert, so hat die Zeigervariable nicht unbedingt mehr mit dem Kontext des Objektes bzw. der Instanz zu tun. Es ist Sache der Programmierers, darauf zu achten.

Zeiger in HTML

Der Programmierer kann in HTML zu einem Tag durch die Kodierung der Attribute ID und/oder NAME einen Zeiger als Referenz auf diesen Tag erzeugen. Diese Referenz ist direkt in Script als Objektbezeichner per Punktnotation verwendbar.

Zeiger und Browserperformance

Die Browser-Performance zur Darstellung des HTML-Dokumentes oder eines Objektes kann wie folgt erhöht werden:

In HTML: Wenn das ID-Attribut zum Tag zulässig ist, dann immer kodieren.
Wenn das NAME-Attribut zum Tag zulässig ist, dann immer kodieren.
Wenn NAME- **und** ID-Attribut zum Tag zulässig sind, dann ID-Attribut kodieren,
es sei denn, es wird eine Referenz über das NAME-Attribut benötigt.

In Script: Bei Referenz auf eine Methode oder Eigenschaften eines Objektes sollte immer **zuerst** ein Zeiger auf das Objekt erzeugt werden, um **dann** mit diesem Zeiger die Methode bzw. Eigenschaft anzusprechen. Das gilt **vor allem** für Objekte, die Kinder haben, welche aufgerufen werden sollen. Die Zeigerbildung ist natürlich nur dann sinnvoll, wenn das betroffene Objekt mehrmals im Scriptcode referenziert werden soll. Die Zeigerbildung erspart dem Browser die mehrmalige Berechnung des Zeigers. Je l**änger** die Punktnotation ist, um so mehr Aufwand hat der Browser beim Parsen, denn für jede Ebene in der Notation muss der Zeiger der Ebene berechnet werden. Die Zeigerbildung schafft auch Übersichtlichkeit im Quellcode.

Bsp.:
```
var BodyObjekt = document.body;
BodyObjekt.eigenschaft = ....;
und nicht document.body.eigenschaft =  ...;

var FeldDerZeigerImDokument = document.all;
FeldDerZeigerImDokument.objekt_bezeichner.eigenschaft  = ....;
```

Null-Zeiger

Das Literal **null** ist der null-Zeiger, also nicht numerisch 0 und nicht das Zeichen "0".

typisierter Zeiger

Dieser Zeiger ist natürlich ebenfalls eine Referenz. Aber der Zeiger **muss** auf eine Größe zeigen, die einen **bestimmten** Datentyp hat. Typisches Beispiel ist der Aufruf einer Funktion:

Der Aufruf einer Funktion darf nur dort stattfinden, wo eine Referenz zulässig ist, da der Funktionsbezeichner einen Zeiger repräsentiert. z.B. ist ein Funktionsaufruf innerhalb eines Literals nicht möglich.

Falls die Funktion etwas liefert, gilt zusätzlich: Anstelle des Funktionsaufrufes wird die gelieferte Größe gesetzt.
Damit **muss** die Funktion auch **datentyp-gerecht** liefern (entspricht einem typisierten Zeiger).

Analog gilt das auch für den Aufruf einer Methode.

Typischer Aufruf erfolgt z.B. innerhalb eines Ausdruckes:

Beispiel:
```
function Test(Wert1, Wert2)
{return ( Wert1 + Wert 2);}

var Wert = 20 + Test(10,20);

alert(Wert.toString());

// Nachfolgender Aufruf ergibt einen Fehler, da die Funktion einen numerischen Wert liefert
// Die Methode alert() kann eventuell automatisch nach String konvertieren.
alert( "Das Ergebnis lautet " + Test(10,20));
```

Zeiger auf den Konstruktor

Die Eigenschaft .constructor liefert den Bezeichner einer JScript-Objektklasse (Objekttyp) oder eines privaten Konstruktors.

Anwendung: Ermittlung der Objektklasse/Konstruktors eines abgeleiteten Objektes.

Ableitung aus der Objektklasse
per Anweisung new mit der Objektklasse als Konstruktor benötigt (siehe dort)

nicht bei Script-Objekt Math möglich

Ableitung aus privatem Konstruktor
per Anweisung new, die den privaten Konstruktor verwendet

Syntax:
[var Wert =] object.constructor

Wert Bezeichner, der nicht innerhalb " " bzw. ' ' liegt

JScript-Objektklassen sind z.B.

Array
Boolean
Date
Function
Number
String

Bezeichner eines privaten Konstruktors

object Zeiger auf ein per abgeleitetes Objekt

Beispiel 1 für Ableitung aus einem JScript-Objekt:

```
var Kette = new String("Hi");
alert( (Kette.constructor == String));          // liefert "true"
alert( (Kette.constructor == "String"));        // liefert "false"
```

Beispiel 2 für Ableitung anhand privaten Konstruktors:

```
function TestFunktion()
{alert("Hallo");}

var ZeigerAufFunktion = new TestFunktion(); // bewirkt Ausführung von TestFunktion() also auch von alert()

// ZeigerAufFunktion();          // nicht möglich und bringt Fehlermeldung wegen fehlernder Instanz,
                                 //          da keine Ableitung vom JScript-Objekt Function
                                 // Kodierung ohne ()  bringt keinen Fehler, da als Variablendeklaration erkannt

alert(ZeigerAufFunktion.constructor == TestFunktion);          // true
alert(TestFunktion.constructor == TestFunktion);       // false
```

3.3.10. frei definierbarer Datentyp anhand Script-Objekt Object

Das Script-Objekt Object, stellt sozusagen den Prototyp aller anderen Objekte dar. Das Script-Objekt Object hat im Gegensatz zu anderen Script-Objekten wie Array oder Date keinen speziellen Datentyp implementiert. Die new Anweisung liefert also einen untypisierten Zeiger. Das Script-Objekt Object ist also für den Programmierer ein **symbolisches** Objekt, das als Objektklasse (Konstruktor) per new Anweisung für die Erzeugung eines einfachen Objektes benutzt werden kann, um es per Prototyping mit gewünschten Eigenschaften und Methoden zu erweitern und damit Datentypen beliebiger Art zu implementieren. Das Objekt ist also ideal für die Erzeugung eigener und freier Datenstrukturen anhand der Basis-Datentypen und Basis-Datenstrukturen.

3.4. Variable

Komponenten der Variable

Eine Variable besteht aus den Komponenten

Namen (Bezeichner) der Variablen, den der Programmierer z.T. selbst festlegen kann,
und Inhalt (Wert) der Variablen, den der Programmierer z.T. selbst festlegen kann.

Der Name der Variablen ist der **Platzhalter** für den Zeiger, der erst zur Laufzeit gebildet wird und der dann auf die Instanz der Variablen im Speicher zeigt. Anhand des Variablennamens kann der Programmierer auf die Instanz zugreifen und z.B. der Variablen einen Wert zuweisen.

Vor dem Variablenzugriff (außer erste Wertzuweisung) muss die Variable initialisiert sein, damit der Datentyp feststeht.

Variable deklariert ohne Wert: Wert ist undefiniert
Browser meldet undefined, wenn null-Zeiger erkannt.

Zuweisung dieser Variablen auf eine andere: diese andere wird ebenfalls "undefined".

Beispiel:
```
var t;                          // undefined
var s=new Array();
```


```
                                    s[0]=1;             // mit Wert
                                    s[0]=t;             // undefined --> ES WIRD null-Zeiger gebildet so dass  s[0] !=
null
                                                        //          false ergibt
                                                        // das Feldelement bleibt aber erhalten, so dass .length nicht
                                                        //          verändert wird
```

Globale Variablen durch Funktion verändern

Zuweisung des null-Zeigers hebt nicht die var-Deklaration auf !

```
<HTML><HEAD>
<SCRIPT LANGUAGE="JScript">
var d2760=1;            // global

function test()
{alert(d2760);          // 1
 d2760=null;
 alert(d2760);          // null
                        // Zuweisung des null-Zeigers hebt nicht die var-Deklaration auf !
 d2760=0;
 alert(d2760);          // 0
}

</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JScript">
test();                                 // 1  null  0
alert(d2760);           // 0
</SCRIPT>
</BODY>
</HTML>
```

Datentyp einer Variable

Ein Datentyp einer Variablen kann bei der Deklaration der Variablen **nicht kodiert** werden.
Javascript ermitteln den Datentyp automatisch mit der ersten Wertzuweisung auf die Variable.
Ist der Datentyp einmal festgelegt, so ist er nicht mehr änderbar.
Datentyp-Konvertierung möglich.

Deklaration einer Variable

Es sollte zur Deklaration immer var kodiert werden (var-Anweisung).
Eine ohne var deklarierte Variable ist immer global !
Beispiel: Eine ohne var deklarierte Variable innerhalb einer Funktion ist
nicht funktionslokal
sondern global !
Eine fehlende var-Anweisung bewirkt die Überschreibung einer bereits
als global deklarierten Variablen
durch eine Variable und deren Wert, die mit gleichem Namen ohne var-Anweisung
später deklariert wird.

Ab Javascript 1.2
ist es Pflicht, die var-Anweisung zu verwenden
sollten eine Variablendeklaration **ihre eigene** Quelltextzeile besitzen und **nicht** mehrere Variablendeklarationen in einer gemeinsamen Zeile kodiert sein (Vermeidung der kommagetrennter Deklaration von Variablen gleichen Typs)

Deklaration ansonsten sonst je nach Variablenart, wobei eine Initialisierung nicht erfolgen muss:
Eine deklarierte aber nicht initialisierte Variable hat den Wert undefined und den Zeigerwert null.
Bsp.:

```
if (Wert == null)           // if liefert true
var Wert;                   // Variable deklariert aber ohne Wert, also Wert undefined automatisch zugewiesen
if (Wert == undefined)      // if liefert true
if (Wert == null)           // if liefert true
var Wert=10;                // Variable deklariert aber mit Wert
if (Wert == undefined)      // if liefert false
if (Wert == null            // if liefert false
```

Vor dem Variablenzugriff (außer erste Wertzuweisung) muss die Variable initialisiert sein, damit der Datentyp feststeht.

var-Anweisung basiert auf dem gleichnamigen Javascript-Objekt "var"

Globale Variablen durch Funktion verändern

Zuweisung des null-Zeigers hebt nicht die var-Deklaration auf !

```
<HTML><HEAD>
<SCRIPT LANGUAGE="JScript">
var d2760=1;                    // global

function test()
{alert(d2760);                  // 1
 d2760=null;
 alert(d2760);                  // null
                                // Zuweisung des null-Zeigers hebt nicht die var-Deklaration auf !
 d2760=0;
 alert(d2760);                  // 0
}

</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JScript">
test();                                 // 1  null  0
alert(d2760);                   // 0
</SCRIPT>
</BODY>
</HTML>
```

lokaler Feldzeiger

```
var X_Feld=new Array();
...

function test();
{var X00=X_Feld;                // erzeugt Zeiger als KOPIE
 var X01=X00[0];                // lesen aus Kopie
 var X01=12;                            // Schreiben in Kopie und NICHT in X_Feld
X_Feld[0]=0;                    // Schreiben in Feld  und NICHT in Kopie
}
```

Instanzierung einer Variablen

durch parsen der Deklaration der Variablen
es wird immer ein Zeiger erzeugt, denn die Variable liegt zur Laufzeit im Speicher und besitzt einen Zeiger:

Zeiger wird repräsentiert durch den Variablennamen (Variablenbezeichner)
Zeiger kann auf die Abfrage der Variablenexistenz verwendet werden:

```
Bsp:    var variable=0;         //  numerische Variable instanzieren
        if (variable) {...}
oder    if (variable != null) {...}
```

Eine deklarierte aber **nicht** initialisierte Variable hat den Wert **undefined** und den Zeigerwert **null**.

Bsp.:

```
if (Wert == null)               // if liefert true
var Wert;                       // Variable deklariert aber ohne Wert, also Wert undefined automatisch zugewiesen
if (Wert == undefined)          // if liefert true
if (Wert == null)               // if liefert true
var Wert=10;                    // Variable deklariert aber mit Wert
if (Wert == undefined)          // if liefert false
if (Wert == null                // if liefert false
```

Vor dem Variablenzugriff (außer erste Wertzuweisung) muss die Variable initialisiert sein, damit der Datentyp feststeht.

Referenzierung einer Variable

immer über den Zeiger laut Instanzierung, also über den Variablennamen (Variablenbezeichner)

Eine deklarierte aber **nicht** initialisierte Variable hat den Wert **undefined** und den Zeigerwert **null**.

Bsp.:

```
if (Wert == null)              // if liefert true
var Wert;                      // Variable deklariert aber ohne Wert, also Wert undefined automatisch zugewiesen
if (Wert == undefined)         // if liefert true
if (Wert == null)              // if liefert true
var Wert=10;                   // Variable deklariert aber mit Wert
if (Wert == undefined)         // if liefert false
if (Wert == null               // if liefert false
```

Vor dem Variablenzugriff (außer erste Wertzuweisung) muss die Variable initialisiert sein, damit der Datentyp feststeht.

Löschung einer Variable

Die Zuweisung des Wertes **null,** also eines Null-Zeigers, bewirkt die Löschung der Variablen (inklusive Speicherplatz)

```
Bsp:    var variable=0;     //  numerische Variable instanzieren
        variable=null;      //  Zeiger löschen, also Instanz aufheben
```

Es kann auch die Anweisung delete verwendet werden, um das gesamte Objekt oder Teile davon zu löschen (inklusive Speicherplatz)

```
Bsp.:   var Wert = delete objekt_instanz.eigenschaft;

                Wert    true,   so Löschung erfolgreich
                        false,  so Löschung nicht erfolgt
```

Script-Objekte sind nicht löschbar
Nur per Prototyping zum Script-Objekt hinzugefügte Eigenschaften und Methoden sind löschbar.

Gültigkeit einer Variable

immer laut Kontext der Kodierung zur Variablen: lokal oder global

Eine ohne var deklarierte Variable ist immer global !
Beispiel: Eine ohne var deklarierte Variable innerhalb einer Funktion ist nicht funktionslokal sondern global !
Eine fehlende var-Anweisung bewirkt die Überschreibung einer bereits als global deklarierten Variablen durch eine Variable und deren Wert, die mit gleichem Namen ohne var-Anweisung später deklariert wird.

lokal: wenn innerhalb einer Funktion **und mit** var-Anweisung deklariert
global: wenn auf der obersten Ebene im Quellcode
wenn nicht innerhalb einer Funktion
wenn **ohne** var-Anweisung deklariert

Beispiel:

```
<SCRIPT>
    var DasIstEineGlobaleVariable = "Hallo";            // oberste Ebene im Quellcode
    var Kette = " und ich";

    function Test()
    {
        var DasIstEineLokaleVariable = ' Du';

        DasIstKeineLokaleVariableWeilVARfehlt = ' !';

        alert(      DasIstEineGlobaleVariable
                  + DasIstEineLokaleVariable
                  + DasIstKeineLokaleVariableWeilVARfehlt
             );

        var Kette = " und wir";         // überschreibt nicht die globale Variable
                                        // trotz Namensgleichheit, sondern ist lokal

    }

    // nachfolgende Anweisung bringt Fehlermeldung
    alert(      DasIstEineGlobaleVariable
              + DasIstEineLokaleVariable
              + DasIstKeineLokaleVariableWeilVARfehlt
         );
```


```
                    // nachfolgende Anweisung bringt keine Fehlermeldung
                    alert(      DasIstEineGlobaleVariable
                              + DasIstKeineLokaleVariableWeilVARfehlt
                        );

                    Kette = " und wir";              // überschreibt die globale Variable
                </SCRIPT>
```

Fehlermeldungen

Es ist darauf zu achten, dass der Browser die HTML-Dokumente nicht aus dem Browser-Cache liest, solange die Dokumente nicht fehlerfrei laufen, also beim Test der Dokumente (Daten im Browser-Cache vor jedem Test löschen).

Das generelle Lesen aus dem Cache ist per Script, META-Tag, sowie per Browsereinstellung abänderbar. Letztere kann der User treffen, in dem er z.B. beim Internet Explorer die Cache-Löschung mit Schließen des Browser abhakt.

Es ist üblich, dass der User den Browser-Cache **nicht löscht**, da der Browser den Cache-Füllstand automatisch verwalten kann. Daher muss der Programmierer damit rechnen, dass die HTML-Dokumente aus dem Cache geladen werden könnten. Deshalb ist es wichtig, dass der Programmierer der Webseiten dafür sorgt, dass per META-Tag immer ein Datum der Webseite besteht, das mit Sicherheit verfallen ist und somit die HTML-Daten vom Server geladen werden. Achtung: Des erneute Laden einer Webseite (Reload), z.B. innerhalb eines bestimmten Zeitraumes, sollte aus dem Cache kommen, damit nicht Daten doppelt zu laden sind, wenn sie bereits gültig im Cache liegen. Will der Programmierer absolut sichergehen, dass nach einem bestimmten Zeitraum die Daten vom Server zu laden sind **und** soll der User dieses Verhalten nicht beeinflussen können, dann muss das Verhalten per Script programmiert werden (Algorithmus ist Sache des Programmierers). Zugleich wird der User und der Server mehr Traffic beim Laden der Daten der HTML-Dokumente haben.

Fehlermeldung beim Internet Explorer zu Objekten:

z.B. "Objekt ..... nicht gefunden" oder " Objekt ... unterstützt Eigenschaft nicht" oder "CLASS nicht gefunden"

Die Fehler, die per Fehlermeldungen zu Objekten im Internet Explorer angezeigt werden, können dessen im Speicher instanzierte Objektstruktur durcheinander bringen, so dass nach Abhilfe des Fehlers im Quelltext und anschließender Aktualisierung des Dokumentes (per Aktualisierungs-Button im Browsermenü) trotzdem dieselbe Fehlermeldung erscheint. Es ist daher ratsam, bei solchen Fehlermeldungen den Browser zu schließen, den Browser-Cache zu löschen und dann den Browser mit dem Quelltext neu zu starten. Ursache ist ein eventuelles automatisches aber fehlerhaftes Prototyping von Objekten (auch von browserinternen). Hinweis: Man sollte immer mit genau 1 Instanz des Browsers testen, damit Objektstrukturen konsistent bleiben können.

Fehlermeldungen zum Script

z.B. "Scriptfehler ...."

Fehlermeldung kann echten Fehler anzeigen (z.B. wegen falscher Browserversion) **oder** Speichermangel, welcher die Abarbeitung des Skriptes verhindert (nicht mögliche Instanzierung von Objekten, Variablen etc.). Bei Speichermangel kann der Neustart des Browsers helfen.

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser leider **nicht als fehlerhaft** erkannt werden:

```
Bsp.:   var Kette =   'test'; + '.jpg';     // kein Syntaxerror wegen dem Semikolon vor dem Zeichen +
        alert(Kette);                       // zeigt nur  test   an  und nicht  test.jpg

Bsp:    for (var i = 0; i <3; i++);         // kein Syntaxerror wegen dem Semikolon vor dem Zeichen {
        {alert(i);}                         // genau eine Anzeige, die 3 anzeigt
```

3.4.1. nicht Objektvariable (Variable nicht per new erzeugt)

```
var bezeichner_liste [= wert_oder_ausdruck];
var bezeichner = (bedingung) ? wert_oder_audruck1 : wert_oder_audruck2;
                                        //entspricht        if (bedingung)
                                        //                  {bezeichner = wert_oder_ausdruck1;}
                                        //                  else
                                        //                  {bezeichner = wert_oder_audruck2;}
                                        //          liefert wert_oder_audruck1 wenn bedingung true liefert
                                        //                  wert_oder_audruck2 wenn bedingung false liefert
```

bezeichner_liste: Kommatrennung

Bezeichner: beginnt mit Buchstabe oder Unterstrich
sonst Buchstabe, Ziffer, Unterstrich

wert_oder_ausdruck: optionale Variablen-Initialisierung
Wert für Zeichenketten-Variable ist ein Literal, also in " " oder ‘ ‘ kodieren

auch Kombination von Literalen, Variablen
Variablentyp: definiert mit Typ des Wertes bzw. des Ausdruckergebnisses

= Zuweisungsoperator

var var Anweisung

3.4.2. Objektvariable

Eine Objektvariable ist eine Variable, die anhand eines Objektes erzeugt wird und mit dem Namen der Variablen einen Zeiger auf ein Objekt, also eine Objektinstanz im Speicher repräsentiert.
siehe auch Datentypen und Objektbeschreibungen

Objektklasse bzw. Objekttyp

Das Objekt, welches zur Deklaration der Objektvariablen verwendet wird, kann auch als **Objektklasse** oder **Objekttyp** bezeichnet werden. Die Objektklasse dient also zur Ermittlung des Typs der Objektvariablen. Z.B. können sind alle Javascript-Objekte als Objektklasse verwendet werden.

Der Programmierer kann unter Javascript eigene Objektklassen definieren. Das geht aber nur, in dem der Konstruktor der new-Anweisung eine Funktion darstellt. Diese Funktion erzeugt das komplette Objekt. In der Funktion können natürlich auch Objektvariablen auftauchen, die sich
auf **andere** Objektklassen beziehen (auch private).

Beispiel für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
<HTML>
<SCRIPT  LANGUAGE="JavaScript">
<!--
 function datenstruktur_ausgeben()
 {
        with (document)
        {       write(person.vorname + " " + person.nachname + "<BR>");
                write(person.strasse    + " " + person.nummer    + "<BR>");
                write(person.plz        + " " + person.ort       + "<BR>");
                //      Erika Mustermann
                //      Musterstrasse 1
                //      .....
        }

        for (i in person)
        {       document.write(i + ": " + person[i] + "<BR>");
                //      vorname: Erika
                //      nachname: Mustermann
                //      ...........
        }
 }

 function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
 {
        this.vorname = vorname;
        this.nachname = nachname;
        this.strasse = strasse;
        this.nummer = nummer;
        this.plz = plz;
        this.ort = ort;
        this.methode = methode;
 }

 person = new datenstruktur_erzeugen
                (       "Erika",                // Vorname
                        "Mustermann",           // Nachnahme
                        "Musterstrasse",        // Strasse
                        "1",                    // Nummer
                        "10000",                // PLZ
                        "Musterstadt",          // Ort
                        datenstruktur_ausgeben  // ohne () kodieren
                );

 // alternativ auch kodierbar:
 // var person = {       vorname:"Erika",        // Vorname
```


```
//                      nachname:"Mustermann",      // Nachnahme
//                      strasse:"Musterstrasse",    // Strasse
//                      nummer:"1",                 // Nummer
//                      plz:"10000",                // PLZ
//                      ort:"Musterstadt"           // Ort
//                      methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
//              };

// Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !

alert(person.vorname + "\n" + person.methode);      // person.methode ohne () kodieren !
// -->
</SCRIPT>
</HTML>
```

vordefinierte und browserinterne Objekte

Javascript-Objekte und die Objekte des Browsers sind in der Scriptmaschine vordefiniert, also implementiert. Sie können als Objektklasse verwendet werden.

private Objekte

Der Programmierer kann **private Objekte** per new-Anweisung definieren. Die Scriptmaschine kann nur dann private Objekte verarbeiten, wenn diese auf vordefinierten Objekten basieren und/oder der Programmierer sämtliche Eigenschaften und Methoden zum Objekt programmiert, wobei diese Eigenschaften und Methoden letztendlich **immer** auf vordefinierten Objekten und vordefinierten Datentypen basieren müssen (siehe auch Datentypen und Objektbeschreibungen). Der Programmierer kann unter Javascript eigene Objektklassen definieren.

Für private Objekte, die nicht aus einem vordefinierten Objekt abgeleiter werden, wird leider nicht die Eigenschaft .prototype erzeugt und ist somit nicht verwendbar !

Beispiel für Erweiterung des Script-Objektes Array, das die Eigenschaft .prototype besitzt:

```
function MaximumErmitteln( )
{
        var max = this[0];      // this referenziert die Objekt-Instanz, in dem die Methode vorhanden ist,
                                //              also Variable Feld

        for (var i = 1; i < this.length; i++)
        {
                if (max < this[i])
                {max = this[i];}
        }

        return max;
}

// neue Methode per Prototyping hinzufügen zum JScript-Objekt Array
Array.prototype.NeueArrayMethode = MaximumErmitteln;        // ohne () kodieren !

// Feld vom Array-Typ erzeugen mit der Methode .NeueArrayMethode()
var Feld = new Array(1, 2, 3, 4, 5, 6);         // 6 numerische Elemente

// neue Methode des JScript-Objektes Array aufrufen
alert(Feld.NeueArrayMethode());
```

Beispiel für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
<HTML>
<SCRIPT  LANGUAGE="JavaScript">
<!--
 function datenstruktur_ausgeben()
{
        with (document)
        {       write(person.vorname + " " + person.nachname + "<BR>");
                write(person.strasse   + " " + person.nummer   + "<BR>");
                write(person.plz       + " " + person.ort      + "<BR>");
                //            Erika Mustermann
                //            Musterstrasse 1
                //            .....
```


```
		}

		for (i in person)
		{	document.write(i + ": " + person[i] + "<BR>");
			//	vorname: Erika
			//	nachname: Mustermann
			//	.....
		}
}

function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
{
		this.vorname = vorname;
		this.nachname = nachname;
		this.strasse = strasse;
		this.nummer = nummer;
		this.plz = plz;
		this.ort = ort;
		this.methode = methode;
}

person = new datenstruktur_erzeugen
			(	"Erika",			// Vorname
				"Mustermann",			// Nachnahme
				"Musterstrasse",		// Strasse
				"1",				// Nummer
				"10000",			// PLZ
				"Musterstadt",			// Ort
				datenstruktur_ausgeben		// ohne () kodieren
			);

// alternativ auch kodierbar:
// var person = {	vorname:"Erika",		// Vorname
//			nachname:"Mustermann",		// Nachnahme
//			strasse:"Musterstrasse",	// Strasse
//			nummer:"1",			// Nummer
//			plz:"10000",			// PLZ
//			ort:"Musterstadt"		// Ort
//			methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
//		};

// Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !

alert(person.vorname + "\n" + person.methode);		// person.methode ohne () kodieren !
// -->
</SCRIPT>
</HTML>
```

instanzierte Objekte

Objekte, die bereits instanziert sind, können als Objektklasse verwendet werden. **Nicht alle** vordefinierten Objektklassen sind bereits instanziert (siehe auch Datentypen und Objektbeschreibungen).

Deklaration Objektvariable (Objektvariable von einer Objektklasse ableiten)

Eine Objektvariable muss speziell deklariert werden, je nach dem, um welche Objektklasse es sich handelt **und** ob diese bereits instanziert ist oder nicht.

Wenn die Objektklasse **nicht bereits** instanziert ist, so muss die var-Anweisung **in Verbindung** mit der Anweisung new bzw. per Literal kodiert werden. Die Objektklasse gilt dabei als Konstruktor in der new-Anweisung.

Wenn die Objektklasse **bereits** instanziert ist, so muss die var-Anweisung **ohne** new-Anweisung verwendet werden. Es reicht eine

Zeigerzuweisung analog zu einer Nicht-Objekt-Variablen aus.

Es sollte **immer** eine Zeigerzuweisung programmiert werden, um den Zeiger weiterzuverwenden:

```
Bsp:		document.all.objekt.eigenschaft  // document.all ist instanziertes Browser-Objekt
ersetzen durch	var Zeiger = document.all.objekt.eigenschaft;
```

Der Vorteil: Der Browser muss genau einmal die Objekthierarchie des Dokumentes abfragen, also nur

während der Zeigerbelegung. Danach ist die Zeigerverwendung ohne weitere Abrage der Objekthierarchie möglich, was die Perfomance des Browsers erhöht.

Erweiterung Objektvariable (Prototyping)

Eine instanzierte Objektvariable kann per Eigenschaft .protoype um neue Eigenschaften und Methoden erweitert werden, wenn die Objektklasse die Eigenschaft .prototype besitzt.

Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
Nur für Objekte, die aus vordefinierten Objekten abgeleitet werden, wird die Eigenschaft .prototype erzeugt.

Nachfolgende Beispiele zeigen, wie ein externes Objekt mit seinem Datum und 1 Methode dynamisch zur Laufzeit des HTML-Dokumentes verwaltet wird, wobei die Kapselung der Daten im Objekt erfolgt. Wichtig ist es, dass der Javascript-Code innerhalb <HEAD> ..</HEAD> und außerhalb einer Funktion kodiert sein muss, da der Code mit **Laden** des HTML-Dokumentes und **vor** der Abarbeitung von <BODY ... </BODY> interpretiert werden muss.

Beispiel für Erweiterung des Script-Objektes Array, das die Eigenschaft .prototype besitzt:

```
function MaximumErmitteln( )
{
        var max = this[0];      // this referenziert die Objekt-Instanz, in dem die Methode vorhanden ist,
                                //              also Variable Feld

        for (var i = 1; i < this.length; i++)
        {
                if (max < this[i])
                {max = this[i];}
        }

        return max;
}

// neue Methode per Prototyping hinzufügen zum JScript-Objekt Array
Array.prototype.NeueArrayMethode = MaximumErmitteln;          // ohne () kodieren !

// Feld vom Array-Typ erzeugen mit der Methode .NeueArrayMethode()
var Feld = new Array(1, 2, 3, 4, 5, 6);          // 6 numerische Elemente

// neue Methode des JScript-Objektes Array aufrufen
alert(Feld.NeueArrayMethode());
```

Beispiel für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
<HTML>
<SCRIPT  LANGUAGE="JavaScript">
<!--
function datenstruktur_ausgeben()
{
        with (document)
        {       write(person.vorname + " " + person.nachname + "<BR>");
                write(person.strasse    + " " + person.nummer    + "<BR>");
                write(person.plz        + " " + person.ort       + "<BR>");
                //            Erika Mustermann
                //            Musterstrasse 1
                //            .....
        }

        for (i in person)
        {       document.write(i + ": " + person[i] + "<BR>");
                //            vorname: Erika
                //            nachname: Mustermann
                //            ..........
        }
}

function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
{
        this.vorname = vorname;
        this.nachname = nachname;
```


```
                this.strasse = strasse;
                this.nummer = nummer;
                this.plz = plz;
                this.ort = ort;
                this.methode = methode;
        }

        person = new datenstruktur_erzeugen
                        (       "Erika",                        // Vorname
                                "Mustermann",                   // Nachnahme
                                "Musterstrasse",                // Strasse
                                "1",                            // Nummer
                                "10000",                        // PLZ
                                "Musterstadt",                  // Ort
                                datenstruktur_ausgeben          // ohne () kodieren
                        );

        // alternativ auch kodierbar:
        // var person = {       vorname:"Erika",                // Vorname
        //                      nachname:"Mustermann",          // Nachnahme
        //                      strasse:"Musterstrasse",        // Strasse
        //                      nummer:"1",                     // Nummer
        //                      plz:"10000",                    // PLZ
        //                      ort:"Musterstadt"               // Ort
        //                      methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
        //              };

        // Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !

        alert(person.vorname + "\n" + person.methode);          // person.methode ohne () kodieren !
        // -->
        </SCRIPT>
        </HTML>
```

Beispiel für dynamisches Zeigerfeld:

```
var RumpfCodeDerFunktion1 =             'if (Wert1 != 0)\n'
                                +       '{alert("Wert1 ist " + Wert1);}\n'
                                +       'else\n'
                                +       '{alert("Wert1 ist Null !");}\n';

// internes Zeigerfeld für Objekte
var ObjektZeigerFeld = new Array();                     // dynamisches Feld
                                                        // Es wird die Eigenschaft .prototype erzeugt !

function ObjektErzeugen(ObjektNummer)
//  Die Objektnummer muss >=0 sein
//  eine lückenlose Vergabe ist nicht nötig.
{
        //      externes Objekt einbinden, wobei der Name unbedingt mit der Objektnummer versehen wird
        document.write(         '<OBJECT ID="OBJECT_ID' + ObjektNummer.toString() + '"></OBJECT>');

        //      Zeiger vom externen Objekt im ObjektZeigerFeld[] merken
        document.write(         '<SCRIPT LANGUAGE="JavaScript1.2">'
                                + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                                + ']=document.OBJECT_ID' + ObjektNummer.toString() + ';\n'
                                + '</SCRIPT>'
                        );

        //      globale Kodierung der objekteigenen Methode mit einem objektinternen Bezeichner, der
        //              eindeutig die Objektzugehörigkeit anzeigt
        //      die Funktion wird zugleich instanziert und ist somit per Zeiger adressierbar
        document.write(         '<SCRIPT LANGUAGE="JavaScript1.2">'
                                + 'function OBJECT_ID' + ObjektNummer.toString() + '_Funktion1()\n'
                                + RumpfCodeDerFunktion1
                                + '</SCRIPT>'
                        );

        //  Erweiterung des Objektes per Prototyping um        die Funktion1 durch deren Zeiger
        //                                                     den Wert1, der zugleich instanziert und initialisiert wird
        document.write(         '<SCRIPT LANGUAGE="JavaScript1.2">'
```


```
                + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                + '].prototype.Funktion1= OBJECT_ID' + ObjektNummer.toString() + '_Funktion1\n'

                + 'ObjektZeigerFeld[' + ObjektNummer.toString()
                + '].prototype.Wert1= 0;    // Initialisierung\n'
                + '</SCRIPT>'
            );
}
```

Beim Prototyping von Funktion1 nicht () kodieren, da ja ein Zeiger übergeben werden soll !

Der Zugriff auf die Daten und Methoden des Objektes erfolgt über ObjektZeigerFeld[ObjektNummer].

z.B. ObjektZeigerFeld[ObjektNummer].Funktion1();
ObjektZeigerFeld[ObjektNummer].Wert1=33;

Die objekteigene Funktion1 kann natürlich auch Parameter haben.

Die Doppelbezeichnung ein und desselben Funktionscodes mit

"Funktion1"
und "OBJECT_IDx_Funktion1" x ist die jeweilige Objektnummer

macht Sinn, sobald mehrere Instanzen des externen Objektes gebildet werden sollen, da somit alle Objekte den selben Funktionsbezeichner haben, aber intern namentlich verschiedene. **Und**: Die Objekte haben je ihre **eigene** Funktion, die von keinem anderen Objekt benutzt werden kann. Das Prinzip der Kapselung wurde somit erfüllt. Der Funktionsname ist für den Programmierer einheitlich. Durch den Bezug per

ObjektZeigerFeld[ObjektNummer].Funktion1
ObjektZeigerFeld[ObjektNummer]Wert1

wird die Kapselung aufrechterhalten.

Diese Methode kostet Ressourcen und bläht die HTML-Datei auf ! Daher ist es natürlich auch möglich, nur 1 Methode zu deklarieren und dann den Objekten den identischen Zeiger zuzuweisen. Wichtig ist dabei, dass dann die Objekte **nicht parallel** auf diese Funktion zugreifen, da die Funktion in keinem Objekt gekapselt ist !!!

Durch die Auslagerung des Code der Funktion1 in die Stringvariable

RumpfCodeDerFunktion1

wird Übersicht erzeugt. Man beachte, dass
" und ' sich nicht inkorrekt mischen !,
die maximal mögliche Gesamtlänge des Strings beachtet werden muss, also eventuell mehrere Variablen zu kodieren sind.

Durch die Auslagerung der Variable

RumpfCode DerFunktion1

in eine eigene JS-Datei und deren Einbindung in das HTML-Dokument, kann der Aufbauder Funktion1, also des Objektes, von außen gesteuert werden, ohne das HTML-Dokument editieren zu müssen.

Empfehlenswert ist es, per \n am Zeilenende einen Zeilenumbruch zu erzeugen, damit die jeweilige erzeugte Zeile mit Ausführung von document.write() nicht unnötig zu lang wird, da Interpreter von Browsern nicht unbedingt überlange Zeilen ausführen können.

3.4.2.1. Objektvariable mit new deklarieren

Die Deklaration mit Anweisung new setzt ein nicht instanziertes Objekt voraus.

Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
Nur für Objekte, die aus vordefinierten Objekten abgeleitet werden, wird die Eigenschaft .prototype erzeugt.

Syntax:

var Zeiger = **new** konstruktor[(init_wert)]

konstruktor Bezeichner der Objektklasse

init_wert je nach Objektklasse

```
var Zeiger = new Function( [parameter_liste] funktions_rumpf)

        parameter_liste:    Kommatrennung

        funktions_rumpf:    Liste aus kommagetrennten Scriptanweisungen
                            jede Scriptanweisung in "  "  oder ‘  ‘ setzen
```

Beispiel 1:

```
var Zeiger1 = new Array("1", 2, 33+44);

var Zeiger2 = new Boolean();

var Zeiger3 = new window.document;      // Diese Anweisung bringt Fehler, da windows.document instanziert ist !!

var Zeiger4 = new Otto;                 // Diese Anweisung bringt Fehler, da Otto zuvor nicht definiert wurde !!

var PrivatesObjekt1 = new String('Private Objekt1');

var PrivatesObjekt2 = new PrivatesObjekt1; // Diese Anweisung bringt Fehler, da PrivatesObjekt1 instanziert ist !!

var PrivatesObjekt3 = new String('Private Objekt3');

var Zeiger = new Function("return init_wert;")
```

Beispiel 2:

```
function erzeuge_zwei_dim_feld(anzahl_spalten, anzahl_zeilen)
{
        // Spaltenfeld erzeugen
        this.spalten_feld= new Array(anzahl_spalten);

        // Zeilenfeld pro Spalte  erzeugen
        for (var spalte=0; spalte < anzahl_spalten; spalte++)
        { this[spalte] = new Array(anzahl_zeilen);}  //Zeilen-Felder
}
var zwei_dim_tabelle= new erzeuge_zwei_dim_feld(10,7); // 10 Spalten zu je 7 Zeilen
....
zwei_dim_tabelle[10,7]=22;   // Spalte 10: 7Zeile: mit  22 belegen
```

Beispiel 3 für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
<HTML>
<SCRIPT  LANGUAGE="JavaScript">
<!--
        function datenstruktur_ausgeben()
        {
                with (document)
                {       write(person.vorname + " " + person.nachname + "<BR>");
                        write(person.strasse   + " " + person.nummer   + "<BR>");
                        write(person.plz       + " " + person.ort      + "<BR>");
                        //      Erika Mustermann
                        //      Musterstrasse 1
                        //      ......
                }

                for (i in person)
                {       document.write(i + ": " + person[i] + "<BR>");
                        //      vorname: Erika
                        //      nachname: Mustermann
                        //      ...........
                }
        }

        function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
        {
                this.vorname = vorname;
                this.nachname = nachname;
                this.strasse = strasse;
                this.nummer = nummer;
                this.plz = plz;
                this.ort = ort;
```


```
                this.methode = methode;
        }

        person = new datenstruktur_erzeugen
                        (       "Erika",                        // Vorname
                                "Mustermann",                   // Nachnahme
                                "Musterstrasse",                // Strasse
                                "1",                            // Nummer
                                "10000",                        // PLZ
                                "Musterstadt",                  // Ort
                                datenstruktur_ausgeben          // ohne () kodieren
                        );

        // alternativ auch kodierbar:
        // var person = {       vorname:"Erika",                // Vorname
        //                      nachname:"Mustermann",          // Nachnahme
        //                      strasse:"Musterstrasse",        // Strasse
        //                      nummer:"1",                     // Nummer
        //                      plz:"10000",                    // PLZ
        //                      ort:"Musterstadt"               // Ort
        //                      methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
        //              };

        // Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !

        alert(person.vorname + "\n" + person.methode);          // person.methode ohne () kodieren !
// -->
</SCRIPT>
</HTML>
```

3.4.2.2. Objektvariable als Array Objekt aus Literalen deklarieren

Syntax:

```
var Zeiger = {literal_liste}
```

literalliste: kommagetrennte Elemente

Element

mit Aufbau "kette1"**:**"kette2"

Doppelpunkt muss kodiert werden

kette 1 Eigenschaft des Literal-Objektes
nur Plain-Text

kette2 Wert der Eigenschaft des Literal-Objektes
nur Plain-Text

Beispiel:

```
var Index = "";
var Menge = {"a" : "Athen" , "b" : "Berlin", "c" : "Paris", "d" : "Kairo"};
var Kette = "";

SchleifenAnweisung :                    // ein Label (Marke)
{
        for (Index in Menge)
        {
                Kette = "Hauptstadt  von ";

                switch (Menge[Index])
                {
                        case "Berlin":          {
                                                        Kette += "Deutschland: " + Menge[Index];
                                                        // nicht weiter auf Athen und Kairo prüfen
                                                        break;
                                                }

                        case "Athen":           {
                                                        Kette += "Griechenland: " + Menge[Index];
                                                        // kein break, also noch auf Kairo prüfen
                                                }

                        case "Kairo":           {       Kette += "Ägypten: " + Menge[Index]; }
```


```
                                        default:        {
                                                                // im Falle von Paris:
                                                                //      Kairo wird nie angezeigt, da im Feld
                                                                //      hinter Paris

                                                                break SchleifenAnweisung;
                                                        }
                                }

                                {alert(Kette);}         // nicht bei Paris abgearbeitet
                        }
                }
```

3.5. Operatoren

Prioritäten:

niedrigste	Komma
	Zuweisungen aller Art
	? :
	||
	&&
	|
	^
	&
	== !=
	< <= > >=
	<< >> >>>
	+ -
	* / %
	! Tilde ++ etc. typeof
höchste	() [] Punkt

Prioritäten durch Klammerung aufhebbar

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser leider **nicht als fehlerhaft** erkannt werden:

Bsp.:

```
var Kette =    'test';  + '.jpg';      // kein Syntaxerror wegen dem Semikolon vor dem Zeichen +
alert(Kette);                          // zeigt nur   test    an  und nicht   test.jpg
```

```
Bsp:    for (var i = 0; i <3; i++);    // kein Syntaxerror wegen dem Semikolon vor dem Zeichen {
        {alert(i);}                    // genau 1 Anzeige, die 3 anzeigt
```

3.5.1. Operatoren logische

logische Operationen liefern immer true oder false

||	Oder	--> true wenn mindestens 1 Operand true
&&	Und	--> true wenn alle Operanden true
!	not	

3.5.2. Operatoren arithmetische

+ - / *	+ für Addition **UND** Zeichenverkettung / ist NICHT die Ganzzahldivision
%	Modulo Bsp: 13%5 ergibt **3**, denn 13/5 ganz ist 2 mit Rest **3** Modulo liefert den Rest immer als Gleitkomma !
Ganzzahldivision	existiert nicht --> selbst programmieren

Variante 1:

```
ergebnis =     (zahl_1-(zahl_1 % zahl_2))
                / zahl_2;
```

Variante 2:

```
ergebnis= zahl_1 % zahl_2;              // Gleitkommrest ermitteln
ergebnis= zahl_1 - ergebnis;            // und von zahl_1 abziehen
ergebnis = ergebnis / zahl_2;
```

```
Variante 3: ergebnis = Math.floor(zahl1 / zahl2);       // liefert nächst kleinere ganze Zahl vom
                                                        // Gleitkomma-Ergebnis
            ergebnis = Math.ceil(zahl1 / zahl2);        // liefert ganze Zahl vom Gleitkommaergebnis,
                                                        // die größer oder gleich dem Gleitkommaergebnis
                                                        //      ist
```

Variablenoperatoren:

Variante 1:

```
bezeichner1+= bezeichner2;      entspricht   bezeichner1 = bezeichner 1 + bezeichner 2;
```


erst operieren dann zuweisen
Hinweis: auch für Strings möglich (Verkettung)
Wenn anstelle bezeichner2 mehrere Kettenvariablen kodiert
werden, die mit dem Operator '+' verkettet sind, so diese Verkettungsfolge als Ganzes innerhalb eines runden Klammernpaares kodieren (also mit höherer Priorität als +=) , damit vor Zuweisung der Verkettung auf bezeichner1 das Ergebnis der Verkettungsfolge als interner Zeiger vorliegt. Es könnte ansonsten dem bezeichner1 ein falscher Wert zugewiesen werden.

analog dazu für -= *= /= %=

Variante 2:

bezeichner++; erst bezeichner verwenden, dann um 1 erhöhen
++bezeichner; erst bezeichner um 1 erhöhen, dann verwenden

analog dazu für --

Hinweis: Math.ceil() dient zur Berechnung mit weiteren arithmetischen Operationsarten (siehe Math Objekt).
Dabein sind innerhalb () auch die Operatoren + - / * und % kodierbar.

3.5.3. Operatoren bitweise (Bitoperatoren)

bitweise Operationen (pro Bitposition)
liefern immer numerischen Wert
immer auf 32-Bit-Basis: für die Bitoperation werden Operanden auf 32-Bit-Breite umgewandelt

& neues Bit ist 1 wenn beide Operandenbits 1
| neues Bit ist 1 wenn mindestens ein Operandenbit 1 ist
^ neues Bit ist 1 wenn beide Operandenbits verschieden sind
~ Tilde (auf Taste mit + und *) Negation

Bitverschiebungen: sind abhängig von der Scriptmaschine
nachfolgende Darstellung der Bitoperationen muss nicht zutreffen (ausprobieren)

bezeichner >> bit_anzahl Bitverschiebung nach rechts:
reinkommende Bits sind 0
haben Wert des linksten Bits
(an höchster Bitposition) VOR der Verschiebung
rausfallende Bits gehen verloren

bezeichner >>> bit_anzahl Bitverschiebung nach rechts:
reinkommende Bits=0
rausfallende Bits gehen verloren

bezeichner << bit_anzahl Bitverschiebung nach links:
reinkommende Bits=0
rausfallende Bits gehen verloren

bezeichner <<< bit_anzahl gibt es leider **nicht**

3.5.4. Operatoren für Vergleich (Vergleichoperatoren)

Vergleichoperationen liefern immer true oder false

== gleich
!= ungleich
< > >= <=
in Teilmengenprüfung
Bsp. bezeichner1 in bezeichenr2 --> true, wenn bezeichner1 Teilmenge von bezeichner2

Achtung: Werte können überhaupt nur gleich sein, wenn sie gemeinsamen Typs sind --> eventuell vorher konvertieren !

3.5.5. Operatoren für Verkettung von Zeichenketten

Bsp: 'Test' + '1' ergibt 'Test1'

3.5.6. Operator für Zuweisung

einfachste Form: variable_1 = wert
spezielle Form: Für variable_1 = variable_1 + variable_2; kann auch kodiert werden variable_1 += variable_2;

Für variable_1 = variable_1 + wert_1; kann auch kodiert werden variable_1 += wert_1;

analog für - * / % << >> >>> & ^ |

3.5.7. Operator für Ermittlung des Datentyps (Operator typeof)

typeof operand;

Typen: "undefined"
"function"
"object"
"number"
"boolean"
"string"

Bsp: typeof 42 ergibt "number"

3.5.8. Ausdruck berechnen, aber den Wert nicht liefern (Operator void)

void(ausdruck) wertet den Ausdruck aus, aber liefert den Wert nicht

Bsp.: <A HREF="javascript:void(0);"> ...</A> erzeugt Link ohne Verweiswert

3.5.9. this (Zeiger auf aktuelle Objekt-Instanz)

(Anwendung auf Einzelanweisung zur Objektinstanz)

this.objekt

Beispiel 1 für Erzeugung eines 2-dimensionalen Feldes:

```
function erzeuge_zwei_dim_feld(anzahl_spalten, anzahl_zeilen)
{
        // Spaltenfeld erzeugen
        this. spalten_feld= new Array(anzahl_spalten);

        // Zeilenfeld pro Spalte  erzeugen
        for ( var spalte=0; spalte < anzahl_spalten; spalte++)
        { this[spalte] = new Array(anzahl_zeilen);}
}

var zwei_dim_tabelle= new erzeuge_zwei_dim_feld(10,7); // 10 Spalten zu je 7 Zeilen
    ....
zwei_dim_tabelle[10,7]=22;   // in Spalte 10 die Zeile7 mit  22 belegen
```

Beispiel 2 für HTML-Kodierung:

```
<INPUT
        TYPE=button
        NAME="logischer_button_name"
        onclick="alert(this.name)"
>
```

3.5.10. with (Zeiger auf aktuelle Objekt-Instanz)

(Anwendung auf Block von Anweisung zur Objektinstanz)

with (objekt_instanz) { .... }

Alle Anweisungen innerhalb { ....} beziehen sich NUR auf die Instanz laut objekt_instanz.
Die Kodierung der Instanz per Punktnotation entfällt (vereinfachte Kodierung per with).

3.5.11. Operatoren in Microsoft JScript

Operatoren sind

+	Addition
-	Subtraktion
*	Multiplikation
/	Division (Gleitkomma)
%	Modulo
--	dekrementieren um 1
++	inkrementieren um 1
=	Zuweisung
&	bitweises AND
~	bitweises NOT
|	bitweises OR

^	bitweises XOR
>>	bitweises Rechtsverschieben
<<	bitweise Linksverschieben
>>>	bitweises unsigned Rechtsverschieben
,	Trenner bei Aufzählung z.B. für Feld oder Funktions-Parameter-Liste
&&	logisches AND
!	logisches NOT
||	logisches OR (zwei senkrechte Striche)
<	Wert-Vergleich auf kleiner
>	Wert-Verlgeich auf größer
<=	Wert-Vergleich auf kleinergleich
>=	Wert-Vergleich gößergleich
==	Wert-Vergleich auf Identität (zwei Gleichheitszeichen)
!=	Wert-Vergleich auf Ungleichheit (bei String reicht die erst gefundene Ungleichheit)
===	Wert- **und** Datentyp-Vergleich auf Identität (drei Gleichheitszeichen)
!===	Wert- **und** Datentyp-Vergleich auf Ungleichheit (drei Gleichheitszeichen)
?:	if-Anweisung in anderer Form
-	Vorzeichen Minus
delete	Eigenschaft oder Methode von einem Objekt löschen (Achtung: nicht bei jedem Objekt zulässig) Element eines Feldes löschen (Achtung: nicht bei jedem Feld zulässig)
instanceof	prüfen ob eine Instanz einer Objektklasse angehört
new	Erzeugung eines Objektes (Achtung: nicht bei jedem Objekt möglich)
typeof	Datentyp eines Ausdruckes
void	Ausdruck liefert immer undefined bzw. null

Achtung: <<< Operator existiert leider **nicht** !

Hinweise zu den Vergleichsoperatoren

<	Vergleich auf kleiner
>	Vergleich auf größer
<=	Vergleich auf kleinergleich
>=	Vergleich gößergleich
==	Vergleich auf Identität (zwei Gleichheitszeichen)
!=	Vergleich auf Ungleichheit (erst reicht die erst gefundene Ungleichheit)
===	Vergleich auf Identität in Wert und Datentyp (drei Gleichheitszeichen)
!===	Vergleich auf Ungleichheit in Wert und/oder Datentyp (drei Gleichheitszeichen)

wenn ein Vergleichoperand NaN, dann wird immer false geliefert
Stringvergleich in lexigraphischer Folge
-0 identisch mit +0

-Infinity	ist immer kleiner als alles andere numerische
Infinity	ist immer größer als alles andere numerische
NaN	ist immer ungleich zu allem anderen
null	bei Zeiger auf eine Instanz (Adresse der Instanz)
undefined	bei Werten (Inhalt einer Instanz)

Hinweise zum Operator %

Ganzzahl-Division und den Rest davon liefern

Syntax:

[var Wert =] number1 **%** number2;

number1 Integer oder Floating point
wenn Floating point, so automatisch auf Integer gerundet

number2 Integer oder Floating point
wenn Floating point, so automatisch auf Integer gerundet

Wert Integer

Ablauf: 1. Schritt interne_variable1 = number1 / number 2;

interne_variable1 automatisch erzeugt
wird auf nächsten kleineren Integerwert gesetzt

2.Schritt interne_variable2 = interne_variable 1 * number 2

interne_variable2 automatisch erzeugt

3.Schritt Wert = number1 - interne_variable2

Beispiel: 33 % 10 33 / 10 = 3,3 in Periode, also 3

3 * 10 = 30

33 - 30 = **3**

19 % 6.7 19 / 7 = 2,7... also 2

2 * 7 = 14

19 - 14 = **5**

Hinweise zu den Operatoren

+	Addition bzw. Verkettung
-	Subtraktion
/	Division
*	Multiplikation
%	
<<	
>>	
>>>	
^	
|	
=	Zuweisung

Es ist auch kodierbar

+=
-=
/=
*=
%=
<<=
>>=
>>>=
^=
|=

Beispiel:
```
var Wert = 10;
Wert +=23;
alert(Wert);          // zeigt 30 an

Wert %=10;
alert(Wert);          // zeigt 3 an
```

Hinweise zum Operator <<

bitweise Linksverschiebung
von rechts reinkommende Bits sind die Bits, die links rausfallen
nach links rausfallende Bits kommen rechts wieder rein

Syntax: [Wert =] variable1 << variable2;

variable1 mit Wert der bitweise nach links zu verschieben ist

variable2 numerisch
Anzahl der Bits, um die verschoben wird

Wert mit Typ laut variable1

Beispiel:

var Wert = -14 << 2

mit -14 als Bitfolge 11***111111 11111111 11111111 11110010***

Wert nach Verschiebung in Bitfolge ***111111 11111111 11111111 11110010***11

Hinweise zum Operator >>

bitweise Rechtsverschiebung
von links reinkommende Bits sind die Bits, die rechts rausfallen
nach rechts rausfallende Bits kommen links wieder rein

Syntax: [Wert =] variable1 >> variable2;

variable1 mit Wert der bitweise nach rechts zu verschieben ist

variable2 numerisch
Anzahl der Bits, um die verschoben wird

Wert mit Typ laut variable1

Beispiel:

var Wert = -14 >> 2

mit -14 als Bitfolge ***11111111 11111111 11111111 111100***10

Wert nach Verschiebung in Bitfolge 10***11111111 11111111 11111111 111100***

Hinweise zum Operator >>>

bitweise Rechtsverschiebung
von links reinkommende Bits sind immer 0
nach rechts rausfallende Bits bleiben weg

Syntax: [Wert =] variable1 >>> variable2;

variable1 mit Wert der bitweise nach rechts zu verschieben ist

variable2 numerisch
Anzahl der Bits, um die verschoben wird

Wert mit Typ laut variable1

Beispiel:

var Wert = -14 >>> 2

mit -14 als Bitfolge ***11111111 11111111 11111111 111100***10

Wert nach Verschiebung in Bitfolge 00***111111 11111111 11111111 11111100***

Hinweis zum Operator ?:

siehe auch if-Anweisung

Syntax: ausdruck **?** statement1 **:** statement2;

ausdruck: **muss** boolean liefern
wenn true, so statement1 aktiviert
wenn false, so statement2 aktiviert

statement Scriptanweisungen in " " kodieren

Hinweise zum Operator delete

Löschen einer Objekt-Eigenschaft oder Objekt-Methode oder des Objektes selbst
bei Feld und Collection: Elemente **nur** durch objekteigene Methoden löschen
JScript-Objekte sind nicht löschbar:
Nur per **Prototyping** zum JScript-Objekt hinzugefügte Eigenschaften und Methoden sind löschbar

Syntax: [var Wert =] **delete** ausdruck;

ausdruck muss Zeiger liefern per
Name einer Eigenschaft
Referenz auf Feldelement

Wert true, so Löschung erfolgreich
false, so Löschung nicht erfolgreich

Hinweise zum Operator instanceof

Syntax: [var Wert =] zeiger_auf_objekt **instanceof** objekt_klasse;

objekt_klasse ist ein Objekt, das im Browser automatisch implementiert wurde, aber nicht Teil des DOM des HTML-Dokumentes ist, also Objekte, die per new erzeugbar sind wie z.B. Date, Array, Object

Wert true, so Instanz existiert **und** Instanz wurde per Objektklasse instanziert (z.B. per new)
false, so Instanz wurde nicht mit dieser Objektklasse instanziert
oder Instanz existiert erst gar nicht

Beispiel:

```
function Testen(ZeigerAufObjekt))
{
	var IndexAlsString = "";
	var Kette = "";

	FeldDerObjektKlassen = new Array();

	// Feldelement mit Index "Date" füllen mit Instanz der Objektklasse Date
	FeldDerObjektKlassen["Date"] = Date;

	// Feldelement mit Index "Object" füllen mit Instanz der Objektklasse Object
	FeldDerObjektKlassen["Object"] = Object;

	// Feldelement mit Index "Array" füllen mit Instanz der Objektklasse Array
	FeldDerObjektKlassen["Array"] = Array;

	for (IndexAlsString in FeldDerObjektKlassen)
	{
		// Vergleich auf Klasse
		if (ZeigerAufObjekt instanceof FeldDerObjektKlassen[IndexAlsString])
		{
			Kette += "Objekt ist eine Instanz von " + IndexAlsString + "\n";
		}
		else
		{
			Kette += "Objekt ist keine Instanz von " + IndexAlsString + "\n";
		}
	}

	return(Kette);
}

var ZeigerAufDateObjekt = new Date();
alert (Testen(ZeigerAufDateObjekt));
```

Hinweise zum Operator typeof

Typ des Ergebnisses eines Audruckes liefern

Syntax:

[var Kette =] **typeof** audruck;

[var Kette =] **typeof**(audruck); // als Funktion

Kette String
"number"
"string"
"boolean"
"object"
"function"
"undefined"

leider **nicht** "array" etc.

Hinweise zum Operator void

Ausdrucksergebnis immer auf **undefined** bzw. **null** setzen

Syntax:

void ausdruck

Beispiel: if (void (10 + 30) == null) // Klammern um 10 + 30, sonst gilt void 10, da Addition geringere
// Priorität als void hat
{ ..... }

Übersicht zu den Standard-Prioritäten der Operatoren für die Abarbeitung von Operationen:

höhere Priorität: der Operator wird zuerst verwendet
von oben (höchster) nach unten (niedrigster)

Operator	Funktion des Operators
.	Punktnotation
[]	Feldnotation
()	Klammerung bzw. Funktionsparameterliste
++	Inkrementierung um 1
--	Dekrementierung um 1
-	Vorzeichen
~	bitweises NOT
!	logisches NOT
delete	
new	
typeof	
void	
*	Multiplikation
/	Division (Gleitkomma)
%	Modulo
+	Addition oder Verkettung
-	Subtraktion
<<	
>>	
>>>	
<	
<=	
>	
>=	
instanceof	
==	Vergleich auf Gleichheit
!=	Vergleich auf Ungleichheit
===	Vergleich auf Gleichheit (Wert und Datentyp)
!==	Vergleich auf Ungleichheit (Wert und Datentyp)
&	bitweises AND
^	bitweises XOR
|	bitweise OR
&&	logisches AND
||	logisches OR
?:	if-Abfrage
=	Zuweisung eines Wertes etc auf eine Instanz auch Kombination z.B. +=
,	Trenner in Aufzählung oder Liste

Aber: Innerhalb eines Audruckes gilt folgende Priorität von oben nach unten:

()	Klammerung
-	**Subtraktion**
+	**Addition oder Verkettung**
*	Multiplikation
/	Division
%	Modulo
=	Zuweisung

Fettgedrucktes ist die Abweichung von obiger Liste der Prioritäten aller Operatoren:
Die Priorität ist **vertauscht** . Diese Regelung in JScript klingt unsinnig, aber kommt dann zum Zuge, wenn Terme addiert **und** subtrahiert werden sollen:
Es wird **zuerst** der Term für die Subtraktion berechnet, um **dann** das Ergebnis für die Addition mit einem anderen Term zu verwenden. Das interne Zwischenergebnis entsteht also nur in dieser Reihenfolge.
Sollte also erst + und dann - kodiert sein, so bewirkt das Minus eine Klammerung.
Empfehlung: Entweder alles klammern oder erst Minus und dann Plus kodieren.

Beispiel: var **Wert** = (10 * 20) + (100 - 10) + 30 - 20;

Schritt 1: 10 * 20 200 intern merken

Schritt 2:	100 - 10	90	intern merken
Schritt 3:	30 - 20	10	intern merken, wie Klammerung
Schritt 4:	200 + 90	290	
Schritt 5:	290 + 10	**300**	

nur im Ergebnis dasselbe wie

Schritt 1:	10 * 20	200	intern merken
Schritt 2:	100 - 10	90	intern merken
Schritt 3:	200 + 90	290	
Schritt 4:	290 + 30	320	
Schritt 5:	320 - 20	**300**	

Beispiel: var **Wert** = (10 * 20) + (100 - 10) - 20 + 30 ;

Schritt 1:	10 * 20	200	intern merken
Schritt 2:	100 - 10	90	intern merken
Schritt 3:	90 - 20	70	wie Klammerung
Schritt 4:	200 + 70	270	
Schritt 5:	270 + 30	**300**	

nur im Ergebnis dasselbe wie

Schritt 1:	10 * 20	200	intern merken
Schritt 2:	100 - 10	90	intern merken
Schritt 3:	200 + 90	290	
Schritt 4:	290 - 30	270	
Schritt 5:	270 - 30	**300**	

Beispiel: var **Wert** = (10 * 20) + 100 - 10 - 20 + 30 ;

Schritt 1:	10 * 20	200	intern merken
Schritt 2:	100 - 10	90	intern merken
Schritt 3:	90 - 20	70	intern merken
Schritt 4:	200 + 70	270	
Schritt 5:	270 + 30	**300**	

Beispiel: var **Wert** = (10 * 20) + 100 - 10 + 30 - 20;

Schritt 1:	10 * 20	200	intern merken
Schritt 2:	100 - 10	90	intern merken
Schritt 3:	30 - 20	10	intern merken
Schritt 4:	200 + 90	270	
Schritt 5:	290 + 10	**300**	

Beispiel: var **Wert** = -20 + (10 * 20) + (100 - 10) + 30 ;

Schritt 1:	10 * 20	200	intern merken
Schritt 2:	100 - 10	90	intern merken
Schritt 3:	-20 + 200	180	
Schritt 4:	180 + 90	270	
Schritt 5:	270 + 30	**300**	

Empfehlung: Um Prioritätenfolgen zu vermeiden oder zu ändern (oder falls man sich diese nicht merken kann, da andere Programmiersprachen wieder andere Regeln haben), sind Klammerungen sinnvoll.

Achtung: Jede Klammerung erzeugt eine interne Stack-Speicherung des Ergebnisses aus dem Ausdruck in der Klammer. Der Stack ist **nur endlich groß**.

Beispiel: var **Wert** = (10 * 20 + 100) - 10;

Schritt 1:	10 * 20	200
Schritt 2:	200 + 100	300
Schritt 3:	300 - 10	**290**

Beispiel: var **Wert** = (10 * 20) + (100 - 10); // identisch mit (10 * 20) + 100 - 10;

Schritt 1:	10 * 20	200
Schritt 2:	100 - 10	90
Schritt 3:	200 + 90	**290**

3.5.12. Operatoren in Javascript 1.5 im Netscape 6.x

arithmetische Operatoren:

+
++
-
--
*
/
%

String-Operatoren:

+
+=

Logische Operatoren:

&&
||
!

Bitweise Operatoren:

&
^
|
~
<<
>>
>>>

Zuweisungsoperatoren:

=
+=
-=
*=
/=
%=
&=
^=
|=
<<=
>>=
>>>=

Vergleichsoperatoren:

==
!=
===
!==
>
>=
<
<=

Sonstige Operatoren:

?:
, (Aufzählung)
delete
function
in
instanceof
new
this
typeof
void

3.6. Anweisungen

mit Semikolon abschließen (außer Blockanweisung und Kommentar). Der Parser ist beim Weglassen des Semikolons nur z.T. fehlerneutral.
Hinweis: Bei Syntaxbeschreibungen bedeutet [ ] eine Optionalkodierung.

Kommentar
wird nicht geparst

```
// das ist ein einzeiliger Kommentar

/*
        das ist ein -
        oder mehrzeiliger
        Kommentar
*/
```

für JScript: siehe auch bedingtes Parsen durch @ Statements wie @if etc.

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser leider **nicht als fehlerhaft** erkannt werden:

```
Bsp.:   var Kette =   'test';  + '.jpg';      // kein Syntaxerror wegen dem Semikolon vor dem Zeichen +
        alert(Kette);                         // zeigt nur   test    an  und nicht   test.jpg

Bsp:    for (var i = 0; i <3; i++);           // kein Syntaxerror wegen dem Semikolon vor dem Zeichen {
        {alert(i);}                           // genau eine Anzeige, die 3 anzeigt
```

3.6.1. Anweisungen in Javascript und JScript

;
Leeranweisung, die nichts bewirkt und nur als Platzhalter dient
anwenden um zu verhindern, dass der Parser einen Teil des Codes nicht berücksichtigt, wenn letzterer ansonsten leer ist
Bsp:

```
if (10 > 9)
{ ;}
else
{alert("kleiner");}
```

{ }
Blockanweisung (.z.Z. in der Syntaxvorschrift enthalten)
endet ohne Semikolon
enthält mindestens 1 Anweisung (auch Leeranweisung)
Anweisung(en) mit Semikolon abschließen

break
beendet Schleifen-Anweisung bzw. per Label markierte Anweisung
Achtung: Die Verwendung von break in Schleifen
zeugt von der Faulheit des Programmierers, denn jede Schleife kann ohne break-Anweisung programmiert werden
bedarf einer intern-erweiterten Zeigerverwaltung durch den Browser

siehe label Anweisung und Schleifen und switch Anweisung

Syntax:

```
break [label];
```

label Label (Sprungmarke) der abzubrechenden Anweisung

Beispiel 1:

```
var i = 0;
while (i < 100)
{
        if (i == 55)
        break;      // mit alert() weitermachen
        i++;
}
alert("Ende der Schleife");
```

Beispiel 2:

```
var Index = "";
var Menge = {"a" : "Athen" , "b" : "Berlin", "c" : "Paris", "d" : "Kairo"};
var Kette = "";

SchleifenAnweisung :                    // ein Label (Marke)
{
        for (Index in Menge)
        {
                Kette = "Hauptstadt  von ";

                switch (Menge[Index])
                {
```


```
                                        case "Berlin":      {
                                                                Kette += "Deutschland: " + Menge[Index];
                                                                // nicht weiter auf Athen und Kairo prüfen
                                                                break;
                                                            }

                                        case "Athen":       {
                                                                Kette += "Griechenland: " + Menge[Index];
                                                                // kein break, also noch auf Kairo prüfen
                                                            }

                                        case "Kairo":       {   Kette += "Ägypten: " + Menge[Index]; }

                                        default:            {
                                                                // im Falle von Paris:
                                                                //        Kairo wird nie angezeigt, da im Feld
                                                                //        hinter Paris

                                                                break SchleifenAnweisung;
                                                            }
                                    }

                                    {alert(Kette);}         // nicht bei Paris abgearbeitet
                                }
                            }
```

continue

Start des nächsten Schleifendurchlaufes
alle Anweisungen hinter continue werden ignoriert, wenn sie in der Schleife liegen
siehe label Anweisung

Syntax:

```
continue [label];
```

label Label (Sprungmarke) der Anweisung z.B. Marke in der Switch-Anweisung, aber der fortgesetzt werden soll
siehe Anweisung label

Beispiel 1:

```
var  Kette= "", i=0;

while (i < 10)
{
    i++;

    // wenn 5 dann Kette nicht erweitern um i
    if (i==5)
    {continue; }

    Kette += i;// wird nicht abgearbeitet, wenn i == 5
}

alert (Kette);          // Ziffer 5 fehlt
```

Beispiel 2:

```
var Feld = new Array();

Outer:  for (var i = 0; i < 5; i++)
        {
            Inner:  for (var j = 0; j < 5; j++)
                    {
                        if (j == 2)
                        {continue Inner;} // Sprung zur Marke Inner
                        else
                            { Feld[i,j] = j + 1;}
                    }
        }
```

do...while

tue etwas (do), solange (while) es erlaubt ist (while liefert true)
mindestens 1 Durchlauf
siehe auch while Anweisung, wenn der Mindestdurchlauf nicht erwünscht ist

Syntax:

```
do
{statements}
while (ausdruck) ;
```

statements — wenn nur 1 Statement so kann Blockanweisung { } entfallen

ausdruck — muss true oder false liefern
wenn true, so — nächster Schleifendurchlauf
wenn false, so — Ende der Schleife

for

tue etwas in genau definierter Anzahl

Syntax:

```
for ([initialization]; test; increment_oder_decrement)
{statements}
```

Semikolons sind Pflichtkodierung

initialization — optional
kann entfallen, wenn z.B. kein Zähler benutzt wird, der in test und increment_oder_decrement und in statements benutzt wird
Achtung: Wird ein Zähler benutzt, dann ist dieser immer lokal zur Schleife, auch wenn der Zähler außerhalb der Schleife deklariert wurde. Der Zähler ist also nur innerhalb des Schleifenkörpers (statements) verwendbar.
Während der Abarbeitung der Schleife findet unter Windows 9x, das kein echtes Multitasking kann, das Anhalten von parallelen Prozessen statt. Sollte die Schleife nie enden, wird der Browser eine Meldung zur Zeitüberschreitung liefern oder das Betriebssystem eventuell abstürzen. Die Schleife ist also im Ablauf zeitlich so kurz wie möglich zu halten, damit z.B. die PC-Uhr nicht falsch geht. **Bei Verwendung vieler Schleifen ist mit dem Nachgehen der PC-Uhr zu rechnen.**

statements — wenn nur 1 Statement so kann Blockanweisung { } entfallen

test — Abbruchbedingung
muss true oder false liefern
Prüfung kann durch Vergleich der in initialization mit numerischen Wert initialisierten Variablen auf einen Maximal bzw. Minimalwert erfolgen
wenn true, so { ... } erneut abarbeiten
wenn false, so Schleife beenden

increment_oder_decrement — der zu prüfenden Variablen
also ++ oder --
(Kodierungsformen von ++ bzw. -- beachten !)

Beispiel 1:

```
for (i = 0; i < 10; i++)
{alert(i);}
```

Beispiel 2:

```
for ( ; ! DateiSystem_Laufwerke.atEnd();DateiSystem_Laufwerke.moveNext())
{
    // Laufwerk ermitteln
    Laufwerk = DateiSystem_Laufwerke.item();

    // Laufwerksbuchstabe ermitteln
    LaufwerkBuchstabe = Laufwerk.DriveLetter;
    Kette = Kette + LaufwerkBuchstabe + " - ";

    // Art des Laufwerkes ermitteln

    if (Laufwerk.DriveType == 3)
    {
        // ist Netzlaufwerk

        // öffentlicher Netz-Name des Laufwerkes
```


```
                    LaufwerkName = Laufwerk.ShareName;
                }
                else
                {
                    // nicht Netzwerk-Laufwerk

                    // prüfen ob Laufwerk bereit ist
                    if (Laufwerk.IsReady)
                    {
                        // ist bereit, dann Volume-Bezeichner ermittelbar
                        LaufwerkName = Laufwerk.VolumeName;
                    }
                    else
                    { LaufwerkName = "[Drive not ready]";}
                }

                Kette += LaufwerkName + "\n";
            }
            alert (Kette);
```

for...in

tue etwas, wenn es erlaubt ist

Syntax:

```
for (variable in [zeiger])
{statements}
```

variable	mit Datentyp laut object bzw. array Inhalt	wenn in der Menge der Werten laut object bzw. array enthalten, so statements abarbeiten Menge der Werte: Anzahl der Werte in der Menge ist die maximale Anzahl der Schleifenabarbeitung
zeiger	auf Object oder Array	
statements	wenn nur 1 Statement so kann Blockanweisung { } entfallen	

Beispiel 1:

```
Index, WerteKette = "";
var Menge = {"a" : "Athen" , "b" : "Belgrad", "c" : "Kairo"};

for (Index in Menge)
{
    WerteKette += Menge[Index];
}
```

Beispiel 2:

```
function ErzeugeObjekt(Wert1, Wert2)
{
    this.ObjektWert1=Wert1;
    this ObjektWert2=Wert2;
}

var MeinObjekt = new ErzeugeObjekt(1,2);
for (i in MeinObjekt)
{alert('Element ' + i + ' ist ' + MeinObjekt[i] + '\n');}
```

function

Deklaration einer frei-programmierten (privaten) Funktion in Script
Verschachtelung möglich
siehe auch Script-Objekt Function

Syntax:

```
function freier_funktions_bezeichner ([ argumenten_liste ])      // Kopf der Funktion
{statements}                                                     // Rumpf der Funktion
```

freier_funktions_bezeichner	darf kein Schlüsselwort von Script sein
statements	Rumpf der Funktion immer Blockanweisung kodieren muss nicht return-Anweisung enthalten, kann aber

Hinweis: Eine PROCEDURE gibt es nicht in Script

argumenten_liste — siehe auch Script-Objekt Function

Folge von per Komma getrennten Elementen

Listenelement:
- immer Referenz
- Platzhalter für Parameter (Wert oder Referenz)
  - Hinweis: intern wird Wert-Parameter auch referenziert
  - Wert: z.B. numerisch, Boolean, String, Ausdruck
- wird in statements verarbeitet
- ist nur funktions-lokal gültig

```
var funktions_name = new function(["argumenten_liste"],"javascript_anweisungen");
```

javascript_anweisungen — sind der Funktionskopf
in " " bzw. ' ' zu setzen
mit ; trennen

Beispiel 1 für Anweisung function:

```
var GlobaleVariable1 = "Bitte laut";
var GlobaleVariable2 = "lein";
var GlobaleVariable3 = "";

function GlobaleFunktion_ZeichenLiefern(ZeichenArt)          // ZeichenArt ist funktionslokal
{
        var FunktionsLokaleVariable = ""                     // Initialisierung des String

        if (ZeichenArt = "Blank")
        { FunktionsLokaleVariable = " ";}

        if (ZeichenArt = "Doppelpunkt")
        { FunktionsLokaleVariable = ":";}

        return FunktionsLokaleVariable; // Funktion liefert Inhalt von FunktionsLokaleVariable
                                        //      und keinen Zeiger auf  FunktionsLokaleVariable,
                                        //      da ein Zeiger auf eine funktionslokale Variable
                                        //      niemals lieferbar ist: Zeiger existiert nur zur Laufzeit
                                        //      der Funktion !!
}

function Globale_BeispielFunktion(      FunktionsLokalesArgument1_Referenz,
                                        FunktionsLokalesArgument2_Wert_Numerisch
                                        FunktionsLokalesArgument2_Wert_String
                                  )
{
        function LokaleFunktion_LeerZeichenLiefern()
        { return (GlobaleFunktion_ZeichenLiefern("Blank")); }        // liefert " "

        function LokaleFunktion_DoppelpunktLiefern()
        { return GlobaleFunktion_ZeichenLiefern("Doppelpunkt ");}    // liefert ":"

        var FunktionsLokaleVariable1 = "kleine";

        var FunktionsLokaleVariable2 =

                // Referenz auf GlobaleVariable1
                  FunktionsLokalesArgument1_Referenz          // "Bitte laut"

                + LokaleFunktion_LeerZeichenLiefern()         // "Bitte laut "

                + "mitsingen"                                 // "Bitte laut mitsingen"

                + LokaleFunktion_LeerZeichenLiefern()         // "Bitte laut mitsingen "

                + LokaleFunktion_DoppelpunktLiefern()         // "Bitte laut mitsingen :"

                + LokaleFunktion_LeerZeichenLiefern()         // "Bitte laut mitsingen : "
```


```
                                        // numerischen Wert 3 konvertieren zu "3"
                                        + FunktionsLokalesArgument2_numerisch_Wert.toString()
                                                                                    // "Bitte laut mitsingen : 3"

                                        + LokaleFunktion_LeerZeichenLiefern()       // "Bitte laut mitsingen : 3 "

                                        // "kleine"
                                        + FunktionsLokaleVariable                   // "Bitte laut mitsingen : 3 kleine"

                                        + LokaleFunktion_LeerZeichenLiefern()       // "Bitte laut mitsingen : 3 kleine "

                                        // String-Wert  "Neger"
                                        + FunktionsLokalesArgument2_Wert_String;    // "Bitte laut mitsingen : 3 kleine Neger"

                                GlobaleVariable3 = FunktionsLokaleVariable2;        // Inhalt der lokalen Variablen nach
                                                                                    //          globale Variable kopieren
                                                                                    // man hätte auch return-Anweisung
                                                                                    //          nehmen können
                        }

                        // Aufruf mit korrekter Argumentenversorgung durch Parameterliste also
                        //                  GlobaleVariable1            "Bitte laut"
                        //                  Ausdruck 1 + 2              3
                        //                  "Neger"
                        //          Funktion belegt GlobaleVariable3 mit dem Wert "Bitte laut mitsingen : 3 kleine Neger"

                        Globale_BeispielFunktion(
                                                GlobaleVariable1,   // "Bitte laut"
                                                1 + 2,              // Ausdruck, der den Wert 3 liefert
                                                "Neger"
                                    );

                        // Anzeige per Alert-Box (Standard-Funktion) von  "Bitte laut mitsingen : 3 kleine Negerlein ..."
                        alert(
                                GlobaleVariable3                    // "Bitte laut mitsingen : 3 kleine Neger"

                                + GlobaleVariable2                  // "Bitte laut mitsingen : 3 kleine Negerlein"

                                + GlobaleFunktion_ZeichenLiefern("Blank")  // "Bitte laut mitsingen : 3 kleine Negerlein "
                                                                           // Achtung: Aufruf von LokaleFunktion_DoppelpunktLiefern()
                                                                           //          ist nicht zulässig

                                + "..."                             // "Bitte laut mitsingen : 3 kleine Negerlein ..."
                        );
```

Beispiel 2 für Ableitung vom Script-Objekt Function :

```
var Zeiger = new Function("return init_wert;")
```

Argumente einer Funktion:

```
Beispiel:   function test(x)
              {alert(x==null);}

              test();       // liefert true
              test(1);      // liefer false
```

Werden Argumente nicht mit Wert belegt, so werden die Variablen der Argumente nicht erzeugt, sind also null.

Argumente sind null-Zeiger, wenn kein Wert mit Funktionsaufruf übergeben wird.

Es müssen also bei korrekter Programmierung die Argumente auf geprüft werden
ERST auf != null
DANN auf Wertbereich.

Achtung: Der Datentyp des Argumentes wird mit dem Wert, dem das Argument erhält, festgelegt.

Das Argument ist ein Zeiger, der auf einen Speicherbereich mit einem Wert, der Daten nach JScript-zulässigen Typen hat.

:

Returnwert einer Funktion:

Returnwert: ist optionale Referenz auf Rückgabewert
Auch wenn Retutrnwert geliefert wird, so muss er nicht verarbeitet werden.

Beispiel 1:

```
function test()
{var X=3;}

alert(test()!=null);    // liefert false, keine Referenz
alert(test());          // liefert undefined da keine Referenz
```

Beispiel 2:

```
function test()
{var X=3;return X;}

alert(test()!=null);    // liefert true, Referenz vorhanden und ausgewertet
alert(test());          // liefert 3 da alert die Referenz auswertet
```

Beispiel 3:

```
function test()
{var X=3;return X;}

test();                 // kein Scriptfehler, aber Referenz nicht ausgewertet
```

if...else

Fallabfrage
siehe auch Operator ?:
Verschachtelung möglich

Syntax:

```
if (condition)
{statements1}
[else {statements2}]
```

condition	Ausdruck der true oder false liefert Referenz auf Boolean-Variable wenn true, so statements1 abgearbeitet wenn false, so statements2 abgearbeitet
statements1 bzw. 2	wenn nur 1 Statement **nur dann** darf Blockanweisung { } entfallen

Beispiel:

```
var x=5;
var y=8;
var z=0;

// aktuelles x prüfen
if (x == 5)
{
    // x ist 5
    // aktuelles y prüfen
    if (y == 6)
    {
        // y ist 6
        z = 17;
    }
    else
    {
        // y ist nicht 6
        z = 20;
        x= 6;
    }
}
else
{ x = 0;}   // Achtung: x ist inzwischen auf 6, wird aber für else nicht verwendet
            //          aber x wird auf 0 gesetzt !!
```

in

prüfen ob String in einem Objekt als Menge von Strings enthalten ist
in Operator arbeitet analog in der Anweisung for in
Syntax:

```
[ var Wert = ] ausdruck_oder_referenz in referenz_auf_objekt_mit string_elementen
```


ausdruck	muss String liefern
referenz	auf String-Variable
Wert	true, so enthalten false, so nicht enthalten

Beispiel für Array Objekt aus Literalen:

```
var Menge = {"a" : "Athen" , "b" : "Belgrad", "c" : "Kairo"};

var Index = "a";
 if( "a" in Menge )
{alert(Menge[Index];}
```

Label

Sprungmarke für die continue Anweisung innerhalb Schleife
Marke für die break Anweisung zum Abbrechen der mit Label markierten Anweisung
siehe auch Anweisungen break und continue
Syntax:

```
label : { statements }
```

label	anstelle von "label" ist ein freier Bezeichner zu kodieren, der kein Bezeichner eines bereits definierten Elementes und kein Schlüsselwort sein darf
statements	wenn nur 1 Statement so kann Blockanweisung { } entfallen

Beispiel 1:

```
var Feld = new Array();

Outer:  for (var i = 0; i < 5; i++)
        {
                Inner:  for (var j = 0; j < 5; j++)
                        {
                                if (j == 2)
                                {continue Inner;} // Sprung zur Marke Inner
                                else
                                        { Feld[i,j] = j + 1;}
                        }
        }
```

Beispiel 2:

```
var Index = "";
var Menge = {"a" : "Athen" , "b" : "Berlin", "c" : "Paris", "d" : "Kairo"};
var Kette = "";

SchleifenAnweisung :                    // ein Label (Marke)
{
        for (Index in Menge)
        {
                Kette = "Hauptstadt  von ";

                switch (Menge[Index])
                {
                        case "Berlin":          {
                                                Kette += "Deutschland: " + Menge[Index];
                                                // nicht weiter auf Athen und Kairo prüfen
                                                break;
                                                }

                        case "Athen":           {
                                                Kette += "Griechenland: " + Menge[Index];
                                                // kein break, also noch auf Kairo prüfen
                                                }

                        case "Kairo":           {       Kette += "Ägypten: " + Menge[Index]; }

                        default:                {
                                                // im Falle von Paris:
                                                //            Kairo wird nie angezeigt, da im Feld
                                                //            hinter Paris

                                                break SchleifenAnweisung;
                                                }
```


```
                                        }

                                        {alert(Kette);}          // nicht bei Paris abgearbeitet
                                }
                        }
```

new

Objekt (Objektinstanz) erzeugen
Zeiger auf Instanz erzeugen und Speicher reservieren (allokieren)
Prototyping einer Instanz ist möglich
es sind auch Instanzen von Script-Objekten erzeugbar (Konstruktor ist der Bezeichner des Script-Objektes)

z.B.

Objekt arguments
Objekt Array
Objekt Boolean
Objekt Date
Objekt Enumerator
Objekt Error
Objekt Function
Objekt Math
Objekt Number
Objekt Object (nicht Objekt obect des Internet Explorer für das HTML-Tag OBJECT)
Objekt RegExp
Objekt String
Objekt var

Achtung: Der Browser kann nur Objekte verarbeiten, die er kennt, z.B. ein Array Objekt, dessen Methoden und Eigenschaften dem Browser bekannt sind. Bei einem privaten Objekt müssen alle Eigenschaften und Methoden auf Script-Komponenten bestehen.
Jedes Objekt kann per Prototyping um Eigenschaften und Methoden erweitert werden, wenn es die Eigenschaft .prototype besitzt.
Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt !
Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .

Nur für Objekte, die aus einem vordefiniertem Objekt abgeleitet sind, existiert die Eigenschaft .prototype .

Punktnotation zum Zeiger:

```
zeiger.prototype.eigenschaft
zeiger.prototype.methode
```

Erweiterung eines Script-Objektes per

```
bezeichner_script_objekt.protoype.eigenschaft
bezeichner_script_objekt.protoype.methode
```

Eigenschaften und Methoden müssen auf Script-Elemente **basieren**, denn nur letztere kennt die Scriptmaschine des Browsers.

Syntax:

```
[ var Zeiger = ] new bezeichner [( [ArgumentenListe] )]
```

ArgumentenListe	siehe Beschreibungen der einzelnen Objekte Hinweis: Script-Objekte werden in der alphabetisch-sortierten Beschreibung der Objekte und Collectionen (z.T. browser-spezifisch) beschrieben.
Zeiger	wenn **null** (nicht numerisch 0 !!), so konnte das Objekt nicht erzeugt werden

Beispiele:

```
var InstanzVomTyp_Object = new Object;

var InstanzVomTypArray = new Array();

var InstanzVomTypeDate = new Date("Jan 5 1996");
```

Beispiel für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
<HTML>
<SCRIPT  LANGUAGE="JavaScript">
<!--
 function datenstruktur_ausgeben()
 {
          with (document)
```


```
		{	write(person.vorname + " " + person.nachname + "<BR>");
			write(person.strasse + " " + person.nummer + "<BR>");
			write(person.plz + " " + person.ort + "<BR>");
			//	Erika Mustermann
			//	Musterstrasse 1
			//	.....
		}

		for (i in person)
		{	document.write(i + ": " + person[i] + "<BR>");
			//	vorname: Erika
			//	nachname: Mustermann
			//	.....
		}
	}

	function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
	{
		this.vorname = vorname;
		this.nachname = nachname;
		this.strasse = strasse;
		this.nummer = nummer;
		this.plz = plz;
		this.ort = ort;
		this.methode = methode;
	}

	person = new datenstruktur_erzeugen
				(	"Erika",			// Vorname
					"Mustermann",		// Nachnahme
					"Musterstrasse",	// Strasse
					"1",				// Nummer
					"10000",			// PLZ
					"Musterstadt",		// Ort
					datenstruktur_ausgeben	// ohne () kodieren
				);

	// alternativ auch kodierbar:
	// var person = {	vorname:"Erika",		// Vorname
	//			nachname:"Mustermann",	// Nachnahme
	//			strasse:"Musterstrasse",	// Strasse
	//			nummer:"1",			// Nummer
	//			plz:"10000",			// PLZ
	//			ort:"Musterstadt"		// Ort
	//			methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
	//		};

	// Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !

	alert(person.vorname + "\n" + person.methode);	// person.methode ohne () kodieren !
	// -->
	</SCRIPT>
	</HTML>
```

Beispiel für Ableitung aus einem JScript-Objekt:

```
	var Kette = new String("Hi");
	alert( (Kette.constructor == String));	// liefert "true"
	alert( (Kette.constructor == "String"));	// liefert "false"
```

Beispiel für Ableitung anhand privaten Konstruktors:

```
	function TestFunktion()
	{alert("Hallo");}

	var ZeigerAufFunktion = new TestFunktion(); // bewirkt Ausführung von TestFunktion() also auch von alert()

	// ZeigerAufFunktion();	// nicht möglich und bringt Fehlermeldung wegen fehlernder Instanz,
				//	da keine Ableitung vom JScript-Objekt Function
				// Kodierung ohne ()  bringt keinen Fehler, da als Variablendeklaration erkannt
```


```
                alert(ZeigerAufFunktion.constructor == TestFunktion);          // true
                alert(TestFunktion.constructor == TestFunktion);               // false
```

return

Funktionsergebnis liefern
letzte Zeile innerhalb einer Funktion
Hinweis: Funktion muss kein return besitzen
siehe Anweisung function und Objekt Function

Syntax:

```
        return ([ausdruck]);

        return [ausdruck];
```

ausdruck — enthält zu lieferndes Funktionsergebnis
wenn nicht kodiert, wo wird undefined bzw. null geliefert

Beispiel:

```
        var GlobaleVariable1 = "Bitte laut";
        var GlobaleVariable2 = "lein";
        var GlobaleVariable3 = "";

        function GlobaleFunktion_ZeichenLiefern(ZeichenArt)          // ZeichenArt ist funktionslokal
        {
                var FunktionsLokaleVariable = ""                     // Initialisierung des String

                if (ZeichenArt = "Blank")
                { FunktionsLokaleVariable = " ";}

                if (ZeichenArt = "Doppelpunkt")
                { FunktionsLokaleVariable = ":";}

                return FunktionsLokaleVariable; // Funktion liefert Inhalt von FunktionsLokaleVariable
                                                //      und keinen Zeiger auf  FunktionsLokaleVariable,
                                                //      da ein Zeiger auf eine funktionslokale Variable
                                                //      niemals lieferbar ist: Zeiger existiert nur zur Laufzeit
                                                //      der Funktion !!
        }

        function Globale_BeispielFunktion(      FunktionsLokalesArgument1_Referenz,
                                                FunktionsLokalesArgument2_Wert_Numerisch
                                                FunktionsLokalesArgument2_Wert_String
                                        )
        {
                function LokaleFunktion_LeerZeichenLiefern()
                { return (GlobaleFunktion_ZeichenLiefern("Blank")); }        // liefert " "

                function LokaleFunktion_DoppelpunktLiefern()
                { return GlobaleFunktion_ZeichenLiefern("Doppelpunkt ");}    // liefert ":"

                var FunktionsLokaleVariable1 = "kleine";

                var FunktionsLokaleVariable2 =

                        // Referenz auf GlobaleVariable1
                          FunktionsLokalesArgument1_Referenz                 // "Bitte laut"

                        + LokaleFunktion_LeerZeichenLiefern()                // "Bitte laut "

                        + "mitsingen"                                        // "Bitte laut mitsingen"

                        + LokaleFunktion_LeerZeichenLiefern()                // "Bitte laut mitsingen "

                        + LokaleFunktion_DoppelpunktLiefern()                // "Bitte laut mitsingen :"

                        + LokaleFunktion_LeerZeichenLiefern()                // "Bitte laut mitsingen : "

                        // numerischen Wert 3 konvertieren zu "3"
                        + FunktionsLokalesArgument2_numerisch_Wert.toString()
                                                                             // "Bitte laut mitsingen : 3"

                        + LokaleFunktion_LeerZeichenLiefern()                // "Bitte laut mitsingen : 3 "
```


```
                                        // "kleine"
                                        + FunktionsLokaleVariable                   // "Bitte laut mitsingen : 3 kleine"

                                        + LokaleFunktion_LeerZeichenLiefern()       // "Bitte laut mitsingen : 3 kleine "

                                        // String-Wert  "Neger"
                                        + FunktionsLokalesArgument2_Wert_String;    // "Bitte laut mitsingen : 3 kleine Neger"

                                GlobaleVariable3 = FunktionsLokaleVariable2;        // Inhalt der lokalen Variablen nach
                                                                                    //          globale Variable kopieren
                                                                                    // man hätte auch return-Anweisung
                                                                                    //          nehmen können
                        }

                        // Aufruf mit korrekter Argumentenversorgung durch Parameterliste also
                        //                  GlobaleVariable1            "Bitte laut"
                        //                  Ausdruck 1 + 2              3
                        //                  "Neger"
                        //          Funktion belegt GlobaleVariable3 mit dem Wert "Bitte laut mitsingen : 3 kleine Neger"

                        Globale_BeispielFunktion(
                                                GlobaleVariable1,   // "Bitte laut"
                                                1 + 2,              // Ausdruck, der den Wert 3 liefert
                                                "Neger"
                                        );

                        // Anzeige per Alert-Box (Standard-Funktion) von  "Bitte laut mitsingen : 3 kleine Negerlein ..."
                        alert(
                                GlobaleVariable3                            // "Bitte laut mitsingen : 3 kleine Neger"

                                + GlobaleVariable2                          // "Bitte laut mitsingen : 3 kleine Negerlein"

                                + GlobaleFunktion_ZeichenLiefern("Blank")   // "Bitte laut mitsingen : 3 kleine Negerlein "
                                                                            // Achtung: Aufruf von LokaleFunktion_DoppelpunktLiefern()
                                                                            //          ist nicht zulässig

                                + "..."                                     // "Bitte laut mitsingen : 3 kleine Negerlein ..."
                        );
```

switch

Auswahl
siehe Anweisung break

Syntax:

```
        switch (referenz)
        {
                case erster_wert : { statements }

                        .....
                case letzter_wert : { statements }

                default : { statements }
        }
```

referenz	auf eine Variable etc.
erster_wert ... letzter_wert	zu referenz typengerechte **Werte,** anhand denen ausgewählt wird beliebig viele Wert möglich Wertelauswahl in der Kodierungsfolge innerhalb switch **und** wenn referenz den Wert hat Achtung: Es werden **alle** entsprechenden Werte ausgewählt **aber**: siehe statements mit break-Anweisung
default:	nur dann abgearbeitet, wenn kein einzigstes Label auswählbar war **aber**: siehe statements mit break-Anweisung
statements	optional (Empfehlung: mindesten Leeranweisung kodieren) wenn nur 1 Statement **dann** darf Blockanweisung { } entfallen muss Anweisung break enthalten als letzte Anweisung, wenn ausgeschlossenwerden soll, dass keine weiteren nachfolgenden Vergleiche der referenz mit Werten erfolgen **und auch nicht** default abgearbeitet werden sollen.

Beispiel:

```
var Index = "";
var Menge = {"a" : "Athen" , "b" : "Berlin", "c" : "Paris", "d" : "Kairo"};
var Kette = "";

SchleifenAnweisung :                    // ein Label (Marke)
{
    for (Index in Menge)
    {
        Kette = "Hauptstadt  von ";

        switch (Menge[Index])
        {
            case "Berlin":      {
                                    Kette += "Deutschland: " + Menge[Index];
                                    // nicht weiter auf Athen und Kairo prüfen
                                    break;
                                }

            case "Athen":       {
                                    Kette += "Griechenland: " + Menge[Index];
                                    // kein break, also noch auf Kairo prüfen
                                }

            case "Kairo":       {   Kette += "Ägypten: " + Menge[Index]; }

            default:            {
                                    // im Falle von Paris:
                                    //        Kairo wird nie angezeigt, da im Feld
                                    //        hinter Paris

                                    break SchleifenAnweisung;
                                }
        }

        {alert(Kette);}         // nicht bei Paris abgearbeitet
    }
}
```

this

Zeiger auf das aktuelle Objekt für Punktnotation
eigentlich keine Anweisung, da nicht mit Semikolon zu beenden ist

try catch finally

Fehlerbehandlung in Script (nicht Ereignisse von Objekten !)
Verschachtelung möglich
siehe auch error JScript-Objekt des Internet Explorer für private Run-Time-Error

Fehler: Runtime Error
oder per throw Anweisung erzeugter Error

Syntax:

```
try                 {tryStatements}
[catch(exception) {catchStatements}]
[finally            {finallyStatements}]
```

tryStatements	können Fehler erzeugen, der abgefangen werden soll
exception	freier Bezeichner als Platzhalter für den Fehler, der aus der Abarbeitung von tryStatements resultiert Variable wird automatisch gefüllt
catchStatements	exception auslesen und daraufhin eine Reaktion ausführen werden nur abgearbeitet, wenn Fehler auch wirklich auftrat in der Abarbeitung von tryStatements
finallyStatements	wird immer abgearbeitet, egal ob ein Fehler in der Abarbeitung von tryStatements und / oder catchStatements auftrat oder nicht

Beispiel:

```
function Anzeige(Kette)
```


```
{  alert(Kette); }

try {Anzeige("try1");
{
        try       { Anzeige("try 2");}
        catch(e) { Anzeige("catch2 " + e); }
        finally   { Anzeige("finally2"); }
}
catch(e) { Anzeige("catch1 " + e); }
finally  { Anzeige("finally1");}
```

Beispiel:

```
X04=false;
// +++++ CLASSID per try-catch zuweisen
try{X02.classid=X00;}catch(e){X04=true;}        // irgendwas tun
                                                // e ist Platzhalter für Fehlercode, der aber nicht ausgewertet
```

wird

```
// +++++ classid-Belegung prüfen
 if(X04){.....}                                 // anstelle e wird X04 ausgewertet
```

throw

freie Fehlerbedingung erzeugen für try...catch…finally
throw im try kodieren als freie Fehlerbedienung
catch auswerten

Syntax:

```
throw exception;

        exception  Wert oder Variable
```

Beispiel 1:

```
throw "Error2";     generiert userdefinierte Ausnahme "Error2"          als String
throw 42;           generiert userdefinierte Ausnahme 42        als numerischer Wert
throw true;         generiert userdefinierte Ausnahme  true     als Boolean
```

Beispiel 2:

```
function Anzeige(Kette)
{  alert(Kette); }

try {Anzeige("try1");
{
        try
        {
                throw "Das ist eine Fehlerbedingung";
                Anzeige("try 2");
        }
        catch(e)
        {
                if ( e == "Das ist eine Fehlerbedingung" )
                {Anzeige("catch2 " + e); }
        }
        finally { Anzeige("finally2"); }
}
catch(e)  { Anzeige("catch1 " + e); }
finally   { Anzeige("finally1");}
```

Beispiel 3:

```
function ErzeugeAusnahme Bedingung(Ausnahme Bedingung)
{
        this.Ausnahme Bedingung=Ausnahme Bedingung;
        this.AusnahmeArt="UserException";
}

function HoleMonatAlsString (MonatsNummer)  // liefert Monat als String
{
        var Index = MonatsNummer -1;

        var MonatsFeld =new Array("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec");

        if (MonatsFeld[Index] != null)
        {return MonatsFeld[Index];}
        else
        {
                var UserDefinierteAusnahmeBedingung =
                                new ErzeugeAusnahme Bedingung ("FalscheMonatsNummer");
```


```
                    throw UserDefinierteAusnahmeBedingung;
                }
        }
```

var

Deklaration einer Variablen
siehe auch Objekt var
Syntax:

```
var variable1 [ = value1 ] [, variablenliste [ = value2]];
```

variablenliste	Bezeichner mit Komma trennen alle Variablen erhalten einen gemeinsamen Wert laut value2

Beispiele:

```
var index;
var name = "Test";
var answer = 42, counter, numpages = 10;
```

while

tue etwas (do), solange (while) es erlaubt ist (also solange while den Wert true liefert)
siehe auch do .. while Anweisung, wenn mindestens 1 Durchlauf erwünscht ist
Syntax:

```
while (ausdruck) ;
{statements}
```

statements	wenn nur 1 Statement so kann Blockanweisung { } entfallen
ausdruck	muss true oder false liefern wenn true, so nächster Schleifendurchlauf wenn false, so Ende der Schleife

with

Instanz auf eine Anweisung zuordnen, die die Instanz verarbeiten soll
Syntax:

```
with (referenz)
        {statements}
```

statements	wenn nur 1 Statement so kann Blockanweisung { } entfallen

Beispiel:

```
with (Math)
{
        // alle Methoden entstammen dem Script-Objekt Math
        var x = cos(3 * PI) + sin (LN10) ;
        var y = tan(14 * E);
}
```

3.6.2. Anweisungen nur in Microsoft JScript

@cc_on und @end

Container für bedingtes Parsen per Scriptmaschine
immer innerhalb

```
/*   @*/
```

oder nach //
also innerhalb eines Kommentars kodieren, falls Browser nicht IE ist

bei // zwischen // und @ darf **kein Leerzeichen** liegen !

Empfehlung: kein Blank nach /* bzw. vor @*/

siehe auch @if und @set

Beispiel:

```
<SCRIPT>
/*@cc_on@*/
/*@if (  @_jscript_version >= 4  )
    {alert("JScript ab Version 4 oder höher");}
    @else @*/

  // nachfolgende Meldung kommt für alle Browser, die nicht bedingt parsen können
    alert("kein IE oder JScript unter Version 4, also kein bedingtes Parsen möglich");

/*@end@*/
</SCRIPT>
```

vordefinierte Variablen für das bedingte Parsen:
werden automatisch belegt mit true oder NaN

@_win32	true, so	Win32-System ist aktiv
@_win16	true, so	Win16-System ist aktiv
@_mac	true, so	Apple Macintosh-System aktiv
@_alpha	true, so	DEC Alpha Prozessor aktiv
@_x86	true, so	Intel Prozessor aktiv
@_mc680x0	true, so	Motorola 680x0 Prozessor aktiv
@_PowerPC	true, so	Motorola PowerPC Prozessor aktiv
@_jscript		immer true, da bedinge Kompilierung nur mit JScript läuft
@_jscript_build		Buildnummer der JScript-Scriptmaschine
@_jscript_version		Version der JScript-Scriptmaschine
		Aufbau major_nummer.minor_nummer

@if und @elif und @else und @end

Analogon zum Operator ?: und zur if-Anweisung
immer innerhalb

```
/*  @*/
```

oder nach //

also innerhalb eines Kommentars kodieren, falls Browser nicht IE ist

bei // zwischen // und @ darf **kein Leerzeichen** liegen !

Empfehlung: kein Blank nach /* bzw. vor @*/

Syntax:

```
@if (condition1)
script1
        [@elif (condition2) script2]
[@else script3]
@end
```

condition1 und 2 müssen boolean liefern

script1 bis 3 werden bedingt ausgeführt

@elif auch mehrfach kodierbar, aber alle **vor** einem eventuellen @else

Beispiel:

```
<SCRIPT>
/*  @cc_on                                  */
/*   @if  (  @_jscript_version >= 4  )
     {alert("JScript ab Version 4 oder höher");}
     @else                                  */

   // nachfolgende Meldung kommt für alle Browser, die nicht bedingt parsen können
     alert("kein IE oder JScript unter Version 4, also kein bedingtes Parsen möglich");


/* @end                                     */
</SCRIPT>
```

vordefinierte Variablen für das bedingte Parsen:
werden automatisch belegt mit true oder NaN

@_win32	true, so	Win32-System ist aktiv
@_win16	true, so	Win16-System ist aktiv
@_mac	true, so	Apple Macintosh-System aktiv
@_alpha	true, so	DEC Alpha Prozessor aktiv
@_x86	true, so	Intel Przessor aktiv
@_mc680x0	true, so	Motorola 680x0 Prozessor aktiv
@_PowerPC	true, so	Motorola PowerPC Prozessor aktiv
@_jscript		immer true, da bedinge Kompilierung nur mit JScript läuft
@_jscript_build		Buildnummer der JScript-Scriptmaschine
@_jscript_version		Version der JScript-Scriptmaschine
		Aufbau major_nummer.minor_nummer

@set

Erzeugung und Belegung einer globalen Variable (numerischer oder Boolean-Wert)

Verwendung und Operationen mit der Variablen **nur** innerhalb des bedingten Parsens möglich
immer innerhalb

/* @*/
oder nach //
also innerhalb eines Kommentars kodieren, falls Browser nicht IE ist

bei // zwischen // und @ darf **kein Leerzeichen** liegen !

Empfehlung: kein Blank nach /* bzw. vor @*/

mögliche Operatoren sind:

! ~ * / % + - << >> >>> < <= > >= == != === !== & ^ | && ||

Bezug auf eine nicht erzeugte Variable liefert immer NaN und keinen Fehler
Bsp: @if (@NichtexistenteVariable == NaN) .....

@if (@newVar != @newVar) ..... ist zulässig und prüft auf NaN

Syntax:

@set @varname = term

varname Aufbau des Namens wie bei Javascript-Bezeichnern

term Ergebnis bzw. Inhalt des Terms wird der Variablen zugewiesen
darf nicht String liefern
kann nur numerisch oder boolean sein

= Zuweisungsoperator

Beispiele:

```
@set @myvar1 = 12;

@set @myvar2 = (@myvar1 * 20);

@set @myvar3 = @_jscript_version:
```

vordefinierte Variablen für das bedingte Parsen:
werden automatisch belegt mit true oder NaN

@_win32	true, so	Win32-System ist aktiv
@_win16	true, so	Win16-System ist aktiv
@_mac	true, so	Apple Macintosh-System aktiv
@_alpha	true, so	DEC Alpha Prozessor aktiv
@_x86	true, so	Intel Prozessor aktiv
@_mc680x0	true, so	Motorola 680x0 Prozessor aktiv
@_PowerPC	true, so	Motorola PowerPC Prozessor aktiv
@_jscript		immer true, da bedinge Kompilierung nur mit JScript läuft
@_jscript_build		Buildnummer der JScript-Scriptmaschine
@_jscript_version		Version der JScript-Scriptmaschine
		Aufbau major_nummer.minor_nummer

3.6.3. Anweisungen in Javascript 1.5 im Netscape 6.x

break

const Deklaration einer Variable mit konstantem Inhalt, der nur lesbar ist
Bsp: const a = 7;

continue

do...while

export Methoden, Eigenschaften, Objekte und Funktionen eines signierten Scripts für ein anderes unsigniertes oder signiertes Script verfügbar machen, das import-Anweisung besitzen muss.

Syntax:

```
export liste
oder  export *
```

liste kommagetrennte Bezeichner von Methoden, Eigenschaften, Objekten und Funktionen

* für alle Methoden, Eigenschaften, Objekte und Funktionen im signierten Script

Beispiel:
zu exportieren sind im signierten Script die Eigenschaften f und p von obj

signierte Script: export obj.f, obj.p

anderes Script: import obj.f, obj.p

Hinweis: export obj schließt alle Eigenschaften etc. ein.

for
for...in
function
if...else
import Methoden, Eigenschaften, Objekte und Funktionen eines signierten Script, die dort per export für ein anderes unsigniertes oder signiertes Script verfügbar gemacht wurden, importieren

Syntax:

import liste
oder import *

liste kommagetrennte Bezeichner von Methoden, Eigenschaften, Objekten und Funktionen

* für alle Methoden, Eigenschaften, Objekte und Funktionen aus dem signierten Script

Beispiel:

zu exportieren sind im signierten Script die Eigenschaften f und p von obj

signierte Script: export obj.f, obj.p

anderes Script: import obj.f, obj.p

Hinweis: export obj schließt alle Eigenschaften etc. ein.

label
return
switch
throw
try...catch
var Deklaration einer les- und schreibbaren Variablen mit variablem Inhalt
while
with

3.7. Ausdruck (Expression)

Kombination aus Variablen, Operatoren, Literalen und Anweisungen
liefert immer Wert oder Referenz (Zeiger)

Beispiele:

```
4 + 5
x += 1
10 / 2
a & b
x++
var Wert = 3 * (4 / 5) + 6;
var Kette = "Test " + Wert.toString()+ " !";
var Feld = new Array("Hallo", Math.PI, 42.456);
var FeldElement = Feld[1];
```

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser leider **nicht als fehlerhaft** erkannt werden:

```
Bsp.:  var Kette =   'test';  + '.jpg';   // kein Syntaxerror wegen dem Semikolon vor dem Zeichen +
       alert(Kette);                      // zeigt nur  test   an  und nicht  test.jpg

Bsp:   for (var i = 0; i <3; i++);        // kein Syntaxerror wegen dem Semikolon vor dem Zeichen {
       {alert(i);}                        // genau 1 Anzeige, die 3 anzeigt
```

3.8. Funktion

basiert auf dem Script-Objekt Function und der Anweisung function

Hinweis: Bei Syntaxbeschreibungen bedeutet [ ] eine Optionalkodierung.

Funktion und Methode

Funktion ist synonym zu Methode, wenn es sich um eine Funktion eines Objektes handelt.

Beispiel für eine Methode eines Objektes:

```
var fenster=window.open(    url_der_html_datei,
                            name_der_html_datei,
```


```
                                        "toolbar=0,location=0,....,height=140"    // eine Zeile und keine Blanks !
                                );
```

Funktion vordefiniert oder frei deklariert

Vordefinierte Funktionen liegen als Methoden von Objekten vor.
Der Programmierer kann Funktionen frei deklarieren:
Eine frei deklarierte Funktion kann per Prototyping als Methode eines Objektes verwendet werden.

Beispiel für eine Funktion, die vom Programmierer definiert wurde:

```
function Test(Wert1, Wert2)
{
        var Summe = Wert1 + Wert2;
        alert (Summe.toString());
}
```

Funktion und Funktionswert

Eine Funktion **kann** einen Funktionswert liefern (siehe return Anweisung), muss aber nicht. Das zu kontrollieren, obliegt leider nur dem Programmierer, der also gewissenhaft kodieren muss.

Beispiel:

```
function Test(Wert1, Wert2)
{return ( Wert1 + Wert 2);}
```

Aufruf einer Funktion

Der Aufruf einer Funktion darf nur dort stattfinden, wo eine Referenz zulässig ist, da der Funktionsbezeichner einen Zeiger repräsentiert.
Z.B. ist der Aufruf einer Funktion innerhalb eines Literals unmöglich.

Falls die Funktion etwas liefert, gilt zusätzlich:
Anstelle des Funktionsaufrufes wird die gelieferte Größe gesetzt.
Die Funktion muss datentyp-gerecht liefern (entspricht einem typisierten Zeiger).

Analog gilt das auch für den Aufruf einer Methode.

Beispiel für Aufruf einer Funktion innerhalb eines Ausdruckes:

```
function Test(Wert1, Wert2)
{return ( Wert1 + Wert 2);}

var Wert = 20 + Test(10,20);

alert(Wert.toString());

// Nachfolgender Aufruf ergibt einen Fehler, da die Funktion einen numerischen Wert liefert
// Die Methode alert() kann eventuell automatisch nach String konvertieren
alert( "Das Ergebnis lautet " + Test(10,20));
```

Der Aufruf einer Funktion innerhalb eines HTML-Attributes anstelle des Wertes des Attributes ist zulässig, wenn die Funktion etwas liefert.

Beispiel für Aufruf einer Funktion im HREF-Attribut von z.B. <A> und <AREA>:

```
<A HREF="javascript:name_der_funktion(....);">
        .....
</A>
```

3.8.1. Funktion und optionale Argumente und Parameter

Eine Funktion kann Argument(e) optional im Funktionskopf kodiert bekommen.

Beispiel:

```
function Test(argument1, argument2)          // Funktionskopf
{
        // das ist der Funktionsrumpf
}
```

Vordefinierte Funktionen als Methoden eines Objektes haben in der Regel Argumente.

Argument und Argumentenliste

Argument ist der Platzhalter für den Wert, der per Funktionsaufruf mit dem Parameter übergeben wird
wird nicht var-deklariert im Funktionskopf (siehe Script-Objekt arguments)
wird nicht in " " bzw. ' ' kodiert
darf kein Schlüsselwort sein
darf keine Funktion sein
darf kein Ausdruck sein

Argument(e) sind optional kodierbar in der Funktionsdeklaration:
im Funktionskopf, also innerhalb von ()
als Liste von kommagetrennten Argumenten

Parameter und Parameterliste

Parameter ist bei Funktionsaufruf der konkrete Wert für den Platzhalter in Form des Argumentes
wird nicht var-deklariert innerhalb von () beim Funktionsaufruf
kann Ausdruck sein, der Zeiger oder Wert liefert
kann Aufruf einer Funktion sein, die per return Anweisung etwas liefern **muss.**
kann Wert sein z.B. String-Wert etc. (siehe Datentypen und Script-Objekte)
Wert muss zum Datentyp des Argumentes passen

Parameterliste Folgen von kommagetrennten Werten, die die Argumente füllen:
Die Folge der Listenelemente entspricht der Argumentenfolge in der Argumentenliste.
Es muss **jedes** Argument gefüllt werden (keine Lücken).

Aufruf einer Funktion mit Parameterliste

nur möglich, wenn Funktion mit Argument(en) im Funktionskopf deklariert wurde

Parameter werden **automatisch** mit dem Script-Objekt arguments (siehe dort) verarbeitet:

arguments Objekt ist ein Feld
dient als Schnittstelle für den Programmierer
Parameter in der Listenfolge von links nach rechts in das Feld automatisch abgelegt
mit Index ab 0 und aufsteigend
Anzahl der Parameter laut arguments.length

Bsp:
```
function test(arg1, arg2, arg3) {.....}
.....
test(10, 20, 30);
```

10 entspricht **test**.arguments[0] für arg1
20 entspricht **test**.arguments[1] für arg2
30 entspricht **test**.arguments[2] für arg3

Anzahl der Argument entspricht **test**.arguments.length und ist 3

Beispiel für Einlesen der Werte der Argumente aus dem Funktionsaufruf:

```
var GlobalesFeld = new Array();

function FunktionsBezeichner()
{
    // Argumenteliste der Funktion lokal zur Funktion referenzieren
    var ArgumentenListeAlsFeld = FunktionsBezeichner.arguments;

    // Anzahl der Argumente
    var ArgumenteAnzahl = ArgumentenListeAlsFeld.length;

    if ( ArgumenteAnzahl > 0 )
    {
        // Arrgumentenliste auslesen
        for (var i = 0; i < ArgumenteAnzahl; i++)
        {GlobalesFeld[i] = ArgumentenListeAlsFeld[i];}
    }

    // hier die weiteren Anweisungen der Funktion
}
```

3.8.2. Funktion und optionaler Funktionswert

Ein Funktionswert ist optional im Funktionsrumpf kodierbar. Dazu wird die return Anweisung verwendet:

return Anweisung:

Funktionsergebnis liefern
letzte Zeile innerhalb einer Funktion
Hinweis: Funktion muss kein return besitzen
siehe Anweisung function und Script-Funktion

Syntax:

return ([ausdruck]);

return [ausdruck];

ausdruck enthält zu lieferndes Funktionsergebnis
wenn nicht kodiert, wo wird undefined bzw. null geliefert

Beispiel:

```
var GlobaleVariable1 = "Bitte laut";
var GlobaleVariable2 = "lein";
var GlobaleVariable3 = "";

function GlobaleFunktion_ZeichenLiefern(ZeichenArt)          // ZeichenArt ist funktionslokal
{
    var FunktionsLokaleVariable = ""                         // Initialisierung des String

    if (ZeichenArt = "Blank")
    { FunktionsLokaleVariable = " ";}

    if (ZeichenArt = "Doppelpunkt")
    { FunktionsLokaleVariable = ":";}

    return FunktionsLokaleVariable; // Funktion liefert Inhalt von FunktionsLokaleVariable
                                    //      und keinen Zeiger auf  FunktionsLokaleVariable,
                                    //      da ein Zeiger auf eine funktionslokale Variable
                                    //      niemals lieferbar ist: Zeiger existiert nur zur Laufzeit
                                    //      der Funktion !!
}

function Globale_BeispielFunktion(      FunktionsLokalesArgument1_Referenz,
                                        FunktionsLokalesArgument2_Wert_Numerisch
                                        FunktionsLokalesArgument2_Wert_String
                                  )
{
    function LokaleFunktion_LeerZeichenLiefern()
    { return (GlobaleFunktion_ZeichenLiefern("Blank")); }        // liefert " "

    function LokaleFunktion_DoppelpunktLiefern()
    { return GlobaleFunktion_ZeichenLiefern("Doppelpunkt ");}    // liefert ":"

    var FunktionsLokaleVariable1 = "kleine";

    var FunktionsLokaleVariable2 =

        // Referenz auf GlobaleVariable1
          FunktionsLokalesArgument1_Referenz                     // "Bitte laut"

        + LokaleFunktion_LeerZeichenLiefern()                    // "Bitte laut "

        + "mitsingen"                                            // "Bitte laut mitsingen"

        + LokaleFunktion_LeerZeichenLiefern()                    // "Bitte laut mitsingen "

        + LokaleFunktion_DoppelpunktLiefern()                    // "Bitte laut mitsingen :"

        + LokaleFunktion_LeerZeichenLiefern()                    // "Bitte laut mitsingen : "

        // numerischen Wert 3 konvertieren zu "3"
        + FunktionsLokalesArgument2_numerisch_Wert.toString()
                                                                 // "Bitte laut mitsingen : 3"

        + LokaleFunktion_LeerZeichenLiefern()                    // "Bitte laut mitsingen : 3 "

        // "kleine"
```


```
                                + FunktionsLokaleVariable                   // "Bitte laut mitsingen : 3 kleine"

                                + LokaleFunktion_LeerZeichenLiefern()       // "Bitte laut mitsingen : 3 kleine "

                                // String-Wert  "Neger"
                                + FunktionsLokalesArgument2_Wert_String;    // "Bitte laut mitsingen : 3 kleine Neger"

                        GlobaleVariable3 = FunktionsLokaleVariable2;        // Inhalt der lokalen Variablen nach
                                                                            //          globale Variable kopieren
                                                                            // man hätte auch return-Anweisung
                                                                            //          nehmen können
                }

                // Aufruf mit korrekter Argumentenversorgung durch Parameterliste also
                //          GlobaleVariable1        "Bitte laut"
                //          Ausdruck 1 + 2          3
                //          "Neger"
                //      Funktion belegt GlobaleVariable3 mit dem Wert "Bitte laut mitsingen : 3 kleine Neger"

                Globale_BeispielFunktion(
                                        GlobaleVariable1,   // "Bitte laut"
                                        1 + 2,              // Ausdruck, der den Wert 3 liefert
                                        "Neger"
                                );

                // Anzeige per Alert-Box (Standard-Funktion) von  "Bitte laut mitsingen : 3 kleine Negerlein ..."
                alert(
                        GlobaleVariable3                    // "Bitte laut mitsingen : 3 kleine Neger"

                    + GlobaleVariable2                      // "Bitte laut mitsingen : 3 kleine Negerlein"

                    + GlobaleFunktion_ZeichenLiefern("Blank") // "Bitte laut mitsingen : 3 kleine Negerlein "
                                                            // Achtung: Aufruf von LokaleFunktion_DoppelpunktLiefern()
                                                            //          ist nicht zulässig

                    + "..."                                 // "Bitte laut mitsingen : 3 kleine Negerlein ..."
                );
```

3.8.3. Funktion vordefiniert

Vordefinierte Funktionen sind Methoden von in der Scriptmaschine definierten Objekten.

Bsp für Methode aller Script-Objekte:

eval() Zeichenkette aus Ziffern, Operatoren, Punkt, Komma und Vorzeichen nach numerisch konvertieren

3.8.4. Funktion durch Programmierer frei deklariert

Die Deklaration erfolgt anhand der Anweisung function oder einer Ableitung vom Script-Objekt Function

function Anweisung:

Deklaration einer frei-programmierten (privaten) Funktion
Verschachtelung möglich z.B. für Rekursion

siehe auch Objekt Function

Syntax:

```
function freier_funktions_bezeichner ([ argumenten_liste ])     // Kopf der Funktion per Anweisung function
{statements}                                                    // Rumpf der Funktion
```

freier_funktions_bezeichner — darf kein Schlüsselwort von Script sein

statements — Rumpf der Funktion
immer Blockanweisung kodieren
muss nicht return-Anweisung enthalten, kann aber
Hinweis: Eine PROCEDURE gibt es nicht in Script

argumenten_liste — Folge von per Komma getrennten Elementen
Listenelement:
immer Referenz
Platzhalter für Parameter (Wert oder Referenz)
Hinweis: intern wird Wert-Parameter auch

referenziert
Wert: z.B. numerisch, Boolean, String, Ausdruck
wird in statements verarbeitet
ist nur funktions-lokal gültig

```
var funktions_name = new Function(["argumenten_liste"],"javascript_anweisungen");
                    // Ableitung von Script-Objekt Function

                    javascript_anweisungen      sind der Funktionsrumpf
                                                in " "  bzw. ' ' zu setzen
                                                mit ; trennen
```

Folgendes Beispiel instanziert keine Funktion als Objekt:

```
        function TestFunktion()
        {alert("Hallo");}

        var ZeigerAufFunktion = new TestFunktion();
                                    // bewirkt sofortige Ausführung von TestFunktion() also von alert()
                                    //        und liefert keinen Zeiger

        // ZeigerAufFunktion();     // nicht möglich und bringt Fehlermeldung wegen fehlernder Instanz,
                                    //        da keine Ableitung vom JScript-Objekt Function,
                                    //        aber eine Funktion erwartet wird
                                    // Kodierung ohne ()  bringt keinen Fehler, da als Variablendeklaration
                                    //        erkannt
        // alert(ZeigerAufFunktion); liefert "(object Object)"
```

Beispiel 1 für Anweisung function:

```
var GlobaleVariable1 = "Bitte laut";
var GlobaleVariable2 = "lein";
var GlobaleVariable3 = "";

function GlobaleFunktion_ZeichenLiefern(ZeichenArt)        // ZeichenArt ist funktionslokal
{
        var FunktionsLokaleVariable = ""                   // Initialisierung des String

        if (ZeichenArt = "Blank")
        { FunktionsLokaleVariable = " ";}

        if (ZeichenArt = "Doppelpunkt")
        { FunktionsLokaleVariable = ":";}

        return FunktionsLokaleVariable; // Funktion liefert Inhalt von FunktionsLokaleVariable
                                    //        und keinen Zeiger auf  FunktionsLokaleVariable,
                                    //        da ein Zeiger auf eine funktionslokale Variable
                                    //        niemals lieferbar ist: Zeiger existiert nur zur Laufzeit
                                    //        der Funktion !!
}

function Globale_BeispielFunktion(      FunktionsLokalesArgument1_Referenz,
                                        FunktionsLokalesArgument2_Wert_Numerisch
                                        FunktionsLokalesArgument2_Wert_String
                                )
{
        function LokaleFunktion_LeerZeichenLiefern()
        { return (GlobaleFunktion_ZeichenLiefern("Blank")); }          // liefert " "

        function LokaleFunktion_DoppelpunktLiefern()
        { return GlobaleFunktion_ZeichenLiefern("Doppelpunkt ");}      // liefert ":"

        var FunktionsLokaleVariable1 = "kleine";

        var FunktionsLokaleVariable2 =

                // Referenz auf GlobaleVariable1
                  FunktionsLokalesArgument1_Referenz          // "Bitte laut"

                + LokaleFunktion_LeerZeichenLiefern()         // "Bitte laut "
```

```
                                    + "mitsingen"                                   // "Bitte laut mitsingen"

                                    + LokaleFunktion_LeerZeichenLiefern()           // "Bitte laut mitsingen "

                                    + LokaleFunktion_DoppelpunktLiefern()           // "Bitte laut mitsingen :"

                                    + LokaleFunktion_LeerZeichenLiefern()           // "Bitte laut mitsingen : "

                                    // numerischen Wert 3 konvertieren zu "3"
                                    + FunktionsLokalesArgument2_numerisch_Wert.toString()
                                                                                    // "Bitte laut mitsingen : 3"

                                    + LokaleFunktion_LeerZeichenLiefern()           // "Bitte laut mitsingen : 3 "

                                    // "kleine"
                                    + FunktionsLokaleVariable                       // "Bitte laut mitsingen : 3 kleine"

                                    + LokaleFunktion_LeerZeichenLiefern()           // "Bitte laut mitsingen : 3 kleine "

                                    // String-Wert  "Neger"
                                    + FunktionsLokalesArgument2_Wert_String;        // "Bitte laut mitsingen : 3 kleine Neger"

                            GlobaleVariable3 = FunktionsLokaleVariable2;            // Inhalt der lokalen Variablen nach
                                                                                    //          globale Variable kopieren
                                                                                    // man hätte auch return-Anweisung
                                                                                    //          nehmen können
                    }

                    // Aufruf mit korrekter Argumentenversorgung durch Parameterliste also
                    //                  GlobaleVariable1            "Bitte laut"
                    //                  Ausdruck 1 + 2              3
                    //                  "Neger"
                    //          Funktion belegt GlobaleVariable3 mit dem Wert "Bitte laut mitsingen : 3 kleine Neger"

                    Globale_BeispielFunktion(
                                            GlobaleVariable1,   // "Bitte laut"
                                            1 + 2,              // Ausdruck, der den Wert 3 liefert
                                            "Neger"
                                        );

                    // Anzeige per Alert-Box (Standard-Funktion) von  "Bitte laut mitsingen : 3 kleine Negerlein ..."
                    alert(
                            GlobaleVariable3                        // "Bitte laut mitsingen : 3 kleine Neger"

                        + GlobaleVariable2                          // "Bitte laut mitsingen : 3 kleine Negerlein"

                        + GlobaleFunktion_ZeichenLiefern("Blank")   // "Bitte laut mitsingen : 3 kleine Negerlein "
                                                                    // Achtung: Aufruf von LokaleFunktion_DoppelpunktLiefern()
                                                                    //          ist nicht zulässig

                        + "..."                                     // "Bitte laut mitsingen : 3 kleine Negerlein ..."
                    );
```

Beispiel 2 für Ableitung vom Script-Objekt function:

```
var Zeiger = new Function("return init_wert;");
```

Beispiel 3 für Rekursion (Sound mit Sekundenanzeige):

```
<HTML>
<HEAD>
<SCRIPT>
        // ++++++++++++ globale Variablen, die verändert werden können
        var SoundUrl                = "56sec.mid";
        var SoundDauerInSekunden    = 56; // Dauer muss exakt stimmen !

        var PixelBreiteProBalkenErweiterung = 10;

        // ++++++++++++ Browser-Typ ermitteln
        // Dieser Quellcode muss VOR allen anderen Routinen codiert sein, damit zuerst abgearbeitet
        var ns = document.layers ? true : false;
```


```
var ie = document.all ? true : false;

// ++++++++++++ Routinen der Sekundenzählung
var SekundenZahler              = 0;
var SekundenZahlerTimoutID      = null;

function SekundenZaehlen()      // wird durch Rekursion alle Sekunde neu gestartet
{
        // Zähler erhöhen
        SekundenZahler++;

        // visuelle Anzeige im Dokument auffrischen, also alle Audrucke der Style-Werte
        //            neu berechnen und damit alle DIV's neu visualisieren
        document.recalc();
}

function SekundenZaehlen_Start()
{
        // prüfen ob Sekundenzählen nicht bereits läuft
        if (SekundenZahlerTimoutID == null)
        {
                // nicht aktiv

                // Rekursion starten: SekundenZaehlen() wird permanent alle Sekunde aktiviert
                SekundenZahlerTimoutID = setInterval("SekundenZaehlen()", 1000);
        }
}

function SekundenZaehlen_Stop()
{
        // prüfen ob Sekundenzählen aktiv ist
        if (SekundenZahlerTimoutID != null)
        {
                // aktiv, also stoppen
                clearInterval(SekundenZahlerTimoutID);
                SekundenZahlerTimoutID = null;
        }
}

// ++++++++++++ Routine zur Erzeugung Sound-Objekt
function SoundObjektErzeugen(SoundFileUrl,SoundFileDauerInSekunden)
{
        this.SoundFileUrl                               = SoundFileUrl;
        this.SoundFileDauerInMillisekundeSekunden = SoundFileDauerInSekunden  * 1000;
                                                                // Timerzeit für Rekursion
        this.SoundFileBeendet                           = true;    // kein Sound aktiv
}

// ++++++++++++ Routinen zur Wiedergabe Sound-Objekt
var SoundTimeoutID=0;

function SoundAbspielen()
{
        // prüfen ob Sound nicht bereits aktiv ist
        if (SoundObjekt.SoundFileBeendet)
        {
                // Anzeige intialisieren
                SekundenAnzeigeInit();

                // Sekundenzähler starten, wobei das Zählen eigenständig und parallel erfolgt
                SekundenZaehlen_Start();

                // Sound erzeugen und sofort starten durch  Url-Zuweisung
                ID_BGSound.src=SoundObjekt.SoundFileUrl ;
                SoundObjekt.SoundFileBeendet=false;

                // Millisekunden warten und danach die Funktion SoundAbspielen() neu aufrufen
                SoundTimeoutID = setTimeout(
                                "SoundAbspielen()",
                                SoundObjekt.SoundFileDauerInMillisekundeSekunden
                                );
```


```
		}
		else
		{
			// Dieser Zweig wird erst mit dem 2. Aufruf der Funktion abgearbeitet

			// Sound zu Ende
			SoundObjekt.SoundFileBeendet=true;

			// Sekundenzähler stoppen
			SekundenZaehlen_Stop();

			// und Meldung
			var TimerUngenauigkeit1 = SoundDauerInSekunden - SekundenZahler;
			var TimerUngenauigkeit2 = TimerUngenauigkeit1 / SoundDauerInSekunden;
			alert(		"Wiedergabe beendet\nUngenauigkeit des Timers = "
				+ TimerUngenauigkeit1.toString()+ " Sekunden\n"
				+ "also " + TimerUngenauigkeit2.toString() + " Ticks pro Sekunde"
			    );
		}
}

// ++++++++++++ Sekunden-Anzeige initialiseren
function SekundenAnzeigeInit()
{
		// ----- Variablen init
		SekundenZahler			= 0;
		SekundenZahlerTimoutID		= null;

		// ----- visuelle Anzeige erzeugen
		// - - - Sekundenbalken und Sekundenzähler dynamisch visualisieren
		//		Es wird jedem DIV als Style-Wert ein Ausdruck hinterlegt, also kein Wert.
		//			Der Ausdruck liefert den Wert , welcher sofort das Layout der
		//			DIV's beeinflusst.
		//		Jeder Ausdruck besitzt den SekundenZahler als Komponente.
		//			Damit ändert sich der Wert des Ausdruckes.
		//		Für die Neuberechnung des Ausdruckes ist der Aufruf von
		//					document.recal()
		//			nötig.
		//			Dieser Aufruf erfolgt in SekundenZaehlen(), also permanent pro Sekunde.
		//			Damit wird der Style-Wert permanent neu berechnet.
		//			Damit visualisieren sich die DIV's permanent neu.

		// Sekundenbalken in der Style-Eigenschaft width (Breite) mit Ausdruck belegen,
		//		also dynamisch anzeigen
		ID_DIV_Balken.style.setExpression(  "width",
						    "SekundenZahler * PixelBreiteProBalkenErweiterung"
						  );

		// Sekundenzähler  in der  Eigenschaft .innerText mit Ausdruck belegen,
		//		also dynamisch anzeigen
		ID_DIV_SekundenZahler.setExpression("innerText","SekundenZahler.toString()");

		// - - - Messlatte statisch anzeigen
		ID_DIV_MessLatte.style.width  = SoundDauerInSekunden * PixelBreiteProBalkenErweiterung;
		ID_DIV_MessLatte.innerText    =    "Der Sound dauert "
						   + SoundDauerInSekunden.toString()
						   + " Sekunden";
}

// ################# Dieser Teil wird mit dem Laden des Dokumentes abgearbeitet #################
if (ie)
{
		document.write('<BODY></BODY>');

		document.write(		'<BGSOUND ID= "ID_BGSound" LOOP="0"">'

		document.write(		  '<DIV     ID="ID_DIV_Balken"'
					+	  'STYLE="background-color:lightblue"'
					+ '>'
					+ '</DIV>'
					+ '<BR>'
```


```
                                        );

                                document.write(     '<DIV     ID="ID_DIV_SekundenZahler"'
                                                  +           'STYLE="color:hotpink;font-weight:bold"'
                                                  + '>'
                                                  + '</DIV>'
                                                  + '<BR>'
                                        );

                                 document.write(     '<DIV     ID="ID_DIV_MessLatte"'
                                                  +           'STYLE="color:white;background-color:gray"'
                                                  + '>'
                                                  + '</DIV>'
                                         );

                                // ++++++++++++ Sound initialisieren und starten mit Laden des Dokumentes

                                //  ----- Sound-Objekt erzeugen anhand globaler Variablen
                                SoundObjekt = new SoundObjektErzeugen(SoundUrl, SoundDauerInSekunden);

                                // ----- Sound-Objekt wiedergeben
                                SoundAbspielen();  // meldet wenn Wiedergabe beendet ist
                        }
                </SCRIPT>
                </HEAD>
                <BODY>
                        <! -- BODY-Teil muss leer bleiben //-->
                </BODY>
                </HTML>
```

3.8.5. Funktion als Objektkonstruktor (Objektklasse) per new

Durch die Definierung einer Funktion als Konstruktor, kann ein Objekt erzeugt werden, dessen Eigenschaften und Methoden durch den Konstruktor festgelegt werden, also durch Protoyping. Aber innerhalb der Funktion entfällt die Eigenschaft .prototype, da das Objekt ja beim Instanzieren und **nicht** nachträglich erweitert wird.

siehe new Anweisung

Man beachte, dass eine Funktion selbst als Objekt per new instanziert werden kann:

Folgendes Beispiel instanziert keine Funktion als Objekt:

```
function TestFunktion()
{alert("Hallo");}

var ZeigerAufFunktion = new TestFunktion();
                                // bewirkt sofortiges Ausführung von TestFunktion() also von alert()

// ZeigerAufFunktion();         // nicht möglich und bringt Fehlermeldung wegen fehlernder Instanz,
                                //          da keine Ableitung vom JScript-Objekt Function,
                                //          aber ein Funktion erwartet wird
                                // Kodierung ohne ()  bringt keinen Fehler, da als Variablendeklaration
                                //          erkannt
// alert(ZeigerAufFunktion);    liefert "(object Object)"
```

Beispiel 1:

```
function erzeuge_zwei_dim_feld(anzahl_spalten, anzahl_zeilen)
{
        // Spaltenfeld erzeugen
        this.spalten_feld= new Array(anzahl_spalten);

        // Zeilenfeld pro Spalte  erzeugen
        for (var spalte=0; spalte < anzahl_spalten; spalte++)
        { this[spalte] = new Array(anzahl_zeilen);}  //Zeilen-Felder
}
var zwei_dim_tabelle= new erzeuge_zwei_dim_feld(10,7); // 10 Spalten zu je 7 Zeilen
....
zwei_dim_tabelle[10,7]=22;   // Spalte 10: 7Zeile: mit  22 belegen
```

Beispiel 2:

```
function Konstruktor()
{
```


```
                // objektinterne Rechen-Methode definieren
                function Addition(operand1,operand2)
                {return (operand1 + operand 2);}

                // objektinternes Objekt, das zur Laufzeit erzeugt wird
                //      Rechenobjekt definieren, dass nur über die objekteigene Methode erreichbar ist
                var internes_rechen_objekt = new Object(); // Script-Objekt mit Eigenschaft .prototype

                // objektinternes Objekt erweitern per Protoyping
                //      keine Eigenschaften zu definieren, da diese aus Funktionsargumenten operand1 und operand2
                //          by Value übernommen werden
                //      Methode dem internen Objekt zuweisen
                //          Achtung: ohne () kodieren, damit nicht sofort ausgeführt wird
                internes_rechen_objekt.prototype.methode= Addition;
        }

        // Objekt instanzieren
        var Zeiger = new Konstruktor();

        // Objekt anwenden
        alert(zeiger.Addition(10,20));     //  zeigt 30 an
```

Beispiel 3:

```
        var Zeiger = new Function("return init_wert;");
```

3.8.6. Funktion und Abarbeitungsfolge z.B. bei Rekursion

Grundsätzlich wird ein Dokument zwar sequentiell in der Vorgabefolge des HTML-Interpreters vom Browser gelesen (erst HEAD-Teil, dann der BODY-Teil), aber die Abarbeitung von ausgelösten Aktionen erfolgt immer parallel: Zwei Javascript-Funktionen, die nacheinander im HEAD-Teil des HTML-Dokumenten-Quelltextes kodiert sind, werden nacheinander aufgerufen, arbeiten aber parallel (Multitasking unter Windows).

Der Versuch, durch Schleifen wie for(..) etc. ein Warten der Routinen aufeinander auszulösen, wird mit sofortigem Stillstand des Betriebsystems Windows 9x quittiert, wenn die Schleife z.B. endlos ist. Der Grund ist einfach: In der Schleife finden Zählerveränderungen statt - meist numerische. Diese werden von Windows 9x nicht als Multitasking realisiert sondern als Singletask (Windows 9x kann kein echtes Multitasking). Das ist auch sinnig, da Zählerschleifen direkt in Maschinencode und CPU-Register untergebracht werden können, also wenig Ressourcen benötigen.

Die Realisierung von abhängigen Routinen, die z.B. aufeinander warten sollen, ist also nicht durch Warteschleifen möglich sondern nur durch Rekursion. Aber: Rekursionen werden wieder parallel abgearbeitet. Daher ist es nötig, die abhängigen Routinen zu verschachteln oder eine zu den Routinen globale Schaltvariable zu instanzieren, die diese Routinen abfragen. Letztere Variante kann aber nicht garantieren, dass die Routinen nacheinander zugreifen, wenn nicht eine von den beiden Routinen eine längere Wartezeit hat. Diese ist z.B. durch die Methode setTimeout() einrichtbar. Die Methode setIntervall() funktioniert wie ein Herzschlag, also endlos bis zum bewussten Abschalten. Die Methode setTimeout() muss immer wieder neu gestartet werden.

Rekursion z.B. per setTimeout() hat einen Nachteil: Es werden die Timer des Betriebssystems benutzt, deren Anzahl begrenzt ist. Je mehr Rekursionen parallel laufen, um so weniger können Betriebsystem-Routinen zugreifen, da diese sich einen Timer teilen müssen. Die Konsequenz: Es kann passieren, dass die Uhr im PC falsch geht, da die Uhr selbst einen immer verfügbaren Timer benötigt. Das Zuweisen von bestimmten Timern ist nicht möglich.

Grundsätzlich sollte für eine Rekursion mit setTimeout() beachtet werden, dass im Programmablauf **nach** Aufruf der Timeout-Anweisung die rekursive Funktion endet und sich somit **unmittelbar** neu startet. Liegen im Programmablauf hinter der Timeout-Anweisung auf **gleicher** Ebene zu ihr noch andere Anweisungen, so wird **mittelbar** rekursiv verfahren: Diese Anweisungen werden **ebenfalls** abgearbeitet, obwohl der Timer bereits aktiv ist oder die Funktion bereits neu gestartet wurde. Wird mit diesen Anweisungen auch noch die Ablaufsteuerung der rekursiven Funktion z.B. in den Bedingungen der Aktivierung des **nächsten** setTimeout() **nachträglich** zur bereits getimten Rekursion beeinflusst, so findet die Funktion bei Rekursion eventuell nicht denjenigen Stand an Daten in Variablen vor, der zum Zeitpunkt des Timerstartes per setTimeout() vorlag und eigentlich erwartet wird. Das kann fatale Folgen für den Programmablauf der Funktion haben: Datenbestände werden also pro Rekursion unerwartet verändert, so dass ein definiertes Ende der letzten Rekursion unvorhersehbar ist. Bei Timerschleifen, die in einem gewissen Zeitraum nicht enden, reagiert entweder der Browser mit einer Empfehlung, dass aktive HTML-Dokument zu beenden, oder Windows stürzt eventuell ab. Als Analogon dient die for-Schleife: Diese Schleifenart wird nicht parallel verarbeitet, sondern zeitecht. Eine for-Schleife, die in ihrer Abarbeitung zeitlich zu lang ist oder nie endet, kann die Performance des Browsers und eventuell des Betriebssystems stark behindern.

Das o.g. Problem tritt bei der Verwendung von setInterval() auch auf, wenn nicht statische Programmabläufe per Zeitintervall aktiviert werden. Es empfiehlt sich, **dynamische** Programmabläufe nicht per setInterval() sondern per setTimeout() zu programmieren, falls möglich. Rekursionen mit setIntervall() enden nie, wenn nicht der zugehörige Timer irgendwann gelöscht werden **kann**. setTimeout() verursacht immer einen einmaligen Aufruf, also keine Intervallfolge wie setIntervall().

Variablen innerhalb einer rekursiven Funktion sollten **nicht in ihr** deklariert werden, sondern global zur Funktion. Der Vorteil ist, dass die Funktion definitiv immer auf die **selbe** Instanz zugreift. Analog dazu sollte auf Parameter der rekursiven Funktion verzichtet werden, denn damit vereinfacht sich die Adressberechnung zur Rekursion: Eine außerhalb der Funktion deklarierte Variable hat bereits **vor Aufruf** der Funktion eine feste Adresse, so dass der Interpreter des Browsers nur noch diese in die Rekursion einzusetzen braucht und keine

Adressberechnungen durchführen muss. Das erhöht die Browserperformance. Allerdings hat diese Vereinfachung auch einen Haken: Sollte die globale Variable der Funktion dummerweise auch ausserhalb der Funktion angesprochen werden, z.B. weil der Programmierer zwar 2 **verschiedene** Variablen meint, aber diese, ohne es zu bemerken, mit **gemeinsamen** Namen versehen hat, dann kann die Rekursion eventuell fehl schlagen. Daher der Tipp: Variablenbezeichner der rekursiven Funktion immer mit dem Funktionsnamen verbinden, so dass Eineindeutigkeit und damit Übersicht herrschen. Wenn möglich, sind diese Variablen unmittelbar vor dem Funktionskopf zu deklarieren.

3.8.7. Funktion und Rekursionen

3.8.7.1. Übergabe von Variablen an die Rekursion

Mit Start der Rekursion als Argumente an die Routine übergebene Variablen sind routinen-lokal und müssen bei setTimeout() immer wieder neu übergeben werden, d. h., sie werden **pro Aufruf** weitergereicht. Natürlich sind globale Variablen stets innerhalb der Rekursion gültig, also verarbeitbar.

3.8.7.2. Rekursion und document.write() bzw. document.writeln() im HEAD

Eine Rekursion kann anhand von Funktionscode im HEAD realisiert werden.

Beim Internet Explorer ist unbedingt zu beachten:

Die Ausführung von document.write() bzw. document.writeln(), das **HTML-Tags** erzeugt, hat 2 Konsequenzen und zwar genau dann, wenn diese Anweisung **nach dem kompletten Laden des Dokumentes, also nach Auslösung des Ereignisses onload** aktiviert wird:

Fakt 1:

Die **ERSTE** Erzeugung eines HTML-Tags bewirkt das **automatische Öffnen eines neuen** HTML-Dokumentes, wenn das aktuelle Dokument **bereits komplett geladen**, also das Ereignis onload **bereits** ausgelöst wurde. Letzteres ist immer dann der Fall, wenn der BODY-Teil des Dokumentes komplett geparst wurde. Grund: Ein komplett geparstes Dokument kann per write() bzw. writeln() nicht um HTML-Elemente verändert werden, da diese Methoden keine des HTML-DOM sind. Mit anderen Worten: Nur die Verwendung von Methoden des HTML-DOM lassen eine **nachträgliche** HTML-Elemente-Veränderung des Dokumentes zu.

Die Methoden write() und writeln() erzeugen einen Datenstrom aus HTML-Elementen und das neue Dokument empfängt diesen so, als würde er aus einer HTML-Datei stammen. Mit Ende des Datenstromes wird quasi ein Dateiende erkannt und das neue Dokument löst das Ereignis onload aus. Damit wird aber das neue Dokument zum aktuellen Dokument, also im Fenster über dem des alten Dokumentes angezeigt. Da das Fenster des neuen Dokumentes automatisch erzeugt wurde (nicht per Methode open()), sind also die Standards für eine Fenstererzeugung verwendet worden. Damit hat das neue Dokument einen History-Eintrag. Der User kann nun mit diesem zwischen dem alten und neuen Dokument umschalten.

Fakt 2:

Im Falle der o.g. nachträglichen Veränderung des Dokumentes um HTML-Elemente per wirte() bzw. writeln() kennt das neue, automatisch geöffnete Dokument das alte Dokument nur **als Eltern**. Es muss also im neuen Dokument mit dem Zeiger auf die Eltern gearbeitet werden, wenn Daten und Routinen der Eltern benutzt werden sollen (siehe Objekt window bzw. Objekt document). Mit anderen Worten: Das neue Dokument muss dann komplett per Script erzeugt werden, denn dieser Zeiger lässt sich nur über Script ansprechen. Jedes geladene Dokument hat ansonsten seine eigenständige Umgebung.

Eine Rekursion im alten Dokument, die nach dem Laden des Dokumentes HTML-Tags per Script erzeugt, ist im neuen, automatisch erzeugten Dokument leider nicht bekannt. Man beachte auch: **Jeder** Aufruf der rekursiven Methode erzeugt erneut HTML-Code. Die Rekursion einer Methode sollten also keinen HTML-Code erzeugen, wenn das Dokument, in dem die Rekursion kodiert ist, bereits komplett geparst wurde.

3.1.9. ASCII-Code und Unicode

ASCII: pro Zeichen 7 Bit verwendet, also 128 Zeichen möglich (0-127)
enthalten im Unicode

Eurozeichen ab HTML4.0 kodierbar per &#8364 oder &#20AC oder &euro

Unicode: pro Zeichen 16 Bit verwendet, also 65535 Zeichen möglich
Zeichen 0-127 identisch mit ASCII-Code

Kodierung als ESC-Sequenz \uxxxx mit xxxx als Hexaziffern

Beispiele: \u0008 Rückschritt

\u0009	horizontaler Tabulator
\u000A	Zeilenvorschub (Line Feed) neue Zeile
\u000B	vertikaler Tabulator
\u000C	Seitenvorschub (Form Feed)
\u000D	für Wagenrücklauf
\u0020	Blank, Leerzeichen
\u0022	"
\u0027	'
\u005C	\

Escape Sequenzen

\b	Backspace, Rückschritt
\f	Form feed, Blattvorschub
\n	Line feed (newline), Zeilenvorschub
\r	Carriage return , Wagenrücklauf (nicht Enter !!)
\t	Horizontal tab (Ctrl-I), horizontaler Tabulator
\'	Entwertung des Zeichen '
\"	Entwertung des Zeichen "
\\	Backslash z.B. für lokale Pfadangabe wie C:\\test\\bilder\\test.gif.
\\\\	Entwertung Backslash
\xhh	hexadezimal aber nur von 00 bis FF
\uhhhh	Unicode

Beispiele:

\u0008	Rückschritt
\u0009	horizontaler Tabulator
\u000A	Zeilenvorschub (Line Feed) neue Zeile
\u000B	vertikaler Tabulator
\u000C	Seitenvorschub (Form Feed)
\u000D	für Wagenrücklauf
\u0020	Blank, Leerzeichen
\u0022	"
\u0027	'
\u005C	\

nicht druckbare Zeichen

z.B. siehe Machcode der Objekte regexp bzw. RegExp

3.10. Fehlerbehandlung in Javascript

Es ist darauf zu achten, dass der Browser die HTML-Dokumente nicht aus dem Browser-Cache liest, solange die Dokumente nicht fehlerfrei laufen, also beim Test der Dokumente (Daten im Browser-Cache vor jedem Test löschen).

Das generelle Lesen aus dem Cache ist per Script, META-Tag, sowie per Browsereinstellung abänderbar. Letztere kann der User treffen, in dem er z.B. beim Internet Explorer die Cache-Löschung mit Schließen des Browser abhakt.

Es ist üblich, dass der User den Browser-Cache **nicht löscht**, da der Browser den Cache-Füllstand automatisch verwalten kann. Daher muss der Programmierer damit rechnen, dass die HTML-Dokumente aus dem Cache geladen werden könnten. Deshalb ist es wichtig, dass der Programmierer der Webseiten dafür sorgt, dass per META-Tag immer ein Datum der Webseite besteht, das mit Sicherheit verfallen ist und somit die HTML-Daten vom Server geladen werden. Achtung: Des erneute Laden einer Webseite (Reload), z.B. innerhalb eines bestimmten Zeitraumes, sollte aus dem Cache kommen, damit nicht Daten doppelt zu laden sind, wenn sie bereits gültig im Cache liegen. Will der Programmierer absolut sichergehen, dass nach einem bestimmten Zeitraum die Daten vom Server zu laden sind **und** soll der User dieses Verhalten nicht beeinflussen können, dann muss das Verhalten per Script programmiert werden (Algorithmus ist Sache des Programmierers). Zugleich wird der User und der Server mehr Traffic beim Laden der Daten der HTML-Dokumente haben.

Fehlermeldung beim Internet Explorer zu Objekten:

z.B. "Objekt ..... nicht gefunden" oder " Objekt ... unterstützt Eigenschaft nicht" oder "CLASS nicht gefunden"

Die Fehler, die per Fehlermeldungen zu Objekten im Internet Explorer angezeigt werden, können dessen im Speicher instanzierte Objektstruktur durcheinander bringen, so dass nach Abhilfe des Fehlers im Quelltext und anschließender Aktualisierung des Dokumentes (per Aktualisierungs-Button im Browsermenü) trotzdem dieselbe Fehlermeldung erscheint. Es ist daher ratsam, bei solchen Fehlermeldungen den Browser zu schließen, den Browser-Cache zu löschen und dann den Browser mit dem Quelltext neu zu starten. Ursache ist ein eventuelles automatisches aber fehlerhaftes Prototyping von Objekten (auch von browserinternen). Hinweis: Man sollte immer mit genau 1 Instanz des Browsers testen, damit Objektstrukturen konsistent bleiben können.

Fehlermeldungen zum Script

z.B. "Scriptfehler ...."

Fehlermeldung kann echten Fehler anzeigen (z.B. wegen falscher Browserversion) **oder** Speichermangel, welcher die Abarbeitung des Skriptes verhindert (nicht mögliche Instanzierung von Objekten, Variablen etc.). Bei Speichermangel kann der Neustart des Browsers helfen.

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser leider **nicht als fehlerhaft** erkannt werden:

```
Bsp.:   var Kette =    'test';  + '.jpg';      // kein Syntaxerror wegen dem Semikolon vor dem Zeichen +
        alert(Kette);                          // zeigt nur   test    an  und nicht   test.jpg

Bsp:    for (var i = 0; i <3; i++);            // kein Syntaxerror wegen dem Semikolon vor dem Zeichen {
        {alert(i);}                            // genau 1 Anzeige, die 3 anzeigt
```


3.10.1. Runtime Fehler (Laufzeitfehler) von JScript (Auswahl)

siehe auch error JScript-Objekt des Internet Explorer für private Run-Time-Error

5000	Zeiger laut this Anweisung nicht zuweisbar	
5001	Instanz vom Objekt Number erwartet	
5002	Instanz von Objekt Function erwartet	
5003	Funktionsergebnis nicht zuweisbar	
5005	Instanz von Objekt String erwartet	
5006	Instanz von Objekt Date erwartet	
5007	Instanz (Objekt) erwartet	
5008	unzulässige Zuweisung	
5009	undefinierter Bezeichner	
5010	Instanz von Objekt Boolean wird erwartet	
5012	Instanz vom Object member erwartet	
5013	Instanz von Objekt VBArray erwartet	
5014	Instanz eines JScript-gültigen Objektes erwartet	
5015	Instanz von Objekt Enumerator erwartet	
5016	Instanz von Objekt Regular Expression erwartet	
5017	Syntax Fehler im regulären Ausdruck	
5018	undefinierte Qualifizierung (Punktnotation)	
5019	']' erwartet im regulären Ausdruck	
5020	')' erwartet im regulären Ausdruck	
5021	falscher Mengenbereich im Character Set	
5022	Ausnahme per thrown Anweisung erwartet	
5023	Funktion hat kein zulässiges Prototyping (Funktionskopf falsch)	
5024	URI enthält fehlerhaftes Zeichen	
5025	URI ist ungültig für encoding	
5026	Anzahl der Ziffern hinter dem Komma ist ungültig	
5027	Bereichgrenze überschritten	
5028	Instanz von Objekt Array oder Objekt arguments erwartet	
5029	Feldlänge ist	nicht >= 0 nicht Integer nicht endlich
5030	Feldlänge ist	nicht >= 0 nicht Integer

3.10.2. Syntax-Fehler von JScript (Auswahl)

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser leider **nicht als fehlerhaft** erkannt werden:

```
Bsp.:   var Kette =   'test';  + '.jpg';      // kein Syntaxerror wegen dem Semikolon vor dem Zeichen +
        alert(Kette);                         // zeigt nur   test    an  und nicht   test.jpg

Bsp:    for (var i = 0; i <3; i++);           // kein Syntaxerror wegen dem Semikolon vor dem Zeichen {
        {alert(i);}                           // genau 1 Anzeige, die 3 anzeigt
```

Es gibt Konstellationen im Quellcode des Skriptes, die vom Browser als irritierend erkannt werden:

Bsp: anstelle von if wird îf (also if mit Hütchen) kodiert --> der Browser verlangt ein **Objekt** und erzeugt keinen üblichen Syntaxerror zur if-Anweisung

Es wird dringend angeraten, einen JavaScript-Debugger zu verwenden, vorallem dann, wenn die Quelltexte gross sind (debuggen = entwanzen von Fehlern).

Debuggersoftware ist in der Regel Bestandteil eines Design-Paketes und für Hobby-Programmierer nicht gerade preiswert. Wichtig: Im Browser müssen Script-Debugging und die Anzeige von Script-Fehlermeldungen genehmigt sein (Nachteil: Bei einer Onlinesitzung wird sich der User bei diesen Einstellungen wundern, wie oft Script-Fehler auch auf (bei abgeschaltetem Scriptdebugging als) professionell wirkenden Seiten auftauchen (z.B. weil nicht browsergerecht programmiert wurde oder über Link eingebundener Scriptcode benötigt wird, der z.B.auf einem anderen, aber gerade nicht erreichbaren Server liegt) und daher eventuell eine Debugging-Anfrage(-Folge) erzeugen.

Einen **kostenlosen** Script-Debugger bietet Microsoft an, der seit Jahren nicht mehr gepflegt wird, vom Design und Umfang abenteuerlich primitiv (steinzeitlich) ist und vermutlich deswegen als kostenlose Software (Gnadenbrot für Hobby-Programmierer) bis in die Ära der hypermodernen, kommerziellen Windows-Net-Software überlebt hat, aber zumindest Script-Fehler mehr oder weniger **erkennt**. Nachfolgend weitere "Vor-" und Nachteile:

- Trotz Zeilenangabe in der Fehlermeldung positioniert der Debugger in der Regel auf die Zeile, die in der Anzahl der Zeilenim Dokument genau in der Mitte liegt und nicht auf die fehlerhafte Zeile.
- Der Debugger ist so alt, dass er die Maus-Rad-Bedienung nicht kennt (Scrollen per Maus nicht möglich).
- Der Debugger kann nicht mit Framesets umgehen: Es wird **immer** nur das Dokument mit dem FRAMESET-Tag angezeigt (natürlich dann auch die, wie oben genannt, falsche Zeilenposition), auch wenn der Fehler in einem Frame auftritt. Frameseiten sind also erst solo auf Fehler zu testen und dann - wagemutig - im Frameset.
- Der Quelltext ist nur dann im Debugger schreibbar, wenn er nicht parallel offen ist. Mit anderen Worten: Wer einen anderen Editor parallel benutzt, hat Pech. Der Debugger besitzt eigene Einstellungen zu Tabs etc..

Der Debugger lässt sich manchmal manuell nicht über den Menüpunkt Ansicht öffnen, wenn der Debugger trotz Fehlermeldung und dortigem geklickten Debuggerstart sich einfach nicht öffnet (aber die Fehlermeldung natürlich verschwindet). Dann ist oft Rätselraten angesagt, wenn der **nun doch** parallel zu benutzende Editor (nur Plain-Text, also z.B. notepad.exe und nicht wordpad.exe) keine Zeilenpositionierung kennt. bzw. diese bei grossen Dateien falsch ausführt (z.B. Editor im Symantec Norton Commander der Windows-Version).

Tritt ein Fehler auf, so **muss** in der Regel der Debugger geöffnet werden (immer dann, wenn der Fehler von der Scriptmaschine nicht übergehbar ist), bevor die Webseite weiter im Browserfenster begutachtet werden kann und das zusätzlich oft nur **nach** dem Schliessen des Debuggers bzw. Quelltextes im Debugger.

Bei Sytanxfehler wegen vergessener } oder) bzw. { oder (ist sehr oft der Debugger nicht in der Lage, die Stelle des Fehlers anzuzeigen. Grund: Die Struktur der Klammerung im gesamten Dokument wird solange geparst, bis eine Klammerung definitiv als fehlend erkannt wird, das aber an einer Stelle, die nicht der Position entsprechen muss, an der die Klammer vom Programmierer wirklich vergessen wurde. Dann ist im wahrsten Sinne des Wortes großes und zeitraubendes Rätselraten angesagt, um die echte Fehlerstelle zu finden. Deswegen: Konzentriert und sauber programmieren (auf Einrückungen bei Verschachtelung unbedingt achten, Kommentare im Quellcode erzeugen, Schnittstellen der Datenübergabe bilden etc.).

Programmierungssoftware, die selbst JavaScript innerhalb der Software für HTML-Design nutzt, ist zum Debuggen nicht zu empfehlen (falls im Design-Paket überhaupt ein Debugger vorhanden ist). Das Problem: Per JavaScript unterstütztes HTML-Design (z.B. visuelle Feinausrichtung eines HTML-Elementes per Maus oder vorgefertigte Menüs) kann JavaScript-Code des Herstellers einbauen. Dieser Code wird natürlich auch debuggt und erschwert die Situation des Programmierers vorallem dann, wenn dieser Code mit programmierer-eigenem Scriptcode nicht über vom Hersteller vordefinierte Schnittstellen (falls diese überhaupt vorhanden und kommentiert sind) erweitert wird und deswegen zusätzlich nicht lauffähig sein kann.

Die Qualität eines Quelltextes entscheidet auch über Fehleranfälligkeit

Beispiele für schlechte Qualität aus eindeutiger Faulheit: Manche Programmierer halten sich mit ihrem Produkt für so perfekt, dass sie nicht an denjenigen denken wollen, der den Quelltext verstehen muss: Die Kreativität im Quelltexterzeugen ist eine Intelligenzfrage und nicht die der Selbstbefriedigung. Letztere Intention ist massenhaft im Internet zu finden, so dass die Quelltextanalyse wohl bedacht sein will (vor allem um dann festzustellen, dass der Programmierer im Wahn seiner Faulheit dein eigenen Quelltext nicht versteht und Fehler eingebaut hat).
Zur Beruhigung: Im professionellen Bereich finden sich ebenfalls Faule, die dann Programmierer, die ihre Quelltexte so strukturieren, dass Dritte sie lesen und verstehen können, als Literaten bezeichnen. Niemand ist unersetzbar, besonders Faule als Angestellte im Job, deren Faulheit im geistigen Eigentum des Arbeitgebers endet, wofür der Arbeitgeber auch noch zahlt. (Sehr beliebt ist das bei C-Programmierern, die ihren Quellcode zum Stacheldraht-Verhauh umwandeln und Logik mit möglichs wenige Quellcode implementieren (und dabei auf Compiler-Intelligenz setzen)

Dass im Quelltext keine Kommentare enthalten sind, sei aus Vereinfachung nicht weiter beachtet.

Bsp. 1

```
<script language="JavaScript">

var NS4 = (document.layers);
var IE4 = (document.all);

var win = window;
var n   = 0;

function findInPage(str) {

 var txt, i, found;

 if (str == "")
  return false;

 if (NS4) {

  if (!win.find(str))
   while(win.find(str, false, true))
    n++;
  else
   n++;

  if (n == 0)
   alert("Nichts gefunden.");
 }

 if (IE4) {
```


```
    txt = win.document.body.createTextRange();


    for (i = 0; i <= n && (found = txt.findText(str)) != false; i++) {
      txt.moveStart("character", 1);
      txt.moveEnd("textedit");
    }

    if (found) {
      txt.moveStart("character", -1);
      txt.findText(str);
      txt.select();
      txt.scrollIntoView();
      n++;
    }

    else {
      if (n > 0) {
        n = 0;
        findInPage(str);
      }

      else
        alert("Nichts gefunden.");
    }
  }

  return false;
}

</script>

<form name="search" onSubmit="return findInPage(this.string.value);">
<font size=3><input name="string" type="text" size=15 onChange="n = 0;"></font>
<input type="submit" value="Suchen">
</form>
```

Jetzt den maschinell aufbereiteten Quellcode, an dem man sehen kann, wie sinnlos o.g. Faulheilt ist.
(Wer allerdings die Aufbereitung manuell machen muss, dem ist nur zu gratulieren :-))

```
<script language="JavaScript">
var NS4=(document.layers);
var IE4=(document.all);
var win=window;
var n=0;

function findInPage(str)
{
 var txt,i,found;
 if(str=="")
 return false;
 if(NS4)
 {
 if(!win.find(str))
 while(win.find(str,false,true))
 n++;
 else
 n++;
 if(n==0)
 alert("Nichts gefunden.");
 }
 if(IE4)
 {
 txt=win.document.body.createTextRange();
 for(i=0;i<=n&&(found=txt.findText(str))!=false;i++)
 {
  txt.moveStart("character",1);
  txt.moveEnd("textedit");
 }
 if(found)
 {
```


```
  txt.moveStart("character",-1);
  txt.findText(str);
  txt.select();
  txt.scrollIntoView();
  n++;
 }
 else
 {
  if(n>0)
  {
   n=0;
   findInPage(str);
  }
  else
  alert("Nichts gefunden.");
 }
}
return false;
}
</script>
<form name="search" onSubmit="return findInPage(this.string.value);">
<font size=3><input name="string" type="text" size=15 onChange="n = 0;"></font>
<input type="submit" value="Suchen">
</form>
```

Bsp. 2:

```
<BODY>
<a href="javascript:function C(v){return '<td>'
+v+'</td><td>'+((v>>4).toString(16)+(v&15).toString(16)).toUpperCase()+'</td><td bgcolor=DDDDDD><b>&'+'#'
+v+';</b></td>';}var c=4,b=Math.ceil(224/c),a='<table border=0><tr>';for(j=0;j<c;j++){
a+='<td>DEC</td><td>HEX</td><td><b>ASC</b></td>';}a+='</tr>';for(i=33;i<33+b;i++){a+='<tr>';
for(j=0;j<c;j++){t=i+(j*b);if(t<=255)a+=C(t);}a+='</tr>';}a+='</table>';var W=open("","width=500,height=600,
left=0,top=0,resizable,scrollbars');W.document.writeln(a);">Klick</a>
<BODY>
```

hier der maschinell aufbereitete Quellcode

```
<BODY>
<a href="javascript: function C(v)
{
 return
 ' <td>'
 +v+'</td><td>'
 +((v>>4).toString(16)+(v&15).toString(16)).toUpperCase()
 +'</td><td bgcolor=DDDDDD><b>&'
 +'#'
 +v
 +';</b></td>';
}
var c=4,b=Math.ceil(224/c),a='<table border=0><tr>';
for(j=0;j<c;j++)
{
 a+='<td>DEC</td><td>HEX</td><td><b>ASC</b></td>';
}
a+='</tr>';
for(i=33;i<33+b;i++)
{
 a+='<tr>';
 for(j=0;j<c;j++)
 {
  t=i+(j*b);if(t<=255)a+=C(t);
 }
 a+='</tr>';
}
a+='</table>';
var W=open("","width=500,height=600,left=0,top=0,resizable,scrollbars');
W.document.writeln(a);">Klick</a>
<BODY>
```

Die Qualität eines Quelltextes entscheidet auch über Schnelligkeit des Parsers:

Je einfacherer strukturiert der Quelltext ist, um so schneller ist der Parser.

z.B. benötigt eine case-Anweisung mehr Parser-Aufwand als eine Folge von optimal verschachtelten if's.

z.B. bewirkt die Zwischenspeicherung eines Zeiger, auf den mehrmals zugegriffen werden soll, eine Vereinfachung für den Parser, da der Zeiger nur 1x berechnet werden muss.

z.B. bewirkt die Nutzung von globalen Variablen in einer Rekursion einen Geschwindigkeitsvorteil, da lokale Variablen nicht pro Rekursion zu erzeugen sind.

Nachfolgend ein Beispiel für den Parser schlecht optimierten Quellcode: Test der Cookiverwaltung im Browser.

```
<script language="JavaScript">

function getCookieVal(offset)
{
 var endstr=document.cookie.indexOf(";",offset);
 if(endstr==-1)
 endstr=document.cookie.length;
 return unescape(document.cookie.substring(offset,endstr));
}

function GetCookie(name)
{
 var arg=name+"=";
 var alen=arg.length;
 var clen=document.cookie.length;
 var i=0;
 while(i<clen)
 {
  var j=i+alen;
  if(document.cookie.substring(i,j)==arg)
  return getCookieVal(j);
  i=document.cookie.indexOf(" ",i)+1;
  if(i==0)break;
 }
 return null;
}

function SetCookie(name,value)
{
 var argv=SetCookie.arguments;
 var argc=SetCookie.arguments.length;
 var expires=(argc>2)?argv[2]:null;
 var path=(argc>3)?argv[3]:null;
 var domain=(argc>4)?argv[4]:null;
 var secure=(argc>5)?argv[5]:false;
 document.cookie=name+"="+escape(value)+
 ((expires==null)?"":("; expires="+expires.toGMTString()))+
 ((path==null)?"":("; path="+path))+
 ((domain==null)?"":("; domain="+domain))+
 ((secure==true)?"; secure":"");
}
var enabled="";
var exp=new Date();
exp.setTime(exp.getTime()+(60*1000));
SetCookie('wannasomecookie',1,exp);
enabled=GetCookie('wannasomecookie');
</script>
<script language="JavaScript">
if(enabled==null)document.write("<b><center>Ihr Browser akzeptiert keine Cookies<br>Your Browser does not accept
Cookies");
else document.write("<b><center>Ihr Browser akzeptiert Cookies<br>Your Browser does accept Cookies");
</script>
```

Fehleranalyse im Internet Explorer

Die Fehleranlayse des IE ist das unterste Niveau, was es im Bereich Programmierung auf Parser-Basis (ohne Compiler) und ohne Hilfesystem per Fremdtool gibt. Die Fehleranalyse des IE liegt im Niveau unterhalb jedes anderen Basic-Sprache-Systems z.B. aus DOS-Zeiten, oder unterhalb diverser Freeware-Parser.

Die Fehleranalyse kann sehr stark eingeschränkt werden, wenn am Quelltextanfang kodiert wurde:

function SymError(){return true;}window.onerror = SymError;

Der Funktionsname ist beliebig.
Genau dann unterbleiben oft Fehlererkennungen.
Wer mit o.g. Funktion das Dokument testet, testet umsonst !
Die Funktion sollte eigentlich nie kodiert werden, es sei denn, man will Scriptfehler verheimlichen, weil das Dokument nicht ausreichend getestet wurde bzw. man mit Scriptfehlern bewusst rechnet.

Der Parser des IE ist ausschliesslich auch Abarbeitung des Dokumentes orientiert und unterstützt Fehlerbehandlung NUR aus dieser Sicht: Fehlermeldungen kommen NUR mit Daten, die der Parser aus seiner Sicht liefert, die aber nicht der des Programmierers entspricht (Beispiele siehe unten bezüglich alert()-Verwendung). Mit anderen Worten: Das Parsen ist Bedingung für Laufzeit des Dokumentes. Fehler, die der Parser anzeigt, verhindern das Laufen des Dokumentes. Und darauf macht der Parser aufmerksam. Nur wegen Ausführung des Dokumentes kommen Fehlermeldungen (Beispiel: siehe unten).

Wer es schafft herauszubekommen, wie der Parser des IE im Einzelnen arbeitet, kann sich viel Zeit ersparen (Parserversionen je nach Version des Windows Script Host also auch des Betriebsystemes und/oder Browserversionen).

Ohne Wissen über den Parser (z.B. aus reichlich Testerfahrung) gilt:

Wer mit dem IE ohne Fremdtools programmieren will, muss sich auf reichlich Ärger und Minimalsupport durch den IE einstellen, oder gleich die Finger davon lassen.

Dann hilt nur entweder Fremdtools zu nutzen oder manuell den gesamten Quelltext der JS-Datei nach Fehler zu durchforsten (Methoden siehe unten).
(Es können aus selbstprogrammierte Tools das Manko ersetzen.)

Grössere JS-Dateien oder Abschnitte des Dokumentes müssen daher schon in Teilabschnitten der Kodierung getestet werden (Testphase parallel zur Kodierung im Try-And-Error-Stil).

Beispiel: Dieses Beispiel verusacht einen Fehler, der eine fehlende ')' anmahnt
Zeile 5 Zeichen 14 also an der Stelle wo '!' steht.

```
          12345678901234567890123456789
----------------------------------------
1         <html>
2         <head>
3         <SCRIPT LANGUAGE='JScript'>
4         <!--
5         var X10=(X04 ! null);
6         //-->
7         </SCRIPT>
8         </head>
9         </html>
```

'!' ist der NOT-Operator, der eine nachfolgende Referenz erwartet;
im Code wäre das also die Referenz null.

Da null ein Zeigerwert ist (Bsp. var x=new Array(); alert(X!=null); liefert true)
kann er nicht negiert werden.
Also ist ! falsch.

Weil der Operator falsch kodiert wurde, muss der Parser den vorhergehenden gültigen Code für komplett gültig machen, indem die Klammer vorgeschlagen wird, was aber aus Programmierersicht völliger Unsinn ist, weil nur '=' vergessen wurde zu kodieren.

Der Parser will also
var X10=(X04)
was syntaktisch korrekt ist.

Kodiert man die Klammer, dann ergibt sich:

```
var X10=(X04) ! null;
oder  var X10=(X04) null;
```

Ergo nächster Fehler: Der Parser erwartet ein ';'

... und so weiter

Der Parser prüft also genau bis zur Fehlerstelle 14 in Zeile 5,
aber prüft nicht, welche Kombinationen es mit dem als
falsch-kodiert-erkannten Operator '!' gibt
z.B. !=
und bemeckert daher nicht ein '=' oder anderes Relationszeichen.
(Der Hinweis "Relationszeichen" neben dem Hinweis auf ')'
würde schon ausreichen).

Mit anderen Worten: Die Kontextanalyse des IE ist minimalistisch und
häufig wertlos. Wieso Microsoft den Parser so
schlecht als Schnittstelle ausstattet, ist völlig unklar.

Jedes Entwicklungssystem, das NUR auf Informationen des Parsers zurückgreift, kann nur so gut sein wie der Parser.

Fremdtools können im Regelfall nicht so optimiert sein wie Tools des Browserherstellers, es sei denn,
der Browserhersteller legt Internas komplett offen, was sehr stark zu bezweifeln ist.
implementieren oft eigenen Scriptcode als Feature des Tools.

Analyse-Methode:

Es muss unbedingt auf Kommentierung des Quelltextes geachtet werden, da Kommentare
ein Hilfe darstellen können.

Der Browser parst schrittweise die Bereiche zwischen <script> und </script>
in der Reiehenfolge der kodierten <script> und </script>.

Eine Fehlermeldung bezieht sich also immer auf einen Bereich zwischen <script> und </script>.

Die Frage ist nur, welchen Bereich zwischen <script> und </script>, wenn mehrere
<script> und </script> also Bereiche vorliegen.

Um festzustellen WELCHE Fehlermeldungen bis zum o.g. </script> generiert werden,
muss ein <script>alert();<script>
hinter obigem </script> kodiert werden.

Es sind dann ALLE Scriptfehler VOR der alert-Meldung zu beheben,
auch wenn nach der alert()-Meldung weitere Scriptfehler angezeigt werden,
die aber aus anderen <script> und </script> stammen können.

Sollte ein alert() am Anfang des Quelltextes nicht erkannt werden, so ist im gesamten
Quelltext ein Scriptfehler:

Der Parser prüft z.B. Paarigkeit von Klammern: Fehlt z.B. genau 1 Klammer
im Quelltext um Funktionen, dann ordnet der Parser die nächste
gefundene Klammer zu, bis das Ende des Quelltextes erreicht wird. Und
genau an dierser Stelle tritt der Scriptfehler auf mit der Nummer der
letzten Zeile im Quelltext, obwohl die fehlende Klammer irgendwo im
Qelltext fehlt. Dann muss der gesamte Quelltext manuell geprüft werden
auf Syntaxfehler.

Wenn der Parser einen Scriptfehler wegen fehlender Klammer inmitten des
Quelltextes anzeigt und auffordert, dass eine Klammer an die Position
laut Fehlermeldung zu setzen ist, dann kann nach Setzung der Klammer
ein erneuter Scriptfehler auftreten: Dann verlangt der Parser zwar
eine Klammer, aber nicht an der Stelle. Also muss die Klammer
woanders fehlen, oder die Klammer ist wegen anderem Syntaxfehler,
der erst später erkannt wird, garnicht notwendig. Wie auch immer:
Dann muss der gesamte Quelltext manuell geprüft werden
auf Syntaxfehler.

Beispiel: Dieses Beispiel verusacht einen Fehler, der eine fehlende '}' anmahnt
Zeile 18 Zeichen 1 also vor //-->

```
          12345678901234567890123456789
----------------------------------------
1         <html>
2         <head>
3         <SCRIPT LANGUAGE='JScript'>
4         <!--
```


```
5       function z()
6       {
7               if (X10)
8               {
9                       if (X08==0)
10                      {
11                              X10=(X01 != null);
12                              if (X10){X11=X01.length;}
13                              if (X10){X10=(X02!=null);
14                              if (X10){X12=X02.length;}
15                      }
16              }
17      }
18      //-->
19      </SCRIPT>
20      </head>
21      </html>
```

Der Fehler liegt in Zeile 13 am Zeilenende: Dort fehlt '}'.

Der Parser ordnet also anstelle der fehlenden Klammer nichts zu, sondern
verändert die Programmlogik:

```
function z()
{
        if (X10)
        {
                if (X08==0)
                {
                        X10=(X01 != null);
                        if (X10){X11=X01.length;}
                        if (X10){
                                        X10=(X02!=null);
                                        if (X10){X12=X02.length;}
                                }
                }
        }
```

fehlende Klammer laut Parser: Am Ende der Funktion, das zufälligerweise
vor dem //--> steht

Der obige fehlerhafte Code inmittem eines Quelltextes mit diversen
Funktionen und Verschachtelungen verursacht Stress pur.

Besonders stressig wird folgender Fall:

alle Inklude sind eigenständig syntaktisch korrekt
im Dokument das inkludet wird ein '}'-Klammer-Fehler entdeckt
jedoch ein alert hinter dem letzten Inklude wird erkannt
aber nicht das alert hinter <script> das direkt hinter dem letzten Inklude
liegt (nach Tagwechsel) und VOR der ersten Klammer '{'
im Dokumentenquelltext

Damit können doch die Inklude nicht syntaktisch korrekt sein !

Da Inklude als Quelltext eingebunden werden wird eine Klammeranalye
auch über die Include vollzogen, also die Programmierlogik
über eventuell alle Include verändert, obowhl die Inkludes
einzeln korrekt sind.

Nur in welcher Inklude liegt der Fehler ? Also ab welcher Inklude
wird die Programmlogik verändert ?

Also alle Inklude als Gesamtbestand sytaktisch testen:
Quelltexte aller Include in neues Dokument kopieren und das testen.

Wird dann ein Fehler angezeigt, der im Dokument auf ein HTML-Tag
weist, das nichts mit Script zu tun hat, z.B. auf <head>,
aber dort ist kein Fehler, dann ist klar, den Parser
kann man endgültig vergessen.
Stunde der Wahrheit: Elan ohne Parser weiter zu machen oder
sich andere (kostenpflichtige) Software

- ev. inklusive anderen Browser-
beschaffen.
Eines steht fest: Viel Spass bei der Try-And-Error-Programmierung :-)

Wird beim Rendern Script-Quelltext im Browserfenster angezeigt, dann wurde ein <script>-Tag
oder </script>-Tag nicht korrekt erkannt.

Der im Browser angezeigte Quelltext könnte zur Suche im Script helfen, muss
aber nicht: Die Anzeige muss nicht Text anzeigen, den es so - wie angezeigt -
im Script tatsächlich auch gibt.
Wegen Fehler könnten Scriptbereiche virtuell in der Anzeige zusammengeschoben sein.
Mann muss dann nach eindeutigen Stellen suchen, die identisch sind in der
Anzeige und im Script.

Achtung: Kommentarzeichen // werden ebenfalls so behandlet wie bei Klammeranalyse

Beispiel: // text document.write('<SCRIPT LANGUAGE="JScript" SRC="...."></SCRIPT>');

Diese Zeile wird als fehlerhat erkannt, wenn das // nicht korrekt erkannt
wurde und dadurch document.write(....) nicht hinpasst aus Sicht
des Parsers.

Entfernt man diese Zeile und es wird kein Script mehr in der Anzeige gerendet,
MUSS ein //-Fehler vorliegen ! ... Nur wo ??????

Hier hilft alert(); , denn wird das ausgeführt, dann ist
Script auch geparst worden und gültig.

alles Script, was nach alert(); im Browserfenster als Text angezeigt wird,
ist falsch.

Man beachte: alert() kann wegen //- oder Klammerfehler-Programmlogik-
Verschiebeung
aktiv werden !

Ist die Fehlerstelle aber korrekt, so liegt eine Programmlogik-Verschiebung vor.
Also muss der Script vor dem alert() untersucht werden per
stückchenweiser Quelltextentfernung. Und zwar solange, bis
Script nicht mehr als Text NACH alert(); angezeigt wird.
Dann hat man die Fehlerstelle, die VOR alert() liegt.

Folgendes Beispiel rendet Script als Text mit folgender Anzeige '); //-->

```
1   <html>
2   <head>
3   <SCRIPT LANGUAGE='JScript'>
4   <!--
5   // Das ist ein Text --> siehe nachfolgendes alert();
6   alert();
7   // Das ist auch ein Text: So bindet man Script dynamisch ein:
document.write('<SCRIPT
    //  LANGUAGE="JScript" SRC="...."></SCRIPT>');
8   //-->
9   </SCRIPT>
10  </head>
11  </html>
```

Der Parser macht folgendes draus:

'); //--> deutet auf Scriptfeher VOR dem ' hin aus Zeile 7
Das letzte > in Zeile 7 wird erkannt, da nicht angezeigt.

Aber mal Zeile 5 ansehen !! dort befindet sich ebenfalls ein '>' und könnte
einen TAG markieren
Wird dieses '>' in Zeile 5 entfernt, ist der Fehler weg !!!
Werden in Zeile 5 direkt VOR dem '>' die beiden '-' entfern, ist
der Fehler auch weg.
'-->' wird also vom Parser erkannt.
alert in Zeile 6 wird immer ausgeführt, also ist Zeile 6 okay.
kann nur ausgeführt werden, wenn Zeile
3

oaky ist.
Also liegt der Fehlern in den Zeilen 4 bis 5.
wegen '--> in Zeile 5 wird '>' am Ende der Zeile 7 erkannt
wobei //--> aus Zeile 8 nicht als
Tagbegrenzer erkannt wird.
Wer mag, kann austesten wie der Browser reagiert, nimmt man anstelle Zeile
5 nun in Zeile 7 ein '>' weg.

Als letztes Mittel hilft, Quelltextteile stückweise solange zu entfernen, bis kein Scriptfehler mehr auftritt, so dass der Fehler dann im entfernten Quelltext liegen muss.

Diesen entfernten quelltext kopiert man dan in eine leeres Dokument und testet mit diesem wie oben beschrieben.

Fehler, die zur Laufzeit erzeugt werden z.B. Variable nicht definiert, werden dann Stress, wenn z.B. mehrere Inklude vorhanden sind, die als Bezeichner für lokale Variablen (innerhalb einer Funktion) identische Bezeichner verwenden.
Dann ist unklar, welche Inklude, da die Fehlermeldung auf das Dokument bezogen sein kann, auch wenn Fehler in einem Inklude liegt. Als Orientierung kann die Renderung des Dokumentes verwendet werden: Der Programmierer weiss, was welche Include bewirkt.
Desweiteren müssen Programmaufrufe vom/aus dem Inklude geprüft werden.

Typisches Beispiel:
Parameter einer Funktion sind in der Argumentenliste lokal referenziert.
Daher können alle Argumente mit dem Bezeichner über alle Funktionen identisch sein.

Zulässig ist z.B.

```
function y(X00,X01,X02)
{...}

function z(X00,X01,X02)
{...}
```

Scriptfehler können einen Absturz des Browsers verursachen, ohne dass der Absturz sichtbar wird:
Es ist möglich, dass das DOM zerschossen wird und nur ein Neustart des Browsers hilft.

Grund: Der Übergang zwischen Parsen und Ausführen des Scriptes ist nicht sichtbar.
Wenn ein Script zwar syntaktisch korrekt, aber z.B. wegen Zeigerfehler das DOM zerschiesst, so kann der Parser nicht mehr korrekt arbeiten, was auf die Fehleranalyse Auswirkungen hat (z.B. unter unbemerkt zerschossenenem DOM den Quelltext ändern und dann neu parsen lassen).

Meist ist dann vergebliche z.T. erhebliche Scripttestzeit vergangen, eh man den Absturz bemerkt.
Lieber einmal zu viel als zu wenig den Browser neu starten !

JS-Dateien:

wenn JS-Dateien inkludet werden, so diese vorher einzeln testen, ob fehlerfrei (in einem Dokument, das NUR das include enthält auf Scriptfehler testen).

erst wenn JS-Dateien syntaktisch sauber, sind die Fehlermeldungen auf das Dokument bezogen, das die JS-Dateien inkludet. Ausnahme: Fehler einer JS-Datei z.B. innerhalb einer Funktion der ERST mit Ausführung der Funktion erkannt wird, also MIT Laufzeit des Deokumentes, das die JS-Datei includet.

Die Fehleranalyse wird im Dokument mit JS-Dateien immer ohne Referenz der Fehler auf die JS-Quelldatei vollzogen und dafür immer bezüglich Dokument. Zeilennummern des Fehlers können sich aber auf JS-Dateien beziehen, ohne Hinweis, dass der Fehler in der JS-Datei liegt und schon garnicht in welcher JS-Datei, obowhl der Parser das Inkluden der JS-Dateien ausführt, also Informationen zur JS-Datei vorliegen könnten.) Mit anderen Worten: Der IE ist ohne Fremdtools bezüglich modularer Programmierung so gut wie garnicht effektiv nutzbar, dafür ideal für Try and Error-Programmierung als Rätselraten im Go-To-Stil.

Endlosschleifen können zum Browserabsturz führen:

Schleifen wie for sind schnell und CPU-nah implementiert (analog zu Assembler-Implementation)

also damit Ressourcen allokierend:
Sollte das Timing des Browsers und Windows gefährdert sein, so bricht der Browser im Regelfall den rechenintensiven Task ab, aber eben nicht immer.

Endlosschleifen sind unbedingt zu vermeiden !
Schleifen sind im Umfang zu minimieren:
Lieber weniger verschachtelte for-schleifen, dafür andere Programmlogik.
Die Laufzeit von Anweisungen innerhalb einer Schleife ist zu minimieren:
Z.B. keine Rekursion innerhalb einer Schleife.

Rekursionen mit oder ohne Timer (window.setTimeout(), window.setInterval()):

Timer-Routinen belasten die Timer von Windows, das nur begrenzte Ressourcen hat.
Z.B. kann die Uhr von Windows beeinflusst werden.
Timer-Routinen mit Schleifen wie for sind daher besonders laufzeitempfindlich !

Lokale Variablen werden mit jedem Betreten der rekursiven Funktion neu erzeugt.
Um Laufzeit zu verkürzen und Ressourcen zu sparen, sollte man lokale Variablen durch globale ersetzen. Die Rekursion nutzt dann bereits vorhandene und sich nicht änderende Variablen, also Zeiger und ist damit schneller.
Globale Variablen werden mit Parsen des Quelltextes generiert, also vor dem Parsen aller Funktionen.

Beispiel:

```
var a=1000;
var b=1;    // Zähler
var c=0;    // Timeout-ID
var d=100;// Wartezeit in Millisekunden

function x()
{
        b++;
        a-=b;       // identisch mit a=a-b;

        c=window.setTimeout('x()',d);    // erneuter Aufruf von x nach Wartezeit
                                                    // also 'x()' wie per eval('x();') ausführen
}
```

Analog sollte bei Rekursion anstelle return ebenfalls eine globale Variable verwenden.

Zur Eineindeutigkeit von globalen Variablen einer Funktion empfiehlt es sich, in den Namen der Variablen den Namen der Funlktion einzubauen.
Da die Funktion eineindeutig, so sind es die Variablen auch.
Beispiel:

```
var IchTeste_Zaehler=0;
function IchTeste()
{IchTeste_Zaeher++;}
```

Zur Vereinfachung sollten Funktionen mit Beginn des Namens mit einer Abkürzung versehen werden, der auf Funktion hinweist. Analog dazu bei Variablen.
Beispiel:

```
var X_IchTeste_Zaehler=0;
function Y_IchTeste()
{X_IchTeste_Zaeher++;}
```

Stack Overflow:

Diese Fehlermeldung weist darauf hin, dass sich eine Funktion selbst aufruft OHNE window.setTimeout() oder window.SetInterval().

Da beim Aufruf verfügbare die Argumente geparst werden und diese pro Aufruf refrenziert werden müssen, legt die Windows-Scriptmaschine diese auf dem Programm-Stack ab (Speicherbereich analog beim Programm aus Compilation).

Ist der Stack voll, kommt diese Meldung.

setTimeout(x,y)

x kann Zeiger (z.B. Feldelement in dem ein Funktionszeiger liegt) sein, der wie folgt kodiert werden muss:

'x()'

setTimeout ruft also eval auf

Daher kann auch eine dynamisch erzeugte Funktion verwendet werden.

Returnwert bei Rekursion:

```
var X=-1;
function test()
{
	X++;
	if (X < 5){window.setTimeout('test()',1000);}
	else{return X;}
}

var Z=test();		// es wird nicht gewartet, bis die Rekursion beendet ist, sondern sofort Z belegt
			//	Die Rekursion läuft parallel weiter und erzeugt nach 5 Sekunden den
			//	Returnwert,
			//	also NACH  Z=test();
			// Da nicht gewartet wird, liefert Z einen null-Zeiger
			//	alert(test() == null);  liefert true

			// Damit gilt: Z=test(); kann für Rekursionen NICHT verwendet werden
			//	Grund: Die Funktion endet korrekt und ruft sich nach 1 Sekunde wieder
auf
			//	(neue Instanz)
			//		also wäre das 5x der Funktionsaufruf
			//		wobei NUR der letzte return ausführt.

			// Lösung: test () muss eine neue Funktion aktivieren, die den gesamten
			// Programmablauf hinter dem
			//		Aufruf der Rekursion beinhaltet. Die alte Funktion muss also
			//		direkt nach dem Start der
			//		Rekursion enden !

alert(test() == null);
```

Timoutwert der Rekursion:

```
window.setTimeout('test()',TimeoutWert);
```

TimeoutWert muss mindestens so hoch sein wie die Laufzeit der Funktion:
Funktion muss enden können OHNE dass die vor Ende erneut aktiviert wird.

Grund: Paralleles Aktivieren bedeutet Dateninkonsitenz für alle globale
Variablen, die von der Funktion beschrieben werden zum Zweck
des Lesens im nächsten Aufruf der Funktion. Dann dürfen
globale Daten nicht parallel verwaltet werden, also z.B.
kein Lesen im nächsten Aufruf wenn Daten noch garnicht geschrieben
sind im aktuellen Aufruf der Funktion.

Der TimeoutWert ist in Millisekunden, der von Windows unabhängig von der CPU-Geschwindigkeit
verwendet wird.

Verwendung von eval():

Bei Verwendung von eval() sind Scriptfehler z.T. erst zur Laufzeit ermittelbar.
Grund: Das Parsen von Script während der Laufzeit.

return (eval-ausdruck mit return-Anweisung);

Es wird erst innere return-Anweisung von eval abgearbeitet
und dann vom äusserem return entgegengenommen

Bsp.: eval("return window.event.returnValue");

Ist (z.B. in einem Eventhandler) eine äussere return-Anweisung in der Funktion
zwar notwendig zu kodieren,
aber nicht kodiert WORDEN,
dann kann eine Fehlermeldung, dass ein return ausserhalb der Funktion liegt

Argumentenliste einer Funktion:

Die Argumentenliste einer Funktion wird ERST und mit JEDEM Ausführen der Funktion geprüft

Beispiel:

```
function x(X00,X01)
{
        X00=1;
        X01=2;
        X03=3;
}
```

verursacht Fehlermeldung, dass X03 nicht verfügbar ist

Werden Argumente nicht mit Wert belegt, so werden die Variablen der Argumente nicht erzeugt, sind also null

Argumente sind null-Zeiger wenn kein Wert mit Funktionsaufruf übergeben wird.

Beispiel:

```
function test(x)
  {alert(x==null);}

  test();          // liefert true
  test(1);         // liefer false
```

Es müssen also bei korrekter Programmierung die Argumente auf geprüft werden
ERST auf != null
DANN auf Wertbereich.

Achtung: Der Datentyp des Argumentes wird mit dem Wert, dem das Argument erhält, festgelegt
Das Argument ist ein Zeiger, der auf einen Speicherbereich mit einem Wert,
der Daten nach JScript-zulässigen Typen hat.

Da mit JEDEM Aufruf der Funktion die Argumentenliste geprüft wird, kostet das Laufzeit.
Gerade bei Rekursionen per Funktion sollte man daher
anstelle der Argumentenliste
globale Variablen verwenden, denn die wurden bereits geparst und als
Zeiger hinterlegt.

Beispiel:

```
var a=1000;
var b=1;    // Zähler
var c=0;    // Timeout-ID
var d=100;// Wartezeit in Millisekunden

function x()
{
        b++;
        a-=b;       // identisch mit a=a-b;

        c=window.setTimeout('x()',d);    // erneuter Aufruf von x nach Wartezeit
                                         // also 'x()' wie per eval('x();') ausführen
}
```

Analog sollte bei Rekursion anstelle return ebenfalls eine globale Variable verwenden.

Returnwert einer Funktion:

Eine Funktion kann immer ein return kodiert haben, da der Returnwert vom
Aufrufer der Funktion ausgewertet werden kann, aber nicht muss
(Auswertung per Ausdruck mit Referenz auf Funktion z.B. Y=test();)
Wird ein returnwert geliefert UND wird dieser auch ausgewertet, so muss
der Aufrufer der Funktion, die return liefert, warten, bis der Returnwert geliefert wurde: Dass ein Returnwert geliefert werden soll, weiss der Aufrufer VOR Aufruf der Funktion nur deshalb, weil die Auswertung kodiert wurde. Ob die aufgerufen Funktion wirklich einen Returnwert liefert, steht in den Sternen, da JScript auf Parserbasis arbeitet, also der Funktionswert nur dann erzeugt wird, wenn return vorgefunden wird (JScript kennt keine PROCEDURE und FUNCTION wie z.B. bei Borland Pascal, also keine Unterscheidung zischen Funktionen, die Funktionswert nicht liefern dürfen bzw. immer liefern müssen).

Will man, dass der Code nach einer Funktion nicht parallel zur Funktion
abgearbeitet wird, dann gilt:
Funktion darf keine Rekursion (mit/ohne Timer) sein
Funktion muss return kodiert haben
Aufruf der Funktion mit Referenz auf Funktion und damit auf den Returnwert.

Function und Stack Overflow:

Diese Fehlermeldung weist darauf hin, dass sich eine Funktion selbst aufruft
OHNE window.setTimeout() oder window.SetInterval().

Da beim Aufruf verfügbare die Argumente geparst werden und diese pro Aufruf
refrenziert werden müssen, legt die Windows-Scriptmaschine diese
auf dem Programm-Stack ab (Speicherbereich analog beim Programm aus Compilation).

Ist der Stack voll, kommt diese Meldung.

Eine Eigenschaft eines Objektes kann per null-Zeiger-test abgefragt werden ohne Scriptfehler

Bsp.: <INPUT ID='ID_INPUT'> ohne TYPE-Attribut hat keine Eigenschaft tagName

```
if (ID_INPUT.style.test != null) .....
```

Achtung: Es ist kann automatisches Prototyping aktiv sein

```
Bsp:    ID_INPUT.style.test='33'
        alert(ID_INPUT.style.test);   liefert 33
```

```
Wenn    <SCRIPT ...>
        <!--
  kodiert wurde, dann sollte auch
        //-->
        </SCRIPT>
  kodiert werden
```

Fehlernummer	Info
1002	allgemeiner Syntax Fehler
1003	':' erwartet
1004	';' erwartet
1005	'(' erwartet
1006	')' erwartet
1007	']' erwartet
1008	'{' erwartet
1009	'}' erwartet 1012 '/' erwartet
1010	Bezeichner erwartet
1011	'=' erwartet
1014	falsches Zeichen
1015	Stringkonstante falsch
1016	Kommentar hier nicht zulässig
1018	'return' Anweisung liegt außerhalb einer Funktion
1019	break Anweisung außerhalb der Schleife unzulässig
1020	continue Anweisung außerhalb der Schliefe ist unzulässig
1023	Hexadezimal-Ziffer erwartet
1024	'while' erwartet
1025	ein und dasselbe Label (Sprungmarke) wurde mehrmals definiert
1026	Label (Sprungmarke) nicht gefunden
1027	'default' kann nur genau 1 mal in der switch Anweisung kodiert werden
1028	Bezeichner oder String oder Number erwartet
1029	'@end' erwartet
1030	bedingtes Parsen ist nicht aktiv
1031	Konstante erwartet
1032	'@' erwartet
1033	'catch' erwartet
1035	throw Anweisung muss hinter dem Ausdruck auf gemeinsamer Quellcode-Zeile folgen

3.10.3. Abfangen von Runtime Fehlern (Laufzeitfehlern)

siehe auch error JScript-Objekt des Internet Explorer für private Run-Time-Error

try catch finally

Fehlerbehandlung in Script (nicht Ereignisse von Objekten !)
Verschachtelung möglich

Fehler: Runtime Error
oder per throw Anweisung erzeugter Error

Syntax:

```
try                 {tryStatements}
[catch(exception) {catchStatements}]
[finally            {finallyStatements}]
```

tryStatements	können Fehler erzeugen, der abgefangen werden soll
exception	freier Bezeichner als Platzhalter für den Fehler enthält, der aus der Abarbeitung von tryStatements resultiert Variable wird automatisch gefüllt
catchStatements	exception auslesen und daraufhin eine Reaktion ausführen werden nur abgearbeitet, wenn Fehler auch wirklich auftrat in der Abarbeitung von tryStatements
finallyStatements	wird immer abgearbeitet, egal ob ein Fehler in der Abarbeitung von tryStatements und / oder catchStatements auftrat oder nicht

Beispiel:

```
function Anzeige(Kette)
{  alert(Kette); }

try {Anzeige("try1");
{
        try       { Anzeige("try 2");}
        catch(e) { Anzeige("catch2 " + e); }
        finally    { Anzeige("finally2"); }
}
catch(e) { Anzeige("catch1 " + e); }
finally   { Anzeige("finally1");}
```

throw

freie Fehlerbedingung erzeugen für try...catch…finally
throw im try kodieren als freie Fehlerbedingung
catch auswerten

Syntax:

```
throw exception;
```

exception Wert oder Variable

Beispiel:

```
function Anzeige(Kette)
{  alert(Kette); }

try {Anzeige("try1");
{
        try
        {
                throw "Das ist eine Fehlerbedingung";
                Anzeige("try 2");
        }
        catch(e)
        {
                if ( e == "Das ist eine Fehlerbedingung" )
                {Anzeige("catch2 " + e); }
        }
        finally { Anzeige("finally2"); }
}
catch(e)  { Anzeige("catch1 " + e); }
finally    { Anzeige("finally1");}
```

onerror

Ereignis ausgelöst, wenn während des Ladens eines HTML-Dokumentes ein Syntaxfehler auftritt
eines Bildes ein Fehler auftritt
der Abarbeitung von Scripten ein Runtime-Error auftritt

sämtliche Fehlermeldungen unterbinden per

```
var RetteOnErrorHandler = window.onerror;
```


```
window.onerror = null;
```

Eventhandler muss wie folgt kodiert werden:

```
function freier_name_fuer_onerror_behandlung
(   error_erklaerung_string,
    url_des_html_dokumentes_als_string,
    zeilen_nr_des_errors_im_html_dokument
)
{
        ......
        return true;        //  nur wenn true geliefert, dann
                                //      wird die Browsereigenen
                                //      onerror-Behandlung
                            //      unterdrückt
}
```

pro Fehler ein Aufruf des Eventhandlers
--> Folge von Fehlern, also Folge von Aufrufen

Die Script-Debugger-Aufforderung ist nur in den Eigenschaften des Browsers abschaltbar

Beispiel 1:

```
<SCRIPT ...>
<!--
    function eigenes_fehler_fenster(meldungs_text,url,zeilen_nr)
    {
        // Fehlermeldung bilden
        var url_als_kette=url.toString();
        var zeilen_nr_als_kette=zeilen_nr.toString();
        var meldung_komplett=medlungs_text + url_als_kette + zeilen_nr_als_kette;

        // Meldungsfenster
        var fenster=window.open();
        with (fenster.document)
        {
            open("text/html");   // HTML-Dokument im Fenster erzeugen

            writeln("<HTML><HEAD><TITLE>Private Errormeldung</TITLE></HEAD>");
            writeln("<BODY><H1>Fehlermeldung</H1>");

            writeln(meldung_komplett);

                writeln("<FORM><INPUT TYPE=BUTTON VALUE='OK'
              onclick='self.close()'>"
            );          // ist EINE Zeile

            // Button anklicken, damit das Meldungs-Fenster geschlossen wird
            close();  //HTML-Dokument schließen
        }
        return true;        // muss true liefern für window.onerrer
    }

    var RetteOnError=window.onerror;

    window.onerror= eigenes_fehler_fenster;     // NICHT onError kodieren !
                                                // keine () kodieren, da sonst die Funktion
                                                //      sofort ausgeführt wird !!
                                                // window.onerror verlangt die Zuweisung
                                                //      von true für Abschaltung des
                                                //  Standard-onerror
// -->
</SCRIPT>
...
<BODY>
...
</BODY>
```

Beispiel 2:

```
<SCRIPT>
    function FehlerAufspueren (MeldungString, UrlString, ZielenNummerString)
    {
```


```
                    ID_Div.innerHTML  ="<B>Fehler erkannt</B>";
                    ID_Div.innerHTML+="Error: "  + MeldungString         + "<BR>";
                    ID_Div.innerHTML+="Line: "   + ZielenNummerString + "<BR>";
                    ID_Div.innerHTML+="URL: "    + UrlString                + "<BR>";

                    return false;
                }

                window.onerror=FehlerAufspueren;          // ohne () kodieren
        </SCRIPT>
        <BODY>
        <DIV ID="ID_Div"></DIV>
        <BODY>
```

Beispiel 3 für Bild:

```
        <SCRIPT>
                var Kette = '<IMG STYLE="display: none;" ID="ID_IMG" ALT="Das ist ein Bild ">';

                function Laden()
                {
                        ID_Div.innerHTML=Kette;                 // wird sofort geparst, also IMG-Tag ausgeführt
                                                                //          ( anstelle von eval()
                                                                //            bzw. document.write() )
                        ID_IMG.src="";
                        ID_IMG.style.display="block";

                        ID_IMG.onerror= IMGAltTextAufFehlerMeldung;     // ohne  ()

                }

                function IMGAltTextAufFehlerMeldung ()
                {
                        ID_IMG.alt="Das Bild konnte nicht geladen werden.";
                        return true;
                }
        </SCRIPT>
        <INPUT TYPE=button VALUE="Klicke zum Bildladen" onclick="Laden()">
        <DIV ID="ID_Div"></DIV>
```

4. Objekte in Javascript und im Browser

Ansatz und Begriffe zur objektorientierten Programmierung eines HTML-Dokumentes mit Javascript

HTML-Dokument und seine HTML-Elemente

Das HTML-Dokument setzt sich aus HTML-Elementen (Tags) zusammen. Um diese mit dem Dokument wirksam werden lassen zu können (z.B. per Anzeige im Browserfenster), muss der Browser wissen, **ob und wie** er ein HTML-Element wirksam werden lassen kann bzw. darf oder muss, also **welche** Möglichkeit das Element zulässt.

Browser und Scriptmaschine

Der Browser muss sich dabei auf die Scriptmaschine berufen, die dem Browser genau diese Möglichkeit zum HTML-Element vorschreibt, also **vordefiniert**. Die Scriptmaschine parst das HTML-Element im HTML-Quellcode und teilt dem Browser genau diejenigen Anweisungen mit, die das HTML-Element wirksam werden lassen. Die Scriptmaschine kann HTML-Code und z.B. auch Scriptcode parsen. HTML und Javascript sind Scriptsprachen.

HTML-Dokument, HTML-Elemente und Scriptmaschine

Die Scriptmaschine hat einen bestimmten Umfang an Wirkungen für HTML-Elemente. Der Umfang zu jedem HTML-Element ist vordefiniert und damit auch die Reaktion des Browsers auf ein HTML-Element, das z.B. im Browserfenster per Anzeige wirksam werden soll. Diese gilt für alle HTML-Elemente des Dokumentes, das mit seinen Elementen sozusagen lebendig wird.

aktives HTML-Dokument und aktive HTML-Elemente

Sind HTML-Elemente wirksam, so spricht man von der **Laufzeit** dieser **Elemente** und damit von der **Laufzeit des Dokumentes**, das dann auch als **aktives Dokument** mit **aktiven Elementen** bezeichnet wird.

HTML-Element und seine Daten als Komponente des Elementes

Jedes HTML-Element hat seine Daten. Soll die Wirkung eines Elementes z.B. die **Anzeige von Text** sein, dann ist dieser Text eine einzelne Date des Elementes. Daten sind also Teil des HTML-Elementes, also dessen Komponenten.

Datenspeicherung und Zeiger

Daten werden über das Betriebssystem im Hauptspeicher (RAM) des PC verwaltet. Der Ort im RAM, also dort, wo die Daten liegen, wird über eine Speicheradresse erreichbar. Eine Adresse im Speicher kann auch als **Zeiger** auf eine Speicherstelle bezeichnet werden. Ein Zeiger ist also der Begriff für eine Datenadresse im Speicher, z.B. der Adresse des Textes eines HTML-Elementes.

HTML-Element und Zeiger auf Daten

Daten zu einem HTML-Element sind nur dann ansprechbar, wenn das HTML-Element **aktiv** ist. Es wäre daher für den Programmierer sehr praktisch, wenn das Ansprechen des **aktiven** HTML-Elementes gleichzeitig die Daten im Speicher **mit** ansprechen würde. Es gibt noch andere wichtige Gründe:

Das Betriebssystem verwaltet Daten der Instanz automatisch und ohne Hinzutun des Programmierers.
Diese Verwaltung betrifft die **physische** Ablage der Daten im Speicher und deren physische Manipulation.
Der Programmierer muss sich nicht um die physische Speicheradressierung kümmern, sondern verwaltet den Speicher **logisch,** also nur im Zusammenhang mit der Programmierung.
Die Scriptmaschine nutzt die Schnittstelle zum Betriebssystem (bzw. ist z.T. eine Komponente des Betriebssystems) und die Schnittstelle zum Programmierer, also Javascript.

HTML-Element und andere Komponenten

Bisher wurden nur die Daten zum HTML-Element, z.B. der Text, betrachtet. Aber bekanntlicherweise hat ein Text auch ein Aussehen (Layout) wie fett oder unterstrichen. Das Layout ist also eine Komponente des HTML-Elementes.

Für den Fall, dass z.B. Einfügen oder Ausschneiden von Text möglich sein sollen, muss das HTML-Element eine Komponente bekommen, die solche Aktionen realisieren.

Typische Komponenten zu einem HTML-Element sind also die Komponenten Daten (z.B. Text), Eigenschaft (z.B. Layout) und Aktion (z.B. Text ausschneiden). Diese Komponenten werden in der Objektsicht auf das HTML-Element erneut auftauchen, dort aber z.T. anders bezeichnet.

HTML-Element als Instanz (Komponentensammlung)

Wie man sieht, reicht damit ein Zeiger nur auf Daten nicht aus. Das aktive HTML-Element muss erweiterbar sein, also zur **Laufzeit** des HTML-Elementes das Layout und die Textveränderung mitgeteilt bekommen können. Ergo sind Layout und Textveränderung **ebenfalls** im Speicher abzulegen. Es muss also 3 Zeiger geben: 1 Zeiger auf Daten, 1 Zeiger auf Layout und 1 Zeiger auf die Textveränderung als Aktion des aktiven HTML-Elementes.

In HTML werden Layout und Aktion durch die Attribute **im** HTML-Element (Tag) beschrieben, wobei anstelle von HTML die scriptgesteuerte Festlegung von Layout und Aktion sehr oft umfangreicher sein kann.

Um den Programmierer die gleichzeitige Verwaltung von 3 Zeigern zu ersparen, wird **ein Zeiger** erzeugt, der auf die o.g. 3 Zeiger automatisch verweist. Dieser verweisende Zeiger wird **Instanz** genannt. Ein **aktives** HTML-Element wird daher auch als **Instanz** eines HTML-Elementes bezeichnet.

Eine Instanz ist die Schnittstelle, die eine Erzeugung und/oder Veränderung von Daten, Layout und Aktion zur **Laufzeit** zulässt. Die Instanz selbst muss daher ebenfalls im Speicher liegen.

Für den Programmierer ist also die Instanz
eine Sammlung von Komponenten des aktiven HTML-Elementes, die alle im Speicher liegen
und ein Zeiger auf diese Komponenten, also auf das aktive HTML-Element, wobei der Zeiger ebenfalls im Speicher liegt.

Anhand der Instanz kann das aktive HTML-Element zu seiner Laufzeit manipuliert werden. Instanz und aktives HTML-Element sind für den Programmierer synonym.

Erzeugung und Verwaltung einer Instanz

in Javascript:

Aufgrund der Tatsache, dass eine Instanz und ihre Komponenten nur zur Laufzeit existieren, liefert Javascript dem Programmierer die Möglichkeit, im Quellcode anstelle von Zeigern (deren Werte zum Zeitpunkt der Quellcode-Erstellung nicht bekannt ist, da die Kodierung ja außerhalb der Laufzeit des HTML-Elementes und seiner Instanz stattfindet) eine **greifbare** Größe zu nutzen, anhand derer die gesamte Manipulation der Instanz und deren Komponenten programmierbar ist. Diese Größe wird **Variable** genannt.

Eine Variable besteht aus den Komponenten
Namen (Bezeichner) der Variablen, den der Programmierer z.T. selbst festlegen kann,
und Inhalt (Wert) der Variablen, den der Programmierer z.T. selbst festlegen kann.

Wie man leicht sieht, ist eine Variable ein Komponentensammlung und tatsächlich wird sie anhand des Javascript-Objektes "var" instanziert, was die gleichnamige Javascript-Anweisung "var" automatisch besorgt.

Der Programmierer weist im Javascript-Quellcode der Variablen einen Wert zu, der aus der Anweisung zur Instanzierung des HTML-Elementes resultiert, im Speicher liegen muss und somit einen Zeiger besitzt. Die Variable wird also **Zeigervariable** genannt. Die Kodierung der Anweisung zur Instanzierung ist somit der **Platzhalter** für den Zeiger, der erst zur Laufzeit gebildet wird.

Anhand des Variablennamens kann der Programmierer auf die Instanz zugreifen. Dass die Variable zur Laufzeit selbst im Speicher liegt, interessiert den Programmierer nicht. Und dass die Variable zur Laufzeit den Zeiger auf die Instanz beinhaltet, erfreut den Programmierer, aber interessiert ihn letztendlich auch nicht. Wichtig ist nur der **Variablennamen**, der vom Programmierer natürlich sinnvoll ausgewählt sein sollte (falls überhaupt möglich). Dieser Variablenname ist also der Dreh- und Angelpunkt beim Zugriff auf die Komponenten des aktiven HTML-Elementes, wobei dem Programmierer völlig egal ist, wo was wie im Speicher liegt.

Für den Programmierer wird somit der **Name** der Zeigervariablen synonym zur **Instanz** des **aktiven** HTML-Elementes, also synonym zum aktiven HTML-Element und seinen Komponenten, die er mit der Zeigervariablen einzeln ansprechen kann.

Damit wird klar, dass Komponenten eines HTML-Elementes ebenfalls einen Namen haben müssen, der im Javascript-Quellcode kodierbar ist, also außerhalb der Laufzeit des aktiven HTML-Elementes. Diese Komponentennamen sind z.T. durch die Scriptmaschine genau definiert. Der Programmierer muss also darüber Kenntnis haben.

durch die Scriptmaschine:

Die Scriptmaschine erzeugt und verwaltet diverse interne Instanzen, auf die der Programmierer im Javascript-Quellcode nur dann zugreifen kann, wenn ihm der vordefinierte Namen der jeweiligen Instanz bekannt ist. Dieser Name fungiert dann **wie** eine bereits erzeugte und gefüllte **Zeigervariable**, ist also der Platzhalter für die jeweilige Instanz zur Laufzeit.

durch den Browser:

Der Browser erzeugt selbst keine Instanz, sondern nutzt dabei die Scriptmaschine: Diese parst im HTML-Quellcode den Tag zum HTML-Element und ermittelt, welche Attribute vorliegen. Anhand dieser und dem Tag erzeugt die Scriptmaschine die Instanz zum HTML-Element, welches damit zum aktiven HTML-Element wird.

HTML-Element im Browser

Der Browser ist die z.T. visuelle Schnittstelle zum User. Die Verarbeitung eines HTML-Elementes im Browser vollzieht **nur** die Scriptmaschine z.T. in Verbindung mit Komponenten des Betriebssystems (z.B. ActiveX unter Windows). Das Ergebnis der Verarbeitung ist z.B. die Anzeige des HTML-Elementes im Browserfenster. Visuell wird also für den User der Browser aktiv, der die Instanz, die Scriptmaschine und z.T. Komponenten des Betriebsystems benötigt. Erst dadurch ist der Browser in der Lage, das HTML-Element visuell zu verarbeiten, also dem User zugänglich zu machen.

Z.B. Abarbeitung eines HTML-Elementes mit Text:

Die Instanz besitzt intern 3 Zeiger:
1 Zeiger für die Textdaten im Speicher
1 Zeiger für das Layout des Textes
1 Zeiger für die Aktion zum Text also der Algorithmus des Anzeigens

Anhand der Instanz ist der Browser überhaupt erst in der Lage, dem Text des HTML-Elementes (Daten) ein Layout (Eigenschaft) zu verpassen und dann den Text anzuzeigen (Aktion).

HTML-Element und Eigenschaft

Ein Text als Datum eines HTML-Elementes kann natürlich in verschiedenen Erscheinungsformen angezeigt werden, z.B. unterstrichen.

Ein HTML-Element kann eine **Eigenschaft** z.B. "unterstrichen" besitzen.

HTML-Element und Methode

Die Anzeige von Text ist eine Aktion zum HTML-Element. Anstelle von **Aktion Anzeige** wird auch von **Methode Anzeige** gesprochen. Anstelle von anzeigen wird auch von **rendern** gesprochen. Ein HTML-Element wird durch sein rendern sichtbar. Nicht alle HTML-Elemente sind renderbar.

Ein HTML-Element kann eine **Methode** z.B. "Anzeige von Text" besitzen.

Scriptmaschine und Implementation

Die Anzeige des Textes, also die Wirksamkeit eines HTML-Elementes per Methode, wird von der Scriptmaschine beeinflusst und das in dem Umfang, wie die Scriptmaschine die Methode "Anzeige von Text" überhaupt zulässt bzw. in Art und Weise realisiert. Zulassung und Art und Weise werden auch **Implementation** der Aktion genannt. Die Scriptmaschine **kann** zur Textanzeige eine Methode besitzen, muss aber nicht. Die Methode ist also in der Scriptmaschine implementiert oder nicht. Analog gilt das auch für eine Eigenschaft, die in der Scriptmaschine implementiert sein kann, aber nicht sein muss. Über Implementationen entscheidet der Hersteller der Scriptmaschine.

Objekt

Der Begriff "Objekt" ist z.B. synonym zur Komponentensammlung eines HTML-Elementes, geht in seiner Bedeutung aber weiter:
In Javascript und per Scriptmaschine sind **alle** deren Elemente grundsätzlich Objekte, also nicht **nur** die HTML-Elemente.

Objekt und Instanz

Ein Objekt kann nur als Instanz verarbeitet werden, egal ob per Scriptmaschine oder per Javascript.

interne Objekte des Browsers

Die Scriptmaschine implementiert ihre Objekte als Instanzen in den Browser, welche dann **browserinterne** Objekte genannt werden. Die Implementation findet während der Laufzeit des Browser statt.

Z.T. sind im Browser Komponenten des Betriebssystems ebenfalls als Objektinstanzen während der Browserlaufzeit zusätzlich implementiert.

Verwaltung von HTML durch Instanzen

Das HTML-Dokument **und** dessen HTML-Elemente werden durch die Scriptmaschine komplett zu Instanzen von Objekten umgewandelt.

Objekt und Vererbung

Wie im Layout eines HTML-Dokumentes ersichtlich, können HTML-Elemente andere Elemente einschließen. Das einschließende Element wird dabei **Container** genannt.

Manchmal ist es erwünscht, dass ein im Container liegendes Element sich **ähnlich** oder zeitlich synchron oder angepasst zum Container verhält. Der Container muss dann also Einfluss auf sein inneres Element haben können, wobei dem inneren Element der "eigene Wille" erhalten bleibt, welcher aber den des Containers zu beachten hat (Der Internet Explorer unterstützt vielfältige Synchronisierung und Filter).

Wenn der Container das innere Element beeinflussen soll, so muss er dem Element mitteilen, wie es sich zu verhalten hat. Das Verhalten ist also eine Methode des Containers. Sogar eine Eigenschaft zum Zweck der Anpassung des inneren Elementes kann der Container übergeben. Der Container vererbt also seine Methode und/oder und Eigenschaft an das innere Element. Das innere Element wird durch Erbschaft um eine Methode und/oder Eigenschaft des Containers erweitert und kann diese ebenfalls erweitern, falls es die Scriptmaschine zulässt. Vererbung bedeutet damit die Erweiterung des **Objektes** des Elementes im Container. Bei der Vererbung ist immer maßgebend, was der Container **vorgibt**, ohne dabei dem inneren Element sein Eigenleben zu nehmen. Diese Vererbung entspricht einem **Eltern-Kind-Verhalten**. Der Container ist also **Eltern** zum inneren Element, seinem **Kind**. Vererbung wird **mit** dem Objekt instanziert.

Objekthierarchie und Baumstruktur

Aufgrund der Vielfalt im Layout des HTML-Dokumentes ist es nötig, dass Objekte mehrere Eigenschaften, Methoden und sogar Kinder haben können. Die so entstehende **Objekthierarchie** hat also eine **Baumstruktur**. Z.B. besitzt das HTML-Dokument eine Baumstruktur, da es als Container aller HTML-Elemente dient und durch diese lebendig wird. Das HTML-Dokument als Container ist die **Wurzel** der Baumstruktur.
Die Wurzel der Baumstruktur nennt man auch **Root**.

Objekthierarchie und Punktnotation in Javascript

Um in dieser Baumstruktur die Übersicht zu behalten, wird in Javascript die Punktnotation benutzt. Jede Hierarchie-Ebene wird also in der Scriptkodierung von Objekten durch einen vorausgehenden Punkt markiert. Keinen Punkt erhält die unterste Ebene, also die Wurzel (Root).

Scriptmaschine und Objekthierarchie

Die Möglichkeit einer Vererbung und die Hierarchie der Objekte sind in der Scriptmaschine **implementiert**.
Die Scriptmaschine bestimmt damit
welche Objekte existieren (Art der Objekte, auch **Objektklasse** oder **Objekttyp** genannt),
welche Eigenschaften und Methoden zum jeweiligen Objekt existieren,
welche Objekte vererben bzw. erben
welche Objekte vererbte Methoden bzw. Eigenschaften erweitern dürfen
und welche geerbte Methoden bzw. Eigenschaften verändert werden dürfen.

Daran muss sich der Programmierer halten, der damit nicht nur Kenntnisse zu Javascript sondern auch zur Scriptmaschine und den Objekten je nach Version der Scriptmaschine haben muss, in der Hoffnung, dass die Hersteller der Scriptmaschine die Abwärtskompatibilität der Versionen aufrecht erhalten (auch bezüglich der Browser und deren möglichen Scriptmaschinen-Versionen sowie bezüglich der diversen HTML- und CSS-Versionen und deren Umsetzung in den jeweiligen Browser- und Scriptmaschinen-Versionen).

Browserfenster als Objekt

Im Zuge der Vererbung muss das Browserfenster ebenfalls ein Objekt sein. Das Browserfenster ist der virtuelle Container des HTML-Dokumentes, das in ein Fenster geladen wird. Browserfenster und HTML-Dokument sind aber komplett getrennte Objekte. Ein HTML-

Dokument muss zwar ein Fenster besitzen, es aber nicht für Anzeige etc. nutzen. Nicht jedes HTML-Dokument wird gerendert, wenn es der Programmierer so wünscht. Browserfenster und Dokument können bezüglich ihres "Willens" getrennte Wege.

Objekt und Ereignisse

Auch für die Synchronisierung von Elementen werden Ereignisse genutzt. Ein Ereignis signalisiert einen bestimmten Zustand eines Elementes, wo bei der Zustand mit seinem Auftreten einem anderen Element mitgeteilt werden kann. Im Gegensatz zur Vererbung ist aber der Ereignisfluss in beiden Richtungen möglich: Von innen nach außen **und** von außen nach innen, **oder** nur eines von beiden. Das Weiterreichen von Ereignissen **kann** durch den Programmierer ermöglicht aber auch unterdrückt werden, dagegen die Entstehung von Ereignissen z.T. nicht. Der Browser verwaltet permanent Ereignisse, von denen der Programmierer nur z.T. Kenntnis bekommt.
Die Ereignisverwaltung ist z.T. in der Scriptmaschine implementiert.

Datenstrukturen und Script-Objekte

Die Kombination von Daten ist in Form eines Datenstruktur-Objektes möglich . Die Struktur ist dabei das Abbild der Menge von einzelnen Daten. Basis-Datentrukturen werden von der Scriptmaschine durch vordefinierte Script-Objekte bestimmt. Basis-Datenstrukturen sind eine Kombination von Basis-Datentypen, die allesamt in der Scriptmaschine implementiert sind und z.T. Objektcharakter haben. Basis-Datentypen und -Datenstrukturen hängen von der Version der Scriptmaschine ab.

Anhand der Basis-Datentypen und -Strukturen kann der Programmierer weitere Datenstruktur-Objekte kreieren.
Der Programmierer kann jedoch keine Basis-Datentypen und -Strukturen erzeugen.

Scriptmaschine und Javascript als Schnittstelle zum Programmierer

Javascript ist die direkte Programmierschnittstelle zur Scriptmaschine und damit zum Browser.
Die Softwareschnittstelle zum Browser ist die Scriptmaschine.
Die Softwareschnittstelle zum User ist der Browser.
Falls der User programmieren möchte, kann er HTML oder Javascript benutzen:
HTML ist die indirekte Schnittstelle zur Scriptmaschine und die visuell direkte Schnittstelle zum Browser.
Der Browser benötigt immer die Scriptmaschine, welche z.T. Komponenten des Betriebssystems benutzt.

Für den Programmierer als praktisch erweist sich die Kombination von HTML und Javascript (analog von HTML und PHP bei Datenbankanschluss).

Javascript und die Scriptmaschine sind **komplett** objektorientiert. Es wird also **grundsätzlich** mit Objekten und deren Instanzen gearbeitet. Inwieweit der Programmierer zugreifen darf, entscheidet der Hersteller der Scriptmaschine. Man spricht dann von einem herstellerspezifischen **Dialekt** der Programmiersprache Javascript.

Hinweis: Die HTML-Programmierung ist nur z.T. objektorientiert: Über das ID-Attribut bzw. NAME-Attribut, dessen Werte je einem Zeiger entsprechen, sind HTML-Elemente im HTML-Code ansprechbar. Die Benutzung dieser Attribute ist aber **wahlweise.**. Es geht also z.T. auch **ohne** Zeiger.

Für Programmierer unter Windows XP mit dem Internet Explorer bitte **unbedingt** beachten: Die Scriptmaschine unter Windows XP ist wesentlich **weniger fehlertolerant** als die Scriptmaschine unter Windows 98 bei identischer Browserversion und identischem Patch-Stand der Browsersoftware. Unter Windows XP ist der Internet Explorer 6.x implementiert. JScript-Code, der unter Windows 98 einwandfrei funktioniert, muss es unter Windows XP **nicht** ! Javascript-Code, der unter Windows XP funktioniert, wird es auch unter Windows 98 tun. Mit anderen Worten: Die Scriptmaschine unter Windows XP ist bezüglich Fehler **nicht** abwärtskompatibel ! Deswegen bitte **unbedingt** den Scriptcode unter Windows XP testen !

Kompatibilität der Javascript-Dialekte

Für die Kompatibilität der Dialekte **wäre** es wichtig, dass Hersteller sich an gemeinsame Standards halten. Wie man weiß, sind Browserhersteller Konkurrenten.

Objektdesing beim Netscape und Internet Explorer

Im Falle von Netscape 6.x ist aber inzwischen eine Änderung gegenüber dessen Vorgänger 4.7x eingetreten. Es wurde mit Fähigkeiten des verbreiteten Netscape 4.7x konsequent gebrochen, was so mancher Internet-User nicht weiß, der zu dem eventuell noch hartnäckig auf "Netscape schwört" (Letzteres ist ein kontraproduktives Verhalten, dass im Kampf der Browserhersteller zum Zweck der Erzielung monopolistischer Internet-Marktanteile und Profite dem User regelrecht suggeriert wurde, wobei Microsoft langwieriger aber cleverer vorging: Marktenteile auf Basis der Browser-Integration in das Betriebssystem Windows und dessen Anwendungen (auch im Internet) **und** das konsequente aber z.Z. nicht standardisierte Design im Internet Explorer, das für die Integration in das objektorientierte Betriebssystem Windows wichtig ist, aber auch dem JavaScript-Programmierer beim Web-Design zum Vorteil gereicht, solange Windows benutzt wird (Linux wird hier nicht betrachtet). Nicht zu verachten ist auch der Hardware-Ressourcen-Bedarf: Während der IE mit Hochfahren von Windows z.T. instanziert wird, muss Netscape eigene Ressourcen allokieren, was nachteilig ist, wenn IE und Netscape auf dem Rechner aktiviert werden.

Der Netscape 6.x ist dem Internet Explorer näher als nie zuvor. Netscape realisiert damit z.T. endlich Möglichkeiten des Web-Design per JavaScript, die auch der Internet Explorer schon länger bedient.

Knackpunkt ist also die Browserprüfung auf unterstützte browsereigene Objekte. Es könnte möglich sein, dass der Netscape auch ein document.all abbilden kann. Netscape 6.x hat definitiv keine <LAYER>- und <ILAYER> Unterstützung mehr, vermutlich auch keine Objekt-Unterstützung von Layer. Die Verwendung des <DIV>-Tag wurde auch geändert, obwohl dieser Tag gerade für DHTML wichtig ist.

Aufgrund der unzähligen Browserversionen und -änderungen ist es nötig, die Objektfähigkeit des Browsers zu prüfen, wenn ein Objekt benutzt werden soll. Natürlich muss der Benutzer im Browser JavaScript eingeschaltet haben. Es wird tendenziell sichtbar, dass der User ohne JavaScript-Zuschaltung mit dem rein HTML-basierenden Seiten ins Abseits gerät. Der Preis für JavaScript: Es kann missbräuchlich auf Basis der Unwissenheit des Internet-Users (der eigentlich "nur surfen will") benutzt werden. Der User muss sich entscheiden, für ihm vertraute Seiten JavaScript zuzuschalten, Software wie ein Virenscanner und eine Firewall zu nutzen, oder auf Design zu verzichten.

Scriptmaschine und implementierte Objekte

Um ein Objekt nutzen zu können, ist es wichtig zuwissen, ob der Browser das Objekt überhaupt kennt, also ob die aktuelle Scriptmaschine das Objekt implementiert, also vordefiniert hat. Anstelle von vordefinierten Objekten spricht man auch von browserinternen Objekten.

Beispiel:

```
if  (window.createPopup())
{ ..... }     // window.createPopup() kennt nur der IE ab 5.5
```

Die Browser-Version ist immer an die Version der Scriptmaschine gebunden. Implementationen in einer jüngeren Version einer Scriptmaschine müssen vom Browser älteren Datums nicht unbedingt nutzbar sein. Dafür ist die Scriptmaschine in der Regel zu älteren Browsern abwärtskompatibel. Browserinterne Objekte hängen also von der Scriptmaschine ab, die diese Objekte implementiert. Was die Scriptmaschine nicht kennt, kann der Browser erst recht nicht wissen.

Da sich die Scriptmaschinen zum Browser unterscheiden, muss also **vor** der Nutzung eines browsereigenen Objektes bekannt sein
welcher Browserhersteller, also Name des Browsers
welche Browserversion
welche Version der Scriptmaschine.

Tipp: Das Prüfen auf ein Objekt kann mit der Zuweisung des Objektzeigers auf eine Variable verbunden werden (analog zur Kodierung des ID-Attributes im HTML-Tag: Der Wert des ID-Attributes ist ein String, der vom Browser als Zeiger auf das Objekt laut HTML-Tag verwendet wird). Die Nutzung einer Zeigervariable erhöht die Performance des Browsers.

Beispiel für den Internet Explorer:

```
//          Prüfung auf Browsertyp
//          Dieser Quellcode muss VOR allen anderen Routinen kodiert sein, damit zuerst abgearbeitet
var ns = document.layers ? true : false;
var ie = document.all ? true : false;

var IE_ZeigerAufFeldAllerElementeImHTMLDokument;        // Variable hier deklarieren,
                                                        //          damit sie im gesamten Scriptcode verfügbar ist
                                                        // Variable ist damit global zum gesamten Scriptcode

if (ns)
{// Code für Netscape}
else
{
        if (ie)
        {
                // Zeiger auf alle Elemente (Objekte) im HTML-Dokument holen
                //          Hinweis: document.all ist eine Collection, also ein Feld der Zeiger aller Elemente im Dokument
                IE_ZeigerAufFeldAllerElementeImHTMLDokument = document.all;
                        // Vorteil der Variable IE_ZeigerAufFeldAllerElementeImHTMLDokument:
                        // es muss später nicht erneut document.all kodiert und damit auch nicht mehr
                        //          geparst werden

                // Feld instanzieren, dass die Zeiger auf alle HTML-Elemente mit P-Tag  im Dokument aufnimmt
                //          Hinweis: document.all also IE_ZeigerAufFeldAllerElementeImHTMLDokument ist ja
                //              der Zeiger auf das Feld (Collection) aller Elemente im Dokument,
                //              also die Feld-Methode .tags() benutzen, die einen Zeiger auf ein Zeigerfeld
                //              (Collection) aller P-Tags im Dokument liefert
                //          Zeigerfeld der P-Tags ist eine Teilmenge aller Elemente im Dokument
                var ZeigerAufFeld = IE_ZeigerAufFeldAllerElementeImHTMLDokument.tags("P");
                        // Vorteil der Variable ZeigerAufFeld
                        //          es muss später nicht mehr erneut
                        //              IE_ZeigerAufFeldAllerElementeImHTMLDokument.tags("P")
                        //          kodiert und damit auch nicht mehr geparst werden

                        // performance-schädigend wäre auch folgende Kodierung:
                        //          var ZeigerAufFeld=document.all tags("P");
```


```
                                        //          denn dann würde das Parsen den Zeiger document.all nochmals ermitteln

                        // prüfen ob überhaupt ein P-Tag im Dokument ist
                        //          Hinweis:   ZeigerAufFeld ist der Zeiger auf eine Teilmenge aller Elemente im Dokument
                        //                     und zwar auf die Teilmenge aller P-Tags im Dokument
                        //                     Die Teilmenge kann nur existieren, wenn überhaupt P-Tags
                        //                     im Dokument enthalten sind.
                        //                     Der Zeiger auf die Teilmenge kann nur korrekt sein, wenn die Teilemenge
                        //                     überhaupt existiert.
                        //                     Ein Zeiger, der nirgendwo hinzeigt, hat den Wert null (nicht numerisch 0 )
                        if (ZeigerAufFeld!=null)
                        {
                                // es existiert mindestens 1 P-Tag

                                // Anzahl der P-Tags im Dokument ermitteln per Feld-Eigenschaft .length

                                var AnzahlDerPTags = ZeigerAufFeld.length;          // Anzahl ab 1

                                // man könnte jetzt noch mal prüfen, ob die Anzahl > 0 ist,
                                //          aber es muss ja bereits mindestens 1 P-Tag-Element existieren

                                // alle P-Tag-Elemente (Objekte) im Dokument im Textbereich unterstreichen
                                //          Hinweis: Da die Teilmenge aller P-Tags ein Feld (Collection) ist,
                                //                   kann also mit dem Feldindex gearbeitet werden, wobei der
                                //                   Wert des Indexes die Position des Feldelementes im Feld angibt
                                //          Das Abklappern aller möglichen Werte im Index erfolgt am besten per Schleife.
                                for ( var Index=0; Index < AnzahlDerPTags; Index++)
                                        // Index immer ab 0 !!
                                        // Ablauf:  Index  nehmen und prüfen ob < Anzahl der P-Tags
                                        //          wenn ja dann Index verwenden im Schleifenkörper
                                        //                   und nach der Verwendung
                                        //                   den Index automatisch um 1 erhöhen
                                        //                   (nach      entspricht  Index++
                                        //                    davor     entspricht  ++Index)
                                {

                                        // Schleifenkörper: Hier den Index verwenden
                                        //          Hinweis: . style ist das style Objekt, mit dem ein Layout
                                        //                   geändert werden kann
                                        //                   .style ist dabei wieder ein Zeiger auf das Layout des P-Tag
                                        //                   ein  P-Tag unterstützt im style Objekt die Eigenschaft
                                        //                   textDecoration
                                        ZeigerAufFeld[Index].style.textDecoration="underline";
                                }
                        }

                }
        }
```

Microsoft ändert fortlaufend die Active-X-Eigenschaften von Windows und somit auch des Internet Explorers

Diese fortlaufenden Änderungen muss der Programmierer in Erfahrung bringen.
Der Programmierer kann sich definitiv nicht auf Verfügbarkeit von Active-X-Controls verlassen und muss damit rechnen, dass seine Webseiten schlagartig nicht mehr komplett laufen weil u.a. Programmcode noch nicht angepasst ist. Ebenfalls muss der Programmierer Varianten von Windows und Patchzustände beachten, die prinzipiell Kostenprobleme verursachen können.
Mit anderen Worten: Wer Microsoft-Komponenten nutzt, muss wissen, was ihm blüht ... siehe nachfolgende Beispiel für Risiken.

Prinzipielle Lizenzprobleme für den Programmierer

Microsoft verlangt Linzensierung von Windows. Bezüglich Windows-Versionen gibt es die Updatestufen z.B. per Servicepacks
Ein Windows mit Servicepack fällt unter die Lizenz des geupdateten Windows.
Ein Windows mit Vorversion zum Servicepack bedarf einer anderen Lizenz.
Will man z.B. den Internet Explorer 7 und 6 parallel testen, benötigt man 2 Windowslizenzen, da beide Versionen nicht parallel installierbar. Dazu kommt, dass es den IE 6 in 2 Versionen gibt: Win SP1 und SP2 (IE 7 nur ab Win SP2).
Für 3 Browserversionen benötigt man 3 Windowslizenzen, will man parallel testen.
Ein Blick auf Browser-Konkurrenzprodukte klärt die Sachlage unschlagbar: Opera ist z.B. parallel installierbar.

Hinweis: Man suche doch mal im Internet nach einem kostenlosen HTTP-Server vom Microsoft, um IE-Seite testen zu können, die JScript nutzen (inklusive Debugger). Denn sollte kein kostenloses Angebot findbar sein, kommen die Kosten von Entwicklungssoftware zum IE hinzu. Ein Blick auf Konkurrenz-HTTP-Server klärt die Sachlage: Apache-HTTP-

Server ist kostenlos, allerdings nicht einfach einzurichten (Hinweis: Der HTTP-Server sollte virtuelle Hosts einrichten können und korrekt mit der Firewall des Users zusammenarbeiten können).

Abänderungen wegen Sicherheitspatches der jeweiligen Windows-Versionen

Abschaltungen von Active-X-Controls erfolgen auch im Rahmen der Sicherheitspatches zu Windows-Versionen.
Es ist auch möglich, dass wegen Sicherheitslücken abgeschaltet wird und somit Komponenten einer Webseite je nach Windowsversion nicht mehr laufen.

Im Rahmen der Sicherheitspatches ist es Microsoft sogar gelungen, Webseiten, die den MS-Encoder zur Komprimierung von HTML- und JScript-Code nutzen, schlagartig unnutzbar zu machen: Ein Bug in einem Patch zu Windows XP - Q918899
Das Patch verusacht IE-Browser-Absturz bei per MS ScriptEncoder gepacktem JScript unter SP1 und 2 wenn HTTP 1.1 mit Kompression genutzt wird z.B. bei
onlick-Handler auf IMG
klick ins Fenster per aktivem Popup

Der Absturz ist "read" -Fehler von immer ein und derselben Speicherstelle.
User, die dieses Patch installiert haben, können ab sofort keine IE-Seiten mit codiertem Script mehr ansehen.
Microsoft stellt Abhilfe nach geraumer Zeit zur Verfügung, jedoch spezifisch nach Windows XP-Version:
Patch Q918899 für
Windows XP SP1Download für jedermann bereitgestellt
SP2 nur auf kostenpflichte telefonische Anfrage des Users per Downloadlink bereitgestellt, da Microsoft explizit die User registriert haben will, bei denen das Patchproblem auftritt (User muss sich Telefonnummer besorgen)
Solange also das Patch zum fehlerhatem Patch vom User nicht installiert wird, z.B. weil der User keine Ahnung hat, dass und wo er sich die Telefonnummer von Microsoft besorgen muss bzw. zu besorgen hat, wird der User IE-Seiten mit komprimierten Code dauerhaft nicht nutzen können. (Microsoft-Support ist z.T. nur in Englisch).

Abänderungen wegen Browser-Inkompatibilität

Popupblocker-Fehler

Die Microsoft Browser-Version IE 7 ist nicht abwärtskompatibel bezüglich Popup per window.createPopup()
Popup per window-Objekt ist ein Markenzeichen des IE, das im IE 7 nicht mehr fehlerfrei nutzbar ist.
Der Fehler liegt in der Popup-Blockerverwaltung des IE und wurde mit dem IE 7 implementiert.
Der Fehler tritt nicht auf, wenn ein Fenster per window.open() erzeugt wurde.
Bedingung:
Scripfehleranzeige ist erlaubt im IE 7
Popupblocker ist im IE abgeschaltet
ein aktives Fenster (Register) mit Dokument, dass fortlaufend (rekursiv) genau 1 window.popup per .show()erzeugt.
ein weiteres Fenster (Register) z.B. leere Seite (about:blank)
beide (Register) liegen in einer gemeinsamen IE-Instanz
Ablauf: Wird Focus auf Register der leeren Seite gehalten und wird parallel das Popup per .show() erzeugt, bricht der Browser das Dokument mit .show() ab (Scriptfehler).
Der Popupblocker für die leere Seite verursacht den Programmfehler im Dokument mit .show(). Es wird folgende Meldung angezeigt (in der Informationsleiste):
'Ein Popup wurde geblockt. Klicken Sie hier, um das Popup bzw. weitere Optionen anzuzeigen.'
Die Bedeutung der Meldung laut Microsoft-Hilfe im IE 7:
Der Popupblocker hat ein Popupfenster geblockt. Sie können den Popupblocker deaktivieren oder Popups temporär zulassen, indem Sie auf die Informationsleiste klicken.
Die Realität zur obigen Meldung ist völlig anders:
Linke oder rechte Maus auf die Meldung liefert z.B. Einstellungen darunter
Popupblocker einschalten
weitere Informationen
jedoch keine Möglichkeit wie laut Bedeutung
Damit gilt: Der abgeschaltete Popupblocker ist in Wirklichkeit aktiv.
Pikant: Ein Popup erscheint normalerweise auch über fremde Fenster, die nicht das Popup erzeugt haben (z.B. Fenster einer Windowsanwendung z.B. einer anderen IE-Instanz)
Der Popupblocker des IE bemeckert aber NUR Webseite, die das Popup erzeugt.
Durch das Abwürgen von Popup wird das Popup natürlich auf und für anderen Seiten nicht relevant; im Falle einer anderen IE-Instanz also auch für diese nicht relevant, obwohl diese Instanz per Popupblocker verwaltet wird.
Der Popupblocker beschneidet die Popup-Reichweite an der Wurzel, ist aber nicht objektorientiert zu den anderen Webseiten (die nicht das Popup erzeugt haben).
Der Popupblocker ist nicht als Filter aufgesetzt sondern reingestrickt worden.
Der Popupblockerfehler verändert die Eventverwaltung:
Es werden u.a. ignoriert
onfocus
onblur
onfocusin

```
   onfocusout
```

und viele andere, so dass trotz Events z.B. des Body der Popupblockerfehler entsteht.

```
// nachfolgender Code setzt focus nicht neu: Fenstereintrag in Taskleiste blinkt eventuell
window.focus();
window.document.focus();
if(document.body!=null)
{if(document.body.style!='hidden')          // wenn  hidden so focus() nicht möglich (Scriptfehler erzeugt)
 {document.body.focus();}
}
// wenn paralleles Fenster offen (on oder offline), so Scriptfehler erzeugt
popupzeiger.show(...);
```

Hinweis: Der Popupfehler ist so elementar, dass die vielen Beta-Testphasen des IE mehr als fragwürdig erscheinen, wie die Angabe von Microsoft, dass Code neu programmiert wurde, um den IE sicherer zu machen.

focus-Methode beim IE 7

windows.focus() document.focus() und body.focus() funktionieren NICHT
zwischen Register in einem IE-Fenster
zwischen Fensters z.B. in Taskleiste

Hinweis:
.focus() setzt Element aktiv, gibt dem Element den Focus und feuert dann onfocus
.setActive() ist Teilmenge von .focus(): nur das aktiv setzen
funktioniert nicht mit allen Elementen, mit denen .focus() funktioniert

animierte Gif (mit Timer)

Animierte Gifs (mit Timer), die unter IE 6 korrekt laufen, müssen unter IE 7 im Timer nicht mehr laufen:
z.B. garnicht mehr sichtbar, oder Timer nicht verwendet.
Dann müssen animierte Gif-Bilder nach IE-Version bereitgestellt werden.

Abänderungen wegen Rechtstreitigkeiten von Microsoft mit Fremdanbietern

Ein sehr bekanntes Beispiel ist die nachträglich eingeführte Einschränkung von Active-X-Controls wegen Patentwahrung durch Microsoft, wobei für den JScript-Programmierer massive Änderungen eintreten.

Wegen Patentwahrung hat Microsoft ein zunächst freiwilliges Patch herausgegeben, dass bei ActiveX-Control per APPLET, EMBED oder OBJECT, die auf dem Bildschirm rendern (mit oder ohne Userschnittstelle), dafür sorgt, dass bei mouseover über das Control eine Sprechblase erscheint, die darauf hinweist, dass das Objekt als ActiveX-Control klickbar ist.
Diese Sprechblase erscheint auch, wenn das Control keine Userschnittstelle hat, also diese gar nicht klickbar ist.

Es wurde das Eventmodell gleichzeitig geändert:

Es werden alle Events solange unterdrückt, bis der User die Sprechblase geklickt hat.
Das Klicken muss auf das Objekt im Sprechblasenrahmen erfolgen, der so groß ist, wie die Dimension, in der gerendert wurde.
Es muss also ERST per Mausklick das Control aktiviert werden, ehe das Control klickbar und damit die Eventsteuerung aktiviert ist.
Ein Control, dass programmtechnisch zwar was rendert, aber ansonsten ohne sichtbare programmtechnisch startet, muss ebenfalls geklickt werden, obwohl es bereits läuft und es nichts zu klicken gäbe (wenn keine Eventsteuerung eingebaut wurde).
Wegen blockierter Eventsteuerung ist also die Sprechblase z.B. nicht automatisch klickbar.
Die Eventauslösung per nicht-objekteigenen Eventhandler, der für das Objekt per fireEvent() ein Event auslöst, ist solange blockiert, bis der User die Sprechblase geklickt hat.

style.visibility='hidden' wird ignoriert

Die Sprechblase erscheint auch dann, wenn das Control mit style.visibility='hidden' belegt ist, also sich unsichtbar rendert:
Der Sprechblasenrahmen hat genau die Dimension wie die des unsichtbaren Controls. Der Sprechblasenrahmen erscheint also Zusammenhangslos, und der User weiß nicht, warum er klicken soll, wenn er nichts sieht. Vor allem weiß er nicht, WAS er klickt ... ideale Basis für Schadsoftware per Script.

Diese Sprechblase erscheint nur DANN NICHT, wenn die Userschnittstelle mit Breite == Höhe == 0 gerendert wird. Sollte die Userschnittstelle in einem Container liegen, z.B. DIV, dann wird der Container, wenn er in der Dimension kleiner ist, also die Userschnittstelle, angepasst. Daher muss der Container ebenfalls mit Breite == Höhe == 0 gerendert werden. Wegen Dimensionierung auf 0 sollte style.visivility="hidden" sein. Im Falle eines Containers reicht es, den style des Containers zu ändern, da visibility normalerweise vererbt wird an Kinder, also auch an das Control.

Abänderung wegen Abschaltungen

DirectX ist wegen Abschaltung von Active-X--Controls nicht mehr abwärtskompatibel:
Z.B. wurde bei Win XP SP2 Direct Animation aus DirectX schlagartig durch Abschaltung von Bibliotheken dezimiert, die es bei Win XP SP1 aber noch gibt.

Hier ein Beispiel aus dem Jahr 2004: Abschaltungen von Active-X-Controls

ActiveX-Controls und Unterstützung/Verbot 20041215

erlaubt sind noch

Tabular Data-Steuerelement{333C7BC4-460F-11D0-BC04-0080C7055A83} Das TDC (Tablular Data-Steuerelement) ermöglicht die Weiterverarbeitung von Daten, die nur im Textformatvorliegen, beispielsweise durch Darstellung in einer Tabelle oder Sortierung. Weitere Informationen:• http://msdn.microsoft.com/workshop/database/tdc/tabular_data_control_node_entry.asp(http://msdn.microsoft.com/workshop/database/tdc/tabular_data_control_node_entry.asp)

Microsoft Agent Control - Version 2.0 {D45FD31B-5C6E-11D1-9EC1-00C04FD7081F} Microsoft Agent repräsentiert die neue Generation des ursprünglichen Office-Assistenten. Anstatt den Assistenten jedoch innerhalb eines Rahmens darzustellen wird hier lediglich der Charakter bzw. Agent selbst dargestellt und kannauch in Webseiten verwendet werden. Weitere Informationen:• http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm(http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm)

Microsoft MSChat-Steuerelement-Objekt 2.0 - 2.5 {D6526FE0-E651-11CF-99CB-00C04FD64497} Dieses Steuerelement wird von Webautoren verwendet, um text- und graphisch basierte Chatgemeinden für Echtzeitkonversationen im Web zu erstellen.

Microsoft ActiveX Upload-Steuerelement, Version 1.5 {886e7bf0-c867-11cf-b1ae-00aa00a3f2c3} Dieses Steuerelement kann auf vielerlei Art genutzt werden, um auf einfache Weise Webinhalte via Drag and Drop zuveröffentlichen. Weitere Informationen:• 230298 (http://support.microsoft.com/kb/230298/DE/) - Posting Acceptor Release Notes
• http://msdn.microsoft.com/workshop/management/tools/reference/file_upload_control.asp (http://msdn.microsoft.com/workshop/management/tools/reference/file_upload_control.asp)

verboten sind

Datenbindung RDS {BD96C556-65A3-11D0-983A-00C04FC29E36} {BD96C556-65A3-11D0-983A-00C04FC29E33} Die RDS (Remote Data Service) Steuerelemente ermöglichen dem Browser, client-basierte SQL Abfragen an einen Webserver zu stellen. Inzwischen wurde RDS jedoch durch neuere Standards wie SOAP abgelöst, von einer weiteren Verwendung von RDS wird daher abgeraten. Weitere Informationen:• 184375 (http://support.microsoft.com/kb/184375/DE/) - Sicherheitsaspekte bei RDS 1.5, IIS 3.0 oder 4.0 und ODBC
http://msdn.microsoft.com/library/en-us/iissdk/iis/remotedatabindingwithremotedataservice.asp (http://msdn.microsoft.com/library/en-us/iissdk/iis/remotedatabindingwithremotedataservice.asp)
http://msdn.microsoft.com/library/en-us/dnmdac/html/data_mdacroadmap.asp (http://msdn.microsoft.com/library/enus/dnmdac/html/data_mdacroadmap.asp)

XMLDSO, XMLDocument, DOMDocument, und XMLIslandPeer {550dda30-0541-11d2-9ca9-0060b0ec3d39} {CFC399AF-D876-11d0-9C10-00C04FC99C8E} {e54941b2-7756-11d1-bc2a-00c04fb925f3}{7108ECB4-AFDC-11D1-ADC1-00805FC752D8}XMLDSO, XMLDocument, DOMDocument, und XMLIslandPeer ermöglichen die Verarbeitung von XML Daten,etwa die Bindung von HTML Elementen an einen XML Datensatz, oder das Einlesen, Manipulieren, und Zurückschreiben von XML Daten.

Die Steuerelemente DOMDocument und XMLIslandPeer bzw. die dazugehörigen ClassIDs sind nicht mehr aktuell, so dass von einer generellen Freigabe dieser Steuerelementgruppe abgeraten wird. Weitere Informationen:• http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_concepts2_7ook.asp(http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xml_concepts2_7ook.asp)

Internet Explorer

Active Setup / IE Active Setup-Steuerelement {F72A7B0E-0DD8-11D1-BD6E-00AA00B92AF1}Dieses Steuerelement enthält die in Microsoft Security Bulletin MS99-037 beschriebene Sicherheitsanfälligkeit. Umeine weitere Ausführung zu verhindern wurde im Rahmen dieses Security Bulletins ein Kill-Bit gesetzt, so dass selbst bei einer Freigabe dieses Controls eine Ausführung blockiert wird. Weitere Informationen:• http://www.microsoft.com/technet/security/bulletin/ms99-037.mspx(http://www.microsoft.com/technet/security/bulletin/ms99-037.mspx)
http://www.microsoft.com/technet/security/bulletin/fq99-037.mspx (http://www.microsoft.com/technet/security/bulletin/fq99-037.mspx)
240797 (http://support.microsoft.com/kb/240797/DE/) - So verhindern Sie die Ausführung von ActiveX-Steuerelementen in Internet Explorer

Media Player / Active Movie Runtime {A4001DE0-7075-11d0-89AB-00A0C9054129} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das Active Movie Runtime Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wirdabgeraten.

Media Player / ActiveMovie-Steuerelement{05589FA1-C356-11CE-BF01-00AA0055595A} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das Active Movie Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wird abgeraten.

Media Player / Microsoft NetShow Player {2179C5D3-EBFF-11CF-B6FD-00AA00B4E220} Die Funktionalität dieses Steuerelements wird wird nun durch das Windows Media Player ActiveX Steuerelement abgedeckt. Das NetShow Player Steuerelement wird daher nicht mehr unterstützt, von einer Freigabe wird abgeraten.

Media Player / Windows Media Player {22D6F312-B0F6-11D0-94AB-0080C74C7E95} Dies ist das Steuerelement für Windows Media Player version 6.4 und war Installationsbestandteil bis einschließlich Windows Media Player Version 8. Ab Windows Media Player 9 wurde diese ClassID durch die neue ClassID {6BF52A52-394A-11D3-B153-00C04F79FAA6} abgelöst, deren Verwendung stattdessen empfohlen wird. Ab Windows Media Player Version 9 wirdferner die alte ClassID anhand eines Wrappers automatisch auf die neue ClassID umgeleitet. Die ClassID für Windows Media Player Version 9 ist jedoch nicht in der Liste der vom Administrator genehmigten Steuerelemente enthalten, undmuss bei Bedarf manuell hinzugefügt werden.

Animierte Schaltflächen{0482B100-739C-11CF-A3A9-00A0C9034920} Dieses Steuerelement erlaubte in frühen Versionen des Internet Explorer die Verwendung animierter Schaltfächen auf Webseiten. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von derFreigabe des Steuerelements wird daher abgeraten.

IE Label-Steuerelement
{99B42120-6EC7-11CF-A6C7-00AA00A47DD2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 auch kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten.

Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

IE Menu-Steuerelement {74701400-9DD9-11CF-A662-00AA00C066D2} Dieses Steuerelement ermöglicht die Handhabung von Menüstrukturen in Webseiten, wird jedoch nicht mehr unterstützt und dürfte nur noch selten Verwendung finden. Von einer Freigabe des Steuerelements wird daher abgeraten.

IE Preloader-Steuerelement {16E349E0-702C-11CF-A3A9-00A0C9034920} Dieses Steuerelement ermöglichte das Vorladen von Webseiten, ist jedoch inzwischen nicht mehr aktuell, wird nicht mehr unterstützt und dürfte nicht mehr im Einsatz sein. Aufgrund einer potentiellen Sicherheitsanfälligkeit in diesem Steuerelement wird von einer Freigabe abgeraten. Weitere Informationen:• 231452 (http://support.microsoft.com/kb/231452/DE/) - Update Available for "Legacy ActiveX Control" Issue

IE Timer-Steuerelement {59CCB4A0-727D-11CF-AC36-00AA00A47DD2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

MCSiMenü {275E2FE0-7486-11D0-89D6-00A0C90C9B67} Dieses Steuerelement dient der Anpassung von Popupmenüs, ist jedoch nicht mehr aktuell und wurde nach Windows 98 nicht mehr ausgeliefert. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten.

Popupmenüobjekt {7823A620-9DD9-11CF-A662-00AA00C066D2} Dieses Steuerelement ist nicht mehr aktuell und seit Internet Explorer Version 5 kein Bestandteil der Installation mehr. Das Steuerelement wird nicht mehr unterstützt und dürfte nur noch vereinzelt im Einsatz sein. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• 190045 (http://support.microsoft.com/kb/190045/DE/) - INFO: ActiveX Controls That Are Removed from Internet Explorer 5

Microsoft Agent Control - Version 1.5 {F5BE8BD2-7DE6-11D0-91FE-00C04FD701A5} Microsoft Agent repräsentiert die neue Generation des ursprünglichen Office-Assistenten. Anstatt den Assistenten jedoch innerhalb eines Rahmens darzustellen wird hier lediglich der Charakter bzw. Agent selbst dargestellt und kann auch in Webseiten verwendet werden. Diese Version des Steuerelements ist jedoch nicht mehr aktuell und wird nicht mehr unterstützt. Von einer Freigabe des Steuerelements wird daher abgeraten. Weitere Informationen:• http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm (http://msdn.microsoft.com/library/partbook/egvb6/introducingmicrosoftagent.htm)

4.1. in Javascript vordefinierte Operationen mit Objekten

Operationen mit Objekten erfolgen in Script neben den bekannten Operatoren und Anweisungen auch per vordefinierte Eigenschaften und Methoden, die objekt-unabhängig (objekt-übergreifend) sind, aber eventuell eine objektspezifische und/oder browserspezifische Erweiterung bzw. Abänderung besitzen. Dazu kommt, dass Eigenschaften und Methoden inzwischen **browserspezifisch** als deprecated (unerwünscht, veraltet) gelten können und nur noch auf Basis von herstellerbezogener Toleranz in der Scriptmaschine implementiert sind. Von diesen (z.T. sehr oft benutzten) Eigenschaften und Methoden sollte Abstand genommen werden, oder es muss pro Browserversion geprüft werden, ob die Implementierung noch vorliegt.

4.1.1. Operationen mit Objektinstanzen (Auswahl)

4.1.1.1. Ermittlung der Objektklasse einer Objektinstanz als Zeichenkette (typeof)

Bsp: if (typeof (objekt_instanz) == "String") .....

Typen: "undefined"

"function"
"object"
"number"
"boolean"
"string"

Bsp: typeof 42 ergibt "number"

4.1.1.2. Vergleich von Objektinstanzen (valueOf) (Zeigervergleich)

Vergleiche erfolgen immer als Zahl im 32-Bit-Format.

Bsp: if (objekt_instanz_1.valueOf() == objekt_instanz_2.valueOf()) ...../// Zeigervergleich

4.1.1.3. Löschen einer Objektinstanz (incl. Speicherfreigabe) per null-Zuweisung

Die Zuweisung des Wertes **null,** also eines Null-Zeigers, bewirkt die Löschung der Variablen (inklusive Speicherplatz)

```
Bsp:    var variable=0;     // numerische Variable instanzieren
        variable=null;      // Zeiger löschen, also Instanz aufheben
```

Es kann auch die Anweisung delete verwendet werden, um das gesamte Objekt oder Teile davon zu löschen (inklusive Speicherplatz)

```
Bsp.:   var Wert = delete objekt_instanz.eigenschaft;

            Wert    true,   so Löschung erfolgreich
                    false,  so Löschung nicht erfolgt
```

Script-Objekte sind nicht löschbar.
Nur per Prototyping zum Script-Objekt hinzugefügte Eigenschaften und Methoden sind löschbar.

4.1.2. Standardmethoden aller Objekte in Javascript (Auswahl)

siehe auch Script-Objekt Object

4.1.2.1. Boolean()

konvertiert einen Wert nach Boolean
Syntax:

```
[ var Zeiger = ] Boolean(Wert)

    Wert    true oder false oder beliebiger Wert
            wenn 0, -0, +0, null, false, NaN, undefined oder Leerkette ""
                so wird false verwendet
            sonst wird true verwendet
                z.B. "false" wird als true verwendet
```

4.1.2.2. decodeURI() (IE ab 5.5, NS 6.x)

dekodiert einen Uniform Ressource Identifier (URI), der mit der Methode encodeURI() erzeugt wurde
ersetzt die Methode unescape() welche deprecated ist
Syntax:

```
[ var Zeiger = ]  decodeURI(Kette)

    Kette   encoded Uniform Resource Identifier, der mit der Methode encodeURI () erzeugt wurde

    Zeiger  auf dekodierten String
```

Beispiel:

```
alert(decodeURI("My%20phone%20 #%20is%20123-456-7890"));
                                        // erzeugt "My phone # is 123-456-7890"
```

4.1.2.3. decodeURIComponent() (IE ab 5.5, NS 6.x)

dekodiert eine Komponente einer Uniform Ressource Identifier (URI), der mit der Methode encodeURI() erzeugt wurde
Syntax:

```
[ var Zeiger =]  decodeURIComponent(Kette)

    Kette   encoded Uniform Ressource Identifier, der mit der Methode encodeURIComponent() erzeugt wurde

    Zeiger  auf dekodierten String
```

4.1.2.4. encodeURI() (IE ab 5.5, NS 6.x)

kodiert einen String als kompletten Uniform Ressource Identifier (URI):
Ersatz von Zeichen durch Escape-Sequenzen (UTF-8 Zeichen) der Form %xx
es werden alle Zeichen kodiert, also durch Escape-Sequenzen ersetzt, außer folgende Zeichen
. , / ? : @ & = + $ - _ . ! ~ * ' ( ) #

Buchstaben
Ziffern

ersetzt die Methode escape() welche deprecated ist
Syntax:
[var Zeiger =] encodeURI(Kette)

Kette URI, der kodiert wird

Zeiger auf kodierten kompletten Uniform Ressource Identifier (URI)

Beispiel:
```
alert(encodeURI("My phone # is 123-456-7890"));
                                        // erzeugt "My%20phone%20 #%20is%20123-456-7890"
```

4.1.2.5. encodeURIComponent () (IE ab 5.5, NS 6.x)

kodiert einen Teil-String aus einem kompletten Uniform Ressource Identifier (URI):
Ersatz von Zeichen durch Escape-Sequenzen (UTF-8 Zeichen) der Form %xx
es werden alle Zeichen kodiert, also durch Escape-Sequenzen ersetzt, außer folgende Zeichen
. , / ? : @ & = + $ - _ . ! ~ * ' ( ) #
Buchstaben
Ziffern

Syntax:
[var Zeiger =] encodeURIComponent (Kette)

Kette Teil-String des URI

Zeiger auf kodierten Teil-String des Uniform Ressource Identifier (URI)

4.1.2.6. escape()

kodiert einen String oder ein Literal in das Unicode-Format
ab Javascript 1.5 (Netscape 6.x) deprecated und zu ersetzen durch die Methoden encodeURI()
encodeURIComponent()
Ersatz von Zeichen durch Escape-Sequenzen (UTF-8 Zeichen) der Form %xx
es werden alle Zeichen kodiert, also durch Escape-Sequenzen ersetzt, außer folgende Zeichen
. , / ? : @ & = + $ - _ . ! ~ * ' ( ) #
Buchstaben
Ziffern
URI (Uniform Resource Identifiers) werden nicht kodiert
UTF-8 Zeichen: Teil des Unicode (0 bis 255)
Syntax:
[var Zeiger =] escape(Kette)

Kette String

Zeiger auf kodierten String

Bsp: Blank --> %20
& --> %26

Beispiel für Einbindung einer Suchmaschine:

```
var markierter_text=document.selection.createRange().text;
        // oder var markierter_text=prompt('Suchbegriff: ');
var suchmaschinen_url='http:// .....';
var suchmaschinen_parameter='.....';

if (markierter_text)
{location.HREF=suchmaschinen_url + suchmaschinen_parameter + escape(markierter_text);}
else
{location.HREF=suchmaschine_url; }
```

Beispiel für altavista: suchmaschinen_url 'http://altavista.de/'
suchmaschinen_parameter 'cgi-bin/query?pg=q&what=web&q='

4.1.2.7. eval()

parst einen String aus JavaScript-Anweisungen (kein HTML-Code) und führt diese sofort aus
JavaScript-Anweisungen
dürfen keine Referenz auf sich selbst haben (sonst endlose Rekursion !)
können Referenzen auf existierende Objekte (Variablen) und deren Objektmethoden wie Objekteigenschaften enthalten
z.B. für Wertzuweisung etc.
Objekte (Variablen) erzeugen und instanzieren
ab JavaScript 1.5 auch eval als Anweisung enthalten (Achtung: endlose Rekursionen vermeiden !)

Syntax:

[var Zeiger =] eval(Kette)

Kette String mit JavaScript-Anweisungen

Zeiger auf Ausdrucksergebnis
nur erzeugt, wenn JavaScript-Anweisungen einen Ausdruck als letzten Teil oder nur einen Ausdruck in der Kette enthalten
Es ist zu empfehlen, anstelle des Zeigers die Wertzuweisung auf eine Variable direkt innerhalb der JavaScript-Anweisungen zu realisieren.

Beispiele:

```
var Kette = eval("new String('2+2')");                  // liefert Zeiger auf den String '2+2'
eval("var Kette = new String('2+2')");                  // liefert nichts
                                                        //         Kette hat den Wert '2+2'
eval("var Kette = new String('2+2'); alert(Kette);");   // liefert nichts, da alert nichts liefert
                                                        //         Kette hat den Wert '2+2'

var StringLiteral  = "2 + 2";
eval(StringLiteral)                                      // liefert Zeiger auf numerische Variable
                                                        //         mit Wert 4
var Wert = eval("2+2");                                 // liefert Zeiger auf numerische Variable
                                                        //         mit Wert 4
var StringObjekt  = new String("2 + 2");
eval(StringObjekt)                                      // liefert Zeiger auf Kette "2 + 2"
```

Beispiel:

```
var w = 2;
var x = 39;
var y = "42";
var z = '"42"'
eval("w + x + 1");                                      // liefert Zeiger auf numerische Variable
                                                        //         mit Wert 42
eval(y);                                                // liefert Zeiger auf numerische Variable
                                                        //         mit Wert 42
                                                        // liefert nicht Zeiger auf y, da der String
                                                        //         "42" ausgewertet wird
eval(z);                                                // liefert Zeiger auf String-Variable
                                                        //         mit Wert '42'
                                                        // liefert nicht Zeiger auf z, da der String
                                                                //    '"42"' ausgewertet wird
```

Beispiel:

```
var Kette =         "var x=5;"
            +       "if (x == 5)"
            +       "{"
            +               "alert('Meldung aus eval\nx ist 5');" // \n ist Zeilenumbruch-Zeichen
            +               "x = 42;"
            +       "}"
            +       "else"
            +       "{"
            +               "x = 0;"
            +       "}";

var Kette1 = "alert('Meldung aus document.write\nx ist '+ eval(Kette));"
document.write(Kette1);
```

Beispiel für Ersatz von eval() durch die Eigenschaft .innerHTML (nur IE):

```
<SCRIPT>
var Kette = '<IMG STYLE="display: none;" ID="ID_IMG" ALT="Das ist ein Bild ">';

function Laden()
{
                ID_Div.innerHTML=Kette;                 // wird sofort geparst, also IMG-Tag ausgeführt
                                                        //         ( anstelle von eval()
                                                        //           bzw. document.write() )
                ID_IMG.src="";
                ID_IMG.style.display="block";

                ID_IMG.onerror= IMGAltTextAufFehlerMeldung;     // ohne  ()

}
```


```
function IMGAltTextAufFehlerMeldung ()
{
        ID_IMG.alt="Das Bild konnte nicht geladen werden.";
        return true;
}
</SCRIPT>
<INPUT TYPE=button VALUE="Klicke zum Bildladen" onclick="Laden()">
<DIV ID="ID_Div"></DIV>
```

4.1.2.8. isFinite()

ermittelt, ob numerischer Wert einer Instanz endlich ist

Syntax:

[var Wert =] isFinite(Zeiger);

Zeiger auf Instanz vom Number Script-Objekt oder Ausdrucksergebnis
auch Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
NaN

Wert true, so Wert der Instanz ist endlich
false, so Wert der Instanz ist unendlich
immer bei
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
NaN

4.1.2.9. isNaN()

ermittelt, ob Wert einer Instanz **nicht numerisch** is (**N**ot **a N**umber)

Syntax:

[var Wert =] isNaN(Zeiger);

Zeiger auf Instanz vom Number Script-Objekt oder Ausdrucksergebnis
auch Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
NaN

Wert true, so Wert der Instanz ist nicht numerisch
immer bei NaN
false, so Wert der Instanz ist numerisch
immer bei
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY

Beispiele:

```
var Kette ="10.23";
var Wert  =10.23;

var Ergebnis1=parseFloat(Kette);
var Ergebnis2=floatValue=parseFloat(Wert);

alert(isNaN(Ergebnis1);
alert(isNaN(Ergebnis2);
```

Hinweis: Methoden parseFloat und parseInt liefern NaN, wenn Parameter nicht numerisch ist
Bsp: var zahl=parseFloat("text"); isNaN() liefert true

4.1.2.10. Number()

konvertiert eine Instanz zu einem numerischen Wert (Number Script-Objekt-Wert)

Syntax:

[var Wert =] Number(Zeiger)

Zeiger 1 auf zu konvertierende Instanz
auch Date Objekt

Wert wenn Zeiger 1 ein Date Objekt ist, so Wert ist die Anzahl der Millisekunden seit dem 1.1.1970 0 Uhr Weltzeit (UTC), wobei Weltzeit genau GMT entspricht
wenn Datum vor dem 1.1.1970 0 Uhr Weltzeit so Wert negativ
NaN wenn Konvertierung nicht möglich ist

Beispiel:

```
var DatumJetzt = new Date();
alert (Number(DatumJetzt));
```


4.1.2.11. parseFloat()

String oder Literal parsen und nach Floating-point umwandeln
können enthalten
Vorzeichen + und -
Ziffern 0 bis 9
Dezimalkomma als Punkt
e oder E
Blanks (werden ignoriert)

falls andere Zeichen enthalten sind, gilt:
String bzw. Literal bis vor die erste fehlerhaften Stelle geparst und dann konvertiert

Erfolg der Konvertierung ist per Methode isNaN() zu prüfen
Syntax:

[var Wert =] parseFloat(Kette)

Kette String oder Literal

Wert Floating-point
NaN wenn String bzw. Literal bereits mit erstem Zeichen fehlerhaft

Beispiele: gültiges Literal

```
parseFloat("3.14")
parseFloat("314e-2")
parseFloat("0.0314E+2")
```

gültiger String

```
var x = "3.14"
parseFloat(x)
```

ungültiges Literal

```
parseFloat("FF2")
```

```
alert(parseFloat("17.50 DM"));
```

4.1.2.12. parseInt()

String oder Literal parsen und nach Integer umwandeln
können enthalten Vorzeichen + und -
Ziffern 0 bis 9, aber keine Vornull(en)
Blanks (werden ignoriert)
eventuelle Buchstaben A bis F

falls andere Zeichen enthalten sind, gilt:
String bzw. Literal bis vor die erste fehlerhaften Stelle geparst und dann konvertiert

Erfolg der Konvertierung ist per Methode isNaN() zu prüfen
Syntax:

[var Wert2 =] parseInt(Kette[, Wert1])

Kette String oder Literal
muss passend zu Wert1 kodiert sein
Hinweis: oktale Escape-Sequenzen sind in ab Javascript 1.5 im Netscape 6.x deprecated

Wert1 Integer
Zahlenbasis
10 für dezimal
8 für oktal
16 für hexadezimal (Wert1 kann Buchstaben A bis F enthalten)
Standard: 0

Wert2 Integer
NaN wenn String bzw. Literal bereits mit erstem Zeichen fehlerhaft
wenn Wert1 0 ist, so
wenn Kette beginnt mit "0x", so hexadezimale Konvertierung
"0" und nicht "0x", so oktale Konvertierung
nicht mit "0" oder "0x", so dezimale Konvertierung

Beispiel:
gültige Literale:

```
parseInt("F", 16)
parseInt("17", 8)
parseInt("15", 10)
```


```
parseInt(15.99, 10)
parseInt("FXX123", 16)
parseInt("1111", 2)
parseInt("15*3", 10)
parseInt("17")
parseInt("0x7", 16)
parseInt("0x7")
parseInt("0")
parseInt("0x11", 16)
parseInt("0x11", 0)
parseInt("0x11")
```

ungültige Literale:

```
parseInt("F")
parseInt("Hello", 8)
parseInt("0x7", 10)
parseInt("FFF", 10)
parseInt('002')
```

4.1.2.13. String()

konvertiert eine Instanz zu String
ist identisch mit Methode .toString()
Syntax:

[var Kette =] String(Zeiger)

Zeiger auch auf Date Objekt

Kette wenn Zeiger auf Date Objekt, so Datum als String geliefert (je nach Ländereinstellung des Windows)
Beispiel: Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983

4.1.2.14. toString()

Wert eines Objektes in einen String umwandeln
Methode ist nicht für jedes Objekt implementiert (aber für die meisten)
Syntax:

[var Kette =] object.toString([Wert])

Wert nur kodierbar bei numerischen Wert (Objekt ist Number Script-Objekt)
Basis des Wertes
z.B. 2 für dual (binär)
10 für dezimal
16 für hexadezimal

object Referenz auf den Wert eines Objektes
Javascript 1.1 liefert Objektname als Literal
Javascript 1.2 liefert Objektnotation als Literal
wenn Objekt ein Script-Objekt Array ist, dann Elemente in der Feldreihenfolge und mit Komma getrennt geliefert
wenn Objekt ein Script-Objekt Boolean ist, dann "true" bzw "false" geliefert (Kleinschreibung !)
wenn Objekt ein Script-Objekt Error ist, dann die Fehlermeldung geliefert (Objekt-Wert ist der Fehlercode)
wenn Objekt ein Script-Objekt Function ist, dann Quellcode der Funktion geliefert (inklusive Funktionskopf)
wenn Objekt ein Script-Objekt Object ist, so wird geliefert: "**[object** objectname**]**"
mit objectname als konkreter Bezeichner der Objektklasse bzw. des Objektes fettgedrucktes wird so geliefert wie angegeben
wenn der Bezug vor .toString() ein Wert laut ID-Attribut oder NAME-Attribut ist, dann wird die Objektklasse geliefert

Kette String

.troString() liefert 'NaN' wenn der Wert nicht nach String konvertierbar

Beispiel 1:

```
var Wert = 33,33;
alert(Wert.toString(2 ) + ' ' + Wert.toString(16) ) + ' ' + Wert.toString(10));
```

Beispiel 2:

```
<BUTTON ID="ID_Button" .... > .... </BUTTON>
ID_Button.toString() liefert "button"
```

Beispiel 3:

```
zahl=new Number("8376");
zahl.Number.toString(10)
```

liefert "8376" mit 10 als Basis der Zahlendarstellung

4.1.2.15. unescape()

ab Javascript 1.5 (Netscape 6.x) deprecated und zu ersetzen durch die Methoden decodeURI()
decodeURIComponent()

dekodiert einen mit der Methode escape() kodierten String (siehe escape())
Syntax:

[var Zeiger =] unescape(Kette)

Kette String aus escape()

Zeiger auf de-kodierten String

4.1.2.16. valueOf()

Konvertierung einer Objektinstanz in eine 32-Bit-Zahl also Zeigerwert z.B. zum Zweck des Vergleiches zweier Objektinstanzen per Zeigervergleich

Bsp: if (objekt_instanz_1.valueOf() == objekt_instanz_2.valueOf())

4.1.3. Standard-Eigenschaften und -Methoden aller Objekte in Microsoft JScript

Nachfolgend werden die gemeinsamen Eigenschaften und Methoden aller Objekte in JScript und der browserinternen Objekte beschrieben, also z.B. von

Objekt arguments
Objekt Array
Objekt Boolean
Objekt Date
Objekt Enumerator
Objekt Error
Objekt Function
Objekt Math
Objekt Number
Objekt Object (nicht Objekt object des Internet Explorer für das HTML-Tag OBJECT)
Objekt RegExp
Objekt String
Objekt var

wobei es Ausnahmen gibt: Z.B. unterstützt das Objekt Math nicht die Eigenschaft .constructor, da der Scripthersteller eine private Instanz von Math verhindern will.

JScript-Objekte sind alle in JScript implementierten Objekte, also **nicht** die browserinternen Objekte des Internet Explorer.

Das Objekt, dem diese objekt-übergreifenden Eigenschaften und Methoden gehört, ist das Script-Objekt Object, welches sozusagen den Prototyp aller anderen Objekte darstellt. Das Script-Objekt Object hat im Gegensatz zu anderen Script-Objekten wie Array oder Date keinen speziellen Datentyp implementiert. Die new Anweisung liefert also einen untypisierten Zeiger. Das Script-Objekt Object ist also für den Programmierer ein **symbolisches** Objekt, das als Objektklasse (Konstruktor) per new Anweisung für die Erzeugung eines einfachen Objektes benutzt werden kann, um es per Prototyping mit gewünschten Eigenschaften und Methoden zu erweitern und damit Datentypen beliebiger Art zu implementieren. Das Objekt ist also ideal für die Erzeugung eigener und freier Datenstrukturen anhand der Basis-Datentypen und Basis-Datenstrukturen.

Die gemeinsamen Eigenschaften und Methoden tauchen in der jeweiligen Objektbeschreibung **nicht** mehr auf (außer Script-Objekt Object).

Eine Instanz eines JScript-Objektes kann per Anweisung new erzeugt werden (siehe dort):

Achtung: Der Browser kann nur Objekte verarbeiten, die er kennt, z.B. ein Array Objekt, dessen Methoden und Eigenschaften dem Browser bekannt sind. Bei einem privaten Objekt müssen alle Eigenschaften und Methoden auf Script-Komponenten bestehen.

Jedes Objekt kann per Prototyping um Eigenschaften und Methoden erweitert werden, wenn es die Eigenschaft .prototype besitzt.

Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .

Nur für Objekte, die aus einem vordefiniertem Objekt abgeleitet sind, existiert die Eigenschaft .prototype .

Punktnotation zum Zeiger:
zeiger.**prototype**.eigenschaft
zeiger.**prototype**.methode

Erweiterung eines Script-Objektes per
bezeichner_script_objekt.**protoype**.eigenschaft
bezeichner_script_objekt.**protoype**.methode

Eigenschaften und Methoden müssen auf JScript-Elemente **basieren**, denn nur letztere kennt die Scriptmaschine des Browsers.

Eine Referenz auf eine Objekt-Instanz ist auch per Anweisungen this und with möglich (siehe dort).

Prinzipiell dürften andere Scripthersteller wegen der Skript-Kompatibilität die gleichen Script-Objekte und deren Eigenschaften und Methoden wie in JScript unterstützen.

Standard-Eigenschaften aller Objekte in JScript:

.constructor — Bezeichner einer JScript-Objektklasse (Objekttyp) oder eines privaten Konstruktors
Anwendung: Ermittlung der Objektklasse/Konstruktors eines abgeleiteten Objektes
Ableitung aus der Objektklasse
per Anweisung new mit der Objektklasse als Konstruktor benötigt (siehe dort)
nicht bei Script-Objekt Math möglich
Ableitung aus privatem Konstruktor
per Anweisung new, die den privaten Konstruktor verwendet
JScript-Objektklassen sind z.B.
Array
Boolean
Date
Function

Beispiel 1 für Ableitung aus einem JScript-Objekt:

```
var Kette = new String("Hi");
alert( (Kette.constructor == String));          // liefert "true"
alert( (Kette.constructor == "String"));        // liefert "false"
```

Beispiel 2 für Ableitung anhand privaten Konstruktors:

```
function TestFunktion()
{alert("Hallo");}

var ZeigerAufFunktion = new TestFunktion(); // bewirkt Ausführung von TestFunktion() also auch von alert()

// ZeigerAufFunktion();          // nicht möglich und bringt Fehlermeldung wegen fehlernder Instanz,
//                               //      da keine Ableitung vom JScript-Objekt Function
                                 // Kodierung ohne ()  bringt keinen Fehler, da als Variablendeklaration erkannt

alert(ZeigerAufFunktion.constructor == TestFunktion);          // true
alert(TestFunktion.constructor == TestFunktion);               // false
```

.propertyIsEnumerable — prüfen ob Stringwert in einem Objekt als Menge von String-Elementen enthalten ist
und ob das Objekt mit der Anweisung for in verarbeitet werden kann
Syntax:
[var Wert =] object.propertyIsEnumerable(zeiger_auf_stringvariable_oder_stringwert)

object — Zeiger auf Instanz der Menge

Wert — true, so enthalten **und per for in** verarbeitbar
false, nicht enthalten

Beispiel:

```
var Feld = new Array("Apfel", "Banane", "Zitrone");
alert( (Feld.propertyIsEnumerable("Apfel"));          // true
```

.prototype — Zeiger auf den Prototyp-Bereich im Objekt, der per Prototyping erweitert wird
Objekt ist entweder Script-Objekt oder bereits instanziertes Objekt:
innerhalb eines Konstruktors ist .prototype nicht zu kodieren
Prototyping eines Objektes verändert den Umfang der Standard-Methoden und -Eigenschaften zum Objekt
zur Laufzeit der Scriptmaschine.
Script-Objekte können nur erweitert werden.
Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
Nur für Objekte, die aus einem vordefiniertem Objekt abgeleitet sind, existiert die Eigenschaft .prototype .
siehe auch .isPrototypeOf() und .hasOwnProperty()
Syntax:
object.prototype.eigenschaft
object.prototype.methode

object — Bezeichner des Objektes (nicht in " " bzw. ' ' kodieren)
Zeiger auf instanziertes Objekt

Beispiel für Erweiterung des Script-Objektes Array, das die Eigenschaft .prototype besitzt:

```
function MaximumErmitteln( )
{
	var max = this[0];	// this referenziert die Objekt-Instanz, in dem die Methode vorhanden ist,
				//		also Variable Feld

	for (var i = 1; i < this.length; i++)
	{
		if (max < this[i])
		{max = this[i];}
	}

	return max;
}

// neue Methode per Prototyping hinzufügen zum JScript-Objekt Array
Array.prototype.NeueArrayMethode = MaximumErmitteln;		// ohne () kodieren !

// Feld vom Array-Typ erzeugen mit der Methode .NeueArrayMethode()
var Feld = new Array(1, 2, 3, 4, 5, 6);		// 6 numerische Elemente

// neue Methode des JScript-Objektes Array aufrufen
alert(Feld.NeueArrayMethode());
```

Beispiel für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
<HTML>
<SCRIPT  LANGUAGE="JavaScript">
<!--
 function datenstruktur_ausgeben()
{
	with (document)
	{	write(person.vorname + " " + person.nachname + "<BR>");
		write(person.strasse	+ " " + person.nummer	+ "<BR>");
		write(person.plz	+ " " + person.ort		+ "<BR>");
		//		Erika Mustermann
		//		Musterstrasse 1
		//		.....
	}

	for (i in person)
	{	document.write(i + ": " + person[i] + "<BR>");
		//		vorname: Erika
		//		nachname: Mustermann
		//		.....
	}
}

function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
{
	this.vorname = vorname;
	this.nachname = nachname;
	this.strasse = strasse;
	this.nummer = nummer;
	this.plz = plz;
	this.ort = ort;
	this.methode = methode;
}

person = new datenstruktur_erzeugen
			(	"Erika",			// Vorname
				"Mustermann",			// Nachnahme
				"Musterstrasse",		// Strasse
				"1",				// Nummer
				"10000",			// PLZ
				"Musterstadt",			// Ort
				datenstruktur_ausgeben		// ohne () kodieren
			);

// alternativ auch kodierbar:
```


```
// var person = {      vorname:"Erika",                  // Vorname
//                     nachname:"Mustermann",            // Nachnahme
//                     strasse:"Musterstrasse",          // Strasse
//                     nummer:"1",                       // Nummer
//                     plz:"10000",                      // PLZ
//                     ort:"Musterstadt"                 // Ort
//                     methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
//               };

// Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !

alert(person.vorname + "\n" + person.methode);        // person.methode ohne () kodieren !
// -->
</SCRIPT>
</HTML>
```

Standard-Methoden aller Objekte in JScript:

Boolean() konvertiert einen Wert nach Boolean

Syntax:

[var Zeiger =] Boolean(Wert)

Wert true oder false oder beliebiger Wert
wenn 0, -0, +0, null, false, NaN, undefined oder Leerkette ""
so wird false verwendet
sonst wird true verwendet
z.B. "false" wird als true verwendet

decodeURI() dekodiert einen Uniform Ressource Identifier (URI), der mit der Methode encodeURI() erzeugt wurde
ersetzt die Methode unescape() welche deprecated ist
ab IE 5.5
Syntax:

[var Zeiger =] decodeURI(Kette)

Kette encoded Uniform Resource Identifier, der mit der Methode encodeURI () erzeugt wurde

Zeiger auf dekodierten String

Beispiel:

```
alert(decodeURI("My%20phone%20 #%20is%20123-456-7890"));
                                        // erzeugt "My phone # is 123-456-7890"
```

decodeURIComponent() dekodiert eine Komponente einer Uniform Ressource Identifier (URI), der mit der Methode encodeURI() erzeugt wurde
ab IE 5.5
Syntax:

[var Zeiger =] decodeURIComponent(Kette)

Kette encoded Uniform Ressource Identifier, der mit der Methode encodeURIComponent() erzeugt wurde

Zeiger auf dekodierten String

encodeURI() kodiert einen String als kompletten Uniform Ressource Identifier (URI):
Ersatz von Zeichen durch Escape-Sequenzen (UTF-8 Zeichen) der Form %xx
Es werden alle Zeichen kodiert, also durch Escape-Sequenzen ersetzt, außer folgende Zeichen

. , / ? : @ & = + $ - _ . ! ~ * ' ( ) #
Buchstaben
Ziffern

ersetzt die Methode escape() welche deprecated ist
ab IE 5.5
Syntax:

[var Zeiger =] encodeURI(Kette)

Kette URI, der kodiert wird

Zeiger auf kodierten kompletten Uniform Ressource Identifier (URI)

Beispiel:

```
alert(encodeURI("My phone # is 123-456-7890"));
                                        // erzeugt "My%20phone%20 #%20is%20123-456-7890"
```


encodeURIComponent() kodiert einen Teil-String aus einem kompletten Uniform Ressource Identifier (URI):
Ersatz von Zeichen durch Escape-Sequenzen (UTF-8 Zeichen) der Form %xx
Es werden alle Zeichen kodiert, also durch Escape-Sequenzen ersetzt, außer folgende Zeichen

. , / ? : @ & = + $ - _ . ! ~ * ' ( ) #
Buchstaben
Ziffern

ab IE 5.5
Syntax:
[var Zeiger =] encodeURIComponent (Kette)

Kette Teil-String des URI

Zeiger auf kodierten Teil-String des Uniform Ressource Identifier (URI)

escape() kodiert einen String oder ein Literal in das Unicode-Format
ab Javascript 1.5 (Netscape 6.x) deprecated und zu ersetzen durch die Methoden
encodeURI() und encodeURIComponent()
Ersatz von Zeichen durch Escape-Sequenzen (UTF-8 Zeichen) der Form %xx
Es werden alle Zeichen kodiert, also durch Escape-Sequenzen ersetzt, außer folgende Zeichen

. , / ? : @ & = + $ - _ . ! ~ * ' ( ) #
Buchstaben
Ziffern

URI (Uniform Resource Identifiers) werden nicht kodiert
UTF-8 Zeichen: Teil des Unicode (0 bis 255)
Syntax:
[var Zeiger =] escape(Kette)

Kette String

Zeiger auf kodierten String

Beispiel für Einbindung einer Suchmaschine:

```
var markierter_text=document.selection.createRange().text;
        // oder var markierter_text=prompt('Suchbegriff: ');
var suchmaschinen_url='http:// .....';
var suchmaschinen_parameter='.....';

if (markierter_text)
{location.HREF=suchmaschinen_url + suchmaschinen_parameter + escape(markierter_text);}
else
{location.HREF=suchmaschine_url; }
```

Beispiel für altavista: suchmaschinen_url 'http://altavista.de/'
suchmaschinen_parameter 'cgi-bin/query?pg=q&what=web&q='

eval() parst einen String aus JavaScript-Anweisungen (kein HTML-Code) und führt diese sofort aus
JavaScript-Anweisungen
dürfen keine Referenz auf sich selbst haben (sonst endlose Rekursion !)
können Referenzen auf existierende Objekte (Variablen) und deren Objektmethoden wie Objekteigenschaften enthalten z.B. für Wertzuweisung etc.
Objekte (Variablen) erzeugen und instanzieren
ab JavaScript 1.5 auch eval als Anweisung enthalten (Achtung: endlose Rekursionen vermeiden !)
Syntax:
[var Zeiger =] eval(Kette)

Kette String mit JavaScript-Anweisungen

Zeiger auf Ausdrucksergebnis
nur erzeugt, wenn JavaScript-Anweisungen einen Ausdruck als letzten Teil oder nur einen Ausdruck in der Kette enthalten
Es ist zu empfehlen, anstelle des Zeigers die Wertzuweisung auf eine Variable direkt innerhalb der JavaScript-Anweisungen zu realisieren.

Beispiele:

```
var Kette = eval("new String('2+2')");          // liefert Zeiger auf den String '2+2'
eval("var Kette = new String('2+2')");          // liefert nichts
                                                //         Kette hat den Wert '2+2'
```


```
eval("var Kette = new String('2+2'); alert(Kette);");   // liefert nichts, da alert nichts liefert
                                                        //          Kette hat den Wert '2+2'

var StringLiteral  = "2 + 2";
eval(StringLiteral)                                      // liefert Zeiger auf numerische Variable
                                                        //          mit Wert 4
var Wert = eval("2+2");                                 // liefert Zeiger auf numerische Variable
                                                        //          mit Wert 4
var StringObjekt  = new String("2 + 2");
eval(StringObjekt)                                      // liefert Zeiger auf Kette "2 + 2"
```

Beispiel:

```
var w = 2;
var x = 39;
var y = "42";
var z = "'42'"
eval("w + x + 1");                                      // liefert Zeiger auf numerische Variable
                                                        //          mit Wert 42
eval(y);                                                // liefert Zeiger auf numerische Variable
                                                        //          mit Wert 42
                                                        // liefert nicht Zeiger auf y, da der String
                                                        //          "42" ausgewertet wird
eval(z);                                                // liefert Zeiger auf String-Variable
                                                        //          mit Wert '42'
                                                        // liefert nicht Zeiger auf z, da der String
                                                                //    "'42'" ausgewertet wird
```

Beispiel:

```
var Kette =             "var x=5;"
            +           "if (x == 5)"
            +           "{"
            +                       "alert('Meldung aus eval\nx ist 5');" // \n ist Zeilenumbruch-Zeichen
            +                       "x = 42;"
            +           "}"
            +           "else"
            +           "{"
            +                       "x = 0;"
            +           "}";

var Kette1 = "alert('Meldung aus document.write\nx ist '+ eval(Kette));"
document.write(Kette1);
```

Beispiel für Ersatz von eval() durch die Eigenschaft .innerHTML (nur IE):

```
<SCRIPT>
var Kette = '<IMG STYLE="display: none;" ID="ID_IMG" ALT="Das ist ein Bild ">';

function Laden()
{
                        ID_Div.innerHTML=Kette;                 // wird sofort geparst, also IMG-Tag ausgeführt
                                                                //          ( anstelle von eval()
                                                                //            bzw. document.write() )
                        ID_IMG.src="";
                        ID_IMG.style.display="block";

                        ID_IMG.onerror= IMGAltTextAufFehlerMeldung;     // ohne  ()
}

function IMGAltTextAufFehlerMeldung ()
{
                        ID_IMG.alt="Das Bild konnte nicht geladen werden.";
                        return true;
}
</SCRIPT>
<INPUT TYPE=button VALUE="Klicke zum Bildladen" onclick="Laden()">
<DIV ID="ID_Div"></DIV>
```

Beispiel:

```
if (VarUserAgent != '')
{
            for ( var i=0; i < 10; i++)
            {
```


```
                                        eval(           'if (VarUserAgent.indexOf("'  + i + '.") != -1)'
                                                      + '{VarBrowser_Version_Haupt = ' + i + ';}'
                                            );
                                }
                        }
```

Beispiel:

```
                        function DatumHolen(Zeiger)
                        {
                                        var Kette = "Heute ist: ";
                                        Kette += Zeiger.getDate()          + "/";
                                        Kette += (Zeiger.getMonth() + 1) + "/";
                                        Kette += Zeiger.getYear();

                                        return(Kette);
                        }

                        var Kette="Date";

                        eval(   "var Jetzt = new " + Kette + "();" // zur Laufzeit erzeugen
                                "alert(DatumHolen(Jetzt));"       // Parameter ist zur Laufzeit bekannt
                            );
```

.hasOwnProperty()

prüfen ob zum Objekt eine Eigenschaft vorhanden ist, jedoch leider **nicht** aus Prototyping einer per new erzeugten Instanz des JScript-Objektes

Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .

siehe auch .prototype

Syntax:

[var Wert =] **object.prototype**.hasOwnProperty(kette)

kette String
Bezeichner der Eigenschaft

objekt auf per new erzeugte Instanz oder Bezeichner eines JScript-Objektes

Wert true, so Eigenschaft vorhanden
false, so Eigenschaft nicht vorhanden

Beispiel:

```
                        function MaximumErmitteln( )
                        {
                                var max = this[0];      // this referenziert die Objekt-Instanz, in dem die Methode vorhanden ist,
                                                        //              also Variable Feld

                                for (var i = 1; i < this.length; i++)
                                {
                                        if (max < this[i])
                                        {max = this[i];}
                                }

                                return max;
                        }

                        // neue Methode per Prototyping hinzufügen zum JScript-Objekt Array
                        Array.prototype.NeueArrayMethode = MaximumErmitteln;          // ohne () kodieren !

                        // Feld vom Array-Typ erzeugen mit der Methode .NeueArrayMethode()
                        var Feld = new Array(1, 2, 3, 4, 5, 6);                                 // 6 numerische Elemente
                                                                                                // Es wird die Eigenschaft .prototype erzeugt
                        alert( (Feld.prototype.hasOwnProperty("NeueArrayMethode"));    // leider false
                        alert( (Array.prototype.hasOwnProperty("NeueArrayMethode "));// true
```

isFinite()

ermittelt, ob Wert einer Instanz numerisch endlich ist

Syntax:

[var Wert =] isFinite(Zeiger);

Zeiger auf Instanz vom Number-Objekt oder Ausdrucksergebnis
auch Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
Number.NaN

Wert true, so Wert der Instanz ist endlich
false, so Wert der Instanz ist unendlich
immer bei
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
Number.NaN

isNaN() ermittelt, ob Wert einer Instanz nicht numerisch ist
Syntax:
[var Wert =] isNaN(Zeiger);

Zeiger auf Instanz vom Number-Objekt oder Ausdrucksergebnis
auch Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
Number.NaN

Wert true, so Wert der Instanz ist nicht numerisch
immer bei Number.NaN
false, so Wert der Instanz ist numerisch
immer bei
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY

Beispiele:

```
var Kette ="10.23";
var Wert  =10.23;

var Ergebnis1=parseFloat(Kette);
var Ergebnis2=floatValue=parseFloat(Wert);

alert(isNaN(Ergebnis1);
alert(isNaN(Ergebnis2);
```

Hinweis: Methoden parseFloat und parseInt liefern NaN, wenn Parameter nicht numerisch ist
Bsp: var zahl=parseFloat("text"); ifNaN() liefert true

.isPrototypeOf() prüfen ob Objekt per new erzeugt wurde, also eine Instanz eines anderen JScript-Objektes ist
Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
siehe auch .prototype
Syntax:
[var Wert =] **object.prototype**.isPrototypeOf(zeiger)

zeiger auf per new erzeugte Instanz

object Zeiger auf JScript-Objekt, dessen Bezeichner als
Konstruktur in der new Anweisung verwendet wurde

Wert true, so Objekt ist Instanz eines JScript-Objektes
false, so keine Instanz eines JScript-Objektes
oder Zeiger auf per new erzeugte Instanz ist ungültig

Beispiel:

```
var Variable = new RegExp();
alert( (RegExp.prototype.isPrototypeOf(Variable));                    // true.
```

Number() konvertiert eine Instanz zu einem numerischen Wert (Number-Objekt-Wert)
Syntax:
[var Wert =] Number(Zeiger)

Zeiger 1 auf zu konvertierende Instanz
auch Date-Objekt

Wert wenn Zeiger 1 ein Date-Objekt ist, so Wert ist die Anzahl der
Millisekunden seit dem 1.1.1970 0 Uhr Weltzeit
(UTC), wobei Weltzeit genau GMT entspricht
wenn Datum vor dem 1.1.1970 0 Uhr Weltzeit so Wert negativ
NaN wenn Konvertierung nicht möglich ist

Beispiel:

```
var DatumJetzt = new Date();
alert (Number(DatumJetzt));
```


parseFloat() String oder Literal parsen und nach Floating-point umwandeln
können enthalten
Vorzeichen + und -
Ziffern 0 bis 9
Dezimalkomma als Punkt
e oder E
Blanks (werden ignoriert)

falls andere Zeichen enthalten sind, gilt:
String bzw. Literal bis vor die erste fehlerhaften Stelle geparst und dann konvertiert

Erfolg der Konvertierung ist per Methode isNaN() zu prüfen
Syntax:
[var Wert =] parseFloat(Kette)

Kette String oder Literal

Wert Floating-point
NaN wenn String bzw. Literal bereits mit erstem Zeichen fehlerhaft

Beispiele:

gültiges Literal
```
parseFloat("3.14")
parseFloat("314e-2")
parseFloat("0.0314E+2")
```

gültiger String
```
var x = "3.14"
parseFloat(x)
```

ungültiges Literal
```
parseFloat("FF2")
```

```
alert(parseFloat("17.50 DM"));
```

parseInt() String oder Literal parsen und nach Integer umwandeln
können enthalten
Vorzeichen + und -
Ziffern 0 bis 9, aber keine Vornull(en)
Blanks (werden ignoriert)
eventuelle Buchstaben A bis F

falls andere Zeichen enthalten sind, gilt:
String bzw. Literal bis vor die erste fehlerhaften Stelle geparst und dann konvertiert

Erfolg der Konvertierung ist per Methode isNaN() zu prüfen
Syntax:
[var Wert2 =] parseInt(Kette[, Wert1])

Kette String oder Literal
muss passend zu Wert1 kodiert sein
Hinweis: oktale Escape-Sequenzen sind in ab Javascript 1.5 im Netscape 6.x deprecated

Wert1 Integer
Zahlenbasis
10 für dezimal
8 für oktal
16 für hexadezimal
(Wert1 kann Buchstaben A bis F enthalten)
Standard: 0

Wert2 Integer
NaN wenn String bzw. Literal bereits mit erstem Zeichen fehlerhaft
wenn Wert1 0 ist, so
wenn Kette beginnt mit
"0x", so hexadezimale Konvertierung
"0" und nicht "0x", so oktale Konvertierung
nicht mit "0" oder "0x", so dezimale Konvertierung

Beispiel:

gültige Literale:

```
parseInt("F", 16)
parseInt("17", 8)
parseInt("15", 10)
parseInt(15.99, 10)
parseInt("FXX123", 16)
parseInt("1111", 2)
parseInt("15*3", 10)
parseInt("17")
parseInt("0x7", 16)
parseInt("0x7")
parseInt("0")
parseInt("0x11", 16)
parseInt("0x11", 0)
parseInt("0x11")
```

ungültige Literale:

```
parseInt("F")
parseInt("Hello", 8)
parseInt("0x7", 10)
parseInt("FFF", 10)
parseInt('002')
```

String() konvertiert eine Instanz zu String
ist identisch mit Methode .toString()
Syntax:

```
[ var Kette = ] String(Zeiger)
```

Zeiger auch auf Date-Objekt

Kette wenn Zeiger auf Date-Objekt, so Datum als String geliefert (je nach Ländereinstellung des Windows)
Beispiel: Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983

.toLocaleString() Wert eines Objektes in System-lokale Einstellungen umwandeln
System-lokale Einstellung z.B. Ländereinstellung, Uhrzeitformat
Konvertierung: Analog wie z.B. bei Exel, das die lokalen Einstellungen ausliest.
nur anwenden für Anzeige des konvertierten Wertes:
Berechnungen mit dem Wert immer unkonvertiert vollziehen, da sämtliche Berechnungsfunktionen **nur** das interne, also unkonvertierte Format kennen (**nicht** analog zu Exel)
Wenn das Objekt ein Script-Objekt Array ist, also Feldelemente in lokale Einstellungen konvertiert werden sollen, dann wird ein String geliefert, der die Feldelemente in der Reihenfolge im Feld und diese getrennt durch **dasjenige** Trenner-Zeichen laut **lokale** Einstellungen enthält.
Wenn das Objekt eine Script-Objekt Date ist, so wird der Wert von dem Objekt in den lokalen Datumeinstellungen geliefert. Dabei sind nur Jahreszahlen von 1601 bis 9999 zulässig. Andere Jahresangaben werden nicht konvertiert.
Wenn das Objekt ein Script-Objekt Number ist, so werden die lokalen Einstellungen zum Zahlenformat geliefert.
Wenn das Objekt ein Script-Objekt Object ist, so werden nur dann lokale Einstellungen berücksichtigt, insoweit das Objekt diese berücksichtigen kann. Ein String wird immer geliefert.
Syntax:

```
[ var Kette = ] object. toLocaleString()
```

object Referenz auf den Wert eines Objektes

Kette String

Beispiel für Konvertierung eines Feld mit Gleitkomma-Werten:

```
var Feld = new Array(6);
var Wert = 3,201,300.20;        // in JScript ist      das Dezimalkomma der Punkt
                                //                     der Tausendertrenner das Komma

// Feld füllen
for( var i = 0; i < 7; i++)
{
        Wert +1 = 10;
        Feld [i] = Wert;
}
```


```
alert(Feld.toLocaleString());
```

Beispiel für Konvertierung eines Datums:

```
var Jetzt = new Date();
alert(Jetzt.toLocaleString());
```

.toString()

Wert eines Objektes in einen String umwandeln
wenn Objekt ein Script-Objekt Array ist, dann Elemente in der Feldreihenfolge und mit Komma getrennt geliefert
wenn Objekt ein Script-Objekt Booelan ist, dann "true" bzw "false" geliefert (Kleinschreibung !)
wenn Objekt ein Script-Objekt Error ist, dann die Fehlermeldung geliefert (Objekt-Wert ist der Fehlercode)
wenn Objekt ein Script-Objekt Function ist, dann Quellcode der Funktion geliefert (inklusive Funktionskopf)
wenn Objekt ein Script-Objekt Object ist, so wird geliefert:
"**[object** objectname**]**"
mit objectname als konkreter Bezeichner der Objektklasse bzw. des Objektes fettgedrucktes wird so geliefert wie angegeben
wenn der Bezug vor .toString() ein Wert laut ID-Attribut oder NAME-Attribut ist, dann wird die Objektklasse geliefert
Syntax:

```
[ var Kette = ] object.toString([Wert])
```

Wert — nur kodierbar bei numerischen Wert (Objekt ist Number-Objekt)
Basis des Wertes
z.B. 2 für dual (binär)
10 für dezimal
16 für hexadezimal

object — Referenz auf den Wert eines Objektes

Kette — String

Beispiel 1:

```
var Wert = 33,33;
alert(Wert.toString(2 ) + ' ' + Wert.toString(16) ) + ' ' + Wert.toString(10));
```

Beispiel 2:

```
<BUTTON ID="ID_Button" .... > .... </BUTTON>
ID_Button.toString() liefert "button"
```

unescape

ab Javascript 1.5 (Netscape 6.x) deprecated und zu ersetzen durch die Methoden decodeURI() und decodeURIComponent()
dekodiert einen mit der Methode escape() kodierten String (siehe escape())
Syntax:

```
[ var Zeiger = ] unescape(Kette)
```

Kette — String aus escape()

Zeiger — auf dekodierten String

.valueOf()

Wert eines JScript-Objektes bzw. Instanz eines JScript-Objektes ermitteln
nicht bei Script-Objekt Math und Script-Objekt Error (auch nicht bei Instanzen dieser Objekte)
Wert ist im Datentyp des Objektes, aber:
wenn Objekt ein Script-Objekt Array ist, dann Elemente in der Feldreihenfolge und mit Komma getrennt geliefert (identisch in der Wirkung mit .toString() und der Array-Methode .join())
wenn Objekt ein Script-Objekt Boolean ist, dann true bzw false geliefert (kein String !)
wenn Objekt ein Script-Objekt Date ist, dann Anzahl der Millisekunden seit dem 1.1.1970 0 Uhr geliefert
wenn Objekt ein Script-Objekt Function ist, dann Zeiger auf Funktion geliefert
wenn Objekt ein Script-Objekt Number ist, dann numerischen Wert geliefert
wenn Objekt ein Script-Objekt Object ist, so Zeiger geliefert
siehe auch .toLocaleString() und .toString()
Syntax:

```
[ var Wert = ] object.valueOf( );
```

Wert — typengerecht zum JScript-Objekt

object — **nur** JScript-Objekt bzw. Instanz davon

4.2. In Javascript vordefinierte Objekte (Script-Objekte) (Objektklassen oder Objekttypen)

Es können alle Objekte aus Javascript/JScript mit deren Eigenschaften und Methoden in sinnvoller Kombination mit den zum Browser vordefinierten Objekten verwendet werden. Die objekt-übergreifenden, also objekt-unabhängigen Methoden, sind alle nutzbar aber z.T. in den zum Browser vordefinierten Objekten abgewandelt implementiert.

Die in Script definierten Objekte sind z.B.

Objekt arguments
Objekt Array
Objekt Boolean
Objekt Date
Objekt Enumerator
Objekt Error
Objekt Function
Objekt Math
Objekt Number
Objekt Object (nicht Objekt obejct des Internet Explorer für das HTML-Tag OBJECT)
Objekt RegExp
Objekt String
Objekt var

Die Bezeichner dieser Objekte werden auch **Objekttypen oder Objektklassen** benannt. Letztere Begriffe finden **nur** in der Ableitung eines vordefinierten Objektes Anwendung: Von den vordefinierten Objekten sind Instanzen per Anweisung new erzeugbar (siehe dort), wobei als Konstruktor der Bezeichner des Objektes verwendet wird. Es gibt Script-Objekte, die sich nicht ableiten lassen (z.B. Math Script-Objekt), da der Hersteller verhindern will, dass Veränderungen am Objekt vorgenommen werden.

Man beachte: Es können auch rein private Objekte erzeugt werden, die nicht von einem vordefiniertem Objekt abstammen.

Prototyping einer Ableitung ist nur durch die Verwendung der Eigenschaft .prototype möglich, allerdings das auch nur dann, wenn es sich um Objekte handelt, die aus den **vordefinierten** Script-Objekten per new-Anweisung abgeleitet wurden, denn **nur dann** wird die Eigenschaft .prototype überhaupt erzeugt.

Die Art und der Umfang der Implementation der vordefinierten Objekte in die Scriptmaschine hängt vom Hersteller ab.

4.2.1. arguments Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
nur verfügbar während Abarbeitung der zugehörigen Funktion und von dieser instanziert
immer funktionslokal

hat Feldcharakter, das alle Argumente der Funktion referenziert:
jedes Argument muss mit Parameter versorgt werden
Anzahl der Elemente in Argumenten- und Parameterliste muss identisch sein
Alle Parameter werden in der Listenfolge von links nach rechts in arguments abgelegt
mit Index ab 0 und aufsteigend
Anzahl der Parameter laut arguments.length

Bsp: function **test**(arg1, arg2, arg3) {.....}
.....
test(10, 20, 30);

10 entspricht **test**.arguments[0] für arg1
20 entspricht **test**.arguments[1] für arg2
30 entspricht **test**.arguments[2] für arg3

selbst Eigenschaft der Funktion ist

siehe Script-Objekt Function und Anweisung function

Syntax:

[var Zeiger =] [funktions_bezeichner.]arguments
[var ZeigerAufElement =] [funktions_bezeichner.]arguments **[**Index**]**
[var ZeigerAufElement =] [funktions_bezeichner.]arguments.0 bis .n

funktions_bezeichner nur dann zu kodieren, wenn arguments innerhalb einer Funktionsverschachtelung verwendet wird, um deren Argumentenlisten zu trennen
Funktionsbezeichner laut Deklaration und Aufruf der Funktion

Index Integer, ab 0
muss in **[ ]** kodiert sein

.0 bis .n Eigenschaften von arguments

Zeiger ist null, wenn keine Argumente vorhanden

ZeigerAufElement ist null, wenn Feldelement nicht vorhanden

Beispiel für Einlesen der Werte der Argumente aus dem Funktionsaufruf:

```
var GlobalesFeld = new Array();

function FunktionsBezeichner()
{

        // Argumenteliste der Funktion lokal zur Funktion referenzieren
        var ArgumentenListeAlsFeld = FunktionsBezeichner.arguments;

        // Anzahl der Argumente
        var ArgumenteAnzahl = ArgumentenListeAlsFeld.length;

        if ( ArgumenteAnzahl > 0 )
        {
                // Arrgumentenliste auslesen
                for (var i = 0; i < ArgumenteAnzahl; i++)
                {GlobalesFeld[i] = ArgumentenListeAlsFeld[i];}
        }

        // hier die weiteren Anweisungen der Funktion
}
```

Eigenschaften:

.0 bis .n Wert des Argumentes mit Index 0 ... bzw. n im Script-Objekt arguments als Eigenschaft eines Funktionsobjektes (siehe Script-Objekt Function und Anweisung function)

.0 entspricht auch funktions_bezeichner.arguments[0]

.n entspricht auch funktions_bezeichner.arguments[n]

n von 0 bis beliebig ganzzahlig-numerische Ziffernfolge ohne Vornull
ist die Position ab 0 innerhalb der Argumentenliste von links nach rechts
Parameterliste von links nach rechts

Hinweis: Argumentenliste und Parameterliste mit gleicher Elementeanzahl, denn **jedes Argument** muss mit Wert versorgt werden

Beispiel für Einlesen der Werte der Argumente aus dem Funktionsaufruf:

```
var GlobalesFeld = new Array();

function FunktionsBezeichner()
{

        // Argumenteliste der Funktion lokal zur Funktion referenzieren
        var ArgumentenListeAlsFeld = FunktionsBezeichner.arguments;

        // Anzahl der Argumente
        var ArgumenteAnzahl = ArgumentenListeAlsFeld.length;

        if ( ArgumenteAnzahl > 0 )
        {
                // Arrgumentenliste auslesen
                GlobalesFeld[0] = ArgumentenListeAlsFeld.0;
                GlobalesFeld[1] = ArgumentenListeAlsFeld.1;
        }

        // hier die weiteren Anweisungen der Funktion
}
```

.callee Zeiger auf den Funktionsrumpf
z.B. verwenden, wenn
Funktion ohne Funktionsbezeichner erzeugt wurde
für Kodierung einer Rekursion ohne den Funktionsbezeichner aber **mit Argumentenliste**
(falls Argumentenliste vorhanden ist)

Beispiel:

```
function Test(n)
{
        if (n <= 0)
```


```
                {return 1;}
                 else
                {return (n * arguments.callee(n - 1));}        // Rekursion per arguments.callee(n - 1)
        }

        alert(Test(3));
```

.length Anzahl der Argumente im Script-Objekt arguments als Eigenschaft einesFunktionsobjektes
Wert ist identisch mit dem Wert der Eigenschaft .length des Scrip-Objektes Function

Beispiel für Einlesen der Werte der Argumente aus dem Funktionsaufruf:

```
        var GlobalesFeld = new Array();

        function FunktionsBezeichner()
        {

                // Argumenteliste der Funktion lokal zur Funktion referenzieren
                var ArgumentenListeAlsFeld = FunktionsBezeichner.arguments;

                // Anzahl der Argumente
                var ArgumenteAnzahl = ArgumentenListeAlsFeld.length;

                if ( ArgumenteAnzahl > 0 )
                {
                        // Arrgumentenliste auslesen
                        for (var i = 0; i < ArgumenteAnzahl; i++)
                        {GlobalesFeld[i] = ArgumentenListeAlsFeld[i];}
                }

                // hier die weiteren Anweisungen der Funktion
        }
```

Methoden:
keine

4.2.2. Array Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
Sammlung von Werten mit beliebigen Datentypen und Referenzierung der Werte per Integer-Index als Position eines Wertes in der Sammlung
frei gestaltbar
siehe auch Basis-Datenstruktur array

4.2.2.1. Array Script-Objekt mit Elemente eines beliebigen Datentyps

Syntax:

Syntax für Felderzeugung: Die new-Anweisung kann zwar entfallen, sollte aber kodiert werden.

```
[ var Zeiger = ] new Array()                // leer, also 0 Elemente
                                            // ermöglicht mehrdimensionales Feld

[ var Zeiger = ]  new Array(wert)

        wert    Anzahl der Feldelemente
                Integer ab 0

[ var Zeiger = ]  new Array(werte_liste)
[ var Zeiger = ]  new Array[werte_liste]    // ab Javascript 1.2

        werte_liste     initialisiert Feldelemente ab Index 0 aufsteigend mit den Werten laut Liste von links
                        nach rechts
                kommagetrennte Werte  z.B.
                        Literal
                        Wert laut Referenz
                        Wert aus Funktionsaufruf
                        Wert als Ergebnis eines Ausdrucks
                [] ist zu kodieren !
```

Syntax für Feldfüllung ohne new-Anweisung aber anhand Basis-Datenstruktur array:

```
Zeiger =  [ [ liste_1] [..... , [ liste_n] ]        für mehrdimensionales Feld
Zeiger =  [ liste ]                                 für eindimensionales Feld

        Zeiger  instanziertes Objekt der Objektklasse Array, also Ableitung vom Script-Objekt Array
```


per var Zeiger = new Array()

[ ] bedeutet hier **nicht** optional !!

[liste_1] bis [liste_n]

Liste aus kommagetrennten Elementen
Liste_x stellt ein symbolisches Feld da (x ist 1 ..... n)

liste Liste aus kommagetrennten Elementen

Listenelement: kann Ausdruck sein, der Wert liefern muss
kann entfallen, aber das trennende Komma muss kodiert werden
Folge der Listenelemente entspricht Initialisierungsfolge des Feldes
pro Komma eine automatische Indexerhöhung
wenn Element nicht kodiert, so wird das Feldelement
zum Index nicht initialisert ---> Feldelement
hat keinen Datentyp

Syntax für Feldfüllung aus Literalen:
siehe weiter unten

Zugriff:

Zeiger**[**Index**]**.eigenschaft
Zeiger**[**Index**]**.methode()
Zeiger**[**Index**]** = wert

Index Integer ab 0
oder String (in " " bzw. ' ' kodieren)
muss in **[ ]** kodiert sein

Wert füllt Feldelement und bestimmt den Datentyp des Feldelementes: Feldelemente können verschiedentypig sein

Beispiele:

Beispiele für Füllen eines instanzierten und leeren Feldes mit Namen "Feld":

Feld = [1,2,3]; Element an Index 0 hat Wert 1
Element an Index 1 hat Wert 2
Element an Index 2 hat Wert 3

Anzahl der Elemente 3

Feld = [1,,,,5]; Initialisiert wird **nur** mit Index 0 auf Wert 1
mit Index 4 auf Wert 5
Anzahl der Elemente 2

```
Feld = [ ["Name", "Tom", "Tim", "Teo"], ["Alter", 6, 5, 4] ];


Feld[0} = 10;
Feld[99]=100;          //  Feldlänge ist 2 !!!
```

Beispiel für Erzeugung eines 2-dimensionalen Feldes:

```
function erzeuge_zwei_dim_feld(anzahl_spalten, anzahl_zeilen)
{
        // Spaltenfeld erzeugen
        this.spalten_feld= new Array(anzahl_spalten);

        // Zeilenfeld pro Spalte  erzeugen
        for ( var spalte=0; spalte < anzahl_spalten; spalte++)
        { this[spalte] = new Array(anzahl_zeilen);}  //Zeilen-Felder
}

var zwei_dim_tabelle= new erzeuge_zwei_dim_feld(10,7); // 10 Spalten zu je 7 Zeilen
....
zwei_dim_tabelle[10,7]=22;   // Spalte 10: 7Zeile: mit  22 belegen

.......
delete zwei_dim_tabelle[10,7];  // löscht in Spalte 10 die 7. Zeile
```

Beispiel für Erzeugung eines 2-dimensionalen Feldes:

```
var Feld = new Array();
var AnzahlUnterfelder = 4;
var AnzahlElementeImUnterfeld = 4;

// Unterfelder erzeugen
for (       var UnterfeldZahler=0;
            UnterfeldZahler < AnzahlUnterfelder;
            UnterfeldZahler++
    )
{
            // jedes Unterfeld als Feld aus Elementen erzeugen
            Feld[UnterfeldZahler] = new Array();

            // Unterfeld elementeweise füllen mit Wert "[UnterfeldZahler,Unterfeld_Elemente_Zahler]"
            //                   wobei UnterfeldZahler und Unterfeld_Elemente_Zahler für die Indexwerte stehen

            for (       Unterfeld_Elemente_Zahler=0;
                        Unterfeld_Elemente_Zahler < AnzahlElementeImUnterfeld;
                        Unterfeld_Elemente_Zahler++
                )
            {
                        FeldGesamt[UnterfeldZahler][Unterfeld_Elemente_Zahler] =
                                                                        "["
                                                                    + UnterfeldZahler
                                                                    + ","
                                                                    + Unterfeld_Elemente_Zahler
                                                                    + "]";
            }
}
```

Beispiel für Erzeugung eines beliebig-dimensionalen Feldes:

```
<HTML>
<HEAD>
<SCRIPT  LANGUAGE="JavaScript">
<!--
            function mehr_dim_array_erzeugen( ein_dim_array_anzahl, ein_dim_array_elemente_anzahl)
            {
                        // ist Konstruktor für Erzeugung eines mehrdimensionalen Arrays
                        // als Menge von eindimensionalen Arrays

                        for (       var ein_dim_array_zahler = 0;
                                    ein_dim_array_zahler < ein_dim_array_anzahl;
                                    ein_dim_array_zahler++
                            )
                        {
                          this[ein_dim_array_zahler] = new Array(ein_dim_array_elemente_anzahl);
                                    // ein_dim_array_zahler immer ab 1
                                    // je ein eindimensionales Array erzeugen und zu this zuordnen
                                    //   this ist Zeiger des Konstruktors auf das per new
                                    //             anzulegende mehrdimensionale Feld
                        }

                        this.breite = ein_dim_array_anzahl;
                        this.hoehe = ein_dim_array_elemente_anzahl;
            }
// -->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT  LANGUAGE="JavaScript">
<!--
            zwei_dim_array = new mehr_dim_array_erzeugen (2, 3);
                        // zwei Arrays zu je 3 Feldelementen,
                        //             zweidimensionale Matrix mit
                        //             2 Zeilen zu je 3 Spalten
                        //    erzeugen
                        // Anzahl immer ab 1
```

```
        zwei_dim_array[1][2] = 17;
                                        // letztes eindimensionale Feld in dessen
                                        //          Element 2 (vorletztes)  mit Wert 17 initialisieren
                                        // Indexe immer ab 0
        document.write("Feld 1-2: " + zwei_dim_array[1][2]);
// -->
</SCRIPT>
</BODY>
</HTML>
```

Beispiel zur Konvertierung von Hexaziffern nach numerisch:

```
var hexaziffern_feld = [                // anstelle von new Array()
                        "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                        "A", "B", "C", "D", "E", "F"
                       ];

function dezimalwert_zu_hexaziffern(dezimal_wert)
{
        var high = Math.floor(dezimal_wert / 16);
        var low  = Math.floor(dezimal_wert - (high * 16));

        return(hexaziffern_feld[high] + hexaziffern_feld[low]);
}
```

Beispiele zu push und pop:

ab IE 5.5 bzw. NS 6.x existiert die Objektmethode .push() und .pop()

```
var names = new Array("Tom", "Mark", "Bart", "John");
names.push("Jim", "Richard", "Tim");
alert(names);
var last = names.pop();
alert("Last = " + last + "  Other = " + names);
```

vor IE 5.5 bzw. NS 6.x muss push und pop durch den Programmierer per Prototyping implementiert werden

Beispiel für push:

```
function push()
{
        var sub = this.length;
        for (var i = 0; i < push.arguments.length; ++i)
{
        this[sub] = push.arguments[i];
        sub++;
}
}

var names = new Array("Tom", "Mark", "Bart", "John");
                                        //  Es wird die Eigenschaft .prototype  erzeugt

// Prototyping der Instanz names
names.prototype.push = push;

// auch möglich: Prototyping des Script-Objektes Array generell
Array.prototype.push = push;
```

Beispiel für pop:

```
function pop()
{
        var lastElement = this[this.length - 1];
        this.length--;
        return lastElement;
}

var names = new Array("Tom", "Mark", "Bart", "John");
                                        //  Es wird die Eigenschaft .prototype  erzeugt

// Prototyping der Instanz names
```


```
names.prototype.pop = pop;

// Auch möglich: Prototyping des Script-Objektes Array generell
Array.prototype.pop = pop;
```

Beispiele zu shift und unshift:

Beispiel für unshift-Methode bei Browsern nicht ab IE5.5 bzw. nicht ab NS6.x:

```
function unshift()
{
    var i = unshift.arguments.length;
    for (var j = this.length - 1; j >= 0; --j)
    {this[j + i] = this[j]; }

    for (j = 0; j < i; ++j)
    {this[j] = unshift.argument[j]; }
}

Array.prototype.unshift = unshift;
```

Beispiel für shift -Methode ab IE5.5 bzw. NS 6.x:

```
var line = new Array("aaa", "bbb", "ccc", "ddd", "eee");
alert("first = " + line.shift() + " Other = " + line);
```

Beispiel für unshift -Methode ab IE5.5 bzw. NS 6.x:

```
var line = new Array("ccc", "ddd", "eee");
line.unshift("aaa", "bbb");
alert(line);
```

Eigenschaften:

.length — Anzahl der Elemente in einem Feld der Objektklasse Script-Objekt Array
Maximale Anzahl von Feldelementen: 4294967295

Methoden:

.concat() — Feld (Quellfeld1) kopieren in eine neue und automatisch erzeugte Instanz (Zielfeld) und optionales Anhängen von Werten (z.B. weiteren Quellfeldern2 ...) an das Ende des Zielfeldes.
Quellfeld1 kann leer sein
Verkettung erfolgt in der Folge der Kodierung der zu verkettenden Objekte, Ausdrücke etc.:
- Erstes Objekt in der Verkettung ist immer Quellfeld1.

wenn mehrere Quellfelder zu verketten sind, dann gilt
- wenn Feldelement **nicht** String **und nicht** Number ist, so
  - wird das Feldelemente als Zeiger kopiert bzw. verkettet
  - der Wert, auf den der Zeiger weist, wird nicht verwendet
  - bewirkt Änderung eines Wertes im Quell- bzw. Zielfeld auch die Änderung im Ziel- bzw. Quellfeld, da identische Zeigerbezüge vorliegen
- wenn Feldelement String oder Number ist, so
  - erfolgt Wertekopierung, also keine Zeigerkopierung
  - bewirkt Änderung eines Wertes im Quell- bzw. Zielfeld keine Änderung im Ziel- bzw. Quellfeld, da keine Zeigerbezüge vorliegen
- Änderung der Anzahl der Feldelemente in den Quellfeldern hat keine Auswirkung auf die Anzahl der Feldelemente im Zielfeld
- Verkettung von Feldern erfolgt
  - feldweise in der Folge der Kodierung der Quellfelder
  - und pro Quellfeld: elementeweise in der Folge der Feldelemente
- Felder haben die Objektklasse Script-Objekt Array

ab IE 5.5 und NS 6.x siehe auch .push()
Syntax:

```
[ var Zeiger = ] zeiger_auf_quell_feld.concat([werte_liste])
```

werte_liste — optional
kommagetrennte Werte
Wert kann z.B. sein
- Literal
- Referenz z.B. auf Feld
- aus Ausdruck
- aus Funktionsaufruf

Zeiger — auf die neue Instanz (Zielfeld), die automatisch erzeugt wird

Beispiel:

```
var Feld1 = new Array(0,1);
```


```
var Feld2_Quelle      = new Array(2, 3);
var Wert_Quelle       = 4;
var Feld3_Ziel = Feld1.concat(Feld2_Quelle, Wert_Quelle,);
alert(Feld3.join());     // "0,1,2,3,4"
```

Beispiel:

```
var QuellFeld1=new Array("a","b","c");
var QuellFeld2=new Array(1,2,3);
var ZielFeld=QuellFeld1.concat(QuellFeld2) // Zielfeld enthält Wertkopien also  ["a","b","c",1,2,3]
```

Beispiel:

```
var Variable1="Hallo ";
var Variable2="Du";
var Variable3="!";

var QuellFeld1=new Array(Variable1,Variable2,Variable3);
var QuellFeld2=new Array("Hallo ","Du","!");
var ZielFeld=QuellFeld1.concat(QuellFeld2) ;
        // Zielfeld enthält Wertkopien also  [      Zeiger_Auf_Variable1,
        //                                          Zeiger_Auf_Variable2,
        //                                          Zeiger_Auf_Variable3,
        //                                          "Hallo ","Du","!"
        //                                  ]
```

.join()

Feld auslesen und als String liefern, wobei jedes Feldelement mit einem freidefinierbaren Trenner im String getrennt wird, der mit eingebaut wird
Feld elementeweise nach String kopieren und als 1 gemeinsamen String liefern
Feld hat die Objektklasse Script-Objekt Array
Syntax:

```
[ var Kette1 = ] zeiger_auf_feld.join([Kette2]);
```

Kette2 optional
String
ist der Trenner als Zeichefolge
wird in Kette1 eingebaut
Standard ist "," also Komma

Kette1 String
Feldinhalt mit Elementetrennung laut Kette 1
ist Leerkette wenn mindestens ein Feldelement nicht vorhanden ist weil undefined bzw. null

Beispiele:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.join("-"));  // "0-1-2-3-4"
alert(Feld.join());      // "0,1,2,3,4"

var Feld = new Array("Wind","Rain","Fire");
var Kette = Feld.join(" + ");       // Kette enthält "Wind + Rain + Fire"
```

.reverse()

Reihenfolge der Feldelemente physisch umkehren, also

alt	**neu**
1. Element	letztes Element
letztes Element	1. Element

Feld hat die Objektklasse Script-Objekt Array
Syntax:

```
zeiger_auf_feld.reverse()
```

liefert nichts

Beipiel:

```
var Feld1 = new Array(0,1,2,3,4);          // Feld 1 ist Zeiger vor reverse
Feld1.reverse();                                  // Feld 2 ist Zeiger nach reverse, Feld1 ist ungültig
alert(Feld1.join());                       // "4,3,2,1,0"
```

.slice()

Elementefolge aus Quellfeld als neues Feld (Zielfeld) liefern
nachträgliche Änderung der Elementeanzahl im Quellfelde wirken sich nicht auf Zielfeld aus (keine dynamische Verkettung)
wenn Quellfeld **nicht** String **und nicht** Number enthält:
neues Feld hat Feldelemente als Zeiger auf die Elemente also nicht als Wertkopie
Werte-Änderung im Quellfeld bewirkt sofortige Änderung im Zielfeld, da identische Zeiger vorliegen
wenn Quellfeld Strings oder Number enthält:

neues Feld hat Feldelemente als Wertkopien aus dem Quellfeld
Werte-Änderung im Quellfeld bewirkt keine Änderung im Zielfeld, da nicht identische Zeiger vorliegen
Felder haben die Objektklasse Script-Objekt Array
Syntax:
[var Zeiger =] zeiger_auf_feld.slice(StartIndex[,EndeIndexPlus1])

StartIndex Index, ab 0 und < Länge des Feldes laut .length

EndeIndexPlus1 Integer,
wenn 0, so > StartIndex
Achtung: Element laut EndeIndexPlus1 wird nicht verwendet !
wenn < 0, so .length + EndeIndexPlus1 verwendet
Bsp.: EndeIndexPlus1 ist -1
also .length + (- 1) verwendet
Achtung: Element laut .length + EndeIndexPlus1 wird nicht verwendet !
Standard: .length (also Elemente verwendet bis Index .length -1)

Beispiel:
```
var Feld1 = new Array(0,1,2,3,4);
var Feld2 = Feld1.slice(0,1);
alert(Feld2.join());     // "0"
var Feld3 = Feld1.slice(1);
alert(Feld3.join());     // "1,2,3,4"
```

Beispiel:
```
var QuellFeld=new Array("a","b","c")
var ZielFeld=QuellFeld.slice(0,2) // Zielfeld enthält Wertkopien also  ["a","b"]
```

Beispiel:
```
var Variable1="Hallo ";
var Variable2="Du";
var Variable3="!";

var QuellFeld=new Array(Variable1,Variable2,Variable3);
var ZielFeld=QuellFeld.slice(0,-1) ; // 2 verwendet, denn Länge 3 minus 1 ist  2
        // Zielfeld enthält Wertkopien also  [      Zeiger_Auf_Variable1,
        //                                         Zeiger_Auf_Variable2
        //                                  ]
```

.sort()

Feldelemente physisch sortieren
sind zwei direkt zusammenliegende Feldelemente wertmäßig identisch, so werden deren Positionen im Feld nicht geändert
ein Feldelement mit Wert undefinded landet am Feldende:
Liegen mehrere Feldelemente mit Wert undefined vor, so landen diese in der Reihenfolge vor der Sortierung am Ende des Feldes nach der Sortierung
Standardsortierung : laut ASCII-Zeichensatz und aufsteigend
Feld hat die Objektklasse Script-Objekt Array
Syntax:
zeiger_auf_feld.sort([Zeiger])

Zeiger optional
wenn nicht kodiert, so
Sortierung lexikographisch
jedes Feldelement wird dafür automatisch temporär in String umgewandelt
Bsp.: Wert 80 wird zu "80"
Wert 9 wird zu "9
sortieren ergibt 80 und dann 9
wenn kodiert, so Zeiger auf Funktion, also
Funktionsbezeichner **ohne** () und **ohne** Parameter kodieren
Funktion vergleicht 2 Feldelemente
Funktion muss folgenden Aufbau haben:

```
freier_bezeichner(Arg1,Arg2)
        // Bezeichner der Argumente frei wählbar
        // Vergleich auf <=   und >=  leider nicht zulässig
        {
```


```
                                        var Wert = 0;

                                        if (Arg1 < Arg2)
                                        {Wert--;} // muss < 0 sein
                                        else
                                        {
                                                if (Arg1 > Arg2)
                                                {Wert++;} // muss > 0 sein
                                }

                                        return Wert; // Arg1 == Arg2 so immer 0
                                }
```

liefert nichts

Hinweis für Feldelement mit undefiniertem Inhalt und Null-Zeiger:

```
if (Wert == null)               // true
var Wert;                       // Variable deklariert aber ohne Wert
if (Wert == undefined)          // true
if (Wert == null)               // true
var Wert=10;                    // Variable deklariert aber mitWert
if (Wert == undefined)          // false
if (Wert == null                // false
```

Beispiel:

```
var Feld1 = new Array("4","3","2","1","0");
Feld1.sort();           // Feld1 ist ungültig
alert(Feld1.join());    // "0,1,2,3,4"
```

Beispiel für numerische Feldelemente:

```
function Sortiere(Arg1, Arg2)
{return Arg1 - Arg2;}           // Arg1 > Arg2 so positiver Wert
                                // Arg1 < Arg2 so negativer Wert
                                // Arg1 == Arg2 so 0
zeiger_auf_feld.sort(Sortiere);
```

.splice()

Feldelemente-Folge aus dem Feld entfernen ab Indexposition
optional neue Objekte in das Feld einfügen ab Indexposition
alle entfernte Feldelemente in einem neuen Feld liefern in der Reihenfolge des Entfernen
Felder haben die Objektklasse Script-Objekt Array
Syntax:

```
[ var Zeiger = ] zeiger_auf_feld.splice(Wert1, Wert2 [, werte_liste])
```

werte_liste — optional
Liste der einzufügenden Objekte ab Index laut Wert1
kommagetrennte Werte
Anzahl der Listenelemente beliebig
Wert kann z.B. sein
- Literal
- Referenz z.B. auf Feld
  - wenn Feld, so dessen Elemente einzeln eingefügt
- aus Ausdruck
- aus Funktionsaufruf

Wert1 — Index, ab 0 und < Länge des Feldes laut .length
Startposition der zu entfernenden Feldelemente-Folge
in der Anzahl laut Wert2
Startposition der einzufügenden Objekte laut werte_liste
in der Anzahl laut Anzahl der Listenelemente

Wert2 — Anzahl der Elemente, die **entfernt** werden sollen
ab Index laut Wert1
<= Länge des Feldes laut .length
> 0 wenn keine werte_liste kodiert, so wird nur entfernt
0 wenn keine werte_liste kodiert, so wird weder entfernt noch eingefügt

Zeiger — auf die neue Instanz, die automatisch erzeugt wird
und alle entfernten Feldelemente in der Reihenfolge des Entfernen enthält, falls Wert2 > 0 ist

Beispiel:

```
var Feld1 = new Array(0,0,1,2,3,4);
var Feld2 = Feld1.splice(0,1);
alert(Feld2.join());     // "0,1,2,3,4"
var Feld3 = Feld1.splice(0,2,-1,0);
alert(Feld3.join());     // "-1,0,1,2,3,4"
```

.toString() Feldinhalt als String liefern, wobei Feldelemente im String mit Komma getrennt sind
Syntax:
[var Kette =] zeiger_auf_feld.toString()

4.2.2.1.1. Array JScript-Objekt im Internet Explorer ab Version 5.5

ab IE 5.5 wurde das Array Script-Objekt um folgende Methoden erweitert:

.pop() letztes Feldelement entfernen und dann als Variable liefern
es wird Eigenschaft .length verändert
Feld hat die Objektklasse JScript-Objekt
Syntax:
[var Zeiger =] zeiger_auf_feld.pop()

Beispiel:

```
var names = new Array("Tom", "Mark", "Bart", "John");
names.push("Jim", "Richard", "Tim");
alert(names);
var last = names.pop();
alert("Last = " + last + "  Other = " + names);
```

Beispiel:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.pop());      // "4"
alert(Feld.join());      // "0,1,2,3"
```

Beispiel für pop vor dem IE 5.5:

```
function pop()
{
    var lastElement = this[this.length - 1];
    this.length--;
    return lastElement;
}

var names = new Array("Tom", "Mark", "Bart", "John");                    // Es wird die Eigenschaft .prototype erzeugt

// Prototyping der Instanz names
names.prototype.pop = pop;

// Auch möglich: Prototyping des Script-Objektes Array generell
Array.prototype.pop = pop;
```

.push() Feld durch Anhängen von neuen Elementen erweitern und danach die neue Feldlänge liefern
es wird Eigenschaft .length verändert
Feld hat die Objektklasse JScript-Objekt
Syntax:
[var Wert =] zeiger_auf_feld.push(Zeigerliste)

Zeigerliste kommagetrennte Zeiger auf anzuhängende Elemente
Anhängen laut Listenelemente-Folge

Beispiel:

```
var names = new Array("Tom", "Mark", "Bart", "John");
names.push("Jim", "Richard", "Tim");
alert(names);
var last = names.pop();
alert("Last = " + last + "  Other = " + names);
```

vor IE 5.5 muss push und pop durch den Programmierer per Prototyping implementiert werden

Beispiel:

```
var Feld1          = new Array(0,1);
var Feld2_Quelle   = new Array(2,3);
var Wert_Quelle    = 4;
alert(Feld1.push(Wert_Quelle , Feld2_Quelle));      //  4 und nicht 5 !
alert(Feld1.join());     // "0,1,2,3"                // 4 wird nicht angehangen
```


Beispiel für push vor dem IE 5.5:

```
function push()
{
	var sub = this.length;
	for (var i = 0; i < push.arguments.length; ++i)
	{
		this[sub] = push.arguments[i];
		sub++;
	}
}

var names = new Array("Tom", "Mark", "Bart", "John");		// Es wird die Eigenschaft .prototype erzeugt

// Prototyping der Instanz names
names.prototype.push = push;

// auch möglich: Prototyping des Script-Objektes Array generell
Array.prototype.push = push;
```

.shift()

erstes Feldelement entfernen und dann als Variable liefern
es wird Eigenschaft .length verändert
Feld hat die Objektklasse JScript-Objekt
Syntax:

[var Zeiger =] zeiger_auf_feld.shift()

Zeiger auf entferntes Feldelement

Beispiel:

```
var line = new Array("aaa", "bbb", "ccc", "ddd", "eee");
alert("first = " + line.shift() + " Other = " + line);
```

Beispiel:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.shift());	// 0
alert(Feld.join());	// "1,2,3,4"
```

.unshift()

neue Feldelemente am Feldanfang einfügen und dann neue Länge des Array liefern
Feld hat die Objektklasse JScript-Objekt
Syntax:

[var Wert =] zeiger_auf_feld.unshift(Zeigerliste)

Beispiel:

```
var line = new Array("ccc", "ddd", "eee");
line.unshift("aaa", "bbb");
alert(line);
```

Beispiel:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.unshift(-1));
alert(Feld.join());	// "-1,0,1,2,3,4"
```

Beispiel für unshift-Methode bei Browsern vor dem IE5.5:

```
function unshift()
{
	var i = unshift.arguments.length;
	for (var j = this.length - 1; j >= 0; --j)
	{this[j + i] = this[j]; }

	for (j = 0; j < i; ++j)
	{this[j] = unshift.argument[j]; }
}

Array.prototype.unshift = unshift;
```

Zur Konvertierung eines VisualBasic-Array in ein JScript-Array dient die Methode .toArray() eines VisualBasic-Array

Beispiel:

```
var DatenSpeicher = new ActiveXObject("Scripting.Dictionary");
```


```
function SchluesselVorhanden(DatenSchluessel)
{return DatenSpeicher.Exists(DatenSchluessel);}

// Daten-Elemente erzeugen
var DatenSchluessel  = "a";
var Date             = "test"

// prüfen ob Schlüssel noch nicht vorhanden ist
if ( ! (SchluesselVorhanden (DatenSchluessel)))
{
        // Schlüssel nicht vorhanden, also Date hinzufügen
        DatenSpeicher.add (DatenSchluessel, Date);

        // und Daten-Referenz bilden
        var DatenReferenz = DatenSpeicher.Items();

        // und Feld bilden
        //        VisualBasic-Feld erzeugen
        var DatenOhneSchluessel_Feld = new VBArray(DatenReferenz);
        //        und nach JScript-Feld konvertieren
        DatenOhneSchluessel_Feld = DatenOhneSchluessel_Feld.toArray();

        // und Daten anzeigen
        var DatenOhneSchluessel_FeldLaenge = DatenOhneSchluessel_Feld.length

        if (DatenOhneSchluessel_FeldLaenge > 0)
        {
                for (var i = 0 ; i < DatenOhneSchluessel_FeldLaenge; i++)
                {alert("Date " + i + " = " + DatenOhneSchluessel_Feld [i]);}
        }
        else
        {alert("Dictionary ist leer!");}
}
```

4.2.2.1.2. Array Script-Objekt im Netscape ab Version 6.x (ab Javascript 1.5)

ab NS 6.x wurde das Array Script-Objekt um folgende Methoden erweitert:

.pop() letztes Feldelement entfernen und dann als Variable liefern
es wird Eigenschaft .length verändert
Feld hat die Objektklasse Script-Objekt
Syntax:
[var Zeiger =] zeiger_auf_feld.pop()

Beispiel:

```
var names = new Array("Tom", "Mark", "Bart", "John");
names.push("Jim", "Richard", "Tim");
alert(names);
var last = names.pop();
alert("Last = " + last + "  Other = " + names);
```

Beispiel:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.pop());      // "4"
alert(Feld.join());      // "0,1,2,3"
```

Beispiel für pop vor dem NS 6.x:

```
function pop()
{
        var lastElement = this[this.length - 1];
        this.length--;
        return lastElement;
}

var names = new Array("Tom", "Mark", "Bart", "John");                    // Es wird die Eigenschaft .prototype erzeugt

// Prototyping der Instanz names
names.prototype.pop = pop;

// Auch möglich: Prototyping des Script-Objektes Array generell
Array.prototype.pop = pop;
```


.push() Feld durch Anhängen von neuen Elementen erweitern und danach die neue Feldlänge liefern
es wird Eigenschaft .length verändert
Feld hat die Objektklasse Script-Objekt
Syntax:

```
[ var Wert = ] zeiger_auf_feld.push(Zeigerliste)

Zeigerliste        kommagetrennte Zeiger auf anzuhängende Elemente
                   Anhängen laut Listenelemente-Folge
```

Beispiel:

```
var names = new Array("Tom", "Mark", "Bart", "John");
names.push("Jim", "Richard", "Tim");
alert(names);
var last = names.pop();
alert("Last = " + last + "  Other = " + names);
vor IE 5.5 muss push und pop durch den Programmierer per Prototyping implementiert werden
```

Beispiel:

```
var Feld1          = new Array(0,1);
var Feld2_Quelle   = new Array(2,3);
var Wert_Quelle    = 4;
alert(Feld1.push(Wert_Quelle , Feld2_Quelle));     //  4 und nicht 5 !
alert(Feld1.join());   // "0,1,2,3"                 // 4 wird nicht angehangen
```

Beispiel für push vor dem NS 6.x:

```
function push()
{
        var sub = this.length;
        for (var i = 0; i < push.arguments.length; ++i)
        {
                this[sub] = push.arguments[i];
                sub++;
        }
}

var names = new Array("Tom", "Mark", "Bart", "John");                    //  Es wird die Eigenschaft .prototype
erzeugt

// Prototyping der Instanz names
names.prototype.push = push;

// auch möglich: Prototyping des Script-Objektes Array generell
Array.prototype.push = push;
```

.shift() erstes Feldelement entfernen und dann als Variable liefern
es wird Eigenschaft .length verändert
Feld hat die Objektklasse Script-Objekt
Syntax:

```
[ var Zeiger = ] zeiger_auf_feld.shift()

        Zeiger      auf entferntes Feldelement
```

Beispiel:

```
var line = new Array("aaa", "bbb", "ccc", "ddd", "eee");
alert("first = " + line.shift() + " Other = " + line);
```

Beispiel:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.shift());     // 0
alert(Feld.join());       // "1,2,3,4"
```

.unshift() neue Feldelemente am Feldanfang einfügen und dann neue Länge des Array liefern
Feld hat die Objektklasse Script-Objekt
Syntax:

```
[ var Wert = ] zeiger_auf_feld.unshift(Zeigerliste)
```

Beispiel:

```
var line = new Array("ccc", "ddd", "eee");
line.unshift("aaa", "bbb");
alert(line);
```

Beispiel:

```
var Feld = new Array(0,1,2,3,4);
alert (Feld.unshift(-1));
alert(Feld.join());       // "-1,0,1,2,3,4"
```

Beispiel für unshift-Methode bei Browsern vor dem NS 6.x:

```
function unshift()
{
        var i = unshift.arguments.length;
        for (var j = this.length - 1; j >= 0; --j)
        {this[j + i] = this[j]; }

        for (j = 0; j < i; ++j)
        {this[j] = unshift.argument[j]; }
}

Array.prototype.unshift = unshift;
```

4.2.2.2. Array Script-Objekt aus Literalen

Diese Variante eines Feldes ist eine vereinfachte Kodierung für Elemente vom Literaltyp (String- oder numerisches Literal). Mit der Instanzierung wird automatisch die new-Anweisung mit internem Konstruktor aufgerufen. Mit diesem Feld findet auch die Anweisung for .. in Anwendung.

Syntax:

```
var Zeiger = {literal_liste}
```

literal_liste: kommagetrennte Elemente

Element
mit Aufbau "kette1"**:**"kette2"

Doppelpunkt muss kodiert werden

kette 1 Eigenschaft des Literal-Objektes
nur Plain-Text

kette2 Wert der Eigenschaft des Literal-Objektes
nur Plain-Text

Beispiel:

```
var LiteralObjekt = {"Vorname":"Otto", "Nachname":"Waalkes"};

var Index = "";
var Menge = {"a" : "Athen" , "b" : "Berlin", "c" : "Paris", "d" : "Kairo"};
var Kette = "";

SchleifenAnweisung :                    // ein Label (Marke)
{
        for (Index in Menge)
        {
                Kette = "Hauptstadt  von ";

                switch (Menge[Index])
                {
                        case "Berlin":          {
                                                        Kette += "Deutschland: " + Menge[Index];
                                                        // nicht weiter auf Athen und Kairo prüfen
                                                        break;
                                                }

                        case "Athen":           {
                                                        Kette += "Griechenland: " + Menge[Index];
                                                        // kein break, also noch auf Kairo prüfen
                                                }

                        case "Kairo":           {       Kette += "Ägypten: " + Menge[Index]; }

                        default:                {
                                                        // im Falle von Paris:
                                                        //            Kairo wird nie angezeigt, da im Feld
                                                        //            hinter Paris
```


```
                                                    break SchleifenAnweisung;
                                                }
                                        }

                                        {alert(Kette);}         // nicht bei Paris abgearbeitet
                                }
                        }
```

Eigenschaften:
wie Array als Script-Objekt
Methoden:
wie Array als Script-Objekt inklusive der Methoden ab IE 5.5 bzw. 6.x

4.2.3. Booelan Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
siehe auch Basis-Datentyp boolean
Erzeugung:

[var Zeiger2 =] new Boolean([Zeiger1]);

var Zeiger2 = Zeiger1;

Zeiger 1

ausdruck	Ausdruck muss true oder false liefern und niemals void(ausdruck)
Wert	true oder false oder beliebiger Wert wenn 0, -0, +0, null, false, NaN, undefined oder Leerkette "" so wird false verwendet sonst wird true verwendet z.B. "false" wird als true verwendet Nicht-Null-Zeiger wird als true verwendet 1 wird als true verwendet
Referenz	auf Objekt mit Wert (siehe oben)
Default	Boolean-Wert false

Hinweis: zeiger_auf_boolean_objekt.toString() liefert String-Werte "true" bzw. "false"
zeiger_auf_boolean_objekt.valueOf () liefert Boolean-Werte true bzw. false

logischen Vergleich mit vor bzw. nach expliziter Wertzuweisung:
Beispiel:

```
var InitWert = false;
var x = new Boolean(InitWert);
if (x)                  // falscher Vergleich da immer true, egal welcher Initwert
if (x == false)         // korrekter Vergleich der true liefert

// neuer Wert
x = false;
if (x == false)         // true
if (!x)                 // true
```

Konvertierung in Boolean-Wert:
Booleanwerte sind true (nicht "true" etc.)
false (nicht "false" etc.)

Syntax: [var Zeiger2 =] Boolean(Zeiger1);

Zeiger 1

ausdruck	Ausdruck muss true oder false liefern und niemals void(ausdruck)
Wert	true oder false oder beliebiger Wert wenn 0, -0, +0, null, false, NaN, undefined oder Leerkette "" so wird false verwendet sonst wird true verwendet z.B. "false" wird als true verwendet Nicht-Null-Zeiger wird als true verwendet 1 wird als true verwendet
Referenz	auf Objekt mit Wert (siehe oben)
Default	Boolean-Wert false

Eigeschaften:

keine
Methoden:
keine

4.2.4. Date Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
Dieses Objekt verwaltet Datum und Zeit.
Berechnungen mit Date Script-Objekt sind nur auf Basis von Millisekunden relativ zum 01.01.19070 0 Uhr Weltzeit möglich
(positiver bzw. negativer Wert der Anzahl der Millisekunden).
Zeitzonen-Berechnungen sind nur per .getTimezoneOffset() möglich.
maximales Datum muss im Jahr 1970 + 285,616 bzw. - 285,616 Jahre liegen
+ 100000000 bzw. -100000000 Tage liegen
Weltzeit: Universal Coordinated Time (UTC)
entspricht GMT (Greenwich Mean Time)
siehe auch Basis-Datenstruktur date

Erzeugung:

var Zeiger = new Date();

Objekt wird initialisiert mit der positiver Anzahl der Millisekunden
ab dem 01.01.1970 0 Uhr **Weltzeit (UTC)**
bis zum aktuellen Zeitpunkt der **Instanzierung** des Objektes in **Weltzeit**

var Zeiger = new Date(Wert);

Wert zum Initialisieren des Objektes auf **Weltzeit**
Anzahl der Millisekunden realtiv zu dem 01.01.1970 0 Uhr **Weltzeit**
positive, negativ

var Zeiger = new Date(Jahr, Monat, Tag [, Stunden [, Minuten [, Sekunden [,Millisekunden]]]]);

alle Argumente dienen zum Initialisieren des Objektes auf **Weltzeit**
Lücke in der Argumentenliste nicht zulässig:
0 kodieren wenn Argument nicht verwendet werden soll
bei Argumenten, deren Werte über den gültigen Wertebereich hinausgehen, ist die Scriptmaschine **fehlertolerant**:

Beispiel: 61 als Angabe für Sekunden --> Script verwendet 1 Minute und 1 Sekunde und berechnet alle darüberliegende Werte neu, falls diese ebenfalls Überschreitungen des Wertebereiches liefern

Ursache: Scriptmaschine rechnet **immer** in Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit
(automatische Umwandlung der Werte in Millisekunden)

Jahr
Integer
immer vierstellig z.B. 1976
minimal 1970

Monat
String mit **englischem** Monatsnamen
oder Integer 0 bis 11
0 entspricht "January"
11 entspricht "December"

Tag
Integer
1 bis 31
Achtung: Der Programmierer muss Schaltjahre und Anzahl der Tage in dem Monaten wissen.

Stunden
Integer
0 bis 23
24 Stunden sind 1 Tag
Default ist 0

Minuten
Integer
0 bis 59
60 Minuten sind 1 Stunde
Default ist 0

Sekunden
Integer
0 bis 59

60 Sekunden sind 1 Minute
Default ist 0

Millisekunde Integer
0 bis 999
1000 Millisekunden sind 1 Sekunde
Default ist 0

Beispiel 1:

```
<SCRIPT>
    function ErzeugeCookie(Name, Value)       // Name und Wert sind Strings
    {
        var date = new Date();
        var document.cookie =      Name
                                 + "="
                                 + escape(Value)
                                 + "; expires=" + date.toGMTString(); // aktuelles Datum
    }

    function LoescheCookie (Name, Value)      // Name und Wert sind Strings
    {
        var document.cookie =     Name
                                + "="
                                + escape(Value)
                                + "; expires=Fri, 31 Dec 1999 23:59:59GMT;"; // altes Datum

    function LeseCookieWert(Name)
    // Name des zu lesenden Cookie ist String
    // liefert Cookiewert aus unescape() als String
    {
        var CookieWert="";   // Annahme: Cookie nicht gefunden oder mit Leerkette als Wert

        //  Feld der Cookies referenzieren
        //          erzeugt durch Separation anhand Semikolon
        var CookieFeld = document.cookie.split("; ");
        var AnzahlCookies = CookieFeld.length;

        // Feldelement umfasst Name und Wert des Cookie
        //          Aufbau     Cookie_Name = Cookie_Wert
        //                                  Separator ist Gleichheitszeichen
        //          Index 0 für Cookien_Name
        //          Index 1 für Cookie_Wert
        var CookieNameUndWertAlsFeld;

        // CookieFeld auslesen und auf Cookie laut Name prüfen
        var Gefunden=false;
        var Index = 0;
        do
        {
            // aktuelles Cookie lesen
            CookieNameUndWertAlsFeld = CookieFeld [Index].split("=");

            // und auf Name prüfen
            if (Name == CookieNameUndWertAlsFeld [0])
            {
                CookieWert = unescape(CookieNameUndWertAlsFeld [1]);
                Gefunden=true;
            }
            else
            {Index++;}
        }
        while (         ( !Gefunden )
            && ( Index < AnzahlCookies)
          );

        return CookieWert;
    }
</SCRIPT>
```

Beispiel 2:

```
<HTML>
<HEAD>
```


```
<STYLE>
	.user_data_speicher_klasse {behavior:url(#default#userData);}
</STYLE>
<SCRIPT>
	// Cachname festlegen
	//	es können diverse Cachenamen definiert und somit Versionen von Cache
	//		verwaltet werden
	var FreierCacheName = "InputCache";

	// Cache-Attribut festlegen
	//	es können diverse Attribute definiert und somit Versionen von Input-Daten
	//		verwaltet werden
	var FreiesCacheAttribut = "InputCacheAttribut";

	// zu cachende Daten referenzieren
	var InputDatenObjekt = ID_Formular.ID_Input;

	function InputSichern()
	{
		// +++++++++ Zeitstempel ++++++++++++++++++++++++
		var ZeitpunktJetzt = new Date();
		var ZeitpunktJetztInMinuten = ZeitpunktJetzt.getMinutes();

		// Zeitstempel festlegen: Ab jetzt + 20 Minuten
		var Zeitstempel = ZeitpunktJetztInMinuten + 20;

		// und als UTC-Format erzeugen
		var ZeitstempelUTC = Zeitstempel.toUTCString();

		// und Zeitstempel dem Input-Objekt verpassen
		ID_Input.expires = ZeitstempelUTC;

		// ++++++++ Daten chachen ++++++++++++++++++++++
		// aktuelle Daten holen laut Input-Objekt
		var InputDaten = InputDatenObjekt.value;

		// Attrbibut instanzieren und mit Daten füllen
		InputDatenObjekt.setAttribute(FreiesCacheAttribut, InputDaten);

		// und Cache saven
		InputDatenObjekt.save(FreierCacheName);
	}

	function InputLaden()
	{
		// Cache laden
		InputDatenObjekt.load(FreierCacheName);

		// und Daten zum Attribut laut Sicherung lesen
		var InputDaten = InputDatenObjekt.getAttribute(FreiesCacheAttribut);
	}

</SCRIPT>
</HEAD>
<BODY>
	<FORM ID="ID_Formular">
		<INPUT	ID="ID_Input"
			CLASS="user_data_speicher_klasse"
			TYPE="text"
		>
		<INPUT TYPE="button" VALUE="sichern der Input-Daten" onclick="InputSichern()">
		<INPUT TYPE="button" VALUE="laden der Input-Daten"    onclick="InputLaden()">
	</FORM>
</BODY>
</HTML>
```

Beispiel 3:

```
<SCRIPT>
	window.onload=fnInit;

	function fnInit()
```


```
		{
			var AnzahlMillisekundenProTag =86400000;

			var DatumDokumentErzeugung =new Date(document.fileCreatedDate);

			var DatumHeute =new Date();

			var TagesDifferenz =	(  DatumHeute.getTime()
						  - DatumDokumentErzeugung.getTime()
						)
					/ AnzahlMillisekundenProTag;

			alert(	"Dokument erzeugt am " + DatumDokumentErzeugung
				+ "\n..... also vor " +parseInt(TagesDifferenz) + " Tagen"
			);
		}
	</SCRIPT>
```

Beispiel 4:

```
var Jetzt = new Date();
alert(Jetzt.toLocaleString());
```

Beispiel 5 füt Datumsanzeige in deutscher Norm:

```
HTML>
<HEAD>
<SCRIPT LANGUAGE=JavaScript1.2>
<!--
	//		Datumsanzeige in deutscher Norm

//*************************************************************************************************
//
//		Vom Programmierer ist nichts zu verändern
//
//*************************************************************************************************
	// Felder der deutschen Datumsangaben
	var WochenTag=new Array(
					"Sonntag",
					"Montag",
					"Dienstag",
					"Mittwoch",
					"Donnerstag",
					"Freitag",
					"Sonnabend"
				);

	var Monate=new Array(
					"Januar",
					"Februar",
					"März",
					"April",
					"Mai",
					"Juni",
					"Juli",
					"August",
					"September",
					"Oktober",
					"November",
					"Dezember"
				);

	// Daten des aktuellen Datums ermitteln
	var Datum_Aktuell=new Date();
	var Datum_Aktuell_Jahr=Datum_Aktuell.getYear();
	var Datum_Aktuell_WochenTag=Datum_Aktuell.getDay();
	var Datum_Aktuell_Monat=Datum_Aktuell.getMonth();
	var Datum_Aktuell_Tag=Datum_Aktuell.getDate();

	// Anpassungen der Daten des aktuellen Datums
	//		wenn Jahresanagabe < 1000 so 1900 dazuaddieren
```


```
                        if (Datum_Aktuell_Jahr < 1000) {Datum_Aktuell_Jahr+=1900;}

                        //        Vornull einbasteln
                        if (Datum_Aktuell_Tag<10)     {Datum_Aktuell_Tag="0"+Datum_Aktuell_Tag;}

                        // Formatierte Anzeige des aktuellen Datums in deutscher Norm
                        document.write(
                                        WochenTag[Datum_Aktuell_WochenTag]
                                      + ", "
                                      + Datum_Aktuell_Tag
                                      + "."
                                      + Monate[Datum_Aktuell_Monat]
                                      + " "
                                      + Datum_Aktuell_Jahr
                                    );
                //-->
                </SCRIPT>
                <BODY>
                </BODY>
                </HTML>
```

Zugriff:

date_instanz.eigenschaft
date_instanz.methode()

Eigenschaften:

keine

Methoden:

wird eine nachfolgend beschriebene Methode als deprecated genannt, so gilt das nur für den IE.
Empfehlung: Methode, die deprecated ist, auch im NS nicht verwenden.

.getDate() Tag des Monats in lokaler Zeit liefern

Syntax:

[Wert =] object.getDate()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
1 bis 31

.getDay() Tag der Woche in lokaler Zeit liefern

Syntax:

[Wert =] object.getDay()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 6
0 ist Sonntag
6 ist Samstag

.getFullYear() Jahreszahl vierstellig in lokaler Zeit liefern

Syntax:

[Wert =] object.getFullYear()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
maximal **1970 + bzw. -285,616 Jahre**

.getHours() Stunden in lokaler Zeit liefern

Syntax:

[Wert =] object.getHours()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 23

.getMilliseconds() Millisekunden in lokaler Zeit liefern

Syntax:

[Wert =] object.getMilliseconds()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 999

.getMinutes() Minuten in lokaler Zeit liefern
Syntax:
[Wert =] object.getMinutes()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 59

.getMonth() Monat in lokaler Zeit liefern
Syntax:
[Wert =] object.getMonth()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 11

.getSeconds() Sekunden in lokaler Zeit liefern
Syntax:
[Wert =] object.getSeconds()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 59

.getTime() Zeitwert als Anzahl der Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit liefern
positiv oder negativ möglich (wenn < 0 so vor obigen Datum)
Syntax:
[Wert =] object.getTime()

Wert Anzahl der Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit
wenn < 0 so vor 01.01. 1970 0 Uhr Weltzeit

.getTimezoneOffset() Minuten der lokalen Zeit als Differenz zur Weltzeit liefern, also das Offset der lokalen Zeitzone
Syntax:
[Wert =] object.getTimezoneOffset()

Wert Integer
ab 0

.getUTCDate() Tag des Monats in Weltzeit liefern
Syntax:
[Wert =] object.getUTCDate()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
1 bis 31

.getUTCDay() Tag der Woche in Weltzeit liefern
Syntax:
[Wert =] object.getUTCDay()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 6
0 ist Sonntag
6 ist Samstag

.getUTCFullYear() Jahreszahl vierstellig in Weltzeit liefern
Syntax:
[Wert =] object.getUTCFullYear()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
maximal **1970 + bzw. -285,616 Jahre**

.getUTCHours() Stunden in Weltzeit liefern
Syntax:
[Wert =] object.getUTCHours()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 23

.getUTCMilliseconds() Millisekunden in Weltzeit liefern
Syntax:
[Wert =] object.getUTCMilliseconds()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 999

.getUTCMinutes() Minuten in Weltzeit liefern
Syntax:
[Wert =] object.getUTCMinutes()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 59

.getUTCMonth() Monat in Weltzeit liefern
Syntax:
[Wert =] object.getUTCMonth()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 11

.getUTCSeconds() Sekunden in Weltzeit liefern
Syntax:
[Wert =] object.getUTCSeconds()

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 59

.getYear() deprecated

.parse() Datum als String parsen und Anzahl der Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit liefern
Datum als String in den Basis-Datentyp date konvertieren
Syntax:
[Wert =] object.parse(Kette)

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String mit Aufbau

n monat tag,jjjj hh:mm:ss x z

n monat/tag/jjjj hh:mm:ss x z

n monat-tag-jjjj hh:mm:ss x z

alle Elemente sind optional, wobei mindestens 1 Element kodiert werden muss

n nur englischer Wochentagname
wenn kalendarisch falscher Wochentag, so

automatisch korrigiert

monat nur englischer Name

tag 1 bis 31, ohne Vornull

jjjj für Jahr
2 oder vierstellig
wenn 2 stellig so unter 2000 gemeint (ab 1970)

hh und mm und ss wie üblich für Stunde Minute und Sekunde

xx AM oder PM bei 12-Stunden-Uhr
wenn hh bis ss nicht passend, so Fehler geliefert
z.B. 23:00 PM gibt es nicht, also Fehler erzeugt

z Zeitzone zB. UTC oder GMT

Wert Integer
Anzahl der Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit
wenn n kodiert wurde und n kalendarisch ein falscher Wochentag ist, so wird n automatisch korrigiert, bevor Wert geliefert wird

Beispiele:

"Jan 5, 1996 08:47:00"
"November 1, 1997 10:15:00 "
"November 1, 1997 10:15 AM"
"7/20/96 08:47:00"
"7-20-96 08:47:00"
"Tuesday November 9 1990"
"7-20-96 08:" es muss Doppelpunkt kodiert sein, damit 08 als Stundenangabe interpretiert wird
"7-20-96 08:47 GMT"

.setDate() Tag des Monats in lokaler Zeit setzen
wenn Tag > Anzahl Tage im Monat, so erfolgt automatisch Monatswechsel und Korrektur aller Angaben (inklusive Jahr)

Syntax:

object.setDate(Wert)

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
1 bis 31

.setFullYear() Jahreszahl vierstellig in lokaler Zeit setzen und optional Monat wie Tag im Monat
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert

Syntax:

object.setFullYear(Wert1 [,Wert2 [,Wert3]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Jahr
Integer
maximal **1970 + bzw. -285,616 Jahre**

Wert2 Monat
Integer
0 bis 11
Default laut .getMonth()

Wert3 Tag im Monat
Integer
1 bis 31
Default laut .getDate()

.setHours() Stunden in lokaler Zeit setzen und optional Minuten, Sekunden, Millisekunden
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert

Syntax:

object.setHours(Wert1 [,Wert2 [,Wert3 [,Wert4]]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Stunden
Integer
0-23

Wert2 Minuten
Integer
0 bis 59
Default laut .getMinutes()

Wert3 Sekunden
Integer
0 bis 59
Default laut .getSeconds()

Wert4 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

.setMilliseconds() Millisekunden in lokaler Zeit setzen
Syntax:
object.setMilliseconds(Wert)

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 999

.setMinutes() Minuten in lokaler Zeit setzen und optional Sekunden, Millisekunden
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setMinutes(Wert1 [,Wert2 [,Wert3]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Minuten
Integer
0 bis 59

Wert2 Sekunden
Integer
0 bis 59
Default laut .getSeconds()

Wert3 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

.setMonth() Monat in lokaler Zeit setzen und optional Tag im Monat
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setMonth(Wert1 [,Wert2])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Monat
Integer
0 bis 11

Wert2 Tag im Monat
Integer
1 bis 31
Default laut .getDate()

.setSeconds() Sekunden in lokaler Zeit setzen und optional Millisekunden
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setSeconds(Wert1 [,Wert2])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Sekunden
Integer
0 bis 59

Wert2 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

.setTime() Zeitwert in Weltzeit setzen, auch vor 1970
Syntax:
object.getTime(Wert)

Wert Anzahl der Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit
wenn < 0 so vor obigem Datum

.setUTCDate() Tag des Monats in Weltzeit setzen
wenn Tag > Anzahl Tage im Monat, so erfolgt automatisch Monatswechsel und Korrektur aller Angaben (inklusive Jahr)
Syntax:
object.setUTCDate(Wert)

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
1 bis 31

.setUTCFullYear() Jahreszahl vierstellig in Weltzeit setzen und optional Monat wie Tag im Monat
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setUTCFullYear(Wert1 [,Wert2 [,Wert3]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Jahr
Integer
maximal **1970 + bzw. -285,616 Jahre**

Wert2 Monat
Integer
0 bis 11
Default laut .getMonth()

Wert3 Tag im Monat
Integer
1 bis 31
Default laut .getDate()

.setUTCHours() Stunden in Weltzeit setzen und optional Minuten, Sekunden, Millisekunden
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setUTCHours(Wert1 [,Wert2 [,Wert3 [,Wert4]]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Stunden
Integer
0-23

Wert2 Minuten
Integer
0 bis 59
Default laut .getMinutes()

Wert3 Sekunden
Integer
0 bis 59
Default laut .getSeconds()

Wert4 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

.setUTCMilliseconds() Millisekunden in Weltzeit setzen
Syntax:
object.setUTCMilliseconds(Wert)

objekt Zeiger auf Instanz von Script-Objekt Date

Wert Integer
0 bis 999

.setUTCMinutes() Minuten in Weltzeit setzen und optional Sekunden, Millisekunden
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setUTCMinutes(Wert1 [,Wert2 [,Wert3]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Minuten
Integer
0 bis 59

Wert2 Sekunden
Integer
0 bis 59
Default laut .getSeconds()

Wert3 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

.setUTCMonth() Monat in Weltzeit setzen und optional Tag im Monat
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setUTCMonth(Wert1 [,Wert2])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Monat
Integer
0 bis 11

Wert2 Tag im Monat
Integer
1 bis 31
Default laut .getDate()

.setUTCSeconds() Sekunden in Weltzeit setzen und optional Millisekunden
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:
object.setUTCSeconds(Wert1 [,Wert2])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Sekunden
Integer
0 bis 59

Wert2 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

.setYear() deprecated

.toDateString() kompletten Inhalt des Date-Objektes als String liefern

Achtung: nur für Anzeige etc. verwenden und nicht für Berechnungen !
Syntax:
[var Kette =] object.toDateString()

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String

.toGMTString() deprecated

.toLocaleString() kompletten Inhalt des Date-Objektes als String in lokaler Einstellung auf User-PC liefern
nur für Jahre ab 2000
aber für Jahre von 1601 bis 1999 immer laut lokale Einstellungen des Betriebssystemes auf User-PC
siehe .toLocaleDateString()
Achtung: nur für Anzeige etc. verwenden und nicht für Berechnungen !
Syntax:
[var Kette =] object.toLocaleString()

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String

.toLocaleDateString() kompletten Inhalt des Date-Objektes als String in lokaler Einstellung des Betriebssystemes auf User-PC liefern
immer verwenden für Jahre von 1601 bis 1999 --> siehe .toLocaleString()
Achtung: nur für Anzeige etc. verwenden und nicht für Berechnungen !
Syntax:
[var Kette =] object.toLocaleDateString()

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String

.toLocaleTimeString() Zeitwert des Date-Objektes als String in lokaler Einstellung des Betriebssystemes auf User-PC liefern
Achtung: nur für Anzeige etc. verwenden und nicht für Berechnungen !
Syntax:
[var Kette =] object.toLocaleTimeString()

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String

.toString() Datum als String geliefert (je nach Ländereinstellung des Windows)
Beispiel: Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983
Syntax:
[var Kette =] zeiger_auf_date_objekt.toString()

.toTimeString() Zeitwert des Date-Objektes als String liefern
Achtung: nur für Anzeige etc. verwenden und nicht für Berechnungen !
Syntax:
[var Kette =] object.toTimeString()

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String

.toUTCString() Zeitwert des Date-Objektes als String in Weltzeit liefern
Achtung: nur für Anzeige etc. verwenden und nicht für Berechnungen !
Syntax:
[var Kette =] object.toUTCString()

objekt Zeiger auf Instanz von Script-Objekt Date

Kette String

.UTC() Inhalt des Date-Objektes setzen nach Weltzeit
(auch wenn Objekt bereits initialisiert wurde per new Anweisung)
sollte ein Wert außerhalb des Wertebereiches sein, so wird automatisch korrigiert
Syntax:

[var Wert8 =] object.UTC(Wert1 ,Wert2 ,Wert3 [, Wert4 [, Wert5 [,Wert6 [,Wert7]]]])

objekt Zeiger auf Instanz von Script-Objekt Date

Wert1 Jahr
Integer
maximal **1970 + bzw. -285,616 Jahre**

Wert2 Monat
Integer
0 bis 11

Wert3 Tag im Monat
Integer
1 bis 31

Wert4 Stunden
Integer
0-23
Default laut .getHours()

Wert5 Minuten
Integer
0 bis 59
Default laut .getMinutes()

Wert6 Sekunden
Integer
0 bis 59
Default laut .getSeconds()

Wert7 Millisekunden
Integer
0 bis 999
Default laut .getMilliseconds()

Wert8 Anzahl der Millisekunden relativ zum 01.01.1970 0 Uhr Weltzeit
wenn < 0 so vor obigem Datum

.valueOf() Wert als Anzahl der Millisekunden seite dem 1.1.1970 0 Uhr Weltzeit (UTC)
wenn Datum vor dem 1.1.1970 0 Uhr Weltzeit so Wert negativ
Syntax
[var Wert =] zeiger_auf_date_objekt.valueOf()

4.2.4. Enumerator JScript-Objekt des Internet Explorer

Dieses Objekt ist eine Komponente der Scriptsprache.
dient der erweiterten Verwaltung von von Elementen einer Collection (nicht reines Array)
durch Verwendung eines internen Indexs für den Zugriff auf ein Element der Collection
ist Analogon zu dem Array Script-Objekt für reine Felder

Erzeugung:

var Zeiger2 = new Enumerator([Zeiger1])

Zeiger1 auf eine Collection

Zeiger2 auf Enumerator-Objekt

Beispiel Laufwerke auf dem PC des Users ermitteln:

```
var DateiSystem              = new ActiveXObject("Scripting.FileSystemObject");
var DateiSystem_Laufwerke    = new Enumerator(DateiSystem.Drives);
var Kette = "";
var Laufwerk;
var LaufwerkBuchstabe;
var LaufwerkName;   // Netzname oder Volume-Bezeichner

for ( ; ! DateiSystem_Laufwerke.atEnd();DateiSystem_Laufwerke.moveNext())
{
        // Laufwerk ermitteln
        Laufwerk = DateiSystem_Laufwerke.item();

        // Laufwerksbuchstabe ermitteln
```


```
                        LaufwerkBuchstabe = Laufwerk.DriveLetter;
                        Kette = Kette + LaufwerkBuchstabe + " - ";

                        // Art des Laufwerkes ermitteln

                        if (Laufwerk.DriveType == 3)
                        {
                                // ist Netzlaufwerk

                                // öffentlicher Netz-Name des Laufwerkes
                                LaufwerkName = Laufwerk.ShareName;
                        }
                        else
                        {
                                // nicht Netzwerk-Laufwerk

                                // prüfen ob Laufwerk bereit ist
                                if (Laufwerk.IsReady)
                                {
                                        // ist bereit, dann Volume-Bezeichner ermittelbar
                                        LaufwerkName = Laufwerk.VolumeName;
                                }
                                else
                                { LaufwerkName = "[Drive not ready]";}
                        }

                        Kette += LaufwerkName + "\n";
                }
                alert (Kette);
```

Eigenschaften:
keine
Methoden:

.atEnd()	prüfen	auf Erreichen des Ende der Collection auf Zustand undefined eines Elementes der Collection auf leere Collection
	verwenden mit .moveNext() innerhalb einer Schleife	

Beispiel Laufwerke auf dem PC des Users ermitteln:

```
var DateiSystem                 = new ActiveXObject("Scripting.FileSystemObject");
var DateiSystem_Laufwerke       = new Enumerator(DateiSystem.Drives);
var Kette = "";
var Laufwerk;
var LaufwerkBuchstabe;
var LaufwerkName;   // Netzname oder Volume-Bezeichner

for ( ; ! DateiSystem_Laufwerke.atEnd(); DateiSystem_Laufwerke.moveNext())
{
        // Laufwerk ermitteln
        Laufwerk = DateiSystem_Laufwerke.item();

        // Laufwerksbuchstabe ermitteln
        LaufwerkBuchstabe = Laufwerk.DriveLetter;
        Kette = Kette + LaufwerkBuchstabe + " - ";

        // Art des Laufwerkes ermitteln

        if (Laufwerk.DriveType == 3)
        {
                // ist Netzlaufwerk

                // öffentlicher Netz-Name des Laufwerkes
                LaufwerkName = Laufwerk.ShareName;
        }
        else
        {
                // nicht Netzwerk-Laufwerk

                // prüfen ob Laufwerk bereit ist
                if (Laufwerk.IsReady)
                {
```


```
                                        // ist bereit, dann Volume-Bezeichner ermittelbar
                                        LaufwerkName = Laufwerk.VolumeName;
                                }
                                else
                                { LaufwerkName = "[Drive not ready]";}
                        }

                        Kette += LaufwerkName + "\n";
                }
                alert (Kette);
```

.item() Element einer Collection liefern laut aktueller Position in der Collection
zur Positionierung innerhalb der Collection wird ein interner Index verwendet

Beispiel Laufwerke auf dem PC des Users ermitteln:

```
var DateiSystem             = new ActiveXObject("Scripting.FileSystemObject");
var DateiSystem_Laufwerke   = new Enumerator(DateiSystem.Drives);
var Kette = "";
var Laufwerk;
var LaufwerkBuchstabe;
var LaufwerkName;   // Netzname oder Volume-Bezeichner

for ( ; ! DateiSystem_Laufwerke.atEnd();DateiSystem_Laufwerke.moveNext())
{
        // Laufwerk ermitteln
        Laufwerk = DateiSystem_Laufwerke.item();

        // Laufwerksbuchstabe ermitteln
        LaufwerkBuchstabe = Laufwerk.DriveLetter;
        Kette = Kette + LaufwerkBuchstabe + " - ";

        // Art des Laufwerkes ermitteln

        if (Laufwerk.DriveType == 3)
        {
                // ist Netzlaufwerk

                // öffentlicher Netz-Name des Laufwerkes
                LaufwerkName = Laufwerk.ShareName;
        }
        else
        {
                // nicht Netzwerk-Laufwerk

                // prüfen ob Laufwerk bereit ist
                if (Laufwerk.IsReady)
                {
                        // ist bereit, dann Volume-Bezeichner ermittelbar
                        LaufwerkName = Laufwerk.VolumeName;
                }
                else
                { LaufwerkName = "[Drive not ready]";}
        }

        Kette += LaufwerkName + "\n";
}
alert (Kette);
```

.moveFirst() aktuelle Position innerhalb der Collection auf das erste Element der Collection
setzen (auch wenn Collection leer ist)
zur Positionierung innerhalb der Collection wird ein interner Index verwendet
Element an aktueller Position ermitteln per .item()

Beispiel Laufwerke auf dem PC des Users ermitteln:

```
var DateiSystem             = new ActiveXObject("Scripting.FileSystemObject");
var DateiSystem_Laufwerke   = new Enumerator(DateiSystem.Drives);
var Kette = "";
var Laufwerk;
var LaufwerkBuchstabe;
```


```
var LaufwerkName;   // Netzname oder Volume-Bezeichner

// erstes Laufwerk im Dateisystem einstellen (erstes Element im Enumerator-Objekt)
DateiSystem_Laufwerke.moveFirst();
do
{
        // Laufwerk ermitteln
        Laufwerk = DateiSystem_Laufwerke.item();

        // Laufwerksbuchstabe ermitteln
        LaufwerkBuchstabe = Laufwerk.DriveLetter;
        Kette = Kette + LaufwerkBuchstabe + " - ";

        // Art des Laufwerkes ermitteln

        if (Laufwerk.DriveType == 3)
        {
                // ist Netzlaufwerk

                // öffentlicher Netz-Name des Laufwerkes
                LaufwerkName = Laufwerk.ShareName;
        }
        else
        {
                // nicht Netzwerk-Laufwerk

                // prüfen ob Laufwerk bereit ist
                if (Laufwerk.IsReady)
                {
                        // ist bereit, dann Volume-Bezeichner ermittelbar
                        LaufwerkName = Laufwerk.VolumeName;
                }
                else
                { LaufwerkName = "[Drive not ready]";}
        }

        Kette += LaufwerkName + "\n";

        // nächstes Laufwerk positionieren (nächstes Element im Enumerator-Objekt)
        DateiSystem_Laufwerke.moveNext();
}
while (!DateiSystem_Laufwerke.atEnd());

alert(Kette);
```

.moveNext() aktuelle Position innerhalb der Collection auf das nächste Element der Collection setzen (auch wenn Collection leer ist)
zur Positionierung innerhalb der Collection wird ein interner Index verwendet
Element an aktueller Position ermitteln per .item()

Beispiel Laufwerke auf dem PC des Users ermitteln:

```
var DateiSystem             = new ActiveXObject("Scripting.FileSystemObject");
var DateiSystem_Laufwerke   = new Enumerator(DateiSystem.Drives);
var Kette = "";
var Laufwerk;
var LaufwerkBuchstabe;
var LaufwerkName;   // Netzname oder Volume-Bezeichner

// erstes Laufwerk im Dateisystem einstellen (erstes Element im Enumerator-Objekt)
DateiSystem_Laufwerke.moveFirst();
do
{
        // Laufwerk ermitteln
        Laufwerk = DateiSystem_Laufwerke.item();

        // Laufwerksbuchstabe ermitteln
        LaufwerkBuchstabe = Laufwerk.DriveLetter;
        Kette = Kette + LaufwerkBuchstabe + " - ";

        // Art des Laufwerkes ermitteln
```

```
                    if (Laufwerk.DriveType == 3)
                    {
                            // ist Netzlaufwerk

                            // öffentlicher Netz-Name des Laufwerkes
                            LaufwerkName = Laufwerk.ShareName;
                    }
                    else
                    {
                            // nicht Netzwerk-Laufwerk

                            // prüfen ob Laufwerk bereit ist
                            if (Laufwerk.IsReady)
                            {
                                    // ist bereit, dann Volume-Bezeichner ermittelbar
                                    LaufwerkName = Laufwerk.VolumeName;
                            }
                            else
                            { LaufwerkName = "[Drive not ready]";}
                    }

                    Kette += LaufwerkName + "\n";

                    // nächstes Laufwerk positionieren (nächstes Element im Enumerator-Objekt)
                    DateiSystem_Laufwerke.moveNext();
            }
            while (!DateiSystem_Laufwerke.atEnd());

            alert(Kette);
```

4.2.5. error JScript-Objekt im Internet Explorer

Dieses Objekt ist eine Komponente der Scriptsprache.
enthält Fehlerinformationen zu einem Run-Time-Error
wird verwendet für Erzeugung eines privaten Run-Time-Errors
von Standard-Run-Time-Error:

Errornummer	32-Bit	Error-Code-Gruppe
	Bit 0 bis 15	Error-Code innerhalb der Gruppe

Erzeugung:

```
var Zeiger = new Error([ [Wert ,] Kette])
```

Wert Fehlernummer
Integer
>= 0
Standard ist 0

Kette Beschreibung des Fehlers
Standard ist Leerkette

Beispiel für private Ausnahmebedingung ohne Error-Objekt

```
function ErzeugeAusnahme Bedingung(Ausnahme Bedingung)
{
        this.Ausnahme Bedingung=Ausnahme Bedingung;
        this.AusnahmeArt="UserException";
}

function HoleMonatAlsString (MonatsNummer)  // liefert Monat als String
{
        var Index = MonatsNummer -1;

        var MonatsFeld =new Array("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec");

        if (MonatsFeld[Index] != null)
        {return MonatsFeld[Index];}
        else
        {
                var UserDefinierteAusnahmeBedingung =
                                        new ErzeugeAusnahme Bedingung ("FalscheMonatsNummer");
                throw UserDefinierteAusnahmeBedingung;
        }
```


```
}
```

Beispiel für private Ausnahmebedingung mit Error-Objekt

```
function HoleMonatAlsString (MonatsNummer)  // liefert Monat als String
{
        var Index = MonatsNummer -1;

        var MonatsFeld =new Array("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec");

        if (MonatsFeld[Index] != null)
        {return MonatsFeld[Index];}
        else
        {
                var UserDefinierteAusnahmeBedingung = new Error(8888 , "test");


                throw UserDefinierteAusnahmeBedingung;
        }
}
```

Beispiele für internes Error-Objekt "e" bei Verwendung von try ... catch:

Beispiel 1:

```
function Anzeige(Kette)
{alert(Kette); }

try {Anzeige("try1");
{
        try
        {
                throw "Das ist eine Fehlerbedingung";
                Anzeige("try 2");
        }
        catch(e)
        {
                if (e == "Das ist eine Fehlerbedingung" )
                {Anzeige("catch2 " + e); }
        }
        finally { Anzeige("finally2"); }
}
catch(e)  { Anzeige("catch1 " + e); }
finally   { Anzeige("finally1");}
```

Beispiel 2:

```
function AlterErmitteln(Wert)
{
        if(Wert < 0)
        {throw new Error(8888,"Fehler: Altersangabe muss >= 0 sein");}
}

try
{AlterErmitteln (-5);}  // aktiviert throw
catch(e)
{alert (e.message);}
```

Eigenschaften:

.description — Beschreibung des Run-Time-Errors
deprecated: dafür Eigenschaft .message verwenden, die indentische Funktion hat

.message — Beschreibung des Run-Time-Errors
identisch mit Eigenschaft .description, die aber deprecated ist

Beispiel:

```
function AlterErmitteln(Wert)
{
        if(Wert < 0)
        {throw new Error("Fehler: Altersangabe muss >= 0 sein");}
}

try
```


```
{AlterErmitteln (-5);}  // aktiviert throw
catch(e)
{alert (e.message);}
```

.name Ausnahmetyp des Run-Time-Error liefern
privater Run-Time-Error

"Error"	Standard-Run-Time-Error
"ConversionError"	Konvertierungsfehler
"RangeError"	Bereichfehler z.B. Feldlänge negativ
"ReferenceError"	Referenz-Fehler z.B. Feld mit negativer Anzahl von Feldelementen soll erzeugt werden
"RegExpError"	Fehler bei regular Expression (regExp-Objekt)
"SyntaxError"	
"TypeError"	Typfehler z.B. bei Wertzuweisung
"URIError"	falscher Uniform Resource Indicator (URI) z.B. bei encode()

Beispiel:

```
function AlterErmitteln(Wert)
{
    if(Wert < 0)
    {throw new Error("Fehler: Altersangabe muss >= 0 sein");}
}

try
{AlterErmitteln (-5);}  // aktiviert throw
catch(e)
{alert (e.message + " " + e.name);}
```

.number Nummer des Run-Time-Error

Beispiel:

```
function AlterErmitteln(Wert)
{
    if(Wert < 0)
    {throw new Error(8888,"Fehler: Altersangabe muss >= 0 sein");}
}

try
{AlterErmitteln (-5);}  // aktiviert throw
catch(e)
{alert (e.message + " " + e.number);}
```

Methoden:
keine

4.2.6. Function Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
Erzeugung:
siehe Anweisung function
Syntax:

```
zeiger_auf_funktion.eigenschaft
zeiger_auf_funktion.methode()

    zeiger_auf_funktion   laut Erzeugung
```

Hinweis: zeiger_auf_funktion.toString() liefert den Quellcode
Eigenschaften:
.arguments Zeiger auf das Script-Objekt arguments
alle Eigenschaften des Script-Objektes arguments können referenziert werden
arguments nur verfügbar während der Ausführung der Funktion

Beispiel:

```
function Test1()
{
    var AnzahleArgumente = arguments.length;
    var Kette = "Anzahl Argumente = " + AnzahleArgumente + "\n";

    for ( var i = 0; i < AnzahleArgumente; i++)
    {Kette = "Argument[" + i + "]= " + arguments[i] + "\n"; }

    alert(Kette);
}


function Test2(Wert1,Wert2)
```


```
{
        var AnzahleArgumente = arguments.length;
        var Kette = "Anzahl Argumente = " + AnzahleArgumente + "\n";

        for ( var i = 0; i < AnzahleArgumente; i++)
        {Kette = "Argument[" + i + "]= " + arguments[i] + "\n"; }

        alert(Kette);
}

        Test1();
        Test2(10,20);
```

.caller Zeiger auf den Aufrufer der Funktion
Aufrufer ist eine andere Funktion (außer bei Rekursion)

.length Wert laut zeiger_auf_funktion.arguments.length

Methoden:

.apply() ein anderes Objekt anstelle des arguments Script-Objekt verwenden
.call() eine Methode eines anderen Objektes aufrufen: Argumentenliste beachten

4.2.7. Math Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
dient für mathematische Operationen

Erzeugung:
keine, da vom Browser implementiert

Zugriff:

```
[ var Wert = ] Math.eigenschaft
[ var Wert = ] Math.methode()
```

Wert ist NaN, wenn Operation nicht erfolgreich ist

Beispiel für Erzeugung einer Zufallszahl:

```
function ZufallsWertGanzzahligErzeugen1(MaxWert) // Zufallszahl von      inklusive 0
                                                 //                      bis inklusive MaxWert
                                                 //            erzeugen als ganze Zahl
                                                 // liefert 0 wenn MaxWert 0 ist
// MaxWert:        wenn Gleitkomma, so vor Verwendung nach gannzahlig konvertiert
//                 wenn negativ, so Absolutwert verwendet und wenn dann Gleitkomma, dann zu ganzer Zahl
//                                  Bsp.:                -22.5 wird zu  22.5 wird zu 22
//                                                 jedoch NICHT -22.5 wird zu -23   wird zu 23
{
        var ZufallsZahl=0;

        // zu positiv
        MaxWert=Math.abs(MaxWert);

        // zu ganzer Zahl
        MaxWert=Math.floor(MaxWert);

        if (MaxWert > 0)
        {
                ZufallsZahl=Math.random();      // ab eventuell einschliesslich 0 bis unter 1, Gleitkomma
                                                //  eventuell: falls nicht inklusive 0, so doch 0
                                                //          letztendlich erzegbar durch nachfolgendes Runden

                if (ZufallsZahl > 0)            // Bps. MaxWert 3 Zufallszahl   0,412345
                                                //                              0,612345
                                                //                              0,0015545
                                                //                              0,29456
                {
                        // Kommaverschiebung
                        while ((ZufallsZahl * 10) <= MaxWert)
                        {ZufallsZahl = ZufallsZahl * 10;}       // 0,412345    0,612345 1,5545  2,9456

                        ZufallsZahl= Math.round(ZufallsZahl);   // Kommastellen abschneiden durch runden
                                                                // ab inklusive 0,5 aufwärts, sonst abwärts
                                                                // 0            1       2       3

                        // falls Runden einen Wert > MaxWert ergab, so den Wert begrenzen auf MaxWert
```


```
                                        if (ZufallsZahl > MaxWert)
                                        {ZufallsZahl  = MaxWert;}
                                }
                        }

                        return ZufallsZahl;
        }

        function ZufallsWertGanzzahligErzeugen(MaxWert)   // Zufallszahl von       inklusive 0
                                                          //                       bis inklusive MaxWert
                                                          //              erzeugen als ganze Zahl
                                                          // liefert 0 wenn MaxWert 0 ist
        // MaxWert:             wenn Gleitkomma, so vor Verwendung nach gannzahlig konvertiert
        //                      wenn negativ, so Absolutwert verwendet und wenn dann Gleitkomma, dann zu ganzer Zahl
        //                                      Bsp.:                   -22.5 wird zu  22.5 wird zu 22
        //                                                              jedoch NICHT -22.5 wird zu -23   wird zu 23
        {
                        var ZufallsZahl=0;

                        // zu positiv
                        MaxWert=Math.abs(MaxWert);

                        // zu ganzer Zahl
                        MaxWert=Math.floor(MaxWert);

                        if (MaxWert > 0)
                        {
                                ZufallsZahl=Math.random();      // ab eventuell einschliesslich 0 bis unter 1, Gleitkomma
                                                                //  eventuell: falls nicht inklusive 0, so doch 0
                                                                //              letztendlich erzegbar durch nachfolgendes Runden

                                if (ZufallsZahl > 0)
                                {
                                        var Zahler   = MaxWert;                 // Maxwert bis 0
                                        var Gefunden = false;
                                        var Divisor  = MaxWert + 1;             // Bsp. Maxwert = 3, also von 0 bis 3 liefern

                                        do
                                        {
                                                if (ZufallsZahl >= (Zahler / Divisor))          //          >= 3/4 so 3
                                                {                                               //          >= 2/4 so 2
                                                        ZufallsZahl = Zahler;                   //          >= 1/4 so 1
                                                        Gefunden    = true;                     //          >= 0/4 so 0
                                                }

                                                Zahler--;               // ab MaxWert bis 0
                                        }
                                        while (         (!Gefunden)
                                                &&      (Zahler >= 0)
                                              );
                                }
                        }

                        return ZufallsZahl;
        }
```

Beispiel für Ermittlung des größten ganzzahligen Teiler:

```
        function AufGanzeZahlPruefen(Wert)
        // liefert true wenn ganzzahlig
        {return (Math.ceil(Wert) == Wert);} // nächste ganze Zahl über Wert, oder Wert wenn Wert bereits ganz ist

        function GanzeZahl_GroesstenGanzahligenTeilerErmitteln(GanzerWert)
        // GanzerWert immer >= 0
        // liefert 0 wenn GanzerWert nicht ganzzahlig ist
        // liefert sonst ab 1, aber wenn GanzerWert 0 ist so wird 0 geliefert
        {
                        var ReturnWert = 0;

                        // prüfen ob GanzwertWert > 0 ist
                        if (GanzerWert > 0)
```


```
		{
			// > 0

			// prüfen ob GanzerWert ganzzahlig ist
			if (AufGanzeZahlPruefen(GanzerWert))
			{
				// ist ganzzahlig, also größten ganzzahligen Teiler ermitteln

				var Teiler	= 1;
				var GroessterTeiler = 1;

				do
				{
					// nächsten Teiler einstellen
					Teiler++; // ab 2, da ganze Zahl durch 1 immer teilbar ist

					// prüfen ob Gleitkomma-Division eine ganze Zahl ergab
					if (AufGanzeZahlPruefen(GanzerWert / Teiler))
					{
						// ergab ganze Zahl

						// prüfen ob neuer Teil größer ist als letzter erkannter Teiler
						if (GroessterTeiler < Teiler)
						{
							// größer als merken
							GroessterTeiler = Teiler;
						}
					}
				}
				while (Teiler < (GanzerWert-1));		// Division durch sich selbst ergibt 1
										//  1 ist aber schon der Initwert
										//  von GroessterTeiler

				ReturnWert=GroessterTeiler;
			}
		}

		return ReturnWert;
	}
```

Beispiel für Ermittlung des größten gemeinsamen Teiler zweier ganzer Zahlen:

```
	function ermittle_ggt_rekursiv(ganze_zahl_1, ganze_zahl_2)	// ganze_zahl_1 >= ganze_zahl_2
	{
		if  (ganze_zahl_2 ==0)
		{return ganze_zahl_1;}
		else
		{return ermittle_ggt_rekursiv((ganze_zahll_2,(ganze_zahl_1 % ganze_zahl_2));}
	}
```

Beispiel für Konvertieren zu ganzzahlig:

```
	function KonvertiereZu_GanzeZahl(Wert)
	{
		var ReturnWert = 0;

		if (Wert != 0)
		{
			var Faktor = 1;

			if (Wert < 0)
			{
				Faktor = -1;		// Ergebnis ist negativ
				Wert   = -1 * Wert;	// nur positiven Werten wegen Math.floor()
							//	Bsp:	3.1456   wird zu 3
							//		also Kommaabschneidung
							//		-22,4567 wird zu -23 da -23 < -22
							//		also keine Kommaabschneidung
							//	aber 22,4567 wird zu 22
							//		also Kommaabschneidung
			}
```


```
                ReturnWert = Faktor * Math.floor(Wert);    // Bsp.: 0,01 wird zu 0
                                                           // Bsp.: 1,01 wird zu 1
        }

        return ReturnWert;
}
```

Beispiel für Konvertierung zu ganzzahlig teilbar:

```
function KonvertiereZu_GanzzahligTeilbar(Wert)
// wenn Wert > 0, so liefert die größte ganze Zahl <= Wert, die durch 2 teilbar ist Bsp:  5, so  4 geliefert
// wenn Wert < 0, so liefert die größte ganze Zahl >= Wert, die durch 2 teilbar ist Bsp: -5, so -4 geliefert
{
        // Wert zu ganze Zahl konvertieren, da Modulo den Rest als Gleitkomma liefert
        //        Bsp.:    5  %  2   ergibt Rest 1
        //                 4,3 %  2   ergibt nicht Rest 0
        var ReturnWert = KonvertiereZu_GanzeZahl(Wert);

        if (ReturnWert != 0)            // Division von 0  durch ReturnWert ungleich Null ergibt immer 0
        {
                if (ReturnWert >= 2)  // 1 ist nicht ganzzahlig teilbar
                {
                        while ((ReturnWert % 2) != 0)
                        {ReturnWert--;}        // Bsp.: 5 % 2 Rest 1, dann 4 % 2 Rest 0, also   4 geliefert
                }
                else
                {
                        if (ReturnWert <= -2) // -1 ist nicht ganzzahlig teilbar
                        {
                                while ((ReturnWert % 2) != 0)
                                {ReturnWert++;}
                                                // Bsp.: -5 % 2 Rest 1, dann -4 % 2 Rest 0, also -4 geliefert
                        }
                }
        }

        return ReturnWert;
}
```

Beispiel für ganzahlige Division:

```
function GanzZahlDivision(Dividend, Divisor)          // wenn Divisor == 0, so 0 geliefert
{
        var ReturnWert = 0;

        // Division durch Null ausschliessen
        if (Divisor != 0)
        {
                // Vorzeichen des Ergebnisses ermitteln UND nur positiven Werten wegen Math.floor()
                //                Bsp:      3.1456   wird zu 3      also Kommaabschneidung
                //                          -22,4567 wird zu -23 da -23 < -22
                //                                   also keine Kommaabschneidung
                //                          aber 22,4567 wird zu 22 also Kommaabschneidung

                var Faktor = 1;

                if (Dividend < 0)
                {
                        Faktor   = -1 * Faktor;          // Ergebnis ist negativ
                        Dividend = -1 * Dividend;        // und positiv setzen
                }

                if (Divisor < 0)
                {
                        Faktor  = -1 * Faktor; // falls Dividend < 0, so Ergebnis positiv; sonst negativ
                        Divisor = -1 * Divisor;                  // und positiv setzen
                }

                ReturnWert = Faktor * Math.floor(Dividend / Divisor);
        }
```


```
        return ReturnWert;
    }
```

Beispiel für Runden auf n Nachkommastellen:

```
    function runden(numerischer_wert, nachkommastellen)
    {
        var faktor = Math.pow(10, nachkommastellen)
        return (Math.round(numerischer_wert * faktor) / faktor);
    }
```

Eigenschaften:

es wird Gross-und Klein-Schreibweise unterschieden

.E	eulersche Zahl
.LN2	Logarithmus zur Basis 2
.LN10	Logarithmus zur Basis 10
.LOG2E	Logarithmus von E (eulersche Zahl) zur Basis 2
.LOG10E	Logarithmus von E (eulersche Zahl) zur Basis 10
.PI	Zahl Pi
.SQRT1_2	Quadratwurzel von 0,5 also von 1 dividiert durch Quadratwurzel von 2
.SQRT2	Quadratwurzel von 2

Methoden:

.abs(zahl)	Absolutbetrag
.acos(zahl)	Arcus Cosinus im Bogenmass liefert Wert von 0 bis PI
.asin(zahl)	Arcus Sinus im Bogenmass liefert Wert von -PI/2 bis +PI/2
.atan(zahl)	Arcus Tangens im Bogenmass liefert Wert von -PI/2 bis +PI/2
.atan2(zahl1,zahl2)	Arcus Tangens im Bogenmass zahl1 y-Position Mittelpunkt zahl2 x-Position Mittelpunkt liefert Wert von -PI bis +PI
.ceil(zahl)	nächste ganze Zahl oberhalb zahl ermitteln Bsp: ceil(1.1) ergibt 2 ceil(1) ergibt 1
.cos(zahl)	Cosinus im Bogenmass liefert Wert von -1 bis +1
.exp(zahl)	liefert den Wert von E hoch zahl
.floor(zahl)	nächste ganze Zahl unterhalb zahl ermitteln Bsp: floor(1.1) ergibt 1 floor(1) ergibt 1
.log(zahl)	liefert den Logarithmus zur Basis E der zahl (natürlicher Logarithmus)
.max(zahl1, zahl2 ....)	größte Zahl aller Zahlen liefern
.min(zahl1, zahl2 .....)	kleinste Zahl aller Zahlen liefern
.pow(basis, exponent)	liefert die Potenz von basis hoch exponent
.random()	Zufallszahl liefern 0<= zufallszahl <1
.round(zahl)	zahl auf ganz runden; ab >=5 aufwärts Bsp: 0,5 ergibt 1
.sin(zahl)	Sinus im Bogenmass liefert Wert von -1 bis +1
.sqrt(zahl)	Quadratwurzel ziehen zahl >=0 liefert 0 wenn zahl < 0
.tan(zahl)	Tangens im Bogenmass liefert Wert von -1 bis +1

4.2.8. Number Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
dient zur Instanzierung eines privaten numerischen Objektes
wird verwendet bei numerischen Werten wie z.B. Unendlich

Nicht alle numerischen Methoden sind im Number Script-Objekt implementiert: Es gibt objektübegreifende Methoden, die auch für das Number Script-Objekt zutreffen.

Erzeugung:

```
    var Zeiger = new Number(javascript_ausdruck_oder_wert)

                javascript_ausdruck_oder_wert  muss numerisch liefern/sein
```

Beispiel:

```
    var x=Number("three");  // liefert NaN
```


Zugriff:

zeiger_auf_number_objekt.eigenschaft
zeiger_auf_number_objekt.methode()

NUMBER.eigenschaft
NUMBER.methode()

Konvertierung nach numerisch:

per Methode Number()
Syntax:

[var Wert =] Number(Zeiger)

Zeiger auf zu konvertierende Instanz

Wert NaN wenn Konvertierung nicht möglich ist

Konvertierung von **Hexaziffern** nach numerisch ist **nicht** per Number() möglich:

```
var hexaziffern_feld = [                    // anstelle von new Array()
                          "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                          "A", "B", "C", "D", "E", "F"
                        ];

function dezimalwert_zu_hexaziffern(dezimal_wert)
{
        var high = Math.floor(dezimal_wert / 16);
        var low  = Math.floor(dezimal_wert - (high * 16));

        return(hexaziffern_feld[high] + hexaziffern_feld[low]);
}
```

Ziffern-Zeichenkettenwert nach numerisch und dabei Dezimalkomma zu Dezimalpunkt umwandeln:

Es wird angenommen, dass genau ein Komma vorkommt.

```
function punkt_zu_komma(zeichenkette)
{
        var pos_komma = zeichenkette.indexOf(",");
        var funktionswert;

        if(pos_komma == -1)
        {
                if (zeichenkette.indexOf(".") == -1)
                { funktionswert = parseInt(zeichenkette);}
                else
                { funktionswert = parseFloat(zeichenkette):}
        }
        else
        {
                funktionswert = parseFloat(
                                        zeichenkette.substring(0, pos_komma)
                                      + "."
                                      + zeichenkette.substring(pos_komma + 1, zeichenkette.length)
                                    );
        }

        return funktionswert;
}
```

Beispiel für Erzeugung einer Zufallszahl:

```
function ZufallsWertGanzzahligErzeugen1(MaxWert)  // Zufallszahl von      inklusive 0
                                                  //                      bis inklusive MaxWert
                                                  //            erzeugen als ganze Zahl
                                                  // liefert 0 wenn MaxWert 0 ist
// MaxWert:         wenn Gleitkomma, so vor Verwendung nach gannzahlig konvertiert
//                  wenn negativ, so Absolutwert verwendet und wenn dann Gleitkomma, dann zu ganzer Zahl
//                                  Bsp.:                   -22.5 wird zu  22.5 wird zu 22
//                                                  jedoch NICHT -22.5 wird zu -23   wird zu 23
{
        var ZufallsZahl=0;
```


```
		// zu positiv
		MaxWert=Math.abs(MaxWert);

		// zu ganzer Zahl
		MaxWert=Math.floor(MaxWert);

		if (MaxWert > 0)
		{
			ZufallsZahl=Math.random();	// ab eventuell einschliesslich 0 bis unter 1, Gleitkomma
							//  eventuell: falls nicht inklusive 0, so doch 0
							//		letztendlich erzegbar durch nachfolgendes Runden

			if (ZufallsZahl > 0)		// Bps. MaxWert 3 Zufallszahl	0,412345
							//				0,612345
							//				0,0015545
							//				0,29456
			{
				// Kommaverschiebung
				while ((ZufallsZahl * 10) <= MaxWert)
				{ZufallsZahl = ZufallsZahl * 10;}	// 0,412345	0,612345 1,5545  2,9456

				ZufallsZahl= Math.round(ZufallsZahl);	// Kommastellen abschneiden durch runden
									// ab inklusive 0,5 aufwärts, sonst abwärts
									// 0		1	2	3

				// falls Runden einen Wert > MaxWert ergab, so den Wert begrenzen auf MaxWert
				if (ZufallsZahl > MaxWert)
				{ZufallsZahl  = MaxWert;}
			}
		}

		return ZufallsZahl;
}

function ZufallsWertGanzzahligErzeugen(MaxWert)	// Zufallszahl von	inklusive 0
							//			bis inklusive MaxWert
							//		erzeugen als ganze Zahl
							// liefert 0 wenn MaxWert 0 ist
// MaxWert:		wenn Gleitkomma, so vor Verwendung nach gannzahlig konvertiert
//			wenn negativ, so Absolutwert verwendet und wenn dann Gleitkomma, dann zu ganzer Zahl
//				Bsp.:		-22.5 wird zu  22.5 wird zu 22
//				jedoch NICHT -22.5 wird zu -23   wird zu 23
{
		var ZufallsZahl=0;

		// zu positiv
		MaxWert=Math.abs(MaxWert);

		// zu ganzer Zahl
		MaxWert=Math.floor(MaxWert);

		if (MaxWert > 0)
		{
			ZufallsZahl=Math.random();	// ab eventuell einschliesslich 0 bis unter 1, Gleitkomma
							//  eventuell: falls nicht inklusive 0, so doch 0
							//		letztendlich erzegbar durch nachfolgendes Runden

			if (ZufallsZahl > 0)
			{
				var Zahler   = MaxWert;		// Maxwert bis 0
				var Gefunden = false;
				var Divisor  = MaxWert + 1;	// Bsp. Maxwert = 3, also von 0 bis 3 liefern

				do
				{
					if (ZufallsZahl >= (Zahler / Divisor))	//		>= 3/4 so 3
					{					//		>= 2/4 so 2
						ZufallsZahl = Zahler;		//		>= 1/4 so 1
						Gefunden    = true;		//		>= 0/4 so 0
					}
```


```
                                        Zahler--;               // ab MaxWert bis 0
                                }
                                while (     (!Gefunden)
                                        &&  (Zahler >= 0)
                                      );
                        }
                }

                return ZufallsZahl;
        }
```

Beispiel für Ermittlung des größten ganzzahligen Teiler:

```
        function AufGanzeZahlPruefen(Wert)
        // liefert true wenn ganzzahlig
        {return (Math.ceil(Wert) == Wert);} // nächste ganze Zahl über Wert, oder Wert wenn Wert bereits ganz ist

        function GanzeZahl_GroesstenGanzahligenTeilerErmitteln(GanzerWert)
        // GanzerWert immer >= 0
        // liefert 0 wenn GanzerWert nicht ganzzahlig ist
        // liefert sonst ab 1, aber wenn GanzerWert 0 ist so wird 0 geliefert
        {
                var ReturnWert = 0;

                // prüfen ob GanzwertWert > 0 ist
                if (GanzerWert > 0)
                {
                        // > 0

                        // prüfen ob GanzerWert ganzzahlig ist
                        if (AufGanzeZahlPruefen(GanzerWert))
                        {
                                // ist ganzzahlig, also größten ganzzahligen Teiler ermitteln

                                var Teiler      = 1;
                                var GroessterTeiler = 1;

                                do
                                {
                                        // nächsten Teiler einstellen
                                        Teiler++; // ab 2, da ganze Zahl durch 1 immer teilbar ist

                                        // prüfen ob Gleitkomma-Division eine ganze Zahl ergab
                                        if (AufGanzeZahlPruefen(GanzerWert / Teiler))
                                        {
                                                // ergab ganze Zahl

                                                // prüfen ob neuer Teil größer ist als letzter erkannter Teiler
                                                if (GroessterTeiler < Teiler)
                                                {
                                                        // größer als merken
                                                        GroessterTeiler = Teiler;
                                                }
                                        }
                                }
                                while (Teiler < (GanzerWert-1));                // Division durch sich selbst ergibt 1
                                                                                //  1 ist aber schon der Initwert
                                                                                //  von GroessterTeiler

                                ReturnWert=GroessterTeiler;
                        }
                }

                return ReturnWert;
        }
```

Beispiel für Ermittlung des größten gemeinsamen Teiler zweier ganzer Zahlen:

```
        function ermittle_ggt_rekursiv(ganze_zahl_1, ganze_zahl_2)      // ganze_zahl_1 >= ganze_zahl_2
        {
                if  (ganze_zahl_2 ==0)
```


```
		{return ganze_zahl_1;}
		else
		{return ermittle_ggt_rekursiv((ganze_zahll_2,(ganze_zahl_1 % ganze_zahl_2));}
	}
```

Beispiel für Konvertieren zu ganzzahlig:

```
	function KonvertiereZu_GanzeZahl(Wert)
	{
		var ReturnWert = 0;

		if (Wert != 0)
		{
			var Faktor = 1;

			if (Wert < 0)
			{
				Faktor = -1;			// Ergebnis ist negativ
				Wert   = -1 * Wert;		// nur positiven Werten wegen Math.floor()
								//		Bsp:	3.1456   wird zu 3
								//			also Kommaabschneidung
								//			-22,4567 wird zu -23 da -23 < -22
								//			also keine Kommaabschneidung
								//		aber 22,4567 wird zu 22
								//			also Kommaabschneidung
			}

			ReturnWert = Faktor * Math.floor(Wert);	// Bsp.: 0,01 wird zu 0
								// Bsp.: 1,01 wird zu 1
		}

		return ReturnWert;
	}
```

Beispiel für Konvertierung zu ganzzahlig teilbar:

```
	function KonvertiereZu_GanzzahligTeilbar(Wert)
	// wenn Wert > 0, so liefert die größte ganze Zahl <= Wert, die durch 2 teilbar ist Bsp:  5, so  4 geliefert
	// wenn Wert < 0, so liefert die größte ganze Zahl >= Wert, die durch 2 teilbar ist Bsp: -5, so -4 geliefert
	{
		// Wert zu ganze Zahl konvertieren, da Modulo den Rest als Gleitkomma liefert
		//		Bsp.:	5  %  2   ergibt Rest 1
		//			4,3 %  2   ergibt nicht Rest 0
		var ReturnWert = KonvertiereZu_GanzeZahl(Wert);

		if (ReturnWert != 0)		// Division von 0  durch ReturnWert ungleich Null ergibt immer 0
		{
			if (ReturnWert >= 2)  // 1 ist nicht ganzzahlig teilbar
			{
				while ((ReturnWert % 2) != 0)
				{ReturnWert--;}		// Bsp.: 5 % 2 Rest 1, dann 4 % 2 Rest 0, also   4 geliefert
			}
			else
			{
				if (ReturnWert <= -2) // -1 ist nicht ganzzahlig teilbar
				{
					while ((ReturnWert % 2) != 0)
					{ReturnWert++;}
								// Bsp.: -5 % 2 Rest 1, dann -4 % 2 Rest 0, also -4 geliefert
				}
			}
		}

		return ReturnWert;
	}
```

Beispiel für ganzahlige Division:

```
	function GanzZahlDivision(Dividend, Divisor)		// wenn Divisor == 0, so 0 geliefert
	{
		var ReturnWert = 0;
```


```
		// Division durch Null ausschliessen
		if (Divisor != 0)
		{
			// Vorzeichen des Ergebnisses ermitteln UND nur positiven Werten wegen Math.floor()
			//		Bsp:	3.1456   wird zu 3	also Kommaabschneidung
			//			-22,4567 wird zu -23 da -23 < -22
			//				also keine Kommaabschneidung
			//			aber 22,4567 wird zu 22 also Kommaabschneidung

			var Faktor = 1;

			if (Dividend < 0)
			{
				Faktor   = -1 * Faktor;		// Ergebnis ist negativ
				Dividend = -1 * Dividend;	// und positiv setzen
			}

			if (Divisor < 0)
			{
				Faktor  = -1 * Faktor; // falls Dividend < 0, so Ergebnis positiv; sonst negativ
				Divisor = -1 * Divisor;			// und positiv setzen
			}

			ReturnWert = Faktor * Math.floor(Dividend / Divisor);
		}

		return ReturnWert;
	}
```

Beispiel für Runden auf n Nachkommastellen:

```
	function runden(numerischer_wert, nachkommastellen)
	{
		var faktor = Math.pow(10, nachkommastellen)
		return (Math.round(numerischer_wert * faktor) / faktor);
	}
```

Eigenschaften:

es wird Gross- und Kleinschreibung unterschieden

MAX_VALUE	maximaler numerischer endlicher Wert > 0 oder < 0, den Javascript zulässt: 1.79E+308	
	Werte über MAX_VALUE sind unendlich (Number.POSITIVE_INFINITY bzw. Number.NEGATIVE_INFINITY)	
	kann als Wert zugewiesen werden	
	nur lesen	
MIN_VALUE	minimaler numerischer endlicher Wert > 0, den Javascript zulässt: 5E-324	
	Werte unter MIN_VALUE sind immer 0	
	kann als Wert zugewiesen werden	
	nur lesen	
NaN	nicht numerischer Wert (Not a Number)	
	kann als Wert zugewiesen werden	
	Bsp: var Wert = Number.NaN;	
	Hinweis: Methoden .parseFloat() und .parseInt() liefern NaN, wenn Parameter nicht numerisch ist.	
	Vergleich mit NaN ist nicht zulässig: Dafür ist die Methode .isNaN() zu verwenden.	
	nur lesen	
NEGATIVE_INFINITY	entspricht negativ unendlich	
	kann als Wert zugewiesen werden	
	Hinweis für numerische Operationen	
	Multiplikation	Wert mal unendlich ergibt unendlich
		unendlich mal unendlich ergibt unenendlich
		0 * undendlich ergibt NaN
		NaN * unendlich ergibt NaN
	Division	Wert durch unendlich ergibt 0
		unendlich durch Wert ergibt unendlich
		unendlich durch undendlich ergibt NaN
		Division durch 0 nicht erlaubt
	Vorzeichen	wird wie üblich ermittelt
	nur lesen	

POSITIVE_INFINITY entspricht positiv unendlich
kann als Wert zugewiesen werden
Hinweis für numerische Operationen

Multiplikation	Wert mal unendlich ergibt unendlich
	unendlich mal unendlich ergibt unenendlich
	0 * undendlich ergibt NaN
	NaN * unendlich ergibt NaN
Division	Wert durch unendlich ergibt 0
	unendlich durch Wert ergibt unendlich
	unendlich durch undendlich ergibt NaN
	Division durch 0 nicht erlaubt
Vorzeichen	wird wie üblich ermittelt

nur lesen

Methoden:

Soll ein Literal verwendet werden, das **nicht** mindestens 1 Dezimalkomma enthält (Dezimalpunkt ist das Komma), so muss unmittelbar vor .methode() 1 Blank kodiert werden, damit das Literal nicht als Bezeichner einer Referenz interpretiert wird.

Beispiel:		
	77.77.toExponential()	liefert "7.777e+1"
	77 .toExponential());	liefert "7.7e+1"
	77.toExponential());	liefert Syntaxfehler
	1250 .toPrecision(2))	liefert "1.3e+3"

.toExponential() String liefern, der den Wert in Exponential-Darstellung enthält
auch bei numerischem Literal z.B. "10.2345678e+13"
IE ab 5.5, NS ab 6.x

Beispiel: var num=77.1234	
num.toExponential()	liefert "7.71234e+1"
num.toExponential(4))	liefert "7.7123e+1"
num.toExponential(2))	liefert "7.71e+1"
77.1234.toExponential())	liefert "7.71234e+1"
77 .toExponential())	liefert "7.7e+1"

.toFixed() String liefern, der den Wert in Festkomma-Darstellung enthält
auch bei numerischem Literal z.B. "10.2345678"
IE ab 5.5, NS ab 6.x

Beispiele: var num=10.1234	
num.toFixed())	liefert "10"
num.toFixed(4))	liefert "10.1234"
num.toFixed(2))	liefert "10.12"
0.124.toFixed(2)	liefert "0.12"
0.125.tofixed(2)	liefert "0.13"
0.126.tofixed(2)	liefert "0.13"
0.045.toFixed(2)	liefert "0.05"
1234.567**89**.toFixed(4)	liefert "1234.567**9**"

.toPrecision() liefert String, der die Nachkommastellen enthält
auch bei numerischem Literal z.B. "10.2345678"
IE ab 5.5, NS ab 6.x

Beispiele: var num=5.123456	
num.toPrecision())	liefert "5.123456"
num.toPrecision(4))	liefert "5.123"
num.toPrecision(2))	liefert "5.1"
num.toPrecision(1))	liefert "5"
1250 .toPrecision(2))	liefert "1.3e+3"
1250 .toPrecision(5))	liefert "1250.0"
1234.56789.toPrecision(2)	liefert "1.2e+3"
1234.56789.toPrecision(9)	liefert "1.234.56789" entspricht also e+0

objektübergreifende Methoden beim Number Script-Objekt:

isFinite() ermittelt, ob Wert einer Instanz numerisch endlich ist
Syntax:
[var Wert =] isFinite(Zeiger);

Zeiger auf Instanz vom Number-Objekt oder Ausdrucksergebnis
auch Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
Number.NaN

Wert true, so Wert der Instanz ist endlich
false, so Wert der Instanz ist unendlich
immer bei
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
Number.NaN

isNaN() ermittelt, ob Wert einer Instanz nicht numerisch ist
Syntax:
[var Wert =] isNaN(Zeiger);

Zeiger auf Instanz vom Number-Objekt oder Ausdrucksergebnis
auch Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY
Number.NaN

Wert true, so Wert der Instanz ist nicht numerisch
immer bei Number.NaN
false, so Wert der Instanz ist numerisch
immer bei
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINITY

Beispiele:

```
var Kette ="10.23";
var Wert  =10.23;

var Ergebnis1=parseFloat(Kette);
var Ergebnis2=floatValue=parseFloat(Wert);

alert(isNaN(Ergebnis1);
alert(isNaN(Ergebnis2);
```

Hinweis: Methoden parseFloat und parseInt liefern NaN, wenn Parameter nicht numerisch ist
Bsp: var zahl=parseFloat("text"); ifNaN(zahl) liefert true

Number() konvertiert eine Instanz zu einem numerischen Wert (Number-Objekt-Wert)
Syntax:
[var Wert =] Number(Zeiger)

Zeiger 1 auf zu konvertierende Instanz
auch Date-Objekt

Wert wenn Zeiger 1 ein Date-Objekt ist, so Wert ist die Anzahl der
Millisekunden seit dem 1.1.1970 0 Uhr Weltzeit
(UTC), wobei Weltzeit genau GMT entspricht
wenn Datum vor dem 1.1.1970 0 Uhr Weltzeit so Wert negativ
NaN wenn Konvertierung nicht möglich ist

Beispiel:

```
var DatumJetzt = new Date();
alert (Number(DatumJetzt));
```

parseFloat() String oder Literal parsen und nach Floating-point umwandeln
können enthalten
Vorzeichen + und -
Ziffern 0 bis 9
Dezimalkomma als Punkt
e oder E
Blanks (werden ignoriert)

falls andere Zeichen enthalten sind, gilt:
String bzw. Literal bis vor die erste fehlerhaften
Stelle geparst und dann konvertiert

Erfolg der Konvertierung ist per Methode isNaN() zu prüfen
Syntax:
[var Wert =] parseFloat(Kette)

Kette String oder Literal

Wert Floating-point
NaN wenn String bzw. Literal bereits mit erstem Zeichen
fehlerhaft

Beispiele:

gültiges Literal
parseFloat("3.14")
parseFloat("314e-2")
parseFloat("0.0314E+2")

gültiger String
var x = "3.14"
parseFloat(x)

ungültiges Literal
parseFloat("FF2")

parseInt() String oder Literal parsen und nach Integer umwandeln
können enthalten
Vorzeichen + und -
Ziffern 0 bis 9, aber keine Vornull(en)
Blanks (werden ignoriert)
eventuelle Buchstaben A bis F

falls andere Zeichen enthalten sind, gilt:
String bzw. Literal bis vor die erste fehlerhaften Stelle geparst und dann konvertiert
Erfolg der Konvertierung ist per Methode isNaN() zu prüfen
Syntax:
[var Wert2 =] parseInt(Kette[, Wert1])

Kette String oder Literal
muss passend zu Wert1 kodiert sein
Hinweis: oktale Escape-Sequenzen sind in ab Javascript 1.5 im Netscape 6.x deprecated

Wert1 Integer
Zahlenbasis
10 für dezimal
8 für oktal
16 für hexadezimal
(Wert1 kann Buchstaben A bis F enthalten)
Standard: 0

Wert2 Integer
NaN wenn String bzw. Literal bereits mit erstem Zeichen fehlerhaft
wenn Wert1 0 ist, so
wenn Kette beginnt mit
"0x", so hexadezimale Konvertierung
"0" und nicht "0x", so oktale Konvertierung
nicht mit "0" oder "0x", so dezimale Konvertierung

Beispiel:
gültige Literale:

```
parseInt("F", 16)
parseInt("17", 8)
parseInt("15", 10)
parseInt(15.99, 10)
parseInt("FXX123", 16)
parseInt("1111", 2)
parseInt("15*3", 10)
parseInt("17")
parseInt("0x7", 16)
parseInt("0x7")
parseInt("0")
parseInt("0x11", 16)
parseInt("0x11", 0)
parseInt("0x11")
```

ungültige Literale:

```
parseInt("F")
parseInt("Hello", 8)
parseInt("0x7", 10)
parseInt("FFF", 10)
parseInt('002')
```


.toString() String liefern, der den Wert enthält, wobei Nachkommastellen automatisch festgelegt werden
und gerundet wird
auch bei numerischem Literal z.B. "10.2345678"
siehe Script-Objekt Number
Syntax:
[var Kette =] zeiger_auf_number_objekt.toString([Wert])

Wert2 Integer, 2 bis 36
Zahlenbasis
10 für dezimal
8 für oktal
16 für hexadezimal
(Wert1 kann Buchstaben A bis F enthalten)
wenn nicht kodiert, so autmatische Festlegung
wenn Kette beginnt mit
"0x", so hexadezimale Basis
"0" und nicht "0x", so oktale Basis
nicht mit "0" oder "0x", so dezimale Basis

Beispiel

```
var howMany=10;
howMany.toString()      liefert "10"
45 .toString()          liefert "45"
```

.valueOf() numerischen Wert liefern
auch bei numerischem Literal z.B. "10.2345678"
siehe Script-Objekt Number
Syntax:
[var Kette =] zeiger_auf_number_objekt.valueOf()

4.2.9. Object JScript-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache und stellt das Basis-Objekt aller JScript-Objekte (außer seiner selbst) dar.
Es repräsentiert ein JScript-Objekt schlechthin und ohne speziellen Datentyp. Es repräsentiert **nicht** das Objekt object im Browser, welches das HTML-Element OBJECT widerspiegelt.

JScript-Objekte gibt es mit spezialisierten Objektklassen (Objekttypen), die einen sepziellen Datentyp verwalten,

z.B.
Objekt arguments
Objekt Array
Objekt Boolean
Objekt Date
Objekt Enumerator
Objekt Error
Objekt Function
Objekt Math
Objekt Number
Objekt object (nicht Objekt obect des Internet Explorer für das HTML-Tag OBJECT)
Objekt RegExp
Objekt String
Objekt var

Anhand dieser speziellen JScript-Objekte sind Beschreibungen von Daten möglich.

Das Script-Objekt object umfasst also alle Eigenschaften und Methoden, die in **fast allen** JScript-Objekten und deren Instanzen implementiert sind.

Ein Instanz eines JScript-Objektes kann per Anweisung new erzeugt werden (siehe dort):

Achtung: Der Browser kann nur Objekte verarbeiten, die er kennt, z.B. ein Array-Objekt, dessen Methoden und Eigenschaften dem Browser bekannt sind. Bei einem privaten Objekt müssen alle Eigenschaften und Methoden aufs Script-Komponenten bestehen.

Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype
erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
Nur für Objekte, die aus einem vordefiniertem Objekt abgeleitet sind, existiert die Eigenschaft .prototype .

Punktnotation zum Zeiger:
zeiger.**prototype**.eigenschaft
zeiger.**prototype**.methode
Erweiterung eines Script-Objektes per
bezeichner_script_objekt.**protoype**.eigenschaft

bezeichner_script_objekt.**protoype**.methode

Eigenschaften und Methoden müssen auf Script-Elemente **basieren**, denn nur letztere kennt die Scriptmaschine des Browsers.

Eine Referenz auf eine Objekt-Instanz ist auch per Anweisungen this und with möglich (siehe dort).

Prinzipiell dürften andere Scripthersteller wegen der Script-Kompabilität die gleichen Script-Objekte und deren Eigenschaften und Methoden wie in JScript ünterstützen.

Eigeschaften:

.constructor

Bezeichner einer JScript-Objektklasse (Objekttyp) oder eines privaten Konstruktors
Anwendung: Ermittlung der Objektklasse/Konstruktors eines abgeleiteten Objektes
Ableitung aus der Objektklasse
per Anweisung new mit der Objektklasse als Konstruktor benötigt (siehe dort)
nicht bei Script-Objekt Math möglich
Ableitung aus privatem Konstruktor
per Anweisung new, die den privaten Konstruktor verwendet
JScript-Objektklassen sind z.B.
Array
Boolean
Date
Function

Beispiel 1 für Ableitung aus einem JScript-Objekt:

```
var Kette = new String("Hi");
alert( (Kette.constructor == String));          // liefert "true"
alert( (Kette.constructor == "String"));        // liefert "false"
```

Beispiel 2 für Ableitung anhand privaten Konstruktors:

```
function TestFunktion()
{alert("Hallo");}

var ZeigerAufFunktion = new TestFunktion(); // bewirkt Ausführung von TestFunktion() also auch von alert()

// ZeigerAufFunktion();          // nicht möglich und bringt Fehlermeldung wegen fehlernder Instanz,
                                 //          da keine Ableitung vom JScript-Objekt Function
                                 // Kodierung ohne ()  bringt keinen Fehler, da als Variablendeklaration erkannt

alert(ZeigerAufFunktion.constructor == TestFunktion);          // true
alert(TestFunktion.constructor == TestFunktion);               // false
```

.propertyIsEnumerable

prüfen ob Stringwert in einem Objekt als Menge von String-Elementen enthalten ist
und ob das Objekt mit der Anweisung for in verarbeitet werden kann

Beispiel:

```
var Feld = new Array("Apfel", "Banane", "Zitrone");
alert( (Feld.propertyIsEnumerable("Apfel"));          // true
```

.prototype

Zeiger auf den Prototyp-Bereich im Objekt, der per Prototyping erweitert wird
Objekt ist entweder Script-Objekt oder bereits instanziertes Objekt:
innerhalb eines Konstruktors ist .prototype nicht zu kodieren
Prototyping eines Objektes verändert den Umfang der Standard-Methoden und -Eigenschaften zum Objekt zur Laufzeit der Scriptmaschine.
Script-Objekte können nur erweitert werden.
Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
Nur für Objekte, die aus einem vordefiniertem Objekt abgeleitet sind, existiert die Eigenschaft .prototype .
siehe auch .isPrototypeOf() und .hasOwnProperty()

Beispiel für Erweiterung des Script-Objektes Array, das die Eigenschaft .prototype besitzt:

```
function MaximumErmitteln( )
{
        var max = this[0];      // this referenziert die Objekt-Instanz, in dem die Methode vorhanden ist,
                                //          also Variable Feld

        for (var i = 1; i < this.length; i++)
        {
                if (max < this[i])
                {max = this[i];}
```


```
		}

		return max;
	}

	// neue Methode per Prototyping hinzufügen zum JScript-Objekt Array
	Array.prototype.NeueArrayMethode = MaximumErmitteln;		// ohne () kodieren !

	// Feld vom Array-Typ erzeugen mit der Methode .NeueArrayMethode()
	var Feld = new Array(1, 2, 3, 4, 5, 6);		// 6 numerische Elemente

	// neue Methode des JScript-Objektes Array aufrufen
	alert(Feld.NeueArrayMethode());
```

Beispiel für private Datenstruktur-Objekt mit Methode anhand einer privaten Konstruktor-Methode:
Es wird die Eigenschaft .prototype **nicht** erzeugt !

```
	<HTML>
	<SCRIPT  LANGUAGE="JavaScript">
	<!--
	function datenstruktur_ausgeben()
	{
		with (document)
		{	write(person.vorname + " " + person.nachname + "<BR>");
			write(person.strasse  + " " + person.nummer  + "<BR>");
			write(person.plz      + " " + person.ort     + "<BR>");
			//	Erika Mustermann
			//	Musterstrasse 1
			//	.....
		}

		for (i in person)
		{	document.write(i + ": " + person[i] + "<BR>");
			//	vorname: Erika
			//	nachname: Mustermann
			//	.....
		}
	}

	function datenstruktur_erzeugen(vorname, nachname, strasse, nummer, plz, ort, methode)
	{
		this.vorname = vorname;
		this.nachname = nachname;
		this.strasse = strasse;
		this.nummer = nummer;
		this.plz = plz;
		this.ort = ort;
		this.methode = methode;
	}

	person = new datenstruktur_erzeugen
			(	"Erika",			// Vorname
				"Mustermann",			// Nachnahme
				"Musterstrasse",		// Strasse
				"1",				// Nummer
				"10000",			// PLZ
				"Musterstadt",			// Ort
				datenstruktur_ausgeben		// ohne () kodieren
			);

	// alternativ auch kodierbar:
	// var person = {	vorname:"Erika",		// Vorname
	//			nachname:"Mustermann",		// Nachnahme
	//			strasse:"Musterstrasse",	// Strasse
	//			nummer:"1",			// Nummer
	//			plz:"10000",			// PLZ
	//			ort:"Musterstadt"		// Ort
	//			methode:datenstruktur_ausgeben// Ausgabemethode ohne () kodieren
	//		};

	// Achtung: Eigenschaft .prototype wird leider nicht (automatisch) erzeugt und ist somit nicht anwendbar !
```


```
alert(person.vorname + "\n" + person.methode);          // person.methode ohne () kodieren !
// -->
</SCRIPT>
</HTML>
```

Methoden:

escape() kodiert einen String oder ein Literal in das Unicode-Format
URI (Uniform Resource Identifiers) werden nicht kodiert
Unicode-Format: kann von jedem Browser verarbeitet werden
folgende Zeichen werden dargestellt im Format %XX
Leerzeichen
Punkt, Komma etc. (alle Satzzeichen)
nicht ASCII-Zeichen
XX ist Hexadezimal-Ziffernfolge
Bsp.: Leerzeichen als %20
folgende Zeichen werden dargestellt im Format %uXXXX
Zeichen des erweiterten Zeichensatzes ab inklusive 256
XX ist Hexadezimal-Ziffernfolge

Beispiel für Einbindung einer Suchmaschine:

```
var markierter_text=document.selection.createRange().text;
        // oder var markierter_text=prompt('Suchbegriff: ');
var suchmaschinen_url='http:// .....';
var suchmaschinen_parameter='.....';

if (markierter_text)
{location.HREF=suchmaschinen_url + suchmaschinen_parameter + escape(markierter_text);}
else
{location.HREF=suchmaschine_url; }
```

Beispiel für altavista: suchmaschinen_url 'http://altavista.de/'
suchmaschinen_parameter 'cgi-bin/query?pg=q&what=web&q='

eval() parsen und, falls fehlerfrei, sofortiges Ausführen eines Scriptcodes

Beispiel 1:

```
if (VarUserAgent != '')
{
    for ( var i=0; i < 10; i++)
    {
        eval(    'if (VarUserAgent.indexOf('"'   + i + '.") != -1)'
                + '{VarBrowser_Version_Haupt = ' + i + ';}'
            );
    }
}
```

Beispiel 2:

```
function DatumHolen(Zeiger)
{
    var Kette = "Heute ist: ";
    Kette += Zeiger.getDate()       + "/";
    Kette += (Zeiger.getMonth() + 1) + "/";
    Kette += Zeiger.getYear();

    return(Kette);
}

var Kette="Date";

eval(   "var Jetzt = new " + Kette + "();" // zur Laufzeit erzeugen
        "alert(DatumHolen(Jetzt));"     // Parameter ist zur Laufzeit bekannt
    );
```

.hasOwnProperty() prüfen ob zum Objekt eine Eigenschaft vorhanden ist, jedoch leider **nicht** aus Prototyping einer per new erzeugten Instanz des JScript-Objektes
Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
siehe auch .prototype

Beispiel:

```
function MaximumErmitteln( )
{
    var max = this[0];      // this referenziert die Objekt-Instanz, in dem die Methode vorhanden ist,
```


```
                                        //          also Variable Feld

                for (var i = 1; i < this.length; i++)
                {
                        if (max < this[i])
                        {max = this[i];}
                }

                return max;
        }

        // neue Methode per Prototyping hinzufügen zum JScript-Objekt Array
        Array.prototype.NeueArrayMethode = MaximumErmitteln;        // ohne () kodieren !

        // Feld vom Array-Typ erzeugen mit der Methode .NeueArrayMethode()
        var Feld = new Array(1, 2, 3, 4, 5, 6);                     // 6 numerische Elemente
                                                                    // Es wird die Eigenschaft .prototype erzeugt.

        alert( (Feld.prototype.hasOwnProperty("NeueArrayMethode"));  // leider false
        alert( (Array.prototype.hasOwnProperty("NeueArrayMethode ")); // true
```

.isPrototypeOf() prüfen ob Objekt per new erzeugt wurde, also eine Instanz eines anderen JScript-Objektes ist
Für private Objekte, die per new-Anweisung mit privatem Konstruktor erzeugt wurden, wird leider **nicht** die Eigenschaft .prototype erzeugt ! Prototyping kann also nur innerhalb des Konstruktors erfolgen und nicht nachträglich per Eigenschaft .prototype .
siehe auch .prototype

Beispiel:

```
var Variable = new RegExp();
alert( (RegExp.prototype.isPrototypeOf(Variable));          // true.
```

.toLocaleString() Wert eines Objektes in System-lokale Einstellungen umwandeln
System-lokale Einstellung z.B. Ländereinstellung, Uhrzeitformat
Konvertierung: Analog wie z.B. bei Exel, das die lokalen Einstellungen ausliest.
nur anwenden für Anzeige des konvertierten Wertes:
Berechnungen mit dem Wert immer unkonvertiert vollziehen, da sämtliche Berechnungsfunktionen **nur** das interne, also unkonvertierte Format kennen (**nicht** analog zu Exel)
Wenn das Objekt ein Feld ist, also Feldelemente in lokale Einstellungen konvertiert werden sollen, dann wird ein String geliefert, der die Feldelemente in der Reihenfolg im Feld und diese getrennt durch **dasjenige** Trenner-Zeichen laut **lokale** Einstellungen enthält.
Wenn das Objekt eine Date-Objekt ist, so wird der Wert von dem Objekt in den lokalen Datumeinstellungen geliefert. Dabei sind nur Jahreszahlen von 1601 bis 9999 zulässig. Andere Jahresangaben werden nicht konvertiert.
Wenn das Objekt ein Number-Objekt ist, so werden die lokalen Einstellungen zum Zahlenformat geliefert.
Wenn das Objekt ein JScript-Objekt object ist, so werden nur dann lokale Einstellungen berücksichtigt, insoweit das Objekt diese berücksichtigen kann. Ein String wird immer geliefert.

Beispiel für Konvertierung eines Feld mit Gleitkomma-Werten:

```
var Feld = new Array(6);
var Wert = 3,201,300.20;        // in JScript ist      das Dezimalkomma der Punkt
                                //                     der Tausendertrenner das Komma

// Feld füllen
for( var i = 0; i < 7; i++)
{
        Wert +1 = 10;
        Feld [i] = Wert;
}

alert(Feld.toLocaleString());
```

Beispiel für Konvertierung eines Datums:

```
var Jetzt = new Date();
alert(Jetzt.toLocaleString());
```

toString() Wert eines Objektes in einen String umwandeln
wenn Objekt ein Script-Objekt Array ist, dann Elemente in der Feldreihenfolge und mit Komma getrennt geliefert

wenn Objekt ein Script-Objekt Booelan ist, dann "true" bzw "false" geliefert (Kleinschreibung !)
wenn Objekt ein Script-Objekt Error ist, dann die Fehlermeldung geliefert (Objekt-Wert ist der Fehlercode)
wenn Objekt ein Script-Objekt Function ist, dann Quellcode der Funktion geliefert
(inklusive Funktionskopf)
wenn Objekt ein Script-Objekt Object ist, so wird geliefert:
"**[object** objectname**]**"
mit objectname als konkreter Bezeichner der Objektklasse bzw. des Objektes
fettgedrucktes wird so geliefert wie angegeben
wenn der Bezug vor .toString() ein Wert laut ID-Attribut oder NAME-Attribut ist, dann wird die Objektklasse geliefert

Beispiel 1:

```
var Wert = 33,33;
alert(Wert.toString(2 ) + ' ' + Wert.toString(16) ) + ' ' + Wert.toString(10));
```

Beispiel 2:

```
<BUTTON ID="ID_Button" .... > .... </BUTTON>
ID_Button.toString() liefert "button"
```

unescape — Dekodiert einen per .escape kodierten String in einen normalen String
URI (Uniform Resource Identifiers) sind nicht per escape()kodierbar

valueOf() — Wert eines JScript-Objektes bzw. Instanz eines JScript-Objektes ermitteln
nicht bei Script-Objekt Math und Script-Objekt Error (auch nicht bei Instanzen dieser Objekte)
Wert ist im Datentyp des Objektes, aber:
wenn Objekt ein Array-Objekt ist, dann Elemente in der Feldreihenfolge und mit Komma getrennt geliefert (identisch in der Wirkung mit .toString() und der Array-Methode .join())
wenn Objekt ein Boolean Objekt ist, dann true bzw false geliefert (kein String !)
wenn Objekt ein Date Objekt ist, dann Anzahl der Millisekunden seit dem 1.1.1970 0 Uhr geliefert
wenn Objekt ein Function Objekt ist, dann Zeiger auf Funktion geliefert
wenn Objekt ein Number Script-Objekt ist, dann numerischen Wert geliefert
wenn Objekt ein Script-Objekt object ist, so Zeiger geliefert
siehe auch .toLocaleString() und .toString()

4.2.10. String Script-Objekt

Dieses Objekt ist eine Komponente der Scriptsprache.
entspricht einer Zeichenkettenvariable
Zeichenkettenkonstante (String-Literal)
ist gleichzeitig Feld der Zeiger aller Zeichenelemente im String:
Index-Operationen beim IE nicht möglich, da Collection und nicht reines Feld

Bsp: document.writeln("Dieser Text ist schmal".small);

Erzeugung:

```
var Zeiger = new String(String-Literal oder Variable oder Ausdruck);
var Zeiger = String-Literal;
```

String-Literal alphanumerische Zeichen
in " " oder ' ' kodieren
Variable muss String sein
Ausdruck muss String liefern

Methode .toString()
.valueOff()
String()
Methoden des Script-Objektes Date

Zugriff:

```
zeiger_auf_string_objekt.eigenschaft
zeiger_auf_string_objekt.methode()
zeiger_auf_string_objekt[Index]
```

zeiger_auf_string_objekt laut Erzeugung
[Index] nicht beim IE
[ ] muss kodiert werden
Index Integer
ab 0 bis .length -1

```
"wert_aus_zeichen".eigenschaft
"wert_aus_zeichen".methode()
"wert_aus_zeichen"[Index]
```

"wert_aus_zeichen" Stringliteral

[Index**]** nicht beim IE
[ ] muss kodiert werden
Index Integer
ab 0 bis .length -1

Index-Zugriff nicht beim IE
Beispiel: var Kette = "Hello"
alert(Kette[0]): // "H"

String-Literal wird wie String-Objekt verwaltet:
Eigenschaften und Methoden des String-Objektes ebenfalls nutzbar, wobei Methoden zum String-Literal , die Zeiger liefern, dann nur Zeiger auf ein String-**Literal** liefern und nicht auf einen String-**Objekt**
dient der vereinfachten Kodierung
vorallem der Parameterbelegung von eval() per Ausdruck:
Wenn Parameter von eval ein Zeiger auf ein String-Objekt ist, so erkennt eval() keinen Ausdruck also z.B. keine Operatoren etc. !
Beispiele:

```
var StringLiteral  = "2 + 2";
var StringObjekt   = new String("2 + 2");

eval(StringLiteral);          // liefert Zeiger auf numerische Variable mit Wert 4
eval("2+2");                  // liefert Zeiger auf numerische Variable mit Wert 4
eval(StringObjekt) ;          // liefert Kette "2 + 2"
```

Eigenschaften:

.length Anzahl der Zeichen im String bzw. String-Literal
Integer, ab 0
wenn 0 so Leerkette
Syntax:
[var Wert =] zeiger_auf_string_objekt.length

zeiger_auf_string_objekt auch direkt kodiertes String-Literal

Beispiel:

```
var StringLiteral ="StringLiteral";
StringLiteral.length          liefert 13
"StringLiteral".length        liefert 13
```

Methoden:

stellen die Stringoperationen dar
siehe auch Operator + für Verkettung

Hinweis: .toString() liefert Zeiger auf Wert des String bzw. Stringliteral
Syntax: [var Zeiger =] zeiger_auf_string_objekt.toString()

zeiger_auf_string_objekt auch direkt kodiertes String-Literal

valueOf() liefert Zeiger auf Wert des String bzw. Stringliteral
Syntax: [var Zeiger =] zeiger_auf_string_objekt.valueOf()

zeiger_auf_string_objekt auch direkt kodiertes String-Literal

String() liefert Zeiger auf String-Wert eines Objektes
ist identisch mit Methode .toString()
Syntax: [var Kette =] String(Zeiger)

Zeiger auch auf Date-Objekt

Kette wenn Zeiger auf Date-Objekt, so Datum als String geliefert (je nach Ländereinstellung des Windows)
Beispiel: Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983

Methoden von IE und NS:

.anchor() HTML-Anker <A> als HTML-Kette erzeugen in der Form "<A NAME='....'>text</A>
und ohne weitere Attribute ID, HREF etc.

Beispiel:

```
var StringLiteral    ="Sichtbarer Ankertext"
var HTML_Kette       = StringLiteral.anchor("WertDesNAMEAttributes");
HTML_Kette           = "Sichtbarer Ankertext".anchor("WertDesNAMEAttributes");
```

entspricht <A NAME="WertDesNAMEAttributes">Sichtbarer Ankertext</A>

```
eval(HTML_Kette);              erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);    zeigt den Anker an und erzeugt Eintrag in der Collection document.anchors
```

.big() HTML-Tag <BIG> erzeugen in der Form <BIG>text</BIG>

Beispiel:

```
var StringLiteral     ="Text der BIG wird"
var HTML_Kette        = StringLiteral.big();
HTML_Kette            = "Text der BIG wird".big();

        entspricht <BIG>Text der BIG wird</BIG>

eval(HTML_Kette);           erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.blink() HTML-Tag <BLINK> erzeugen in der Form <BLINK>text</BLINK>

Beispiel:

```
var StringLiteral     ="Text der BLINK wird"
var HTML_Kette        = StringLiteral.blink();
HTML_Kette            = "Text der BLINK wird".blink();

        entspricht <BLINK>Text der BLINK wird</BLINK>

eval(HTML_Kette);           erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.bold() HTML-Tag <B> erzeugen in der Form <B>text</B>

Beispiel:

```
var StringLiteral     ="Text der BOLD wird"
var HTML_Kette        = StringLiteral.bold();
HTML_Kette            = "Text der BOLD wird".bold();

        entspricht <B>Text der BOLD wird</B>

eval(HTML_Kette);           erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.charAt() Zeichen liefern unter Nutzung des Indexes

Beispiel 1:

```
var StringLiteral ="Text"
StringLiteral.charAt(0)     liefert "T"
StringLiteral.charAt(5)     liefert Leerkette
"Text".charAt(0)            liefert "T"
```

Beispiel 2 Zeichenkette auf Wertebereich der Zeichen prüfen:

```
<HTML>
<HEAD>
 <SCRIPT  LANGUAGE="JavaScript">
<!--
        function pruefe_zeichenkette(zeichenkette, wertebereich)
        {       //  wertebereich ist Zeichenkette z.B. "0123456789 -+/.,()"

                var rueckgabewert = true;       // Annahme: Zeichenkette hat NUR Zeichen
                                                //                                  aus dem Wertebereich
                var zeichen_aus _zeichenkette;

                //        Zeichenkette zeichenweise analysieren: Jedes Zeichen mit Wertebereich vergleichen
                for (var i = 0; i < zeichenkette.length; i++)
                 {
                        zeichen_aus_zeichenkette = zeichenkette.charAt(i);

                        if (wertebereich.indexOf(zeichen_aus_zeichenkette) == -1)
                        {
                                rueckgabewert = false;
                                break;      // ungültiges Zeichen gefunden, also abbrechen
                         }
                }

                return rueckgabewert;
        }

        function pruefe_eingabe(zu_pruefende_zeichenkette)
        {
                if (pruefe_zeichenkette(zu_pruefender_zeichenkette, "0123456789 -+/.,()") )
                 {alert("Eingabe ist korrekt !");}
```


```
                        else
                        {alert("Eingabe ist nicht korrekt !"); }
                }
        //-->
        </SCRIPT>
        </HEAD>
        <BODY>
        <FORM>
                Telefon:
                <INPUT  TYPE="text"
                        NAME="Telefon"
                        VALUE=""
                >
                <INPUT  TYPE="button"
                        VALUE="Ueberpruefen"
                        onclick="pruefe_eingabe(this.form.Telefon.value)">
        </FORM>
        </BODY>
        </HTML>
```

.charCodeAt() Unicode des Zeichen liefern unter Nutzung des Indexes
Unicode von 0 bis 65535, wobei
0 bis 127 der ASCII-Zeichensatz ist
0 bis 255 der ISO-Latin-1-Zeichensatz ist

Beispiel:

```
var StringLiteral ="Text";
StringLiteral.charCodeAt(0)             liefert 84 für "T"
StringLiteral.charCodeAt(5)             liefert NaN
"Text".charCodeAt(0)                    liefert 84 für "T"
```

.concat() Verkettung von String-Objekten bzw. String-Literalen zu einem neuen String-Objekt
Alternative: + Operator der Stringverkettung

Beispiel:

```
var StringLiteral1 ="Hallo ";
var StringLiteral2 ="Du ";
var StringLiteral3 ="!";
var Kette =StringLiteral1.concat(StringLiteral2,StringLiteral3) // "Hallo Du !"
var Kette ="Hallo ".concat("Du ","!") // "Hallo Du !"
```

.fixed() HTML-Tag <TT> erzeugen in der Form <TT>text</TT>

Beispiel:

```
var StringLiteral       ="Text der TT wird"
var HTML_Kette      = StringLiteral.fixed();
HTML_Kette          = "Text der TT wird".fixed();
```

entspricht <TT>Text der TT wird</TT>

```
eval(HTML_Kette);               erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.fontcolor() HTML-FONT als HTML-Kette erzeugen in der Form "<FONT COLOR='....'>text</FONT>
und ohne weitere Attribute ID etc.

Beispiel:

```
var StringLiteral       ="Sichtbarer Text"
var HTML_Kette      = StringLiteral.fontcolor("WertDesCOLORAttributes");
HTML_Kette          = "Sichtbarer Text".fontcolor("WertDesCOLORAttributes");
```

entspricht <FONT COLOR="WertDesCOLORAttributes">Sichtbarer Text</FONT>

```
eval(HTML_Kette);               erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.fontsize() HTML-FONT als HTML-Kette erzeugen in der Form "<FONT SIZE='....'>text</FONT>
und ohne weitere Attribute ID etc.

Beispiel:

```
var StringLiteral       ="Sichtbarer Text"
var HTML_Kette      = StringLiteral.fontsize("WertDesSIZEAttributes");
HTML_Kette          = "Sichtbarer Text".fontsize("WertDesSIZEAttributes");
```

entspricht <FONT SIZE="WertDesSIZEAttributes">Sichtbarer Text</FONT>

```
eval(HTML_Kette);          erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.fromCharCode() String-Literal erzeugen aus Folge von Unicoden
Unicode von 0 bis 65535, wobei
0 bis 127 der ASCII-Zeichensatz ist
0 bis 255 der ISO-Latin-1-Zeichensatz ist

Beispiel:

```
String.fromCharCode(65,66,67) // liefert "ABC"
```

.indexOf() Suche einer Teilkette in einem String bzw. Stringliteral und die Startposition der Teilkette als Index im String bzw. Stringliteral liefern
es wird nur das ERSTE Auffinden der Teilkette geliefert
Suche im String erfolgt von links nach rechts
unterscheidet Gross und Klein

Beispiel 1:

```
var StringLiteral ="StringLiteral";
StringLiteral.indexOf("gLi", 2)          liefert 5
StringLiteral.indexOf("gLi")             liefert 5
StringLiteral.indexOf("gLi", 10)         liefert -1
"StringLiteral".indexOf("gLi", 2)        liefert 5
```

Beispiel 2 E-Mail-Adresse auf "@" prüfen:

```
<HTML>
<HEAD>
 <SCRIPT  LANGUAGE="JavaScript">
<!--
        function pruefe_zeichenkette(zeichenkette)
        {
                if (zeichenkette.indexOf('@') = -1)
                {alert("@ fehlt !");}
                else
                 {alert("@ gefunden !");}
        }
//-->
 </SCRIPT>
</HEAD>

<BODY>
 <FORM>
        e-Mail:
        <INPUT   TYPE="text"
                NAME="Mail"
                VALUE=""
        >
        <INPUT   TYPE="button"
                VALUE="Ueberpruefen"
                onclick=" pruefe_zeichenkette (this.form.Mail.value)"
       >
 </FORM>
</BODY>
</HTML>
```

.italics() HTML-Tag <I> erzeugen in der Form <I>text</I>

Beispiel:

```
var StringLiteral      ="Text der I wird"
var HTML_Kette         = StringLiteral.italics();
HTML_Kette             = "Text der I wird".italics();
```

entspricht <I>Text der I wird</I>

```
eval(HTML_Kette);          erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.lastIndexOf() Suche einer Teilkette in einem String bzw. Stringliteral und die Startposition der Teilkette als Index im String bzw. Stringliteral liefern
es wird nur das LETZTE Auffinden der Teilkette geliefert
Suche im String erfolgt von links nach rechts
unterscheidet Gross und Klein

Beispiel:

```
var StringLiteral ="StringLiteral";
StringLiteral.lastIndexOf("t", 2)          liefert 8
StringLiteral.lastIndexOf("t")             liefert 8
StringLiteral.lastIndexOf("t", 10)         liefert -1
"StringLiteral".lastIndexOf("t", 2)        liefert 8
```

.link() HTML-Link <A> als HTML-Kette erzeugen in der Form "<A HREF='....'>text</A>
und ohne weitere Attribute ID, HREF etc.

Beispiel:

```
var StringLiteral      ="Sichtbarer Ankertext"
var HTML_Kette         = StringLiteral.link("WertDesHREFAttributes");
HTML_Kette             = "Sichtbarer Ankertext".link("WertDesHREFAttributes");

        entspricht <A HREF="WertDesHREFAttributes">Sichtbarer Ankertext</A>

eval(HTML_Kette);              erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);    zeigt den Anker an und erzeugt Eintrag in der Collection document.anchors
```

.match() Suche in einem String oder String-literal per RegExp-Objekt (siehe dort Methode .exec())

Beispiel 1:

```
var Kette                      = "The rain in Spain falls mainly in the plain";
var RegExpressionAusdruck      = /ain/i;
var Feld1                      = Kette.match(RegExpressionAusdruck);
var Feld2                      = "The rain in Spain falls mainly in the plain".match(RegExpressionAusdruck);
```

Beispiel 2:

```
var zu_durchsuchende_kette     ="Otto";
var such_muster                =/(\wOtto)/g; //  vom Typ RegExp, also regulärer Ausdruck
var ergebnis_feld              =zu_durchsuchende_kette.match(such_muster);
```

.replace() Suche in einem String oder String-literal per RegExp Objekt (siehe dort Methode .exec())
und gefundenen String ersetzen

Beispiel 1:

```
var Kette                      =
                "The man hit the ball with the bat.\nwhile the fielder caught the ball with the glove.";
var RegExpressionAusdruck      = /The/g;
var Feld1                      = Kette.replace(RegExpressionAusdruck, "A");    // "A" ersetzt "The"
var Feld2                      =
"The man hit the ball with the bat.while the fielder caught the ball with the glove.".replace(
                               RegExpressionAusdruck, "A");    // "A" ersetzt "The"
```

Beispiel 2:

```
function F_ahrenheitTauschenGegen_C_eslius(Kette)
{
        var RegExpressionAusdruck = /(\d+(\.\d*)?)F\b/g;

        return (    Kette.replace(      RegExpressionAusdruck,
                                        function($0,$1,$2) {return((($1-32) * 5/9) + "C");}
                                    )
                );
}
document.write(F_ahrenheitTauschenGegen_C_eslius ("Wasser kristalliert bei 32F und siedet bei 212F."));
```

Beispiel 3:

```
var zu_durchsuchende_kette     ="Otto Waalkes";    // oder RegExp.input="Otto Waalkes"
var such_muster                =/(\wOtto)/g;       // such_muster ist ein regulärer Ausdruck
                                                   //       (Objekt RegExp)
                                                   //  Suchmuster ist Otto, wobei Otto gefunden
                                                   //    werden muss für eine erfolgreiche Suche
                                                   // anstelle von  " ist / zu kodieren
                                                   //  alternativ nicht möglich
                                                   //            RegExp.input="Otto"
                                                   //            dann  kein detailiertes Suchmuster
                                                   // Option g alternativ kodierbar per
                                                   //          RegExp.multiline=true;
var ergebnis_kette             =zu_durchsuchende_kette. replace(such_muster,"Heinrich");
```

.search() Suche in einem String oder String-literal per RegExp-Objekt (siehe dort Methode .exec())

Beispiel 1:

```
var Kette                   = "The rain in Spain falls mainly in the plain.";
var RegExpressionAusdruck   = /falls/i;
var Wert                    = Kette.search(RegExpressionAusdruck);
```

Beispiel 2:

```
var zu_durchsuchende_kette  ="Otto";
var such_muster             =/(\wOtto)/g; //  vom Typ RegExp, also regulärer Ausdruck
var ergebnis                =zu_durchsuchende_kette.search(such_muster);
```

.slice() Teilkette aus String oder Stringliteral liefern als neuen String

Beispiel:

```
var StringLiteral ="StringLiteral";
StringLiteral.length            liefert 13
StringLiteral.slice(3,-5)       liefert "ingLit"   // 13 + (-5) ergibt 8
StringLiteral.slice(3,5)        liefert "ing"
"StringLiteral".slice(3,5)      liefert "ing"
```

.small() HTML-Tag <SMALL> erzeugen in der Form <SMALL>text</SMALL>

Beispiel:

```
var StringLiteral    ="Text der SMALL wird"
var HTML_Kette       = StringLiteral.small();
HTML_Kette           = "Text der SMALL wird".small();
```

entspricht <SMALL>Text der SMALL wird</SMALL>

```
eval(HTML_Kette);          erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.split() String bzw. Stringliteral zerlegen und als Feld der Teilketten liefern
Zerlegung von links nach rechts
Feldelemente-Folge wie Zerlegungsfolge

Beispiel:

```
var StringLiteral ="StringLiteral";
var Feld = StringLiteral.split("r", 2);      Feld enthält 2 Elemente mit "St" und "ingLiteral"
Feld = StringLiteral.split("r", 1);          Feld enthält 1 Elemente mit "St"
Feld = StringLiteral.split("r");             Feld enthält 3 Elemente mit "St" und "ingLite" und "al"
Feld = "StringLiteral".split("r");           Feld enthält 3 Elemente mit "St" und "ingLite" und "al"
```

.strike() HTML-Tag <STRIKE> erzeugen in der Form <STRIKE>text</STRIKE>

Beispiel:

```
var StringLiteral    ="Text der STRIKE wird"
var HTML_Kette       = StringLiteral.strike();
HTML_Kette           = "Text der STRIKE wird".strike();
```

entspricht <STRIKE>Text der STRIKE wird</STRIKE>

```
eval(HTML_Kette);          erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.sub() HTML-Tag <SUB> erzeugen in der Form _{text}

Beispiel:

```
var StringLiteral    ="Text der SUB wird"
var HTML_Kette       = StringLiteral.sub();
HTML_Kette           = "Text der SUB wird".sub();
```

entspricht _{Text der SUB wird}

```
eval(HTML_Kette);          erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.substr() Teilkette aus String oder Stringliteral liefern als neuen String

.substring() Teilkette aus String oder Stringliteral liefern als neuen String

Beispiel:

```
var StringLiteral ="StringLiteral";
StringLiteral.substring(3,3)      liefert ""
StringLiteral.substring(3,5)      liefert "in"
"StringLiteral".substring(3,5)    liefert "in"
```

.sup() HTML-Tag <SUP> erzeugen in der Form ^{text}

Beispiel:

```
var StringLiteral     ="Text der SUP wird"
var HTML_Kette        = StringLiteral.sup();
HTML_Kette            = "Text der SUP wird".sup();

        entspricht <SUP>Text der SUB wird</SUP>

eval(HTML_Kette);             erzeugt Fehler, da HTML-Code von eval nicht ausgeführt werden kann
document.write(HTML_Kette);
```

.toLowerCase() String bzw. Stringliteral nach neuen String kopieren und dort nach Kleinbuchstaben umwandeln

Beispiel:

```
var StringLiteral ="StringLiteral";
StringLiteral.toLowerCase()          liefert "stringliteral"
"StringLiteral".toLowerCase()        liefert "stringliteral"
```

.toUpperCase() String bzw. Stringliteral nach neuen String kopieren und dort nach Grossbuchstaben umwandeln

Beispiel:

```
var StringLiteral ="StringLiteral";
StringLiteral.toUpperCase()          liefert "STRINGLITERAL"
"StringLiteral".toUpperCase()        liefert "STRINGLITERAL"
```

zusätzliche Methoden vom IE:

.localeCompare() Vergleich zweier Strings oder Stringliterale bezüglich ihrer Sortierfolge laut aktuelle Vorgaben zur Sortierungsfolge auf dem PC des Users
nur IE ab 5.5

.toLocaleLowerCase String bzw. Stringliteral nach neuen String kopieren und dort nach Kleinbuchstaben umwandeln laut aktuelle Sprach-Einstellungen der Umgebung auf PC des Users
nur IE ab 5.5
siehe Script-Objekt String

.toLocaleUpperCase() String bzw. Stringliteral nach neuen String kopieren und dort nach Grossbuchstaben umwandeln laut aktuelle Sprach-Einstellungen der Umgebung auf PC des Users
nur IE ab 5.5
siehe Script-Objekt String

4.3. vordefinierte Objekte zum Browser (Auswahl)

4.3.1. Ansatz

4.3.1.1. vordefinierte Objekte in Javascript /JScript

Es können alle Objekte aus Javascript/JScript mit deren Eigenschaften und Methoden in sinnvoller Kombination mit den zum Browser vordefinierten Objekten verwendet werden. Die objekt-übergreifenden, also objekt-unabhängigen Methoden, sind alle nutzbar aber z.T. in den zum Browser vordefinierten Objekten abgewandelt implementiert.

4.3.1.2. Browserfenster und HTML-Dokument (Objekt window und Objekt document)

Ein HTML-Dokument muss im Browserfenster nicht angezeigt werden, wenn das HTML-Element keine sichtbaren HTML-Elemente besitzt. So gesehen kann also ein HTML-Dokument nichts mit dem Browserfenster zu tun haben.

Der Browser kann allerdings nur dann ein HTML-Dokument verwalten, wenn es einem Browserfenster zugeordnet ist. So gesehen muss also ein HTML-Dokument immer in einem Fenster liegen, egal ob das Dokument sichtbar ist oder nicht.

Ein Browserfenster
- verwaltet das HTML-Dokument als Ganzheit (Container)
- verwaltet **nicht** die HTML-Elemente des Dokumentes
- visualisiert das HTML-Dokument
- besitzt weitere Objekte für die Realisierung eines Fensters

Das HTML-Dokument
- verwaltet seine Elemente anhand des HTML-DOM (siehe weiter unten)
- besitzt weitere Objekte und andere Elemente für die Verwaltung

Aus Sicht der gesamten Browserverwaltung wird folgende Hierarchie gebildet:

```
            Fenster
            /     \
      Dokument      andere Objekten, die der Browser benötigt
      /     \
HTML-Elemente   andere Objekte, die das Dokument benötigt
```

Zur Umsetzung der Hierarchie werden Zeiger verwendet:

Zeiger als Eigenschaften des Fensters (Objekt window)

Zeiger auf das HTML-Dokument