

What's New in Javascript 1.2

JavaScript

Version 1.2

Netscape Communications Corporation ("Netscape") and its licensors retain all ownership rights to this document (the "Document"). Use of the Document is governed by applicable copyright law. Netscape may revise this Document from time to time without notice.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL NETSCAPE BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OR DATA, INTERRUPTION OF BUSINESS, OR FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, ARISING FROM ANY ERROR IN THIS DOCUMENT.

The Document is copyright © 1997 Netscape Communications Corporation. All rights reserved.

Netscape and Netscape Navigator are registered trademarks of Netscape Communications Corporation in the United States and other countries. Netscape's logos and Netscape product and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. Other product and brand names are trademarks of their respective owners.

The downloading, export or reexport of Netscape software or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of Netscape software or documentation to the U.S. Government is with restricted rights as described in the license agreement accompanying Netscape software.



Recycled and Recyclable Paper

The Team:

Engineering:

Marketing:

Publications:

Quality Assurance:

Technical Support:

Version 1.0

©Netscape Communications Corporation 1997

All Rights Reserved

Printed in USA

99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

Introduction	15
Feature Summary	15
Event Model	15
Functions	17
Layers	17
Methods	17
Objects	21
Operators	21
Properties	22
Regular Expressions	23
Signed Scripts	23
Statements	24
Style Sheets	24
Miscellaneous	24
Compatibility With Earlier Versions of Navigator	25
Chapter 1 Event Model	27
The event Object	27
Details of the event object	28
Event Capturing	30
Example	32
Chapter 2 Events	33
Click (revised to include new properties)	33
Syntax	34
Parameters	34
Event of	34
Event properties used	34
DbClick	35
Syntax	35

Parameters	35
Event of	35
Event properties used	35
DragDrop	36
Syntax	36
Parameters	36
Event of	36
Event properties used	36
Description	36
KeyDown	37
Syntax	37
Parameters	37
Event of	37
Event properties used	37
Description	38
See also	38
KeyPress	38
Syntax	38
Parameters	38
Event of	38
Event properties used	38
Description	39
See also	39
KeyUp	39
Syntax	39
Parameters	39
Event of	40
Event properties used	40
MouseDown	40
Syntax	40
Parameters	40
Event of	41
Event properties used	41

Description	41
MouseMove	41
Syntax	41
Parameters	42
Event of	42
Event properties used	42
Description	42
See Also	42
MouseOut (revised to include new properties)	42
Syntax	43
Parameters	43
Event of	43
Event properties used	43
MouseOver (revised to include new properties)	43
Syntax	43
Parameters	44
Event of	44
Event properties used	44
MouseUp	44
Syntax	44
Parameters	44
Event of	45
Event properties used	45
Description	45
Move	45
Syntax	46
Parameters	46
Event of	46
Event properties used	46
Resize	46
Syntax	46
Parameters	47
Event of	47

Event properties used	47
Chapter 3 Functions	49
Nesting Functions Within Functions	49
Function Constructor	49
Number	50
Syntax	50
Parameter	50
Description	50
Example	50
String	50
Syntax	51
Parameter	51
Description	51
Example	51
Chapter 4 Layers	53
Chapter 5 Methods	55
Document Method	55
getSelection	55
Navigator Method	56
preference	56
Window Methods	57
back	57
disableExternalCapture	58
enableExternalCapture	58
find	59
forward	60
home	61
moveBy	61
moveTo	62
open (window object)	63
resizeBy	65
resizeTo	66

scrollBy	67
scrollTo	68
stop	68
Shared Methods	69
captureEvents	69
clearInterval	70
handleEvent	71
print	71
releaseEvents	72
routeEvent	72
setInterval	73
setTimeout	74
toString	75
Chapter 6 Objects	77
Creating Objects With Literal Notation	77
Syntax	78
Properties	78
Description	78
Example	78
arguments	78
Number	79
screen	80
Syntax	80
Parameters	80
Property of	80
Properties	80
Methods	81
Event handlers	81
Chapter 7 The String Object	83
charCodeAt	84
Syntax	84
Parameters	84
Method of	84

Example	84
concat	85
Syntax	85
Parameters	85
Method of	85
Description	85
Example	85
fromCharCode	86
Syntax	86
Parameters	86
Method of	86
Description	86
Examples	86
match	87
Syntax	87
Parameters	87
Method of	87
Description	87
Examples	88
replace	88
Syntax	89
Parameters	89
Method of	89
Description	89
Examples	89
search	90
Syntax	90
Parameters	90
Description	90
Example	91
slice	91
Syntax	91
Parameters	91

Method of	92
Description	92
Example	92
split	92
Syntax	93
Parameters	93
Method of	93
Description	93
Examples	93
substr	94
Syntax	94
Parameters	95
Method of	95
Description	95
Example	95
substring	96
Syntax	96
Parameters	96
Method of	96
Description	96
Example	97
Chapter 8 The Array Object	99
Creating Arrays With Literal Notation	100
Syntax	100
Properties	100
Description	100
Example	100
Methods	101
concat	101
slice	102
sort	104

Creating Arrays Under JavaScript 1.2	104
Working With Arrays and Regular Expressions	105
Syntax	105
Parameters	105
Properties and Elements	105
Description	106
Chapter 9 Operators	109
Equality Operators	109
Equality Operators Without LANGUAGE=JavaScript1.2	109
Equality Operators With LANGUAGE=JavaScript1.2	110
Data Conversion	110
delete	112
Syntax	112
Parameters	112
Description	112
Chapter 10 Properties	113
Function Property	113
arity	113
navigator Properties	114
language	114
platform	115
window Properties	116
innerHeight	116
innerWidth	117
locationbar	117
menubar	118
outerHeight	119
outerWidth	120
pageXOffset	120
pageYOffset	121
personalbar	122
scrollbars	123
statusbar	124

toolbar	125
Chapter 11 Regular Expressions	127
Constructing Regular Expressions	128
The Regular Expression Syntax	128
Writing a Regular Expression Pattern	128
Working With Regular Expressions	130
Using Parenthesized Substring Matches	131
Executing a Global Search and Ignoring Case	133
A Complete Example	133
Special Characters Used in Regular Expressions	135
Example Using Special Characters	138
Chapter 12 The RegExp Object	141
Syntax	141
Parameters	141
Properties	141
Methods	142
Description	143
Examples	143
Chapter 13 Regular Expression Object	145
Syntax	145
Parameters	146
Description	146
Properties	147
Methods	148
compile	148
Syntax	148
Parameters	148
Description	149
exec	149
Syntax	149
Parameters	149
Description	149

Examples	151
test	152
Syntax	152
Parameters	152
Description	152
Example	153
Chapter 14 Signed Scripts	155
Recommended Reading	156
Signed Script Requirements	156
ARCHIVE attribute	156
ID Attribute	157
Request Expanded Privileges	158
Sign All Scripts	159
Re-sign Changed Scripts	159
Creating Signed Scripts	160
International Characters in Signed Scripts	160
Targets	161
JavaScript Features Requiring Privileges	162
Example	164
Accessing Expanded Privileges Without Signed Scripts	164
Risks	165
Activating Codebased Principles	165
Error Checking	166
Chapter 15 Statements	167
break	167
Syntax	167
Argument	167
Example	168
See Also	168
continue	168
Syntax	168
Argument	169
Example	169

See Also	169
do while statement	170
Syntax	170
Arguments	170
Example	170
export	170
Syntax	171
Parameters	171
Description	171
See Also	171
import	171
Syntax	171
Parameters	172
Description	172
See Also	172
labeled statement	173
Syntax	173
Arguments	173
Example	173
See Also	173
switch statement	174
Syntax	174
Arguments	174
Example	175
Chapter 16 Style Sheets	177
Chapter 17 Miscellaneous Features	179
Activating JavaScript Commands From the Personal Toolbar	179

This document describes the changes and new features for JavaScript in Navigator 4.0. These features will be included in the new JavaScript Guide at a later date. For additional information about JavaScript, see the “JavaScript Guide.”

- “Feature Summary” provides a summary of the new and changed features.
* indicates a change to an existing feature.
- “Compatibility With Earlier Versions of Navigator” describes how write scripts for different versions of Navigator.

Feature Summary

Event Model

The event model has changed to include a new event object, new events, and event capturing.

- event object — contains properties that describe a JavaScript event, such as the event type and the cursor location at the time of the event. The object is passed as an argument to an event handler when an event occurs.
- event capturing — enables a window, frame, or document to capture an event before the target object (link, button, etc.) gets the event. JavaScript implements event capturing with the following new methods:
 - captureEvents — sets the window or document to capture the specified events.
 - disableExternalCapture — disables external event capturing set by the **enableExternalCapture** method.

- `enableExternalCapture` — allows a window with frames to capture events in pages loaded from different locations (servers).
- `releaseEvents` — sets the window or document to release the specified events.
- `routeEvent` — routes the event from its capturer through its normal event hierarchy.
- `handleEvent` — fires the event handler for the specified event.
- Events have been added or revised.
 - `Click*` — occurs when a user clicks a link or form element.
 - `DbClick` — occurs when a user double-clicks over a link or form element.
 - `DragDrop` — occurs when a user drops an object onto a Navigator window.
 - `KeyDown` — occurs when a user depresses a key.
 - `KeyPress` — occurs when a user presses or holds down a key.
 - `KeyUp` — occurs when a user releases a key.
 - `MouseDown` — occurs when a user depresses a mouse button.
 - `MouseMove` — occurs when a user moves the cursor.
 - `MouseOut*` — occurs when a user moves the cursor out of an object.
 - `MouseOver*` — occurs when a user moves the cursor over an object.
 - `MouseUp` — occurs when a user releases a mouse button.
 - `Move` — occurs when a user or script moves a window or frame.
 - `Resize` — occurs when a user or script resizes a window or frame.

Functions

- You can nest functions within functions.
- Functions can now be created with a function constructor.
- Number — converts a specified object to a number.
- String — converts a specified object to a string.

Layers

Layers, new in Navigator 4.0, let you define overlapping layers of transparent or solid content in a web page. Each layer in HTML has a corresponding layer object that allows you to use JavaScript to manipulate the layer.

For information on using layers, see "Dynamic HTML in Netscape Communicator." Note that this link takes you to a different set of documents.

Methods

Array Methods

- `concat` — joins two arrays and returns a new array.
- `slice` — extracts a section from an array and returns a new array
- `sort*` — now works on all platforms. It no longer converts undefined elements to null and sorts them to the high end of the array.

Document Method

- `getSelection` — returns a string containing the text of the current selection.

Navigator Method

- `preference` — allows a script to get and set certain Navigator preferences, such as enabling or disabling Java.

String Methods

- `charCodeAt` — returns a number specifying the ISO-Latin-1 codeset value of the character at the specified index in a string object.
- `concat` — combines the text of two strings and returns a new string.
- `fromCharCode` — constructs a string from a specified sequence of numbers that are ISO-Latin-1 codeset values.
- `match` — executes a search for a match between a string and a regular expression.
- `replace` — executes a search for a match between a string and a regular expression, and replaces the matched substring with a new substring.
- `search` — tests for a match between a string and a regular expression
- `slice` — extracts a section of an string and returns a new string.
- `split*` — includes the following new features and changes:
 - It can take a regular expression argument, as well as a fixed string, by which to split the object string.
 - It can take a limit count so that it won't include trailing empty elements in the resulting array.
 - If you specify `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag, `string.split(" ")` splits on any run of one or more white space characters including spaces, tabs, line feeds, and carriage returns.
- `substr` — returns the characters in a string collecting the specified number of characters beginning with a specified location in the string.
- `substring*` — no longer swaps index numbers when the first index is greater than the second. For this behavior, `LANGUAGE="JavaScript1.2"` must be specified in the `<SCRIPT>` tag.

Window Methods

- `back` — points the Navigator window to the previous URL in the current history list.

- `disableExternalCapture` — disables external capturing of events.
 - `enableExternalCapture` — allows a window with frames to capture events in pages loaded from different locations.
 - `find` — finds the specified text string in the contents of the window.
 - `forward` — points the Navigator window to the next URL in the current history list.
 - `home` — points the Navigator window to the URL specified in the preferences as the user's home page.
 - `moveBy` — moves the window by the specified amount
 - `moveTo` — moves the window to the specified coordinates
 - `open` — includes the following new window features:
 - *alwaysLowered* — forms a new window which floats below the other screen windows, whether it is currently active or not.
 - *alwaysRaised* — forms a new window which floats on top of the other screen windows, whether it is currently active or not.
 - *dependent* — creates a new window as a child of an existing window.
 - *hotkeys* — disables most hotkeys in a new window.
 - *innerHeight* and *innerWidth* — reflect the size of the Navigator window's content area.
- replace *height* and *width*. Both are kept for backwards compatibility.
- *outerHeight* and *outerWidth* — reflect the size of the Navigator window's outside boundary.
 - *screenX* — the distance the new window is placed from the left side of the screen
 - *screenY* — the distance the new window is placed from the top of the screen
 - *titlebar* — creates a window with a titlebar.

- *z-lock* — forms a new window which does not rise above other windows when given focus.
- *resizeBy* — resizes the window by the specified amount
- *resizeTo* — resizes the window to the specified size
- *scrollBy* — scrolls the window by the specified amount
- *scrollTo* — scrolls the window to the specified coordinates
 - *scrollTo* extends the capabilities of *scroll*. *scroll* remains for backward compatibility
- *stop* — stops the current download.

Shared Methods

- *captureEvents* — window and document method. Sets the window or document to capture the specified events.
- *clearInterval* — window and frame method. Cancels *setInterval*.
- *handleEvent* — method of all objects with event handlers. Invokes the handler for the specified event.
- *print* — window and frame method. Prints the contents of the window or frame.
- *releaseEvents* — window and document. Sets the window or document to release the specified events.
- *routeEvent* — window and document method. Passes a captured event along the normal event hierarchy.
- *setInterval* — window and frame method. Repeatedly calls a function or evaluates an expression after a specified number of milliseconds has elapsed.
- *setTimeout** — can now be used to either evaluate an expression or call a function.
- *toString** — converts the object or array to a literal. For this behavior, `LANGUAGE="JavaScript1.2"` must be specified in the `<SCRIPT>` tag.

Objects

- You can create objects using literal notation.
- arguments — includes new properties that provide information about the invoked function.
- Array* — includes the following new features and changes:
 - Literal notation — arrays can now be created using literal notation
 - Under JavaScript 1.2 — when the <SCRIPT> tag includes LANGUAGE="JavaScript1.2," array(1) creates a new array with a[0]=1
 - With regular expressions — When created as the result of a match between a regular expression and a string, arrays have new properties that provide information about the match
- Number* — now produces NaN rather than an error if x is a string that does not contain a well-formed numeric literal.
- screen — contains information about the display screen resolution and colors.
- String* — has new methods as described in String Methods.

Operators

- Equality operators* (= = and !=)
If the <SCRIPT> tag uses LANGUAGE=JavaScript1.2, the equality operators = = and != don't try to convert operands from one type to another, and always compare identity of like-typed operands.
- delete — deletes an object, an object's property, or an element at a specified index in an array.

Properties

Function Property

- `arity` — indicates the number of arguments expected by a function.

Navigator Properties

- `language` — indicates what translation of the Navigator is being used. This property is particularly useful for JAR management.
- `platform` — indicates the machine type for which the Navigator was compiled. This property is particularly useful for JAR management.

Window Properties

- `innerHeight` — specifies the vertical dimension, in pixels, of the window's content area.
- `innerWidth` — specifies the horizontal dimension, in pixels, of the window's content area.
- `locationbar` — object that allows you to show or hide the location bar of the targeted window.
- `menubar` — object that allows you to show or hide the menu bar of the targeted window.
- `outerHeight` — specifies the vertical dimension, in pixels, of the window's outside boundary.
- `outerWidth` — specifies the horizontal dimension, in pixels, of the window's outside boundary.
- `pageXOffset` — specifies the x-position, in pixels, of the window's viewed page.
- `pageYOffset` — specifies the y-position, in pixels, of the window's viewed page.
- `personalbar` — object that allows you to show or hide the personal bar (also known as the directories bar) of the targeted window.

- `scrollbars` — object that allows you to show or hide the scroll bar of the targeted window.
- `statusbar` — object that allows you to show or hide the status bar of the targeted window.
- `toolbar` — object that allows you to show or hide the tool bar of the targeted window.

Regular Expressions

Regular Expressions are patterns used to match character combinations in strings. In JavaScript, you create a regular expression as an object which has methods used to execute a match against a string. You can also pass the regular expression as arguments to the String methods `match`, `replace`, `search`, and `split`. A global `RegExp` object has properties most of which are set when a match is successful, such as `lastMatch` which specifies the last successful match. Finally, the array object has new properties that provide information about a successful match such as the `input` which specifies the original input string against which the match was executed.

This section includes:

- A Description of Regular Expressions
- The Regular Expression Object
- The `RegExp` Object
- String Methods
- Array Properties

Signed Scripts

For additional power and functionality, scripts can now gain access to normally restricted information. This is achieved through signed scripts that request expanded privileges. This new functionality provides greater security than tainting.

Statements

- The `break*` and `continue*` statements can now be used with the new labeled statement.
- `do while` statement — repeats a loop until the test condition evaluates to false.
- `export` — allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.
- `import` — allows a script to import properties, functions, and objects from a signed script which has exported the information.
- labeled statement — allows the program to break outside nested loops or to continue a loop outside the current loop.
- `switch` statement — allows the program to test several conditions easily.

Style Sheets

Using style sheets, you gain finer control over the presentation of your web pages. Navigator 4.0 supports two syntaxes for designing style sheets: Cascading Style Sheets (CSS) and JavaScript style sheets. CSS is the static approach to specifying style and JavaScript is the programmatic approach.

For information on using style sheets, see “Dynamic HTML in Netscape Communicator.” Note that this link takes you to a different set of documents.

Miscellaneous

- You can add a button to the Personal Toolbar that, when clicked, activates a JavaScript command. toolbar. See “Miscellaneous Features”.

Compatibility With Earlier Versions of Navigator

As JavaScript evolves along with Navigator, its capabilities expand greatly. This means that JavaScript written for Navigator 4.0 may work in Navigator 4.0 only. To ensure that users of earlier versions of Navigator avoid problems when viewing pages that use JavaScript 1.2, use the LANGUAGE attribute in the `<SCRIPT>` tag to indicate which version of Javascript you're using. If you use `LANGUAGE="JavaScript1.2"`, you need to be aware of the equality rules described below.

Statements within a `<SCRIPT>` tag are ignored if the browser does not have the level of JavaScript support specified in the LANGUAGE attribute; for example:

- Navigator 2.0 executes code within the `<SCRIPT LANGUAGE="JavaScript">` tag; it ignores code within the `<SCRIPT LANGUAGE="JavaScript1.1">` and `<SCRIPT LANGUAGE="JavaScript1.2">` tags.
- Navigator 3.0 executes code within the `<SCRIPT LANGUAGE="JavaScript">` and `<SCRIPT LANGUAGE="JavaScript1.1">` tags; it ignores code within the `<SCRIPT LANGUAGE="JavaScript1.2">` tag.
- Navigator 4.0 executes code within the `<SCRIPT LANGUAGE="JavaScript">`, `<SCRIPT LANGUAGE="JavaScript1.1">`, and `<SCRIPT LANGUAGE="JavaScript1.2">` tags.

By using the LANGUAGE attribute, you can write general JavaScript that Navigator version 2.0 and higher recognize, and include additional or refined behavior for newer versions of Navigator.

Event Model

JavaScript's event model includes several new events, an *event* object, and the ability to capture and handle events before they reach their intended target. This section contains the following information:

- The event Object
- Event Capturing

New events are described in the Events section.

The event Object

The *event* object contains properties that describe a JavaScript event, and is passed as an argument to an event handler when the event occurs. In the case of a *mousedown* event, for example, the event object contains the type of event (in this case "mousedown"), the x and y position of the cursor at the time of the event, a number representing the mouse button used, and a field containing the modifier keys (Control, Alt, Meta, or Shift) that were depressed at the time of the event. The properties used within the event object vary from one type of event to another. This variation is provided in the individual event descriptions.

JavaScript supports the following events. This document describes the new events, the "JavaScript Guide" describes pre-Navigator 4.0 events.

Abort	MouseDown
Blur	MouseMove
Click (revised)	MouseOut (revised)
Change	MouseOver (revised)
DblClick	MouseUp
DragDrop	Move
Error	Reset
Focus	Resize
KeyDown	Select
KeyPress	Submit
KeyUp	Unload
Load	

Details of the event object

Syntax

`event.propertyName`

Argument of

All event handlers.

Properties

The following properties are specific to an event and are passed with the *event* object. To learn which properties are used by an event, see the “Event object properties used” section of the individual event.

Property	Description
<i>type</i>	String representing the event type.
<i>target</i>	String representing the object to which the event was originally sent.
<i>layerX</i>	Number specifying either the object width when passed with the resize event, or the cursor's horizontal position in pixels relative to the layer in which the event occurred. Note that <i>layerX</i> is synonymous with <i>x</i> .

Property	Description
<i>layerY</i>	Number specifying either the object height when passed with the resize event, or the cursor's vertical position in pixels relative to the layer in which the event occurred. Note that <i>layerY</i> is synonymous with <i>y</i> .
<i>pageX</i>	Number specifying the cursor's horizontal position in pixels, relative to the page.
<i>pageY</i>	Number specifying the cursor's vertical position in pixels relative to the page.
<i>screenX</i>	Number specifying the cursor's horizontal position in pixels, relative to the screen.
<i>screenY</i>	Number specifying the cursor's vertical position in pixels, relative to the screen.
<i>which</i>	Number specifying either the mouse button that was pressed or the ASCII value of a pressed key.
<i>modifiers</i>	String specifying the modifier keys associated with a mouse or key event. Modifier key values are: ALT_MASK, CONTROL_MASK, SHIFT_MASK, and META_MASK.
<i>data</i>	Returns an array of strings containing the URLs of the dropped objects. Passed with the dragdrop event.

Example

The following example uses the *event* object to provide the type of event to the alert message.

```
<A HREF="http://home.netscape.com" onClick='alert("Link got an event: "
+ event.type)'+>Click for link event</A>
```

The following example uses the *event* object in an explicitly called event handler.

```
<SCRIPT>

function fun1(e) {
    alert ("Document got an event: " + e.type);
    alert ("x position is " + e.layerX);
    alert ("y position is " + e.layerY);
    if (e.modifiers & Event.ALT_MASK)
        alert ("Alt key was down for event.");
    return true;
}
```

```

}

document.onmousedown = fun1;

</SCRIPT>

```

Event Capturing

You can now have a window or document capture and handle an event before it reaches its intended target. To accomplish this, the *window*, *document*, and *layer* objects have these new methods:

- `captureEvents`
- `releaseEvents`
- `routeEvent`
- `handleEvent` — not a method of the *layer* object

For example, suppose you wanted to capture all click events occurring in a window.

Note If a window with frames wants to capture events in pages loaded from different locations, you need to use **`captureEvents`** in a signed script and call `enableExternalCapture`.

First, you need to set up the window to capture all Click events:

```
window.captureEvents(Event.CLICK);
```

The argument to **`captureEvents`** is a property of the *event* object and indicates the type of event to capture. To capture multiple events, the argument is a list separated by or (`|`). For example:

```
window.captureEvents(Event.CLICK | Event.MOUSEDOWN | Event.MOUSEUP)
```

Next, you need to define a function that handles the event. The argument *e* is the *event* object for the event.

```

function clickHandler(e) {
    //What goes here depends on how you want to handle the event.
    //This is described below.
}

```

You have four options for handling the event:

- Return `true`. In the case of a link, the link is followed and no other event handler is checked. If the event cannot be canceled, this ends the event handling for that event.

```
function clickHandler(e) {
    return true;
}
```

- Return `false`. In the case of a link, the link is not followed. If the event is non-cancelable, this ends the event handling for that event.

```
function clickHandler(e) {
    return false;
}
```

- Call **routeEvent**. JavaScript looks for other event handlers for the event. If another object is attempting to capture the event (such as the document), JavaScript calls its event handler. If no other object is attempting to capture the event, JavaScript looks for an event handler for the event's original target (such as a button). The **routeEvent** function returns the value returned by the event handler. The capturing object can look at this return and decide how to proceed.

Note When **routeEvent** calls an event handler, the event handler is activated. If **routeEvent** calls an event handler whose function is to display a new page, the action takes place without returning to the capturing object.

```
function clickHandler(e) {
    var retval = routeEvent(e);
    if (retval == false) return false;
    else return true;
}
```

- Call the **handleEvent** method of an event receiver. Any object that can register event handlers is an event receiver. This method explicitly calls the event handler of the event receiver and bypasses the capturing hierarchy. For example, if you wanted all Click events to go to the first link on the page, you could use:

```
function clickHandler(e) {
    window.document.links[0].handleEvent(e);
}
```

As long as the link has an `onClick` handler, the link will handle any click event it receives.

Finally, you need to register the function as the window's event handler for that event:

```
window.onClick = clickHandler;
```

Example

In the following example, the window and document capture and release events:

```
<HTML>
<SCRIPT>

function fun1(e) {
    alert ("The window got an event of type: " + e.type + " and will call routeEvent.");
    window.routeEvent(e);
    alert ("The window returned from routeEvent.");
    return true;
}

function fun2(e) {
    alert ("The document got an event of type: " + e.type);
    return false;
}

function setWindowCapture() {
    window.captureEvents(Event.CLICK);
}

function releaseWindowCapture() {
    window.releaseEvents(Event.CLICK);
}

function setDocCapture() {
    document.captureEvents(Event.CLICK);
}

function releaseDocCapture() {
    document.releaseEvents(Event.CLICK);
}

window.onclick=fun1;
document.onclick=fun2;

</SCRIPT>
...
</HTML>
```


Events

This section describes new and revised events.

All information about an event, such as its type, is now passed to its handler through the event object. The properties passed with the *event* object are listed under the heading "Event properties used." Pre-Navigator 4.0 events that aren't listed here use the *type* and *target* property only.

In Navigator 4.0, a window or document can capture events before they reach their intended target. For more information see "Event Capturing".

Note Navigator 4.0 recognizes mixed-case and lower case use of events and event handlers. For example, you can explicitly call an event handler using either `element.onclick` or `element.onClick`.

Click (revised to include new properties)

Client-side event. Occurs when the user clicks a link or a form element (a Click is a combination of the MouseDown and MouseUp events).

If the event handler returns false, the default action of the object is canceled as follows:

- Buttons — no default action; nothing is canceled
- Radio buttons and checkboxes — nothing is set
- Submit buttons — form is not submitted
- Reset buttons — form is not reset

Syntax

```
onClick="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

document, *Button*, *Checkbox*, *Link*, *Radio*, *Reset*, and *Submit* objects

Event properties used

type indicates a Click event.

target indicates the object to which the event was originally sent.

When a link is clicked, *layerX*, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the Click event occurred.

When a button is clicked, *layerX*, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* are unused.

which represents 1 for a left-mouse click and 3 for a right-mouse click.

modifiers contains the list of modifier keys held down when the Click event occurred.

DbClick

Client-side event. Occurs when the user double-clicks a form element or a link.

Note DbClick is not implemented on the Macintosh.

Syntax

```
onDbClick="handlerText"
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

document, *area*, *link* object

Event properties used

type indicates a DbClick event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the DbClick event occurred.

which represents 1 for a left-mouse double-click and 3 for a right-mouse double-click.

modifiers contains the list of modifier keys held down when the DbClick event occurred.

DragDrop

Client-side event. Occurs when the user drops an object onto the Navigator window, such as dropping a file on the Navigator window.

Syntax

```
onDragDrop="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

window object

Event properties used

type indicates a DragDrop event.

target indicates the object to which the event was originally sent.

data returns an Array of Strings containing the URLs of the dropped objects.

Description

The DragDrop event is fired whenever a system item (file, shortcut, etc.) is dropped onto the Navigator window via the native system's drag and drop mechanism. The normal response for the Navigator is to attempt to load the item into the browser window. If the event handler for the DragDrop event returns true, the browser will load the item normally. If the event handler returns false, the drag and drop is canceled.

KeyDown

Event. Occurs when the user depresses a key.

Syntax

```
onKeyDown="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

document, *Image*, *Link*, and *Textarea* objects

Event properties used

type indicates a KeyDown event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the KeyDown event occurred.

which represents the ASCII value of the key pressed. To get the actual letter, number, or symbol of the pressed key, use the `fromCharCode` method. To set this property when the ASCII value is unknown, use the `charCodeAt` method.

modifiers contains the list of modifier keys held down when the KeyDown event occurred.

Description

A `KeyDown` event always occurs before a `KeyPress` event. If `onKeyDown` returns `false`, no `KeyPress` events occur. This prevents `KeyPress` events occurring due to the user holding down a key.

See also

`KeyPress` and `KeyUp` events

KeyPress

Client-side event. Occurs when the user presses or holds down a key.

Syntax

```
onKeyPress="handlerText"
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

document, *Image*, *Link*, and *Textarea* objects

Event properties used

type indicates a `KeyPress` event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the KeyPress event occurred.

which represents the ASCII value of the key pressed. To get the actual letter, number, or symbol of the pressed key, use the `fromCharCode` method. To set this property when the ASCII value is unknown, use the `charCodeAt` method.

modifiers contains the list of modifier keys held down when the KeyPress event occurred.

Description

A KeyPress event occurs immediately after a KeyDown event only if `onKeyDown` returns something other than `false`. A KeyPress event repeatedly occurs until the user releases the key. You can cancel individual KeyPress events.

See also

KeyDown and KeyUp events

KeyUp

Event. Occurs when the user releases a key.

Syntax

```
onKeyUp="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

document, *Image*, *Link*, and *Textarea* objects

Event properties used

type indicates a KeyUp event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the KeyUp event occurred.

which represents the ASCII value of the key pressed. To get the actual letter, number, or symbol of the pressed key, use the `fromCharCode` method. To set this property when the ASCII value is unknown, use the `charCodeAt` method.

modifiers contains the list of modifier keys held down when the KeyUp event occurred.

MouseDown

Client-side event. Occurs when the user depresses a mouse button.

Syntax

```
onMouseDown="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

Button, *document*, and *Link* objects

Event properties used

type indicates a `MouseDown` event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the `MouseDown` event occurred.

which represents 1 for a left-mouse-button down and 3 for a right-mouse-button down.

modifiers contains the list of modifier keys held down when the `MouseDown` event occurred.

Description

If `onMouseDown` returns `false`, the default action (entering drag mode, entering selection mode, or arming a link) is canceled.

Note Arming is caused by a `MouseDown` over a link. When a link is armed it changes color to represent its new state.

MouseMove

Client-side event. Occurs when the user moves the cursor.

Syntax

```
onMouseMove="handlerText"
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

None

Event properties used

type indicates a MouseMove event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the MouseMove event occurred.

Description

The MouseMove event is sent only when a capture of the event is requested by an object (see “Event Capturing”).

See Also

captureEvents method

MouseOut (revised to include new properties)

Client-side event. Occurs when the user moves the cursor out of an object.

Syntax

```
onMouseOut="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

Area, *Layer*, and *Link* objects

Event properties used

type indicates a MouseOut event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the MouseOut event occurred.

MouseOver (revised to include new properties)

Client-side event. Occurs when the user moves the cursor over an object.

Syntax

```
onMouseOver="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

Area, *Layer*, and *Link* objects

Event properties used

type indicates a *MouseOver* event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the *MouseOver* event occurred.

MouseUp

Client-side event. Occurs when the user releases a mouse button.

Syntax

```
onMouseUp="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

Button, *document*, and *Link* objects

Event properties used

type indicates a `MouseUp` event.

target indicates the object to which the event was originally sent.

layerX, *layerY*, *pageX*, *pageY*, *screenX*, and *screenY* represent the cursor location at the time the `MouseUp` event occurred.

which represents 1 for a left-mouse-button up and 3 for a right-mouse-button up.

modifiers contains the list of modifier keys held down when the `MouseUp` event occurred.

Description

If `onMouseUp` returns `false`, the default action is canceled. For example, if `onMouseUp` returns `false` over an armed link, the link is not triggered. Also, if `MouseUp` occurs over an unarmed link (possibly due to `onMouseDown` returning `false`), the link is not triggered.

Note Arming is caused by a `MouseDown` over a link. When a link is armed it changes color to represent its new state.

Move

Client-side event. Occurs when the user or script moves a window or frame.

Syntax

```
onMove="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

window and *Frame* objects

Event properties used

type indicates a Move event.

target indicates the object to which the event was originally sent.

screenX and *screenY* represent the position of the top-left corner of the window or frame.

Resize

Client-side event. Occurs when a user or script resizes a window or frame.

Syntax

```
onResize="handlerText "
```

Parameters

handlerText is JavaScript code or a call to a JavaScript function.

Event of

window and Frame objects

Event properties used

type indicates a Resize event.

target indicates the object to which the event was originally sent.

width and *height* represent the width and height of the window or frame.

Functions

This section describes the following new functions and features of functions:

- Nested Functions
- Function Constructor
- Number
- String

Nesting Functions Within Functions

You can nest a function within a function. The nested function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the nested function. The nested function is unknown outside of the outer function and cannot be called independently.

Function Constructor

You can now create a function using a constructor function. For example:

```
var f = new Function('x', 'y', "return x * y")
```

Number

Core function. Converts the specified object to a number.

Syntax

`Number (x)`

Parameter

x is any object.

Description

When the object is a *Date* object, `Number` returns a value in milliseconds measured from 01 January, 1970 UTC (GMT), positive after this date, negative before.

Example

The following example converts the *Date* object to a numerical value:

```
<SCRIPT>
d = new Date ("December 17, 1995 03:24:00");
document.write (Number(d) + "<BR>");
</SCRIPT>
```

This prints "819199440000."

String

Core function. Converts the specified object to a string.

Syntax

`String(x)`

Parameter

x is any object.

Description

When the object is a *Date* object, `String` returns a string representation of the date. Its format is: Thu Aug 18 04:37:43 Pacific Daylight Time 1983.

Example

The following example converts the `Date` object to a readable string.

```
<SCRIPT>
D = new Date (430054663215);
document.write (String(D) + " <BR>");
</SCRIPT>
```

This prints "Thu Aug 18 04:37:43 Pacific Daylight Time 1983."

String

Layers

Methods

This section describes the new and revised methods for the following objects:

- Array (links to a different page)
- Document
- Navigator
- String (links to a different page)
- Window

In addition, Shared Methods describes methods used by several objects.

Document Method

getSelection

Client-side method. Returns a string containing the text of the current selection.

Syntax

```
document.getSelection()
```

Method of

document object

Navigator Method

preference

Client-side method. Allows a signed script to get and set certain Navigator preferences.

Note This method must be called in a signed script.

Syntax

To set a preference:

```
navigator.preference(prefName)
```

To set a preference:

```
navigator.preference(prefName, setValue)
```

Parameters

prefName is the name of the preference you want to get or set. Description lists the allowed preferences.

setValue is the value you want to assign to the preference. This can be a string, number, or Boolean.

Method of

navigator object

Description

This method must be used in a signed script that has UniversalPreferencesRead or UniversalPreferencesWrite permission.

With permission, you can get and set the following preferences (additional preferences will be included in future documentation):

Task from Navigator Advanced Preferences	Preference	Value
Automatically load images	<code>general.always_load_images</code>	true or false
Enable Java	<code>security.enable_java</code>	true or false
Enable JavaScript	<code>javascript.enabled</code>	true or false
Enable style sheets	<code>browser.enable_style_sheets</code>	true or false
Enable autoinstall	<code>autoupdate.enabled</code>	true or false
Accept all cookies	<code>network.cookie.cookieBehavior</code>	0
Accept only cookies that get sent back to the originating server	<code>network.cookie.cookieBehavior</code>	1
Disable cookies	<code>network.cookie.cookieBehavior</code>	2
Warn before accepting cookie	<code>network.cookie.warnAboutCookies</code>	true or false

Window Methods

back

Client-side method. Points the Navigator to the previous URL in the current history list; equivalent to the user pressing the Navigator Back button.

Syntax

`windowReference.back()`

Parameters

windowReference is the name of a window object.

Method of

window object

disableExternalCapture

Client-side method. Disables external event capturing set by the **enableExternalCapture** method.

Syntax

```
disableExternalCapture()
```

Method of

window object

Description

See the description for enableExternalCapture method.

enableExternalCapture

Client-side method. Allows a window with frames to capture events in pages loaded from different locations (servers).

Syntax

```
enableExternalCapture( )
```

Method of

window object

Description

Use this method in a signed script requesting UniversalBrowserWrite privileges, and use it before calling the **captureEvents** method.

If additional scripts are seen by Communicator that cause the set of principals in effect for the container to be downgraded, external capture of events will be disabled. Additional calls to **enableExternalCapture** (after acquiring the UniversalBrowserWrite privilege under the reduced set of principals) can be made to enable external capture again.

Example

In the following example, the window is able to capture all Click events that occur across its frames.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="2">
...
function captureClicks() {
    netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
    enableExternalCapture();
    captureEvents(Event.CLICK);
    ...
}
...
</SCRIPT>
```

See also

`disableExternalCapture` method and `captureEvents` method

find

Client-side method. Finds the specified text string in the contents of the specified window.

Syntax

```
windowReference.find(["string"][,true|false][,true|false])
```

Parameters

windowReference is the name of a window object.

string is the text string for which to search.

Returns

true if the string is found; otherwise false.

Method of

window object

Description

When a string is specified, the browser performs a case-insensitive, forward search. If a string is not specified, the method displays the Find dialog box, allowing the user to enter the search string.

The two optional Boolean parameters allow you to specify search options. The first parameter, if true, specifies a case-sensitive search. The second parameter, if true, specifies a backward search. To use either parameter, both must be specified.

forward

Client-side method. Points the Navigator to the next URL in the current history list; equivalent to the user pressing the Navigator Forward button.

Syntax

windowReference.forward()

Parameters

windowReference is the name of a window object.

Method of

window object

home

Client-side method. Points the Navigator to the URL specified in preferences as the user's home page; equivalent to the user pressing the Navigator Home button.

Syntax

```
windowReference.home()
```

Parameters

windowReference is the name of a window object.

Method of

window object

moveBy

Client-side method. Moves the window by the specified amounts.

Syntax

```
windowReference.moveBy(horizontal, vertical)
```

Parameters

windowReference is a valid way of referring to a window.

horizontal is an integer representing the number of pixels by which to move the window horizontally.

vertical is an integer representing the number of pixels by which to move the window vertically.

Method of

window object

Description

To move a window offscreen, call this method in a signed script.

See Also

`moveTo` method

moveTo

Client-side method. Moves the top-left corner of the window to the specified screen coordinates.

Syntax

```
windowReference.moveTo(x-coordinate, y-coordinate)
```

Parameters

windowReference is a valid way of referring to a window.

x-coordinate is an integer representing the left edge of the window in screen coordinates.

y-coordinate is an integer representing the top edge of the window in screen coordinates.

Method of

window object

Description

To move a window offscreen, call this method in a signed script.

See Also

`moveBy` method

open (window object)

Client-side method. Opens a new web browser window. The following provides a description of the **open** method and the new window features. For a complete description of **open**, see **open (window object)** in the “JavaScript Guide.”

Syntax

```
[windowVar = ][window].open("URL", "windowName", ["windowFeatures"])
```

Parameters

windowVar is the name of a new window. Use this variable when referring to a window's properties, methods, and containership.

URL specifies the *URL* to open in the new window.

windowName is the window name to use in the TARGET attribute of a FORM or <A> tag. *windowName* can contain only alphanumeric or underscore (`_`) characters.

windowFeatures is a comma-separated list of any of the following options and values:

```
alwaysLowered [=yes|no|][=1|0]
alwaysRaised [=yes|no|][=1|0]
dependent [=yes|no|][=1|0]
hotkeys [=yes|no|][=1|0]
innerWidth=pixels replaces width
innerHeight=pixels replaces height
outerWidth=pixels
outerHeight=pixels
```

```

screenX=pixels
screenY=pixels
titlebar [=yes|no][=1|0]
z-lock [=yes|no][=1|0]

```

Note Several of these features require the use of signed scripts. This is stated in the feature's description.

Not specifying a chrome part is equivalent to setting the property to no (except for *hotkeys* and *titlebar* which are set to true by default).

alwaysRaised if true, creates a new window that floats on top of other windows, whether it is active or not. This is a secure feature and must be set in signed scripts.

Note How this feature behaves depends on the windowing hierarchy of the platform. For example, on Windows, an *alwaysRaised* Navigator window is on top of all windows in all open applications. On Macintosh, an *alwaysRaised* Navigator window is on top of all Navigator windows, but not necessarily on top of windows in other open applications.

alwaysLowered if true, creates a new window that floats below other windows, whether it is active or not. This is a secure feature and must be set in signed scripts.

Note How this feature behaves depends on the windowing hierarchy of the platform. For example, on Windows, an *alwaysLowered* Navigator window is below all windows in all open applications. On Macintosh, an *alwaysLowered* Navigator window is below all Navigator windows, but not necessarily below windows in other open applications.

dependent if true, creates a new window as a child of the current window. A dependent window closes when its parent window closes. On Windows platforms, a dependent window does not show on the taskbar.

hotkeys if true, disables most hotkeys in a new window that has no menu bar. The security and quit hotkeys remain enabled.

innerWidth specifies the width, in pixels, of the window's content area. To create a window smaller than 100 x 100 pixels, set this feature in a signed script.

Note Replaces *width*. *width* remains for backwards compatibility.

innerHeight specifies the height, in pixels, of the window's content area. To create a window smaller than 100 x 100 pixels, set this feature in a signed script.

Note Replaces *height*. *height* remains for backwards compatibility.

outerWidth specifies the horizontal dimension, in pixels, of the window's outside boundary. To create a window smaller than 100 x 100 pixels, set this feature in a signed script.

outerHeight specifies the vertical dimension, in pixels, of the outside boundary of the window. To create a window smaller than 100 x 100 pixels, set this feature in a signed script.

screenX is the distance the new window is placed from the left side of the screen. To place a window offscreen, set this feature in a signed script.

screenY is the distance the new window is placed from the top of the screen. To place a window offscreen, set this feature in a signed script.

titlebar if true, creates a window with a title bar. To set the titlebar to false, set this feature in a signed script.

z-lock if true, creates a new window that does not rise above other windows when activated. This is a secure feature and must be set in signed scripts.

Note How this feature behaves depends on the windowing hierarchy of the platform. For example, on Windows, a *z-locked* Navigator window is below all windows in all open applications. On Macintosh, a *z-locked* Navigator window is below all Navigator windows, but not necessarily below windows in other open applications.

Method of

window object

resizeBy

Client-side method. Resizes the entire window by moving the window's bottom-right corner by the specified amount.

Syntax

```
windowReference.resizeBy(horizontal, vertical)
```

Parameters

windowReference is a valid way of referring to a window.

horizontal is an integer representing the number of pixels by which to resize the window horizontally.

vertical is an integer representing the number of pixels by which to resize the window vertically.

Method of

window object

Description

To resize a window below a minimum size of 100 x 100 pixels, call this method in a signed script.

See Also

`resizeTo` method

resizeTo

Client-side method. Resizes the entire window to the specified outer height and width.

Syntax

```
windowReference.resizeTo(outerwidth, outerheight)
```

Parameters

windowReference is a valid way of referring to a window.

outerwidth is an integer representing the window's width in pixels.

outerheight is an integer representing the window's height in pixels.

Method of

window object

Description

To resize a window below a minimum size of 100 x 100 pixels, call this method in a signed script.

See Also

`resizeBy` method

scrollBy

Client-side method. Scrolls the viewing area of the window by the given amount.

Syntax

```
windowReference.scrollBy(horizontal, vertical)
```

Parameters

windowReference is a valid way of referring to a window.

horizontal is an integer representing the number of pixels by which to scroll the viewing area horizontally.

vertical is an integer representing the number of pixels by which to scroll the viewing area vertically.

Method of

window object

See Also

`scrollTo` method

scrollTo

Client-side method. Scrolls the viewing area of the window to the specified coordinates, such that the point (x, y) becomes the top-left corner.

Note **scrollTo** extends the capabilities of **scroll**. **scroll** remains for backward compatibility.

Syntax

```
windowReference.scrollTo(x-coordinate, y-coordinate)
```

Parameters

windowReference is a valid way of referring to a window.

x-coordinate is an integer representing the x-coordinate of the viewing area in pixels.

y-coordinate is an integer representing the y-coordinate of the viewing area in pixels.

Method of

window object

See Also

`scrollBy` method

stop

Client-side method. Stops the current download; equivalent to the user pressing the Navigator Stop button.

Syntax

windowReference.stop()

Parameters

windowReference is the name of a window object.

Method of

window object

Shared Methods

captureEvents

Client-side method. Sets the window or document to capture all events of the specified type.

Syntax

objectReference.captureEvents(eventType)

Parameters

objectReference is the name of a window or document object.

eventType is the type of event to be captured. The available event types are listed with the event object.

Method of

window, *document*, and *layer* objects

Description

When a window with frames wants to capture events in pages loaded from different locations (servers), you need to use **captureEvents** in a signed script and precede it with **enableExternalCapture**. For more information and an example, see “enableExternalCapture”.

captureEvents works in tandem with **releaseEvents**, **routeEvent**, and **handleEvent**. For more information, see “Event Capturing”.

clearInterval

Client-side method. Cancels a timeout set with the **setInterval** method.

Syntax

```
clearInterval(intervalID)
```

Parameters

intervalID is a timeout setting that was returned by a previous call to the **setInterval** method.

Method of

Frame object, *window* object

Description

See the description for **setInterval** method.

See Also

setInterval method

handleEvent

Client-side method. Invokes the handler for the specified event.

Syntax

objectReference.handleEvent(event)

Parameters

objectReference is the name of an object.

event is the name of an event for which the specified object has an event handler.

Method of

objects with event handlers

Description

handleEvent works in tandem with captureEvents, releaseEvents, and routeEvent. For more information, see Event Capturing.

print

Client-side method. Prints the contents of the window or frame; equivalent to the user pressing the Navigator Print button.

Syntax

windowReference.print()

frameReference.print()

Parameters

windowReference is the name of a window object.

frameReference is the name of a frame object.

Method of

window and *Frame* objects

releaseEvents

Client-side method. Sets the window or document to release captured events of the specified type, sending the event to objects further along the event hierarchy.

Note If the original target of the event is a window, the window receives the event even if it is set to release that type of event.

Syntax

```
objectReference.releaseEvents(eventType)
```

Parameters

objectReference is the name of a window, document, or layer object.

eventType is the type of event to be captured.

Method of

window, *document*, and *layer* objects

Description

releaseEvents works in tandem with captureEvents, routeEvent, and handleEvent. For more information, see “Event Capturing”.

routeEvent

Client-side method. Passes a captured event along the normal event hierarchy.

Syntax

```
objectReference.routeEvent(event)
```

Parameters

objectReference is the name of a window, document, or layer object.

event is the name of the event to be routed.

Method of

window, *document*, and *layer* objects

Description

If a sub-object (*document* or *layer*) is also capturing the event, the event is sent to that object. Otherwise, it is sent to its original target.

routeEvents works in tandem with captureEvents, releaseEvents, and handleEvent. For more information, see “Event Capturing”.

setInterval

Client-side method. Repeatedly calls a function or evaluates an expression after a specified number of milliseconds has elapsed.

The timeouts continue to fire until the associated window or frame is destroyed or the interval is canceled using the **clearInterval** method.

Syntax

Used to call a function:

```
intervalID=setInterval(function, msec, [arg1, ..., argn])
```

Used to evaluate an expression:

```
intervalID=setInterval(expression, msec)
```

Parameters

intervalID is an identifier that is used only to cancel the function call with the **clearInterval** method.

function is any function.

expression is a string expression or a property of an existing object. The expression must be quoted; otherwise, **setInterval** calls it immediately. For example `setInterval("calcnun(3, 2)", 25)`.

msec is a numeric value, numeric string, or a property of an existing object in millisecond units.

arg1, ..., *argn* are the arguments, if any, passed to *function*.

Method of

Frame object, **window** object

See Also

`clearInterval` and `setTimeout` methods

setTimeout

Client-side method. Calls a function or evaluates an expression after a specified number of milliseconds has elapsed.

The **setTimeout** method calls a function after a specified amount of time. It does not call the function repeatedly. For example, if a **setTimeout** method specifies five seconds, the function is evaluated after five seconds, not every five seconds. For repetitive timeouts, use the **setInterval** method.

setTimeout does not stall the script. The script continues immediately (not waiting for the timeout to expire). The call simply schedules an additional future event.

Syntax

Used to call a function:

```
timeoutID=setTimeout("function", msec, [arg1, ..., argn])
```

Used to evaluate an expression:

```
timeoutID=setTimeout(expression, msec)
```

Parameters

timeoutID is an identifier that is used only to cancel the evaluation with the **clearTimeout** method.

function is any function.

expression is a string expression or a property of an existing object. The expression must be quoted; otherwise, **setTimeout** calls it immediately. For example `setTimeout("calcnum(3, 2)", 25)`.

msec is a numeric value, numeric string, or a property of an existing object in millisecond units.

arg1, ..., *argn* are the arguments, if any, passed to *function*.

Method of

Frame object, *window* object

See Also

clearTimeout in the "JavaScript Guide" and `setInterval` methods

toString

Client-side method. If you specify `LANGUAGE="JavaScript1.2"` in the script tag, using the **toString** method converts objects and arrays to literals. An object literal has the form `{property1:value1, property2:value2, ...}`. An array literal has the form `[element0, element1, ...]`.

Converting to literals allows you to capture a persistent form of the object for debugging or as source for another JavaScript program.

Example

The following example converts *myHonda* to a literal.

```
<SCRIPT LANGUAGE="JavaScript1.2">
myHonda = new Object();
myHonda.color = "red";
myHonda.wheels = 4;
document.write(myHonda.toString());
</SCRIPT>
```

Prints {color:"red", wheels:4}

Without LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, this prints [object Object]

Objects

This section describes the following new objects and changes to existing objects:

- arguments
- event
- Array (links to a different page)
- Number
- the regular expression object
- RegExp
- screen
- String (links to a different page)

In addition, objects can now be created using literal notation.

Creating Objects With Literal Notation

In addition to creating an object using its constructor function, you can create it using literal notation.

Syntax

```
objectName = {property1:value1, property2:value2, ..., propertyn:valuen}
```

Properties

objectName is the name of the new object

propertyn is a property.

valuen is the value assigned to the *propertyn*.

Description

JavaScript interprets objects created through literal notation once only, when the HTML page is loaded.

Example

The following example creates *myHonda* with three properties. Note that the *engine* property is also an object with its own properties.

```
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

arguments

Core object. The *arguments* array object provides information about a function at the time the function is invoked. In previous JavaScript versions, *arguments* provided a list of indexed elements and a length property. In JavaScript 1.2, *arguments* includes these additional properties:

- formal arguments — each formal argument of a function is a property of the arguments array.
- local variables — each local variable of a function is a property of the arguments array.

- caller — a property whose value is the arguments array of the outer function. If there is no outer function, the value is undefined.
- callee — a property whose value is the function reference.

For example, the following script demonstrates several of the *arguments* properties;

```
<SCRIPT>

function b(z) {
    document.write(arguments.z + "<BR>")
    document.write (arguments.caller.x + "<BR>")
    return 99
}

function a(x, y) {
    return b(534)
}

document.write (a(2,3) + "<BR>")

</SCRIPT>
```

This writes:

```
534
2
99
```

534 is the actual parameter to b, so it is the value of arguments.z

2 is a's actual x parameter, so (viewed within b) it is the value of arguments.caller.x.

99 is what a(2,3) returns.

Number

Core object. Number (x) now produces NaN rather than an error if x is a string that does not contain a well-formed numeric literal. For example,

```
x=Number("three");

document.write(x + "<BR>");

prints NaN
```

screen

Client-side object. Contains information about the display screen resolution and colors.

Syntax

```
screen.propertyName
```

Parameters

propertyName is one of the properties listed below.

Property of

None

Properties

Property	Description
<i>availHeight</i>	Specifies the height of the screen, in pixels, minus permanent or semi-permanent user interface features displayed by the operating system, such as the Taskbar on Windows.
<i>availWidth</i>	Specifies the width of the screen, in pixels, minus permanent or semi-permanent user interface features displayed by the operating system, such as the Taskbar on Windows.
<i>height</i>	Specifies the height of the screen in pixels.
<i>width</i>	Specifies the width of the screen in pixels.

Property	Description
<i>pixelDepth</i>	Specifies the number of bits per pixel in the display.
<i>colorDepth</i>	Specifies the number of colors possible to display. The number of colors is found using the color palette if one is available, or using the pixel depth.

Methods

None

Event handlers

None

screen

The String Object

This section describes the new methods for strings. * indicates a change to an existing method.

- `charCodeAt` — returns a number indicating the ISO-Latin-1 codeset value of the character at the given index.
- `concat` — combines the text of two strings and returns a new string.
- `fromCharCode` — constructs a string from the specified sequence of numbers that are ISO-Latin-1 codeset values.
- `match` — used to match a regular expression against a string
- `replace` — used to find a match in a string, and replace the matched substring with a replacement substring.
- `search` — used to test for a match in a string.
- `slice` — extracts a section of an string and returns a new string.
- `split*` — uses a regular expression or a fixed string as its argument to split a string.
- `substr` — returns the characters in a string collecting the specified number of characters beginning with a specified location in the string.

- `substring*` — When the `<SCRIPT>` tag includes `LANGUAGE="JavaScript1.2"`, `substring(x,y)` no longer swaps `x` and `y`.

charCodeAt

Core method. Returns a number indicating the ISO-Latin-1 codeset value of the character at the given index.

Syntax

```
string.charCodeAt([ index ])
```

Parameters

string is any string.

index, an optional argument, is any integer from zero to *string*.length -1, or a property of an existing object. The default value is 0.

Method of

String object

Description

The ISO-Latin-1 codeset ranges from 0 to 255. The first 0 to 127 are a direct match of the ASCII character set.

Example

The following example returns the ISO-Latin-1 codeset value of 65.

```
"ABC".charCodeAt(0)
```

concat

Core method. Combines the text of two strings and returns a new string.

Syntax

```
string1.concat(string2)
```

Parameters

string1 is the first string.

string2 is the second string.

Method of

String object

Description

concat combines the text from two strings and returns a new string. Changes to the text in one string do not affect the other string.

Example

The following example combines two strings into a new string.

```
<SCRIPT>
str1="The morning is upon us. "
str2="The sun is bright."
str3=str1.concat(str2)
document.write(str3)
</SCRIPT>
```

This writes:

The morning is upon us. The sun is bright.

fromCharCode

Core method. Returns a string from the specified sequence of numbers that are ISO-Latin-1 codeset values.

Syntax

```
String.fromCharCode(num1, num2, ..., numn)
```

Parameters

numn is a sequence of numbers that are ISO-Latin-1 codeset values.

Method of

String object

Description

This method returns a string and not a *String* object.

Examples

Example 1. The following example returns the string "ABC".

```
String.fromCharCode(65,66,67)
```

Example 2. The *which* property of the *KeyDown*, *KeyPress*, and *KeyUp* events contains the ASCII value of the key pressed at the time the event occurred. If you want to get the actual letter, number, or symbol of the key, you can use **fromCharCode**. The following example returns the letter, number, or symbol of the *KeyPress* event's *which* property.

```
String.fromCharCode(KeyPress.which)
```

match

Core method. Used to match a regular expression against a string.

Syntax

```
string.match(regexp)
```

Parameters

string is any string.

regexp is the name of the regular expression. It can be a variable name or a literal.

Method of

String object

Description

If you want to execute a global match, or a case insensitive match, include the **g** (for global) and **i** (for ignore case) flags in the regular expression. These can be included separately or together. The following two examples below show how to use these flags with **match**.

Note If you are executing a match simply to find `true` or `false`, use **search** or the regular expression **test** method.

Examples

Example 1. In the following example, **match** is used to find 'Chapter' followed by one or more numeric characters followed by a decimal point and numeric character zero or more times. The regular expression includes the `i` flag so that case will be ignored.

```
<SCRIPT>
str = "For more information, see Chapter 3.4.5.1";
re = /(chapter \d+(\.\d)*)/i;
found = str.match(re);
document.write(found);
</SCRIPT >
```

This returns the array containing

```
Chapter 3.4.5.1,Chapter 3.4.5.1,.1
```

'Chapter 3.4.5.1' is the first match and the first value remembered from `(Chapter \d+(\.\d)*)`. '.1' is the second value remembered from `(\.\d)`.

Example 2. The following example demonstrates the use of the global and ignore case flags with **match**.

```
<SCRIPT>
str = "abcDdcba";
newArray = str.match(/d/gi);
document.write(newArray);
</SCRIPT >
```

The returned array contains D, d.

replace

Core method. Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.

Syntax

```
string.replace(regex, newSubStr)
```

Parameters

string is any string.

regex is the name of the regular expression. It can be a variable name or a literal.

newSubStr is the string to replace the string found with *regex*.

Method of

String object

Description

If you want to execute a global search and replace, or a case insensitive search, include the *g* (for global) and *i* (for ignore case) flags in the regular expression. These can be included separately or together. The following two examples below show how to use these flags with **replace**.

Examples

Example 1. In the following example, the regular expression includes the global and ignore case flags which permits **replace** to replace each occurrence of 'apples' in the string with 'oranges.'

```
<SCRIPT>
re = /apples/gi;
str = "Apples are round, and apples are juicy.";
newstr=str.replace(re, "oranges");
document.write(newstr)
</SCRIPT>
```

This prints "Oranges are round, and oranges are juicy."

Example 2. In the following example, the regular expression is defined in **replace** and includes the ignore case flag.

```
<SCRIPT>
str = "Twas the night before Xmas...";
newstr=str.replace(/xmas/i, "Christmas");
document.write(newstr)
</SCRIPT>
```

This prints "Twas the night before Christmas..."

search

Core method. Executes the search for a match between a regular expression and a specified string.

Syntax

```
string.search(regex)
```

Parameters

string is any string.

regex is the name of the regular expression. It can be a variable name or a literal.

Description

When you want to know whether a pattern is found in a string use **search** (similar to the regular expression test method); for more information (but slower execution) use **match** (similar to the regular expression exec method).

Example

The following example prints a message which depends on the success of the test.

```
function testinput(re, str){
  if (str.search(re))
    midstring = " contains ";
  else
    midstring = " does not contain ";
  document.write (str + midstring + re.source);
}
```

slice

Core method. Extracts a section of an string and returns a new string.

Syntax

```
string.slice(beginslice, [endSlice])
```

Parameters

string is a string.

beginslice is the zero-based index at which to begin extraction.

endSlice is the zero-based index at which to end extraction.

- **slice** extracts up to but not including *endSlice*. `string.slice(1,4)` extracts the second character through the fourth character (characters indexed 1, 2, and 3)
- As a negative index, *endSlice* indicates an offset from the end of the string. `string.slice(2,-1)` extracts the third character through the second to last character in the string.
- If *endSlice* is omitted, **slice** extracts to the end of the string.

Method of

String object

Description

slice extracts the text from one string and returns a new string. Changes to the text in one string do not affect the other string.

Example

The following example uses slice to create a new string.

```
<SCRIPT>

str1="The morning is upon us. "
str2=str1.slice(3,-5)
document.write(str2)

</SCRIPT>
```

This writes:

morning is upon

split

Core method. **split** has the following additions:

- It can take a regular expression argument, as well as a fixed string, by which to split the object string.
- It can take a limit count so that it won't include trailing empty elements in the resulting array.
- If you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, `string.split(" ")` splits on any run of one or more white space characters including spaces, tabs, line feeds, and carriage returns.

Syntax

```
string.split([separator], [limit])
```

Parameters

string is any string.

separator specifies the character or regular expression to use for separating the string.

limit is an optional integer that specifies a limit on the number of splits to be found.

Method of

String object

Description

When found, *separator* is removed from the string and the substrings are returned in an array. If *separator* is omitted, the array contains one element consisting of the entire string.

If *separator* is a regular expression, any included parenthesis cause submatches to be included in the returned array.

Using the optional limit argument, you can avoid including trailing empty elements in the returned array.

Examples

Example 1. Using `LANGUAGE="JavaScript1.2"`, the following script produces `["She", "sells", "seashells", "by", "the", "seashore"]`.

```
<SCRIPT LANGUAGE="JavaScript1.2">
```

```
str="She sells      seashells \nby    the\n seashore"
document.write(str + "<BR>")
a=str.split(" ")
document.write(a)
</SCRIPT>
```

Without `LANGUAGE="JavaScript1.2"`, the above script splits only on single space characters, producing

```
She,sells,,,,seashells, by,,,the ,seashore
```

Example 2. In the following example, **split** looks for zero to many spaces followed by a semi-colon followed by zero to many spaces and, when found, removes them from the string. *nameList* is the array returned as a result of **split**.

```
<SCRIPT>
names = "Harry  Trump ;Fred Barney; Helen  Rigby ;  Bill Abel ;Chris Hand ";
document.write (names + "<BR>" + "<BR>");
re = /\s*;\s*/;
nameList = names.split (re);
document.write(nameList);
</SCRIPT>
```

This prints two lines; the first line prints the original string, and the second line prints the resulting array.

```
Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ;Chris Hand
```

```
Harry Trump,Fred Barney,Helen Rigby,Bill Abel,Chris Hand
```

substr

Core method. Returns the characters in a string beginning at the specified location through the specified number of characters.

Syntax

```
string.substr(start, [length])
```

Parameters

string is any string.

start is the location at which to begin extracting characters.

length, an optional argument, is the number of characters to extract.

Method of

String object

Description

start is a character index. The index of the first character is zero, and the index of the last character is *stringName.length* -1. **substr** begins extracting characters at *start* and collects *length* number of characters.

If *length* is 0 or negative, no string is extracted.

If *length* is omitted, *start* extracts characters to the end of the string.

Example

The following example collects 4 characters beginning with b, and returns "bean".

```
<SCRIPT>
str = "jellybeans";
newstr = str.substr(5,4);
document.write(newstr);
</SCRIPT>
```

substring

If you specify `LANGUAGE="JavaScript1.2"` in the script tag, `substring(x,y)` no longer swaps `x` and `y`.

Syntax

```
string.substring(indexA, [indexB])
```

Parameters

string is any string.

indexA is any integer from zero to *stringName*.length - 1.

indexB, an optional argument, is any integer from zero to *stringName*.length.

Method of

String object

Description

substring behaves as follows:

- It extracts characters from *indexA* up to but not including *indexB*.
- If *indexA* is less than zero, *indexA* is treated as if it were 0.
- If *indexB* is greater than *stringName*.length, *indexB* is treated as if it were *stringName*.length.
- If *indexA* equals *indexB*, substring returns an empty string.
- If *indexB* is omitted, *indexA* extracts characters to the end of the string.

Using `LANGUAGE="JavaScript1.2"` in the `<SCRIPT>` tag,

- If *indexA* is greater than *indexB*, JavaScript produces a runtime error (out of memory).

Without `LANGUAGE="JavaScript1.2"` ,

- If *indexA* is greater than *indexB*, JavaScript returns a substring beginning with *indexB* and ending with *indexA* -1.

Example

Using `LANGUAGE="JavaScript1.2"`, the following script produces a runtime error (out of memory).

```
<SCRIPT LANGUAGE="JavaScript1.2">
str="Netscape"
document.write(str.substring(0,3);
document.write(str.substring(3,0);
</SCRIPT>
```

Without `LANGUAGE="JavaScript1.2"`, the above script prints

Net Net

In the second write, the index numbers are swapped.

substring

The Array Object

This section describes the new features and changes for arrays.

- Literal notation – arrays can now be created using literal notation
- Methods –
 - `concat` joins two arrays and returns a new array.
 - `slice` extracts a section from an array and returns a new array
 - `sort` now works on all platforms, no longer converts undefined elements to `null`, and sorts undefined elements to the high end of the array
- Under JavaScript 1.2 – when the `<SCRIPT>` tag includes `LANGUAGE="JavaScript1.2,"` `array(1)` creates a new array with `a[0]=1`
- With regular expressions – When created as the result of a match between a regular expression and a string, arrays have new properties that provide information about the match

Creating Arrays With Literal Notation

In addition to creating an array using its constructor function, you can create it using literal notation.

Syntax

```
arrayName = [element0, element1, ..., elementn]
```

Properties

arrayName is the name of the new array.

elementn is a list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's length is set to the number of arguments.

Description

JavaScript initializes arrays created through literal notation, when the HTML page is loaded.

Example

The following example creates the `coffees` array with three elements and a length of three.

```
coffees = ["French Roast", "Columbian", "Kona"]
```

Methods

concat

Core method. Joins two arrays and returns a new array.

Syntax

```
arrayName1.concat(arrayName2)
```

Parameters

arrayName1 is the name of the first array.

arrayName2 is the name of the second array.

Method of

Array object

Description

concat does not alter the original arrays, but returns a "one level deep" copy that contains copies of the same elements combined from the original arrays. Elements of the original arrays are copied into the new array as follows:

- Object references (and not the actual object) – **concat** copies object references into the new array. Both the original and new array refer to the same object. If a referenced object changes, the changes are visible to both the new and original arrays.
- Strings and numbers (not *String* and *Number* objects)– **concat** copies strings and numbers into the new array. Changes to the string or number in one array does not affect the other arrays.

If a new element is added to either array, the other array is not affected.

slice

Core method. Extracts a section of an array and returns a new array.

Syntax

```
arrayName.slice(beginSlice, [endSlice])
```

Parameters

arrayName is the name of an array.

beginSlice is the zero-based index at which to begin extraction.

endSlice is the zero-based index at which to end extraction.

- **slice** extracts up to but not including *endSlice*. `arrayName.slice(1,4)` extracts the second element through the fourth element (elements indexed 1, 2, and 3)
- As a negative index, *endSlice* indicates an offset from the end of the sequence. `arrayName.slice(2,-1)` extracts the third element through the second to last element in the sequence.
- If *endSlice* is omitted, **slice** extracts to the end of the sequence.

Method of

Array object

Description

slice does not alter the original array, but returns a new "one level deep" copy that contains copies of the elements sliced from the original array. Elements of the original array are copied into the new array as follows:

- Object references (and not the actual object) – **slice** copies object references into the new array. Both the original and new array refer to the same object. If a referenced object changes, the changes are visible to both the new and original arrays.

- Strings and numbers (not *String* and *Number* objects) – **slice** copies strings and numbers into the new array. Changes to the string or number in one array does not affect the other array.

If a new element is added to either array, the other array is not affected.

Example

In the following example slice creates a new array, newCar, from myCar. Both include a reference to the object myHonda. When the color of myHonda is changed to purple, both arrays are aware of the change.

```
<SCRIPT LANGUAGE="JavaScript1.2">

//Using slice, create newCar from myCar.

myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
myCar = [myHonda, 2, "cherry condition", "purchased 1997"]
newCar = myCar.slice(0,2)

//Write the values of myCar, newCar, and the color of myHonda
//referenced from both arrays.
document.write("myCar = " + myCar + "<BR>")
document.write("newCar = " + newCar + "<BR>")
document.write("myCar[0].color = " + myCar[0].color + "<BR>")
document.write("newCar[0].color = " + newCar[0].color + "<BR><BR>")

//Change the color of myHonda
myHonda.color = "purple"
document.write("The new color of my Honda is " + myHonda.color + "<BR><BR>")

//Write the color of myHonda referenced from both arrays.
document.write("myCar[0].color = " + myCar[0].color + "<BR>")
document.write("newCar[0].color = " + newCar[0].color + "<BR>")

</SCRIPT>
```

This writes:

```
myCar = [{color:"red", wheels:4, engine:{cylinders:4, size:2.2}}, 2, "cherry
condition", "purchased 1997"]
newCar = [{color:"red", wheels:4, engine:{cylinders:4, size:2.2}}, 2]
myCar[0].color = red newCar[0].color = red
```

The new color of my Honda is purple

```
myCar[0].color = purple
newCar[0].color = purple
```

sort

Core method. **sort** now works on all platforms. It no longer converts undefined elements to null, and it sorts them to the high end of the array. For example:

```
<SCRIPT>
a = new Array();
a[0] = "Ant";
a[5] = "Zebra";

function writeArray(x) {
    for (i = 0; i < x.length; i++) {
        document.write(x[i]);
        if (i < x.length-1) document.write(", ");
    }
}

writeArray(a);
a.sort();
document.write("<BR><BR>");
writeArray(a);
</SCRIPT>
```

JavaScript in Navigator 3 prints:

ant, null, null, null, null, zebra

ant, null, null, null, null, zebra

JavaScript in Navigator 4 prints:

ant, undefined, undefined, undefined, undefined, zebra

ant, zebra, undefined, undefined, undefined, undefined

Creating Arrays Under JavaScript 1.2

If you specify `LANGUAGE="JavaScript1.2"` in the script tag, using `new Array(1)` creates a new array with `a[0]=1`.

Without the `LANGUAGE="JavaScript1.2"` specification, `new Array(1)` sets the array's length to 1 and `a[0]` to undefined.

Example

The following example prints 1. Without `LANGUAGE="JavaScript1.2"`, it prints undefined.

```
<SCRIPT LANGUAGE="JavaScript1.2">
a=new Array(1);
document.write(a[0] + "<BR>");
</SCRIPT>
```

Working With Arrays and Regular Expressions

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `regexp.exec`, `string.match`, and `string.replace`.

Syntax

```
arrayName.propertyName
arrayName[element0,..., elementn]
```

Parameters

arrayName is the name of the array.

propertyName is one of the properties listed below.

elementn is one of the elements listed below.

Properties and Elements

To help explain the properties and elements, look at the following example and then refer to the table below:

```
<SCRIPT>
//Match one d followed by one or more b's followed by one d
//Remember matched b's and the following d
//Ignore case

myRe=/d(b+)(d)/i;
myArray = myRe.exec("cdbBdbsbz");

</SCRIPT>
```

The properties and elements returned from a match between a regular expression and a string are as follows (the examples are a result of the above script):

Property/Element	Description	Example
<i>input</i>	A read-only property that reflects the original string against which the regular expression was matched.	cdbBdbsbz
<i>index</i>	A read-only property that is the zero-based index of the match in the string.	1
[0]	A read-only element that specifies the last matched characters.	dbBd
[1], ...[n]	Read-only elements that specify the parenthesized substring matches, if included in the regular expression. The number of possible parenthesized substrings is unlimited.	[1]=bB [2]=d

Description

The returned array varies depending on the type of match that was executed. The following lists shows what is returned if the match is successful. (*regexp* is a regular expression, *str* is a string, and *replaceText* is the replacement text for the replace method.

regexp.exec(str) returns:

An array containing the match string, any parenthesized substring matches, and the *input* and *index* properties as described above.

`str.match(regex)` returns:

An array containing the match string, the last parenthesized substring match (if included), and the *input* and *index* properties as described above.

`str.match(regex)` where *regex* includes the global flag (e.g. `/abc/g`) returns:

An array of all matches and the last parenthesized substring match (if included). *input* and *index* are undefined.

`str.match(regex)` where *regex* includes the global and ignore case flags (e.g. `/abc/gi`) returns:

An array of all matches. *input* and *index* are undefined.

`str.replace(regex, "replaceText")` returns:

A string with the regular expression match replaced by *replaceText*. *input* and *index* are undefined.

Operators

This section provides information about changes to the equality operators and the new delete operator.

Equality Operators

If the `<SCRIPT>` tag uses `LANGUAGE=JavaScript1.2`, the equality operators (`==` and `!=`) work differently. This section describes how the equality operators work without `LANGUAGE=JavaScript1.2` and how they work with `LANGUAGE=JavaScript1.2`. For instructions on writing code to convert strings and numbers, see “Data Conversion”.

Equality Operators Without `LANGUAGE=JavaScript1.2`

The following describes how the equality operators (`=`, `==`, and `!=`) worked in previous versions of JavaScript and how they work in JavaScript 1.2 when `LANGUAGE=JavaScript1.2` is *not* used in the `<SCRIPT>` tag. Note that if `LANGUAGE=JavaScript`, and `LANGUAGE=JavaScript1.1` are used, the equality operators maintain their previous behavior.

- If both operands are objects, compare object references.
- If either operand is null, convert the other operand to an object and compare references.
- If one operand is a string and the other is an object, convert the object to a string and compare string characters.
- Otherwise, convert both operands to numbers and compare numeric identity.

Equality Operators With LANGUAGE=JavaScript1.2

If LANGUAGE=JavaScript1.2 is used in the <SCRIPT> tag, the equality operators (= and !=) behave as follows:

- They never attempt to convert operands from one type to another. To convert operands, do the following:
- They always compare identity of like-typed operands. If the operands do not have like type, they are not equal.

This approach avoids errors, maintains transitivity, and simplifies the language.

Data Conversion

To write JavaScript code that converts strings to numbers and numbers to strings in the different versions of Navigator, follow these guidelines:

- When writing for all versions of Navigator:
 - To convert x to a string, use " " + x. For example,

```
(( " " + 3 )=="3 ")
```
 - To convert x to a number, use (x - 0). For example,

```
(( "3 "-0)==3 )
```
- When writing for Navigator 4.0 only:

- To convert `x` to a string, in addition to `" " + x`, you can use `String(x)`. For example,

```
var x = 3
String(x) = "3"
```

- To convert `x` to a number, in addition to `(x - 0)`, you can use `Number(x)`. For example,

```
var x = "3"
Number(x) = 3
```

Example

The following example demonstrates the `==` operator with different `<SCRIPT>` tags.

```
<HTML>

<SCRIPT>
document.write("3" == 3);
</SCRIPT>
<BR>

<SCRIPT>
document.write(("3"-0) == 3);
</SCRIPT>
<BR>

<SCRIPT LANGUAGE="JavaScript">
document.write("3" == 3);
</SCRIPT>
<BR>

<SCRIPT LANGUAGE="JavaScript1.1">
document.write("3" == 3);
</SCRIPT>
<BR>

<SCRIPT LANGUAGE="JavaScript1.2">
document.write("3" == 3);
</SCRIPT>
<BR>

<SCRIPT LANGUAGE="JavaScript1.2">
document.write(("3"-0) == 3);
</SCRIPT>
<BR>

<SCRIPT LANGUAGE="JavaScript1.2">
```

delete

```
document.write(String(3) == "3");  
</SCRIPT>  
<BR>  
</HTML>
```

The output from the above is:

```
true  
true  
true  
true  
false  
true  
true
```

delete

Core operator. Deletes an object's property or an element at a specified index in an array.

Syntax

```
delete objectName.property  
delete objectname[index]
```

Parameters

property is any existing property.

index is an integer representing the location of an element in an array.

Description

If the deletion succeeds, the delete operator sets the property or element to undefined and returns true; otherwise, it returns false.

Properties

This section list new and revised properties for:

- Function object
- navigator object
- window object

Function Property

arity

Core property. When the LANGUAGE attribute of the SCRIPT tag is "JavaScript1.2", this property indicates the number of arguments expected by a function.

Syntax

```
functionName.arity
```

Parameters

functionName is the name of a function.

Property of

Function object

Description

arity is external to the function, and indicates how many arguments the function expects. The *length* property is internal to the function and indicates how many arguments were passed to the function. For more information on the *length* property, see the “JavaScript Guide.”

Example

The following example demonstrates the use of *arity* and *length*.

```
<SCRIPT LANGUAGE = "JavaScript1.2">

function addNumbers(x,y){
    document.write("length = " + arguments.length + "<BR>")
    z = x + y
}

document.write("arity = " + addNumbers.arity + "<BR>")

addNumbers(3,4,5)

</SCRIPT>
```

This writes:

```
arity = 2
length = 3
```

navigator Properties

language

Client-side property. Indicates the translation of the Navigator being used.

Syntax

```
navigator.language
```

Property of

navigator object

Description

The language property has been added for use with the JAR Manager. For more information, see the “JAR Installation Manager Developer's Guide.”

The value for language is usually a two-letter code, such as "en" and occasionally a five-character code to indicate a language sub-type, such as "zh_CN".

Webpage authors would use this property when they need to determine the language of the Navigator client software being used. For example the scripter could display translated text for the user.

language is a read-only property.

platform

Client-side property. Indicates the machine type for which the Navigator was compiled.

Syntax

`navigator.platform`

Property of

navigator object

Description

The platform property has been added for use with the JAR Manager. For more information, see the "JAR Installation Manager Developer's Guide."

The machine type the Navigator was compiled for may differ from the actual machine type due to version differences, emulators, or other reasons.

Webpage authors would use this property to ensure that their triggers download the appropriate JAR files. The triggering page checks the Navigator version before checking the platform property.

JAR-install writers would use this property to double-check that their package is being installed on an appropriate machine, or for small JAR's to decide which of several machine-specific files to install.

Platform values are Win32, Win16, Mac68k, MacPPC and various Unix.

platform is a read-only property.

window Properties

innerHeight

Client-side property. Specifies the vertical dimension, in pixels, of the window's content area.

Syntax

```
[windowReference.]innerWidth
```

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

To create a window smaller than 100 x 100 pixels, set this property in a signed script.

innerWidth

Client-side property. Specifies the horizontal dimension, in pixels, of the window's content area.

Syntax

```
[windowReference.]innerWidth
```

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

To create a window smaller than 100 x 100 pixels, set this property in a signed script.

locationbar

Client-side object and a property of the *window* object. Represents the Navigator window's location bar.

Syntax

```
[windowReference.]locationbar.visible
```

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *locationbar* object has one property, *visible*, that allows you to hide or show the location bar of the specified window.

This property must be set in a signed script.

Property

`visible = [true, false] | [1, 0]`

Example

The following example hides the location bar of *myWindow*.

```
myWindow.locationbar.visible=false
```

menubar

Client-side object and a property of the *window* object. Represents the Navigator window's menu bar.

Syntax

`[windowReference.]menubar.visible`

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *menubar* object has one property, *visible*, that allows you to hide or show the menu bar of the specified window.

This property must be set in a signed script.

Property

```
visible = [true, false] | [1, 0]
```

Example

The following example hides the menu bar of *myWindow*.

```
myWindow.menubar.visible=false
```

outerHeight

Client-side property. Specifies the vertical dimension, in pixels, of the window's outside boundary.

Syntax

```
[windowReference.]innerWidth
```

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

To create a window smaller than 100 x 100 pixels, set this property in a signed script.

outerWidth

Client-side property. Specifies the horizontal dimension, in pixels, of the window's outside boundary.

Syntax

`[windowReference].innerWidth`

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

To create a window smaller than 100 x 100 pixels, set this property in a signed script.

pageXOffset

Client-side property. Provides the current x-position, in pixels, of a window's viewed page.

Syntax

`windowReference.pageXOffset`

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *pageXOffset* property provides the current x-position of a page as it relates to the upper-left corner of the window's content area. This property is useful when you need to find the current location of the scrolled page before using `scrollTo` or `scrollBy`.

This is a read-only property.

Example

The following example returns the x-position of the viewed page.

```
x = myWindow.pageXOffset
```

See Also

`pageYOffset`

pageYOffset

Client-side property. Provides the current y-position, in pixels, of a window's viewed page.

Syntax

windowReference.*pageYOffset*

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *pageYOffset* property provides the current y-position of a page as it relates to the upper-left corner of the window's content area. This property is useful when you need to find the current location of the scrolled page before using `scrollTo` or `scrollBy`.

This is a read-only property.

Example

The following example returns the y-position of the viewed page.

```
x = myWindow.pageYOffset
```

See Also

`pageXOffset`

personalbar

Client-side object and a property of the *window* object. Represents the Navigator window's personal bar (also called the directories bar).

Syntax

`[windowReference.]personalbar.visible`

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *personalbar* object has one property, *visible*, that allows you to hide or show the personal bar of the specified window.

This property must be set in a signed script.

Property

`visible = [true, false] | [1, 0]`

Example

The following example hides the personal bar of *myWindow*.

```
myWindow.personalbar.visible=false
```

scrollbars

Client-side object and a property of the *window* object. Represents the Navigator window's scroll bars.

Syntax

`[windowReference.]scrollbar.visible`

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *scrollbars* object has one property, *visible*, that allows you to hide or show the scroll bar of the specified window.

This property must be set in a signed script.

Property

`visible = [true, false] | [1, 0]`

Example

The following example hides the scroll bar of *myWindow*.

```
myWindow.scrollbars.visible=false
```

statusbar

Client-side object and a property of the *window* object. Represents the Navigator window's status bar.

Syntax

`[windowReference.]statusbar.visible`

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *statusbar* object has one property, *visible*, that allows you to hide or show the status bar of the specified window.

This property must be set in a signed script.

Property

```
visible = [true, false] | [1, 0]
```

Example

The following example hides the status bar of *myWindow*.

```
myWindow.statusbar.visible=false
```

toolbar

Client-side object and a property of the *window* object. Represents the Navigator window's tool bar.

Syntax

```
[windowReference.]toolbar.visible
```

Parameters

windowReference is either the name of a window object or one of the synonyms *top* or *parent*.

Property of

window object

Description

The *toolbar* object has one property, *visible*, that allows you to hide or show the toolbar of the specified window.

This property must be set in a signed script.

Property

```
visible = [true, false] | [1, 0]
```

Example

The following example hides the tool bar of *myWindow*.

```
myWindow.toolbar.visible=false
```

Regular Expressions

Regular expressions are patterns used to match character combinations in strings. For example, to search for all occurrences of 'the' in a string, you create a pattern consisting of 'the' and use the pattern to search for its match in a string. Regular expression patterns are constructed using literal notation (`/abc/`) or the `RegExp` constructor function (`re = new RegExp("abc")`). These patterns are used with the regular expression methods, **`exec`** and **`test`** and with the *String* methods, **`match`**, **`replace`**, **`search`**, and **`split`**.

This section includes:

- Constructing Regular Expressions
- Working With Regular Expressions
- A Complete Example
- Special Characters Used in Regular Expressions

For complete information on the objects, properties, and methods used with regular expressions, see:

- The Regular Expression Object
- The `RegExp` Object
- The `String` Object
- The `Array` Object

Constructing Regular Expressions

In JavaScript, a regular expression is an object that contains the pattern used to search for a match in a string. This section describes the regular expression syntax and how to write a pattern.

The Regular Expression Syntax

You construct a regular expression in one of two ways:

- Using literal notation,

```
re = /ab+c/
```

Literal notation provides compilation of the regular expression once only, when the script is first loaded. When the regular expression will remain constant, use literal notation for better performance.

- Calling the constructor function of the RegExp object,

```
re = new RegExp("ab+c")
```

- Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input. Once you have a defined regular expression, and if the regular expression is used throughout the script, you can use the compile method to compile the regular expression for efficient reuse.

The regular expression object is explained in detail in “Regular Expression Object.”

The examples used in the remainder of this section are shown in literal form.

Writing a Regular Expression Pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use.

Using Simple Patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string "Grab crab" because it does not contain the substring 'abc'.

Using Special Characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding a whitespace, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (* means zero or more of the preceding character) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

"Special Characters Used in Regular Expressions" provides a complete list and description of the special characters that can be used in regular expressions.

Using Parentheses

Parentheses around any part of the regular expression pattern cause that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use.

For example, the pattern `/Chapter, (\d+)\.\d*/` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter, ' followed by one or more numeric characters (\d means any numeric character and + means one or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character zero or more times (\d means numeric character, * means zero or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapters 3 and 4."

How you use parenthesized substring matches is described in “Using Parenthesized Substring Matches.”

Working With Regular Expressions

Regular expressions are used with the regular expression methods **test** and **exec** and with the *String* methods **match**, **replace**, **search**, and **split**. These methods are explained in detail at their linked locations.

<code>exec</code>	A regular expression method that executes a search for a match in a string. It returns an array of useful information.
<code>test</code>	A regular expression method that tests for a match in a string. It returns true or false.
<code>match</code>	A <i>String</i> method that executes a search for a match in a string. It returns an array of useful information.
<code>search</code>	A <i>String</i> method that tests for a match in a string. It returns true or false.
<code>replace</code>	A <i>String</i> method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
<code>split</code>	A <i>String</i> method that uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string use the **test** or **search** method; for more information (but slower execution) use the **exec** or **match** methods. If you use **exec** or **match** and if the match succeeds, these methods return an array and update properties of the regular expression object and the global regular expression object, *RegExp*.

For information about the returned array and its properties, see “Working With Regular Expressions.”

For information about the global *RegExp* object and its properties, see “The RegExp Object.”

In the following example, the script uses the **exec** method to find a match in a string.

```
<SCRIPT>

myRe=/db+d/ ;
myArray = myRe.exec( "cbbdbbsbz" ) ;
```

</SCRIPT>

The match succeeds and returns the following array and updates the following properties:

Object	Property/Index	Description	Example
<i>myArray</i>		all array elements	dbbd
	<i>index</i>	the zero-based index of the match in the string	1
	<i>input</i>	the original string	cdbbdbsbz
	[0]	the last matched characters	dbbd
<i>myRe</i>	<i>lastIndex</i>	the index at which to start the next match.	5
	<i>source</i>	the text of the pattern	db+d
<i>RegExp</i>	<i>lastMatch</i>	the last matched characters	dbbd
	<i>leftContext</i>	the string up to the most recent match	c
	<i>rightContext</i>	the string past the most recent match	bsbz

If the match fails, the **exec** method returns null (which converts to Boolean false).

Using Parenthesized Substring Matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the global *RegExp* properties `$1`, ..., `$9` or the *Array* elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. The *RegExp* object holds up to the last nine and the returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

Example 1. The following script uses the **replace** method to switch the words in the string. For the replacement text, the script uses the values of the *\$1* and *\$2* properties of the global *RegExp* object. Note that the *RegExp* object name is not be prepended to these properties when they are passed as the second argument to the **replace** method.

```
<SCRIPT>
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

Example 2. In the following example, *RegExp.input* is set by the *Change* event. In the **getInfo** function, the **exec** method uses the value of *RegExp.input* as its argument. Note that *RegExp* must be prepended to its *\$* properties (since they appear outside the context of a regular expression).

```
<HTML>

<SCRIPT>
function getInfo(){
re = /(\w+)\s(\d+)/;
re.exec();
window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>
```

Enter your first name and your age, and then press Enter.

```
<FORM>
<INPUT TYPE="TEXT" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

Example 3. The following example is similar to Example 2. Instead of using the *RegExp.\$1* and *RegExp.\$2*, this example creates an array and uses *a[1]* and *a[2]*.

```
<HTML>

<SCRIPT>
function getInfo(){
re = /(\w+)\s(\d+)/;
a = re.exec();
window.alert(a[1] + ", your age is " + a[2]);
}
</SCRIPT>
```

Enter your first name and your age, and then press Enter.

```
<FORM>
<INPUT TYPE="TEXT" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

Executing a Global Search and Ignoring Case

Regular expressions have two optional flags that allow for global and case insensitive searching. To indicate a global search, use the `g` flag. To indicate a case insensitive search, use the `i` flag. These flags can be used separately or together in either order, and are included as part of the regular expression.

To include a flag with the regular expression, use this syntax

```
re = /pattern/[g|i|gi]
re = new RegExp("pattern", [g|i|gil])
```

Note that the flags, `i` and `g`, are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for any number of characters followed by a space, and it looks for this combination throughout the string.

```
<SCRIPT>
re = /\w+\s/g;
str = "fee fi fo fum";
myArray = str.match(re);
document.write(myArray);
</SCRIPT>
```

This writes "fee ,fi ,fo".

A Complete Example

The following example illustrates the formation of regular expressions and the use of `string.split()` and `string.replace()`.

It cleans a roughly-formatted input string containing names (first name first) separated by blanks, tabs and exactly one semicolon.

Finally, it reverses the name order (last name first) and sorts the list.

```
<SCRIPT LANGUAGE="JavaScript1.2">

/*****
 * The name string contains multiple spaces and tabs,
 * and may have multiple spaces between first and last names.
 *****/
names = new String ( "Harry Trump ;Fred Barney; Helen Rigby ;\
                    Bill Abel ;Chris Hand ")

document.write ("----- Original String" + "<BR>" + "<BR>")
document.write (names + "<BR>" + "<BR>")

/*****
 * Prepare two regular expression patterns and array storage.
 * Split the string into array elements.
 *****/
// pattern: possible white space then semicolon then possible white space
pattern = /\s*;\s*/
// break the string into pieces separated by the pattern above and
// and store the pieces in an array called nameList
nameList = names.split (pattern)

// new pattern: one or more characters then spaces then characters
// use parentheses to "memorize" portions of the pattern
// the memorized portions are referred to later

pattern = /(\w+)\s+(\w+)/

// new array for holding names being processed
bySurnameList = new Array;

/*****
 * Display the name array and populate the new array
 * with comma-separated names, last first.
 *
 * The replace method removes anything matching the pattern
 * and replaces it by the memorized string - 2nd memorized portion
 * followed by comma space followed by 1st memorized portion.
 *
 * The variables $1 and $2 refer to the portions
 * memorized while matching the pattern.
 *****/
document.write ("----- After Split by Regular Expression" + "<BR>")
for ( i = 0; i < nameList.length; i++) {
    document.write (nameList[i] + "<BR>")
    bySurnameList[i] = nameList[i].replace (pattern, "$2, $1")
}
```

```

/*****
* Display the new array.
*****/
document.write ("----- Names Reversed" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
    document.write (bySurnameList[i] + "<BR>")
}

/*****
* Sort by last name, then display the sorted array.
*****/
bySurnameList.sort()
document.write ("----- Sorted" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
    document.write (bySurnameList[i] + "<BR>")
}

document.write ("----- End" + "<BR>")

</SCRIPT>

```

Special Characters Used in Regular Expressions

The following list describes the special characters that can be used in regular expressions.

\	indicates that the next character is special and not to be interpreted literally. For example, /b/ matches the character 'b'. By placing a backslash in front of b, e.g. /\b/, the character becomes special to mean match a word boundary.
-or-	
	indicates that the next character is not special and should be interpreted literally. For example, * is a special character that means zero or more of the preceding character should be matched, e.g. /a*/ means match zero or more a's. To match * literally, precede the it with a backslash, e.g. /a*/ matches 'a*'. matches beginning of input or line, e.g. /^A/ matches only the first 'A' in "An A+ for Kelly."
^	
\$	matches end of input or line, e.g. /t\$/ matches only the last 't' in "A cat in the hat".
*	matches the preceding character zero or more times, e.g. /bo*/ matches 'boooo' in "The ghost screamed boooo."
+	matches the preceding character one or more times (equivalent to {1,}), e.g. /a+/ matches the 'a' in "candy" and all the a's in "caaaaaaandy."
?	matches the preceding character zero or one time, e.g. /e?le?/ matches the 'el' in "angel" and the 'le' in "angle."
.	(the decimal point) matches any single character except new line, e.g. /.n/ matches 'an' and 'on' in "an apple is on the tree."
(x)	matches 'x' and remembers the match, e.g. /(foo)/ matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the result <i>Array</i> elements [1], ..., [n], or the global <i>RegExp</i> properties \$1, ..., \$9.
x y	matches either 'x' or 'y', e.g. /green red/ matches 'green' in "green apple" and 'red' in "red apple."
{x}	where x is a non-negative integer. Matches exactly x times, e.g. /a{2}/ doesn't match the 'a' in "candy," matches all of the a's in "caandy," and the first two a's in "caaaaaaandy."

<code>{x,}</code>	where x is a non-negative integer. Matches at least x times, e.g. <code>/a{2,}</code> doesn't match the 'a' in "candy" and matches all of the a's in "caandy" and in "caaaaaaandy."
<code>{x,y}</code>	where x and y are non-negative integers. Matches at least x and at most y times, e.g. <code>/a{1,3}/</code> matches the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy."
<code>[xyz]</code>	a character set. Matches any one of the enclosed characters, e.g. <code>[abc]</code> matches the 'b' in "brisket" and the 'c' in "chop."
<code>[^xyz]</code>	a negative character set. Matches anything that is not enclosed in the brackets, e.g. <code>[^abc]</code> matches 'r' in "brisket" and 'h' in "chop."
<code>\b</code>	matches a word boundary, such as a space, e.g. <code>/\bn\w/</code> matches the 'no' in "noonday", and <code>/\wy\b/</code> matches the 'ly' in "possibly yesterday."
<code>\B</code>	matches a non-word boundary, e.g. <code>/\w\Bn/</code> matches 'on' in "noonday", and <code>/y\B\w/</code> matches 'ye' in "possibly yesterday."
<code>\d</code> <code>[0-9]</code>	matches a digit character, e.g. <code>/\d/</code> or <code>/[0-9]/</code> matches '2' in "B2 is the suite number."
<code>\D</code> <code>[^0-9]</code>	matches any non-digit character, e.g. <code>/\D/</code> or <code>/[^0-9]/</code> matches 'B' in "B2 is the suite number."
<code>\f</code>	matches a form-feed.
<code>\n</code>	matches a linefeed.
<code>\r</code>	matches a carriage return.
<code>\s</code> <code>[\f\n\r\t\v]</code>	matches any white space including space, tab, form feed, line feed, e.g. <code>/\s\w*/</code> matches ' bar' in "foo bar."
<code>\S</code> <code>[^\f\n\r\t\v]</code>	matches any non-white space, e.g. <code>/\S/\w*</code> matches 'foo' in "foo bar."
<code>\t</code>	matches a tab
<code>\v</code>	matches a vertical tab.

<code>\w</code>	matches any word character including the underscore, e.g.
<code>[A-Za-z0-9_]</code>	<code>/\w/</code> matches 'a' in "apple," '5' in "\$5.28," and '3' in "3D."
<code>\W</code>	matches any non-word character, e.g. <code>/\W/</code> or <code>/[^A-Za-z0-9_]/</code> matches '%' in "50%."
<code>^\#</code>	where # is a positive integer. A back-reference to the last substring matching the # parenthetical in the regular expression (counting left parentheses), e.g. <code>/apple(,)\sorange\1/</code> matches 'apple, orange', in "apple, orange, cherry, peach." A more complete example follows this table. Note: if the number of left parentheses is less than the number specified in <code>\#</code> , the <code>\#</code> is taken as an octal escape as described in the next row.
<code>/x/</code>	where x is an octal, hexadecimal, or decimal escape value. Allows you to embed ASCII codes into regular expressions.

Example Using Special Characters

In the following example, a user enters a phone number. When the user presses Enter, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script posts a window thanking the user and confirming the number. If the number is invalid, the script posts a window telling the user that the phone number isn't valid.

The regular expression looks for zero or one open parenthesis `\(`, followed by three digits `\d{3}`, followed by zero or one close parenthesis `\)?`, followed by one dash, forward slash, or decimal point and when found, remember the character `([-\./])`, followed by three digits `\d{3}`, followed by the remembered match of a dash, forward slash, or decimal point `\1`, followed by four digits `\d{4}`.

The Change event activated when the user presses Enter, sets the value of `RegExp.input`.

```
<HTML>

<SCRIPT LANGUAGE = "JavaScript1.2">

re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/
```

```
function testInfo() {  
    OK = re.exec()  
    if (!OK)  
        window.alert (RegExp.input + " isn't a phone number with area code!")  
    else  
        window.alert ("Thanks, your phone number is " + OK[0])  
}  
  
</SCRIPT>  
  
Enter your phone number (with area code) and then press Enter.  
  
<FORM> <INPUT TYPE:"TEXT" NAME="Phone" onChange="testInfo(this);"> </FORM>  
  
</HTML>
```


The RegExp Object

Core object. A global object with properties that are set either before a search for a regular expression in a string, or after a match is found. (This is not to be confused with the “Regular Expression Object” which contains the regular expression pattern.)

Syntax

`RegExp.propertyName`

Parameters

propertyName is one of the properties listed below.

Properties

Note that six of the *RegExp* properties have both long and short (Perl-like) names. Both names always refer to the same value. Perl is the programming language from which JavaScript modeled its regular expressions.

<i>input</i> \$ ₋	A read/write property that reflects the string against which the regular expression is matched. This value is preset as described in the “Description” below. If no string argument is provided to the regular expression's exec or test methods, and if <code>RegExp.input</code> has a value, its value is used as the argument.
<i>multiline</i> \$ [*]	A read/write Boolean property that reflects whether or not to search in strings across multiple lines; <code>true</code> if multiple lines are searched, <code>false</code> if searches must stop at line breaks.
<i>lastMatch</i> \$&	A read-only property that specifies the last matched characters.
<i>lastParen</i> \$+	A read-only property that specifies the last parenthesized substring match, if any.
<i>leftContext</i> \$ [~]	A read-only property that specifies the string up to the most recent match.
<i>rightContext</i> \$'	A read-only property that specifies the string past the most recent match.
\$1, ..., \$9	Read-only properties that contain parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited, but the <i>RegExp</i> object can only hold the last nine. You can access all parenthesized substrings through the returned array's indexes.

These properties can be used in the replacement text for the *String* **replace** method. When used this way, do not prepend them with *RegExp*. “Example 1” illustrates this. When parentheses are not included in the regular expression, the script interprets `$#`s literally (where `#` is a positive integer).

Methods

None.

Description

The *RegExp* global object contains the properties listed above. Except for *input* and *multiline* whose values can be preset, property values are set after execution of the regular expression methods **exec** and **test**, and the *String* methods, **match**, and **replace**.

The script or the Navigator can preset the *input* property. If preset and if no string argument is explicitly provided, *input*'s value is used as the string argument to the **exec** or **test** methods. *input* is set by the Navigator in the following cases:

- When an event handler is called for a TEXT form element, *input* is set to the value of the contained text.
- When an event handler is called for a TEXTAREA form element, *input* is set to the value of the contained text. Note that *multiline* is also set to `true` so that the match can be executed over the multiple lines of text.
- When an event handler is called for a SELECT form element, *input* is set to the value of the selected text.
- When an event handler is called for a *Link* object, *input* is set to the value of the text between `` and ``.

input is cleared after each of the above calls.

The script or the Navigator can preset the *multiline* property. When an event handler is called for a TEXTAREA form element, the Navigator sets *multiline* to `true`. *multiline* is cleared after a call by any event handler. This means that, if you've preset *multiline* to `true`, it is reset to `false` after the execution of any event handler.

Examples

Example 1 The following script uses the **replace** method to switch the words in the string. For the replacement text, the script uses the values of the *\$1* and *\$2* properties of the global *RegExp* object. Note that the *RegExp* object name is not be prepended to the *\$* properties when they are passed as the second argument to the **replace** method.

```
<SCRIPT>
```

```

re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>

```

This prints "Smith, John".

Example 2. In the following example, `RegExp.input` is set by the `Change` event. In the **getInfo** function, the **exec** method uses the value of `RegExp.input` as its argument. Note that *RegExp* is prepended to the `$` properties.

```

<HTML>

<SCRIPT>
function getInfo(){
re = /(\w+)\s(\d+)/;
re.exec();
window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="TEXT" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>

```


Regular Expression Object

This section describes the regular expression object and its methods.

Regular Expression Object

compile method

exec method

test method

Core object. A regular expression object contains the pattern of a regular expression. (This is not to be confused with the single global object, *RegExp*.)

Syntax

Note The literal text format is compiled into a compact and efficient internal representation.

Literal notation:

```
regex = /pattern/[i|g|gi]
```

Constructed:

```
regex = new RegExp("pattern", ['i'|'g'|'gi'])
```

Parameters

regexp is the name of the regular expression object.

pattern is the text of the regular expression.

Optional flags:

- i ignore case
- g global match
- gi global match and ignore case

Description

The literal notation, e.g. `/ab+c/`, provides compilation of the regular expression once only, when the script is first loaded. Use literal notation when the regular expression will remain constant. For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

The constructor of the regular expression object, e.g. `new RegExp("ab+c")`, provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input. Once you have a defined regular expression, and if the regular expression is used throughout the script, you can use the **compile** method to compile the regular expression for efficient reuse.

Note that when using the constructor function, the normal string escape rules (preceding special characters with `\` when included in a string) are necessary. For example, `new RegExp("\\w+")` is the runtime equivalent of `/\w+/`. (The special pattern `\w+` looks for a match of an alphanumeric character one or more times.)

Properties

<i>global</i>	A read-only Boolean property that reflects whether or not the 'g' flag is used with the regular expression: <code>true</code> if used, <code>false</code> if not used. The 'g' flag indicates that the regular expression should be tested against all possible matches in a string.
<i>ignoreCase</i>	A read-only Boolean property that reflects whether or not the 'i' flag is used with the regular expression: <code>true</code> if used, <code>false</code> if not used. The 'i' flag indicates that case should be ignored while attempting a match in a string.
<i>lastIndex</i>	A read/write integer property that specifies the index at which to start the next match. The following rules apply:

- If *lastIndex* is greater than the length of the string, `regexp.test` and `regexp.exec` fail, and *lastIndex* is set to 0.
- If *lastIndex* is equal to the length of the string, there are two cases:
 - If the regular expression matches the empty string, it matches input starting at *lastIndex*.
 - if the regular expression does not match the empty string, it mismatches input, and *lastIndex* is reset to 0.
- Otherwise, *lastIndex* is set to the next position following the most recent match.

For example:

```
re = /(hi)?/g //matches empty string
re("hi")      //returns ["hi", "hi"] with
               //lastIndex == 2
re("hi")      //returns [""], an empty array whose
               //[0] element is the match string, in
               //this case, the empty string
               //because lastIndex was 2 (and still
               //is 2) and "hi" has length 2.
```

<i>source</i>	A read-only property that contains the text of the pattern.
---------------	---

Methods

compile	Compiles a regular expression object.
exec	Executes a search for a match in its string parameter. Returns a result array.
test	Tests for a match in its string parameter. Returns <code>true</code> or <code>false</code> .

compile

Core method. Compiles a regular expression object during execution of a script.

Syntax

```
regex.compile(pattern, ['i'|'g'|'gi'])
```

Parameters

regex is the name of the regular expression.

pattern is the text of the regular expression.

Optional flags:

i	ignore case
g	global match
gi	global match and ignore case

Description

Use the **compile** method to compile a regular expression created with the constructor function. This forces compilation of the regular expression once only which means the regular expression isn't compiled each time it is encountered. Use the **compile** method when you know the regular expression will remain constant (after getting its pattern) and will be used repeatedly throughout the script.

You can also use the **compile** method to change the regular expression during execution. For example, if the regular expression changes, you can use the **compile** method to recompile the object for more efficient repeated use.

exec

Core method. Executes the search for a match in a specified string.

Syntax

```
regex.exec(str)
```

or use the shortcut version

```
regex(str)
```

Parameters

regex is the name of the regular expression. It can be a variable name or a literal.

str is the string against which to match the regular expression.

Description

Note If you are executing a match simply to find `true` or `false`, use the `test` method or the *String* search method.

To explain the **exec** method, look at the following example and then refer to the table below:

```
<SCRIPT>
//Match one d followed by one or more b's followed by one d
//Remember matched b's and the following d
//Ignore case
myRe=/d(b+)(d)/i;
myArray = myRe.exec("cdbBdbbsbz");
</SCRIPT>
```

If the match succeeds, the **exec** method returns an array and updates properties of the regular expression object and the global regular expression object, *RegExp*, as follows (the examples are the result of the above script):

Object	Property/Index	Description	Example
<i>myArray</i>		the contents of myArray	dbBd , bB , d
	<i>index</i>	the zero-based index of the match in the string	1
	<i>input</i>	the original string	cdbBdbbsbz
	[0]	the last matched characters	dbBd
	[1], ...[n]	the parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited.	[1] = bB [2] = d
<i>myRe</i>	<i>lastIndex</i>	the index at which to start the next match.	5
	<i>ignoreCase</i>	indicates if the 'i' flag was used to ignore case	true
	<i>global</i>	indicates if the 'g' flag was used for a global match	false
	<i>source</i>	the text of the pattern	d(b+)d
<i>RegExp</i>	<i>lastMatch</i> ¹ \$&	the last matched characters	dbBd
	<i>leftContext</i> \$^	the string up to the most recent match	c
	<i>rightContext</i> \$'	the string past the most recent match	bbsbz

Object	Property/Index	Description	Example
	<i>\$1, ...\$9</i>	the parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited, but <i>RegExp</i> can only hold the last nine.	<i>\$1</i> = bB <i>\$2</i> = d
	<i>lastParen</i> <i>\$+</i>	the last parenthesized substring match, if any.	d

1. Note that four of the *RegExp* properties have both long and short (Perl-like) names. Both names always refer to the same value. Perl is the programming language from which JavaScript modeled its regular expressions.

If the match fails, the **exec** method returns null (which converts to Boolean false).

Examples

In the following example, the user enters a name and the script executes a match against the input. It then cycles through the array to see if other names match the user's name.

This script assumes that first names of registered party attendees are preloaded into the array A, perhaps by gathering them from a party database.

```
<HTML>

<SCRIPT LANGUAGE = "JavaScript1.2">
A = ["Frank", "Emily", "Jane", "Harry", "Nick", "Beth", "Rick", \
    "Terrence", "Carol", "Ann", "Terry", "Frank", "Alice", "Rick", \
    "Bill", "Tom", "Fiona", "Jane", "William", "Joan", "Beth"]
re = /\w+/i

function lookup() {
    firstName = re.exec()
    if (!firstName)
        window.alert (RegExp.input + " isn't a name!")
    else {
        count = 0
        for (i=0; i<A.length; i++)
            if (firstName[0].toLowerCase() == A[i].toLowerCase())
                count++
        if (count ==1)
            midstring = " other has "
        else
            midstring = " others have "
```

```

        window.alert ("Thanks, " + count + midstring + "the same name!")
    }
}
</SCRIPT>

Enter your first name and then press Enter.

<FORM> <INPUT TYPE:"TEXT" NAME="FirstName" onChange="lookup(this);"> </
FORM>

</HTML>

```

test

Core method. Executes the search for a match between a regular expression and a specified string.

Syntax

```
regex.test(str)
```

Parameters

regex is the name of the regular expression.

str is the string against which to match the regular expression.

Description

When you want to know whether a pattern is found in a string use the **test** method (similar to the *String* search method); for more information (but slower execution) use the *exec* method (similar to the *String* match method).

Example

The following example prints a message which depends on the success of the test.

```
function testinput(re, str){  
    if (re.test(str))  
        midstring = " contains ";  
    else  
        midstring = " does not contain ";  
    document.write (str + midstring + re.source);  
}
```

test

Signed Scripts

For additional functionality, scripts can gain access to restricted information. This is achieved through signed scripts that request expanded privileges. The digital signature allows the user to confirm the validity of the certificate used to sign the script. It also allows the user to ensure that the script hasn't been tampered with since it was signed. The user then can decide whether to grant privileges based on the validated identity of the certificate owner and validated integrity of the script.

Note This functionality provides greater security than tainting. Tainting has been disabled.

This section contains:

- Signed Script Requirements
- Creating Signed Scripts
- JavaScript Features Requiring Privileges
- Example
- Accessing Expanded Privileges Without Signed Scripts
- Error Checking

Recommended Reading

"Netscape Object Signing" provides a list of documents and resources that provide information on Object Signing, from creating the Java applet to getting a certificate to packaging and signing it

"Object-Signing Tools" provides information about the signing tools that allow you to create and manipulate JAR archives and digitally sign the files they contain. Tools include JAR Packager, JAR Packager Command Line Edition, and Page Signer.

Signed Script Requirements

You can sign JavaScript files, in-line scripts and event handler scripts. You cannot sign `javascript:` URLs or JavaScript entities .

Signed scripts require:

- An ARCHIVE attribute in the <SCRIPT> tag
- An ID Attribute (in-line and event handler scripts only)
- A request for expanded privileges
- That all scripts on the page be signed
- Re-signing if changed

ARCHIVE attribute

All signed scripts (JavaScript file, in-line, event handler) require the <SCRIPT> tag's ARCHIVE attribute whose value is the name of the Java archive (JAR) file containing the digital signature. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" SRC="myJavaScript.js">
</SCRIPT>
```

Event handler scripts do not directly specify the ARCHIVE; instead, the handler must be preceded by a script containing ARCHIVE. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="1">
```

```

...
</SCRIPT>

<FORM>
<INPUT TYPE="button" VALUE="OK" onClick="alert('A signed script')"
ID="2">
</FORM>

```

Unless you are using more than one JAR file, you need only specify it once. Include the ARCHIVE tag in the first script on the HTML page and the remaining scripts on the page will use the same file. For example:

```

<SCRIPT ARCHIVE="myArchive.jar" ID="1">
document.write("This script is signed.");
</SCRIPT>

<SCRIPT ID="2">
document.write("This script is signed too.");
</SCRIPT>

```

ID Attribute

Signed in-line and event handler scripts require the ID attribute whose value is a string that relates the script to its signature in the JAR file. The ID must be unique within a JAR file.

When more than one event handler script exists in a tag, you only need one ID. The entire tag is signed as one piece.

In the following example, the first three scripts use the same JAR file. The third script accesses a JavaScript file so it doesn't use the ID tag. The fourth script uses a different JAR file, and its ID of "1" is unique to that file.

```

<HTML>

<SCRIPT ARCHIVE="firstArchive.jar" ID="1">
document.write("This is a signed script.");
</SCRIPT>

<BODY onLoad="alert('A signed script using firstArchive.jar')"
      onLoad="alert('One ID needed for these event handler scripts')"
ID="2">

<SCRIPT SRC="myJavaScript.js">
</SCRIPT>

<SCRIPT ARCHIVE="secondArchive.jar" ID="1">
document.write("This script uses the secondArchive.jar file.");
</SCRIPT>

```

```
</BODY>
```

```
</HTML>
```

Request Expanded Privileges

The script must include a function that calls Netscape's Java security classes and requests expanded privileges.

This requires one line of code that asks permission to access *someTarget* representing the resource you want to access. Targets are described below. For example:

```
netscape.security.PrivilegeManager.enablePrivilege("someTarget");
```

When the script calls this function, the signature is verified, and if the signature is valid, expanded privileges are granted. If necessary, a dialog displays with information about the application's author, and gives the user the option to grant or deny expanded privileges.

Java classes are explained in "Java Capabilities API."

Privileges are granted only in the scope of the requesting function and only after the request has been granted. This includes any functions called by the requesting function. When the script leaves that function, privileges no longer apply.

The example below demonstrates this by printing:

```
7: disabled
5: disabled
2: disabled
3: enabled
1: enabled
4: enabled
6: disabled
8: disabled
```

Function `g` requests expanded privileges, and only the commands and functions called *after* the request and *within* function `g` are granted privileges.

```
<SCRIPT ARCHIVE="ckHistory.jar" ID="1">
function printEnabled(i) {
  if (history[0] == "") {
```

```

        document.write(i + ": disabled<br>");
    } else {
        document.write(i + ": enabled<br>");
    }
}

function f() {
    printEnabled(1);
}

function g() {
    printEnabled(2);
    netscape.security.PrivilegeManager.enablePrivilege( "UniversalBrowserRead" );
    printEnabled(3);
    f();
    printEnabled(4);
}

function h() {
    printEnabled(5);
    g();
    printEnabled(6);
}

printEnabled(7);
h();
printEnabled(8);
</SCRIPT>

```

Sign All Scripts

For any one script to request privileges, all scripts on the HTML page or layer must be signed. If you are using layers, you can have both signed and unsigned scripts as long as you keep them in separate layers.

You can sign JavaScript files (accessed with the `<SCRIPT>` SRC attribute), in-line scripts, and event handler scripts. You cannot sign javascript: URLs or JavaScript entities. If a `javascript: URL`, a JavaScript entity or an unsigned script is included on a page with signed scripts, the signed scripts act as if they had not been signed.

Re-sign Changed Scripts

Changed scripts must be re-signed.

Changes to a signed script's byte stream invalidate the script's signature. This includes moving the HTML page between platforms that have different representations of text. For example, moving an HTML page from a Windows server to a UNIX server changes the byte stream and invalidates the signature. (This doesn't affect viewing pages from multiple platforms.) To avoid this, you can move the page in binary mode. Note that doing so changes the appearance of the page in your text editor but not in the browser.

During development, you can request expanded privileges without signing the script by activating codebased principles as explained in "Accessing Expanded Privileges Without Signed Scripts."

Creating Signed Scripts

1. Include the ARCHIVE and ID attributes in the <SCRIPT> tag (ID for in-line and event handler scripts only).
2. Include calls to Java classes requesting expanded privileges.
3. Sign the script. For information see "Object-Signing Tools."

If a window with frames needs to capture events in pages loaded from different locations (servers), use `enableExternalCapture` in a signed script requesting `UniversalBrowserWrite` privileges. Use this method before calling the **`captureEvents`** method.

For a signed script to provide properties, functions, and objects to other signed or unsigned scripts, use the `export` statement. The script wishing to import these exported features needs to use the `import` statement.

International Characters in Signed Scripts

When used in scripts, international characters may appear in string constants and in comments (JavaScript keywords and variables cannot include special international characters). Scripts that include international characters cannot be signed because the process of transforming the characters to the local character set will invalidate the signature. To work around this limitation:

- Escape the international characters ('0x\ea', etc.)
- Put the data containing the international characters in a hidden form element, and access the form element through the signed script.
- Separate signed and unsigned scripts into different layers, and use the international characters in the unsigned scripts
- Remove comments that include international characters

Note There is no restriction on international characters the HTML surrounding the signed scripts.

Targets

The types of information you can access are called targets. These are listed below.

Target	Description
UniversalBrowserRead	allows reading of privileged data from the browser.
UniversalBrowserWrite	allows modification of privileged data in a browser.
UniversalFileRead	allows a script to set the 'file' part of a file upload widget. This allows an arbitrary local file to be uploaded to wherever the form is submitted.
UniversalPreferencesRead	allows the script to read preferences using the navigator.preference() method.
UniversalPreferencesWrite	allows the script to set preferences using the navigator.preference() method.
UniversalSendMail	allows the program to send mail in the user's name.

For a complete list of targets, see "Introduction to the Capabilities Classes."

JavaScript Features Requiring Privileges

The following table lists the JavaScript features that require privileges and the target used to access the feature.

Feature	Target
<i>event</i> object: setting any property	UniversalBrowserWrite
<i>history</i> object: Getting the value of any property Setting the preference property	UniversalBrowserRead UniversalBrowserWrite
DragDrop event: getting the value of the data property	UniversalBrowserRead
<i>navigator</i> object: Getting the value of a preference using the preference method. Setting the value of a preference using the preference method.	UniversalPreferencesRead UniversalPreferencesWrite

Feature	Target
<p><i>window</i> object: Adding or removing:</p> <ul style="list-style-type: none"> • directory bar • location bar • menu bar • personal bar • scroll bar • status bar • toolbar <p>Using methods:</p> <p>enableExternalCapture – when a window wants to capture events in pages loaded from different servers. Follow this method with captureEvents.</p> <ul style="list-style-type: none"> • close – unconditional ability to close a browser window. • moveBy – to move a window off screen • moveTo – to move a window off screen • open – when using <ul style="list-style-type: none"> • innerWidth, innerHeight, outerWidth, and outerHeight to create a window smaller than 100 x 100 pixels or larger than the screen can accommodate • screenX and screenY to place a window off screen • titlebar to create a window without a titlebar • alwaysRaised, alwaysLowered, z-lock for any setting • resizeTo – to resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate • resizeBy – to resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate <p>Setting properties:</p> <ul style="list-style-type: none"> • innerWidth – to set the inner width of a window to a size smaller than 100 x 100 or larger than the screen can accommodate • innerHeight – to set the inner height of a window to a size smaller than 100 x 100 or larger than the screen can accommodate 	UniversalBrowserWrite
Setting a file upload widget	UniversalFileRead

Feature	Target
Submitting a form to mailto: or news: URL	UniversalSendMail
Using an "about:" URL other than "about:blank"	UniversalBrowserRead

Example

The following script includes a button, that, when clicked, displays an alert dialog containing part of the URL history of the browser. To work properly, the script must be signed.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="1">

function getHistory(i) {
    //Attempt to access privileged information
    return history[i];
}

function getImmediateHistory() {
    //Request privilege
    netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
    return getHistory(1);
}

</SCRIPT>

...

<INPUT TYPE="button" onClick="alert(getImmediateHistory());" ID="2">
```

Accessing Expanded Privileges Without Signed Scripts

Situations exist where you want to access privileged information without using signed scripts. Such a situation might be during development when you want to test your program, change code, retest, etc., and you don't want to sign the script after each change. You can request expanded privileges without signing the script by activating codebased principles. With codebase principals activated, Communicator allows the URL of the script to function as a principal for enabling privileges.

Risks

An unsigned script is vulnerable to tampering and should be used as a temporary measure or in the confines of an intranet.

Activating Codebased Principles

To activate codebased principles:

1. Users of your program need to add the following line to their Netscape preferences file:

```
user_pref("signed.applets.codebase_principal_support", true);
```

File location varies from platform to platform. The following are likely locations:

- On Windows, \Program Files\Netscape\Users\default\prefs.js
- On Windows NT, \Netscape\Users\username\prefs.js
- On UNIX, ~/.netscape/preferences.js
- On Macintosh, System Folder:Preferences:Netscape f: (where the last f is a special script f character)

All instances of Communicator must be shut down before editing this file. After editing, start Communicator.

2. Write the script with calls to the Java Class requesting expanded privileges. You can include an ARCHIVE and ID, but they aren't required until you sign the script.

When the user accesses the script, a dialog displays similar to the one displayed with signed scripts. The difference is that this dialog asks the user to grant privileges based on the URL and doesn't provide author verification. It advises the user that the script has not been digitally signed and may have been tampered with.

Note If a page includes signed scripts and codebased scripts, and `signed.applets.codebase_principal_support` is enabled, all of the scripts on that page are treated as though they are unsigned and codebased principles apply.

Error Checking

To check for errors during development, open the Java Console which displays error messages. In the browser, choose Communicator > Java Console.

Statements

Statements are core to the JavaScript language. This section includes new statements and changed statements.

break

The **break** statement can now include an optional label that allows the program to break out of a labeled statement. This type of break must be in a statement identified by the label used by **break**.

The statements in a labeled statement can be of any type.

Syntax

```
break label
```

Argument

label is the identifier associated with the label of the statement.

Example

In the following example, a statement labeled `checkiandj` contains a statement labeled `checkj`. If **break** is encountered, the program breaks out of the `checkj` statement and continues with the remainder of the `checkiandj` statement. If **break** had a label of `checkiandj`, the program would break out of the `checkiandj` statement and continue at the statement following `checkiandj`.

```
checkiandj :
  if (4==i) {
    document.write("You've entered " + i + "<BR>");
    checkj :
      if (2==j) {
        document.write("You've entered " + j + "<BR>");
        break checkj;
        document.write("The sum is " + (i+j) + "<BR>");
      }
    document.write(i + "-" + j + "=" + (i-j) + "<BR>");
  }
```

See Also

labeled statement

switch statement

continue

The **continue** statement can now include an optional label that allows the program to terminate execution of a labeled statement and continue to the specified labeled statement. This type of `continue` must be in a looping statement identified by the label used by **continue**.

Syntax

```
continue label
```


Argument

label is the identifier associated with the label of the statement.

Example

In the following example, a statement labeled `checkiandj` contains a statement labeled `checkj`. If **continue** is encountered, the program continues at the top of the `checkj` statement. Each time **continue** is encountered, `checkj` re-iterates until its condition returns false. When false is returned, the remainder of the `checkiandj` statement is completed. `checkiandj` re-iterates until its condition returns false. When false is returned, the program continues at the statement following `checkiandj`.

If **continue** had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

```
checkiandj :
    while (i<4) {
        document.write(i + "<BR>");
        i+=1;
        checkj :
            while (j>4) {
                document.write(j + "<BR>");
                j-=1;
                if ((j%2)==0);
                    continue checkj;
                document.write(j + " is odd.<BR>");
            }
        document.write("i = " + i + "<br>");
        document.write("j = " + j + "<br>");
    }
```

See Also

labeled statement

do while statement

The **do while** statement executes its statements until the test condition evaluates to false. Statement is executed at least once.

Syntax

```
do
    statement
while (condition);
```

Arguments

statement is a block of statements that is executed at least once and is re-executed each time the condition evaluates to true.

condition is evaluated after each pass through the loop. If *condition* evaluates to true, the statements in the preceding block are re-executed. When *condition* evaluates to false, control passes to the statement following **do while**.

Example

In the following example, the do loop iterates at least once and re-iterates until *i* is no longer less than 5.

```
do {
    i+=1
    document.write(i);
} while (i<5);
```

export

The **export** statement allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.

Syntax

```
export name1, name2, ..., nameN
```

-OR-

```
export *
```

Parameters

nameN is a list of properties, functions, and objects to be exported.

* exports all properties, functions, and objects from the script.

Description

Typically, information in a signed script is available only to scripts signed by the same principals. By exporting properties, functions, or objects, a signed script makes this information available to any script (signed or unsigned). The receiving script uses the companion **import** statement to access the information.

See Also

import

import

The **import** statement allows a script to import properties, functions, and objects from a signed script which has exported the information.

Syntax

```
import objectName.name1, objectName.name2, ..., objectName.nameN
```

import

-OR-

```
import objectName.*
```

Parameters

nameN is a list of properties, functions, and objects to import from the export file.

objectName is the name of the object that will receive the imported names. For example, if *f* and *p* have been exported, and if *obj* is an object from the importing script, then

```
import obj.f, obj.p
```

will make *f* and *p* accessible in the importing script as properties of *obj*.

* imports all properties, functions, and objects from the export script.

Description

Typically, information in a signed script is available only to scripts signed by the same principals. By exporting (using the **export** statement) properties, functions, or objects, a signed script makes this information available to any script (signed or unsigned). The receiving script uses the **import** statement to access the information.

The script must load the export script into a window, frame, or layer before it can import and use any exported properties, functions, and objects.

See Also

export

labeled statement

A labeled statement provides an identifier that can be used with **break** or **continue** to indicate where the program should continue execution.

In a labeled statement, **break** or **continue** must be followed with a label, and the label must be the identifier of the labeled statement containing **break** or **continue**.

Syntax

```
label :  
    statement
```

Arguments

statement is a block of statements. **break** can be used with any labeled statement, and **continue** can be used with looping labeled statements.

Example

For an example of a labeled statement using **break**, see [break](#).

For an example of a labeled statement using **continue**, see [continue](#).

See Also

[break](#), [continue](#)

switch statement

A **switch** statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

The program first looks for a label matching the value of *expression* and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of **switch**.

The optional **break** statement associated with each case label ensures that the program breaks out of **switch** once the matched statement is executed and continues execution at the statement following **switch**. If **break** is omitted, the program continues execution at the next statement in the switch statement.

Syntax

```
switch (expression){  
    case label :  
        statement;  
        break;  
    case label :  
        statement;  
        break;  
    ...  
    default : statement;  
}
```

Arguments

expression is the value matched against *label*.

label is an identifier used to match against *expression*.

statement is any statement.

Example

In the following example, if *expression* evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When **break** is encountered, the program breaks out of **switch** and executes the statement following **switch**. If **break** were omitted, the statement for case "Cherries" would also be executed.

```
switch (i) {
    case "Oranges" :
        document.write("Oranges are $0.59 a pound.<BR>");
        break;
    case "Apples" :
        document.write("Apples are $0.32 a pound.<BR>");
        break;
    case "Bananas" :
        document.write("Bananas are $0.48 a pound.<BR>");
        break;
    case "Cherries" :
        document.write("Cherries are $3.00 a pound.<BR>");
        break;
    default :
        document.write("Sorry, we are out of " + i + ".<BR>");
}

document.write("Is there anything else you'd like?<BR>");
```

switch statement

Style Sheets

Miscellaneous Features

Activating JavaScript Commands From the Personal Toolbar

The personal toolbar, new in Navigator 4.0, provides simplified access to links, commands, and page location information. It is located below the menu bar in the Navigator window. Besides adding links to web pages, you can add JavaScript methods that are activated when you click on their corresponding button in the toolbar. For example, you can add a method that opens a new window. To add a JavaScript method to the personal toolbar, you need to create a bookmark and define a command for that bookmark (instead of a link).

1. In the location bar, choose Bookmarks > Edit Bookmarks...

This opens the Bookmarks window.

2. Open the Personal Toolbar Folder, and select the Folder.

Opening and selecting the folder insures that the new bookmark appears in the Personal Toolbar.

3. Choose File > New Bookmark...

This opens the Bookmark Properties dialog.

4. In the Name field, type the name you want to appear on the toolbar.

For example, if you are writing a command that opens a window, you might have a name of "Open Window."

5. In the Location(URL) field, type the command using the form `javascript:void(command)`. Only JavaScript methods can be used.

For example, to open a window, use
`javascript:void(window.open(" "))`

Using `void` ensures that the original page is left unchanged.

6. Click OK and close the Bookmarks window.

A new button appears on the Personal Toolbar. Clicking the button activates the command.