JavaScript

Appendices

December 20, 2015 | Proprietary and Confidential | - 1 -

Appendix B : Predefined Objects – Methods

String : The string object is used to manipulate a stored piece of text. It has a number of methods and properties.

String.charCodeAt([index]): Returns: Integer code number for a character; concatenated string value of code numbers supplied as parameters.

String.fromCharCode(num1 [, num2 [, ...numn]])
"abc".charCodeAt() // result = 97
"abc".charCodeAt(0) // result = 97
"abc".charCodeAt(1) // result = 98

To convert numeric values to their characters, use the String.fromCharCode() method. Notice that the object beginning the method call is the generic string object, not a string value. Then, as parameters, you can include one or more integers separated by commas. In the conversion process, the method combines the characters for all of the parameters into one string, an example of which is shown here.

String.fromCharCode(97, 98, 99) // result "abc"

string.indexOf(searchString [, startIndex])

Returns: Index value of the character within string where searchString begins.

string.lastIndexOf(searchString [, startIndex])

Returns: Index value of the last character within string where searchString begins.

JavaScript's indexOf() method enables your script to obtain the number of the character in the main string where a search string begins. Optionally, you can specify where in the main string the search should begin —but the returned value is always relative to the very first character of the main string. Like all string object methods, index values start their count with 0. If no match occurs within the main string, the returned value is -1. Thus, this method is a convenient way to determine whether one string contains another.

The string.lastIndexOf() method is closely related to the string.indexOf() method. The only difference is that this method starts its search for a match from the end of the string (string.length - 1) and works its way backward through the string. All index values are still counted, starting with 0, from the front of the string. In the examples that follow, we use the same values as in the examples for string.IndexOf so that you can compare the results. In cases where only one instance of the search string is found, the results are the same; but when multiple instances of the search string exist, the results can vary widely — hence the need for this method.

String Objects (Parsing Methods contd..):

Example	f = indexOf	f = lastIndexOf
offset = “bananas”.f(“b”)	0	0
offset = “bananas”.f(“a”)	1	5
offset = “bananas”.f(“a”,1)	1	1
offset = “bananas”.f(“a”,2)	3	1
offset = “bananas”.f(“a”,4)	5	3
offset = “bananas”.f(“nan”)	2	2
offset = “bananas”.f(“nas”)	4	4
offset = “bananas”.f(“s”)	6	6
offset = “bananas”.f(“z”)	-1	-1

Table 4.1 Comparison of output of indexOf() and lastIndexOf() functions

string.split(“delimiterCharacter”[, limitInteger])

Returns: Array of delimited items.

var myString = “Anderson,Smith,Johnson,Washington”

var myArray = myString.split(“,”)

var itemCount = myArray.length // result: 4

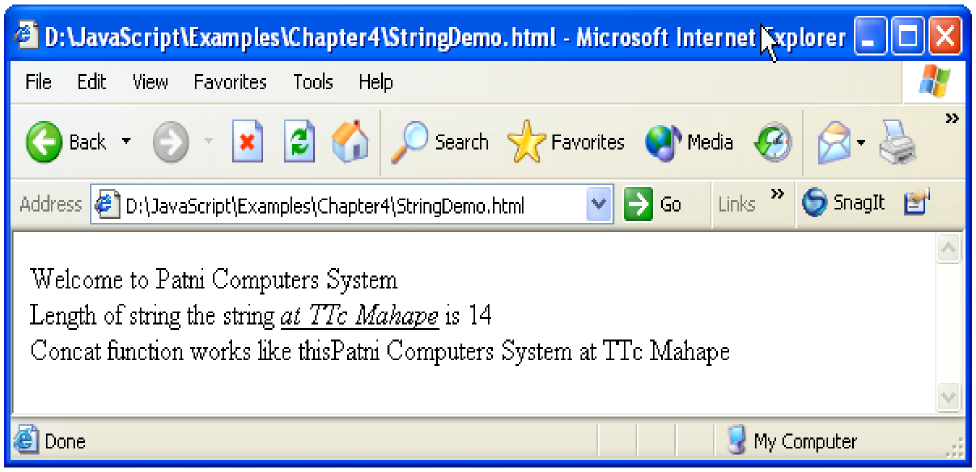
The function splits a long string into pieces delimited by a specific character and then creates a dense array with those pieces. You do not need to initialize the array via the new Array() constructor. Given the powers of array object methods such as array.sort(), you may want to convert a series of string items to an array to take advantage of those powers. Also, if your goal is to divide a string into an array of single characters, you can still use the split() method, but specify an empty string as a parameter.

string.substring(indexA, indexB)
Returns: Characters of string between index values indexA and indexB.
The string.substring() method enables your scripts to extract a contiguous range of characters from any string. The parameters to this method are the starting and ending index values (first character of the string object is index value 0) of the main string from which the excerpt should be taken. An important item to note is that the excerpt goes up to, *but does not includ*e, the character pointed to by the higher index value.
It makes no difference which index value in the parameters is larger than the other: The method starts the excerpt from the lowest value and continues to (but does not include) the highest value. If both index values are the same, the method returns an empty string; and if you omit the second parameter, the end of the string is assumed to be the endpoint.
string.search(regExpression)
Returns: Offset Integer.
The results of the string.search() method should remind you of the string.indexOf() method. In both cases, the returned value is the index number where the matching string first appears in the main string, or -1 if no match occurs. The big difference, of course, is that the matching string for string.search() is a regular expression.
string.slice(startIndex [, endIndex])
Returns: String.
string.substring(4, (string.length-2))
string.slice(4, -2)
The string.slice() method (new in Navigator 4) resembles the string.substring() method in that both let you extract a portion of one string and create a new string as a result (without modifying the original string). A helpful improvement in string.slice(), however, is that it is easier to specify an ending index value relative to the end of the main string. The code snippet compares the substring and slice methods in extracting a substring that ends before the end of the string. You can assign a negative number to the second parameter of string.slice() to indicate an offset from the end of the string. The second parameter is optional. If you omit it, the returned value is a string from the starting offset to the end of the main string.
String object also provides with some formatting methods:
string.anchor("anchorName") : Creates an HTML anchor
string.blink() : Displays a blinking string
string.link(locationOrURL) : Displays a string as a hyperlink
string.fixed() : Displays a string as teletype text
string.strike() : Displays a string with a strikethrough
string.sub() : Displays a string as subscript
string.sup() : Displays a string as superscript

```
<html>
<head>
</script>
</head>
<body>
<script language="JavaScript">
var x="Patni Computers System"
var y=" at TTc Mahape"
document.write("Welcome to "+x);
document.write("<BR>Length of string the string <u>“
+ y.italics()+"</u> is "+y.length);
document.write("<BR>Concat function works like this"+x.concat(y))
</script>
</body>
</html>
```

Example 4.1 Demo of String Object (StringDemo.html)

And it produces the output as:


D:\JavaScript\Examples\Chapter4\StringDemo.html - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Media
Address
D:\JavaScript\Examples\Chapter4\StringDemo.html
Go
Links
SnagIt
Done
My Computer

```
Welcome to Patni Computers System
Length of string the string at TTc Mahape is 14
Concat function works like thisPatni Computers System at TTc Mahape
```

Math : The Math object allows you to perform mathematical tasks. It is not a constructor. All properties and methods can be called using Math as an object without creating it.

Property/Method	Description
Math.E	Euler's Constant (2.718)
Math.LN	Natural Log of 2 (0.693)
Math.LN10	Natural Log of 10 (2.302)
Math.LOG2E	Log base-2 of E (1.442)
Math.LOG10E	Log base-10 of E (0.434)
Math.SQRT1_2	Square Root of ½ (0.707)
Math.abs(x)	Returns the absolute value of x
Math.acos(val)	Arc cosine (in radians) of *val*
Math.asin(val)	Arc sine (in radians) of *val*
Math.atan(val)	Arc tangent (in radians) of *val*
Math.atan2(val1, val2)	Angle of polar coordinates *x* and *y*
Math.cos(val)	Cosine of *val*
Math.exp(val)	Euler's constant to the power of *val*
Math.pow(val1, val2)	*Val1* to the *val2* power
Math.sin(val)	Sine (in radians) of *val*
Math.sqrt(val)	Square root of *val*
Math.tan(val)	Tangent (in radians) of *val*
Math.log(val)	Natural logarithm (base e) of val
Math.ceil(x)	Returns x, rounded upwards to nearest integer
Math. Floor(x)	Returns x, rounded downwards to the nearest integer

Number: The number object is rarely used, because for the most part, JavaScript satisfies day-to-day numeric needs with a plain number value. But the number object contains some information and power of value to serious programmers. Primarily, there are properties that define the ranges for numbers in the language. The largest number (in both Navigator and Internet Explorer) is 1.79E+308; the smallest number is 2.22E-308. Any number larger than the maximum is POSITIVE_INFINITY; any number smaller than the minimum is NEGATIVE_INFINITY. It will be a rare day on which you accidentally encounter these values.

Similar to the string prototype property , the Number.prototype property adds a method to a Number object's prototype such that every newly created object contains that method. values. If you have a need to add common functionality to every number object, this is where to do it.
This prototype facility is unique to objects and does not apply to plain number values. JavaScript number objects and values are defined internally as IEEE double-precision 64-bit
var val = new Number(number)

Properties	**Methods**
MAX_VALUE	toString()
MIN_VALUE	
NaN	
NEGATIVE_INFINITY	
POSITIVE_INFINITY	
prototype	

Boolean Objects: Properties and Methods
var val = new Boolean(BooleanValue)
You work with Boolean values a lot in JavaScript - especially as the result of conditional tests. Just as string values benefit from association with string objects and their properties and methods, so, too, do Boolean values receive aid from the Boolean object. For example, when you display a Boolean value in a text box, the "true" or "false" string is provided by the Boolean object's toString() method, even though you don't have to invoke it directly.
The only time you need to even think about a Boolean object is if you wish to attach some property or method to Boolean objects that you create with the new Boolean() constructor. Parameter values for the constructor include the string versions of the values, numbers (0 for false; any other integer for true), and expressions that evaluate to a Boolean value. Any such new Boolean object would be imbued with the new properties or methods you have added to the prototype property of the core Boolean object.

Property	Method
Prototype	toString()

Date: The date object is used to work with Date and time values. JavaScript maintains its date information in the form of a count of milliseconds starting from January 1, 1970, in the GMT time zone

Properties	**Description**
dateObj.getDate()	Date within the month
dateObj.getDay()	Day of week (Sunday = 0)
dateObj.getHours()	Hour of the day in 24-hour time
dateObj.getMinutes()	Minute of the specified hour
dateObj.getSeconds()	Second within the specified minute
dateObj.setTime(val)	Milliseconds since 1/1/70 00:00:00 GMT
dateObj.setYear(val)	Specified year minus 1900
dateObj.setMonth(val)	Month within the year (January = 0)
dateObj.setDate(val)	Date within the month
dateObj.setDay(val)	Day of week (Sunday = 0)
dateObj.setHours(val)	Hour of the day in 24-hour time
dateObj.setMinutes(val)	Minute of the specified hour
dateObj.setSeconds(val)	Second within the specified minute
Date.parse("dateString")	Converts string date to milliseconds
Date.UTC(date values)	Generates a date value from GMT values

For example:

```
myBirthday = new Date("September 11, 1996")
result = myBirthday.getDay() // result = 3, a Wednesday
myBirthday.setYear(97) // bump up to next year
result = myBirthday.getDay() // result = 4, a Thursday
```

Array object and its methods:

After you have information stored in an array, JavaScript provides several methods to help you manage that data.

```
arrayObject.concat(array2)
var array1 = new Array(1,2,3)
var array2 = new Array("a","b","c")
var array3 = array1.concat(array2)
// result: array with values 1,2,3,"a","b","c"
```

The ***array.concat()*** method allows you to join together two array objects into a new, third array object. The action of concatenating the arrays does not alter the contents or behavior of the two original arrays. To join the arrays together, you refer to the first array object to the left of the period before the method; a reference to the second array is the parameter to the method.

If an array element is a string or number value (not a string or number object), then the values are copied from the original arrays into the new one. All connection with the original arrays ceases for those items. However, if an original array element is a reference to an object of any kind, then JavaScript copies a reference from the original array's entry into the new array. This means that if you make a change to either array's entry, the change occurs to the object, and both array entries reflect the change to the object.

Array.slice(

Behaving like its like-named string method, ***array.slice()*** lets you extract a contiguous series of items from an array. The extracted segment becomes an entirely new array object. One parameter is required — the starting index point for the extraction. If you do not specify a second parameter, then the extraction goes all the way to the end of the array. Otherwise the extraction goes to, but does not include, the index value supplied as the second parameter.

```
arrayObject.slice(startIndex [, endIndex])
Returns: Array
var solarSys = new
Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
"Uranus","Nep
tune","Pluto")
var nearby = solarSys.slice(1,4)
// result: new array of "Venus", "Earth", "Mars"
```

arrayObject.sort([compareFunction])

It returns array of entries in the order as determined by the compareFunction algorithm.

```
myArray = new Array(12, 5, 200, 80)
function compare(a,b) {
return a - b
}
myArray.sort(compare)
function compare(a,b) {
// last character of array strings
var aComp = a.charAt(a.length - 1)
var bComp = b.charAt(b.length - 1)
if (aComp < bComp) {return -1}
if (aComp > bComp) {return 1}
return 0
}
```

When no parameter is specified, JavaScript takes a snapshot of the contents of the array and converts items to strings. From there, it performs a string sort of the values. **ASCII values** of characters govern the sort, which means that numbers are sorted by their **string values**, not their **numeric values**. This fact has strong implications if your array consists of numeric data. The value 201 sorts before 88, because the sorting mechanism compares the first characters of the strings ("2" versus "8") to determine the sort order. For simple alphabetical sorting of string values in arrays, the plain ***Array.sort()*** method should do the trick.

You can define a function that helps the ***sort()*** method compare items in the array. A comparison function is passed two values from the array and it lets the ***sort()*** method know which of the two items comes before the other based on the value the function returns.

The array has four numeric values in it. To sort the items in numerical order, you define a comparison function (arbitrarily named ***compare()***), which is called from the ***sort()*** method. Note that unlike invoking other functions, the parameter of the ***sort()*** method uses a reference to the function, which lacks parentheses.

When the ***compare()*** function is called, JavaScript automatically sends two parameters to the function in rapid succession until each element has been compared against the others. Every time ***compare()*** is called, JavaScript assigns two of the array's values to the parameter variables (a and b).

In this example, the returned value is the difference between a and b. If a is larger than b, then a positive value goes back to the ***sort()*** method, telling it to sort a above b (that is, position a at a lower value index position than b). Therefore, a may end up at ***myArray[0]***, whereas b ends up at a higher index-valued location. On the other hand, if b is larger than a, then the returned negative value tells ***sort()*** to put b in a lower index value spot than a.

The second function sorts alphabetically by the last character of each array string entry.
First, this function extracts the final character from each of the two values passed to it. Then, because strings cannot be added or subtracted like numbers, you compare the ASCII values of the two characters, returning the corresponding values to the ***sort()*** method to let it know how to treat the two values being checked at that instant.

```
<html>
<head>      <title> arrays </title></head>
<script language = "javascript">
solarSys = new Array("Mercury", "Venus", "Earth",
"Mars", "Jupiter", "Saturn", "Uranus", "Neptune",
"Pluto")
var array1 = solarSys.join(",")
document.write(array1)
document.write("<br>Slice three elements <br>")
document.write(solarSys.slice(3,6))
document.write("<br>REverse and join with ,")
s=solarSys.reverse()
var arrayText = s.join(",")
document.write(arrayText)
document.write("<br>solarsys after reversing<br>")
array1 = solarSys.join(",")
document.write(array1)
solarSys.reverse()
document.write("<br>Concat and join<br>")
s2=solarSys.concat(s)
document.write(s2.join(","))
solarSys.sort();
arrayText = s.join(",")
document.write("<br>Sort and join")
document.write(arrayText)</script>
</html>
```

JavaScript Document Object Model
Link Object

Event Handler	**Description**
onClick onDblClick	By and large, the HREF attribute determines the action that a link makes when a user clicks it — which is generally a navigational action. But if you need to execute a script before navigating to a specific link (or to change the contents of more than one frame), you can include an *onClick=* or *onDblClick=* event handler in that link's definition. Any statements or functions called by either click event handler execute before any navigation takes place.
onMouseDown onMouseUp	Events for the mouse being pressed and released supplement the long-standing click event. A click event fires after a matched set of *mouseDown* and *mouseUp* events occurs on the same link. If the user presses down on the link and slides the mouse pointer off the link to release the mouse button, only the *mouseDown* event fires on the link.
onMouseOver onMouseOut	As you drag the mouse pointer atop a link in a document, the status line at the bottom of the window shows the URL defined in the link's HREF attribute. You can override the display of a link's URL by triggering a function with the *onMouseOver* event handler assigned to a link. The *onMouseOut* handler fires when the pointer leaves the link's rectangular region. If you use *onMouseOver* to set the status bar, you should return the status bar to its default setting with an *onMouseOut* event handler.

Text Related Objects:

Event Handler	**Description**
onSelect	The onSelect event is fired when the user selects the text inside the text field.
onKeyDown onKeyPress onKeyUp	These events let your scripts capture user activity from the keyboard while the field has focus. The keyDown event occurs the instant the user presses the key far enough to make contact; the keyUp event occurs when electrical contact with the key breaks; and a keyPress event occurs after the keyUp event, signaling the completion of a matched pair of keyDown and keyUp events.

Select Object (Option Object)

Property	**Description**
options[index]	You typically won't summon this property by itself. Rather, it becomes part of a reference to a specific option's properties within the entire select object. In other words, the options property becomes a kind of gateway to more specific properties, such as the value assigned to a single option within the list.
options[index]. Index	The index value of any single option in a select object will likely be a redundant value in your scripting. Because you cannot access the option without knowing the index anyway, you have little need to extract the index value. The value is a property of the item, just the same.
options[index]. value	In many instances, the words in the options list appear in a form that is convenient for the document's users but inconvenient for the scripts behind the page. Rather than set up an elaborate lookup routine to match the selectedIndex or options[index].text values with the values your script needs, an easier technique is to store those values in the VALUE attribute of each <OPTION> definition of the select object. You can then extract those values as needed.

RegExp Object
Following table shows properties of RegExp object:

Properties	Methods	Events
input	None	None
lastMatch		
Lastparen		
LeftContext		
Multiline		
RightContext		
$1..$9		

Beginning with Navigator 4 and Internet Explorer 4, the browser maintains a single instance of a RegExp object for each window or frame. The object oversees the action of all methods that involve regular expressions (including the few related string object methods). Properties of this object are exposed not only to JavaScript in the traditional manner, but also to a parameter of the *string.replace()* method for some shortcut access.
With one RegExp object serving all regular expression-related methods in your document's scripts, you must exercise care to access or modify this object's properties. You must make sure that the RegExp object has not been affected by another method. Most properties are subject to change as the result of any method that involves a regular expression. This may be reason enough to use the properties of the array object returned by most regular expression methods instead of the RegExp properties. The former stick with a specific regular expression object even after other regular expression objects are used in the same script. RegExp properties reflect the most recent activity, irrespective of the regular expression object involved.

Properties
Input: *RegExp.input* property is the main string against which a regular expression is compared in search of a match.

However, many text-related document objects have an unseen relationship with RegExp. If a text, textarea, select, or link object contains an event handler that invokes a function that contains a regular expression, *RegExp.input* property is set to the relevant textual data from the object. You need not specify parameters for the event handler call or in the function called by it. For text and textarea objects, the input property value becomes the object content; for the select object, it is the text (not the value) of the selected option; and for a link, it is the text highlighted in the browser associated with the link (and reflected in the link's text property).To have JavaScript set the *RegExp.input* property for you may simplify your script. You can invoke either of the regular expression methods without specifying the main string parameter. When that parameter is empty, JavaScript applies the *RegExp.input* property to the task. You can also set this property on the fly if you like. The short version of this property is **$_** (dollar sign underscore).

Multiline: *RegExp.multiline* property determines whether searches extend across multiple lines of a target string. This property is automatically set to true when an event handler of a textarea triggers a function containing a regular expression. You can also set this property on the fly if you like. The short version of this property is **$***.

lastMatch: After you execute a regular expression-related method, any text in the main string that matches the regular expression specification is automatically assigned to the *RegExp.lastMatch* property. This value is also assigned to the [0] property of the object array returned when a match is found by the *exec()* and *string.match()* methods. The short version of this property is **$&**.

lastParen: When a regular expression contains many parenthesized subcomponents, the RegExp object maintains a list of the resulting strings in the $1,...$9 properties. You can also extract the value of the last matching parenthesized subcomponent with the *RegExp.lastParen* property, which is a read-only property. The short version of this property is **$+**.

leftContext, rightContext

After a match is found in the course of one of the regular expression methods, the RegExp object is informed of some key contextual information about the match. The *leftContext* property contains the part of the main string to the left of (up to but not including) the matched string. Be aware that the *leftContext* starts its string from the point at which the most recent search began. Therefore, for second or subsequent times through the same string with the same regular expression, the leftContext substring varies widely from the first time through. The *rightContext* consists of a string that starts immediately after the current match and extends to the end of the main string. As subsequent method calls work on the same string and regular expression, this value obviously shrinks in length until no more matches are found. At this point, both properties revert to null. Short versions of these properties are $` and $' for leftContext and rightContext, respectively.

$1…$9
As a regular expression method executes, any parenthesized result is stored in RegExp's nine properties reserved for just that purpose (called backreferences). The same values (and any beyond the nine that RegExp has space for) are stored in the array object returned with the *exec()* and *string.match()* methods.

Values are stored in the order in which the left parenthesis of a pair appears in the regular expression, regardless of nesting of other components.

You can use these backreferences directly in the second parameter of the *string.replace()* method, without using the RegExp part of their address. The ideal situation is to encapsulate components that need to be rearranged or recombined with replacement characters. For example, the following script function turns a name that is last name first into first name last:

```
function swapEm() {
var re = /(\w+),\s*(\w+)/
var input = "Lincoln, Abraham"
return input.replace(re,"$2 $1")
}
```

In the *replace()* method, the second parenthesized component (just the first name) is placed first, followed by a space and the first component. The original comma is discarded. You are free to combine these shortcut references as you like, including multiple times per replacement, if it makes sense to your application.
When you create a regular expression with the literal notation (that is, with the two forward slashes), the expression is automatically compiled for efficient processing as the assignment statement executes. The same is true when you use the new *RegExp()* constructor. Specify a pattern (and optional modifier flags) as a parameter. Whenever the regular expression is fixed in the script, use the literal notation; when some or all of the regular expression is derived from an external source (for example, user input from a text field), assemble the expression as a parameter to the new *RegExp()* constructor.
Use a compiled regular expression at whatever stage the expression is ready to be applied and reused within the script. Compiled regular expressions are not saved to disk or given any more permanence beyond the life of a document's script (that is, it dies when the page unloads).

However, there may be times in which the specification for the regular expression changes with each iteration through a loop construction. For example, if statements in a while loop modify the content of a regular expression, you should compile the expression inside the while loop, as shown in the following skeletal script fragment:

```
var srchText = form.search.value
var re = new RegExp() // empty constructor
while ( someCondition) {
re.compile("\\s+" + srchText + "\\s+", "gi")
statements that change srchText
}
```

Each time through the loop, the regular expression object is both given a new expression (concatenated with metacharacters for one or more white spaces on both sides of some search text whose content changes constantly) and compiled into an efficient object for use with any associated methods.