FieldCommander®

Javascript Reference Guide

FC-SW

Table of contents

About this manual

This document serves as a reference guide for the core Javascript language on which FieldCommander's scripting is based.

Although this document is targeted towards people who are not necessarilly experienced in programming, it only touches the generic concept of developing software. When you haven't written macros or scripts before, you might want to pick up a book on programming too.

Organisation of the documentation

Installation	*Installation Guide Hardware Platform 1* (FCHWP1IG) *Installation Guide Hardware Platform 2* (FCHWP2IG) Bundles information related to the hardware, such as the installation of the device, IP networking, serial interface details and technical specifications.
Employment	*User's Guide* (FCSWUG) Explains how FieldCommander is used in your application development, and documents all its features, including the databases, communication and web based configuration.
Programming	*FCscript Programmer's Guide* (FCSCRIPTPG) Function reference of FieldCommander's scripting language, used to control the data flow and event management in your applications. *FCphp Programmer's Guide* (FCPHPPG) Function descriptions of the FieldCommander specific PHP interface, used to build dynamic user interfaces or reports on top of the your application. *Javascript Reference Manual* (FCJSREF) Extensive documentation of the Javascript (ECMAscript) core language used by FieldCommander for application programming.
Software options	*Modbus Option Guide* (FCOPT10OG) *Galaxy Option Guide* (FCOPT20OG) *EIB/KNX Option Guide* (FCOPT30OG) *RDA Option Guide* (FCOPT40OG) The "plug in" modules (optionally available) have their own documentation to explain the features, installation, configuration and programming interfaces.

1. The Javascript language

Javascript is one of the most popular scripting language in today's world. It uses the ECMAScript standard for Javascript. ECMAScript is the core version of Javascript which has been defined by the European Computer Manufacturers Association and is the only standardization of Javascript.

This chapter starts with a quick introduction of the language. Then follow the functions and other programming concepts of the Javascript language. When you are comfortable with the C language, you might want to skip to chapter 3 which discusses the differences between Javascript and C.

1.1 Javascript quick start

This section provides examples and information to get started with Javascript Keep in mind that Javascript scripts may be written as simple scripts, much like simple batch files, in which lines of code execute sequentially, or they may be written as structured programs.

When a script has code outside of functions and code inside of functions, it shares characteristics of both batch and program scripts. For example, the following fragment:

```
writeLog("first ");

function main()
{
   writeLog("third.");
}

writeLog("second ");
```

results in the following output in the diagnostic log:

```
first second third.
```

1.1.1 Simple script

The following line is a simple and complete script.

```
writeLog('A simple script')
```

The `writeLog()` function is a simple way to get feedback from the script. This will create an entry in the diagnostics log file.

Any text editor may be used to work with script files. Assume that the single line of this example has been saved to a file named *simple.jse*. To execute this script, upload it FieldCommander and start the script from the System configuration. Refer to the User's Guide for details.

1.1.2 Date and time display

The following fragment:

```
var d = new Date
writeLog(d.toLocaleString())
```

produces output similar to the following.

```
Fri Oct 23 10:29:05 1998
```

The first line creates a variable d as a new Date object, or more accurately, as a new instance of a Date object. The second line uses the `Date toLocaleString()` method of the Date object to display local date and time information. This batch script could be written as a program script as shown in the following fragment.

```
function main()
{
   var d = new Date;
   writeLog(d.toLocaleString());
}
```

The `main() function`, if it exists, is the first function to be executed in a script. This script, using a structured programming style, produces the exact same result as the first two lines, which follow a batch style. The following fragment is another variation that produces the same result.

```
var d = new Date;

function main()
{
   writeLog(d.toLocaleString());
}
```

Remember that lines of script outside of functions are executed before the `main()` function. The following fragment is yet another variation.

```
function main()
{
   DisplayTime();
}

function DisplayTime()
{
   var d = new Date;
   writeLog(d.toLocaleString());
}
```

To repeat, the first fragment shown consists of two lines of code written as a simple batch script. The fragments, shown after it, are all written as program scripts. All of the fragments accomplish the same thing, namely, displaying local day, date, and time information. All fragments work equally well. What are the differences? If a user wanted a simple script to display date and time information, then the first batch script would likely be the best choice. However, if a user wanted to write a more involved program, one in which the display of the date and time was only a small part, then one of the program scripts would be the best choice. Remember, Javascript scripts may be as simple or as powerful as a user chooses.

1.1.3 Function with parameters

In the section above on date and time display, several variations of scripts were presented showing different ways to accomplish the same result. The last variation shown defined the function DisplayTime() which was called from the `main()` function. When DisplayTime() was called, no parameters, that is, no information or arguments, were passed to the function. Many times such functions are used, but often scripts need to be able to pass data or information to a function which then works with different data when called for differing reasons in a script. See the section on `passing information to functions` for more information about arguments and parameters.

The following script fragment illustrates the use of a function with parameters. The purpose of the fragment is to write a custom message to the diagnostics if the day of the week is Saturday. A detailed explanation follows.

```
var dat = new Date();

// Sun == 0 . . . Sat == 6
if (dat.getDay() == 6) {
   var FirstLine = "It is Saturday.";
   WriteMessage(FirstLine);
}

function WriteMessage(LineOne)
{
   writeLog(LineOne);
}

// The rest of the script follows
writeLog("The program is continuing.");
```

The first line creates a new `Date object`, which holds information about the current date and time that can be retrieved in various formats. In this script, the only date information used is the day of the week.

The third line of the script calls the method `Date getDay()` which returns the day of the week as a number. Sunday is the first day of the week and is zero. The Date object has many methods, such as `getDay()`, that are available to all Date objects that are created as in this example. The variable `dat` is only one instance of a Date object. A script can create or construct as many Date objects as desired, and each one may use all the methods of the Date object. However, if date information is altered in one instance, the date information in the other instances is not affected. This behavior, of constructing an object which is insulated from operations within other instances of the same type of Object, is the same for all objects, not just Date objects.

The third line tests, with an if statement, whether the current day is day number 6, Saturday. If the day is 6, then the variable, FirstLine is created with string information in it. Then the function WriteMessage() is called with the variable as the first parameter of the function.

The function WriteMessage() uses the information passed to it in its parameters: LineOne. Notice that the variable, FirstLine, does not have the same name as the parameter, LineOne. Arguments, such as FirstLine, do not have to have the same names as the parameters to which they are passed, in this case, LineOne. The variable FirstLine did not have to be created at all. The function WriteMessage() could have been called with a literal strings instead of a variable name. But such a line could become too long. The use of variables in the if statement makes the code easier to read and to alter. Without the variables, the call to `WriteMessage()` would have been:

```
 WriteMessage("It is Saturday.");
```

1.1.4 Terminology

Before going further, a little explanation of terminology might help. One problem with terminology is that it is has developed over the years and is not used uniformly. But in general, the term routine refers to a function or procedure that may be called in a program. A procedure is a routine that does something but does not return a value. A function is a routine that returns a value. Said another way, a procedure is a function that does not return a value.

In Javascript, the terms used are methods and functions, and these terms do not make the distinction between a function that returns a value and one that does not. The term procedure is not used. In the current discussion, the term routine is a general term used for functions and methods (and procedures, though this term is not used). The term method is normally used for a function that has been attached as a property of an object. The term function is used for functions of the global object and functions that a user defines that are not attached to a specific object. Such functions are actually methods of the global object.

The methods of the `global object` may be called without placing `global.` in front of the method name. Thus, they look like and act like plain functions in other languages, such as C. For example, the function `parseFloat()` is actually a method of the global object. The following fragment calls `parseFloat()` like a function.

```
 var n = parseFloat("3.21");
 writeLog(typeof n);
 writeLog(n);
```

The following fragment, which is the same as the one above with the addition of `global`, calls `parseFloat()` as a method, but both fragments are identical in behavior.

```
 var n = global.parseFloat("3.21");
 writeLog(typeof n);
 writeLog(n);
```

Thus, `parseFloat()` may be referred to as a function reflecting these calling conventions. The line displaying typeof n displays a number in both cases. The `typeof operator` returns the type of data of the value following it. The `typeof` operator may be invoked with "`()`". For example, `typeof n` and `typeof(n)` are the same.

The following fragment has a user defined function, MyFunction(), that is called like a function and then as a method. Both calls to MyFunction() are identical in behavior.

```
 function MyFunction()
 {
    writeLog("My function has been called.");
 }

 MyFunction();
 global.MyFunction();
```

In the current Javascript manual, the following distinctions generally are followed.

- The term routine is generally used for functions and methods.
- The term function is used for methods of the global object, that is, for methods that do not require an object name or name of an instance of an object to precede the method name.
- The term method is used for methods that require an object name or name of an instance of an object. The `Date getDay()` method, which was used above in the section about a function with parameters, is an example of such a method.

1.1.5 Function with a return

Functions may simply do something as the function ExitOnError() above does, or they may return a value to a calling routine. Of course, functions may do things and return values. The following fragment illustrates a function that returns a value.

```
function Cubed(n)
{
   return n * n * n;
} //Cubed

var CubedNumber = Cubed(3);
writeLog(CubedNumber);
```

The function Cubed() simply receives a number as parameter n, multiplies the number times itself three times, and returns the result. The variable CubedNumber is assigned the return value from the function Cubed(). CubedNumber is writen to the diagnostics file, and in this example, the number 9 is displayed.

1.2 Basics of Javascript

1.2.1 Case sensitivity

Javascript is case sensitive. A variable named "testvar" is a different variable than one named "TestVar", and both of them can exist in a script at the same time. Thus, the following code fragment defines two separate variables:

```
var testvar = 5
var TestVar = "five"
```

All identifiers in Javascript are case sensitive. For example, to display the word "dog" on the screen, the `writeLog()` method could be used: `writeLog("dog")`. But, if the capitalization is changed to something like, `WriteLog("dog")`, then the Javascript interpreter generates an error message. Control statements and preprocessor directives are also case sensitive. For example, the statement `while` is valid, but the word `While` is not. The directive `#if` works, but the letters `#IF` fail.

1.2.2 White space characters

White space characters, space, tab, carriage-return and new-line, govern the spacing and placement of text. White space makes code more readable for humans, but is ignored by the interpreter.

Lines of script end with a carriage-return, and each line is usually a separate statement.

(Technically, in many editors, lines end with a carriage-return and linefeed pair, "\r\n".) Since the interpreter usually sees one or more white space characters between identifiers as simply white space, the following Javascript statements are equivalent to each other:

```
var x=a+b
var x = a + b
var x =         a         +          b
var x = a
          + b
```

White space separates identifiers into separate entities. For example, "ab" is one variable name, and "a b" is two. Thus, the fragment, `var ab = 2` is valid, but `var a b = 2` is not.

Many programmers use all spaces and no tabs, because tab size settings vary from editor to editor and programmer to programmer. By using spaces only, the format of a script will look the same on all editors.

1.2.3 Comments

A comment is text in a script to be read by humans and not the interpreter which skips over comments. Comments help people to understand the purpose and program flow of a program. Good comments, which explain lines of code well, help people alter code that they have written in the past or that was written by someone else.

There are two formats for comments: end of line comments and block comments. End of line comments begin with two slash characters, "//". Any text after two consecutive slash characters is ignored to the end of the current line. The interpreter begins interpreting text as code on the next line. Block comments are enclosed within a beginning block comment, "/*", and an end of block comment, "*/". Any text between these markers is a comment, even if the comment extends over multiple lines. Block comments may not be nested within block comments, but end of line comments can exist within block comments.

The following code fragments are examples of valid comments:

```
// this is an end of line comment

/* this is a block comment
 This is one big comment block.
 // this comment is okay inside the block
 Isn't it pretty?
*/

var FavoriteAnimal = "dog"; // except for poodles

//This line is a comment but
var TestStr = "this line is not a comment";
```

1.2.4 Expressions, statements, and blocks

An expression or statement is any sequence of code that performs a computation or an action, such as the code `var TestSum = 4 + 3` which computes a sum and assigns it to a variable. Javascript code is executed one statement at a time in the order in which it is read. Many programmers put semicolons at the end of statements, although they are not required. Each statement is usually written on a separate line, with or without semicolons, to make scripts easier to read and edit.

A statement block is a group of statements enclosed in curly braces, "{}", which indicate that the enclosed individual statements are a group and are to be treated as one statement. A block can be used anywhere that a single statement can.

A while statement causes the statement after it to be executed in a loop. By enclosing multiple statements in curly braces, they are treated as one statement and are executed in the while loop. The following fragment illustrates:

```
while( ThereAreUncalledNamesOnTheList() == true)
{
   var name = GetNameFromTheList();
   CallthePerson(name);
   LeaveTheMessage();
}
```

All three lines after the while statement are treated as a unit. If the braces were omitted, the while loop would only apply to the first line. With the braces, the script goes through all lines until everyone on the list has been called. Without the braces, the script goes through all names on the list, but only the last one is called. Two very different procedures.

Statements within blocks are often indented for easier reading.

1.3 Identifiers

Identifiers are merely names for variables and functions. Programmers must know the names of built in variables and functions to use them in scripts and must know some rules about identifiers to define their own variables and functions. The following rules are simple and intuitive.

- Identifiers may use only ASCII letters, upper or lower case, digits, the underscore, "_", and the dollar sign, "$". That is, they may use only characters from the following sets of characters.
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  "abcdefghijklmnopqrstuvwxyz"
  "0123456789"
  "_$"
- Identifiers may **not** use the following characters.
  "+-<>&|=!*/%^~?:{};()[].'"`#,"
- Identifiers must begin with a letter, underscore, or dollar sign, but may have digits anywhere else.
- Identifiers may not have white space in them since white space separates identifiers for the interpreter.
- Identifiers may be as long a programmer needs.

The following identifiers, variables and functions, are valid:

```
Sid
Nancy7436
annualReport
sid_and_nancy_prepared_the_annualReport
$alice
CalculateTotal()
$SubtractLess()
_Divide$All()
```

The following identifiers, variables and functions, are not valid:

```
1sid
2nancy
this&that
Sid and Nancy
ratsAndCats?
(Minus)()
Add Both Figures()
```

1.3.1 Prohibited identifiers

The following words have special meaning for the interpreter and cannot be used as identifiers, neither as variable nor function names:

- abstract
- Break
- boolean
- byte
- case
- catch
- cfunction
- Char
- class
- const
- continue
- debugger
- default
- Delete
- do
- double
- else
- enum
- export
- Extends
- false
- final
- finally
- float
- for
- Function
- goto
- if
- implements
- import
- in
- Instanceof
- int
- interface
- long
- native
- new
- Null
- package
- private
- protected
- public
- return
- Short
- static
- super
- switch
- synchronized
- this
- Throw
- throws
- transient
- true
- try
- typeof
- While
- with
- var
- void
- volatile

1.3.2 Variables

A variable is an identifier to which data may be assigned. Variables are used to store and represent information in a script. Variables may change their values, but literals may not. For example, if programmers want to display a name literally, they must use something like the following fragment multiple times.

```
writeLog("Rumpelstiltskin Henry Constantinople")
```

But they could use a variable to make their task easier, as in the following.

```
var Name = "Rumpelstiltskin Henry Constantinople"
writeLog(Name)
```

Then they can use shorter lines of code for display and use the same lines of code repeatedly by simply changing the contents of the variable Name.

1.3.3 Variable scope

Variables in Javascript may be either global or local. Global variables may be accessed and modified from anywhere in a script. Local variables may only be accessed from the functions in which they are created. There are no absolute rules for preferring or using global or local variables. Each type has a value. In general, programmers prefer to use local variables when reasonable since they facilitate modular code that is easier to alter and develop over time. It is generally easier to understand how local variables are used in a single function than how global variables are used throughout an entire program. Further, local variables conserve system resources.

To make a local variable, declare it in a function using the var keyword:

```
var perfectNumber;
```

A value may be assigned to a variable when it is declared:

```
var perfectNumber = 28;
```

The default behavior of Javascript is that variables declared outside of any function or inside a function without the `var` keyword are global variables. However, this behavior can be changed by the `DefaultLocalVars` and `RequireVarKeyword` settings of the `#option` preprocessor directive. This directive is explained in the section on preprocessing. For now, consider the following code fragment.

```
var a = 1;

function main()
{
   b = 1;
   var d = 3;
   someFunction(d);
}

function someFunction(e)
{
   var c = 2
}
```

In this example, a and b are both global variables, since a is declared outside of a function and b is defined without the var keyword. The variables, d and c, are both local, since they are defined within functions with the var keyword. The variable c may not be used in the main() function, since it is `undefined` in the scope of that function. The variable d may be used in the `main()` function and is explicitly passed as an argument to someFunction() as the parameter e. The following lines show which variables are available to the two functions:

```
main():       a, b, d
someFunction():  a, b, c, e
```

It is possible, though not usually a good idea, to have local and global variables with the same name. In such a case, a global variable must be referenced as a property of the global object, and the variable name used by itself refers to the local variable. In the fragment above, the global variable a can be referenced anywhere in its script by using: `global.a`.

1.3.4 Function identifier

Functions are identified by names, as variables are. Functions perform script operations, and variables store data. Functions do the work of a script and will be discussed in more detail later. The reason they are mentioned here is simply to point out that they have identifiers, names, that follow the same rules for identifiers as variable names do.

1.3.5 Function scope

Functions are all global in scope, much like global variables. A function may not be declared within another function so that its scope is merely within a certain function or section of a script. All functions may be called from anywhere in a script. If it is helpful, think of functions as methods of the global object. The following two code fragments do exactly the same thing. The first calls a function, SumTwo(), as a function, and the second calls SumTwo() as a method of the global object.

```
// fragment one
function SumTwo(a, b)
{
   return a + b
}
writeLog(SumTwo(3, 4))

// fragment two
function SumTwo(a, b)
{
   return a + b
}
writeLog(global.SumTwo(3, 4))
```

1.4 Data types

Data types in Javascript can be classified into three groupings: primitive, composite, and special. In a script, data can be represented by literals or variables. The following lines illustrates variables and literals:

```
var TestVar = 14;
var aString = "test string";
```

The variable TestVar is assigned the literal 14, and the variable aString is assigned the literal "test string". After these assignments of literal values to variables, the variables can be used anywhere in a script where the literal values could to be used.

In the fragment above which defines and uses the function SumTwo(), the literals, 3 and 4, are passed as arguments to the function SumTwo() which has corresponding parameters, a and b. The parameters, a and b, are variables for the function the hold the literal values that were passed to it.
Data types need to be understood in terms of their literal representations in a script and of their characteristics as variables.

Data , in literal or variable form, is assigned to a variable with an assignment operator which is often merely an equal sign, "=" as the following lines illustrate.

```
var happyVariable = 7;
var joyfulVariable = "free chocolate";
var theWorldIsFlat = true;
var happyToo = happyVariable;
```

The first time a variable is used, its type is determined by the interpreter, and the type remains until a later assignment changes the type automatically. The example above creates three variables, each of a different type. The first is a number, the second is a string, and the third is a boolean variable. Variable types are described below. Since Javascript automatically converts variables from one type to another when needed, programmers normally do not have to worry about type conversions as they do in strongly typed languages, such as C.

1.4.1 Primitive data types

Variables that have primitive data types pass their data by value, by actually copying the data to the new location. The following fragment illustrates:

```
var a = "abc";
var b = ReturnValue(a);

function ReturnValue(c)
{
   return c;
}
```

After "abc" is assigned to variable a, two copies of the string "abc" exist, the original literal and the copy in the variable a. While the function ReturnValue is active, the parameter/variable c has a copy, and three copies of the string "abc" exist. If c were to be changed in such a function, variable a, which was passed as an argument to the function, would remain unchanged. After the function ReturnValue is finished, a copy of "abc" is in the variable b, but the copy in the variable c in the function is gone because the function is finished. During the execution of the fragment, as many as three copies of "abc" exist at one time.

The primitive data types are: Number, Boolean, and String.

Number type

Integer

Integers are whole numbers. Decimal integers, such as 1 or 10, are the most common numbers encountered in daily life. Javascript has three notations for integers: decimal, hexadecimal, and octal.

Decimal

Decimal notation is the way people write numbers in everyday life and uses base 10 digits from the set of 0-9. Examples are:

```
1, 10, 0, and 999
var a = 101;
```

Hexadecimal

Hexadecimal notation uses base 16 digits from the sets of `0-9`, `A-F`, and `a-f`. These digits are preceded by `0x`. Javascript is not case sensitive when it comes to hexadecimal numbers.

Examples are:

```
0x1, 0x01, 0x100, 0x1F, 0x1f, 0xABCD
var a = 0x1b2E;
```

Octal

Octal notation uses base 8 digits from the set of `0-7`. These digits are preceded by 0. Examples are:

```
00, 05, and 077
var a = 0143;
```

Floating point

Floating point numbers are number with fractional parts which are often indicated by a period, for example, 10.33. Floating point numbers are often referred to as floats.

Decimal floats

Decimal floats use the same digits as decimal integers but allow a period to indicate a fractional part. Examples are:

```
0.32, 1.44, and 99.44
var a = 100.55 + .45;
```

Scientific floats

Scientific floats are often used in the scientific community for very large or small numbers. They use the same digits as decimals plus exponential notation. Scientific notation is sometimes referred to as exponential notation. Examples are:

```
4.087e2, 4.087E2, 4.087e+2, and 4.087E-2
var a = 5.321e33 + 9.333e-2;
```

Boolean type

Booleans may have only one of two possible values: `false` or `true`. Since Javascript automatically converts values when appropriate, Booleans can be used as they are in languages such as C. Namely, `false` is zero, and `true` is non-zero. A script is more precise when it uses the actual Javascript values, `false` and `true`, but it will work using the concepts of zero and not zero. When a Boolean is used in a numeric context, it is converted to 0, if it is `false`, and 1, if it is `true`.

String type

A String is a series of characters linked together. A string is written using quotation marks, for example: "I am a string", 'so am I', `` `me too` ``, and "344". The string "344" is different from the number 344. The first is an array of characters, and the second is a value that may be used in numerical calculations.

Javascript automatically converts strings to numbers and numbers to string, depending on context. If a number is used in a string context, it is converted to a string. If a string is used in a number context, it is converted to a numeric value. Automatic type conversion is discussed more fully in a later section.

Strings, though classified as a primitive, are actually a hybrid type that shares characteristics of primitive and composite data types. Strings are discussed more fully a later section.

1.4.2 Composite data types

Whereas primitive types are passed by value, composite types are passed by reference. When a composite type is assigned to a variable or passed to a parameter, only a reference that points to its data is passed. The following fragment illustrates:

```
var AnObj = new Object;
AnObj.name = "Joe";
AnObj.old = ReturnName(AnObj)

function ReturnName(CurObj)
{
   return CurObj.name
}
```

After the object AnObj is created, the string "Joe" is assigned, by value since a property is a variable within an Object, to the property AnObj.name. Two copies of the string "Joe" exist. When AnObj is passed to the function ReturnName, it is passed by reference. CurObj does not receive a copy of the Object, but only a reference to the Object. With this reference, CurObj can access every property and method of the original. If CurObj.name were to be changed while the function was executing, then AnObj.name would be changed at the same time. When AnObj.old receives the return from the function, the return is assigned by value, and a copy of the string "Joe" transferred to the property. Thus, AnObj holds two copies of the string "Joe": one in the property .name and one in the property .old. Three total copies of "Joe" exist, counting the original string literal.

The composite data types are: Object and Array.

Object type

An object is a compound data type, consisting of one or more pieces of data of any type which are grouped together in an object. Data that are part of an object are called properties of the object. The Object data type is similar to the structure data type in C and in previous versions of Javascript. The object data type also allows functions, called methods, to be used as object properties. Indeed, in Javascript, functions are considered to be like variables. But for practical programming, think of objects as having methods, which are functions, and properties, which are variables and constants.

Objects and their characteristics are discussed more fully in a later section.

Array type

An array is a series of data stored in a variable that is accessed using index numbers that indicate particular data. The following fragments illustrate the storage of the data in separate variables or in one array variable:

```
var Test0 = "one";
var Test1 = "two";
var Test2 = "three";

var Test = new Array;
Test[0] = "one";
Test[1] = "two";
Test[2] = "three";
```

After either fragment is executed, the three strings are stored for later use. In the first fragment, three separate variables have the three separate strings. These variables must be used separately. In the second fragment, one variable holds all three strings. This array variable can be used as one unit, and the strings can be accessed individually. The similarities, in grouping, between Arrays and Objects is more than slight. In fact, Arrays and Objects are both objects in Javascript with different notations for accessing properties. For practical programming, Arrays may be considered as a data type of their own.

Arrays and their characteristics are discussed more fully in a later section.

1.4.3 Special values

undefined

If a variable is created or accessed with nothing assigned to it, it is of type `undefined`. An `undefined` variable merely occupies space until a value is assigned to it. When a variable is assigned a value, it is assigned a type according to the value assigned. Though variables may be of type `undefined`, there is no literal representation for `undefined`. Consider the following invalid fragment.

```
var test;
if (test == undefined)
   writeLog("test is undefined")
```

After var test is declared, it is `undefined` since no value has been assigned to it. But, the test, `test == undefined`, is invalid because there is no way to literally represent `undefined`.

null

The value `null` is a special data type that indicates that a variable is empty, a condition that is different from being `undefined`. A `null` variable holds no value, though it might have previously. The `null` type is represented literally by the identifier, `null`. Since Javascript automatically converts data types, `null` is both useful and versatile. The code fragment above will work if `undefined` is changed to `null`, as shown in the following:

```
var test;
if (test == null)
   writeLog("test is undefined")
```

Since `null` has a literal representation, assignments like the following are valid:

```
var test = null;
```

Any variable that has been assigned a value of `null` can be compared to the `null` literal.

The value `null` is an internal standard ECMAScript value. However, the value `NULL` is defined as 0 and is used in some scripts as it is found in C based documentation. Because of automatic conversion in Javascript, the two values often operate alike, but not always. They are two separate values.

NaN

The `NaN` type means "Not a Number". `NaN` is an acronym for the phrase. However, `NaN` does not have a literal representation. To test for `NaN`, the function, `global.isNaN()`, must be used, as illustrated in the following fragment:

```
var Test = "a string";
if (isNaN(parseInt(Test)))
   writeLog("Test is Not a Number");
```

When the `global.parseInt()` function tries to parse the string "a string" into an integer, it returns `NaN`, since "a string" does not represent a number like the string "22" does.

Number constants

Several numeric constants can be accessed as properties of the Number object, though they do not have a literal representation.

Constant	Value	Description
`Number.MAX_VALUE`	`1.7976931348623157e+308`	Largest number (positive)
`Number.MIN_VALUE`	`2.2250738585072014e-308`	Smallest number (negative)
`Number.NaN`	`NaN`	Not a Number
`Number.POSITIVE_INFINITY`	`Infinity`	Number above `MAX_VALUE`
`Number.NEGATIVE_INFINITY`	`-Infinity`	Number below `MIN_VALUE`

1.5 Automatic type conversion

When a variable is used in a context where it makes sense to convert it to a different type, Javascript automatically converts the variable to the appropriate type. Such conversions most commonly happen with numbers and strings. For example:

```
"dog" + "house" == "doghouse"   // two strings are joined
"dog" + 4 ==  "dog4"            // a number is converted
4 + "4" == "44"                 // to a string
4 + 4 == 8                      // two numbers are added
23 - "17" == 6                  // a string is converted
                                // to a number
```

Converting numbers to strings is fairly straightforward. However, when converting strings to numbers there are several limitations. While subtracting a string from a number or a number from a string converts the string to a number and subtracts the two, adding the two converts the number to a string and concatenates them. String always convert to a base 10 number and must not contain any characters other than digits. The string "110n" will not convert to a number, because the Javascript interpreter does not know what to make of the "n" character.

You can specify more stringent conversions by using the global methods, `global.parseInt()` and `global.parseFloat()` methods. Further, Javascript has many global functions to cast data as a specific type, functions that are not part of the ECMAScript standard. These functions are described in the section on global functions that are specific to Javascript.

1.6 Properties and methods of basic data types

The basic data types, such as Number and String, have properties and methods assigned to them that may be used with any variable of that type. For example, all String variables may use all String methods.

The properties and methods of the basic data types are retrieved in the same way as from objects. For the most part, they are used internally by the interpreter, but you may use them if choose. For example, if you have a numeric variable called number and you want to convert it to a string, you can use the `toString()` method as illustrated in the following fragment.

```
var n = 5
var s = n.toString()
```

After this fragment executes, the variable n contains the number 5 and the variable s contains the string "5".

The following two methods are common to all variables and data types.

1.6.1 toString()

This method returns the value of a variable expressed as a string. Every data type has `toString()` as a method. Thus, `toString()` is documented here and not in every conceivable place that it might be used.

1.6.2 valueOf()

This method returns the value of a variable. Every data type has `valueOf()` as a method. Thus, `valueOf()` is documented here and not in every conceivable place that it might be used.

1.7 Operators

1.7.1 Object operator

The object operator is a period, "`.`". This operator allows properties and methods of an object to be accessed and used. For example, `Math.abs()` is a method of the `Math object`. It may be accessed as follows:

```
var AbsNum = Math.abs(-3)
```

The variable AbsNum now equals 3. The variable AbsNum is an instance of the Number object, not an instance of the Math object. Why? It is assigned the number 3 which is the return of the `Math.abs()` method.

The `Math.abs()` method is a static method, that is, it is used directly with the Math object instead of an instance of the object. Many methods are instance methods, that is, they are used with instances of an object instead of the object itself.

The `String substring()` method is an instance method of the `String object`. An instance method is not used with an object itself but only with instances of an object. The `String substring()` method is never used with the String object as `String.substring()`. The following fragment declares and initializes a string variable, which is an instance of the String object, and then uses the `String substring()` method with this instance by using the object operator.

```
var s = "One Two Three";
var new = s.substring(4,7);
```

The variable s is an instance of the String object since it is initialized as a string. The variable new now equals "Two" and is also an instance of the String object since the `String substring()` method returns a string.

The main point here is that the period "`.`" is an object operator that may be used with both static and instance methods and properties. A method or property is simply attached to an appropriate identifier using the object operator, which then accesses the method or property.

1.7.2 Mathematical operators

Mathematical operators are used to make calculations using mathematical data. The following sections illustrate the mathematical operators in Javascript.

Basic arithmetic

The arithmetic operators in Javascript are pretty standard.

`=`	assignment	assigns a value to a variable
`+`	addition	adds two numbers
`-`	subtraction	subtracts a number from another
`*`	multiplication	multiplies two numbers
`/`	division	divides a number by another
`%`	modulo	returns a remainder after division

The following are examples using variables and arithmetic operators.

`var i;`	
`i = 2;`	i is now 2
`i = i + 3;`	i is now 5, (2+3)
`i = i - 3;`	i is now 2, (5-3)
`i = i * 5;`	i is now 10, (2*5)
`i = i / 3;`	i is now 3, (10/3) (remainder is ignored)
`i = 10;`	i is now 10
`i = i % 3;`	i is now 1, (10%3)

Expressions may be grouped to affect the sequence of processing. All multiplications and divisions are calculated for an expression before additions and subtractions unless parentheses are used to override the normal order. Expressions inside parentheses are processed first, before other calculations. In the following examples, the information inside square brackets, "[]," are summaries of calculations provided with these examples and not part of the calculations.

Notice that:

```
4 * 7 - 5 * 3;     [28 - 15 = 13]
```

has the same meaning, due to the order of precedence, as:

```
(4 * 7) - (5 * 3); [28 - 15 = 13]
```

but has a different meaning than:

```
4 * (7 - 5) * 3;   [4 * 2 * 3 = 24]
```

which is still different from:

```
4 * (7 - (5 * 3)); [4 * -8 = -32]
```

The use of parentheses is recommended in all cases where there may be confusion about how the expression is to be evaluated, even when they are not necessary.

Assignment arithmetic

Each of the above operators can be combined with the assignment operator, =, as a shortcut for performing operations. Such assignments use the value to the right of the assignment operator to perform an operation with the value to the left. The result of the operation is then assigned to the value on the left.

`=`	assignment	assigns a value to a variable
`+=`	assign addition	adds a value to a variable
`-=`	assign subtraction	subtracts a value from a variable
`*=`	assign multiplication	multiplies a variable by a value
`/=`	assign division	divides a variable by a value
`%=`	assign remainder	returns a remainder after division

The following lines are examples using assignment arithmetic.

`var i;`		
`i  = 2;`	i is now 2	
`i += 3;`	i is now 5, (2+3)	same as `i = i + 3`
`i -= 3;`	i is now 2, (5-3)	same as `i = i - 3`
`i *= 5;`	i is now 10, (2*5)	same as `i = i * 5`
`i /= 3;`	i is now 3, (10/3)	same as `i = i / 3`
`i  = 10;`	i is now 10	
`i %= 3;`	i is now 1, (10%3)	same as `i = i % 3`

Auto-increment (++) and auto-decrement (--)

To add or subtract one, 1, to or from a variable, use the auto-increment, ++, or auto-decrement, `--`, operator. These operators add or subtract 1 from the value to which they are applied. Thus, `i++` is a shortcut for `i += 1`, which is a shortcut for `i = i + 1`.

These operators can be used before, as a prefix operator, or after, as a postfix operator, their variables. If they are used before a variable, it is altered before it is used in a statement, and if used after, the variable is altered after it is used in the statement.

The following lines demonstrates prefix and postfix operations.

`i = 4;`	i is 4	
`j = ++i;`	j is 5, i is 5	(i was incremented before use)
`j = i++;`	j is 5, i is 6	(i was incremented after use)
`j = --i;`	j is 5, i is 5	(i was decremented before use)
`j = i--;`	j is 5, i is 4	(i was decremented after use)
`i++;`	i is 5	(i was incremented)

1.7.3 Bit operators

Javascript contains many operators for operating directly on the bits in a byte or an integer. Bit operations require a knowledge of bits, bytes, integers, binary numbers, and hexadecimal numbers. Not every programmer needs to or will choose to use bit operators.

<<	shift left	`i = i << 2;`
<<=	assignment shift left	`i <<= 2;`
>>	shift right	`i = i >> 2;`
>>=	assignment shift right	`i >>= 2;`
>>>	shift left with zeros	`i = i >>> 2`
>>>=	assignment shift left with zeros	`i >>>= 2`
&	bitwise and	`i = i & 1`
&=	assignment bitwise and	`i &= 1;`
|	bitwise or	`i = i | 1`
|=	assignment bitwise or	`i |= 1;`
^	bitwise xor, exclusive or	`i = i ^ 1`
^=	assignment bitwise xor, exclusive or	`i ^= 1`
~	Bitwise not, complement	`i = ~i;`

1.7.4 Logical operators and conditional expressions

Logical operators compare two values and evaluate whether the resulting expression is `false` or `true`. The value `false` is zero, and `true` is not `false`, that is, anything not zero. A variable or any other expression may be `false` or `true`, that is, zero or non-zero. An expression that does a comparison is called a conditional expression.

Many values are evaluated as `true`, in fact, everything except 0. It is often safer to make comparisons based on `false`, which is only one value, rather than to `true`, which can be many. Expressions can be combined with logic operators to make complex `true`/`false` decisions.

Logical operators are used to make decisions about which statements in a script will be executed, based on how a conditional expression evaluates. As an example, suppose that you are designing a simple guessing game. The computer thinks of a number between 1 and 100, and you guess what it is. The computer tells you if you are right or not and whether your guess is higher or lower than the target number. This procedure uses the if statement, which is introduced in the next section. Basically, if the conditional expression in the parenthesis following an if statement is `true`, the statement block following the if statement is executed. If `false`, the statement block is ignored, and the computer continues executing the script at the next statement after the ignored block. The script might have a structure similar to the one below in which GetTheGuess() is a function that gets your guess.

```
var guess = GetTheGuess(); //get the user input

if (guess > target_number)
{
   ...guess is too high...
}
if (guess < target_number)
{
   ...guess is too low...
}
if (guess == target_number)
{
   ...you guessed the number!...
}
```

This example is simple, but it illustrates how logical operators can be used to make decisions in Javascript.

The logical operators are:

!	not	reverses an expression. If (a+b) is `true`, then !(a+b) is `false`.
&&	and	`true` if, and only if, both expressions are `true`. Since both expressions must be `true` for the statement as a whole to be `true`, if the first expression is `false`, there is no need to evaluate the second expression, since the whole expression is `false`.
||	or	`true` if either expression is `true`. Since only one of the expressions in the or statement needs to be `true` for the expression to evaluate as `true`, if the first expression evaluates as `true`, the interpreter returns `true` and does not bother with evaluating the second.
==	equality	`true` if the values are equal, else `false`. Do not confuse the equality operator, ==, with the assignment operator, =.
!=	inequality	`true` if the values are not equal, else `false`.
===	identity	`true` if the values are identical or strictly equal, else `false`. No type conversions are performed as with the equality operator.
!==	non-identity	`true` if the values are not identical or not strictly equal, else `false`. No type conversions are performed as with the inequality operator.
<	less than	a < b is `true` if a is less than b.
>	greater than	a > b is `true` if a is greater than b.
<=	less than or equal to	a <= b is `true` if a is less than or equal to b.
>=	greater than or equal to	a >= b is `true` if a is greater than b.

Remember, the assignment operator, =, is different than the equality operator, ==. If you use one equal sign when you intend two, your script will not function the way you want it to. This is a common pitfall, even among experienced programmers. The two meanings of equal signs must be kept separate, since there are times when you have to use them both in the same statement, and there is no way the computer can differentiate them by context.

1.7.5 delete operator

The `delete` operator deletes properties from objects and elements from arrays. Deleted properties and arrays are actually undefined. Any memory cleanup is handled by normal garbage collection.

The following fragment defines an array with three elements: 0, 1, and 2, and an object with three properties: four, five, and six. It then deletes the middle, that is, the second, element of the array and property of the object.

```
var a = {"one", "two", "three"};
var o = {four:444, five:555, six:666};

delete(a[1]);
delete(o.five);
```

1.7.6 instanceof operator

The `instanceof` operator, which also may used as `instanceof()`, determines if a variable is an instance of a particular object. Since the variable `s` is created as an instance of the String object in the following code fragment, the second line displays `true`.

```
var s = new String("abcde");
writeLog(s instanceof String);   // Displays true
```

The second line could also be written as:

```
writeLog(s instanceof(String));
```

1.7.7 typeof operator

The `typeof` operator, which also may be used as `typeof()`, provides a way to determine and to test the data type of a variable and may use either of the following notations, with or without parentheses.

```
var result = typeof variable
var result = typeof(variable)
```

After either line, the variable result is set to a string that is represents the variable's type: "undefined", "boolean", "string", "object", "number", or "function".

1.8 Flow decisions statements

This section describes statements that control the flow of a program. Use these statements to make decisions and to repeatedly execute statement blocks.

1.8.1 if

The `if` statement is the most commonly used mechanism for making decisions in a program. It allows you to test a condition and act on it. If an `if` statement finds the condition you test to be `true`, the statement or statement block following it are executed. The following fragment is an example of an if statement.

```
if ( goo < 10 )
{
   writeLog("goo is smaller than 10");
}
```

1.8.2 else

The `else` statement is an extension of the if statement. It allows you to tell your program to do something else if the condition in the if statement was found to be `false`. In Javascript code, it looks like the following.

```
if ( goo < 10 )
{
   writeLog("goo is smaller than 10");
}
else
{
   writeLog("goo is not smaller than 10");
}
```

To make more complex decisions, else can be combined with if to match one out of a number of possible conditions. The following fragment illustrates using `else` with `if`.

```
if ( goo < 10 )
{
   writeLog("goo is less than 10");
   if ( goo < 0 )
   {
      writeLog("goo is negative; so it's less than 10");
   }
}
else if ( goo > 10 )
{
   writeLog("goo is greater than 10");
}
else
{
   writeLog("goo is 10");
}
```

1.8.3 while

The `while` statement is used to execute a particular section of code, over and over again, until an expression evaluates as `false`.

```
while (expression)
{
   DoSomething();
}
```

When the interpreter comes across a `while` statement, it first tests to see whether the expression is `true` or not. If the expression is `true`, the interpreter carries out the statement or statement block following it. Then the interpreter tests the expression again. A while loop repeats until the test expression evaluates to `false`, whereupon the program continues after the code associated with the while statement.

The following fragment illustrates a while statement with a two lines of code in a statement block.

```
while( ThereAreUncalledNamesOnTheList() != false)
{
   var name=GetNameFromTheList();
   sendMail("mymessage.txt", "Alert", "address@domain.com");
}
```

1.8.4 do {...} while

The `do` statement is different from the `while` statement in that the code block is executed at least once, before the test condition is checked.

```
var value = 0;
do
{
   value++;
   ProcessData(value);
} while( value < 100 );
```

The code used to demonstrate the `while` statement could also be written as the following fragment.

```
do
{
   var name = GetNameFromTheList();
   sendMail("mymessage.txt", "Alert", "address@domain.com");
} while (name != TheLastNameOnTheList());
```

Of course, if there are no names on the list, the script will run into problems!

1.8.5 for

The `for` statement is a special looping statement. It allows for more precise control of the number of times a section of code is executed. The `for` statement has the following form.

```
for ( initialization; conditional; loop_expression )
{
   statement
}
```

The initialization is performed first, and then the expression is evaluated. If the result is `true` or if there is no conditional expression, the statement is executed. Then the loop_expression is executed, and the expression is re-evaluated, beginning the loop again. If the expression evaluates as `false`, then the statement is not executed, and the program continues with the next line of code after the statement. For example, the following code displays the numbers from 1 to 10.

```
for(var x=1; x<11; x++)
{
   writeLog(x);
}
```

None of the statements that appear in the parentheses following the for statement are mandatory, so the above code demonstrating the while statement would be rewritten this way if you preferred to use a `for` statement:

```
for( ; ThereAreUncalledNamesOnTheList() ; )
{
   var name=GetNameFromTheList();
   sendMail("mymessage.txt", "Alert", "address@domain.com");
}
```

Since we are not keeping track of the number of iterations in the loop, there is no need to have an initialization or loop_expression statement. You can use an empty `for` statement to create an endless loop:

```
for(;;)
{
   //the code in this  block will repeat forever,
   //unless the program breaks out of the for loop somehow.
}
```

1.8.6 break

`Break` and `continue` are used to control the behavior of the looping statements: `for`, `switch`, `while`, and `do {...} while`. The `break` statement terminates the innermost loop of `for`, `while`, or `do` statements. The program resumes execution on the next line following the loop. The following code fragment does nothing but illustrate the `break` statement.

```
for(;;)
{
   break;
}
```

The `break` statement is also used at the close of a `case` statement, as shown below. See `switch, case, and default`.

1.8.7 continue

The `continue` statement ends the current iteration of a loop and begins the next. Any conditional expressions are reevaluated before the loop reiterates. The `continue` statement works with the same loops as the `break` statement.

1.8.8 switch, case, and default

The `switch` statement makes a decision based on the value of a variable or statement.

The `switch` statement follows the following format:

```
switch( switch_variable )
{
case value1:
   statement1
   break;
case value2:
   statement2
   break;

...

default:
   default_statement
}
```

The variable switch_variable is evaluated, and then it is compared to all of the values in the case statements (value1, value2, `. . .` , default) until a match is found. The statement or statements following the matched case are executed until the end of the `switch` block is reached or until a `break` statement exits the `switch` block. If no match is found, the `default` statement is executed, if there is one.

A common mistake is to omit a `break` statement to end each case. In the preceding example, if the `break` statement after the `writeLog("B")` statement were omitted, the computer would print both "B" and "C", since the interpreter executes commands until a `break` statement is encountered.

1.8.9 goto and labels

You may jump to any location within a function block by using the `goto` statement. The syntax is:

```
goto LABEL;
```

where `label` is an identifier followed by a colon (:). The following code fragment continuously prompts for a number until a number less than 2 is entered.

```
beginning:
writeLog("Enter a number less than 2:")
var x = getche();     //get a value for x
if (a >= 2)
   goto beginning;
```

```
 writeLog(a);
```

As a rule, `goto` statements should be used sparingly, since they make it difficult to track program flow.

1.8.10 Conditional operator

The conditional operator, "`? :`", provides a shorthand method for writing if statements. It is harder to read than conventional if statements, and so is generally used when the expressions in the if statements are brief.

The syntax is:

```
 test_expression ? expression_if_true : expression_if_false
```

First, test_expression is evaluated. If test_expression is non-zero, `true`, then expression_if_true is evaluated, and the value of the entire expression replaced by the value of expression_if_true. If test_expression is `false`, then expression_if_false is evaluated, and the value of the entire expression is that of expression_if_false.

The following fragment illustrates the use of the conditional operator.

```
 foo = ( 5 < 6 ) ? 100 : 200; // foo is set to 100
 writeLog("Name is " + ((null==name) ? "unknown" : name));
```

1.8.11 Exception handling

Exception handling statements consist of: `throw`, `try`, `catch`, and `finally`. The concept of exception handling includes dealing with unusual results in a function and with errors and recovery from them. Exception handling that uses the `try` related statements is most useful with complex error handling and recovery. Testing for simple errors and unwanted results is usually handled most easily with familiar `if` or `switch` statements. In this section, the discussion and examples deal with simple situations, since explanation and illustration are the goals. The exception handling statements might seem clumsy or bulky here, but do not lose sight of the fact that they are very powerful and elegant in real world programming where error recovery can be very complex and require much code when using traditional statements.

Another advantage of using `try` related exception handling is that much of the error trapping code may be in a function rather than in the all the places that call a function.

Before getting to specifics, here is some generalized phrasing that might help working with exception handling statements. A function has code in it to detect unusual results and to **throw an exception**. The function is called from inside a `try` statement block which **tries to run the function** successfully. If there is a problem in the function, the exception thrown is **caught and handled** in a `catch` statement block. If all exceptions have been handled when execution reaches the finally statement block, the **final code is executed**.

Remember these execution guides:

- When a `throw` statement executes, the rest of the code in a function is ignored, and the function does not return a value.
- A program continues in the next `catch` statement block after the `try` statement block in

which an exception occurred., and any value thrown is caught in a parameter in the catch statement.

- A program executes a `finally` statement block if all exceptions, that have been thrown, have been caught and handled.

The following simple script illustrates all exception handling statements. The `main() function` has `try`, `catch`, and `finally` statement blocks. The `try` block calls `SquareEven()`, which throws an exception if an odd number is passed to it. If an even number is passed to the function, then the number is squared and returned. If an odd number is passed, it is fixed, and an exception is thrown. When the `throw` statement executes, it passes an object, as an argument, with information for the `catch` statement to use.

For example, the script below, as shown, displays:

```
16
We caught odd and squared even.
```

If you change `rtn = SquareEven(4)` to `rtn = SquareEven(3)`, the display is:

```
Fixed odd number to next higher even. 16
We caught odd and squared even.
```

The example script below does not actually handle errors. Its purpose is to illustrate how exception handling statements work. For purposes of this illustration, assume that an odd number being passed to `SquareEven()` is an error or extraordinary event.

```
function main()
{
   var rtn;

   try {
      rtn = SquareEven(4);
      // No display here if number is odd
      writeLog(rtn);
   }
   catch (err) {
      // Catch the exception info that was thrown by the function.
      // In this case, the info was returned in an object.
      writeLog(err.msg + err.rtn);
   }
   finally {
      // Finally, display this line after normal processing
      // or exceptions have been caught.
      writeLog("We caught odd and squared even.");
   }
}

// Check for odd integers.
// If odd, make even, simplistic by adding 1
// Square even number
function SquareEven(num)
{
   // Catch an odd number and fix it.
   // "throw an exception" to be caught by caller
   if ((num % 2) != 0) {
      num += 1;
      throw {msg:"Fixed odd number to next higher even. ", rtn:num * num};

      // We throw an object here. We could have thrown a primitive, such as:
      //  throw("Caught and odd");
      // We would have to alter the catch statement to expect whatever data
      // type is used.
   }

   // Normal return for an even number.
   return num * num;
}
```

1.9 Functions

A function is an independent section of code that receives information from a program and performs some action with it. Once a function has been written, you do not have to think again about how to perform the operations in it. Just call the function, and let it handle the work for you. You only need to know what information the function needs to receive, that is, the parameters, and whether it returns a value to the statement that called it.

`writeLog()` is an example of a function which provides an easy way to write formatted text to the Diagnostic Log. It receives a string from the function that called it and writes the string to the Log. writeLog is a void function, meaning it has no return value.

In Javascript, functions are considered a data type, evaluating to whatever the function's return value is. You can use a function anywhere you can use a variable. Any valid variable name may be used as a function name. Like comments, using descriptive function names helps you keep track of what is going on with your script.

Two things set functions apart from the other variable types: instead of being declared with the "var" keyword, functions are declared with the "function" keyword, and functions have the function operator, "()", following their names. Data to be passed to a function is included within these parentheses.

Several sets of built-in functions are included as part of the Javascript interpreter. These functions are described in this manual. They are internal to the interpreter and may be used at any time.

Javascript allows you to have two functions with the same name. The interpreter uses the function nearest the end of the script, that is, the last function to load is the one that to be executed when the function name is called. By taking advantage of this behavior, you can write functions that supersede the ones included in the interpreter or .jsh files.

1.9.1 Function return statement

The `return` statement passes a value back to the function that called it. Any code in a function following the execution of a `return` statement is not executed.

```
 function DoubleAndDivideBy5(a)
 {
    return (a*2)/5
 }
```

Here is an example of a script using the above function.

```
function main()
{
   var a = DoubleAndDivideBy5(10);
   var b = DoubleAndDivideBy5(20);
   writeLog(a + b);
}
```

This script displays 12.

1.9.2 Passing information to functions

Javascript uses different methods to pass variables to functions, depending on the type of variable being passed. Such distinctions ensure that information gets to functions in the most complete and logical ways. To be technically correct, the data that is passed to a function are called arguments, and the variables in a function definition that receive the data are called parameters.

Primitive types, namely, strings, numbers, and booleans, are passed by value. The value of theses variables are passed to a function. If a function changes one of these variables, the changes will not be visible outside of the function where the change took place.

Composite types, objects and arrays, are passed by reference. Instead of passing the value of the object, that is, the values of each property, a reference to the object is passed. The reference indicates where in a computer's memory that values of an object's properties are stored. If you make a change in a property of an object passed by reference, that change will be reflected throughout in the calling routine.

In Javascript it is possible to pass primitive types by reference instead of by value, which is the default. When a function is defined, an ampersand, &, may be put in front of one or more of its parameters. Thus, when the function is called, an argument, corresponding to a parameter with an ampersand, is passed by reference instead of by value. The following fragment illustrates.

```
var num1 = 4;
var num2 = 4;
var num3;

SetNumbers(num1, num2, num3, 6)

function SetNumbers(&n1, n2, &n3, &n4)
{
  n1 = n2 = n3 = n4 = 5;
}
```

After executing this code, the values of variables is:

```
num1 == 5
num2 == 4
num3 == 5
```

The variable num1 was passed by reference to parameter n1. When n1 was set to 5, num1 was actually set to 5 since n1 merely pointed to num1. The variable num2 was passed by value to parameter n2. When n2, which received an actual value of 4, was set to 5, num2 remained unchanged. The variable num3 was `undefined` when passed by reference to parameter n3. When n3, which pointed to num3, was set to 5, num3 was actually set to 5 and defined as an integer type.

The literal value 6 was passed to parameter n4, but not by reference since 6 is not a variable that can be changed. Though n4 has an ampersand, the literal value 6 was passed by value to n4 which, in this example, becomes merely a local variable for the function SetNumbers().

1.9.3 Simulated named parameters

The properties of object data types may be used like named parameters. The following line simulates named parameters in a call to a function (note the use of curly braces `{}`):

```
var area = RectangleArea({length:4, width:2});
```

The following line uses traditional ordered parameters:

```
var area = RectangleArea(4, 2);
```

The following function definition receives the named and ordered parameters in the lines above. The definition allows for named or ordered parameters to be used.

```
function RectangleArea(length, width)
{
   if (typeof(length) == "object") {
      width = length.width;
      length = length.length;
   }
   return length * width;
}
```

The function above could be rewritten as:

```
function RectangleArea(length, width)
{
   if (typeof(arguments[0]) == "object") {
      width = arguments[0].width;
      length = arguments[0].length;
   }
   return length * width;
}
```

Either function definition works the same. The choice of one over the other is a matter of personal preference.

Though Javascript allows many variations in how objects may be used, this straightforward example illustrates the essence of simulating named parameters in Javascript.

1.9.4 Function property arguments[]

The `arguments[]` property is an array of all of the arguments passed to a function. The first variable passed to a function is referred to as `arguments[0]`, the second as `arguments[1]`, and so forth.

The most useful aspect of this property is that it allows you to have functions with an indefinite number of parameters. Here is an example of a function that takes a variable number of arguments and returns the sum of them all.function SumAll()

```
    var total = 0;
    for (var ssk = 0; ssk < SumAll.arguments.length; ssk++) {
       total += SumAll.arguments[ssk];
    }
    return total;
```

1.9.5 Function recursion

A recursive function is a function that calls itself or that calls another function that calls the first function. Recursion is permitted in Javascript. Each call to a function is independent of any other call to that function. (See the section on `variable scope`.) Be aware that recursion has limits. If a function calls itself too many times, a script will run out of memory and abort.

Do not worry if recursion is confusing, since you rarely have to use it. Just remember that a function can call itself if it needs to. For example, the following function, factor(), factors a number. Factoring is an ideal candidate for recursion because it is a repetitive process where the result of one factor is then itself factored according to the same rules.

```
 // recursive function to print all factors of i,
 // and return the number of factors in i
 function factor(i)
 {
    if ( 2 <= i ) {
       for ( var test = 2; test <= i; test++ ) {
          if ( 0 == (i % test) ) {
             // found a factor, so print this factor then call
             // factor() recursively to find the next factor
             return( 1 + factor(i/test) );
          }
       }
    }
    // if this point was reached, then factor not found
    return( 0 );
 }
```

1.9.6 Error checking for functions

Some functions return a special value if they fail to do what they are supposed to do. For example, the `Clib.fopen()` method opens or creates a file for a script to read from or write to. But suppose that the computer is unable to open a file. In such a case, the `Clib.fopen()` method returns `null`.

If you try to read from or write to a file that was not properly opened, you get all kinds of errors. To prevent these errors, make sure that `Clib.fopen()` does not return `null` when it tries to open a file.

Instead of just calling `Clib.fopen()` as follows:

```
 var fp = Clib.fopen("myfile.txt", "r");
```

check to make sure that `null` is not returned:

```
if (null == (var fp = Clib.fopen("myfile.txt", "r"))) {
   writeLog("Clib.fopen returned null");
}
```

1.9.7 main() function

If a script has a function called `main()`, it is the first function executed. (For more information on what takes place when a script is run, see the section on `running a script`.) Other than the fact that `main()` is the first function executed, it is like other functions.

1.10 Objects

Variables and functions may be grouped together in one variable and referenced as a group. A compound variable of this sort is called an object in which each individual item of the object is called a property. In general, it is adequate to think of object properties, which are variables or constants, and of object methods, which are functions.

To refer to a property of an object, use both the name of the object and of the property, separated by the object operator ".", a period. Any valid variable name may be used as a property name. For example, the code fragment below assigns values to the width and height properties of a rectangle object and calculates the area of a rectangle and displays the result:

```
var Rectangle;

Rectangle.height = 4;
Rectangle.width = 6;

writeLog(Rectangle.height * Rectangle.width);
```

The main advantage of objects occurs with data that naturally occurs in groups. An object forms a template that can be used to work with data groups in a consistent way. Instead of having a single object called Rectangle, you can have a number of Rectangle objects, each with their own values for width and height.

1.10.1 Predefining objects with constructor functions

A constructor function creates an object template. For example, a constructor function to create Rectangle objects might be defined like the following.

```
function Rectangle(width, height)
{
   this.width = width;
   this.height = height;
}
```

The keyword `this` is used to refer to the parameters passed to the constructor function and can be conceptually thought of as "this object." To create a Rectangle object, call the constructor function with the "new" operator:

```
var joe = new Rectangle(3,4)
var sally = new Rectangle(5,3);
```

This code fragment creates two rectangle objects: one named joe, with a width of 3 and a height of 4, and another named sally, with a width of 5 and a height of 3.

Constructor functions create objects belonging to the same class. Every object created by a constructor function is called an instance of that class. The examples above creates a Rectangle class and two instances of it. All of the instances of a class share the same properties, although a particular instance of the class may have additional properties unique to it. For example, if we add the following line:

```
joe.motto = "ad astra per aspera";
```

we add a motto property to the Rectangle joe. But the rectangle sally has no motto property.

1.10.2 Initializers for objects and arrays

Variables may be initialized as objects and arrays using lists inside of "`{}`" and "`[]`". By using these initializers, instances of Objects and Arrays may be created without using the `new` constructor. Objects may be initialized using a syntax similar to the following:

```
var o = {a:1, b:2, c:3};
```

This line creates a new object with the properties a, b, and c set to the values shown. The properties may be used with normal object syntax, for example, `o.a == 1`.

Arrays may initialized using a syntax similar to the following:

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

This line creates a new array with three elements set to 1, 2, and 3. The elements may be used with normal array syntax, for example, `a[0] == 1`.

The distinction between Object and Array initializer might be a bit confusing when using a line with syntax similar to the following:

```
var a = {1, 2, 3};
```

This line also creates a new array with three elements set to 1, 2, and 3. The line differs from the first line, Object initializer, in that there are no property identifiers and differs from the second line, Array initializer, in that it uses "`{}`" instead of "`[]`". In fact, the second and third lines produce the same results. The elements may be used with normal array syntax, for example, `a[0] == 1`.

The following code fragment shows the differences.

```
var o= {a:1, b:2, c:3};
writeLog(typeof o +" | "+ o._class +" | "+ o);

var a = [1, 2, 3];
writeLog(typeof a +" | "+ a._class +" | "+ a);

var a= {1, 2, 3};
writeLog(typeof a +" | "+ a._class +" | "+ a);
```

The display from this code is:

```
object | Object | [object Object]
object | Array | 1,2,3
object | Array | 1,2,3
```

As shown in the first display line, the variable `o` is created and initialized as an Object. The second and third lines both initialize the variable `a` as an Array. Notice that in all cases the `typeof` the variable is object, but the class, which corresponds to the particular object and which is reflected in the `_class` property, shows which specific object is created and initialized.

1.10.3 Methods - assigning functions to objects

Objects may contain functions as well as variables. A function assigned to an object is called a method of that object.

Like a constructor function, a method refers to its variables with the `this` operator. The following fragment is an example of a method that computes the area of a rectangle.

```
function rectangle_area()
{
   return this.width * this.height;
}
```

Because there are no parameters passed to it, this function is meaningless unless it is called from an object. It needs to have an object to provide values for `this.width` and `this.height`.

A method is assigned to an object as the following lines illustrates.

```
joe.area = rectangle_area;
```

The function will now use the values for height and width that were defined when we created the rectangle object joe.

Methods may also be assigned in a constructor function, again using the this keyword. For example, the following code:

```
function rectangle_area()
{
   return this.width * this.height;
}

function Rectangle(width, height)
{
   this.width = width;
   this.height = height;
   this.area = rectangle_area;
}
```

creates an object class Rectangle with the rectangle_area method included as one of its properties.

The method is available to any instance of the class:

```
var joe = Rectangle(3,4);
var sally = Rectangle(5,3);

var area1 = joe.area;
var area2 = sally.area;
```

This code sets the value of area1 to 12, and the values of area2 to 15.

1.10.4 Object prototypes

An object prototype lets you specify a set of default values for an object. When an object property that has not been assigned a value is accessed, the prototype is consulted. If such a property exists in the prototype, its value is used for the object property.

Object prototypes are useful for two reasons: they ensure that all instances of an object use the same default values, and they conserve the amount of memory needed to run a script. When the two Rectangles, joe and sally, were created in the previous section, they were each assigned an area method. Memory was allocated for this function twice, even though the method is exactly the same in each instance. This redundant memory waste can be avoided by putting the shared function or property in an object's prototype. Then all instances of the object will use the same function instead of each using its own copy of it.

The following fragment shows how to create a Rectangle object with an area method in a prototype.

```
function rectangle_area()
{
   return this.width * this.height;
}

function Rectangle(width, height)
{
   this.width = width;
   this.height = height;
}

Rectangle.prototype.area = rectangle_area;
```

The rectangle_area method can now be accessed as a method of any Rectangle object as shown in the following.

```
var area1 = joe.area();
var area2 = sally.area();
```

You can add methods and data to an object prototype at any time. The object class must be defined, but you do not have to create an instance of the object before assigning it prototype values. If you assign a method or data to an object prototype, all instances of that object are updated to include the prototype.

If you try to write to a property that was assigned through a prototype, a new variable will be created for the newly assigned value. This value will be used for the value of this instance of the object's property. All other instances of the object will still refer to the prototype for their values. If, for the sake of this example, we assume that joe is a special Rectangle, whose area is equal to three times its width plus half its height, we can modify joe as follows.

```
joe.area = function joe_area()
{
   (this.width * 3) + (this.height/2);
}
```

This fragment creates a value, which in this case is a function, for joe.area that supercedes the prototype value. The property `sally.area` is still the default value defined by the prototype. The instance joe uses the new definition for its area method.

1.10.5 for/in

The `for/in` statement is a way to loop through all of the properties of an object, even if the names of the properties are unknown. The statement has the following form.

```
for (var property in object)
{
   DoSomething(object[property]);
}
```

where object is the name of an object previously defined in a script. When using the `for . . . in` statement in this way, the statement block will execute once for every property of the object. For each iteration of the loop, the variable property contains the name of one of the properties of object and may be accessed with "object[property]". Note that properties that have been marked with the `DontEnum` attribute are not accessible to a `for . . . in` statement.

1.10.6 with

The `with` statement is used to save time when working with objects. It lets you assign a default object to a statement block, so you need not put the object name in front of its properties and methods. The object is automatically supplied by the interpreter. The following fragment illustrates using the `Clib object`.

```
with (Math)
{
   xxx = random() * 100);
   yyy = floor(xxx);
   writeLog(yyy);
}
```

The Math methods, `Math.random()` and `Math.floor()` in the sample above are called as if they had been written with `Math` prefixed. All code in the block following a with statement seems to be treated as if the methods associated with the object named by the with statement were global functions. Global functions are still treated normally, that is, you do not need to prefix "global." to them unless you are distinguishing between two like-named functions common to both objects.

If you were to jump, from within a with statement, to another part of a script, the with statement would no longer apply. In other words, the with statement only applies to the code within its own block, regardless of how the interpreter accesses or leaves the block.

You may not use `goto and labels` to jump into or out of the middle of a `with` statement block.

1.11 Predefined constants and values

The following values are predefined values in Javascript and are available during the execution of a script. These values may be used in any normal statements in scripts.

`false`	Boolean `false`
`null`	A `null` value with multiple uses
`true`	Boolean `true`
`0`	0, `false`, indicating that the processor stores the low byte of a value in low memory, such as Intel.
`1`	1, `true`, indicating that the processor stores the low byte of a value in high memory, such as Motorola.
`EOF`	In file operations, indicates that the end of a file has been reached
`NaN`	Not a Number
`Number.MAX_VALUE`	Largest positive number that can be represented in Javascript
`Number.MIN_VALUE`	Small negative number that can be represented in Javascript
`Number.NaN`	Not a Number
`Number.POSITIVE_INFINITY`	Any number greater than `MAX_VALUE`
`Number.NEGATIVE_INFINITY`	Any number smaller than `MIN_VALUE`
`SEEK_CUR`	Position in a file relative to the current position in a file
`SEEK_END`	Position in a file relative to the end of the file
`SEEK_SET`	Position in a file relative to the beginning of the file
`VERSION_MAJOR`	The major version number of Javascript, for example, 4 in 4.10b
`VERSION_MINOR`	The minor version number of Javascript, for example, 10 in 4.10b
`VERSION_STRING`	The revision letter of Javascript, for example, b in 4.10b

1.12 Extending the FCscript

One FCscript can be extended by including other FCscripts:

```
#include "/home/public/script/morefunctions.js"
```

Note: Files with extension .js are not listed in the 'installed scripts' overview in the Admin pages of FieldCommander.

2. Javascript versus C language

This section is primarily for those who already know how to program in C, though novice programmers can learn more about the Clib objects and C concepts by reading it. The emphasis is on those elements of Javascript that differ from standard C. Most of the pertinent differences involve the `Clib object`. Users who are not familiar with C should first read the section on Javascript.

The most basic idea underlying this section is that the C portion of Javascript is C without type declarations and pointers.

2.1 Automatic type declaration

There are no type declarations nor type castings as found in C. Types are determined from context. In the statement, `var i = 6`, the variable i is a number type. For example, the following C code:

```
int max(int a, int b)
{
   int result;
   result = (a < b) ? b : a;
   return result;
}
```

could be converted to the following Javascript code:

```
Clib.max(a, b)
{
   var result = (a < b) ? b : a;
   return result;
}
```

The code could be made even more like C by using a with statement as in the following fragment.

```
with (Clib)
{
   max(a, b)
   {
      var result = (a < b) ? b : a;
      return result;
   }
}
```

A with statement can be used with large blocks of code which would allow Clib methods to be called like C functions. C programmers will appreciate this ability. Other users who decide to use the extra power of C functions will come to appreciate this ability.

2.2 Array representation

This section on the representation of arrays in memory only deals with automatic arrays which are part of the C portion of Javascript. Javascript uses constructor functions that create instances of Javascript arrays which are actually objects more than arrays. Everything said in this section is about automatic arrays compared to C arrays. The methods and functions used to work with Javascript constructed arrays and Javascript automatic arrays are different. The following fragment creates a Javascript array.

```
var aj = new Array();
```

The following line creates an automatic array in Javascript.

```
var ac[3][3];
```

The two arrays are different entities that require different methods and functions. For example, the property `aj.length` provides the length of the aj array, but the function `getArrayLength(ac)`provides the length of the ac automatic array. When the term array is used in the rest of this section, the reference is to an automatic array. Javascript arrays are covered in the section on Javascript.

Arrays are used in Javascript much like they are in C, except that they are stored differently. A single dimension array, for example, an array of numbers, is stored in consecutive bytes in memory, just as in C, but arrays of arrays are not in consecutive memory locations. The following C declaration:

```
char c[3][3];  // this is the C version
```

indicates that there are nine consecutive bytes in memory. In Javascript a similar statement such as the following:

```
var c[2][2] = 'a';  // this is the Javascript version
```

indicates that there are at least three arrays of characters, and the third array of arrays has at least three characters in it. Though the characters in c[0] and the characters in c[1] are in consecutive bytes, the two arrays c[0] and c[1] are not necessarily adjacent in memory.

2.3 Automatic array allocation

Arrays are dynamic, and any index, positive or negative, into an array is always valid. If an element of an array is referenced, then Javascript ensures that such an element exists. For example, if a statement in a script is:

```
var foo[4] = 7;
```

then Javascript makes an array of 5 integers referenced by the variable foo. If a later statement refers to foo[6] then Javascript expands foo, if necessary, to ensure that the element foo[6] exists. The same is `true` for negative indices. When foo[-10] is referenced, foo is grown in the negative direction if necessary, but foo[4] still refers to the initial 7. Arrays can be of any order of dimensions, thus foo[6][7][34][-1][4] is a valid variable or array.

2.4 Literal strings

A literal string in Javascript is any array of characters, that is, a string, appearing in source code within double, single, or back quotes. Back quotes are sometimes referred to as back-ticks.

The following lines show examples of literal strings in Javascript:

```
"dog"                  // literal string (double quote)
```

```
'dog'                   // literal string (single quotes)
`dog`                   // literal string (back-ticks)
{'d','o','g','\0'}      // not a literal string, but array initialization
```

Literal strings have special treatment for certain Javascript operations for the following reasons.

- To protect literal string data from being overwritten accidentally
- To reduce confusion for novice programmers who do not think of strings as arrays of bytes
- To simplify writing code for common operations.

2.4.1 Literal strings and assignments

When a literal string is assigned to a variable, a copy is made of the string, and the variable is assigned the copy of the literal string. For example, the following code:

```
for (var i = 0; i < 3; i++) {
   var str = "dog";
   str = str + "house";
   writeLog(str);
}
```

results in the following output:

```
doghouse
doghouse
doghouse
```

A strict C interpretation of this code would not only overwrite memory, but would also generate the following output:

```
doghouse
doghousehouse
doghousehousehouse
```

2.4.2 Literal strings and comparisons

The following examples demonstrate how literal strings compare.

```
if (animal == "dog")
if (animal <  "dog")
if ("dog" <= animal)
```

In Javascript, the following fragment:

```
var animal = "dog";
if (animal == "dog")
writeLog("hush puppy");
```

displays:

```
"hush puppy"
```

2.4.3 Literal strings and parameters

When a literal string is a parameter to a function, it is passed as a copy, that is, by value. For example, the following code:

```
for (var i = 0; i < 3; i++) {
   var str = "dog" + "house";
   writeLog(str)
}
```

results in the following output:

```
doghouse
doghouse
doghouse
```

2.4.4 Literal strings and returns

When a literal string is returned from a function by a return statement, it is returned as a copy of the string. The following code:

```
for (var i = 0; i < 3; i++) {
   var str = dog() + "house";
   writeLog(str)
}

function dog() {
   return "dog";
}
```

results in the following output:

```
doghouse
doghouse
doghouse
```

2.5 Structures

Structures are created dynamically, and their elements are not necessarily contiguous in memory. When Javascript encounters a statement such as:

```
foo.animal = "dog"
```

it creates a structure element of foo that is referenced by "animal" and that is an array of characters. The "animal" variable becomes an element of the "foo" variable. Though foo, in this example, may be thought of and used as a structure and animal as an element, in actuality, foo is a Javascript object and animal is a property. The resulting code looks like regular C code, except that there is no separate structure definition anywhere. The following C code:

```
struct Point
{
   int Row;
   int Column;
}
```

```
struct Square
{
   struct Point BottomLeft;
   struct Point TopRight;
}

void main() {
   struct Square sq;
   int Area;
   sq.BottomLeft.Row = 1;
   sq.BottomLeft.Column = 15;
   sq.TopRight.Row = 82;
   sq.TopRight.Column = 120;
   Area = AreaOfASquare(sq);
}

int AreaOfASquare(struct Square s) {
   int width, height;
   width = s.TopRight.Column - s.BottomLeft.Column + 1;
   height = s.TopRight.Row - s.BottomLeft.Row + 1;
   return( width * height );
}
```

can be easily converted into Javascript code as shown in the following.

```
function main() {
   var sq.BottomLeft.Row = 1;
   sq.BottomLeft.Column = 15;
   sq.TopRight.Row = 82;
   sq.TopRight.Column = 120;
   var Area = AreaOfASquare(sq);
}

function AreaOfASquare(s) {
   var width = s.TopRight.Column - s.BottomLeft.Column + 1;
   var height = s.TopRight.Row - s.BottomLeft.Row + 1;
   return( width * height );
}
```

Structures can be passed, returned, and modified just as any other variable. Of course, structures and arrays are different and independent, which allows a statement like the following.

```
foo[8].animal.forge[3] = bil.bo
```

Some operations, such as addition, are not defined for structures.

2.6 Pointer operator * and address operator &

No pointers. None. The * symbol never means pointer in Javascript, which might cause seasoned C programmers to gasp in disbelief. But the situation turns out not to be such a big deal. The pointer operator is easily replaced. For example, `*var` can be replaced by `var[0]`.

2.7 Case statements

Case statements in a switch statement may be constants, variables, or other statements that can be evaluated to a value. The following switch statement has case statements which are valid in Javascript.

```
switch(i)
{
   case 4:
   case foe():
   case "thorax":
   case Math.sqrt(foe()):
   case (PILLBOX * 3 - 2):
   default:
}
```

2.8 Initialization code which is external to functions

All code not inside a function block is interpreted before `main()` is called and can be thought of as initialization code. When a script has initialization code outside of functions and code inside of functions, it shares characteristics of both batch and program scripts. Thus, the following Javascript code:

```
writeLog("first ");

function main()
{
   writeLog("third.");
}

writeLog("second ");
```

results in the following output:

```
first second third.
```

2.9 Unnecessary tokens

If symbols are redundant, they are usually unnecessary in Javascript which allows more flexibility in writing scripts and is less onerous for users not trained in C. Semicolons that end statements are usually redundant and do not do anything extra when a script is interpreted. C programmers are trained to use semicolons to end statements, a practice that can be followed in Javascript. Indeed, some programmers think that the use of semicolons in Javascript is a good to be pursued. Many people who are not trained in C wonder at the use of redundant semicolons and are sometimes confused by their use. The use of semicolons is personal. If a programmer wants to use them, then he should, but if he does not want to, then he should not.

In Javascript the two statements, "`foo()`" and "`foo();`" are identical. It does not hurt to use semicolons, especially when used with return statements, such as "`return;`". But widespread or regular use of semicolons simply is not necessary. Similarly, parentheses, "(" and ")", are often unnecessary. For example, the following fragment is valid and results in both of the variables, n and x, being equal to 7.

```
var n = 1 + 2 * 3  var x = 2 * 3 + 1
```

The following fragment is identical and is clearer, but it requires more typing because of the addition of redundant tokens.

```
var n = 1 + (2 * 3); var x = (2 * 3) + 1;
```

The fragments could be rewritten to be:

```
var n = 1 + 2 * 3
var x = 2 * 3 + 1
```

and:

```
var n = 1 + (2 * 3);
var x = (2 * 3) + 1;
```

Which fragment is better? The answer depends on personal taste. Efforts to standardize programming styles over the last three decades have been abysmal failures, not unlike efforts to control the Internet.

2.10 Macros

Function macros are not supported. Since speed is not of primary importance in a scripting language, a macro gains little over a function call. Macros simply become functions.

2.11 Token replacement macros

The #define preprocessor directive, which can be thought of and used as a macro, is supported by Javascript. As an example, the following token replacement is recognized and implemented during the preprocessing phase of script interpretation.

```
#define NULL 0
```

2.12 Back quote strings

Back quotes are not used at all for strings in the C language. The back quote character, `, also known as a back-tick or grave accent, may be used in Javascript in place of double or single quotes to specify strings. However, strings that are delimited by back quotes do not translate escape sequences. For example, the following two lines describe the same file name:

```
"c:\\autoexec.bat"  // traditional C method, which is also
                    // valid in Javascript
`c:\autoexec.bat`   // alternative Javascript method
```

2.13 Converting existing C code to Javascript

Converting existing C code to Javascript is mostly a process of deleting unnecessary text. Type declarations, such as *int*, *float*, *struct*, *char*, and `[]`, should be deleted. The following two columns give examples of how to make such changes. C code is on the left and can be replaced by the Javascript code on the right.

C

```
int i;
int foo = 3;
struct
{
   int row;
   int col;
}
char name[] = "George";
int goo(int a, char *s, int c);
int zoo[] = {1, 2, 3};
```

Javascript

```
var i; // or nothing
var foo = 3;
var st; // no struct type
   // Simply use st.row
   // and st.col
   // when needed.

var name = "George";
var goo(a, buf, c);
var zoo = {1, 2, 3};
```

Another step in converting C to Javascript is to search for pointer and address operators, `*` and `&`. Since the `*` operator and `&` operator work together when the address of a variable is passed to a function, these operators are unnecessary in the C portion of Javascript. If code has `*` operators in it, they usually refer to the base value of a pointer address. A statement like "`*foo = 4`" can be replaced by "`foo[0] = 4`".

Finally, the `->` operator in C which is used with `structures` may be replaced by a period for values passed by address and then by reference.

3. Javascript API reference

3.1 Array Object

An Array object is an object in Javascript and is in the underlying ECMAScript standard. Be careful not to confuse an array variable that has been constructed as an instance of the Array object with the automatic or dynamic arrays of Javascript. Javascript offers automatic arrays in addition to the Array object of ECMAScript. The purpose is to ease the programming task by providing another easy to use tool for scripters. The current section is about Array objects.

An Array is a special class of object that refers to its properties with numbers rather than with variable names. Properties of an Array object are called elements of the array. The number used to identify an element, called an index, is written in brackets following an array name. Array indices must be either numbers or strings.
Array elements can be of any data type. The elements in an array do not all need to be of the same type, and there is no limit to the number of elements an array may have.

The following statements demonstrate assigning values to arrays.

```
var array = new Array();
array[0] = "fish";
array[1] = "fowl";
array["joe"] = new Rectangle(3,4);
array[foo] = "creeping things"
array[goo + 1] = "etc."
```

The variables foo and goo must be either numbers or strings.
Since arrays use a number to identify the data they contain, they provide an easy way to work with sequential data. For example, suppose you wanted to keep track of how many jellybeans you ate each day, so you can graph your jellybean consumption at the end of the month. Arrays provide an ideal solution for storing such data.

```
var April = new Array();
April[1] = 233;
April[2] = 344;
April[3] = 155;
April[4] = 32;
```

Now you have all your data stored conveniently in one variable. You can find out how many jellybeans you ate on day x by checking the value of April[x]:

```
for(var x = 1; x < 32; x++)
   writeLog("On April " + x + " I ate " + April[x] + " jellybeans.");
```

Arrays usually start at index [0], not index [1]. Note that arrays do not have to be continuous, that is, you can have an array with elements at indices 0 and 2 but none at 1.

3.1.1 Creating arrays

Like other objects, arrays are created using the new operator and the Array constructor function. There are three possible ways to use this function to create an array. The simplest is to call the function with no parameters:

```
var a = new Array();
```

This line initializes variable a as an array with no elements. The parentheses are optional when creating a new array, if there are no arguments. If you wish to create an array of a predefined size, pass variable a the size as a parameter of the `Array()` function. The following line creates an array with a length of the size passed.

```
var b = new Array(31);
```

In this case, an array with length 31 is created.
Finally, you can pass a list of elements to the `Array()`function, which creates an array containing all of the parameters passed. For example:

```
var c = new Array(5, 4, 3, 2, 1, "blast off");
```

creates an array with a length of 6. c[0] is set to 5, c[1] is set to 4, and so on up to c[5], which is set to the string "blast off". Note that the first element of the array is array[0], not array[1]. Arrays may also be created dynamically. By referring to a variable with an index in brackets, a variable is created as or converted to an array. The array that is created is an automatic or dynamic array which is different than an instance of an `Array object` created as described in this section. Automatic arrays, created as described in this paragraph, are unable to use the methods and properties described below, so it is recommended that you use, in most circumstances, the `new Array()` constructor function to create arrays.

Initializers for arrays and objects

Variables may be initialized as objects and arrays using lists inside of "{}" and "[]". By using these initializers, instances of Objects and Arrays may be created without using the `new` constructor. Objects may be initialized using syntax similar to the following:

```
var o = {a:1, b:2, c:3};
```

This line creates a new object with the properties a, b, and c set to the values shown. The properties may be used with normal object syntax, for example, `o.a == 1`.
Arrays may be initialized using syntax similar to the following:

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

This line creates a new array with three elements set to 1, 2, and 3. The elements may be used with normal array syntax, for example, `a[0] == 1`.
The distinction between Object and Array initializer might be a bit confusing when using a line with syntax similar to the following:

```
var a = {1, 2, 3};
```

This line also creates a new array with three elements set to 1, 2, and 3. The line differs from the first line, Object initializer, in that there are no property identifiers and differs from the second line, Array initializer, in that it uses "{}" instead of "[]". In fact, the second and third lines produce the same results. The elements may be used with normal array syntax, for example, `a[0] == 1`.

The following code fragment shows the differences.

```
var o = {a:1, b:2, c:3};
```

```
writeLog(typeof o +" | "+ o._class +" | "+ o);

var a = [1, 2, 3];
writeLog(typeof a +" | "+ a._class +" | "+ a);

var a= {1, 2, 3};
writeLog(typeof a +" | "+ a._class +" | "+ a);
```

The display from this code is:

```
object | Object | [object Object]
object | Array | 1,2,3
object | Array | 1,2,3
```

As shown in the first display line, the variable `o` is created and initialized as an Object. The second and third lines both initialize the variable `a` as an Array. Notice that in all cases the `typeof` the variable is object, but the class, which corresponds to the particular object and which is reflected in the `_class` property, shows which specific object is created and initialized.

3.1.2 Array object instance properties

Array length

SYNTAX:	`array.length`
DESCRIPTION:	The length property returns one more than the largest index of the array. Note that this value does not necessarily represent the actual number of elements in an array, since elements do not have to be contiguous. By changing the value of the length property, you can remove array elements. For example, if you change `ant.length` to 2, ant will only have the first two members, and the values stored at the other indices will be lost. If we set bee.length to 2, then bee will consist of two members: `bee[0]`, with a value of 88, and `bee[1]`, with an `undefined` value.
SEE:	Array(), global.getArrayLength(), global.setArrayLength()
EXAMPLE:	

```
// Suppose we had two arrays "ant" and "bee",
// with the following elements:

var ant = new Array();
ant[0] = 3;
ant[1] = 4;
ant[2] = 5;
ant[3] = 6;

var bee = new Array();
bee[0] = 88;
bee[3] = 99;

// The length property of both ant and bee
// is equal to 4, even though ant has twice
// as many actual elements as bee does.
```

3.1.3 Array object instance methods

Array()

SYNTAX:	`new Array(length)`
	`new Array([element1, ...])`

WHERE:	length - If this is a number, then it is the length of the array to be created. Otherwise, it is the element of a single-element array to be created. elementN - list of elements to be in the new Array object being created.
RETURN:	object - an `Array object` of the length specified or an Array object with the elements specified.
DESCRIPTION:	The array returned from this function is an empty array whose length is equal to the `length` parameter. If `length` is not a number, then the length of the new array is set to 1, and the first element is set to the `length` parameter. Note that this can also be called as a function, without the new operator. The alternate form of the Array constructor initializes the elements of the new array with the arguments passed to the function. The arguments are inserted in order into the array, starting with element 0. The length of the new array is set to the total number of arguments. If no arguments are supplied, then an empty array of length 0 is created.
SEE:	Automatic array allocation

EXAMPLE	`var a = new Array(5);`
	`var a = new Array(1,"two",three);`

Array concat()

SYNTAX:	`array.concat([element1, ...])`
WHERE:	elementN - list of elements to be concatenated to this Array object.
RETURN:	object - a new array consisting of the elements of the current object, with any additional arguments appended.
DESCRIPTION:	The return array is first constructed to consist of the elements of the current object. If the current object is not an Array object, then the object is converted to a string and inserted as the first element of the newly created array. This method then cycles through all of the arguments, and if they are arrays then the elements of the array are appended to the end of the return array, including empty elements. If an argument is not an array, then it is first converted to a string and appended as the last element of the array. The length of the newly created array is adjusted to reflect the new length. Note that the original object remains unaltered.
SEE:	String concat()

EXAMPLE	`var a = new Array(1,2);`
	`var b = a.concat(3);`

Array join()

SYNTAX:	`array.join([separator])`
WHERE:	separator - a value to be converted to a string and used to separate the list of array elements. The default is an empty string.
RETURN:	string - string consisting of the elements, delimited by separator, of an array.
DESCRIPTION:	The elements of the current object, from 0 to the length of the object, are sequentially converted to strings and appended to the return string. In between each element, the separator is added. If `separator` is not supplied, then the single-character string "," is used. The string conversion is the standard conversion, except the `undefined` and `null` elements are converted to the empty string "". The `Array join()` method creates a string of all of array elements. The `join()` method has an optional parameter, a string which represents the character or characters that will separate the array elements. By default, the array elements will be separated by a comma. For example: `var a = new Array(3, 5, 6, 3);`

```
var string = a.join();
```

will set the value of "string" to "3,5,6,3". You can use another string to separate the array elements by passing it as an optional parameter to the `join()` method. For example,

```
var a = new Array(3, 5, 6, 3);
var string = a.join("*/*");
```

```
creates the string "3*/*5*/*6*/*3".
```

see: Array toString()

EXAMPLE

```
// The following code:

var array = new Array( "one", 2, 3, undefined );
writeLog( array.join("::") );

// Will print out the string "one::2::3::".
```

Array pop()

SYNTAX: Array.pop()

RETURN: value - the last element of the current `Array object`. The element is removed from the array after being returned.

DESCRIPTION: this method first gets the length of the current object. If the length is `undefined` or 0, then `undefined` is returned. Otherwise, the element at this index is returned. This element is then deleted, and the length of current object is decreased by one. The `pop()` method works on the end of an array, whereas, the `Array shift()` method works on the beginning.

see: Array push()

EXAMPLE

```
// The following code:

var array = new Array( "four" );
writeLog( array.pop() );
writeLog( array.pop() );

// Will first print out the string "four", and then print out
// "undefined", which is the result of converting the undefined
// value to a string. The array will be empty after these calls.
```

Array push()

SYNTAX: `Array.push([element1, ...])`

WHERE: elementN - a list of elements to append to the end of an array.

RETURN: number - the length of the new array.

DESCRIPTION: this method appends the arguments to the end of this array, in the order that they appear. The length of the current `Array object` is adjusted to reflect the change.

see: Array pop()

EXAMPLE

```
// The following code:

var array = new Array( 1, 2 );
array.push( 3, 4 );
writeLog( array );

// Will print the array converted to the string "1,2,3,4".
```

Array reverse()

SYNTAX: Array.reverse()

RETURN: object - a new array consisting of the elements in the current Array object in reverse order.

DESCRIPTION: If the length of the current `Array object` is 0, then the current Array object is simply returned. Otherwise, a new Array object is created, and the elements of the current Array object are put into this new array in reverse order, preserving any empty or `undefined` elements.

EXAMPLE

```
var a = new Array(1,2,3);
var b = a.reverse();

// The following code:
var array = new Array;
array[0] = "ant";
array[1] = "bee";
array[2] = "wasp";
array.reverse();

// Produces the following array:
array[0] == "wasp"
array[1] == "bee"
array[2] == "ant"
```

Array shift()

SYNTAX: `array.shift()`

RETURN: value - the first element of the current Array object. The element is removed from the array after being returned.

DESCRIPTION: if the length of the current `Array object` is 0, then `undefined` is returned. Otherwise, the first element is returned. This element is deleted from the array, and any remaining elements are shifted down to fill the gap that was created. The `shift()` method works on the beginning of an array, whereas, the `Array pop()` method works on the end.

SEE: Array unshift(), Array pop()

EXAMPLE

```
//The following code:
var array = new Array( 1, 2, 3 );
writeLog( array.shift() );
writeLog( array );

// First prints out "1", and then the contents of the array,
// which converts to the string "2,3".
```

Array slice()

SYNTAX: `array.slice(start[, end])`

WHERE: start - the element offset to start from.
end - the element offset to end at.

RETURN: object - a new array containing the elements of the current object from `start` up to, but not including, element end.

DESCRIPTION: this method creates a subset of the current array. If end is not supplied, then the length of the current object is used instead. If either `start` or end is negative, then it is treated as an offset from the end of the array, and the value `length+start` or `length+end` is used instead. If either is beyond the length of the array, then the length is used instead. If either is less than 0 after adjusting for negative values, then the value 0 is used instead. The elements are then copied into the newly created array, starting at `start` and proceeding to (but not including) end.

see: String substring()

EXAMPLE

```
// The following code:
var array = new Array( 1, 2, 3, 4 );
writeLog( array.slice( 1, -1 ) );
```

```
// Print out the elements from 1 up to 4,
// which results in the string "2,3".
```

Array sort()

SYNTAX: `Array.sort([compareFunction])`

WHERE: compareFunction - identifier for a function which expects two parameters x and y, and returns a negative value if x < y, zero if x = y, or a positive value if x > y.

RETURN: object - this `Array object` after being sorted.

DESCRIPTION: this method sorts the elements of the array. The sort is not necessarily stable (that is, elements which compare equal do not necessarily remain in their original order). The comparison of elements is done based on the supplied `compareFunction`. If `compareFunction` is not supplied, then the elements are converted to strings and compared. Non-existent elements are always greater than any other element, and consequently are sorted to the end of the array. Undefined values are also always greater than any defined element, and appear at the end of the Array before any empty values. Once these two tests are performed, then the appropriate comparison is done.

If a compare function is supplied, the array elements are sorted according to the return value of the compare function. If a and b are two elements being compared, then:

If `compareFunction(a, b)` is less than zero, sort b to a lower index than a.

If `compareFunction(a, b)` returns zero, leave a and b unchanged relative to each other.

If `compareFunction(a, b)` is greater than zero, sort b to a higher index than a.

EXAMPLE

```
// Consider the following code,
// which sorts based on numerical values,
// rather than the default string comparison.

function compare( x, y )
{
   x = ToNumber(x);
   y = ToNumber(y);

   if( x < y )
      return -1;
   else if ( x == y )
      return 0;
   else
      return 1;
}

var array = new Array( 3, undefined, "4", -1 );
array.sort(compare);
writeLog(array);

// Prints out the sorted array,
// which is "-1,3,4,,".
//  Notice the undefined value
// at the end of the array.
```

Array splice()

SYNTAX: `Array.splice(start, deleteCount[, element1, ...])`

WHERE: start - the index at which to splice in the items. If this is negative, then (length+start) is used instead, and if it beyond the end of the array, then the length of the array is used.

deletecount - the number of items to remove from the array.

elementN - a list of elements to insert into the array in place of the ones which were deleted.

RETURN: object - an array consisting of the elements which were removed from the current `Array object`.

DESCRIPTION: this method splices in any supplied elements in place of any elements deleted. Beginning at index `start`, `deleteCount elements` are first deleted from the array and inserted into the newly created return array in the same order. The elements of the current object are then adjusted to make room for the all of the items passed to this method. The remaining arguments are then inserted sequentially in the space created in the current object.

SEE: Array push()

EXAMPLE

```
// The following code:
var array = new Array( 1, 2, 3, 4, 5 );
writeLog( array.splice( 1, 2, 6, 7, 8 ) );
writeLog( array );

// Will print "2,3" and then "1,6,7,8,4,5".
// The array has been modified to include the extra items in
// place of those that were deleted.
```

Array toString()

SYNTAX: `Array.toString()`

RETURN: string - string representation of an `Array object`.

DESCRIPTION: this method behaves exactly the same as if `Array join()` was called on the current object with no arguments. The result is a string consisting of the string representation of the array elements (except for `null` and `undefined`, which are empty strings) separated by commas.

SEE: Array join()

EXAMPLE

```
// The following code:
var array = new Array( 1, "two", , null, false );
writeLog( array.toString() );

// Will print out the string "1,two,,,false".
// Note that this method is rarely called,
// rather the function ToString() is used,
// which implicitly calls this method.
```

Array unshift()

SYNTAX: `Array.unshift([element1, ...])`

WHERE: elementN - a list of items to insert at the beginning of the array.

RETURN: number - the length of the new array after inserting the items.

DESCRIPTION: any arguments are inserted at the beginning of the array, such that their order within the array is the same as the order in which they appear in the argument list. Note that this method is the opposite of Array.push(), which adds the items to the end of the array.

SEE: Array shift(), Array push()

EXAMPLE

```
var a = new Array(2,3);
var b = a.unshift(1);
```

3.2 Boolean Object

3.2.1 Boolean object instance methods

Boolean()

SYNTAX:	`new Boolean(value)`
WHERE:	value - a value to be converted to a boolean.
RETURN:	object - a Boolean object with the parameter value converted to a boolean value.
DESCRIPTION:	this function creates a `Boolean object` that has the parameter value converted to a boolean value. If the function is called without the new constructor, then the return is simply the parameter value converted to a boolean.
SEE:	Boolean toString()

EXAMPLE	`var name = "Joe";`
	`var b = new Boolean( name == "Joe" );`
	`// The Boolean object "b" is now true.`

Boolean.toString()

SYNTAX:	`Boolean.toString()`
RETURN:	string - "true" or "false" according to the value of the Boolean object.
DESCRIPTION:	this `toString()` method returns a string corresponding to the value of a `Boolean object` or primitive data type.

EXAMPLE	`var name = "Joe";`
	`var b = new Boolean( name === "Joe" );`
	`var bb = false;`
	`writeLog( b.toString() );   // "true"`
	`writeLog( bb.toString() );  // "false"`

3.3 Buffer Object

The Buffer object provides a way to manipulate data at a very basic level. It is needed whenever the relative location of data in memory is important. Any type of data may be stored in a Buffer object. A new Buffer object may be created from scratch or from a string, buffer, or Buffer object, in which case the contents of the string or buffer will be copied into the newly created Buffer object.

NOTE: the Javascript Buffer Object is not the same as the FieldCommander data buffer that is created with the `addBuffer()` command.

3.3.1 Buffer object instance properties

Buffer bigEndian

SYNTAX:	`buffer.bigEndian`
DESCRIPTION:	This property is a boolean flag specifying whether to use bigEndian byte ordering when calling `Buffer getValue()` and `Buffer putValue()`. This value is set when a buffer is created, but may be changed at any time. This property defaults to the state of the underlying OS and processor.
SEE:	Buffer unicode
EXAMPLE:	`buffer.bigEndian = true;`

Buffer cursor

SYNTAX:	`buffer.cursor`
DESCRIPTION:	The current position within a buffer. This value is always between 0 and `.size`. It can be assigned to as well. If a user attempts to move the cursor beyond the end of

	a buffer, than the buffer is extended to accommodate the new position, and filled with `null` bytes. If a user attempts to set the cursor to less than 0, then it is set to the beginning of the buffer, to position 0.
SEE:	Buffer bigEndian
EXAMPLE:	`var p = buffer.cursor;`

Buffer data

SYNTAX:	`buffer.data`
DESCRIPTION:	This property is a reference to the internal data of a buffer. It is only a temporary value to assist in passing parameters to OS and system library type calls. In the future, all Javascript library functions should be able to recognize Buffer objects and to get this member on their own.
SEE:	Buffer size

Buffer size

SYNTAX:	`buffer.size`
DESCRIPTION:	The size of the `Buffer object`. This property may be assigned to, such as `foo.size = 5`. If a user changes the size of the buffer to something larger, then it is filled with `NULL` bytes. If the user sets the size to a value smaller than the current position of the cursor, then the cursor is moved to the end of the new buffer.
SEE:	Buffer cursor
EXAMPLE:	`var n = buffer.size;`

Buffer unicode

SYNTAX:	`buffer.unicode`
DESCRIPTION:	This property is a boolean flag specifying whether to use unicode strings when calling `Buffer getString()` and `Buffer putString()`. This value is set when the buffer is created, but may be changed at any time. This property defaults to the unicode status of the underlying Javascript engine.
SEE:	Buffer bigEndian
EXAMPLE:	`buffer.bigEndian = false;`

Buffer[] Array

SYNTAX:	`Buffer[offset]`
DESCRIPTION:	This is an array-like version of the `Buffer getValue()` and `Buffer putValue()` methods, which works only with bytes. A user may either get or set these values, such as `goo = foo[5]` or `foo[5] = goo`. Every get/put operation uses byte types, that is, `SWORD8`. If offset is less than 0, then 0 is used. If offset is beyond the end of a buffer, the size of the buffer is extended with `null` bytes to accommodate it.
SEE:	Buffer getValue(), Buffer putValue()

EXAMPLE:	`var c = 'a';`
	`buffer[5] = c;`
	`c = buffer[4];`

3.3.2 Buffer object instance methods

Buffer()

SYNTAX:	`new Buffer([size[, unicode[, bigEndian]]])`
	`new Buffer(string[, unicode[, bigEndian]]])`
	`new Buffer(buffer[, unicode[, bigEndian]]])`
	`new Buffer(bufferObject)`

WHERE:	size - size of buffer to be created. string - string of characters from which to create a buffer.

buffer - buffer of characters from which to create another buffer.
bufferObject - buffer to be duplicated.
unicode - boolean flag for the initial state of the unicode property of the buffer
bigEndian - numeric description of the initial state of the bigEndian property of the buffer.

RETURN: object - the new buffer created.

DESCRIPTION: To create a `Buffer object`, follow of the syntax below.

```
new Buffer([size[, unicode[, bigEndian]]]);
```

A line of code following this syntax creates a new Buffer object. If size is specified, then the new buffer is created with the specified size, filled with `null` bytes. If no size is specified, then the buffer is created with a size of 0, though it can be extended dynamically later. The unicode parameter is an optional boolean flag describing the initial state of the .unicode flag of the object. Similarly, bigEndian describes the initial state of the bigEndian parameter of the buffer. If unspecified, these parameters default to the values described below.

```
new Buffer(string[, unicode[, bigEndian]]]);
```

A line of code following this syntax creates a new Buffer object from the string provided. If string is a unicode string (unicode is enabled within the application), then the buffer is created as a unicode string. This behavior can be overridden by specifying `true` or `false` with the optional boolean unicode parameter. If this parameter is set to `false`, then the buffer is created as an ASCII string, regardless of whether or not the original string was in unicode or not. Similarly, specifying `true` will ensure that the buffer is created as a unicode string. The size of the buffer is the length of the string (twice the length if it is unicode). This constructor does not add a terminating `null` byte at the end of the string. The bigEndian flag behaves the same way as in the first constructor.

```
new Buffer(buffer[, unicode[, bigEndian]])
```

A line of code following this syntax creates a new Buffer object from the buffer provided. The contents of the buffer are copied as is into the new Buffer object. The unicode and bigEndian parameters do not affect this conversion, though they do set the relevant flags for future use.

```
new Buffer(bufferObject);
```

A line of code following this syntax creates a new Buffer object from another Buffer object. Everything is duplicated exactly from the other bufferObject, including the cursor location, size, and data.
All of the above calls have an equivalent call form (such as `Buffer(15)`), except that this simply returns the buffer part (equivalent to the data member), rather than the entire Buffer object.

Buffer getString()

SYNTAX: `buffer.getString([length])`

WHERE: length - number of characters to get from the buffer.

RETURN: string - starting from the current cursor location and continuing for length bytes. If no length is specified, then the method reads until a `null` byte is encountered or the end of the buffer is reached.

DESCRIPTION: The string is read according to the value of the .unicode flag of the buffer. A terminating `null` byte is not added, even if a length parameter is not provided.

SEE: Buffer putString()

EXAMPLE:

```
foo = new Buffer("abcd");
foo.cursor = 1;
goo = foo.getString(2);
//goo is now "bc"
```

Buffer getValue()

SYNTAX:	`buffer.getValue([valueSize[, valueType]])`
WHERE:	valueSize - a positive number describing the number of bytes to be used and defaults to 1. The following are acceptable values: 1,2,3,4,8, and 10 valueType - One of the following types: "`signed`", "`unsigned`", or "`float`". The default type is: "`signed`."
RETURN:	value - from the specified position in a `Buffer object`.
DESCRIPTION:	This call is similar to the `Buffer putValue()` function, except that it gets a value instead of puts a value.
SEE:	Buffer putValue(), Buffer[] Array
EXAMPLE:	

```
/*
To explicitly put a value at a specific location
while preserving the cursor location,
do something similar to the following.
*/

// Save the old cursor location
var oldCursor = foo.cursor;
// Set to new location
foo.cursor = 20;
// Get goo at offset 20
bar = foo.getValue(goo);
// Restore cursor location
foo.cursor = oldCursor

//Please see Buffer.putValue
// for a more complete description.
```

Buffer putString()

SYNTAX:	`buffer.putString(string)`
WHERE:	string - Any string.
RETURN:	void.
DESCRIPTION:	This method puts a string into the `Buffer object` at the current cursor position. If the .unicode flag is set within the Buffer object, then the string is put as a unicode string, otherwise it is put as an ASCII string. The cursor is incremented by the length of the string (or twice the length if it is put as a unicode string). Note that terminating `null` byte is not added at end of the string.
EXAMPLE:	

```
// To put a null terminated string,
// the following can be done.

// Put the string into the buffer
foo.putString( "Hello" );
// Add terminating null byte
foo.putValue( 0 );
```

Buffer putValue()

SYNTAX:	`buffer.putValue(value[, valueSize[, valueType]])`
WHERE:	value - value to be put into the buffer. valueSize - a positive number describing the number of bytes to be used and defaults to 1. The following are acceptable values: 1,2,3,4,8, and 10 valueType - One of the following types: "`signed`", "`unsigned`", or "`float`". The default type is: "`signed`."
RETURN:	The value is put into buffer at the current cursor position, and the cursor value is automatically incremented by the size of the value to reflect this addition.
DESCRIPTION:	This method puts the specified value into a buffer. The value must be a number. The parameter `valueSize` or both `valueSize` and `valueType` may be passed as additional parameters. The parameter valueSize is a positive number describing the number of bytes to be used and defaults to 1. Acceptable values for `valueSize` are

1,2,3,4,8, and 10, providing that it does not conflict with the optional `valueType` flag. (See listing below.)
The parameter valueType must be one of the following: "`signed`", "`unsigned`", or "`float`". It defaults to "`signed`." The `valueType` parameter describes the type of data to be read. Combined with valueSize, any type of data can be put. The following list describes the acceptable combinations of valueSize and valueType:

```
 valueSize  valueType
 1          signed, unsigned
 2          signed, unsigned
 3          signed, unsigned
 4          signed, unsigned, float
 8          float
 10         float  (Not supported on every system)
```

Any other combination will cause an error. The value is put into buffer at the current cursor position, and the cursor value is automatically incremented by the size of the value to reflect this addition.

SEE: Buffer getValue(), Buffer[] Array

EXAMPLE:

```
/*
To explicitly put a value at a specific location
while preserving the cursor location,
do something similar to the following.
*/

var oldCursor = foo.cursor;
// Save the old cursor location
foo.cursor = 20;
// Set to new location
foo.putValue(goo);
// Put goo at offset 20
foo.cursor = oldCursor
// Restore cursor location

/*.
The value is put into the buffer with byte-ordering
according to the current setting of the .bigEndian
flag. Note that when putting float values as a
smaller size, such as 4, some significant figures
are lost. A value such as "1.4" will actually be
converted to something to the effect
of "1.39999974". This is sufficiently
insignificant to ignore, but note
that the following does not hold true.
.*/

foo.putValue(1.4,4,"float");
foo.cursor -= 4;
if( foo.getValue(4,"float") != 1.4 )
// This is not necessarily true due
// to significant figure loss.

/*.
This situation can be prevented by using 8 or 10
as a valueSize instead of 4. A valueSize of 4
may still be used for floating point values,
but be aware that some loss of significant figures
may occur (though it may not be enough
to affect most calculations).
.*/
```

Buffer subBuffer()

SYNTAX:	`buffer.subBuffer(begin, end)`
WHERE:	begin - start of offset end - end of offset (up to but not including this point)
RETURN:	object - another `Buffer object` consisting of the data between the positions specified by the parameters: beginning and end.
DESCRIPTION:	If the parameter beginning is less than 0, then it is treated as 0, the start of the buffer. If the parameter end is beyond the end of the buffer, then the new sub-buffer is extended with `null` bytes, but the original buffer is not altered.
SEE:	String subString()

EXAMPLE:	`foo = new Buffer("abcd");`
	`bar = foo.subBuffer(1,3);`
	`// bar is now the string "bc"`
	`// "a" was at position 0, "b" at position 1, etc.`
	`// The parameter "3"`
	`// or "nEnd" is the postion to go up to,`
	`// but NOT to be included in the string.`

Buffer toString()

SYNTAX:	`buffer.toString()`
RETURN:	string - a string equivalent of the current state of the buffer, with all characters, including `"\0"`.
DESCRIPTION:	Any conversion to or from unicode is done according to the `.unicode` flag of the object.
SEE:	Buffer getString()

EXAMPLE:	`foo = new Buffer("hello");`
	`bar = foo.toString(void);`
	`//bar is now the string "hello"`

3.4 Clib Object

The Clib object contains functions that are a part of the standard C library. Methods to access files and formatted strings are part of the Clib object.

3.4.1 File I/O

Clib.fopen()

SYNTAX:	`Clib.fopen(filename, mode)`
WHERE:	filename - a string with a filename to open. mode - how or for what operations the file will be opened.
RETURN:	number - a file pointer to the file opened, `null` in case of failure.
DESCRIPTION:	This method opens the file specified by filename for file operations specified by mode, returning a file pointer to the file opened. `null` is returned in case of failure. The parameter filename is a string. It may be any valid file name, excluding wildcard characters. The parameter mode is a string composed of one or more of the following characters. For example, "`r`" or "`rt`" • `r` open file for reading; file must already exist • `w` open file for writing; create if doesn't exist; if file exists then truncate to zero length • `a` open file for append; create if doesn't exist; set for writing at end-of-file • `b` binary mode; if b is not specified then open file in text mode (end-of-line

translation)

- t
  text mode
- +
  open for update (reading and writing)

When a file is successfully opened, its error status is cleared and a buffer is initialized for automatic buffering of reads and writes to the file.

SEE:	Clib.fclose()

EXAMPLE:

```
// Open the text file "ReadMe.txt"
// for text mode reading, and display each line in the file.

var fp = Clib.fopen("ReadMe.txt", "r");
if ( fp == null )
   writeLog("Error opening file for reading.");
else
   while ( null != (line=Clib.fgets(fp)) ) {
      Clib.fputs(line, stdout);
   }
Clib.fclose(fp);
```

Clib.fclose()

SYNTAX:	`Clib.fclose(filePointer)`
WHERE:	filePointer - pointer to file to close.
RETURN:	number - 0 on success, else `EOF`.
DESCRIPTION:	The parameter filePointer is a file pointer as returned by `Clib.fopen()`. This method flushes the file buffers of a stream and closes the file. The file pointer ceases to be valid after this call. Returns zero if successful, otherwise returns `EOF`.
SEE:	Clib.fopen()

Clib.feof()

SYNTAX:	`Clib.feof(filePointer)`
WHERE:	filePointer - pointer to file to use.
RETURN:	number - a non-zero number if at end of file, else 0.
DESCRIPTION:	The parameter filePointer is a file pointer as returned by `Clib.fopen()`. This method returns an integer which is non-zero if the file cursor is at the end of the file, and 0 if it is NOT at the end of the file.
SEE:	Clib.fopen()

Clib.fflush()

SYNTAX:	`Clib.fflush(filePointer)`
WHERE:	filePointer - pointer to file to use.
RETURN:	number - 0 on success, else `EOF`.
DESCRIPTION:	Causes any unwritten buffered data to be written to filePointer. If filePointer is `null` then flushes buffers in all open files. Returns zero if successful; otherwise `EOF`.
SEE:	Clib.fclose()

Clib.fgetc()

SYNTAX:	`Clib.fgetc(filePointer)`
WHERE:	filePointer - pointer to file to use.
RETURN:	number - `EOF` if there is a read error or the file cursor is at the end of the file. If there is a read error then `Clib.ferror()` will indicate the error condition.
DESCRIPTION:	This method returns the next character in the file stream indicated by filePointer as a byte converted to an integer.
SEE:	Clib.fgets()

Clib.fgetpos()

SYNTAX:	`Clib.fgetpos(filePointer, pos)`
WHERE:	filePointer - pointer to file to use. pos - variable to hold the current file position.
RETURN:	number - 0 on success, else non-zero and stores an error value in `Clib.errno`.
DESCRIPTION:	This method stores the current position of the file stream filePointer for future restoration using `Clib.fsetpos()`. The file position will be stored in the variable pos; use it with `Clib.fsetpos()` to restore the cursor to its position.
SEE:	Clib.fsetpos()

Clib.fgets()

SYNTAX:	`Clib.fgets([length,] filePointer)`
WHERE:	length - maximum length of string. filePointer - pointer to file to use.
RETURN:	string - the characters in a file from the current file cursor to the next newline character on success, else `null`.
DESCRIPTION:	This method returns a string consisting of the characters in a file from the current file cursor to the next newline character. The newline will be returned as part of the string. If there is an error or the end of the file is reached, `null` will be returned. A second syntax of this function takes a number as its first parameter. This number is the maximum length of the string to be returned if no newline character was encountered.
SEE:	Clib.fgetc()

Clib.fprintf()

SYNTAX:	`Clib.fprintf(filePointer, formatString[, variables ...])`
WHERE:	filePointer - pointer to file to use. formatString - string that specifies the final format. variables - values to be converted to and formatted as a string.
RETURN:	number - characters written on success, else a negative number.
DESCRIPTION:	This flexible function writes a formatted string to the file associated with filePointer. The second parameter, formatString, is a string of the same pattern as `Clib.sprintf()`.

Clib.fputc()

SYNTAX:	`Clib.fputc(chr, filePointer)`
WHERE:	chr - character to write to file. filePointer - pointer to file to use.
RETURN:	number - character written on success, else `EOF`.
DESCRIPTION:	If chr is a string, the first character of the string will be written to the file indicated by filePointer. If chr is a number, the character corresponding to its unicode value will be added.
SEE:	Clib.fputs()

Clib.fputs()

SYNTAX:	`Clib.fputs(str, filePointer)`
WHERE:	str - string to write to file. filePointer - pointer to file to use.
RETURN:	number - non-negative number on success, else `EOF`.
DESCRIPTION:	This method writes the value of str to the file indicated by filePointer. Returns `EOF` if write error, else returns a non-negative value.
SEE:	Clib.fputc()

Clib.fread()

SYNTAX: `Clib.fread(dstVar, varDescription, filePointer)`

WHERE: dstVar - variable to hold data read from file.
varDescription - description of the data to read, that is, how and how much.
filePointer - pointer to file to use.

RETURN: number - elements read on success, 0 on failure.

DESCRIPTION: This method reads data from an open file and stores it in dstVar. If it does not yet exist, dstVar will be created. varDescription is a variable that describes the how and how much data is to be read: if dstVar is a buffer, it will be the length of the buffer; if dstVar is an object, varDescription must be an object descriptor; and if dstVar is to hold a single datum then varDescription must be one of the following.

- `UWORD8`
  Stored as a byte in dstVar
- `SWORD8`
  Stored as an integer in dstVar
- `UWORD16`
  Stored as an integer in dstVar
- `SWORD16`
  Stored as an integer in dstVar
- `UWORD24`
  Stored as an integer in dstVar
- `SWORD24`
  Stored as an integer in dstVar
- `UWORD32`
  Stored as an integer in dstVar
- `SWORD32`
  Stored as an integer in dstVar
- `FLOAT32`
  Stored as a float in dstVar
- `FLOAT64`
  Stored as a float in dstVar
- `FLOAT80`
  Stored as a float in dstVar

In all cases, this function returns the number of elements read. For dstVar being a buffer, this would be the number of bytes read, up to length specified in varDescription. For dstVar being an object, this method returns 1 if the data is read or 0 if read error or end-of-file is encountered.
For example, the definition of an object might be:

```
 ClientDef.Sex = UWORD8;
 ClientDef.MaritalStatus = UWORD8;
 ClientDef._Unused1 = UWORD16;
 ClientDef.FirstName = 30; ClientDef.LastName = 40;
 ClientDef.Initial = UWORD8;
```

The Javascript version of `Clib.fread()` differs from the standard C version in that the standard C library is set up for reading arrays of numeric values or structures into consecutive bytes in memory. In Javascript, this is not necessarily the case. Data types will be read from the file in a byte-order described by the current value of the _BigEndianMode global variable.

SEE: Clib.fopen(), Clib.fwrite()

EXAMPLE:

```
// To read the 16 bit integer "i", the 32 bit float "f", and then 10
// byte buffer "buf" from the open file "fp" use code like this:
if ( !Clib.fread(i,SWORD16,fp) ||
     !Clib.fread(f,FLOAT32,fp) ||
     (10 != Clib.fread(buf,10,fp)) ) {
   writeLog("Error reading from file.");
}
```

Clib.freopen()

SYNTAX: `Clib.freopen(filename, mode, filePointer)`

WHERE:
filename - a string with a filename to open.
ode - how or for what operations the file will be opened.
filePointer - pointer to file to use.

RETURN: number - file pointer on success, else `null`.

DESCRIPTION: This method closes the file associated with filePointer, ignoring any close errors, opens filename according to mode, as with `Clib.fopen()`, and reassociates filePointer with the new file specification. The method returns a copy of the modified filePointer, or `null` if it fails.

SEE: Clib.fopen()

EXAMPLE:

```
if ( null == Clib.freopen("log.txt, "w", fp) )
   writeLog("Error reopening log file");
```

Clib.fscanf()

SYNTAX: `Clib.fscanf(filePointer, formatString[, variables ...])`

WHERE:
filePointer - pointer to file to use.
formatString - string that specifies the final format.
variables - values to be converted to and formatted as a string.

RETURN: number - input items assigned on success, else `EOF`.

DESCRIPTION: This flexible function reads input from the file indicated by filePointer and stores in parameters following formatString according the character combinations in the format string, which indicate how the file data is to be read and stored. The file must be open, with read access. It returns the number of input items assigned. This number may be fewer than the number of parameters requested if there was a matching failure. If there is an input failure, before the conversion occurs, this function returns `EOF`.

The formatString is formatted in the same way as Clib.sscanf().

EXAMPLE:

```
// Given the following text file, weight.dat:
//  Crow, Barney      180
//  Claus, Santa      306
//  Mouse, Mickey     2
// the following code:

var fp = Clib.fopen("weight.dat", "r");
var FormatString = "%[,] %*c %s %d";
while (3 == Clib.fscanf(fp, FormatString, LastName, Firstame, weight)) {
  var Message = FirstName+" "+LastName+" weighs "+weight+" pounds.";
  writeLog( Message );
}
Clib.fclose(fp);

// results in the following output:
//  Barney Crow weighs 180 pounds.
//  Santa Claus weighs 306 pounds.
//  Mickey Mouse weighs 2 pounds.
```

Clib.fseek()

SYNTAX: `Clib.fseek(filePointer, offset[, mode])`

WHERE:
filePointer - pointer to file to use.
offset - number of bytes past or offset from the point indicated by mode.
mode - file position to use as a starting point. Default is `SEEK_SET` and may be one of the following:

- `SEEK_CUR`
  seek is relative to the current position of the file
- `SEEK_END`
  position is relative from the end of the file

- SEEK_SET
  position is relative to the beginning of the file

RETURN: number - 0 on success, else non-zero.

DESCRIPTION: Set the position of the file pointer of the open file stream filePointer. The parameter offset is a number indicating how many bytes the new position will be past the starting point indicated by mode.
If mode is not supplied then absolute offset from the beginning of file, SEEK_SET, is assumed. For text files, not opened in binary mode, the file position may not correspond exactly to the byte offset in the file.

SEE: Clib.fsetpos(), Clib.ftell()

Clib.fsetpos()

SYNTAX: `Clib.fsetpos(filePointer, pos)`

WHERE: filePointer - pointer to file to use.
pos - position in file to set.

RETURN: number - zero on success, otherwise returns non-zero and stores an error value in `Clib.errno`.

DESCRIPTION: This method sets the current file stream pointer to the value defined by pos, which must be a value obtained from a previous call to `Clib.fgetpos()` on the same open file. Returns zero for success, otherwise returns non-zero and stores an error value in `Clib.errno`.

SEE: Clib.fseek()

Clib.ftell()

SYNTAX: `Clib.ftell(filePointer)`

WHERE: filePointer - pointer to file to use.

RETURN: number - current value of the file position indicator, or -1 if there is an error, in which case an error value will be stored in `Clib.errno`.

DESCRIPTION: This method sets the position offset of the file pointer of an open file stream from the beginning of the file. For text files, not opened in binary mode, the file position may not correspond exactly to the byte offset in the file. Returns the current value of the file position indicator, or -1 if there is an error, in which case an error value will be stored in `Clib.errno`.

SEE: Clib.fseek()

Clib.fwrite()

SYNTAX: `Clib.fwrite(srcVar, varDescription, filePointer)`

WHERE: srcVar - variable to hold data to write to file.
varDescription - description of the data to write, that is, how and how much.
filePointer - pointer to file to use.

RETURN: number - elements written on success, else 0 if a write error occurs.

DESCRIPTION: This method writes the data in srcVar to the file indicated by filePointer and returns the number of elements written. 0 will be returned if a write error occurs. Use `Clib.ferror()` to get more information about the error. varDescription is a variable that describes the how and how much data is to be read. If srcVar is a buffer, it will be the length of the buffer. If srcVar is an object, varDescription must be an object descriptor. If srcVar is to hold a single datum then varDescription must be one of the values listed in the description for `Clib.fread()`.
The Javascript version of `Clib.fwrite()` differs from the standard C version in that the standard C library is set up for writing arrays of numeric values or structures from consecutive bytes in memory. This is not necessarily the case in Javascript.

SEE: Clib.fread()

EXAMPLE:

```
// To write the 16_bit integer "i", the 32_bit float "f", and
// then 10_byte buffer "buf" into open file "fp", use this code:
if (!Clib.fwrite(i, SWORD16, fp) ||
    !Clib.fwrite(f, FLOAT32, fp) ||
    (10 !=  fwrite(buf, 10, fp)))  {
```

```
   writeLog("Error writing to file.");
}
```

Clib.getc()

SYNTAX:	`Clib.getc(filePointer)`
WHERE:	filePointer - pointer to file to use.
RETURN:	number - on success, the next character, as an unsigned byte converted to an integer, in a file. Else `EOF` if a read error or at the end of file.
DESCRIPTION:	This method is identical to `Clib.fgetc()`. It returns the next character in a file as an unsigned byte converted to an integer. Returns EOF if there is a read error or if at the end of the file. If there is a read error then `Clib.ferror()` will indicate the error condition.

Clib.putc()

SYNTAX:	`Clib.putc(chr, filePointer)`
WHERE:	chr - character to write to file. filePointer - pointer to file to use.
RETURN:	number - character written on success, else `EOF` on write error.
DESCRIPTION:	This method writes the character chr, converted to a byte, to an output file stream. This method is identical to `Clib.fputc()`. It returns chr on success and `EOF` on a write error.
SEE:	Clib.fputc()

Clib.remove()

SYNTAX:	`Clib.remove(filename)`
WHERE:	filename - the name of the file to delete from a disk.
RETURN:	number - 0 on success, else non-zero.
DESCRIPTION:	Delete a file with the filename provided.
SEE:	Clib.rename(), Clib.fopen()

Clib.rename()

SYNTAX:	`Clib.rename(oldFilename, newFilename)`
WHERE:	oldFilename - current name of file on disk to be renamed. newFilename - new name for file on disk.
RETURN:	number - 0 on success, else non-zero.
DESCRIPTION:	This method renames oldFilename to newFilename. Both oldFilename and newFilename are strings. Returns zero if successful and non-zero for failure.
SEE:	Clib.remove()

Clib.rewind()

SYNTAX:	`Clib.rewind(filePointer)`
WHERE:	filePointer - pointer to file to use.
RETURN:	void.
DESCRIPTION:	This method sets the file cursor to the beginning of file. This call is the same as `Clib.fseek(filePointer, 0, SEEK_SET)` except that it also clears the error indicator for this stream.
SEE:	Clib.fseek()

Clib.sscanf()

SYNTAX:	`Clib.sscanf(str, formatString[, variables ...])`
WHERE:	str - string holding the data to read into variables according to formatString. formatString - specifies how to read and store data in variables. variables - list of variables to hold data input according to formatString.
RETURN:	number - input items assigned. May be lower than the number of items requested if there is a matching failure. void.

DESCRIPTION: This flexible method reads data from a string and stores it in variables passed as parameters following formatString. The parameter formatString specifies how data is read and stored in variables.
The format string specifies the admissible input sequences, and how the input is to be converted to be assigned to the variable number of arguments passed to this function.

Characters are matched against the input as read and as it matches a portion of the format string until a % character is reached. % indicates that a value is to be read and stored to subsequent parameters following the format string. Each subsequent parameter after the format string gets the next parsed value takes from the next parameter in the list following format. A parameter specification takes this form (square brackets indicate optional fields, angled brackets indicate required fields):

%[*][width]<type>
*, width, and type may be:

- * : suppress assigning this value to any parameter
- width :maximum number of characters to read; fewer will be read if white space or nonconvertible character
- type : may be one of the following:
    - d, D, i, I : signed integer
    - u, U : unsigned integer
    - o, O : octal integer
    - x, X : hexadecimal integer
    - f, e, E, g, G : floating point number
    - c : character; if width was specified then this will be an array of characters of the specified length
    - s : string
    - [abc] : string consisting of all characters within brackets; where A-Z represents range "A" to "Z"
    - [^abc] : string consisting of all character NOT within brackets.

Modifies any number of parameters following the format string, setting the parameters to data according to the specifications of the format string.

SEE: Clib.sprintf()

Clib.sprintf()

SYNTAX: `Clib.sprintf(str, formatString[, variables ...])`

WHERE: str - to hold the formatted output
formatString - string that specifies the final format
variables - values to be converted to and formatted as a string.

RETURN: number - characters written to string on success, else EOF on failure. assigned. May be lower than the number of items requested if there is a matching failure. void.

DESCRIPTION: This method writes output to the string variable specified by str according to formatString, and returns the number of characters written or EOF if there was an error. The parameter formatString may contain character combinations indicating how following parameters are to be written. The parameter str need not be previously defined. It will be created large enough to hold the result.
The format string can contain character combinations indicating how following parameters are to be treated. Characters are printed as read to standard output until a percent character, %, is reached. % indicates that a value is to be printed from the parameters following the format string. Each subsequent parameter specification takes from the next parameter in the list following format. A parameter specification has the following form in which square brackets indicate optional fields and angled brackets indicate required fields:
%[flags][width][.precision]<type>
flags may be:

- - : Left justification in the field with blank padding; else right justifies with zero or blank padding

- + : Force numbers to begin with a plus (+) or minus (-)
- blank : Negative values begin with a minus (-); positive values begin with a blank
- # : Convert using the following alternate form, depending on output data type:
    - c, s, d, i, u : No effect
    - o : 0 (zero) is prepended to non-zero output
    - x, X : 0x, or 0X, are prepended to output
    - f, e, E : Output includes decimal even if no digits follow decimal
    - g, G : Same as e or E but trailing zeros are not removed

width may be:

- n : (n is a number e.g., 14) At least n characters are output, padded with blanks
- 0n : At least n characters are output, padded on the left with zeros
- * : The next value in the argument list is an integer specifying the output width
- .precision : If precision is specified, then it must begin with a period (.), and may be as follows:
    - 0 : For floating point type, no decimal point is output
    - n : n characters or n decimal places (floating point) are output
    - * : The next value in the argument list is an integer specifying the precision width

type may be:

- d, I : signed integer
- u : unsigned integer
- o : octal integer x
- x : hexadecimal integer with 0-9 and a, b, c, d, e, f
- X : hexadecimal integer with 0-9 and A, B, C, D, E, F
- f : floating point of the form [-]dddd.dddd
- e : floating point of the form [-]d.ddde+dd or [-]d.ddde-dd
- E : floating point of the form [-]d.dddE+dd or [-]d.dddE-dd
- g : floating point of f or e type, depending on precision
- G : floating point of For E type, depending on precision
- c : character (e.g. 'a', 'b', '8')
- s : string

To include the % character as a character in the format string, you must use two % characters together, %%, to prevent the computer from trying to interpret it as one of the above forms.

SEE: Clib.sscanf()

Clib.ungetc()

SYNTAX:	`Clib.ungetc(chr, filePointer)`
WHERE:	chr - character to write to file. filePointer - pointer to file to use.
RETURN:	number - on success, the character put back into a file stream, else EOF.
DESCRIPTION:	This method pushes character chr back into an input stream. When chr is put back, it is converted to a byte and is again in an input stream for subsequent retrieval. Only one character is guaranteed to be pushed back. The method returns chr on success, else EOF on failure.
SEE:	Clib.getc()

3.5 Date Object

To create a Date object which is set to the current date and time, use the new operator, as you would with any object.

```
var currentDate = new Date();
```

There are several ways to create a Date object which is set to a date and time. The following lines all demonstrate ways to get and set dates and times.

```
var aDate = new Date(milliseconds);
var bDate = new Date(datestring);
var cDate = new Date(year, month, day);
var dDate = new Date(year, month, day, hours, minutes, seconds);
```

The first syntax returns a date and time represented by the number of milliseconds since midnight, January 1, 1970. This representation in milliseconds is a standard way of representing dates and times that makes it easy to calculate the amount of time between one date and another. Generally, you do not create dates in this way. Instead, you convert them to milliseconds format before doing calculations.
The second syntax accepts a string representing a date and optional time. The format of such a datestring is:

```
month day, year hours:minutes:seconds
```

For example, the following string:

```
"Friday 13, 1995 13:13:15"
```

specifies the date, Friday 13, 1995, and the time, one thirteen and 15 seconds p.m., which, expressed in 24 hour time, is 13:13 hours and 15 seconds. The time specification is optional and if included, the seconds specification is optional.

The third and fourth syntaxes are self-explanatory. All parameters passed to them are integers.

- **year**
  If a year is in the twentieth century, the 1900s, you need only supply the final two digits. Otherwise four digits must be supplied.
- **month**
  A month is specified as a number from 0 to 11. January is 0, and December is 11.
- **day**
  A day of the month is specified as a number from 1 to 31. The first day of a month is 1 and the last is 28, 29, 30, or 31.
- **hours**
  An hour is specified as a number from 0 to 23. Midnight is 0, and 11 p.m. is 23.
- **minutes**
  A minute is specified as a number from 0 to 59. The first minute of an hour is 0, and the last is 59.
- **seconds**
  A second is specified as a number from 0 to 59. The first second of a minute is 0, and the last is 59.

For example, the following line of code:

```
var aDate = new Date(1492, 9, 12)
```

creates a Date object containing the date, October 12, 1492.

The following list of methods has brief descriptions of the methods of the Date object. Instance methods are shown with a period, ".", in the syntax: line. A specific instance of a variable should be put in front of the period to call a method. For example, the Date object aDate was created above, and, to call the `Date getDate()` method, the call would be: `aDate.getDate()`. Static methods have "`Date.`" at their beginnings since these methods are called with literal calls, such as `Date.parse()`. These methods are part of the Date object itself instead of instances of the Date object.

3.5.1 Date object instance methods

Date getDate()

SYNTAX:	`date.getDate()`
RETURN:	number - a day of a month.
DESCRIPTION:	This method returns the day of the month, as a number from 1 to 31, of a `Date object`. The first day of a month is 1, and the last is 28, 29, 30, or 31.

Date getDay()

SYNTAX:	`date.getDay()`
RETURN:	number - a day in a week.
DESCRIPTION:	This method returns the day of the week, as a number from 0 to 6, of a `Date object`. Sunday is 0, and Saturday is 6.

Date getFullYear()

SYNTAX:	`date.getFullYear()`
RETURN:	number - four digit year.
DESCRIPTION:	This method returns the year, as a number with four digits, of a `Date object`.

Date getHours()

SYNTAX:	`date.getHours()`
RETURN:	number - an hour in a day.
DESCRIPTION:	This method returns the hour, as a number from 0 to 23, of a `Date object`. Midnight is 0, and 11 p.m. is 23.

Date getMilliseconds()

SYNTAX:	`date.getMilliseconds()`
RETURN:	number - a millisecond in a second.
DESCRIPTION:	This method returns the millisecond, as a number from 0 to 999, of a `Date object`. The first millisecond in a second is 0, and the last is 999.

Date getMinutes()

SYNTAX:	`date.getMinutes()`
RETURN:	number - a minute in an hour.
DESCRIPTION:	This method returns the minute, as a number from 0 to 59, of a `Date object`. The first minute of an hour is 0, and the last is 59.

Date getMonth()

SYNTAX:	`date.getMonth()`
RETURN:	number - of a month in a year.
DESCRIPTION:	This method returns the month, as a number from 0 to 11, of a `Date object`.

January is 0, and December is 11.

Date getSeconds()

SYNTAX: `date.getSeconds()`
RETURN: number - a second in a minute.
DESCRIPTION: This method returns the second, as number from 0 to 59, of a `Date object`. The first second of a minute is 0, and the last is 59.

Date getTime()

SYNTAX: `date.getTime()`
RETURN: number - the milliseconds representation of a `Date object`.
DESCRIPTION: Gets time information in the form of an integer representing the number of seconds from midnight on January 1, 1970, GMT, to the date and time specified by a Date object.

Date getTimezoneOffset()

SYNTAX: `date.getTimezoneOffset()`
RETURN: number - minutes.
DESCRIPTION: This method returns the difference, in minutes, between Greenwich Mean Time (GMT) and local time.

Date getUTCDate()

SYNTAX: `date.getUTCDate()`
RETURN: number - a day of a month.
DESCRIPTION: This method returns the UTC day of the month, as a number from 1 to 31, of a `Date object`. The first day of a month is 1, and the last is 28, 29, 30, or 31.

Date getUTCDay()

SYNTAX: `date.getUTCDay()`
RETURN: number - a day in a week.
DESCRIPTION: This method returns the day of the week, as a number from 0 to 6, of a `Date object`. Sunday is 0, and Saturday is 6.

Date getUTCFullYear()

SYNTAX: `date.getUTCFullYear()`
RETURN: number - four digit year.
DESCRIPTION: This method returns the UTC year, as a number with four digits, of a `Date object`.

Date getUTCHours()

SYNTAX: `date.getUTCHours()`
RETURN: number - an hour in a day.
DESCRIPTION: This method returns the UTC hour, as a number from 0 to 23, of a `Date object`. Midnight is 0, and 11 p.m. is 23.

Date getUTCMilliseconds()

SYNTAX: `date.getUTCMilliseconds()`
RETURN: number - a millisecond in a second.
DESCRIPTION: This method returns the UTC millisecond, as a number from 0 to 999, of a `Date object`. The first millisecond in a second is 0, and the last is 999.

Date getUTCMinutes()

SYNTAX: `date.getUTCMinutes()`
RETURN: number - a minute in an hour.

DESCRIPTION:	This method returns the UTC minute, as a number from 0 to 59, of a `Date object`. The first minute of an hour is 0, and the last is 59.

Date getUTCMonth()

SYNTAX:	`date.getUTCMonth()`
RETURN:	number - of a month in a year.
DESCRIPTION:	number - of a month in a year.

Date getUTCSeconds()

SYNTAX:	`date.getUTCSeconds()`
RETURN:	number - a second in a minute.
DESCRIPTION:	This method returns the UTC second, as number from 0 to 59, of a `Date object`. The first second of a minute is 0, and the last is 59.

Date getYear()

SYNTAX:	`date.getYear()`
RETURN:	number - two digit year.
DESCRIPTION:	This method returns the year, as a number with two digits, of a `Date object`.

Date setDate()

SYNTAX:	`date.setDate(day)`
WHERE:	day - a day in a month.
RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the day, as a number from 1 to 31, of a `Date object` to the parameter day. The first day of a month is 1, and the last is 28, 29, 30, or 31.

Date setFullYear()

SYNTAX:	`date.setFullYear(year[, month[, date]])`
WHERE:	year - a four digit year. month - a month in a year. day - a day in a month.
RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the year of a `Date object` to the parameter year. The parameter year is expressed with four digits. The parameter month is the same as for `Date setMonth()`. The parameter day is the same as for `Date setDate()`.

Date setHours()

SYNTAX:	`Date.setHours(hour[, minute[, second[, millisecond]]])`
WHERE:	hour - an hour in a day. minute - a minute in an hour. second - a second in a minute. millisecond - a millisecond in a second.
RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the hour, as a number from 0 to 23, of a `Date object` to the parameter hours. Midnight is 0, and 11 p.m. is 23. The parameter minute is the same as for `Date setMinutes()`. The parameter second is the same as for `Date setSeconds()`. The parameter milliseconds is the same as for `Date setMilliseconds()`.

Date setMilliseconds()

SYNTAX:	`date.setMilliseconds(millisecond)`
WHERE:	millisecond - a millisecond in a minute.

RETURN: number - time in milliseconds as set.
DESCRIPTION: This method sets the millisecond, as a number from 0 to 59, of a `Date object` to the parameter millisecond. The first millisecond in a second is 0, and the last is 999.

Date setMinutes()

SYNTAX: `date.setMinutes(minute[, second[, millisecond]])`
WHERE: minute - a minute in an hour.
second - a second in a minute.
millisecond - a millisecond in a second.
RETURN: number - time in milliseconds.
DESCRIPTION: This method sets the minute, as a number from 0 to 59, of a `Date object` to the parameter minute. The first minute of an hour is 0, and the last is 59.
The parameter second is the same as for `Date setSeconds()`.
The parameter milliseconds is the same as for `Date setMilliseconds()`.

Date setMonth()

SYNTAX: `Date.setMonth(month[, day])`
WHERE: month - a month in a year.
day - a day in a month.
RETURN: number - time in milliseconds.
DESCRIPTION: This method sets the month, as a number from 0 to 11, of a `Date object` to the parameter month. January is 0, and December is 11.
The parameter day is the same as for `Date setDate()`.

Date setSeconds()

SYNTAX: `date.setSeconds(second[, millisecond])`
WHERE: second - a second in a minute.
millisecond - a millisecond in a second.
RETURN: number - time in milliseconds.
DESCRIPTION: This method sets the second, as a number from 0 to 59, of a `Date object` to the parameter second. The first second of a minute is 0, and the last is 59.
The parameter milliseconds is the same as for `Date setMilliseconds()`.

Date setTime()

SYNTAX: `date.setTime(millisecond)`
WHERE: millisecond - the time in milliseconds.
RETURN: number - time in milliseconds as set.
DESCRIPTION: This method sets a `Date object` to the date and time specified by the parameter milliseconds which is the number of milliseconds from midnight on January 1, 1970, GMT.

Date setUTCDate()

SYNTAX: `date.setUTCDate(day)`
WHERE: day - a day in a month.
RETURN: number - time in milliseconds as set.
DESCRIPTION: This method sets the UTC day, as a number from 1 to 31, of a `Date object` to the parameter day. The first day of a month is 1, and the last is 28, 29, 30, or 31.

Date setUTCFullYear()

SYNTAX: `date.setUTCFullYear(year[, month[, date]])`
WHERE: year - a four digit year.
month - a month in a year.
day - a day in a month.
RETURN: number - time in milliseconds as set.
DESCRIPTION: This method sets the UTC year of a `Date object` to the parameter year. The

	parameter year is expressed with four digits. The parameter month is the same as for `Date setUTCMonth()`. The parameter day is the same as for `Date setUTCDate()`.

Date setUTCHours()

SYNTAX:	`Date.setUTCHours(hour[, minute[, second[, millisecond]]])`
WHERE:	hour - an hour in a day. minute - a minute in an hour. second - a second in a minute. millisecond - a millisecond in a second.
RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the UTC hour, as a number from 0 to 23, of a `Date object` to the parameter hours. Midnight is 0, and 11 p.m. is 23. The parameter minute is the same as for `Date setUTCMinutes()`. The parameter second is the same as for `Date setUTCSeconds()`. The parameter milliseconds is the same as for `Date setUTCMilliseconds()`.

Date setUTCMilliseconds()

SYNTAX:	`date.setUTCMilliseconds(millisecond)`
WHERE:	millisecond - a millisecond in a minute.
RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the UTC millisecond, as a number from 0 to 59, of a `Date object` to the parameter millisecond. The first millisecond in a second is 0, and the last is 999.

Date setUTCMinutes()

SYNTAX:	`date.setUTCMinutes(minute[, second[, millisecond]])`
WHERE:	minute - a minute in an hour. second - a second in a minute. millisecond - a millisecond in a second.
RETURN:	number - time in milliseconds.
DESCRIPTION:	This method sets the UTC minute, as a number from 0 to 59, of a `Date object` to the parameter minute. The first minute of an hour is 0, and the last is 59. The parameter second is the same as for `Date setUTCSeconds()`. The parameter milliseconds is the same as for `Date setUTCMilliseconds()`.

Date setUTCMonth()

SYNTAX:	`Date.setUTCMonth(month[, day])`
WHERE:	month - a month in a year. day - a day in a month.
RETURN:	number - time in milliseconds.
DESCRIPTION:	This method sets the UTC month, as a number from 0 to 11, of a `Date object` to the parameter month. January is 0, and December is 11. The parameter day is the same as for `Date setUTCDate()`.

Date setUTCSeconds()

SYNTAX:	`date.setUTCSeconds(second[, millisecond])`
WHERE:	second - a second in a minute. millisecond - a millisecond in a second.
RETURN:	number - time in milliseconds.
DESCRIPTION:	This method sets the UTC second, as a number from 0 to 59, of a `Date object` to the parameter second. The first second of a minute is 0, and the last is 59. The parameter milliseconds is the same as for `Date setUTCMilliseconds()`.

Date setYear()

SYNTAX:	`date.setYear(year)`
WHERE:	year - four digit year, unless in the 1900s in which case it may be a two digit year.
RETURN:	number - time in milliseconds as set.
DESCRIPTION:	This method sets the year of a `Date object` to the parameter year. The parameter year may be expressed with two digits for a year in the twentieth century, the 1900s. Four digits are necessary for any other century.

Date toDateString()

SYNTAX:	`date.toDateString()`
RETURN:	string - representation of the date portion of the current object.
DESCRIPTION:	Returns the Date portion of the current date as a string. This string is formatted to read "Month Day, Year", for example, "May 1, 2000". This method uses the local time, not UTC time.
SEE:	Date toString(), Date toTimeString(), Date toLocaleDateString()

EXAMPLE:	`var d = new Date();`
	`var s = d.toDateString();`

Date toGMTString()

SYNTAX:	`date.toGMTString()`
RETURN:	string - string representation of the GMT date and time.
DESCRIPTION:	This method converts a `Date object` to a string, based on Greenwich Mean Time.

EXAMPLE:	`var d = new Date();`
	`writeLog(d.toGMTString());`
	
	`// The fragment above would produce something like:`
	`// Mon May 1 15:48:38 2000 GMT`

Date toLocaleDateString()

SYNTAX:	`date.toLocaleDateString()`
RETURN:	string - locale-sensitive string representation of the date portion of the current date.
DESCRIPTION:	This function behaves in exactly the same manner as `Date toDateString()`. This function is designed to take in the current locale when formatting the string. Locale reflects the time zone of a user.
SEE:	Date toString(), Date toLocaleTimeString(), Date toLocaleString()

EXAMPLE:	`var d = new Date();`
	`var s = d.toLocaleDateString();`

Date toLocaleString()

SYNTAX:	`date.toLocaleString()`
RETURN:	string - locale-sensitive string representation of the current date.
DESCRIPTION:	This function behaves in exactly the same manner as `Date toString()`. This function is designed to take in the current locale when formatting the string, though this functionality is currently unimplemented. Locale reflects the time zone of a user.
SEE:	Date toString(), Date toLocaleTimeString(), Date toLocaleDateString()

EXAMPLE:	`var d = new Date();`
	`var s = d.toLocaleString();`

Date toLocaleTimeString()

SYNTAX:	`date.toLocaleTimeString()`
RETURN:	string - locale-sensitive string representation of the time portion of the current date.
DESCRIPTION:	This function behaves in exactly the same manner as `Date toTimeString()`. This function is designed to take in the current locale when formatting the string. Locale reflects the time zone of a user.

Date toString()

SYNTAX:	`date.toString()`
RETURN:	string - representation of the date and time data in a `Date object`.
DESCRIPTION:	Converts the date and time information in a Date object to a string in a form such as: "Mon May 1 09:24:38 2000"
SEE:	Date toDateString(), Date toLocaleString(), Date toTimeString()

EXAMPLE:	`var d = new Date();`
	`var s = d.toString();`

Date toSystem()

SYNTAX:	`date.toSystem()`
RETURN:	number - the `Date object` date and time value converted to the system date and time.
DESCRIPTION:	This method converts a Date object to a system time format which is the same as that returned in the `timestamp` structure. To create a Date object from a variable in system time format, see the `Date.fromSystem()` method.

Date toTimeString()

SYNTAX:	`date.toTimeString()`
RETURN:	string - representation of the Time portion of the current object.
DESCRIPTION:	This function returns the time portion of the current date as a string. This string is formatted to read "Hours:Minutes:Seconds", as in "16:43:23". This function uses the local time, rather than the UTC time.
SEE:	Date toString(), Date toDateString(), Date toLocaleDateString()

EXAMPLE:	`var d = new Date();`
	`var s = d.toTimeString();`

Date toUTCString()

SYNTAX:	`date.toUTCString()`
RETURN:	string - representation of the UTC date and time data in a `Date object`.
DESCRIPTION:	Converts the UTC date and time information in a Date object to a string in a form such as: "Mon May 1 09:24:38 2000"
SEE:	Date toDateString(), Date toLocaleString(), Date toTimeString()

EXAMPLE:	`var d = new Date();`
	`var s = d.toString();`

Date valueOf()

SYNTAX:	`date.valueOf()`
RETURN:	number - the value of the date and time information in a Date object.
DESCRIPTION:	The numeric representation of a `Date object`.
SEE:	Date toString()

3.5.2 Date object static methods

The `Date object` has three special methods that are called from the object itself, rather than from an instance of it: `Date.fromSystem()`, `Date.parse()`, and `Date.UTC()`.

Date.fromSystem()

SYNTAX:	`Date.fromSystem(time)`
WHERE:	time - time in system data format in the format as returned in `sec` of `timestamp`
RETURN:	object - Date object with the time passed
DESCRIPTION:	This method converts the parameter time, which is in the same format as returned in `timestamp`, to a standard Javascript `Date object`.
EXAMPLE:	`// To create a Date object`

```
// from date information obtained using getBufferDataElement()
// use code similar to:

var SysDate = getBufferDataElement(bufferID, DT_RS, "timestamp");
var ObjDate = Date.fromSystem(SysDate.sec);

// To convert a Date object to system format
// use code similar to:

SysDate.sec = ObjDate.toSystem();
```

Date.parse()

SYNTAX:	`Date.parse(dateString)`
WHERE:	dateString - A string representing the date and time to be passed
RETURN:	number - milliseconds between the datestring and midnight , January 1, 1970 GMT.
DESCRIPTION:	This method converts the string dateString to a `Date object`. The string must be in the following format: `Friday, October 31, 1998 15:30:00 -0500` This format is used by the `Date toGMTString()` method and by email and Internet applications. The day of the week, time zone, time specification or seconds field may be omitted.
SEE:	Date object, Date setTime(), Date toGMTString(), Date.UTC()

EXAMPLE:

```
//The following code sets the date to March 2, 1992
var theDate = Date.parse("March 2, 1992")
//Note:
var theDate = Date.parse(datestring);
//is equivalent to:
var theDate = new Date(datestring);
```

Date.UTC()

SYNTAX:

```
Date.UTC(year, month, day[, hours[, minutes[,
          seconds[, milliseconds]]]])
```

WHERE:	year - A year, represented in four or two-digit format after 1900. NOTE: For year 2000 compliance, this year MUST be represented in four-digit format month - A number between 0 (January) and 11 (December) representing the month day - A number between 1 and 31 representing the day of the month. Note that `Month` uses 1 as its lowest value whereas many other arguments use 0 hours - A number between 0 (midnight) and 23 (11 PM) representing the hours minutes - A number between 0 (one minute) and 59 (59 minutes) representing the minutes. This is an optional argument which may be omitted if `Seconds` and `Minutes` are omitted as well. seconds - A number between 0 and 59 representing the seconds. This parameter is optional. milliseconds - A number between 0 and 999 which represents the milliseconds. This is an optional parameter.
RETURN:	number - milliseconds from midnight, January 1, 1970, to the date and time specified.
DESCRIPTION:	The method interprets its parameters as a date. The parameters are interpreted as referring to Greenwich Mean Time (GMT).
SEE:	Date object, Date.parse(), Date setTime()

EXAMPLE:

```
// The following code creates a Date object
// using UTC time:
foo = new Date(Date.UTC(1998, 3, 9, 1, 0, 0, 8))
```

3.6 Function Object

The Function object is one of three ways to define and use objects in Javascript. The three ways to work with objects are:

- Use the function keyword and define a function in a normal way:
  function `myFunc(x) {return x + 4;}`
- Construct a new Function object:
  `var myFunc = new Function("x", "return x + 4;");`
- Define and assign a function literal:
  `var myFunc = function(x) {return x + 4;}`

All three of three of these ways of defining and using functions produce the same result, x + 4. The differences are in definition and use of functions. Each way has a strength that is very powerful in some circumstances, power that allows elegance in programming. The methods and discussion in this segment on the Function object deal with the second way shown above, the construction of a new Function object.

3.6.1 Function object instance methods

Function()

SYNTAX:	`new Function(params[, ...], body)`
WHERE:	params - one or a list of parameters for the function. body - the body of the function as a string.
RETURN:	object - a new function object with the specified parameters and body that can later be executed just like any other function.
DESCRIPTION:	The parameters passed to the function can be in one of two formats. All parameters are strings representing parameter names, although multiple parameter names can be grouped together with commas. These two options can be combined as well. For example, `new Function("a", "b", "c", "return")` is the same as `new Function("a, b", "c", "return")`. The body of the function is parsed just as any other function would be. If there is an error parsing either the parameter list or the function body, a runtime error is generated. If this function is later called as a constructor, then a new object is created whose internal `_prototype` property is equal to the `prototype` property of the new function object. Note that this function can also be called directly, without the *new* operator.

EXAMPLE:

```
// The following will create a new Function object
// and provide some properties
// through the prototype property.

var myFunction = new Function("a", "b",
    "this.value = a + b");
var printFunction = new Function
    ("writeLog(this.value)");
myFunction.prototype.print = printFunction;

var foo = new myFunction( 4, 5 );
foo.print();

// This code will print out the value "9", which was the value stored
// in foo when it was created with the myFunction constructor.
```

Function apply()

SYNTAX:	`function.apply([thisObj[, arguments])`
WHERE:	thisObj - object that will be used as the "this" variable while calling this function. If this is not supplied, then the global object is used instead.

arguments - array of arguments to pass to the function as an Array object or a list in the form of [arg1, arg2[, ...]]. The brackets "[]" around a list of arguments are required. Note that the similar method `Function call()` can receive the same arguments as a list. Compare the following ways of passing arguments:

```
 // Uses an Array object
 function.apply(this, argArray)
 // Uses brackets
 function.apply(this,[arg1,arg2])
 // Uses argument list
 function.call(this,arg1,arg2)
```

RETURN: variable - the result of calling the function object with the specified "this" variable and arguments.

DESCRIPTION: This method is similar to calling the function directly, only the user is able to pass a variable to use as the "this" variable, and the arguments to the function are passed as an array. If `arguments` is not supplied, then no arguments are passed to the function. If the `arguments` parameter is not a valid Array object or list of arguments inside of brackets "[]", then a runtime error is generated.

SEE: Function(), Function call()

EXAMPLE:

```
var myFunction = new Function("a,b","return a + b");
var args = new Array(4,5);
myFunction.apply(global, args);
//or
myFunction.apply(global, [4,5]);

// This code sample will return 9, which is
// the result of calling myFunction with
// the arguments 4 and 5, from the args array.
```

Function call()

SYNTAX: `function.call([thisObj[, arguments[, ...]]])`

WHERE: thisObj - An object that will be used as the "this" variable while calling this function. If this is not supplied, then the global object is used instead.

arguments - list of arguments to pass to the function. Note that the similar method `Function apply()` can receive the same arguments as an array. Compare the following ways of passing arguments:

```
    // Uses an Array object
 function.apply(this, argArray)
    // Uses brackets
 function.apply(this,[arg1,arg2])
    // Uses argument list
 function.call(this,arg1,arg2)
```

RETURN: variable - the result of calling the function object with the specified "this" variable and arguments.

DESCRIPTION: This method is almost identical to calling the function directly, only the user is able to supply the "this" variable that the function will use. Otherwise, it is the same.

SEE: Function(), Function.apply()

EXAMPLE:

```
// The following code:

var myFunction = new Function("arg", "return this.a + arg");
var obj = { a:4 };
myFunction( obj, 5 );

// This code fragment returns the value 9,
// which is the result of fetching this.a//
// from the current object (which is obj) and
// adding the first parameter passed, which is 5.
```

Function toString()

SYNTAX: `function.toString()`

RETURN: string - a representation of the function.

DESCRIPTION: This method attempts to generate the same code that built the function. Any spacing, semicolons, newlines, etc., are implementation-dependent. This method tries to make the output as human-readable as possible. Note that the function name is always "anonymous", because the function itself is unnamed, even though the function object has a name. Also, note that this function is very rarely called directly, rather it is called implicitly through conversions such as `global.ToString()`.

EXAMPLE:

```
var myFunction =  new Function("a", "b", "this.value = a + b");
writeLog( myFunction );

// This fragment will print the followingto the screen:

function anonymous(a, b)
{
  this .value = a + b;
}
```

3.7 Global Object

The properties and methods of the `global` object may be thought of as global variables and functions. The object identifier `global` is not required when invoking a `global` method or function. Indeed, the object name generally is not used. For example, the following two `if` statements are identical, but the first one illustrates how `global` functions are usually invoked.

```
if (defined(name))
  writeLog("name is defined");

if (global.defined(name))
   writeLog("name is defined");
```

The following two lines of code are also equivalent.

```
var aString = ToString(123)
var aString = global.ToString(123)
```

Remember, global variables are members of the global object. To access global properties, you do not need to use an object name. The exception to this rule occurs when you are in a function that has a local variable with the same name as a global variable. In such a case, you must use the global keyword to reference the global variable.

3.7.1 Conversion or casting

Though Javascript does well in automatic data conversion, there are times when the types of variables or data must be specified and controlled. Each of the following casting functions, the functions below that begin with "To", has one parameter, which is a variable or piece of data, to be converted to or cast as the data type specified in the name of the function. For example, the following fragment creates two variables.

```
var aString = ToString(123);
var aNumber = ToNumber("123");
```

The first variable aString is created as a string from the number 123 converted to or cast as a string. The second variable aNumber is created as a number from the string "123" converted to or cast as a number. Since aString had already been created with the value "123", the second line could also have been:

```
var aNumber = ToNumber(aString);
```

The type of the variable or piece of data passed as a parameter affects the returns of some of these functions.

3.7.2 global object methods/functions

global.defined()

SYNTAX:	`defined(value)`
WHERE:	value - a value or variable to check to see if it is defined.
RETURN:	boolean - `true` if the value has been defined, else `false`
DESCRIPTION:	This function tests whether a variable, object property, or value has been defined. The function returns `true` if a value has been defined, or else returns `false`. The function `defined()` may be used during script execution and during preprocessing. When used in preprocessing with the directive `#if`, the function `defined()` is similar to the directive `#ifdef`, but is more powerful. The following fragment illustrates three uses of `defined()`.
SEE:	global.undefine()

EXAMPLE:

```
var t = 1;
#if defined(_WIN32_)
   writeLog("in Win32");
   if (defined(t))
      writeLog("t is defined");
   if (!defined(t.t))
      writeLog("t.t is not defined");
#endif

// The first use of defined() checks whether a value
// is available to the preprocessor
// to determine which platform is running the script.
// The second use checks a variable "t".
// The third use checks an object "t.t"
```

global.escape()

SYNTAX:	`escape(str)`
WHERE:	str - with special characters that need to be handled specially, that is, escaped.
RETURN:	string - with special characters escaped or fixed so that the string may be used in special ways, such as being a URL.
DESCRIPTION:	The `escape()` method receives a string and escapes the special characters so that the string may be used with a URL. This escaping conversion may be called encoding. All uppercase and lowercase letters, numbers, and the special symbols, @ * + - . /, remain in the string. All other characters are replaced by their respective unicode sequence, a hexadecimal escape sequence. This method is the reverse of `global.unescape()`.
SEE:	global.unescape()

EXAMPLE:

```
escape("Hello there!");
// Returns "Hello%20there%21"
```

global.eval()

SYNTAX:	`eval(expression)`
WHERE:	expression - a valid expression to be parsed and treated as if it were code or script.
RETURN:	value - the result of the evaluation of expression as code.
DESCRIPTION:	Evaluates whatever is represented by the parameter expression. If expression is not

a string, it will be returned. For example, calling eval(5) returns the value 5. If expression is a string, the interpreter tries to interpret the string as if it were Javascript code. If successful, the method returns the last variable with which was working, for example, the return variable. If the method is not successful, it returns the special value, `undefined`.

EXAMPLE:

```
var a = "who";
   // Displays the string as is
writeLog('a == "who"');
   // Evaluates the contents of the string as code,
   // and displays "true",
   // the result of the evaluation
writeLog(eval('a == "who"'));
```

---

global.isFinite()

SYNTAX:	`isFinite(number)`
WHERE:	number - to check if it is a finite number.
RETURN:	boolean - if the parameter is or can be converted to a number, else `false`.
DESCRIPTION:	This method returns `true` if the parameter, number, is or can be converted to a number. If the parameter evaluates as `NaN`, `Number.POSITIVE_INFINITY`, or `Number.NEGATIVE_INFINITY`, the method returns `false`.
SEE:	global.isNaN()

EXAMPLE:

```
if (isFinite(99)) writeLog("A number");
```

global.isNaN()

SYNTAX:	`isNaN(number)`
WHERE:	number - a value to if it is not a number.
RETURN:	boolean - `true` if number is not a number, else `false`.
DESCRIPTION:	This method returns `true` if the parameter, number, evaluates to `NaN`, "Not a Number". Otherwise it returns `false`.
SEE:	global.isFinite()

EXAMPLE:

```
if (isNan(99)) writeLog("Not a number");
```

global.getArrayLength()

SYNTAX:	`getArrayLength(array[, minIndex])`
WHERE:	array - an automatic array. minIndex - the minimum index to use.
RETURN:	number - the length of an array.
DESCRIPTION:	This function should be used with dynamically created arrays, that is, with arrays that were **not** created using the `new Array()` operator and constructor. When working with arrays created using the `new Array()` operator and constructor, use the `length` property of the Array object. The `length` property is not available for dynamically created arrays which must use the functions, `global.getArrayLength()` and `global.setArrayLength()`, when working with array lengths. The `getArrayLength()` function returns the length of a dynamic array, which is one more than the highest index of an array, if the first element of the array is at index 0, which is most common. If the parameter minIndex is passed, then it is used to set to the minimum index, which will be zero or less. You can use this function to get the length of an array that was not created with the `Array()` constructor function. This function and its counterpart, `setArrayLength()`, are intended for use with dynamically created arrays, that is, arrays not created with the `Array()` constructor function. Use the `Array length` property to get the length of arrays created with the constructor function and not `getArrayLength()`.
SEE:	global.setArrayLength(), Array length

EXAMPLE:

```
var arr = {4,5,6,7};
writeLog(getArrayLength(arr));
```

global.getAttributes()

SYNTAX:	`getAttributes(variable)`
WHERE:	variable - a variable identifier, name.
RETURN:	number - representing the attributes set for a variable. If no attributes are set, the return is 0. See `global.setAttributes()` for a list of predefined constants for the attributes that a variable may have.
DESCRIPTION:	Gets and returns the variable attributes for the parameter variable. Variable attributes may be set using the function `setAttributes()`. See `global.setAttributes()` for more information and descriptions of the attributes of variables that can be set.
SEE:	global.setAttributes()

global.parseFloat()

SYNTAX:	`parseFloat(str)`
WHERE:	str - to be converted to a decimal float.
RETURN:	number - the float to which the string converts, else `NaN`.
DESCRIPTION:	This method is similar to `global.parseInt()` except that it reads decimal numbers with fractional parts. In other words, the first period, ".", in the parameter string is considered to be a decimal point, and any following digits are the fractional part of the number. The method `parseFloat()` does not take a second parameter.
SEE:	global.parseInt()
EXAMPLE:	`var i = parseInt("9.3");`

global.parseInt()

SYNTAX:	`parseInt(str[, radix])`
WHERE:	str - to be converted to an integer. radix - the number base to use, default is 10.
RETURN:	number - the integer to which string converts, else `NaN`.
DESCRIPTION:	This method converts an alphanumeric string to an integer number. The first parameter, str, is the string to be converted, and the second parameter, radix, is an optional number indicating which base to use for the number. If the radix parameter is not supplied, the method defaults to base 10, which is decimal. If the first digit of string is a zero, radix defaults to base 8, which is octal. If the first digit is zero followed by an "x", that is, "0x", radix defaults to base 16, which is hexadecimal. White space characters at the beginning of the string are ignored. The first non-white space character must be either a digit or a minus sign (-). All numeric characters following the string will be read, up to the first non-numeric character, and the result will be converted into a number, expressed in the base specified by the radix variable. All characters including and following the first non-numeric character are ignored. If the string is unable to be converted to a number, the special value `NaN` is returned.
SEE:	global.parseFloat()

EXAMPLE:	`var i = parseInt("9");`
	`var i = parseInt("9.3");`
	`// In both cases, i == 9`

global.setArrayLength()

SYNTAX:	`setArrayLength(array[, minIndex[, length]])`
WHERE:	array - an automatic array. minIndex - the minimum index to use. Default is 0. length - the length of the array to set.
RETURN:	void.
DESCRIPTION:	This function sets the first index and length of an array. Any elements outside the bounds set by MinIndex and length are lost, that is, become `undefined`. If only two arguments are passed to `setArrayLength()`, the second argument is length and the minimum index of the newly sized array is 0. If three arguments are passed to

`setArrayLength()`, the second argument, which must be 0 or less, is the minimum index of the newly sized array, and the third argument is the length.

SEE: global.getArrayLength(), Array length

EXAMPLE:

```
var arr = {4,5,6,7};
writeLog(getArrayLength(arr));
setArrayLength(arr, 9);
```

---

global.setAttributes()

SYNTAX: `setAttributes(variable, attributes)`

WHERE: variable - a variable identifier, name.
attributes - the attribute or attributes to be set for a variable. If more than one attribute is being set, use the or operator, "`|`", to combine them.

RETURN: void.

DESCRIPTION: This function sets the variable attributes for the parameter variable using the parameter attributes. Variables in Javascript may have various attributes set that affect the behavior of variables. This function has no return.
The following list describes the attributes that may be set for variables. Multiple attributes may be set for variables by combining them with the or operator. For example, the flag setting `READ_ONLY | DONT_ENUM` sets both of these attributes for one variable.

- `DONT_DELETE`
  This variable may not be deleted. If the delete operator is used with a variable, nothing is done.
- `DONT_ENUM`
  This variable is not enumerated when using a for/in loop.
- `IMPLICIT_PARENTS`
  This attribute applies only to local functions and allows a scope chain to be altered based on the __parent__ property of the "this" variable. If this flag is set, if the __parent__ property is present, and if a variable is not found in the local variable context, activation object, of a function, then the parents of the "this" variable are searched backwards before searching the global object. The example below illustrates the effect of this flag.
- `IMPLICIT_THIS`
  This attribute applies only to local functions. If this flag is set, then the "this" variable is inserted into a scope chain before the activation object. For example, if variable TestVar is not found in a local variable context, activation object, the interpreter searches the current "this" variable of a function.
- `READ_ONLY`
  This variable is read-only. Any attempt to write to or change this variable fails.

SEE: global.getAttributes()

EXAMPLE:

```
// The following fragment illustrates the use
// of setAttributes() and the behavior affected
// by the IMPLICIT_PARENTS flag.
function foo()
{
   value = 5;
}
setAttributes(foo, IMPLICIT_PARENTS)

var a;
a.value = 4;
var b;
b.__parent__ = a;
b.foo = foo;
b.foo();

// After this code is run, a.value is set to 5.
```

global.ToBoolean()

SYNTAX: `ToBoolean(value)`
WHERE: value - to be cast as a boolean.
RETURN: boolean - conversion of value.
DESCRIPTION: The following list indicates how different data types are converted by this function.

- `Boolean`
  same as value
- `Buffer`
  same as for String
- `null`
  `false`
- `Number`
  `false`, if value is 0, +0, -0 or `NaN`, else `true`
- `Object`
  `true`
- `String`
  `false` if empty string, "", else `true`
- `undefined`
  `false`

global.ToBuffer()

SYNTAX: `ToBuffer(value)`
WHERE: value - to be cast as a buffer.
RETURN: buffer - conversion of value.
DESCRIPTION: This function converts value to a buffer in a manner similar to `global.ToString()` except that the resulting array of characters is a sequence of ASCII bytes and not a unicode string.
SEE: global.ToBytes()

global.ToBytes()

SYNTAX: `ToBytes(value)`
WHERE: value - to be cast as a buffer.
RETURN: buffer - conversion of value.
DESCRIPTION: This function converts value to a buffer and differs from `global.ToBuffer()` in that the conversion is actually a raw transfer of data to a buffer. The raw transfer does not convert unicode values to corresponding ASCII values. For example, the unicode string `"Hit"` is stored in a buffer as `"\0H\0\i\0t"`, that is, as the hexadecimal sequence: 00 48 00 69 00 74.
SEE: global.ToBuffer()

global.ToInt32()

SYNTAX: `ToInt32(value)`
WHERE: value - to be cast as a signed 32-bit integer.
RETURN: number - conversion of value.
DESCRIPTION: This function is the same as `global.ToInteger()` except that if the return is an integer, it is in the range of -2^{31} through $2^{31} - 1$.
SEE: global.ToInteger(), global.ToNumber()

global.ToInteger()

SYNTAX: `ToInteger(value)`
WHERE: value - to be cast as an integer.
RETURN: number - conversion of value.
DESCRIPTION: This function converts value to an integer type. First, call `global.ToNumber()`. If result is `NaN`, return +0. If result is +0, -0, +Infinity or -Infinity, return result. Else

	return floor(abs(result)) with the appropriate sign. For example, the value -4.8 is converted to -4.
SEE:	global.ToInt32(), global.ToNumber()

global.ToNumber()

SYNTAX:	`ToNumber(value)`
WHERE:	value - to be cast as a number.
RETURN:	number - conversion of value.
DESCRIPTION:	The following table lists how different data types are converted by this function. · `Boolean` +0, if value is `false`, else 1 · `Buffer` same as for String · `null` +0 · `Number` same as value · `Object` first, call ToPrimitive(), then call ToNumber() and return result · `String` number, if successful, else `NaN` · `undefined` `NaN`
SEE:	global.ToInteger(), global.ToInt32()

global.ToObject()

SYNTAX:	`ToObject(value)`
WHERE:	value - to be cast as an object.
RETURN:	object - conversion of value.
DESCRIPTION:	The following table lists how different data types are converted by this function. · `Boolean` new Boolean object with value · `null` generate runtime error · `Number` new Number object with value · `Object` same as parameter · `String` new String object with value · `undefined` generate runtime error
SEE:	global.ToPrimitive()

global.ToPrimitive()

SYNTAX:	`ToPrimitive(value)`
WHERE:	value - to be cast as a primitive.
RETURN:	value - conversion of value to one of the primitive data types.
DESCRIPTION:	This function does conversions only for parameters of type Object. An internal default value of the Object is returned.
SEE:	global.ToObject()

global.ToString()

SYNTAX: `ToString(value)`

WHERE: value - to be cast ass a string.

RETURN: string - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by is this function.

- `Boolean`
  "false", if value is `false`, else "true"
- `null`
  "null"
- `Number`
  if value is `NaN`, return "NaN". If +0 or -0, return "0". If Infinity, return "Infinity". If a number, return a string representing the number. If a number is negative, return "-" concatenated with the string representation of the number.
- `Object`
  first, call ToPrimitive(), then call ToString() and return result
- `String`
  same as value
- `undefined`
  "undefined"

SEE: global.ToPrimitive(), global.ToNumber()

global.ToUint16()

SYNTAX: `ToUint16(value)`

WHERE: value - to be cast as a 16 bit unsigned integer.

RETURN: number - conversion of value.

DESCRIPTION: This function is the same as `global.ToInteger()` except that if the return is an integer, it is in the range of 0 through 2^{16} - 1.

SEE: global.ToUint32(), global.ToInteger()

global.ToUint32()

SYNTAX: `ToUint32(value)`

WHERE: value - to be cast as a 32 bit unsigned integer.

RETURN: number - conversion of value.

DESCRIPTION: This function is the same as `global.ToInteger()` except that if the return is an integer, it is in the range of 232 - 1.

SEE: global.ToInt32(), global.ToInteger()

global.unescape()

SYNTAX: `unescape(str)`

WHERE: str - holding escape characters.

RETURN: string - with escape characters replaced by appropriate characters.

DESCRIPTION: This method is the reverse of the `global.escape()` method and removes escape sequences from a string and replaces them with the relevant characters. That is, an encoded string is decoded.

SEE: global.escape()

EXAMPLE: `unescape("Hello%20there%21"); // Returns "Hello there!"`

global.undefine()

SYNTAX: `undefine(value)`

WHERE: value - value, variable, or property to be `undefined`.

RETURN: void.

DESCRIPTION: This function undefines a variable, Object property, or value. If a value was previously defined so that its use with the function `global.defined()` returns `true`, then after using `undefine()` with the value, `defined()` returns `false`. Undefining a value is different than setting a value to `null`.

SEE: global.defined()

EXAMPLE:

```
// In the following fragment, the variable n
// is defined with the number value of 2 and
// then undefined.
var n = 2;
undefine(n);

// In the following fragment an object o is created and a
// property o.one is defined. The property is then undefined but
// the object o remains defined.
var o = new Object;
o.one = 1;
undefine(o.one);
```

3.8 Math Object

The methods in this section are preceded with the Object name Math, since individual instances of the Math Object are not created. For example, `Math.abs()` is the syntax to use to get the absolute value of a number.

3.8.1 Math object static properties

Math.E

SYNTAX: `Math.E`

DESCRIPTION: The number value for e, the base of natural logarithms. This value is represented internally as approximately 2.7182818284590452354.

EXAMPLE: `var n = Math.E;`

Math.LN10

SYNTAX: `Math.LN10`

DESCRIPTION: The number value for the natural logarithm of 10. This value is represented internally as approximately 2.302585092994046.

EXAMPLE: `var n = Math.LN10;`

Math.LN2

SYNTAX: `Math.LN2`

DESCRIPTION: The number value for the natural logarithm of 2. This value is represented internally as approximately 0.6931471805599453.

EXAMPLE: `var n = Math.LN2;`

Math.LOG2E

SYNTAX: `Math.LOG2E`

DESCRIPTION: The number value for the base 2 logarithm of e, the base of the natural logarithms. This value is represented internally as approximately 1.4426950408889634. The value of Math.LOG2E is approximately the reciprocal of the value of `Math.LN2`.

EXAMPLE: `var n = Math.LOG2E;`

Math.LOG10E

SYNTAX: `Math.LOG10E`

DESCRIPTION: The number value for the base 10 logarithm of e, the base of the natural logarithms. This value is represented internally as approximately 0.4342944819032518. The value of `Math.LOG10E` is approximately the reciprocal of the value of `Math.LN10`

EXAMPLE: `var n = Math.LOG10E`

Math.PI

SYNTAX:	`Math.PI`
DESCRIPTION:	The number value for pi, the ratio of the circumference of a circle to its diameter. This value is represented internally as approximately 3.14159265358979323846.
EXAMPLE:	`var n = Math.PI;`

Math.SQRT1_2

SYNTAX:	`Math.SQRT1_2`
DESCRIPTION:	The number value for the square root of 2, which is represented internally as approximately 0.7071067811865476. The value of `Math.SQRT1_2` is approximately the reciprocal of the value of `Math.SQRT2`.
EXAMPLE:	`var n = Math.SQRT1_2;`

Math.SQRT2

SYNTAX:	`Math.SQRT2`
DESCRIPTION:	The number value for the square root of 2, which is represented internally as approximately 1.4142135623730951.
EXAMPLE:	`var n = Math.SQRT2;`

3.8.2 Math object static methods

Math.abs()

SYNTAX:	`Math.abs(x)`
WHERE:	x - a number.
RETURN:	number - the absolute value of `x`. Returns `NaN` if `x` cannot be converted to a number.
DESCRIPTION:	Computes the absolute value of a number.

EXAMPLE:

```
//The function returns the absolute value
// of the number -2 (i.e. the return value is 2):
var n = Math.abs(-2);
```

Math.acos()

SYNTAX:	`Math.acos(x)`
WHERE:	x - a number between 1 and -1.
RETURN:	number - the arc cosine of `x`.
DESCRIPTION:	The return value is expressed in radians and ranges from 0 to `pi`. Returns `NaN` if `x` cannot be converted to a number, is greater than 1, or is less than -1.

EXAMPLE:

```
function compute_acos(x)
{
   return Math.acos(x)
}

// If you pass -1 to the function compute_acos(), the return is the
// value of pi (approximately 3.1415...), if you pass 3 the
// return is NaN since 3 is out of the range of Math.acos.
```

Math.asin()

SYNTAX:	`Math.asin(x)`
WHERE:	x - a number between 1.0 and -1.0
RETURN:	number - implementation-dependent approximation of the arc sine of the argument.
DESCRIPTION:	The return value is expressed in radians and ranges from `-pi/2` to `+pi/2`. Returns `NaN` if `x` cannot be converted to a number, is greater than 1, or less than -1.

EXAMPLE:

```
function compute_asin(x)
{
   return Math.asin(x)
}
// If you pass -1 to the function compute_acos(), the return is the
// value of -pi/2 , if you pass 3 the return is
// NaN since 3 is out of Math.acos's range.
```

Math.atan()

SYNTAX:	`Math.atan(x)`
WHERE:	x - any number.
RETURN:	number - an implementation-dependent approximation of the arctangent of the argument.
DESCRIPTION:	The return value is expressed in radians and ranges from `-pi/2` to `+pi/2`.
EXAMPLE:	

```
//The arctangent of x is returned
//in the following function:
function compute_arctangent(x)
{
   return Math.arctangent(x)
}
```

Math.atan2()

SYNTAX:	`Math.atan2(x, y)`
WHERE:	x - x coordinate of the point. x - y coordinate of the point.
RETURN:	number - an implementation-dependent approximation to the arc tangent of the quotient, `y/x`, of the arguments `y` and `x`, where the signs of the arguments are used to determine the quadrant of the result.
DESCRIPTION:	It is intentional and traditional for the two-argument arc tangent function that the argument named `y` be first and the argument named `x` be second. The return value is expressed in radians and ranges from `-pi` to `+pi`.
EXAMPLE:	

```
//The arctangent of the quotient y/x
//is returned in the following function:
function compute_arctangent_of_quotient(x, y)
{
   return Math.arctangent2(x, y)
}
```

Math.ceil()

SYNTAX:	`Math.ceil(x)`
WHERE:	x - any number or numeric expression.
RETURN:	number - the smallest number that is not less than the argument and is equal to a mathematical integer.
DESCRIPTION:	If the argument is already an integer, the result is the argument itself. Returns `NaN` if `x` cannot be converted to a number.
EXAMPLE:	

```
//The smallest number that is
//not less than the argument and is
//equal to a mathematical integer is returned
//in the following function:
function compute_small_arg_eq_to_int(x)
{
   return Math.ceil(x)
}
```

Math.cos()

SYNTAX:	`Math.cos()`
WHERE:	x - an angle, measured in radians.
RETURN:	number - an implementation-dependent approximation of the cosine of the argument
DESCRIPTION:	The argument is expressed in radians. Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by `2pi/360`.
EXAMPLE:	

```
//The cosine of x is returned
//in the following function:
function compute_cos(x)
```

```
{
   return Math.cos(x)
}
```

Math.exp()

SYNTAX: `Math.exp(x)`

WHERE: x - either a number or a numeric expression to be used as an exponent

RETURN: number - an implementation-dependent approximation of the exponential function of the argument.

DESCRIPTION: For example returns e raised to the power of the `x`, where e is the base of the natural logarithms. Returns `NaN` if `x` cannot be converted to a number.

EXAMPLE:

```
//The exponent of x is returned
//in the following function:
function compute_exp(x)
{
   return Math.exp(x)
}
```

Math.floor()

SYNTAX: `Math.floor(x)`

WHERE: x - a number.

RETURN: number - the greatest number value that is not greater than the argument and is equal to a mathematical integer.

DESCRIPTION: If the argument is already an integer, the return value is the argument itself.

EXAMPLE:

```
//The floor of x is returned
//in the following function:
function compute_floor(x)
{
   return Math.floor(x)
}
//If 6.78 is passed to compute_floor,
//7 will be returned. If 89.1
//is passed, 90 will be returned.
```

Math.log()

SYNTAX: `Math.log(x)`

WHERE: x - a number.greater than zero.

RETURN: number - an implementation-dependent approximation of the natural logarithm of `x`.

DESCRIPTION: If a negative number is passed to `Math.log()`, the return is `NaN`

EXAMPLE:

```
//The natural log of x is returned
//in the following function:
function compute_log(x)
{
   return Math.log(x)
}
//If the argument is less than 0 or NaN,
//the result is NaN
//If the argument is +0 or -0,
//the result is -infinity
//If the argument is 1, the result is +0
//If the argument is +infinity,
//the result is +infinity
```

Math.max()

SYNTAX: `Math.max(x, y)`

WHERE: x - a number.
y - a number.

RETURN: number - the larger of `x` and `y`.

DESCRIPTION: Returns `NaN` if either argument cannot be converted to a number.

EXAMPLE:

```
//The larger of x and y is returned
//in the following function:
function compute_max(x, y)
{
   return Math.max(x, y)
}
//If x = a and y = 4 the return is NaN
//If x > y the return is x
//If y > x the return is y
```

Math.min()

SYNTAX: `Math.min(x, y)`

WHERE:

x - a number.

y - a number.

RETURN: number - the smaller of `x` and `y`. Returns `NaN` if either argument cannot be converted to a number.

DESCRIPTION: Returns `NaN` if either argument cannot be converted to a number.

EXAMPLE:

```
//The smaller of x and y is returned
//in the following function:
function compute_min(x, y)
{
   return Math.min(x, y)
}
//If x = a and y = 4 the return is NaN
//If x > y the return is y
//If y > x the return is x
```

Math.pow()

SYNTAX: `Math.pow(x, y)`

WHERE:

x - The number which will be raised to the power of `Y`

y - The number which `X` will be raised to

RETURN: number - the value of `x` to the power of `y`.

DESCRIPTION: If the result of `Math.pow()` is an imaginary or complex number, `NaN` will be returned. Please note that if `Math.pow()` unexpectedly returns infinity, it may be because the floating-point value has experienced overflow.

EXAMPLE:

```
//x to the power of y is returned
//in the following function:
function compute_x_to_power_of_y(x, y)
{
   return Math.pow(x, y)
}
//If the result of Math.pow is
//an imaginary or complex number,
//the return is NaN
//If y is NaN, the result is NaN
//If y is +0 or -0, the result is 1,
//even if x is NaN
//If x = 2 and y = 3 the return value is 8
```

Math.random()

SYNTAX: `Math.random()`

RETURN: number - a number which is positive and pseudo-random and which is greater than or equal to 0 but less than 1.

DESCRIPTION: Calling this method numerous times will result in an established pattern (the sequence of numbers will be the same each time). This method takes no arguments. Seeding is not yet possible.

EXAMPLE:

```
//Return a random number:
function compute_rand_numb()
```

```
{
   return Math.random()
}
```

Math.round()

SYNTAX:	`Math.round(x)`
WHERE:	x - a number.
RETURN:	number - value that is closest to the argument and is equal to a mathematical integer. `X` is rounded up if its fractional part is equal to or greater than 0.5 and is rounded down if less than 0.5.
DESCRIPTION:	The value of `Math.round(x)` is the same as the value of `Math.floor(x+0.5)`, except when x is *0 or is less than 0 but greater than or equal to -0.5; for these cases `Math.round(x)` returns *0, but `Math.floor(x+0.5) returns +0`.
SEE:	Math.floor()
EXAMPLE:	

```
//Return a mathematical integer:
function compute_int(x)
{
   return Math.round(x)
}
//If the argument is NaN, the result is NaN
//If the argument is already an integer
//such as any of the
//following values: -0, +0, 4, 9, 8;
//then the result is the
//argument itself.
//If the argument is .2, then the result is 0.
//If the argument is 3.5, then the result is 4
//Note: Math.round(3.5) returns 4,
//but Math.round(-3.5) returns -3.
```

Math.sin()

SYNTAX:	`Math.sin(x)`
WHERE:	x - an angle in radians.
RETURN:	number - the sine of `x`, expressed in radians.
DESCRIPTION:	Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by `2pi/360`.
EXAMPLE:	

```
//Return the sine of x:
function compute_sin(x)
{
   return Math.sin(x)
}
//If the argument is NaN, the result is NaN
//If the argument is +0, the result is +0
//If the argument is -0, the result is -0
//If the argument is +infinity or -infinity,
//the result is NaN
```

Math.sqrt()

SYNTAX:	`Math.sqrt(x)`
WHERE:	x - a number or numeric expression greater than or equal to zero.
RETURN:	number - the square root of `x`.
DESCRIPTION:	Returns `NaN` if `x` is a negative number or cannot be converted to a number.
SEE:	Math.exp()
EXAMPLE:	

```
//Return the square root of x:
function compute_square_root(x)
{
   return Math.sqrt(x)
}
//If the argument is NaN, the result is NaN
```

```
//If the argument is less than 0,
//the result is NaN
//If the argument is +0, the result is +0
//If the argument is -0, the result is -0
//If the argument is +infinity,
//the result is +infinity
```

Math.tan()

SYNTAX:	`Math.tan(x)`
WHERE:	x - an angle measured in radians.
RETURN:	number - the tangent of `x`, expressed in radians.
DESCRIPTION:	Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by `2pi/360`.

EXAMPLE:

```
//Return the tangent of x:
function compute_tan(x)
{
   return Math.tan(x)
}
//If the argument is NaN, the result is NaN
//If the argument is +0, the result is +0
//If the argument is -0, the result is -0
//If the argument is +infinity or -infinity,
//the result is NaN
```

3.9 Number Object

3.9.1 Number object instance methods

Number toExponential()

SYNTAX:	`number.toExponential(fractionDigits)`
WHERE:	fractionDigits - the digits after the significand's decimal point.
RETURN:	string - A string representation of this number in exponential notation.
DESCRIPTION:	This method returns a string containing the number represented in exponential notation with one digit before the significand's decimal point and fractionDigits digits after the significand's decimal point.

Number toFixed()

SYNTAX:	`number.toFixed(fractionDigits)`
WHERE:	fractionDigits - the digits after the decimal point.
RETURN:	string - A string representation of this number in fixed-point notation.
DESCRIPTION:	This method returns a string containing the number represented in fixed-point notation with fractionDigits digits after the decimal point.

Number toLocaleString()

SYNTAX:	`number.toLocaleString()`
RETURN:	string - a string representation of this number.
DESCRIPTION:	This method behaves like `Number toString()` and converts a number to a string in a manner specific to the current locale. Such things as placement of decimals and comma separators are affected.
SEE:	Number toString()

EXAMPLE:

```
var n = 8.9;
var s = n.toLocaleString();
```

Number toPrecision()

SYNTAX:	`number.toPrecision(precision)`
WHERE:	precision - significant digits in fixed notation, or digits after the significand's decimal point in exponential notation.
RETURN:	string - A string representation of this number in either exponential notation or in fixed notation.
DESCRIPTION:	This method returns a string containing the number represented in either in exponential notation with one digit before the significand's decimal point and precision-1 digits after the significand's decimal point or in fixed notation with precision significant digits.

Number toString()

SYNTAX:	`number.toString()`
RETURN:	string - a string representation of this number.
DESCRIPTION:	This method behaves similarly to `Number toLocaleString()` and converts a number to a string using a standard format for numbers.
SEE:	Number toLocaleString()

EXAMPLE:	`var n = 8.9;`
	`var s = n.toString();`

3.10 Object Object

3.10.1 Object object instance methods

Object hasOwnProperty()

SYNTAX:	`object.hasOwnProperty(propertyName)`
WHERE:	property - name of the property about which to query.
RETURN:	boolean - indicating whether or not the current object has a property of the specified name.
DESCRIPTION:	This method simply determines if the object has a property with the name propertyName. This is almost the same as testing `defined(object[propertyName])`, except that `undefined` values are different from non-existent values, and the internal `_hasProperty()` method of the object may be called.

Object isPrototypeOf()

SYNTAX:	`object.isPrototypeOf(variable)`
WHERE:	variable - the object to test.
RETURN:	boolean - `true` if variable is an object and the current object is present in the prototype chain of the object, otherwise it returns `false`.
DESCRIPTION:	If variable is not an object, then this method immediately returns `false`. Otherwise, the method recursively searches the internal `_prototype` property of the object and if at any point the current object is equal to one of these prototype properties, then the method returns `true`.

Object propertyIsEnumerable()

SYNTAX:	`object.propertyIsEnumerable(propertyName)`
WHERE:	property - name of the property about which to query.
RETURN:	boolean - `true` if the current object has an enumerable property of the specified name, otherwise `false`.
DESCRIPTION:	If the current object has no property of the specified name, then `false` is immediately returned. If the property has the `DontEnum` attribute set, then `false` is returned. Otherwise, `true` is returned.

Object toLocaleString()

SYNTAX:	`object.toLocaleString()`
RETURN:	string - a string representation of this object.
DESCRIPTION:	This method is intended to provide a default `toLocaleString()` method for all objects. It behaves exactly if `toString()` had been called on the original object.
SEE:	Object toString()

Object toString()

SYNTAX:	`object.toString()`
RETURN:	string - a string representation of this object.
DESCRIPTION:	When this method is called, the internal class property, `_class`, is retrieved from the current object. A string is then constructed whose contents are "[object classname]", where classname is the value of the property from the current object. Note that this function is rarely called directly, rather it is called implicitly through such functions as `global.ToString()`.
SEE:	Object toLocaleString()

3.11 RegExp Object

Regular expressions do not seem very regular to average people. Regular expressions are used to search text and strings, searches that are very powerful if a person makes the effort to learn how to use them. Simple searches may be done like the following:

```
var str = "one two three";
str.indexOf("two");   // == 4
```

The `String indexOf()` method searches `str` for "two" and returns the beginning position of "two", which is 4. What if you wanted to find "t" and "o" with or without any characters in between, an "o" only at the beginning of a string, or an "e" only at the end of a string? Before answering, lets consider wildcards.

Most computer users are familiar with wildcards in searching, especially since they may be used in finding files. For example, the DOS command:

```
dir t*o.bat
```

will list all files that begin with "t" and end "o" in the filename and that have an extension of "bat". Javascript does not use wildcards to extend search capability. Instead, ECMAScript, the standard for Javascript, has implemented regular expression searches that do everything that wildcards do and much, much more. Regular expressions follow the PERL standard, though the syntax has been made easier to read. Anyone who can use regular expressions in PERL already knows how to use Javascript regular expressions. For advanced information on regular expressions, there are many books in the PERL community, in addition to Javascript books, that explain regular expressions.

Now lets answer the question about how to find the three cases mentioned above.

```
var str = "one two three";
var pat = /t.*o/;
str.search(pat);   // == 4
```

This fragment illustrates one way to use regular expressions to find "t" followed by "o" with any number of characters between them. Two things are different. One the variable `pat` which is assigned `/t.*o/`. The slashes indicate the beginning and end of a regular expression pattern, similar to how quotation marks indicate a string. The `String search()` method is a method of the String object that uses a regular expression pattern to search a string, similar to the String `indexOf()` method. In fact, they both return 4, the start position of "two", in these examples. The `String object` has three methods for searching using regular expression patterns. The three methods are:

```
String match()
String replace()
String search()
```

The methods in the RegExp object, for using regular expressions, are explained below in this section. Before we move on to the cases of an "o" at the start or an "e" at the end of a string, consider the current example a little further. What do the slashes "/ . . . /" do? First, they define a regular expression pattern. Second, they create a RegExp object. In our example, the quotes cause `str` to be a String object, and the slashes cause `pat` to be a RegExp object. Thus, `pat` may be used with RegExp methods and with the three String methods that use regular expression patterns.

```
var str = "one two three";
var pat = /t.*o/;
pat.test(str);   // == true
```

By using a method, such as `test()`, of the RegExp object, the string to be searched becomes the argument rather than the pattern to search for, as with the string methods. The `RegExp test()` method simply returns `true` or `false` indicating whether the pattern is found in the string.

```
var str = "one two three";
var pat = /t.*o/;
str.match(pat);   // == an Array with pertinent info
pat.exec(str);    // == an Array with pertinent info
```

The `String match()` and `RegExp exec()` methods return very similar, often the same, results in an Array. The return may vary depending on exactly which attributes, discussed later, are set for a regular expression.
To find an "o" only at the start of a string, use something like:

```
var str = "one two three";
var pat = /^o/;
str.search(pat);   // == 0
```

The caret "^" has a special meaning, namely, the start of a string or line. It anchors the characters that follow to the start of a string or line and is one of the special anchor characters.
To find an "e" only at the end of a string, use something like:

```
var str = "one two three";
var pat = /e$/;
str.search(pat);   // == 12
```

The dollar sign "$" has a special meaning, namely, the end of a string or line. It anchors the characters that follow to the end of a string or line and is one of the special anchor characters.
Note that there is a very important distinction between searching for pattern matches using the String methods and using the RegExp methods. The RegExp methods execute much faster, but the String methods are often quicker to program. So, if you need to do intensive searching in which a single regular expression pattern is used many times in a loop, use the RegExp

methods. If you just need to do a few searches, use the String methods. Every time a RegExp object is constructed using `new`, the pattern is compiled into a form that can be executed very quickly. Every time a new pattern is compiled using the `RegExp compile()` method, a pattern executes much faster. Other than the difference in speed and script writing time, the choice of which methods to use depends on personal preferences and the particular tasks at hand.
In general, the RegExp object allows the use of regular expression patterns in searches of strings or text. The syntax follows the ECMAScript standard, which may be thought of as a large and powerful subset of PERL regular expressions.

3.11.1 Regular expression syntax

The general form for defining a regular expression pattern is:

```
/characters/attributes
```

Assume that we are searching the string "THEY, the three men, left". The following are valid regular expression patterns followed by a description of what they find:

```
/the three/      // "the three"
/THE THREE/ig    // "the three"
/th/             // "th" in "the"
/th/igm          // "th" in "THEY", "the", and "three"
```

The slashes delimit the characters that define a regular expression. Everything between the **slashes** is part of a **regular expression**, just as everything between quotation marks is part of a string. Three letters may occur after the second slash that are not part of the regular expression. Instead, they define **attributes** of the regular expression. Any one, two, or three of the letters may be used, that is, any one or more of the attributes may be defined. Thus, a regular expression has three elements: literals, characters, and attributes.

Regular expression literals

Regular expression literals delimit a regular expression pattern. The literals are a slash "/" at the beginning of some characters and a slash "/" at the end of the characters. These regular expression literals operate in the same way as quotation marks do for string literals. The following two lines of code accomplish the same thing, namely, they define and create an instance of a `RegExp object`:

```
var re = /^THEY/;
var re = new RegExp("^THEY");
```

and so do the following two lines:

```
var re = /^THEY/i;
var re = new RegExp("^THEY", "i");
```

Regular expression characters

Each character or special character in a regular expression represents one character. Though some special characters, such as, the range of lowercase characters represented by `[a-z]`, may have multiple matches, only one at a time is matched. Thus, `[a-z]` will only find one of these 26 characters at one position in a string being searched. Just as strings have special characters, namely, escape sequences, regular expression patterns have various kinds of special characters and metacharacters that are explained below.

Regular expression attributes

The following table lists allowable attribute characters and their effects on a regular expression. No other characters are allowed as attributes.

Character	Attribute meaning
`g`	Do a global match. Allow the finding of all matches in a string using the RegExp and String methods and properties that allow global operations. The instance property `global` is set to `true`. Example: `/pattern/g`
`i`	Do case insensitive matches. The instance property `ignoreCase` is set to `true`. Example: `/pattern/i`
`m`	Work with multiple lines in a string. When working with multiple lines the "^" and "$" anchor characters will match the start and end of a string and the start and end of lines within the string. The newline character "\n" in a string indicates the end of a line and hence lines in a string. The instance property `multiline` is set to `true`. Example: /pattern/m

Attributes are the characters allowed after the end slash "/" in a regular expression pattern. The following regular expressions illustrate the use of attributes.

```
var pat = /^The/i;    // any form of "the" at start of a string
var pat = /the/g;     // all occurrences of "the" may be found
var pat = /test$/m;   // first "test" at the end of any line
var pat = /test$/igm; // all forms of "test" at end of all lines

// The following four examples do the same as the first four
var pat = new RegExp("^The",  "i");
var pat = new RegExp("the",   "g");
var pat = new RegExp("test$", "m");
var pat = new RegExp("test$", "igm");
```

3.11.2 Regular expression special characters

Regular expressions have many special characters, which are also known as metacharacters, with special meanings in a regular expression pattern. Some are simple escape sequences, such as, a newline "\n", with the same meaning as the same escape sequence in strings. But, regular expressions have many more special characters that add much power to working with strings and text, much more power than is initially recognized by people being introduced to regular expressions. For anyone who works with strings and text, the effort to become proficient with regular expression parsing is more than worthwhile.

Regular expression summary

Search pattern

?	zero or one of previous, {0,1}	be?t
0	zero or more of previous, maximal, {0,}	b.*t
*?	zero or more of previous, minimal, {0,}?	b.*?t
0	one or more of previous, maximal, {1,}	b.+t
+?	one or more of previous, minimal, {1,}?	b.+?t
{n}	n times of previous	be{n}t
{n,}	n or more times of previous, maximal	b.{n,}t
{n,}?	n or more times of previous, minimal	b.{n,}?t
{n,m}	n to m times of previous	be{1,2}t
.	any character	b.t
[]	any one character in a class	[a-m]
[^]	any one not in a character class	[^a-m]
[\b]	one backspace character	my[\b]word
\d	any one digit, [0-9]	file\d
\D	any one not digit, [^0-9]	file\D
\s	any one white space character, [\t\n\r\f\v]	my\sword
\S	any one not white space character, [^ \t\n\r\f\v]	my \sord
\w	any one word character, [a-zA-Z0-9_]	my big\w
\W	any one not word character, [^a-zA-Z0-9_]	my\Wbig
^	anchor to start of string	^string
$	anchor to end of string	string$
\b	anchor to word boundary	\bbig
\B	anchor to not word boundary	\Bbig
|	or	(bat)|(bet)
\n	group n	(bat)a\1
()	group	my(.?)fil
(?:)	group without capture	my(?:.?)fil
(?=)	group without capture with positive look ahead	my(?=.?)fil
(?!)	group without capture with negative look ahead	my(?!.?)fil
\f	form feed character	string\f
\n	newline	string\n
\r	carriage return character	string\r
\t	horizontal tab character	one\tfour
\v	vertical tab character	one\vtwo
\/	/ character	\/fil
\\	\ character	\\fil
\.	. character	fil\.bak
*	* character	one*two
\+	+ character	\+fil
\?	? character	when\?
\\|	| character	one\\|two
\(	(character	\(fil\)
\)	) character	\(fil\)
\[	[character	\[fil\]
\]	] character	\[fil\]
\{	{ character	\{fil\}

\}	} character	\{fil\}
\C	a character itself. Seldom used.	b\at
\cC	a control character	one\cIfour
\x##	character by hexadecimal code	\x41
\###	character by octal code	\101

Replace pattern

$n	group n in search pattern, $1, $2, . . . $9	big$1
$+	last group in search pattern	big$+
$`	text before matched pattern	big$`
$'	text after matched pattern	big$'
$&	text of matched pattern	big$&
\$	$ character	big\$

Regular expression repetition characters

Notice that the character "?" pulls double duty. When used as the only repetition specifier, "?" means to match zero or more occurrences of the previous character. For example, /a?/ matches one or more "a" characters in sequence. When used as the second character of a repetition specifier, as in "*?", "+?", and "{n,}?", a question mark "?" indicates a minimal match. What is meant by a minimal match?

Well obviously, it is the counterpart to a maximal match, which is the default for Javascript and PERL regular expressions. A maximal match will include the maximum number of characters in a text that will qualify to match a regular expression pattern. For example, in the string "one two three", the pattern /o.*e/ will match the text "one two three". Why? The pattern says to match text that begins with the character "o" followed by zero or more of any characters up to the character "e". Since the default is a maximal match, the whole string is matched since it begins with "o" and ends with "e". Often, this maximal match behavior is not what is expected or desired.

Now consider a similar match using the minimal character. The string is still "one two three", but the pattern becomes /o.*?e/. Notice that the only difference is the addition of a question mark "?" as the second repetition character after the "*". The text matched this time is "one", which is the minimal number of characters that match the conditions of the regular expression pattern.

So, it might be a good habit to begin reading regular expression patterns with a maximal and minimal vocabulary. As an example, lets spell out how we could read the two patterns in the current example.

- "o.*e" - match text that begin with "o" and has the maximum number of characters possible until the last "e" is encountered.
- "o.*?e" - match text that begins with "o" and has the minimum number of characters possible until the first "e" is encountered.

Sometimes a maximal match is called a greedy match and a minimal match is called a non-greedy match.

Repetition	**How many characters matched**
?	Match zero or one occurrence of the previous character or sub pattern. Same as {0,1}
0	Match zero or more occurrences of the previous character or sub pattern. A maximal match, that is, match as many characters as will fulfill the regular expression. Same as {0,}

`*?`	Match zero or more occurrences of the previous character or sub pattern. A minimal match, that is, match as few characters as will fulfill the regular expression. Same as `{0,}?`
`0`	Match one or more occurrences of the previous character or sub pattern. A maximal match, that is, match as many characters as will fulfill the regular expression. Same as `{1,}`
`+?`	Match one or more occurrences of the previous character or sub pattern. A minimal match, that is, match as few characters as will fulfill the regular expression. Same as `{1,}?`
`{n}`	Match `n` occurrences of the previous character or sub pattern.
`{n,}`	Match `n` or more occurrences of the previous character or sub pattern. A maximal match, that is, match as many characters as will fulfill the regular expression.
`{n,}?`	Match `n` or more occurrences of the previous character or sub pattern. A minimal match, that is, match as few characters as will fulfill the regular expression.
`{n, m}`	Match the previous character or sub pattern at least `n` times but not more than `m` times.

Regular expression character classes

Class	**Character matched**
`.`	Any character except newline, `[^\n]`
`[...]`	Any one of the characters between the brackets
`[^...]`	Any one character not one of the characters between the brackets
`[\b]`	A backspace character (special syntax because of the `\b` boundary)
`\d`	Any digit, `[0-9]`
`\D`	Any character not a digit, `[^0-9]`
`\s`	Any white space character, `[ \t\n\r\f\v]`
`\S`	Any non-white space character, `[^ \t\n\r\f\v]`
`\w`	Any word character, `[a-zA-Z0-9_]`
`\W`	Any non-word character, `[^a-zA-Z0-9_]`

Regular expression anchor characters

Anchor characters indicate that the following or preceding characters must be next to a special position in a string. The characters next to anchor characters are included in a match, not the anchor characters themselves. For example, in the string "The big cat and the small cat", the regular expression `/cat$/` will match the "cat" at the end of the string, and the match will include only the three characters "cat". The "$" is an anchor character indicating the end of a string (or line if a multiline search is being done).
The following table lists the anchor characters, metacharacters, and their meanings.

Character	**Anchor meaning**
`^`	The beginning of a string (or line if doing a multiline search). Example: `/^The/`
`$`	The end of a string (or line if doing a multiline search). Example: `/cat$/`
`\b`	A word boundary. Match any character that is not considered to be a valid character for a word in programming. The character class "\W", not a word character, is similar. There are two differences. One, "\b" also matches a

	backspace. Two, "\W" is included in a match, since it is regular expression character, but "\b" is not included in a match. Example: `/\bthe\b/`
`\B`	Not a word boundary. The character class "\w" is similar. The most notable difference is that "\w" is included in a match, and "\B" is not. Example: `/l\B/`

Regular expression reference characters

Character	**Meaning**
|	Or. Match the character or sub pattern on the left **or** the character or sub pattern on the right.
`\n`	Reference to group. Match the same characters, not the regular expression itself, matched by group `n`. Groups are sub patterns that are contained in parentheses. Groups may be nested. Groups are numbered according to the order in which the left parenthesis of a group appears in a regular expression.
`(...)`	Group with capture. Characters inside of parentheses are handled as a single unit or sub pattern in specified ways, such as with the first two explanations, | and `\n`, in this table. The characters that are actually matched are captured and may be used later in an expression (as with `\n`) or in a replacement expression (as with `$n`). For example, if the string "one two three two one" and the pattern `/(o.e).+(w.+?e)/` are used, then the back references `$1` or `\1` use the text "one".
`(?:...)`	Group without capture. Matches the same text as `(...)`, but the text matched is not captured or saved and is not available for later use using `\n` or `$n`. The overhead of not capturing matched text becomes important in faster execution time for searches involving loops and many iterations. Also, some expressions and replacements can be easier to read and use with fewer numbered back references with which to keep up. For example, if the string "one two three two one" and the pattern `/(?:o.e).+(w.+?e)/` are used, then the back references `$1` or `\1` use the text "wo thre".
`(?=...)`	Positive look ahead group without capture. The position of the match is at the beginning of the text that matches the sub pattern. For example, `/Javascript (?=Desktop|ISDK)/` matches "Javascript " in "Javascript Desktop" or "Javascript ISDK", but not "Javascript " in "Javascript Web Server". When a search continues, it begins after "Javascript ", not after "Desktop" or "ISDK". That is, the search continues after the last text matched, not after the text that matches the look ahead sub pattern.
`(?!...)`	Negative look ahead group without capture. The position of the match is at the beginning of the text not matching the sub pattern. For example, `/Javascript (?!Desktop|ISDK)/` matches "Javascript " in "Javascript Web Server", but not "Javascript " in "Javascript Desktop" or "Javascript ISDK". When a search continues, it begins after "Javascript ", not after "Desktop" or "ISDK". That is, the search continues after the last text matched, not after the text that matches the look ahead sub pattern.

Regular expression escape sequences

Sequence	Character represented
`\f`	Form feed, `\cL`, `\x0C`, `\014`
`\n`	Line feed, newline, `\cJ`, `\x0A`, `\012`
`\r`	Carriage return, `\cM`, `\x0D`, `\015`
`\t`	Horizontal tab, `\cI`, `\x09`, `\011`
`\v`	Vertical tab, `\cK`, `\x0B`, `\013`
`\/`	The character: /
`\\`	The character: \
`\.`	The character: .
`\*`	The character: *
`\+`	The character: +
`\?`	The character: ?
`\\|`	The character: |
`\(`	The character: (
`\)`	The character:)
`\[`	The character: [
`\]`	The character:]
`\{`	The character: {
`\}`	The character: }
`\C`	A character itself, if not one of the above. Seldom, if ever, used.
`\cC`	A control character. For example, `\cL` is a form feed (`^L` or `Ctrl-L`), same as `\f`.
`\x##`	A character represented by its code in hexadecimal. For example, `\x0A` is a newline, same as `\n`, and `\x41` is "A".
`\###`	A character represented by its code in octal. For example, `\012` is a newline, same as `\n`, and `\101` is "A".

Regular expression replacement characters

All of the special characters that have been discussed so far pertain to regular expression patterns, that is, to finding and matching strings and patterns in a target string. If all you want to do is find text, then you do not need to know about regular expression replacement characters. However, most people not only want to do powerful searches, but they also want to make powerful replacements of found text. This section describes special characters that are used in replacement strings and that are related to special characters used in search patterns.

Expression	Meaning
`$1, $2 ... $9`	The text that is matched by sub patterns inside of parentheses. For example, `$1` substitutes the text matched in the first parenthesized group in a regular expression pattern. See the groups, `(...)`, `(?:...)`, `(?=...)`, and `(?!...)`, under `regular expression reference characters`.
`$+`	The text matched by the last group, that is, parenthesized sub pattern.
`` $` ``	The text before, to the left of, the text matched by a pattern.
`$'`	The text after, to the right of, the text matched by a pattern
`$&`	The text matched by a pattern
`\$`	A literal dollar sign, $.

3.11.3 Regular expression precedence

The patterns, characters, and metacharacters of regular expressions comprise a sub language for working with strings. Some of the metacharacters can be understood as operators, and, like operators in all programming languages, there is an order of precedence. The following tables list regular expression operators in the order of their precedence.

Operator	**Descriptions**
`\`	Escape
`()`, `(?:)`, `(?=)`, `(?!)`, `[]`	Groups and sets
`*`, `+`, `?`, `{n}`, `{n,}`, `{n,m}`	Repetition
`^`, `$`, `\metacharacter`	Anchors and metacharacters
`|`	Alternation

3.11.4 RegExp object instance properties

RegExp global

SYNTAX: `regexp.global`

DESCRIPTION: A read-only property of an instance of a `RegExp object`. It is `true` if "g" is an attribute in the regular expression pattern being used.
Read-only property. Use RegExp compile() to change.

SEE: Regular expression attributes

EXAMPLE:

```
var pat = /^Begin/g;
//or
var pat = new RegExp("^Begin", "g");
```

RegExp ignoreCase

SYNTAX: `regexp.ignoreCase`

DESCRIPTION: A read-only property of an instance of a `RegExp object`. It is `true` if "i" is an attribute in the regular expression pattern being used.
Read-only property. Use RegExp compile() to change.

SEE: Regular expression attributes

EXAMPLE:

```
var pat = /^Begin/i;
//or
var pat = new RegExp("^Begin", "i");
```

RegExp lastIndex

SYNTAX: `regexp.lastIndex`

DESCRIPTION: The character position after the last pattern match and which is the basis for subsequent matches when finding multiple matches in a string. That is, in the next search, `lastIndex` is the starting position. This property is used only in global mode after being set by using the "g" attribute when defining or compiling a search pattern. `RegExp exec()` and `RegExp test()` use and set the `lastIndex` property. If a match is not found by one of them, then `lastIndex` is set to 0. Since the property is read/write, you may set the property at any time to any position.
Read/write property.

SEE: RegExp exec(), String match()

EXAMPLE:

```
var str = "one tao three tio one";
var pat = /t.o/g;
pat.exec(str);
   // pat.lastIndex == 7
```

RegExp multiline

SYNTAX: `regexp.multiline`

DESCRIPTION: A read-only property of an instance of a `RegExp object`. It is `true` if "m" is an

attribute in the regular expression pattern being used. There is no static (or global) RegExp `multiline` property in Javascript Javascript since the presence of one is based on old technology and is confusing now that an instance property exists. This property determines whether a pattern search is done in a multiline mode. When a pattern is defined, the `multiline` attribute may be set, for example, `/^t/m`. A pattern definition such as this one, sets the instance property `regexp.multiline` to `true`.
Read-only property. Use `RegExp compile()` to change.

SEE: Regular expression attributes

EXAMPLE:

```
// In all these examples, pat.multiline is set
// to true. If there were no "m" in the attributes,
// then pat.multiline would be set to false.
var pat = /^Begin/m;
//or
var pat = new RegExp("^Begin", "igm");
//or
var pat = /^Begin/m;
//or
var pat = new RegExp("^Begin", "igm");
```

RegExp source

SYNTAX: `regexp.source`

DESCRIPTION: The regular expression pattern being used to find matches in a string, not including the attributes.
Read-only property. Use `RegExp compile()` to change.

SEE: Regular expression syntax

EXAMPLE:

```
var str = "one tao three tio one";
var pat = /t.o/g;
pat.exec(str);
   // pat.source == "t.o"
```

3.11.5 RegExp returned array properties

Some methods, `String match()` and `RegExp exec()` return arrays in which various elements and properties are set that provide more information about the last regular expression search. The properties that might be set are described in this section, not the contents of the array elements.

index (RegExp)

SYNTAX: `returnedArray.index`

DESCRIPTION: When `String match()` is called and the "g" is not used in the regular expression, String `match()` returns an array with two extra properties, `index` and `input`. The property `index` has the start position of the match in the target string.

SEE: input (RegExp), RegExp exec(), String match()

EXAMPLE:

```
var str = "one tao three tio one";
var pat = /(t.o)\s(t.r)/g;
var rtn = pat.exec(str);
   // rtn[0] == "tao thr"
   // rtn[1] == "tao"
   // rtn[2] == "thr"
   // rtn.index == 4
   // rtn.input == "one tao three tio one"
```

input (RegExp)

SYNTAX: `returnedArray.input`

DESCRIPTION: When `String match()`is called and the "g" is not used in the regular expression, String `match()` returns an array with two extra properties, `index` and `input`. The property `input` has a copy of the target string.

SEE: index (RegExp), RegExp exec(), String match()

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)\s(t.r)/g;
var rtn = pat.exec(str);
   // rtn[0] == "two thr"
   // rtn[1] == "two"
   // rtn[2] == "thr"
   // rtn.index == 4
   // rtn.input == "one two three two one"
```

3.11.6 RegExp object instance methods

RegExp()

SYNTAX: `new RegExp([pattern[, attributes]])`

WHERE: pattern - a string containing a regular expression pattern to use with this `RegExp object`.

attributes - a string with the attributes for this RegExp object.

RETURN: object - a new regular expression object, or `null` on error.

DESCRIPTION: Creates a new regular expression object using the search pattern and options if they are specified.

If the attributes string is passed, it must contain one or more of the following characters or be an empty string `""`:

`i` - sets the ignoreCase property to `true`
`g` - sets the global property to `true`
`m` - set the multiline property to `true`

SEE: Regular expression syntax, String match(), String replace(), String search()

EXAMPLE:

```
// no options
var regobj = new RegExp( "r*t", "" );
// ignore case
var regobj = new RegExp( "r*t", "i" );
// global search
var regobj = new RegExp( "r*t", "g" );
// set both to be true
var regobj = new RegExp( "r*t", "ig" );
```

RegExp compile()

SYNTAX: `regexp.compile(pattern[, attributes])`

WHERE: pattern - a string with a new regular expression pattern to use with this RegExp object.

attributes - a string with the new attributes for this RegExp object.

RETURN: void.

DESCRIPTION: This method changes the pattern and attributes to use with the current instance of a `RegExp object`. An instance of a RegExp object may be used repeatedly by changing it with this method.

If the attributes string is supplied, it must contain one or more of the following characters or be an empty string `""`:

`i` - sets the ignoreCase property to `true`
`g` - sets the global property to `true`
`m` - set the multiline property to `true`

SEE: RegExp(), Regular expression syntax

EXAMPLE:

```
var regobj = new RegExp("now");
// use this RegExp object
regobj.compile("r*t");
// use it some more
regobj.compile("t.+o", "ig");
// use it some more
```

RegExp exec()

SYNTAX: `regexp.exec([str])`

WHERE: str - a string on which to perform a regular expression match. Default is `RegExp.input`.

RETURN: array - an array with various elements and properties set depending on the attributes of a regular expression. Returns `null` if no match is found.

DESCRIPTION: This method, of all the RegExp and String methods, is both the most powerful and most complex. For many, probably most, searches, other methods are quicker and easier to use. A string, the target, to be searched is passed to `exec()` as a parameter. If no string is passed, then `RegExp.input`, which is a read/write property, is used as the target string.

When executed without the global attribute, "g", being set, if a match is found, element 0 of the returned array is the text matched, element 1 is the text matched by the first sub pattern in parentheses, element 2 the text matched by the second sub pattern in parentheses, and so forth. These elements and their numbers correspond to groups in regular expression patterns and replacement expressions. The `length` property indicates how many text matches are in the returned array. In addition, the returned array has the `index` and `input` properties. The `index` property has the start position of the first text matched, and the `input` property has the target string that was searched. These two properties are the same as those that are part of the returned array from `String match()` when used without its global attribute being set.

When executed with the global attribute being set, the same results as above are returned, but the behavior is more complex which allows further operations. This method `exec()` begins searching at the position, in the target string, specified by `this.lastIndex`. After a match, `this.lastIndex` is set to the position after the last character in the text matched. Thus, you can easily loop through a string and find all matches of a pattern in it. The property `this.lastIndex` is read/write and may be set at anytime. When no more matches are found, `this.lastIndex` is reset to 0.

Since `RegExp exec()` always includes all information about a match in its returned array, it is the best, perhaps only, way to get all information about all matches in a string.

As with `String match()`, if any matches are made, appropriate `RegExp object static properties`, such as `RegExp.leftContext`, `RegExp.rightContext`, `RegExp.$n`, and so forth are set, providing more information about the matches.

SEE: String match(), RegExp object static properties

EXAMPLE:

```
var str = "one two three tio one";
var pat = new RegExp("t.o", "g");

while ((rtn = pat.exec(str)) != null)
   writeLog("Text = " + rtn[0] +
            " Pos = " + rtn.index +
            " End = " + pat.lastIndex);
// Display is:
//  Text = two Pos = 4 End = 7
//  Text = tio Pos = 14 End = 17
```

RegExp test()

SYNTAX: `regexp.test([str])`

WHERE: str - a string on which to perform a regular expression match. Default is `RegExp.input`.

RETURN: boolean - `true` if there is a match, else `false`.

DESCRIPTION: Tests a string to see if there is a match for a regular expression pattern.

This method is equivalent to `regexp.exec(string)!=null`.

If there is a match, appropriate `RegExp object static properties`, such as `RegExp.leftContext`, `RegExp.rightContext`, `RegExp.$n`, and so forth, are set, providing more information about the matches.

Though it is unusual, `test()` may be used in a special way when the global attribute, "g", is set for a regular expression pattern. Like with `RegExp exec()`, when a match is found, the `lastIndex` property is set to the character position after the text match. Thus, `test()` may be used repeatedly on a string, though there are few reasons to do so. One reason would be if you only wanted to know if a string had more than one match.

SEE: RegExp exec(), String match(), String search()

EXAMPLE:

```
var rtn;
var str = "one two three tio one";
var pat = /t.o/;
   // rtn == true
rtn = pat.test(str);
```

3.11.7 RegExp object static properties

RegExp.$n

SYNTAX: `RegExp.$n`

DESCRIPTION: The text matched by the n^th^ group, that is, the n^th^ sub pattern in parenthesis. The numbering corresponds to `\n`, back references in patterns, and `$n`, substitutions in replacement patterns.
Read-only property.

SEE: Regular expression reference characters, regular expression replacement characters

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)\s/
str.match(pat)
   // RegExp.$1 == "two"
```

RegExp.input

SYNTAX: `RegExp.input`

DESCRIPTION: If no string is passed to `RegExp exec()` or to `RegExp test()`, then `RegExp.input` is used as the target string. To be used as the target string, it must be assigned a value. `RegExp.input` is equivalent to `RegExp.$_`, for compatibility with PERL.
Read/write property.

SEE: RegExp exec(), RegExp test()

EXAMPLE:

```
var pat = /(t.o/;
RegExp.input = "one two three two one";
pat.exec();
   // "two" is matched
```

RegExp.lastMatch

SYNTAX: `RegExp.lastMatch`

DESCRIPTION: This property has the text matched by the last pattern search. It is the same text as in element 0 of the array returned by some methods. `RegExp.lastMatch` is equivalent to `RegExp["$&"]`, for compatibility with PERL.
Read-only property.

SEE: RegExp exec(), String match(), RegExp returned array properties

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)/
pat.exec(str);
   // RegExp.lastMatch == "two"
```

RegExp.lastParen

SYNTAX: `RegExp.lastParen`

DESCRIPTION: This property has the text matched by the last group, parenthesized sub pattern, in the last pattern search. `RegExp.lastParen` is equivalent to `RegExp["$+"]`, for compatibility with PERL.
Read-only property.

SEE:	RegExp.$n

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)+\s(t.r)/
pat.exec(str);
   // RegExp.lastParen == "thr"
```

RegExp.leftContext

SYNTAX:	`RegExp.leftContext`
DESCRIPTION:	This property has the text before, that is, to the left of, the text matched by the last pattern search. `RegExp.leftContext` is equivalent to ``RegExp["$`"]``, for compatibility with PERL. Read-only property.
SEE:	RegExp.lastMatch, RegExp.rightContext

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)/
pat.exec(str);
   // RegExp.leftContext == "one "
```

RegExp.rightContext

SYNTAX:	`RegExp.rightContext`
DESCRIPTION:	This property has the text after, that is, to the right of, the text matched by the last pattern search. `RegExp.leftContext` is equivalent to `RegExp["$'"]`, for compatibility with PERL. Read-only property.
SEE:	RegExp.lastMatch, RegExp.leftContext

EXAMPLE:

```
var str = "one two three two one";
var pat = /(t.o)/
pat.exec(str);
   // RegExp.leftContext == " three two one"
```

3.12 String Object

The String object is a data type and is a hybrid that shares characteristics of primitive data types and of composite data types. The String is presented in this section under two main headings in which the first describes its characteristics as a primitive data type and the second describes its characteristics as an object.

3.12.1 String as data type

A string is an ordered series of characters. The most common use for strings is to represent text. To indicate that text is a string, it is enclosed in quotation marks. For example, the first statement below puts the string `"hello"` into the variable `hello`. The second sets the variable `word` to have the same value as a previous variable `hello`:

```
var hello = "hello";
var word = hello;
```

Escape sequences for characters

Some characters, such as a quotation mark, have special meaning to the interpreter and must be indicated with special character combinations when used in strings. This allows the interpreter to distinguish between a quotation mark that is part of a string and a quotation mark that indicates the end of the string. The table below lists the characters indicated by escape sequences:

`\a`	Audible bell	
`\b`	Backspace	
`\f`	Formfeed	
`\n`	Newline	
`\r`	Carriage return	
`\t`	Horizontal Tab	
`\v`	Vertical tab	
`\'`	Single quote	
`\"`	Double quote	
`\\`	Backslash character	
`\0`	Null character	(e.g., "`\0`"is the `null` character)
`\###`	Octal number (0-7)	(e.g., "`033`"is the escape character)
`\x##`	Hex number (0-F)	(e.g., "`x1B`"is the escape character)
`\u####`	Unicode number (0-F)	(e.g., "`u001B`"is escape character)

Note that these escape sequences cannot be used within strings enclosed by back quotes, which are explained below.

Single quote

You can declare a string with single quotes instead of double quotes. There is no difference between the two in Javascript, except that double quote strings are used less commonly by many scripters.

Back quote

Javascript provides the back quote "`` ` ``", also known as the back-tick or grave accent, as an alternative quote character to indicate that escape sequences are not to be translated. Any special characters represented with a backslash followed by a letter, such as "`\n`", cannot be used in back tick strings.
For example, the following lines show different ways to describe a single file name:

```
"c:\\autoexec.bat" // traditional C method
'c:\\autoexec.bat' // traditional C method
`c:\autoexec.bat`  // alternative Javascript method
```

Back quote strings are not supported in most versions of Javascript. So if you are planning to port your script to some other Javascript interpreter, you should not use them.

Long Strings

You can use the + operator to concatenate strings. The following line:

```
var proverb = "A rolling stone " + "gathers no moss."
```

creates the variable proverb and assigns it the string "A rolling stone gathers no moss." If you try to concatenate a string with a number, the number is converted to a string.

```
var newstring = 4 + "get it";
```

This bit of code creates newstring as a string variable and assigns it the string "4get it".

The use of the + operator is the standard way of creating long strings in Javascript. In Javascript, the + operator is optional. For example, the following:

```
var badJoke =
   "I was standing in front of an Italian "
   "restaurant waiting to get in when this guy "
   "came up and asked me, \"Why did the "
   "Italians lose the war?\" I told him I had "
   "no idea. \"Because they ordered ziti"
   "instead of shells,\" he replied."
```

creates a long string containing the entire bad joke.

3.12.2 String as object

Strictly speaking, the String object is not truly an object. It is a hybrid of a primitive data type and of an object. As an example of its hybrid nature, when strings are assigned using the assignment operator, the equal sign, the assignment is by value, that is, a copy of a string is actually transferred to a variable. Further, when strings are passed as arguments to the parameters of functions, they are passed by value. Objects, on the other hand, are assigned to variables and passed to parameters by reference, that is, a variable or parameter points to or references the original object.

Strings have both properties and methods which are listed in this section. These properties and methods are discussed as if strings were pure objects. Strings have instance properties and methods and are shown with a period, ".", at their beginnings. A specific instance of a variable should be put in front of a period to use a property or call a method. The exception to this usage is a static method which actually uses the identifier String, instead of a variable created as an instance of String. The following code fragment shows how to access the .length property, as an example for calling a String property or method:

```
var TestStr = "123";
var TestLen = TestStr.length;
```

3.12.3 String object instance properties

String length

SYNTAX:	`string.length`
DESCRIPTION:	The length of a string, that is, the number of characters in a string. Javascript strings may contain the "`\0`" character.
SEE:	String lastIndexOf()

EXAMPLE:	`var s = "a string";`
	`var n = s.length;`

3.12.4 String object instance methods

String()

SYNTAX:	`new String([value])`
WHERE:	value - value to be converted to a string as this string object.
RETURN:	This method returns a new string object whose value is the supplied value.
DESCRIPTION:	If `value` is not supplied, then the empty string "" is used instead. Otherwise, the value `ToString(value)` is used. Note that if this function is called directly, without the new operator, then the same construction is done, but the returned variable is

converted to a string, rather than being returned as an object.

SEE: RegExp()

EXAMPLE:

```
var s = new String(123);
```

String charAt()

SYNTAX: `string.charAt(position)`

WHERE: position - offset within a string.

RETURN: string - character at position

DESCRIPTION: This method gets the character at the specified position. If no character exists at location `position`, or if `position` is less than 0, then `NaN` is returned.

SEE: String charCodeAt()

EXAMPLE:

```
// To get the first character in a string,
// use as follows:

var string = "a string";
string.charAt(0);

// To get the last character in a string, use:
string.charAt(string.length - 1);
```

String charCodeAt()

SYNTAX: `string.charCodeAt(index)`

WHERE: position - index of the character the encoding of which is to be returned.

RETURN: number - representing the unicode value of the character at position index of a string. Returns `NaN` if there is no character at the position.

SEE: String charAt(), String.fromCharCode()

DESCRIPTION: This method gets the nth character code from a string.

String concat()

SYNTAX: `string.concat([string1, ...])`

WHERE: stringN - A list of strings to append to the end of the current object.

RETURN: This method returns a string value (not a string object) consisting of the current object and any subsequent arguments appended to it.

DESCRIPTION: This method creates a new string whose contents are equal to the current object. Each argument is then converted to a string using `global.ToString()` and appended to the newly created string. This value is then returned. Note that the original object remains unaltered. The '+' operator performs the same function.

SEE: Array concat()

EXAMPLE:

```
// The following line:

var proverb = "A rolling stone " + "gathers no moss."

// creates the variable proverb and
// assigns it the string
// "A rolling stone gathers no moss."
// If you try to concatenate a string with a number,
// the number is converted to a string.

 var newstring = 4 + "get it";

// This bit of code creates newstring as a string
// variable and assigns it the string
// "4get it".

// The use of the + operator is the standard way of
// creating long strings in Javascript.
// In Javascript, the + operator is optional.
// For example, the following:
```

```
var badJoke = "I was in front of an Italian "
    "restaurant waiting to get in when this guy "
    "came up and asked me, \"Why did the "
    "Italians lose the war?\" I told him I had "
    "no idea. \"Because they ordered ziti"
    "instead of shells,\" he replied."

// creates a long string containing the entire bad joke.
```

String indexOf()

SYNTAX: `string.indexOf(substring[, offset])`

WHERE: substring - substring to search for within string.
offset - optional integer argument which specifies the position within string at which the search is to start. Default is 0.

RETURN: number - index of the first appearance of a substring in a string, else -1, if `substring` not found.

DESCRIPTION: String `indexOf()` searches the string for the string specified in `substring`. The search begins at `offset` if `offset` is specified; otherwise the search begins at the beginning of the string. If `substring` is found, String `indexOf()` returns the position of its first occurrence. Character positions within the string are numbered in increments of one beginning with zero.

SEE: String charAt(), String lastIndexOf(), String substring()

EXAMPLE:

```
var string = "what a string";
string.indexOf("a")

// returns the position, which is 2 in this example,
// of the first "a" appearing in the string.
// The method indexOf()may take an optional second
// parameter which is an integer indicating the index
// into a string where the method starts searching
// the string. For example:

var magicWord = "abracadabra";
var secondA = magicWord.indexOf("a", 1);

// returns 3, index of the first "a" to be found in
// the string when starting from the second letter of // the string.
// Since the index of the first character is 0, the
// index of second character is 1.
```

String lastIndexOf()

SYNTAX: `string.lastIndexOf(substring[, offset])`

WHERE: substring - The substring that is to be searched for within string
offset - An optional integer argument which specifies the position within string at which the search is to start. Default is 0.

RETURN: number - index of the last appearance of a substring in a string, else -1, if `substring` not found.

SEE: String indexOf()

DESCRIPTION: This method is similar to `String indexOf()`, except that it finds the last occurrence of a character in a string instead of the first.

String localeCompare()

SYNTAX: `string.localeCompare(compareStr)`

WHERE: compareStr - a string with which to compare an instance string.

RETURN: number - indicating the relationship of two strings.

- **< 0** if string is less than compareStr
- **= 0** if string is the same as compareStr
- **> 0** if string is greater than compareStr

DESCRIPTION: This method returns a number that represents the result of a locale-sensitive string comparison of this object with that object. The result is intended to order strings in the sort order specified by the system default locale, and will be negative, zero, or positive, depending on whether string comes before compareStr in the sort order, the strings are equal, or string comes after compareStr.

String match()

SYNTAX: `string.match(pattern)`

WHERE: pattern - a regular expression pattern to find or match in string.

RETURN: array - an array with various elements and properties set depending on the attributes of a regular expression. Returns `null` if no match is found.

DESCRIPTION: This method behaves differently depending on whether pattern has the "g" attribute, that is, on whether the match is global.

If the match is not global, string is searched for the first match to pattern. A `null` is returned if no match is found. If a match is found, the return is an array with information about the match. Element 0 has the text matched. Elements 1 and following have the text matched by sub patterns in parentheses. The element numbers correspond to group numbers in `regular expression reference characters` and `regular expression replacement characters`. The array has two extra properties: `index` and `input`. The property `index` has the position of the first character of the text matched, and `input` has the target string.

If the match is global, string is searched for all matches to pattern. A `null` is returned if no match is found. If one or more matches are found, the return is an array in which each element has the text matched for each find. There are no `index` and `input` properties. The `length` property of the array indicates how many matches there were in the target string.

If any matches are made, appropriate `RegExp object static properties`, such as `RegExp.leftContext`, `RegExp.rightContext`, `RegExp.$n`, and so forth are set, providing more information about the matches.

SEE: RegExp exec(), String replace(), String search(), Regular expression replacement characters, RegExp object static properties

EXAMPLE:

```
   // not global
var pat = /(t(.)o)/;
var str = "one two three tio one";
   // rtn == "two"
   // rtn[0] == "two"
   // rtn[1] == "two"
   // rtn[2] == "w"
   // rtn.index == 4
   // rtn.input == "one two three two one"
rtn = str.match(pat);

   // global
var pat = /(t(.)o)/g;
var str = "one two three tio one";
   // rtn[0] == "two"
   // rtn[1] == "tio"
   // rtn.length == 2
rtn = str.match(pat);
```

String replace()

SYNTAX: `string.replace(pattern, replexp)`

WHERE: pattern - a regular expression pattern to find or match in string.
replexp - a replacement expression which may be a string, a string with regular expression elements, or a function.

RETURN: string - the original string with replacements in it made according to pattern and replexp.

DESCRIPTION: This string is searched using the regular expression pattern defined by pattern. If a match is found, it is replaced by the substring defined by replexp. The parameter replexp may be a:

- a simple string
- a string with special regular expression replacement elements in it
- a function that returns a value that may be converted into a string

If any replacements are done, appropriate `RegExp object static properties`, such as `RegExp.leftContext`, `RegExp.rightContext`, `RegExp.$n`, and so forth are set, providing more information about the replacements.

The special characters that may be in a replacement expression are (see `regular expression replacement characters`):

- `$1, $2 ... $9`
  The text that is matched by regular expression patterns inside of parentheses. For example, $1 will put the text matched in the first parenthesized group in a regular expression pattern. See (...) under `regular expression reference characters`.
- `$+`
  The text that is matched by the last regular expression pattern inside of the last parentheses, that is, the last group.
- `$&`
  The text that is matched by a regular expression pattern.
- `` $` ``
  The text to the left of the text matched by a regular expression pattern.
- `$'`
  The text to the right of the text matched by a regular expression pattern.
- `\$`
  The dollar sign character.

SEE: String match(), String search(), Regular expression replacement characters, RegExp object static properties

EXAMPLE:

```
var rtn;
var str = "one two three two one";
var pat = /(two)/g;

   // rtn == "one zzz three zzz one"
rtn = str.replace(pat, "zzz");
   // rtn == "one twozzz three twozzz one";
rtn = str.replace(pat, "$1zzz");
   // rtn == "one 5 three 5 one"
rtn = str.replace(pat, five());
   // rtn == "one twotwo three twotwo one";
rtn = str.replace(pat, "$&$&);

function five()
{
   return 5;
}
```

String search()

SYNTAX: `string.search(pattern)`

WHERE: pattern - a regular expression pattern to find or match in string.

RETURN: number - the starting position of the first matched portion or substring of the target string. Returns -1 if there is no match.

DESCRIPTION: This method returns a number indicating the offset within the string where the pattern matched or -1 if there was no match. The return is the same character position as returned by the simple search using `String indexOf()`. Both `search()` and `indexOf()` return the same character position of a match or find. The difference is that `indexOf()` is simple and `search()` is powerful.

The `search()` method ignores a "g" attribute if it is part of the regular expression pattern to be matched or found. That is, `search()` cannot be used for global searches in a string.

After a search is done, the appropriate RegExp object static properties are set.

SEE: String match(), String replace(), RegExp exec(), Regular expression syntax, RegExp Object, RegExp object static properties

EXAMPLE:

```
var str = "one two three four five";
var pat = /th/;
str.search(pat);     // == 8, start of th in three
str.search(/t/);     // == 4, start of t in two
str.search(/Four/i); // == 14, start of four
```

String slice()

SYNTAX: `string.slice(start[, end])`

WHERE: start - index from which to start.
end - index at which to end.

RETURN: string - a substring (not a String object) consisting of the characters.

SEE: String substring()

DESCRIPTION: This method is very similar to `String substring()`, in that it returns a substring from one index to another. The only difference is that if either `start` or end is negative, then it is treated as `length + start` or `length + end`. If either exceeds the bounds of the string, then either 0 or the length of the string is used instead.

String split()

SYNTAX: `string.split([delimiterString])`

WHERE: delimiterString - character, string or regular expression where the string is split. If `substring` is not specified, an array will be returned with the name of the `string` specified. Essentially this will mean that the string is split character by character.

RETURN: object - if no delimiters are specified, returns an array with one element which is the original string.

DESCRIPTION: This method splits a string into an array of strings based on the delimiters in the parameter delimiterString. The parameter delimiterString is optional and if supplied, determines where the string is split.

SEE: Array join()

EXAMPLE:

```
/*
For example, to create an array of all
of the words in a sentence, use code similar
to the following fragment:
*/

var sentence = "I am not a crook";
var wordArray = sentence.split(' ');
```

String substr()

SYNTAX: `string.substr(start, length)`

WHERE: start - integer specifying the position within the string to begin the desired substring. If start is positive, the position is relative to the beginning of the string. If start is

negative, the position is relative to the end of the string.
length - the length, in characters, of the substring to extract.

RETURN: string - a substring starting at position start and including the next number of characters specified by length.

DESCRIPTION: This method gets a section of a string. The start parameter is the first character in the new string. The length parameter determines how many characters to include in the new substring.
This method, `substr()` differs from `String substring()` in two basic ways. One, in substring() the start position cannot be negative, that is, it must be 0 or greater. Two, the second parameter in `substring()` indicates a position to go to, not the length of the new substring.

SEE: String substring()

EXAMPLE:

```
var str = ("0123456789");
str.substr(0, 5)      // == "01234"
str.substr(2, 5)      // == "23456"
str.substr(-4, 2)     // == "56"
```

String substring()

SYNTAX: `string.substring(start, end)`

WHERE: start - integer specifying the position within the string to begin the desired substring.
end - integer specifying the position within the string to end the desired substring. This integer must be one greater than the desired end position to allow for the terminating `null` byte.

RETURN: string - a substring starting at position start and going to but not including position end.

DESCRIPTION: This method retrieves a section of a string. The start parameter is the index or position of the first character to include. The end parameter marks the end of the string. The end position is the index or position after the last character to be included. The length of the substring retrieved is defined by end minus start. Another way to think about the start and end positions is that end equals start plus the length of the substring desired.

SEE: String charAt(), String indexOf(), String lastIndexOf(), String slice(), String substr()

EXAMPLE:

```
// For example, to get the first nine characters
// in string, use a Start position
// of 0 and add 9 to it, that is,
// "0 + 9", to get the End position
// which is 9. The following fragment illustrates.

var str = "0123456789";
str.substring(0, 5)    // == "01234"
str.substring(2, 5)    // == "234"
str.substring(0, 10)   // == "0123456789"
```

String toLocaleLowerCase()

SYNTAX: `string.toLocaleLowerCase()`

RETURN: string - a copy of a string with each character converted to lower case.

DESCRIPTION: This method behaves exactly the same as `String toLowerCase()`. It is designed to convert the string to lower case in a locale sensitive manner, though this functionality is currently unavailable. Once it is implemented, this function may behave differently for some locales (such as Turkish), though for the majority it will be identical to `toLowerCase()`.

SEE: String toLowerCase(), String toLocaleUpperCase()

String toLocaleUpperCase()

SYNTAX: `string.toLocaleUpperCase()`

RETURN: string - a copy of a string with each character converted to upper case.

DESCRIPTION: This method behaves exactly the same as `String toUpperCase()`. It is designed to convert the string to upper case in a locale sensitive manner, though this functionality is currently unavailable. Once it is implemented, this function may behave differently for some locales (such as Turkish), though for the majority it will be identical to `toUpperCase()`.

SEE: String toUpperCase(), String toLocaleLowerCase()

String toLowerCase()

SYNTAX: `string.toLowerCase()`

RETURN: string - copy of a string with all of the letters changed to lower case.

DESCRIPTION: This method changes the case of a string.

SEE: String toUpperCase(), String toLocaleLowerCase()

EXAMPLE:

```
var string = new String("Hello, World!");
string.toLowerCase()

// This will return the string "hello, world!".
```

String toUpperCase()

SYNTAX: `string.toUpperCase()`

RETURN: string - a copy of a string with all of the letters changed to upper case.

DESCRIPTION: This method changes the case of a string.

SEE: String toLowerCase(), String toLocaleUpperCase()

EXAMPLE:

```
var string = new String("Hello, World!");
string.toUpperCase()

// This will return the string
//  "HELLO, WORLD!".
```

3.12.5 String object static methods

String.fromCharCode()

SYNTAX: `string.fromCharCode(chrCode[, ...])`

WHERE: chrCode - character code, or list of codes, to be converted.

RETURN: string - string created from the character codes that are passed to it as parameters.

DESCRIPTION: The identifier String is used with this static method, instead of a variable name as with instance methods. The arguments passed to this method are assumed to be unicode characters.

SEE: String(), String charCodeAt()

EXAMPLE:

```
// The following code:
var string = String.fromCharCode(0x0041,0x0042)
// will set the variable string to be "AB".
```

Function index

Trademarks

CER and FieldCommander are registered trademarks of CER International bv.

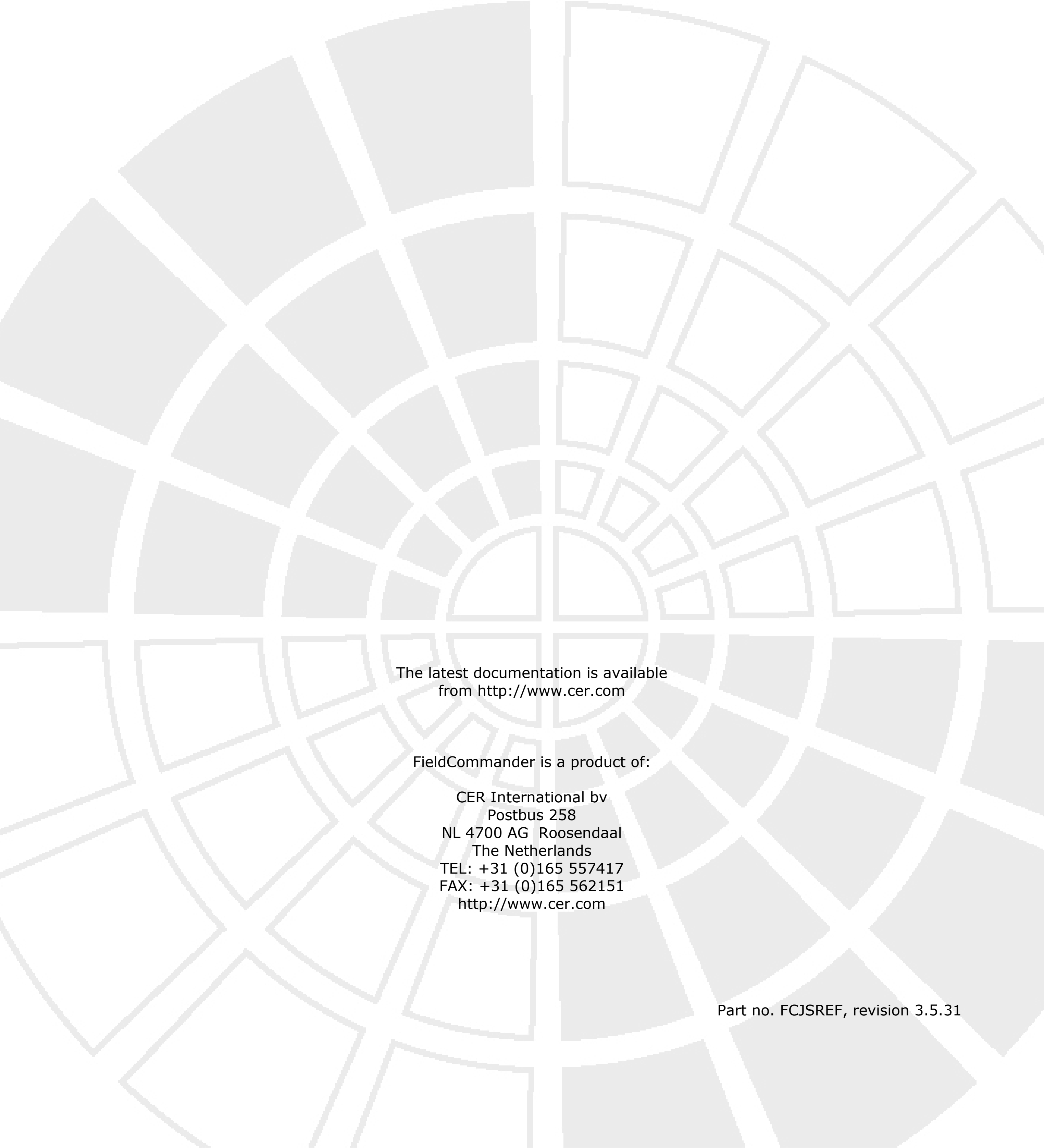
All other product names and services identified throughout this book are trademarks or registered trademarks of their respective companies. They are used throughout this manual in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with this manual.

Copyrights

No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior written permission of CER International bv, except as permitted by the Copyright Act of 1976 and except that program listings may be entered, stored and executed in a computer system.

THE INFORMATION AND MATERIAL CONTAINED IN THIS MANUAL ARE PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY WARRANTY CONCERNING THE ACCURACY, ADEQUACY, OR COMPLETENESS OF SUCH INFORMATION OR MATERIAL OR THE RESULT TO BE OBTAINED FROM USING SUCH INFORMATION OR MATERIAL, NEITHER CER INTERNATIONAL BV NOR THE AUTHORS SHALL BE RESPONSIBLE FOR ANY CLAIMS ATTRIBUTABLE TO ERRORS, OMISSIONS, OR OTHER INACCURACIES IN THE INFORMATION OR MATERIAL CONTAINED IN THIS MANUAL, AND IN NO EVENT SHALL CER INTERNATIONAL BV OR THE AUTHORS BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF SUCH INFORMATION OR MATERIAL.

from Device to Enterprise



The latest documentation is available
from http://www.cer.com

FieldCommander is a product of:

CER International bv
Postbus 258
NL 4700 AG Roosendaal
The Netherlands
TEL: +31 (0)165 557417
FAX: +31 (0)165 562151
http://www.cer.com

Part no. FCJSREF, revision 3.5.31