SIEBEL eBusiness 7

SIEBEL *e*SCRIPT LANGUAGE REFERENCE

VERSION 7.5, REV. B

12-FAUN9W

MARCH 2003

Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404
Copyright © 2003 Siebel Systems, Inc.
All rights reserved.
Printed in the United States of America

No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior agreement and written permission of Siebel Systems, Inc.

Siebel, the Siebel logo, TrickleSync, TSQ, Universal Agent, and other Siebel product names referenced herein are trademarks of Siebel Systems, Inc., and may be registered in certain jurisdictions.

Other product names, designations, logos, and symbols may be trademarks or registered trademarks of their respective owners.

U.S. GOVERNMENT RESTRICTED RIGHTS. Programs, Ancillary Programs and Documentation, delivered subject to the Department of Defense Federal Acquisition Regulation Supplement, are “commercial computer software” as set forth in DFARS 227.7202, Commercial Computer Software and Commercial Computer Software Documentation, and as such, any use, duplication and disclosure of the Programs, Ancillary Programs and Documentation shall be subject to the restrictions contained in the applicable Siebel license agreement. All other use, duplication and disclosure of the Programs, Ancillary Programs and Documentation by the U.S. Government shall be subject to the applicable Siebel license agreement and the restrictions contained in subsection (c) of FAR 52.227-19, Commercial Computer Software - Restricted Rights (June 1987), or FAR 52.227-14, Rights in Data—General, including Alternate III (June 1987), as applicable. Contractor/licensor is Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404.

Proprietary Information
Siebel Systems, Inc. considers information included in this documentation and in Siebel eBusiness Applications Online Help to be Confidential Information. Your access to and use of this Confidential Information are subject to the terms and conditions of: (1) the applicable Siebel Systems software license agreement, which has been executed and with which you agree to comply; and (2) the proprietary and restricted rights notices included in this documentation.

Contents

Introduction

Typographic Conventions 16

Revision History 17

Chapter 1. Quick Reference: Methods and Properties

Array Methods 20

Buffer Methods 21

Character Classification Methods 22

Conversion or Casting Methods 23

Data Handling Methods 24

Date and Time Functions 25

Disk and File Functions 28

Disk and Directory Functions 28

File Control Functions 28

File-Manipulation Functions 29

Error Handling Methods 30

Math Methods 31

Numeric Functions 31

Trigonometric Functions 32

Math Properties 32

Operating System Interaction Methods 34

String and Byte-Array Methods 35

Miscellaneous Methods 37

Chapter 2. Siebel eScript Language Overview

Siebel eScript Programming Guidelines 40
Basic Siebel eScript Concepts 42
Case Sensitivity 42
White-Space Characters 43
Comments 44
Expressions, Statements, and Blocks 45
Identifiers 46
Variables 47
Data Types 50
Primitive Data Types 51
Composite Data Types 54
Special Data Types 56
Number Constants 57
Automatic Type Conversion 57
Properties and Methods of Basic Data Types 58
Expressions 60
Operators 61
Mathematical Operators 61
Bit Operators 63
Logical Operators and Conditional Expressions 64
Typeof Operator 66
Conditional Operator 67
String Concatenation Operator 67
Functions 69
Function Scope 70
Passing Variables to Functions 70
The Function Arguments[] Property 71
Function Recursion 71
Error Checking for Functions 73

eScript Statements 74
break Statement 74
continue Statement 75
do...while Statement 76
for Statement 77
for...in Statement 78
goto Statement 79
if Statement 80
switch Statement 83
throw Statement 84
try Statement 85
while Statement 88
with Statement 89

Chapter 3. Siebel eScript Commands

Applet Objects 92
The Application Object 94
Array Objects 97
The Array Constructor 98
join() Method 99
length Property 99
reverse() Method 100
sort() Method 101
BLOB Objects 103
The blobDescriptor Object 103
Blob.get() Method 105
Blob.put() Method 105
Blob.size() Method 107
Buffer Objects 109
The Buffer Constructor 110
Properties 111

Methods 112
bigEndian Property 112
cursor Property 113
data Property 113
getString() Method 114
getValue() Method 114
offset[] Method 115
putString() Method 116
putValue() Method 117
size Property 119
subBuffer() Method 120
toString() Method 120
unicode Property 121

Business Component Objects 122

Business Object Objects 128

Business Service Objects 129

The Clib Object 131

Redundant Functions in the Clib Object 131
File I/O Functions 133
The Time Object 135
Time Functions 136
Character Classification 136
Formatting Data 137
Clib.asctime() Method 141
Clib.bsearch() Method 141
Clib.chdir() Method 143
Clib.clearerr() Method 144
Clib.clock() Method 144
Clib.cosh() Method 144
Clib.ctime() Method 145
Clib.difftime() Method 146

Clib.div() Method and Clib.ldiv() Method 146
Clib.errno Property 147
Clib.fclose() Method 148
Clib.feof() Method 148
Clib.ferror() Method 149
Clib.fflush() Method 149
Clib.fgetc() Method and Clib.getc() Method 150
Clib.fgetpos() Method 151
Clib.fgets() Method 152
Clib.flock() Method 153
Clib.fopen() Method 154
Clib.fprintf() Method 157
Clib.fputc() Method and Clib.putc() Method 157
Clib.fputs() Method 158
Clib.fread() Method 158
Clib.freopen() Method 160
Clib.frexp() Method 161
Clib.fscanf() Method 162
Clib.fseek() Method 163
Clib.fsetpos() Method 163
Clib.ftell() Method 164
Clib.fwrite() Method 164
Clib.getcwd() Method 166
Clib.getenv() Method 167
Clib.gmtime() Method 168
Clib.isalnum() Method 169
Clib.isalpha() Method 169
Clib.isascii() Method 170
Clib.iscntrl() Method 170
Clib.isdigit() Method 170
Clib.isgraph() Method 171

Clib.islower() Method 171
Clib.isprint() Method 172
Clib.ispunct() Method 173
Clib.isspace() Method 173
Clib.isupper() Method 174
Clib.isxdigit() Method 174
Clib.ldexp() Method 175
Clib.localtime() Method 175
Clib.memchr() Method 176
Clib.memcmp() Method 177
Clib.memcpy() Method and Clib.memmove() Method 177
Clib.memset() Method 178
Clib.mkdir() Method 178
Clib.mktime() Method 179
Clib.modf() Method 179
Clib.perror() Method 180
Clib.putenv() Method 181
Clib.qsort() Method 182
quot Method 183
Clib.rand() Method 183
rem Method 184
Clib.remove() Method 185
Clib.rename() Method 185
Clib.rewind() Method 186
Clib.rmdir() Method 186
Clib.rsprintf() Method 187
Clib.sinh() Method 188
Clib.sprintf() Method 188
Clib.srand() Method 189
Clib.sscanf() Method 190
Clib.strchr() Method 191

Clib.stricmp() Method and Clib.strcmpi() Method 192
Clib.strcspn() Method 192
Clib.strerror() Method 194
Clib.strftime() Method 195
Clib.strlwr() Method 196
Clib.strncat() Method 197
Clib.strncmp() Method 198
Clib.strncmpi() Method and Clib.strnicmp() Method 198
Clib.strncpy() Method 199
Clib.strpbrk() Method 200
Clib.strrchr() Method 201
Clib.strspn() Method 202
Clib.strstr() Method 203
Clib.strstri() Method 204
Clib.system() Method 205
Clib.tanh() Method 206
Clib.time() Method 206
Clib.tmpfile() Method 207
Clib.tmpnam() Method 208
Clib.toascii() Method 208
Clib.ungetc()Method 209

The Date Object 210
The Date Constructor 210
Universal Time Functions 212
GetDate() Method 212
Date.fromSystem() Static Method 213
getDay() Method 214
getFullYear() Method 215
getHours() Method 216
getMilliseconds() Method 216
getMinutes() Method 217

getMonth() Method 218
getSeconds() Method 218
getTime() Method 219
getTimezoneOffset() Method 220
getUTCDate() Method 221
getUTCDay() Method 221
getUTCFullYear() Method 222
getUTCHours() Method 223
getUTCMilliseconds() Method 224
getUTCMinutes() Method 224
getUTCMonth() Method 225
getUTCSeconds() Method 226
getYear() Method 226
Date.parse() Static Method 226
setDate() Method 227
setFullYear() Method 228
setHours() Method 229
setMilliseconds() Method 229
setMinutes() Method 231
setMonth() Method 231
setSeconds() Method 232
setTime() Method 232
setUTCDate() Method 234
setUTCFullYear() Method 234
setUTCHours() Method 235
setUTCMilliseconds() Method 236
setUTCMinutes() Method 237
setUTCMonth() Method 238
setUTCSeconds() Method 239
setYear() Method 239
toGMTString() Method 240

toLocaleString() Method and toString() Method 241

Date.toSystem() Method 242

toUTCString() Method 242

Date.UTC() Static Method 243

The Exception Object 245

Function Objects 246

The Global Object 249

Global Functions Unique to Siebel eScript 249

Conversion or Casting Functions 250

COMCreateObject() Method 251

CORBACreateObject() Method 252

defined() Method 255

escape() Method 256

eval() Method 257

getArrayLength() Method 258

isNaN() Method 259

isFinite() Method 259

parseFloat() Method 260

parseInt() Method 261

setArrayLength() Method 261

ToBoolean() Method 262

ToBuffer() Method 263

ToBytes() Method 264

ToInt32() Method 265

ToInteger() Method 266

ToNumber() Method 267

ToObject() Method 268

ToString() Method 268

ToUint16() Method 269

ToUint32() Method 270

undefine() Method 271

unescape(string) Method 272

The Math Object 274

Math.abs() Method 275

Math.acos() Method 275

Math.asin() Method 276

Math.atan() Method 277

Math.atan2() Method 278

Math.ceil() Method 279

Math.cos() Method 280

Math.E Property 281

Math.exp() Method 281

Math.floor() Method 282

Math.LN10 Property 282

Math.LN2 Property 283

Math.log() Method 284

Math.LOG2E Property 284

Math.LOG10E Property 285

Math.max() Method 286

Math.min() Method 286

Math.PI Property 287

Math.pow() Method 287

Math.random() Method 288

Math.round() Method 289

Math.sin() Method 290

Math.sqrt() Method 290

Math.SQRT1_2 Property 291

Math.SQRT2 Property 291

Math.tan() Method 292

User-Defined Objects 293

Predefining Objects with Constructor Functions 293

Assigning Functions to Objects 294

Object Prototypes 295

Property Set Objects 298

The SElib Object 300

String Objects 303

The String as Data Type 303

Escape Sequences for Characters 303

Single Quote Strings 304

Back-Quote Strings 304

The String as Object 305

charAt() Method 305

String.fromCharCode() Static Method 306

indexOf() Method 306

lastIndexOf() Method 307

length Property 308

split() Method 309

string.replace() Method 311

substring() Method 312

toLowerCase() Method 313

toUpperCase() Method 314

Index

Introduction

Siebel eScript is an enhanced configuration environment that includes:

- A fully functional procedural programming language
- An editing environment to create and maintain custom Siebel eScript routines
- A debugger to assist in detecting errors in Siebel eScript routines
- A compiler for the custom Siebel eScript routines
- A run-time engine (similar to a JavaScript interpreter) to process the custom Siebel eScript routines

The topics in this guide explain the Siebel eScript programming language, which is embedded in Siebel Tools. You can use Siebel eScript to create scripts that automate a variety of daily tasks.

This book will be useful primarily to people whose title or job description matches the following:

Siebel Application Developers	Persons who plan, implement, and configure Siebel applications, possibly adding new functionality.

Programmers with experience in other languages can use this and the related volumes to become proficient in Siebel eScript. Those with no programming experience should turn to other sources for basic information about programming.

Typographic Conventions

Because Siebel eScript is a case-sensitive language, the language’s capitalization conventions are followed; however, the syntax diagrams use the conventions shown in Table 1.

Table 1. Typographic Conventions

To Represent	Help Syntax Is
Instantiated objects	Lowercase italics; an internal capital may be used to indicate multiple English words: *stringVar, blobVar, dateVar*
Arguments to statements or functions	Lowercase, italics; an internal capital may be used to indicate multiple English words: *variable, number, intVar*
Optional arguments or characters	Arguments or characters in brackets: [, *caption*], [*type*], [*arg1, arg2, …, arg*n]

Revision History

Siebel eScript Language Reference, Version 7.5, Rev. B

March 2003 Bookshelf

Table 2. Changes Made in Rev. B for March 2003 Bookshelf

Topic	Revision
"CORBACreateObject() Method" on page 252	Added note about support for methods with out or in/out parameters.
"parseInt() Method" on page 261	Added information on the parseInt method.
"The Buffer Constructor" on page 110	Added an additional syntax.

Additional Changes:

- Revised examples to correct syntax errors.

November 2002 Bookshelf

Table 3. Changes Made in Rev. A for November 2002 Bookshelf

Topic	Revision
"Clib.bsearch() Method" on page 141	Repaired syntax error in the example.

Quick Reference: Methods and Properties 1

The links that follow provide access to a list of Siebel eScript functions, methods, and properties by functional group, rather than by object. Properties can be distinguished from methods by the fact that they do not end with a pair of parentheses.

- "Array Methods" on page 20
- "Buffer Methods" on page 21
- "Character Classification Methods" on page 22
- "Conversion or Casting Methods" on page 23
- "Data Handling Methods" on page 24
- "Date and Time Functions" on page 25
- "Disk and File Functions" on page 28
- "Error Handling Methods" on page 30
- "Math Methods" on page 31
- "Operating System Interaction Methods" on page 34
- "String and Byte-Array Methods" on page 35
- "Miscellaneous Methods" on page 37

Array Methods

The following is a list of array methods.

Method or Property	Function
getArrayLength()	Determines size of an array
Array.join()	Creates a string from array elements
Array.length	Returns the length of an array
setArrayLength()	Sets the size of an array
Array.sort()	Sorts array elements
Array.reverse()	Reverses the order of elements of an array

Buffer Methods

The following is a list of buffer methods.

Method or Property	Function
bufferVar.bigEndian	Stores a Boolean flag for bigEndian byte ordering
bufferVar.cursor	Stores the current position of the buffer cursor
bufferVar.data	Refers to the internal data of a buffer
bufferVar.getString()	Returns a string starting from the current cursor position
bufferVar.getValue()	Returns a value from a specified position
bufferVar.putString()	Puts a string into a buffer
bufferVar.putValue()	Puts a specified value into a buffer
bufferVar.size	Stores the size of a buffer object
bufferVar.subBuffer()	Returns a section of a buffer
bufferVar.toString()	Returns a string equivalent of the current state of a buffer
bufferVar.unicode	Stores a Boolean flag for the use of unicode strings

Character Classification Methods

The following is a list of character classification methods.

Method	Function
Clib.isalnum()	Tests for an alphanumeric character
Clib.isalpha()	Tests for an alphabetic character
Clib.isascii()	Tests for an ASCII-coded character
Clib.iscntrl()	Tests for any control character
Clib.isdigit()	Tests for any decimal-digit character
Clib.isgraph()	Tests for any printing character except space
Clib.islower()	Tests for a lowercase alphabetic letter
Clib.isprint()	Tests for any printing character
Clib.ispunct()	Tests for a punctuation character
Clib.isspace()	Tests for a white-space character
Clib.isupper()	Tests for an uppercase alphabetic character
Clib.isxdigit()	Tests for a hexadecimal-digit character

Conversion or Casting Methods

The following is a list of conversion or casting methods.

Method	Function
escape()	Escapes special characters in a string
parseFloat()	Converts a string to a float
parseInt()	Converts a string to an integer
ToBoolean()	Converts a value to a Boolean
ToBuffer()	Converts a value to a buffer
ToBytes()	Converts a value to a buffer (raw transfer)
ToInt32()	Converts a value to a large integer
ToInteger()	Converts a value to an integer
ToNumber()	Converts a value to a number
ToObject()	Converts a value to an object
ToPrimitive()	Converts a value to a primitive
ToString()	Converts a value to a string
ToUint16()	Converts a value to an unsigned integer
ToUint32()	Converts a value to an unsigned large integer
unescape()	Removes escape sequences in a string

Data Handling Methods

The following is a list of data handling methods.

Method	Function
Blob.get()	Reads data from a specified location in a BLOB
Blob.put()	Writes data into a specified location in a BLOB
Blob.size()	Determines the size of a BLOB
defined()	Tests if a variable has been defined
isFinite()	Determines whether a value is finite
isNaN()	Determines whether a value is Not a Number (NaN)
toString()	Converts any variable to a string representation
undefine()	Makes a variable undefined

Date and Time Functions

The following is a list of date and time functions.

Method	Function
Clib.asctime()	Converts a date-time to an ASCII string
Clib.clock()	Gets the processor time
Clib.ctime()	Converts a date-time to an ASCII string
Clib.difftime()	Computes the difference between two times
dateVar.getDate()	Returns the day of the month
dateVar.getDay()	Returns the day of the week
dateVar.getFullYear()	Returns the year as a four-digit number
dateVar.getHours()	Returns the hour
dateVar.getMilliseconds()	Returns the millisecond
dateVar.getMinutes()	Returns the minute
dateVar.getMonth()	Returns the month
dateVar.getSeconds()	Returns the second
dateVar.getTime()	Returns the date-time, in milliseconds, of a Date object
dateVar.getTimezoneOffset()	Returns the difference, in minutes, from GMT
dateVar.getUTCDate()	Returns the UTC day of the month
dateVar.getUTCDay()	Returns the UTC day of the week
dateVar.getUTCFullYear()	Returns the UTC year as a four-digit number
dateVar.getUTCHours()	Returns the UTC hour
dateVar.getUTCMilliseconds()	Returns the UTC millisecond
dateVar.getUTCMinutes()	Returns the UTC minute
Clib.gmtime()	Converts a date-time to GMT
Clib.localtime()	Converts a date-time to a structure

Method	Function
Clib.mktime()	Converts a time structure to calendar time
Clib.strftime()	Writes a formatted date-time to a string
Clib.time()	Gets the current time
dateVar.getUTCMonth()	Returns the UTC month
dateVar.getUTCSeconds()	Returns the UTC second
dateVar.getYear()	Returns the year as a two-digit number
dateVar.setDate()	Sets the day of the month
dateVar.setFullYear()	Sets the year as a four-digit number
dateVar.setHours()	Sets the hour
dateVar.setMilliseconds()	Sets the millisecond
dateVar.setMinutes()	Sets the minute
dateVar.setMonth()	Sets the month
dateVar.setSeconds()	Sets the second
dateVar.setTime()	Sets the date-time in a Date object, in milliseconds
dateVar.setUTCDate()	Sets the UTC day of the month
dateVar.setUTCFullYear()	Sets the UTC year as a four-digit number
dateVar.setUTCHours()	Sets the UTC hour
dateVar.setUTCMilliseconds()	Sets the UTC millisecond
dateVar.setUTCMinutes()	Sets the UTC minute
dateVar.setUTCMonth()	Sets the UTC month
dateVar.setUTCSeconds()	Sets the UTC second
dateVar.setYear()	Sets the year as a two-digit number
dateVar.toGMTString()	Converts a Date object to a string
dateVar.toLocaleString()	Returns a string for local date and time
Date.toSystem()	Converts a Date object to a system time

Method	Function
dateVar.toUTCString()	Returns a string that represents the UTC date
Date.fromSystem()	Converts system time to Date object time
Date.parse()	Converts a Date string to a Date object
dateVar.UTC()	Returns the date-time, in milliseconds from January 1, 1970, of its parameters

Disk and File Functions

The eScript language provides the following disk and file functions.

- "Disk and Directory Functions" on page 28
- "File Control Functions" on page 28
- "File-Manipulation Functions" on page 29

Disk and Directory Functions

The following is a list of disk and directory functions.

Method	Function
Clib.chdir()	Changes directory
Clib.flock()	Handles file locking and unlocking
Clib.getcwd()	Gets the current working directory
Clib.mkdir()	Creates a directory
Clib.rmdir()	Removes a directory

File Control Functions

The following is a list of file control controls.

Method	Function
Clib.fclose()	Closes an open file
Clib.fopen()	Opens a file
Clib.freopen()	Assigns a new file spec to a file handle
Clib.remove()	Deletes a file
Clib.rename()	Renames a file

Method	Function
Clib.tmpfile()	Creates a temporary binary file
Clib.tmpnam()	Gets a temporary filename

File-Manipulation Functions

The following is a list of file manipulation functions.

Method	Function
Clib.feof()	Tests whether at the end of a file stream
Clib.fflush()	Flushes the stream of one or more open files
Clib.fgetc()	Gets a character from a file stream
Clib.fgetpos()	Gets the current file cursor position in a file stream
Clib.fgets()	Gets a string from an input stream
Clib.fprintf()	Writes formatted output to a file stream
Clib.fputc()	Writes a character to a file stream
Clib.fputs()	Writes a string to a file stream
Clib.fscanf()	Gets formatted input from a file stream
Clib.fread()	Reads data from a file
Clib.fseek()	Sets the file cursor position in an open file stream
Clib.fsetpos()	Sets the file cursor position in a file stream
Clib.ftell()	Gets the current value of the file cursor
Clib.fwrite()	Writes data to a file
Clib.getc()	Gets a character from a file stream
Clib.putc()	Writes a character to a file stream
Clib.rewind()	Resets the file cursor to the beginning of a file
Clib.ungetc()	Pushes a character back to the input stream

Error Handling Methods

The following is a list of error handling methods.

Method	Function
Clib.clearerr()	Clears end-of-file and error status of a file
Clib.errno()	Returns the value of an error condition
Clib.ferror()	Tests for an error on a file stream
Clib.perror()	Prints a message describing an error number
Clib.strerror()	Gets a string describing an error number

Math Methods

The eScript language provides the following math methods.

- "Numeric Functions" on page 31
- "Trigonometric Functions" on page 32
- "Math Properties" on page 32

Numeric Functions

The following is a list of numeric functions.

Method	Function
Math.abs()	Returns the absolute value of an integer
Math.ceil()	Rounds up
Clib.div()	Integer division, returns quotient and remainder
Math.exp()	Computes the exponential function
Math.floor()	Rounds down
Clib.frexp()	Breaks into a mantissa and an exponential power of 2
Clib.ldexp()	Calculates mantissa * 2 ^ exp
Clib.ldiv()	Integer division, returns quotient and remainder
Math.log()	Calculates the natural logarithm
Math.max()	Returns the largest of one or more values
Math.min()	Returns the smallest of one or more values
Clib.modf()	Splits a value into integer and fractional parts
Math.pow()	Calculates *x* to the power of *y*
Clib.rand()	Generates a random number
Math.random()	Returns a random number

Method	Function
Math.round()	Rounds a value up or down
Math.sqrt()	Calculates the square root
Clib.srand()	Seeds the random number generator

Trigonometric Functions

The following is a list of trigonometric functions.

Method	Function
Math.acos()	Calculates the arc cosine
Math.asin()	Calculates the arc sine
Math.atan()	Calculates the arc tangent
Math.atan2()	Calculates the arc tangent of a fraction
Math.cos()	Calculates the cosine
Clib.cosh()	Calculates the hyperbolic cosine
Math.sin()	Calculates the sine
Clib.sinh()	Calculates the hyperbolic sine
Math.tan()	Calculates the tangent
Clib.tanh()	Calculates the hyperbolic tangent

Math Properties

The following is a list of math properties.

Property	Value
Math.E	Value of *e*, the base for natural logarithms
Math.LN10	Value of the natural logarithm of 10

Property	Value
Math.LN2	Value of the natural logarithm of 2
Math.LOG2E	Value of the base 2 logarithm of *e*
Math.LOG10E	Value of the base 10 logarithm of *e*
Math.PI	Value of pi
Math.SQRT1_2	Value of the square root of ½
Math.SQRT2	Value of the square root of 2

Operating System Interaction Methods

The following is a list of operating system interaction methods.

Method	Function
Clib.getenv()	Gets the value of an environment string
Clib.putenv()	Creates an environment string and assigns a value to it
Clib.system()	Passes a command to the command processor

String and Byte-Array Methods

The following is a list of string and byte-array methods.

Method	Function
stringVar.charAt()	Returns a character in a string
stringVar.charCodeAt()	Returns a unicode character in a string
String.fromCharCode()	Creates a string from character codes
stringVar.indexOf()	Returns the index of the first substring in a string
stringVar.lastIndexOf()	Returns the index of the last substring in a string
Clib.memchr()	Searches a byte array
Clib.memcmp()	Compares two byte arrays
Clib.memcpy()	Copies from one byte array to another
Clib.memmove()	Moves from one byte array to another
Clib.memset()	Copies from one byte array to another
Clib.rsprintf()	Returns a formatted string
stringVar.split()	Splits a string into an array of strings
Clib.sprintf()	Writes formatted output to a string
Clib.sscanf()	Reads and formats input from a string
Clib.strcat()	Concatenates strings
Clib.strchr()	Searches a string for a character
Clib.strcmp()	Makes a case-sensitive comparison of two strings
Clib.strcmpi()	Makes a case-insensitive comparison of two strings
Clib.strcpy()	Copies one string to another
Clib.strcspn()	Searches a string for the first character in a set of characters
Clib.stricmp()	Makes a case-insensitive comparison of two strings
Clib.strlen()	Gets the length of a string

Method	Function
Clib.strlwr()	Converts a string to lowercase
Clib.strncat()	Concatenates a portion of one string to another
Clib.strncmp()	Makes a case-sensitive comparison of parts of two strings
Clib.strncmpi()	Makes a case-insensitive comparison of parts of two strings
Clib.strncpy()	Copies a portion of one string to another
Clib.strnicmp()	Makes a case-insensitive comparison of parts of two strings
Clib.strpbrk()	Searches string for a character from a set of characters
Clib.strrchr()	Searches a string for the last occurrence of a character
Clib.strspn()	Searches a string for a character not in a set of characters
Clib.strstr()	Searches a string for a substring (case-sensitive)
Clib.strstri()	Searches a string for a substring (case-insensitive)
stringVar.substring()	Retrieves a section of a string
Clib.toascii()	Converts to ASCII
Clib.tolower()	Converts to lowercase
stringVar.toLowerCase()	Converts a string to lowercase
stringVar.toUpperCase()	Converts a string to uppercase

Miscellaneous Methods

The following is a list of miscellaneous methods.

Method	Function
Clib.atexit()	Sets a function to be called at program exit
Clib.bsearch()	Does a binary search for a member of a sorted array
Clib.qsort()	Sorts an array; may use comparison function

Siebel eScript Language Overview 2

Siebel eScript is a scripting or programming language that application developers use to write simple scripts to extend Siebel applications. JavaScript, a popular scripting language used primarily on Web sites, is its core language.

Siebel eScript is ECMAScript compliant. ECMAScript is the standard implementation of JavaScript as defined by the ECMA-262 standard.

Siebel eScript provides access to local system calls through two objects, Clib and SElib, so you can use C-style programming calls to certain parts of the local operating system. This allows programmers to write files to the local hard disk and perform other tasks that standard JavaScript cannot.

"Siebel eScript Programming Guidelines" on page 40

"Basic Siebel eScript Concepts" on page 42

"Data Types" on page 50

"Expressions" on page 60

"Operators" on page 61

"Functions" on page 69

"eScript Statements" on page 74

Siebel eScript Programming Guidelines

If you have never programmed in JavaScript before, you should start with a general-purpose JavaScript reference manual. You need to understand how JavaScript handles objects before you can program using the Siebel eScript.

Declare your variables. Standard ECMAScript does not require that you declare variables. Variables are declared implicitly as soon as they are used. However, Siebel eScript requires you to declare variables with the `var` keyword. Declare variables used in a module before you use them, because this makes it easier for others to understand your code and for you to debug the code. The only exception to this standard is declaring a variable inside a loop controller, which restricts the scope of that reference to the loop. This prevents the accumulation of unwanted values.

Pay attention to case. Be aware that Siebel eScript is case-sensitive. Therefore, if you instantiate an object using the variable name *SiebelApp*, for example, eScript does not find that object if the code references it as *siebelapp* or *SIEBELAPP* instead of *SiebelApp*. Case sensitivity also applies to method names and other parts of Siebel eScript.

Use parentheses () with functions. Siebel eScript functions, like those in standard JavaScript, require trailing parentheses () even when they have no parameters.

Use four-digit years in dates. Siebel applications and the ECMA-262 Standard handle two-digit years differently. Siebel applications assume that a two-digit year refers to the appropriate year between 1950 and 2049. The ECMA-262 Standard assumes that a two-digit year refers to a year between 1900 and 1999, inclusive. If your scripts do not enforce four-digit date entry and use four-digit dates, your users may unintentionally enter the wrong century when performing a query or creating or updating a record.

(BusComp) methods GetFormattedFieldValue() and SetFormattedFieldValue() are examples of Y2K sensitivities in Siebel eScript that use two-digit dates. If you use these methods in a script, users requesting orders for the years from 00 to 02 may find that they have retrieved orders for the years 1900–1902 (probably an empty list), instead of 2000–2002, which was what they wanted.

If you use only four-digit dates in your programs, you will not have Y2K problems with your scripts. With the preceding example, you could use GetFieldValue() and SetFieldValue(), which require dates to be specified using the canonical Siebel format (MM/DD/YYYY), instead of GetFormattedFieldValue() and SetFormattedFieldValue().

The this object reference. The special object reference *this* is eScript shorthand for "the current object." You should use *this* in place of references to active business objects and components. For example, in a business component event handler, you should use *this* in place of *ActiveBusComp,* as shown in the following example:

```
function BusComp_PreQuery ()
{
   this.ActivateField("Account");
   this.ActivateField("Account Location");
   this.ClearToQuery();
   this.SetSortSpec( "Account(DESCENDING)," +
      " Account Location(DESCENDING)");
   this.ExecuteQuery();

   return (ContinueOperation);
}
```

Make effective use of the Switch construct. The Switch construct directs the program to choose among any number of alternatives you require, based on the value of a single variable. This is greatly preferable to a series of nested If statements because it simplifies code maintenance. It also improves performance, because the variable must be evaluated only once.

Basic Siebel eScript Concepts

Standard JavaScript, or ECMAScript, is usually part of Web browsers and is therefore used while users are connected to the Internet. Most people are unaware that JavaScript is being executed on their computers when they are connected to various Internet sites.

Siebel eScript is implemented as part of Siebel applications. You do not need a Web browser to use it. It also contains a number of functions that do not exist in ECMAScript. These functions give you access to the hard disk and other parts of the Siebel client workstation or server.

"Case Sensitivity" on page 42

"White-Space Characters" on page 43

"Comments" on page 44

"Expressions, Statements, and Blocks" on page 45

"Identifiers" on page 46

"Variables" on page 47

Case Sensitivity

Siebel eScript 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"
```

Identifiers in Siebel eScript are case-sensitive. For example, to raise an error from the server, the TheApplication().RaiseErrorText method could be used:

```
TheApplication().RaiseErrorText("an error has occurred")
```

If you change the capitalization to

```
TheApplication().raiseerrortext("an error has occurred")
```

the Siebel eScript interpreter generates an error message.

Control statements are also case-sensitive. For example, the statement `while` is valid, but the statement `While` is not.

White-Space Characters

White-space characters (space, tab, carriage-return, and newline) govern the spacing and placement of text. White space makes code more readable for the users, but the Siebel eScript interpreter ignores it.

Lines of script end with a carriage-return character, and each line is usually a separate statement. (Technically, in many editors, lines end with a carriage-return and linefeed pair, `"\r\n"`.) Because the Siebel eScript interpreter usually sees one or more white-space characters between identifiers as simply white space, the following Siebel eScript statements are equivalent to one another:

```
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 spaces and not tabs, because tab size settings vary from editor to editor and programmer to programmer. If programmers use only spaces, the format of a script appears the same on every editor.

CAUTION: Siebel eScript treats white space in string literals differently from other white space. In particular, placing a line break within a string causes the Siebel eScript interpreter to treat the two lines as separate statements, both of which contain errors because they are incomplete. To avoid this problem, either keep string literals on a single line or create separate strings and associate them with the string concatenation operator.

For example:

```
var Gettysburg = "Fourscore and seven years ago, " +
"our fathers brought forth on this continent a " +
"new nation."
```

For more information about string concatenation, read “String Concatenation Operator” on page 67.

Comments

A comment is text in a script to be read by users and not by the Siebel eScript interpreter, which skips over comments. Comments that explain lines of code help users understand the purpose and program flow of a program, making it easier to alter code.

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 Siebel eScript 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.
The interpreter ignores it.
*/

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

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

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. Siebel eScript 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. If multiple statements are enclosed within 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();
}
```

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

Statements within blocks are often indented for easier reading.

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.

Rules for Identifiers

Siebel eScript identifiers follow these rules:

- Identifiers may use only uppercase or lowercase ASCII letters, digits, the underscore (_), and the dollar sign ($). They may use only characters from the following sets:

  ```
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz
  0123456789
  _$
  ```

- Identifiers may not use any of the following characters:

  ```
  +<>&|=!*/%^~?:{};()[].‘"'#,
  ```

- Identifiers must begin with a letter, underscore, or dollar sign, but they may have digits anywhere else.
- Identifiers may not have white space in them, because white space separates identifiers for the Siebel eScript interpreter.
- Identifiers have no built-in length restrictions, so you can make them as long as necessary.

The following identifiers, variables, and functions are valid:

```
George
Martha7436
annualReport
George_and_Martha_prepared_the_annualReport
$alice
CalculateTotal()
$SubtractLess()
_Divide$All()
```

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

```
1george
2nancy
this&that
Martha and Nancy
ratsAndCats?
=Total()
(Minus)()
Add Both Figures()
```

Prohibited Identifiers

The following words have special meaning for the Siebel eScript interpreter and cannot be used as identifiers:

```
break    case     catch    class    const  continue debugger
default  delete   do       else     enum   export   extends
false    finally  for      function if     import   in
new      null     return   super    switch this     throw
true     try      typeof   while    with   var      void
```

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 you want to display a name literally, you must use something like the following fragment multiple times:

```
TheApplication().RaiseErrorText("Aloysius Gloucestershire
Merkowitzky")
```

But you could use a variable to make this task easier, as in the following:

```
var Name = "Aloysius Gloucestershire Merkowitzy"
TheApplication().RaiseErrorText(Name)
```

The preceding method allows you to use shorter lines of code for display and to use the same lines of code repeatedly by changing the contents of the variable Name.

Variable Scope

Variables in Siebel eScript may be either global or local. Global variables can be accessed and modified from any function associated with the Siebel object for which the variables have been declared. Local variables can be accessed only within the functions in which they are created, because their *scope* is local to that function.

Variables can also be shared across modules. A variable declared outside a function has scope global to the module. If you declare a local variable with the same name as a module variable, the module variable is not accessible.

NOTE: Siebel eScript variables declared outside of a particular function are global only to their object (the module in which they are declared), not across every object in the application.

There are no absolute rules that indicate when global or local variables should be used. It is generally easier to understand how local variables are used in a single function than how global variables are used throughout an entire module. Therefore, local variables facilitate modular code that is easier to debug and to alter and develop over time. Local variables also use fewer resources.

Variable Declaration

To declare a variable, use the var keyword. To make it local, declare it in a function.

```
var perfectNumber;
```

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

```
var perfectNumber = 28;
```

In the following example, `a` is global to its object because it was declared outside of a function. The variables `b`, `c`, and `d` are local because they are defined within functions.

```
var a = 1;
function myFunction()
{
   var b = 1;
   var d = 3;
   someFunction(d);
}

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

The variable `c` may not be used in the `myFunction()` function, because it is has not been defined within the scope of that function. The variable `d` is used in the `myFunction()` 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:

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

Data Types

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

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 Siebel eScript interpreter, and the type remains until a later assignment changes the type automatically. The preceding example creates three different types of variables. The first is a number, the second is a string, and the third is a Boolean variable.

Because Siebel eScript 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.

- "Primitive Data Types" on page 51
- "Composite Data Types" on page 54
- "Special Data Types" on page 56
- "Number Constants" on page 57
- "Automatic Type Conversion" on page 57
- "Properties and Methods of Basic Data Types" on page 58

Primitive Data Types

Variables that have primitive data types pass their data by value. If an argument is passed by value, the variable used for that argument retains its value when the subroutine or function returns to the routine that called it (the caller). 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 or 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 in memory at one time.

The primitive data types are number, Boolean, and string.

Number

The number data type includes integers and floating-point numbers, which can be represented in one of several ways.

NOTE: Numbers that contain characters other than a decimal point are treated as string values. For example, eScript treats the number 100,000 (notice the comma) as a string.

Integer

Integers are whole numbers. Integer constants and literals can be expressed in decimal, hexadecimal, or octal notation. Decimal constants and literals are expressed by using the decimal representation. See the following two sections to learn how to express hexadecimal and octal integers.

Hexadecimal

Hexadecimal notation uses base 16 digits from the sets of 0–9 and A–F or a–f. These digits are preceded by 0x. Case sensitivity does not apply to hexadecimal notation in Siebel eScript. Examples are:

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

The decimal equivalents are:

```
1, 1, 256, 31, 31, 43981
var a = 6958
```

Octal

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

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

The decimal equivalents are:

```
0, 5, 63
var a = 99
```

Floating Point

Floating-point numbers are numbers with fractional parts that are indicated by decimal notation, such as 10.33. Floating-point numbers are often referred to as floats.

Decimal

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

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

Scientific

Scientists often use scientific notation to express very large or small numbers. It uses the decimal digits in conjunction with exponential notation, represented by e or E. Scientific notation is also referred to as exponential notation. Examples are:

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

The decimal equivalents are:

```
408.7, 408.7, 408.7, 0.04087
var a = 53210000000000000000000000000000 + 0.09333
```

Boolean

Boolean variables evaluate to either false or true. Because Siebel eScript automatically converts values when appropriate, when a Boolean variable is used in a numeric context, its value is converted to 0 if it is false, or 1 if it is true. A script is more precise when it uses the actual Siebel eScript values, false and true, but it works using the concepts of zero and nonzero.

String

A string is a series of characters linked together. A string is written using a pair of either double or single quotation marks, for example:

```
"I am a string"
'so am I'
'me too'
"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.

Siebel eScript automatically converts strings to numbers and numbers to strings, depending on the 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 “Automatic Type Conversion” on page 57.

Although strings are classified as a primitive data type, they are actually a hybrid type that shares characteristics of primitive and composite data types. A string may be thought of as an array (a composite data type) of characters, each element of which contains one character. For an explanation of arrays, read "Array" on page 55.

Composite Data Types

Although primitive data types are passed by value, composite types are passed by reference. If an argument is passed by reference, the variable's value may be changed for the calling procedure. When a composite type is assigned to a variable or passed to a parameter, only a reference that points to its data is passed, as in the following fragment:

```
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 to the property `AnObj.name`. The string is assigned by value because a property is a variable within an object. Two copies of the string `"Joe"` exist.

When `AnObj` is passed to the function `ReturnName()`, it is passed by reference. `CurObj` receives a reference to the object, but does not receive a copy of the object.

With this reference, `CurObj` can access every property and method of `AnObj`, which was passed to it. 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"` is transferred to the property.

Thus, `AnObj` holds two copies of the string `"Joe"`: one in the property .name and one in the .old property. Three total copies of `"Joe"` exist, including the original string literal.

Two commonly used composite data types are Object and Array.

Object

An object is a compound data type that consists of one or more pieces of data of any type 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 object data type in Visual Basic and the structure data type in C. The object data type also allows functions, called *methods,* to be used as object properties.

In Siebel eScript, functions are considered as variables. It is best to think of objects as having methods, which are functions, and properties, which are variables and constants.

Array

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 contain the three separate strings. These variables must be used separately.

In the second fragment, one variable holds the three strings. This array variable can be used as one unit, and the strings can also be accessed individually, by specifying the array subscript of the element containing the string to be used.

Arrays and objects use grouping similarly. Both are objects in Siebel eScript, but they have different notations for accessing properties. While arrays use subscripts, objects use property names or methods. In practice, arrays should be regarded as a unique data type.

Arrays and their characteristics are discussed more fully in "Array Objects" on page 97.

Special Data Types

This section discusses the undefined, null, and NaN (not a number) data types.

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.

Although variables may be of type undefined, there is no literal representation for undefined. Consider the following invalid fragment:

```
var test;
if (typeof test == "undefined")
TheApplication().RaiseErrorText("test is undefined");
```

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

Null

Null is a special data type that indicates that a variable is empty, and this condition is different from undefined. A null variable holds no value, although it might have previously held one.

The null type is represented literally by the identifier, null. Because Siebel eScript automatically converts data types, null is both useful and versatile.

Because null has a literal representation, an assignment such as the following is valid:

```
var test = null;
```

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

NaN

The NaN type means “not a number,” and NaN is an abbreviation for the phrase. However, NaN does not have a literal representation. To test for NaN, the function, isNaN(), must be used, as illustrated in the following fragment:

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

When the parseInt() function tries to parse the string `"a string"` into an integer, it returns NaN, because `"a string"` does not represent a number as 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 positive nonzero value
`Number.NaN`	NaN	Not a number
`Number.POSITIVE_INFINITY`	Infinity	Number greater than `MAX_VALUE`
`Number.NEGATIVE_INFINITY`	-Infinity	Number less than `MIN_VALUE`

Automatic Type Conversion

Conversion occurs automatically during concatenation involving both strings and numbers, and is subject to the following rules:

- Subtracting a string from a number or a number from a string converts the string to a number and performs subtraction on the two values.

- Adding a string to a number converts the number to a string and concatenates the two strings.
- Strings always convert to a base 10 number and must not contain any characters other than digits. The string `"110n"` does not convert to a number because the *n* character is meaningless as part of a number in Siebel eScript.

The following examples illustrate these automatic conversions:

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

However, to make sure that your code does not break if the conversion is not performed, use one of the casting functions to perform the appropriate conversion. (For details on these functions, read "Conversion or Casting Functions" on page 250.) The following example accepts string input and converts it to numeric to perform arithmetic:

```
var n = "55";
var d = "11";
divide it by:");
var division = Clib.div(ToNumber(n), ToNumber(d));
```

To specify more stringent conversions, use the parseFloat() Method of the global object. Siebel eScript has many global functions to cast data as a specific type. Some of these are not part of the ECMAScript standard. Read "parseFloat() Method" on page 260.

NOTE: There are circumstances under which conversion is not performed automatically. If you encounter such a circumstance, you must use one of the casting functions to get the desired result. For an explanation of casting functions, read "Conversion or Casting Functions" on page 250.

Properties and Methods of Basic Data Types

The basic data types, such as number and string, have properties and methods that may be used with any variable of that type. Any string variable may use any string method.

The properties and methods of the basic data types are retrieved in the same way as objects. They are commonly used internally by the Siebel eScript interpreter, but you may use them if you choose. 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 number = 5
var s = number.toString()
```

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

The following two methods are common to variables.

toString()

This method returns the value of a variable expressed as a string.

valueOf()

This method returns the value of a variable.

Expressions

An expression is a collection of two or more terms that perform a mathematical or logical operation. The terms are usually either variables or functions that are combined with an operator to evaluate to a string or numeric result. You use expressions to perform calculations, manipulate variables, or concatenate strings.

Expressions are evaluated according to order of precedence. Use parentheses to override the default order of precedence.

The order of precedence (from high to low) for the operators is:

- Arithmetic operators
- Comparison operators
- Logical operators

Operators

- “Mathematical Operators” on page 61
- “Bit Operators” on page 63
- “Logical Operators and Conditional Expressions” on page 64
- “Typeof Operator” on page 66
- “Conditional Operator” on page 67
- “String Concatenation Operator” on page 67

Mathematical Operators

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

Basic Arithmetic

The arithmetic operators in Siebel eScript are 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 examples use 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) (the remainder is ignored)
i = 10;    //i is now 10
i = i % 3;  //i is now 1, (10 mod 3)
```

Expressions may be grouped to affect the sequence of processing. 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 before other calculations.

In the following examples, the information in the remarks represents intermediate forms of the example calculations.

Notice that, because of the order of precedence,

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

has the same meaning as

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

but has a different meaning from

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

which is also different from

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

The use of parentheses is recommended whenever there may be confusion about how the expression is to be evaluated, even when parentheses are not required.

Assignment Arithmetic

Each of the operators shown in the previous section 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 on 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 mod 3), same as i = i % 3
```

Auto-Increment (++) and Auto-Decrement (--)

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

The auto-increment and auto-decrement operators can be used before their variables, as a prefix operator, or after, as a suffix operator. If they are used before a variable, the variable 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 demonstrate 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)
```

Bit Operators

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

Bit operators available in Siebel eScript are:

< <	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`

Logical Operators and Conditional Expressions

Logical operators compare two values and evaluate whether the resulting expression is false or true. A variable or any other expression may be false or true. An expression that performs a comparison is called a conditional expression.

Logical operators are used to make decisions about which statements in a script are executed, based on how a conditional expression evaluates.

The logical operators available in Siebel eScript are:

!	not	Reverse of an expression. If `(a+b)` is true, then `!(a+b)` is false.
&&	and	True if, and only if, both expressions are true. Because 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, because the whole expression is false.
||	or	True if either expression is true. Because 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 Siebel eScript interpreter returns true and does not evaluate the second.
= =	equality	True if the values are equal, otherwise false. Do not confuse the equality operator, = =, with the assignment operator, =.
! =	inequality	True if the values are not equal, otherwise false.
<	less than	The expression `a < b` is true if a is less than b.
>	greater than	The expression `a > b` is true if a is greater than b.
< =	less than or equal to	The expression `a <= b` is true if a is less than or equal to b.
> =	greater than or equal to	The expression `a >= b` is true if a is greater than b.

For example, if you were designing a simple guessing game, you might instruct the computer to select a number between 1 and 100, and you would try to guess what it is. The computer tells you whether you are right 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. If the conditional expression in the parenthesis following an `if` statement is true, the statement block following the `if` statement is executed. If the conditional expression is false, the statement block is ignored, and the computer continues executing the script at the next statement after the ignored block.

The script implementing this game might have a structure similar to the one that follows, in which GetTheGuess() is a function that obtains your guess.

```
var guess = GetTheGuess(); //get the user input
target_number = 37;
if (guess > target_number)
{
   TheApplication().RaiseErrorText('Guess is too high.');
}
if (guess < target_number)
{
   TheApplication().RaiseErrorText('guess is too low.');
}
if (guess == target_number);
{
   TheApplication().RaiseErrorText('You guessed the number!');
}
```

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

CAUTION: Remember that the assignment operator, =, is different from the equality operator, = =. If you use the assignment operator when you want to test for equality, your script fails because the Siebel eScript interpreter cannot differentiate between operators by context. This is a common mistake, even among experienced programmers.

Typeof Operator

The typeof operator 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 represents the variable's type: `"undefined"`, `"boolean"`, `"string"`, `"object"`, `"number"`, `"function"`, or `"buffer"`.

Conditional Operator

The conditional operator, a question mark, provides a shorthand method for writing `else` statements. Statements using the conditional operator are more difficult to read than conventional `if` statements, and so they are 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 true, then *expression_if_true* is evaluated, and the value of the entire expression is 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 fragments illustrate the use of the conditional operator:

```
foo = ( 5 < 6 ) ? 100 : 200; \
```

In the previous statement foo is set to 100, because the expression is true.

```
TheApplication().RaiseErrorText("Name is " + ((null==name) ?
"unknown" : name));
```

In the previous statement, the message box displays `"Name is unknown"` if the name variable has a null value. If it does not have a null value, the message box displays `"Name is "` plus the contents of the variable.

String Concatenation Operator

You can use the `+` operator to join strings together, or *concatenate* them. 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 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"`.

Functions

A function is an independent section of code that receives information from a program and performs some action with it. Functions are named using the same conventions as variables.

Once a function has been written, you do not have to think again about how to perform the operations in it. When you call the function, it handles the work for you. You only need to know what information the function needs to receive—the parameters—and whether it returns a value to the statement that called it.

TheApplication().RaiseErrorText is an example of a function that provides a way to display formatted text in the event of an error. It receives a string from the function that called it, displays the string in an alert box on the screen, and terminates the script. TheApplication().RaiseErrorText is a void function, which means that it has no return value.

In Siebel eScript, functions are considered a data type. They evaluate the function's return value. You can use a function anywhere you can use a variable. You can use any valid variable name as a function name. Use descriptive function names that help you keep track of what the functions do.

Two rules 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, a pair of parentheses, following their names. Data to be passed to a function is enclosed within these parentheses.

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

- "Function Scope" on page 70
- "Passing Variables to Functions" on page 70
- "The Function Arguments[] Property" on page 71
- "Function Recursion" on page 71
- "Error Checking for Functions" on page 73

Function Scope

Functions are global in scope and can be called from anywhere in a script. Think of functions as methods of the global object. A function may not be declared within another function so that its scope is merely within a certain function or section of a script.

The following two code fragments perform the same function. 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
}

TheApplication().RaiseErrorText(SumTwo(3, 4))


// fragment two
function SumTwo(a, b)
{
   return a + b
}

TheApplication().RaiseErrorText(global.SumTwo(3, 4))
```

In the fragment that 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 that hold the literal values that were passed to the function.

Passing Variables to Functions

Siebel eScript uses different methods to pass variables to functions, depending on the type of variable being passed. Such distinctions make sure that information gets to functions in the most logical way.

Primitive types such as strings, numbers, and Booleans are passed by value. The values of these variables are passed to a function. If a function changes one of these variables, the changes are not visible outside of the function in which the change took place.

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

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. For details, read "return Statement" on page 247.

The Function Arguments[] Property

The arguments[] property is an array 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.

This property 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:

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

NOTE: The arguments[] property for a particular function can be accessed only from within that function.

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 Siebel eScript. Each call to a function is independent of any other call to that function. However, recursion has limits. If a function calls itself too many times, a script runs out of memory and aborts.

Remember that a function can call itself if necessary. For example, the following function, factor(), factors a number. Factoring is a good candidate for recursion because it is a repetitive process where the result of one factor is then itself factored according to the same rules.

```
function factor(i) //recursive function to print factors of i,
{// and return the number of factors in 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 );
}
```

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. If the computer is unable to open a file, the Clib.fopen() method returns null.

If you try to read from or write to a file that was not properly opened, you receive errors. To prevent these errors, make sure that Clib.fopen() does not return null when it tries to open a file. Instead of calling Clib.fopen() as follows:

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

check to make sure that null is not returned:

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

if (null == fp)
{
   TheApplication().RaiseErrorText("Clib.fopen returned null");
}
```

You may abort a script in such a case, but you then know why the script failed. Read "The Clib Object" on page 131.

eScript Statements

This section describes statements that allow your program to make decisions and to direct the flow based on those decisions.

- "break Statement" on page 74
- "continue Statement" on page 75
- "do...while Statement" on page 76
- "for Statement" on page 77
- "for...in Statement" on page 78
- "goto Statement" on page 79
- "if Statement" on page 80
- "switch Statement" on page 83
- "throw Statement" on page 84
- "try Statement" on page 85
- "while Statement" on page 88
- "with Statement" on page 89

break Statement

The break statement terminates the innermost loop of for, while, or do statements. It is also used to control the flow within switch statements.

Syntax A

```
break;
```

Syntax B

```
break label;
```

Placeholder	Description
label	The name of the label indicating where execution is to resume

Returns Not applicable

Usage The break statement is legal only in loops or switch statements. In a loop, it is used to terminate the loop prematurely when the flow of the program eliminates the need to continue the loop. In the switch statement, it is used to prevent execution of cases following the selected case and to exit from the switch block.

When used within nested loops, break terminates execution only of the innermost loop in which it appears.

A label may be used to indicate the beginning of a specific loop when the break statement appears within a nested loop to terminate execution of a loop other than the innermost loop. A label consists of a legal identifier, followed by a colon, placed at the left margin of the work area.

Example For an example, read "switch Statement" on page 83.

See Also "do...while Statement" on page 76, "for Statement" on page 77, "if Statement" on page 80, and "while Statement" on page 88

continue Statement

The continue statement starts a new iteration of a loop.

Syntax A

```
continue;
```

Syntax B

```
continue label;
```

Placeholder	Description
label	The name of the label indicating where execution is to resume

Returns Not applicable

Usage The continue statement ends the current iteration of a loop and begins the next. Any conditional expressions are reevaluated before the loop reiterates.

A label may be used to indicate the point at which execution should continue. A label consists of a legal identifier, followed by a colon, placed at the left margin of the work area.

See Also "do...while Statement" on page 76, "for Statement" on page 77, "goto Statement" on page 79, and "while Statement" on page 88

do...while Statement

The do...while statement processes a block of statements until a specified condition is met.

Syntax

```
do

{

   statement_block;

} while (condition)
```

Placeholder	Description
statement_block	One or more statements to be executed within the loop
condition	An expression indicating the circumstances under which the loop should be repeated

Returns Not applicable

Usage The do statement processes the *statement_block* repeatedly until *condition* is met. Because *condition* appears at the end of the loop, *condition* is tested for only after the loop executes. For this reason, a do...while loop is always executed at least once before *condition* is checked.

Example This example increments a value and prints the new value to the screen until the value reaches 100.

```
var value = 0;
do
{
      value++;
      Clib.sprintf(value);
   } while( value < 100 );
```

See Also “for Statement” on page 77 and “while Statement” on page 88

for Statement

The for statement repeats a series of statements a fixed number of times.

Syntax

```
for ( [var] counter = start; condition; increment )

{

   statement_block;

}
```

Placeholder	Description
counter	A numeric variable for the loop counter
start	The initial value of the counter
condition	The condition at which the loop should end
increment	The amount by which the counter is changed each time the loop is run
statement_block	The statements or methods to be executed

Returns Not applicable

Usage The counter variable must be declared with var if it has not already been declared. If it is declared in the for statement, its scope is local to the loop.

First, the expression `counter = start` is evaluated. Then *condition* is evaluated. If *condition* is true or if there is no conditional expression, the statement is executed. Then the *increment* is executed and *condition* is reevaluated, which begins the loop again. If the expression is false, the statement is not executed, and the program continues with the next line of code after the statement.

Within the loop, the value of *counter* should not be changed, because changing the counter makes your script difficult to maintain and debug.

A for statement can control multiple nested loops. The various counter variables and their increments must be separated by commas. For example:

```
for (var i = 1, var j = 3; i < 10; i++, j++)
   var result = i * j;
```

Example For an example of the for statement, read "eval() Method" on page 257.

See Also "do...while Statement" on page 76 and "while Statement" on page 88

for...in Statement

The for...in statement loops through the properties of an object.

Syntax

```
for (propertyVar in object)

{

   statement_block;

}
```

Placeholder	Description
object	An object previously defined in the script
propertyVar	A variable that iterates over every property of the object

Returns Not applicable

Usage

NOTE: An object must have at least one defined property or it cannot be used in a for...in statement.

When using the for … in statement in this way, the statement block executes one time for every property of the object. For each iteration of the loop, the variable *propertyVar* contains the name of one of the properties of object and may be accessed with a statement of the form `object[propertyVar]`.

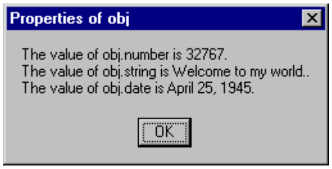
NOTE: Properties that have been marked with the DONT_ENUM attribute are not accessible to a for...in statement.

Example This example creates an object called `obj`, and then uses the for...in statement to read the object's properties. The result appears in the accompanying illustration.

```
function PropBtn_Click ()
{
   var obj = new Object;
   var propName;
   var msgtext = "";

   obj.number = 32767;
   obj.string = "Welcome to my world.";
   obj.date = "April 25, 1945";

   for (propName in obj)
   {
      msgtext = msgtext + "The value of obj." + propName +
         " is " + obj[propName] + ".\n";
   }
   TheApplication().RaiseErrorText(msgtext);
}
```



The value of obj.number is 32767.
The value of obj.string is Welcome to my world..
The value of obj.date is April 25, 1945.

goto Statement

The goto statement redirects execution to a specific point in a function.

Syntax

```
goto label;
```

Placeholder	Description
label	A marker, followed by a colon, for a line of code at which execution should continue

Returns Not applicable

Usage You can jump to any location within a function by using the goto statement. To do so, you must create a label—an identifier followed by a colon—at the point at which execution should continue. As a rule, goto statements should be used sparingly because they make it difficult to track program flow.

Example The following example uses a label to loop continuously until a number greater than 0 is entered:

```
function clickme_Click ()
{
restart:
   var number = 10;
   if (number <= 0 )
      goto restart;
   var factorial = 1;
   for ( var x = number; x >= 2; x-- )
      factorial = (factorial * x);
   TheApplication().RaiseErrorText( "The factorial of " +
      number + " is " + factorial + "." );
}
```

if Statement

The if statement tests a condition and proceeds depending on the result.

Syntax A

```
if (condition)

   statement;
```

Syntax B

```
if (condition)

{
   statement_block;
}
[else [if (condition)
{
   statement_block;
}]
[else
{
   statement_block;]
}]
```

Placeholder	Description
condition	An expression that evaluates to true or false
statement_block	One or more statements or methods to be executed if *expression* is true

Returns Not applicable

Usage The if statement is the most commonly used mechanism for making decisions in a program. When multiple statements are required, use the block version (Syntax B) of the if statement. When *expression* is true, the *statement* or *statement_block* following it is executed. Otherwise, it is skipped.

The following fragment is an example of an if statement:

```
if ( i < 10 )
{
   TheApplication().RaiseErrorText("i is smaller than 10.");
}
```

Note that the brackets are not required if only a single statement is to be executed if *condition* is true. You may use them for clarity.

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 Siebel eScript code, the else statement looks like this, if only one action is to be taken in either circumstance:

```
if ( i < 10 )
   TheApplication().RaiseErrorText("i is smaller than 10.");
else
   TheApplication().RaiseErrorText("i is not smaller than 10.");
```

If you want more than one statement to be executed for any of the alternatives, you must group the statements with brackets, like this:

```
if ( i < 10 )
{
   TheApplication().RaiseErrorText("i is smaller than 10.");
   i += 10;
}
else
{
   i -= 5;
   TheApplication().RaiseErrorText("i is not smaller than 10.");

}
```

To make more complex decisions, an else clause can be combined with an if statement to match one of a number of possible conditions.

Example

The following fragment illustrates using else with if. For another example, read "setTime() Method" on page 232.

```
if ( i < 10 )
{
   //check to see if I is less than or greater than 0
   if ( i < 0 )
{
      TheApplication().RaiseErrorText("i is negative; so it's " +
         "less than 10.");
}
   else if ( i > 10 )
{
      TheApplication().RaiseErrorText("i is greater than 10.");
}
```

```
   else
   {
      TheApplication().RaiseErrorText("i is 10.");
   }
```

See Also "switch Statement" on page 83

switch Statement

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

Syntax

```
switch( switch_variable )
{
   case value1:
      statement_block
      break;
   case value2:
      statement_block
      break;
   .
   .
   .
   [default:
      statement_block;]
}
```

Placeholder	Description
switch_variable	The variable upon whose value the course of action depends
value1, value2	Various values of *switch_variable,* which are followed by a colon
statement_block	One or more statements to be executed if the value of *switch_variable* is the value listed in the case statement

Returns Not applicable

Usage The switch statement is a way of choosing among alternatives when each choice depends upon the value of a single variable.

The variable *switch_variable* is evaluated, and then it is compared to the values in the case statements (*value1, value2*, ..., default) until a match is found. The statement block following the matched case is executed until the end of the switch block is reached or until a break statement exits the switch block.

If no match is found and a default statement exists, the default statement is executed.

Make sure to use a break statement to end each case. In the following example, if the break statement after the "I = I + 2;" statement were omitted, the computer would execute both "I = I + 2;" and "I = I + 3;", because the Siebel eScript interpreter executes commands in the switch block until it encounters a break statement.

Example Suppose that you had a series of account numbers, each beginning with a letter that indicates the type of account. You could use a switch statement to carry out actions depending on the account type, as in the following example:

```
switch ( key[0] )
{
case 'A':
   I=I+1;
   break;
case 'B':;
   I=I+2
   break;
case 'C':
   I=I+3;
   break;
default:
   I=I+4;
   break;
}
```

See Also "if Statement" on page 80

throw Statement

The throw statement is used to make sure that script execution is halted when an error occurs.

Syntax

```
throw exception
```

Parameter	Description
exception	An object in a named error class

Returns Not applicable

Usage Throw can be used to make sure that a script stops executing when an error is encountered, regardless of what other measures may be taken to handle the error. In the following code, the throw statement is used to stop the script after the error message is displayed.

```
try
{
   do_something;
}
catch( e )
{
   TheApplication().Trace (e.toString()));

throw e;
}
```

See Also "try Statement" on page 85 and "CORBACreateObject() Method" on page 252

try Statement

The try statement is used to process exceptions that occur during script execution.

Syntax

```
try
{
   statement_block
}
catch
{
   exception_handling_block
   [throw exception]
}
finally
```

```
{
   statement_block_2
}
```

Placeholder	Description
statement_block	A block of code that may generate an error
exception_handling_block	A block of code to process the error
exception	An error of a named type
statement_block_2	A block of code that is always executed, unless that block transfers control to elsewhere in the script

Returns Not applicable

Usage The try statement is used to handle functions that may raise *exceptions*, which are error conditions that cause the script to branch to a different routine. A try statement generally includes a catch clause or a finally clause, and may include both. The catch clause is used to handle the exception. To raise an exception, use the throw statement (see “throw Statement” on page 84).

When you want to trap potential errors generated by a block of code, place that code in a try statement, and follow the try statement with a catch statement. The catch statement is used to process the exceptions that may occur in the manner you specify in the *exception_handling_block*.

The following example demonstrates the general form of the try statement with the catch clause. In this example, the script continues executing after the error message is displayed:

```
try
{
   do_something;
}
catch( e )
{
   TheApplication().RaiseErrorText(Clib.rsprintf(
      "Something bad happened: %s\n",e.toString()));
}
```

The finally clause is used for code that should always be executed before exiting the try statement, regardless of whether the catch clause halts the execution of the script. Statements in the finally clause are skipped only if the finally clause redirects the flow of control to another part of the script. The finally statement can be exited by a goto, throw, or return statement.

Here is an example:

```
try
{
   return 10;
}
finally
{
   goto no_way;
}

no_way: statement_block
```

Execution continues with the code after the label, so the return statement is ignored.

You can use the try statement to process the exceptions thrown by CORBA objects, and to access their data members and exception names. If the exception contains nested objects or CORBA structures, they are skipped. For more information on creating and using CORBA objects in eScript, read "CORBACreateObject() Method" on page 252.

Example

The following example processes a CORBA exception. It assumes that the user is running the Account server and calling the function clear_balance(-1) on it. This raises the exception `AccountFrozen,` which is described in the CORBA IDL file as follows:

```
exception AccountFrozen {
   float mmx;
   long minimum;
};
```

This server assigns the value `7777.5555` to the variable *mmx*, and assigns `50` to the variable minimum, and then throws the AccountFrozen exception object. The eScript code might resemble the following:

```
try
{
   var cObj = CORBACreateObject("Account");
   var d1   = cObj.clear_balance(-1);
}
catch(obj)
{
   TheApplication().RaiseErrorText(obj.name + "\n" + obj.mmx +
"\n" + obj.minimum);
}
```

See Also “throw Statement” on page 84

while Statement

The while statement executes a particular section of code repeatedly until an expression evaluates to false.

Syntax

```
while (condition)
{
     statement_block;
}
```

Placeholder	Description
condition	The condition whose falsehood is used to determine when to stop executing the loop
statement_block	One or more statements to be executed while *condition* is true

Returns Not applicable

Usage The *condition* must be enclosed in parentheses. If *expression* is true, the Siebel eScript interpreter executes the *statement_block* following it. Then the interpreter tests the expression again. A while loop repeats until *condition* evaluates to false, and the program continues after the code associated with the while statement.

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

```
while(ThereAreUncalledNamesOnTheList() != false)
{
   var name = GetNameFromTheList();
   SendEmail(name);
}
```

with Statement

The with statement assigns a default object to a statement block, so you need to use the object name with its properties and methods.

Syntax

```
with (object)
{
     method1;
     method2;
     .
     .
     .
     methodn;
}
```

Placeholder	Description
object	An object with which you wish to use multiple methods
*method1, method2, method*n	Methods to be executed with the object

Returns Not applicable

Usage The with statement is used to save time when working with objects. It prepends the object name and a period to each method used.

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

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

Example The following fragment illustrates the use of the with statement:

```
var bcOppty;
var boBusObj;
boBusObj = TheApplication().GetBusObject("Opportunity");
bcOppty = boBusObj.GetBusComp("Opportunity");
var srowid = bcOppty.GetFieldValue("Id");

with (bcOppty)
{
   SetViewMode(SalesRepView);
   ActivateField( "Sales Stage" );
   SetSearchSpec("Id", srowid);
   ExecuteQuery(ForwardOnly);
}
   bcOppty = null;
   boBusObj = null;
```

The portion in the with block is equivalent to:

```
bcOppty.SetViewMode(SalesRepView);
bcOppty.ActivateField( "Sales Stage" );
bcOppty.SetSearchSpec("Id", srowid);
bcOppty.ExecuteQuery(ForwardOnly);
```

Siebel eScript Commands 3

This chapter presents the eScript commands sorted alphabetically by object type and then by command name. The following list shows the object types.

- “Applet Objects” on page 92
- “The Application Object” on page 94
- “Array Objects” on page 97
- “BLOB Objects” on page 103
- “Buffer Objects” on page 109
- “Business Component Objects” on page 122
- “Business Object Objects” on page 128
- “Business Service Objects” on page 129
- “The Clib Object” on page 131
- “The Date Object” on page 210
- “The Exception Object” on page 245
- “Function Objects” on page 246
- “The Global Object” on page 249
- “The Math Object” on page 274
- “User-Defined Objects” on page 293
- “Property Set Objects” on page 298
- “The SElib Object” on page 300
- “String Objects” on page 303

Applet Objects

Within a Siebel application, an applet serves as a container for the collection of user interface objects that together represent the visible representation of one business component (BusComp) object. Applets are combined to form views. Views constitute the display portions of a Siebel application. Applet objects are available in Browser Script. Methods of applet objects are documented in the *Siebel Object Interfaces Reference*.

A Web applet represents an applet that is rendered by the Siebel Web Engine. It exists only as a scriptable object in Server Script and is accessed by using the Edit Server Script command on the selected applet. Because applet events and methods are not supported in the Siebel Web Engine, the Web applet interfaces are available in their place.

Method or Event	Description
ActiveMode() Method	ActiveMode returns a string containing the name of the current Web Template mode.
Applet_ChangeFieldValue() Event	The ChangeFieldValue event is fired when the data in a field changes.
Applet_ChangeRecord() Event	The ChangeRecord event is called when the user moves to a different row or view.
Applet_InvokeMethod() Event	The InvokeMethod event is triggered by a call to applet.InvokeMethod, a call to a specialized method, or by a user-defined menu.
Applet_Load() Event	The Load event is triggered after an applet has loaded and after data is displayed.
Applet_PreInvokeMethod() Event	The PreInvokeMethod event is called before a specialized method is invoked by the system, by a user-defined applet menu, or by calling InvokeMethod on an applet.
BusComp() Method	BusComp() returns the business component that is associated with the applet.
BusObject() Method	BusObject() returns the business object for the business component for the applet.

Method or Event	**Description**
FindActiveXControl() Method	FindActiveXControl returns a reference to a DOM element based upon the name specified in the name argument.
FindControl() Method	FindControl returns the control whose name is specified in the argument. This applet must be part of the displayed view.
InvokeMethod() Method	The InvokeMethod() method calls an argument-specified specialized method.
Name() Method	The Name() method returns the name of the applet.
WebApplet_InvokeMethod() Event	The InvokeMethod() event is called after a specialized method or a user-defined method on the Web applet has been executed.
WebApplet_Load() Event	The WebApplet_Load() event is triggered just after an applet is loaded.
WebApplet_PreCanInvokeMethod() Event	The PreCanInvokeMethod() event is called before the PreInvokeMethod, allowing the developer to determine whether or not the user has the authority to invoke a specified WebApplet method.
WebApplet_PreInvokeMethod() Event	The PreInvokeMethod() event is called before a specialized method for the Web applet is invoked by the system, or a user-defined method is invoked through *oWebAppVar*.InvokeMethod.
WebApplet_ShowControl() Event	This event allows scripts to modify the HTML generated by the Siebel Web Engine to render a control on a Web page in a customer or partner application.
WebApplet_ShowListColumn() Event	This event allows scripts to modify the HTML generated by the Siebel Web Engine to render a list column on a Web page in a customer or partner application.

The Application Object

The application object represents the Siebel application that is currently active and is an instance of the Application object type. An application object is created when a Siebel software application is started. This object contains the properties and events that interact with Siebel software as a whole. An instance of a Siebel application always has exactly one application object. Methods of the application object are documented in the *Siebel Object Interfaces Reference*.

Method or Event	Description
ActiveBusObject() Method	ActiveBusObject() returns the business object for the business component for the active applet.
ActiveViewName() Method	ActiveViewName() returns the name of the active view.
Application_Close() Event	The Close() event is called before the application exits. This allows Basic scripts to perform last-minute cleanup (such as cleaning up a connection to a COM server). It is called when the application is notified by Windows that it should close, but not if the process is terminated directly.
Application_InvokeMethod() Event	The Application_InvokeMethod() event is called after a specialized method is invoked.
Application_Navigate() Event	The Navigate() event is called after the client has navigated to a view.
Application_PreInvokeMethod() Event	The PreInvokeMethod() event is called before a specialized method is invoked by a user-defined applet menu or by calling InvokeMethod on the application.
Application_PreNavigate() Event	The PreNavigate() event is called before the client has navigated from one view to the next.
Application_Start() Event	The Start() event is called when the client starts and the user interface is first displayed.
CurrencyCode() Method	CurrencyCode() returns the operating currency code associated with the division to which the user’s position has been assigned.
GetProfileAttr() Method	GetProfileAttr() returns the value of an attribute in a user profile.

Method or Event	Description
GetService() Method	The GetService() method returns a specified business service. If the service is not already running, it is constructed.
GetSharedGlobal() Method	The GetSharedGlobal() method gets the shared user-defined global variables.
GotoView() Method	GotoView() activates the named view and its BusObject. As a side effect, this method activates the view's primary applet, its BusComp, and its first tab sequence control. Further, this method deactivates any BusObject, BusComp, applet, or control objects that were active prior to this method call.
InvokeMethod() Method	InvokeMethod() calls a specialized or user-created method specified by its argument.
LoginId() Method	The LoginId() method returns the login ID of the user who started the Siebel application.
LoginName() Method	The LoginName() method returns the login name of the user who started the Siebel application (the name typed in the login dialog box).
LookupMessage() Method	The LookupMessage method returns the translated string for the specified key, in the current language, from the specified category.
NewPropertySet() Method	The NewPropertySet() method constructs a new property set object.
PositionId() Method	The PositionId() method returns the position ID (ROW_ID from S_POSTN) of the user's current position. This is set by default when the Siebel application is started and may be changed (using Edit > Change Position) if the user belongs to more than one position.
PositionName() Method	The PositionName() method returns the position name of the user's current position. This is set by default when the Siebel application is started and may be changed (using Edit > Change Position) if the user belongs to more than one position.
RaiseError() Method	The RaiseError method raises a scripting error message to the browser. The error code is a canonical number.

Method or Event	Description
RaiseErrorText() Method	The RaiseErrorText method raises a scripting error message to the browser. The error text is the specified literal string.
SetPositionId() Method	SetPositionId() changes the position of the current user to the value specified in the input argument. For SetPositionId() to succeed, the user must be assigned to the position to which they are changing.
SetPositionName() Method	SetPositionName() changes the position of the current user to the value specified in the input argument. For SetPositionName() to succeed, the user must be assigned to the position to which they are changing.
SetProfileAttr() Method	SetProfileAttr() is used in personalization to assign values to attributes in a user profile.
SetSharedGlobal() Method	The SetSharedGlobal() method sets a shared user-defined global variable, which may be accessed using GetSharedGlobal.
Trace() Method	The Trace() method appends a message to the trace file. Trace is useful for debugging the SQL query execution.
TraceOff() Method	TraceOff() turns off the tracing started by the TraceOn method.
TraceOn() Method	TraceOn() turns on the tracking of allocations and deallocations of Siebel objects, and SQL statements generated by the Siebel application.

Array Objects

An array is a special class of object that holds several values rather than one. You refer to a single value in an array by using an index number or string assigned to that value.

The values contained within an array object are called elements of the array. The index number used to identify an element follows its array name in brackets. Array indices must be either numbers or strings.

Array elements can be of any data type. The elements in an array do not 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 how to assign values to an array:

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

The variables foo and goo must be either numbers or strings.

Because arrays use a number to identify the data they contain, they provide an easy way to work with sequential data. For example, suppose you want to keep track of how many jellybeans you ate each day, so you could 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 your data stored 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++)
TheApplication().Trace("On April " + x + " I ate " + April[x] +
   " jellybeans.\n");
```

Arrays usually start at index [0], not index [1].

NOTE: Arrays do not have to be continuous. You can have an array with elements at indices 0 and 2 but none at 1.

See Also “The Array Constructor” on page 98, “join() Method” on page 99, “length Property” on page 99, “reverse() Method” on page 100, “sort() Method” on page 101

The Array Constructor

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 number of elements, declare the array using the number of elements as a parameter of the Array() function. The following line creates an array with 31 elements:

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

Finally, you can pass a number of elements to the Array() function, which creates an array containing the parameters passed. The following example creates an array with six elements. `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 `c[0]`, not `c[1]`.

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

You can also create arrays dynamically. If you refer to a variable with an index in brackets, the variable becomes an array. Arrays created in this manner cannot use the methods and properties described in the next section, so use the Array() constructor function to create arrays.

join() Method

The join() method creates a string of array elements.

Syntax *arrayName*.join([*separatorString*])

Parameter	Description
separatorString	A string of characters to be placed between consecutive elements of the array; if not specified, a comma is used

Returns A string containing the elements of the specified array, separated either by commas or by instances of *separatorString.*

Usage By default, the array elements are separated by commas. The order in the array is the order used for the join() method. The following fragment sets 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.

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

Example This example creates the string "3*/*5*/*6*/*3":

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

length Property

The length property returns a number representing the largest index of an array, plus 1.

Syntax *arrayName*.length

Returns The number of the largest index of the array, plus 1.

NOTE: This value does not necessarily represent the actual number of elements in an array, because elements do not have to be contiguous.

Usage For example, suppose you had two arrays, ant and bee, with the following elements:

```
var ant = new Array;     var bee = new Array;
ant[0] = 3               bee[0] = 88
ant[1] = 4               bee[3] = 99
ant[2] = 5
ant[3] = 6
```

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

By changing the value of the length property, you can remove array elements. For example, if you change ant.length to 2, ant loses elements after the first two, and the values stored at the other indices are lost. If you set bee.length to 2, then bee consists of two members: bee[0], with a value of 88, and bee[1], with an undefined value.

reverse() Method

The reverse() method switches the order of the elements of an array, so that the last element becomes the first.

Syntax *arrayName*.reverse()

Parameter	Description
Not applicable	

Returns *arrayName* with the elements in reverse order.

Usage The reverse() method sorts the existing array, rather than returning a new array. In any references to the array after the reverse() method is used, the new order is used.

Example The following code:

```
var communalInsect = new Array;
communalInsect[0] = "ant";
communalInsect[1] = "bee";
communalInsect[2] = "wasp";
communalInsect.reverse();
```

produces the following array:

```
communalInsect[0] == "wasp"
communalInsect[1] == "bee"
communalInsect[2] == "ant"
```

sort() Method

The sort() method sorts the elements of an array into the order specified by the *compareFunction*.

Syntax `arrayName.sort([compareFunction])`

Parameter	Description
compareFunction	A user-defined function that can affect the sort order

Returns *arrayName* with its elements sorted into the order specified.

Usage If no *compareFunction* is supplied, then elements are converted to strings before sorting. When numbers are sorted into ASCII order, they are compared left-to-right, so that, for example, 32 comes before 4. This may not be the result you want. However, the *compareFunction* enables you to specify a different way to sort the array elements. The name of the function you want to use to compare values is passed as the only parameter to sort().

If a compare function is supplied, the array elements are sorted according to the return value of the compare function.

Example The following example demonstrates the use of the sort() method with and without a compare function. It first displays the results of a sort without the function and then uses a user-defined function, `compareNumbers(a, b)`, to sort the numbers properly. In this function, if a and b are two elements being compared, then:

- If `compareNumbers(a, b)` is less than zero, b is given a lower index than a.
- If `compareNumbers(a, b)` returns zero, the order of a and b is unchanged.
- If `compareNumbers(a, b)` is greater than zero, b is given a higher index than a.

BLOB Objects

The following topics describe binary large objects (BLOBs).

- "The blobDescriptor Object" on page 103
- "Blob.get() Method" on page 105
- "Blob.put() Method" on page 105
- "Blob.size() Method" on page 107

The blobDescriptor Object

The blobDescriptor Object describes the structure of the BLOB. When an object needs to be sent to a process other than the Siebel eScript interpreter, such as to a Windows API function, a blobDescriptor object must be created that describes the order and type of data in the object. This description tells how the properties of the object are stored in memory and is used with functions like Clib.fread() and SElib.dynamicLink().

A blobDescriptor has the same data properties as the object it describes. Each property must be assigned a value that specifies how much memory is required for the data held by that property. The keyword "this" is used to refer to the parameters passed to the constructor function and can be conceptually thought of as "this object." Consider the following object:

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

The following code creates a blobDescriptor object that describes the Rectangle object:

```
var bd = new blobDescriptor();

bd.width  = UWORD32;
bd.height = UWORD32;
```

You can now pass bd as a blobDescriptor parameter to functions that require one. The values assigned to the properties depend on what the receiving function expects. In the preceding example, the function that is called expects to receive an object that contains two 32-bit words or data values. If you write a blobDescriptor for a function that expects to receive an object containing two 16-bit words, assign the two properties a value of UWORD16.

One of the following values must be used with blobDescriptor object properties to indicate the number of bytes needed to store the property:

WCHAR	Handled as a native UNICODE string
UWORD8	Stored as an unsigned byte
SWORD8	Stored as an integer
UWORD16	Stored as an unsigned, 16-bit integer
SWORD16	Stored as a signed 16-bit integer
UWORD24	Stored as an unsigned 24-bit integer
SWORD24	Stored as a signed 24-bit integer
UWORD32	Stored as an unsigned 32-bit integer
SWORD32	Stored as a signed 32-bit integer
FLOAT32	Stored as a floating-point number
FLOAT64	Stored as a double-precision floating-point number
STRINGHOLDER	Used to indicate a value that is assigned a string by the function to which it is passed. (It allocates 10,000 bytes to contain the string, then truncates this length to the appropriate size, removes any terminating null characters, and initializes the properties of the string.)

If the blobDescriptor describes an object property that is a string, the corresponding property should be assigned a numeric value that is larger than the length of the longest string the property may hold. Object methods usually may be omitted from a blobDescriptor.

BlobDescriptors are used primarily for passing eScript's JavaScript-like data structures to C or C++ programs and to the Clib methods, which expect a very rigid and precise description of the values being passed.

Blob.get() Method

This method reads data from a binary large object.

Syntax A `Blob.get(blobVar, offset, dataType)`

Syntax B `Blob.get(blobVar, offset, bufferLen)`

Syntax C `Blob.get(blobVar, offset, blobDescriptor dataDefinition)`

Argument	Description
blobVar	The name of the binary large object to use
offset	The position in the BLOB from which to read the data
dataType	An integer value indicating the format of the data in the BLOB
bufferLen	An integer indicating the size of the buffer in bytes
blobDescriptor dataDefinition	A blobDescriptor object indicating the form of the data in the BLOB

Returns The data read from the BLOB.

This method reads data from a specified location of a binary large object (BLOB), and is the companion function to Blob.put().

Use Syntax A for byte, integer, and float data. Use Syntax B for byte[] data. Use Syntax C for object data.

dataType must have one of the values listed for blobDescriptors in "The blobDescriptor Object" on page 103.

See Also "The blobDescriptor Object" on page 103 and "Blob.put() Method" on page 105

Blob.put() Method

The Blob.put method puts data into a specified location within a binary large object.

Syntax A `Blob.put(blobVar[, offset], data, dataType)`

Syntax B `Blob.put(blobVar[, offset], buffer, bufferLen)`

Syntax C

```
Blob.put(blobVar[, offset], srcStruct, blobDescriptor
dataDefinition)
```

Argument	Description
blobVar	The name of the binary large object to use
offset	The position in the BLOB at which to write the data
data	The data to be written
dataType	The format of the data in the BLOB
buffer	A variable containing a buffer
bufferLen	An integer representing the length of *buffer*
srcStruct	A BLOB containing the data to be written
blobDescriptor dataDefinition	A blobDescriptor object indicating the form of the data in the BLOB

Returns An integer representing the byte offset for the byte after the end of the data just written. If the data is put at the end of the BLOB, the size of the BLOB.

Usage This method puts data into a specified location of a binary large object (BLOB) and, along with Blob.get(), allows for direct access to memory within a BLOB variable. Data can be placed at any location within a BLOB. The contents of such a variable may be viewed as a packed structure, that is, a structure that does not pad each member with enough nulls to make every member a uniform length. (The exact length depends on the CPU, although 32 bytes is common.)

Syntax C is used to pass the contents of an existing BLOB (*srcStruct*) to the *blobVar.*

If a value for *offset* is not supplied, then the data is put at the end of the BLOB, or at offset 0 if the BLOB is not yet defined.

The *data* is converted to the specified *dataType* and then copied into the bytes specified by *offset*.

If d*ataType* is not the length of a byte buffer, then it must have one of the values listed for blobDescriptors in "The blobDescriptor Object" on page 103.

Example If you were sending a pointer to data in an external C library and knew that the library expected the data in a packed C structure of the form:

```
struct foo
{
   signed char a;
   unsigned int b;
   double c;
};
```

and if you were building this structure from three corresponding variables, then such a building function might look like the following, which returns the offset of the next available byte:

```
function BuildFooBlob(a, b, c)
{
   var offset = Blob.put(foo, 0, a, SWORD8);
   offset = Blob.put(foo, offset, b, UWORD16);
   Blob.put(foo, offset, c, FLOAT64);
   return foo;
}
```

or, if an offset were not supplied:

```
functionBuildFooBlob(a, b, c)
{
   Blob.put(foo, a, SWORD8);
   Blob.put(foo, b, UWORD16);
   Blob.put(foo, c, FLOAT64);
   return foo;
}
```

See Also "The blobDescriptor Object" on page 103 and "Blob.get() Method" on page 105

Blob.size() Method

This method determines the size of a binary large object (BLOB).

Syntax A `Blob.size(`*`blobVar`*`[, `*`SetSize`*`])`

Syntax B `Blob.size(`*`dataType`*`)`

Syntax C `Blob.size(`*`bufferLen`*`)`

Syntax D `Blob.size(`*`blobDescriptor dataDefinition`*`)`

Argument	Description
blobVar	The name of the binary large object to use
setSize	An integer that determines the size of the BLOB
dataType	An integer value indicating the format of the data in the BLOB
bufferLen	An integer indicating the number of bytes in the buffer
blobDescriptor dataDefinition	A blobDescriptor object indicating the form of the data in the BLOB

Returns The number of bytes in *blobVar;* if *setSize* is provided, returns *setSize.*

Usage The parameter *blobVar* specifies the blob to use. If SetSize is provided, then the blob *blobVar* is altered to this size or created with this size.

If *dataType*, *bufferLen*, or *dataDefinition* are used, these parameters specify the type to be used for converting Siebel eScript data to and from a BLOB.

The *dataType* argument must have one of the values listed for blobDescriptors in "The blobDescriptor Object" on page 103.

See Also "The blobDescriptor Object" on page 103

Buffer Objects

Buffer objects provide a way to manipulate data at a very basic level. A Buffer object 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 is copied into the newly created Buffer object.

In the examples that follow, *bufferVar* is a generic variable name to which a Buffer object is assigned.

- "The Buffer Constructor" on page 110
- "Properties" on page 111
- "Methods" on page 112
- "bigEndian Property" on page 112
- "cursor Property" on page 113
- "data Property" on page 113
- "getString() Method" on page 114
- "getValue() Method" on page 114
- "offset[] Method" on page 115
- "putString() Method" on page 116
- "putValue() Method" on page 117
- "size Property" on page 119
- "subBuffer() Method" on page 120
- "toString() Method" on page 120
- "unicode Property" on page 121

The Buffer Constructor

To create a Buffer object, use one of the following syntax forms.

Syntax A

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

Argument	Description
size	The size of the new buffer to be created
unicode	True if the buffer is to be created as a Unicode string, otherwise, false; default is false
bigEndian	True if the largest data values are stored in the most significant byte; false if the largest data values are stored in the least significant byte; default is true

Usage

If *size* is specified, then the new buffer is created with the specified size and filled with null bytes. If no *size* is specified, then the buffer is created with a size of 0, although 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.

Syntax B

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

Usage

This syntax creates a new Buffer object from the string provided. If the string parameter is a Unicode string (if 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 the original string was in Unicode or not.

Similarly, specifying true makes sure 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.

Syntax C `new Buffer(buffer [, unicode] [, bigEndian]);`

Argument	Description
buffer	The buffer object from which the new buffer is to be created
unicode	True if the buffer is to be created as a Unicode string, otherwise, false; default is the Unicode status of the underlying Siebel eScript engine
bigEndian	True if the largest data values are stored in the most significant byte; false if the largest data values are stored in the least significant byte; default is true

Usage 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, although they do set the relevant flags for future use.

Syntax D `new Buffer(bufferobject);`

Argument	Description
bufferobject	The buffer object from which the new buffer is to be created

Usage 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.

Properties

- “bigEndian Property” on page 112
- “cursor Property” on page 113
- “data Property” on page 113
- “size Property” on page 119
- “unicode Property” on page 121

Methods

The following is a list of buffer object methods.

- "getString() Method" on page 114
- "getValue() Method" on page 114
- "offset[] Method" on page 115
- "putValue() Method" on page 117
- "size Property" on page 119
- "subBuffer() Method" on page 120
- "toString() Method" on page 120

bigEndian Property

This property is a Boolean flag specifying whether to use bigEndian byte ordering when calling getValue() and putValue().

Syntax

```
bufferVar.bigEndian
```

Parameter	Description
Not applicable	

Returns Not applicable

Usage When a data value consists of more than one byte, the byte containing the smallest units of the value is called the *least significant byte;* the byte containing the biggest units of the value is called the *most significant byte*. When the bigEndian property is true, the bytes are stored in descending order of significance. When false, they are stored in ascending order of significance.

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 operating system and processor.

cursor Property

The current position within a buffer.

Syntax `bufferVar.cursor`

Parameter	Description
Not applicable	

Returns Not applicable

Usage The value of cursor is always between 0 and the value set in the size property. A value can be assigned to this property.

If the cursor is set beyond the end of a buffer, the buffer is extended to accommodate the new position and filled with null bytes. Setting the cursor to a value less than 0 places the cursor at the beginning of the buffer, position 0.

Example For examples, read "getString() Method" on page 114 and "size Property" on page 119.

See Also "size Property" on page 119

data Property

This property is a reference to the internal data of a buffer.

Syntax `bufferVar.data`

Parameter	Description
Not applicable	

Returns Not applicable

Usage This property is used as a temporary value to allow passing of buffer data to functions that do not recognize buffer objects.

getString() Method

This method returns a string of a specified length, starting from the current cursor location.

Syntax `bufferVar.getString( [length] )`

Parameter	Description
length	The length of the string to return, in bytes

Returns A string of *length* characters, starting at the current cursor location in the buffer.

Usage This method returns a string starting from the current cursor location and continuing for *length* bytes.

If no length is specified, the method reads until a null byte is encountered or the end of the buffer is reached. 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 Also "getValue() Method" on page 114, "offset[] Method" on page 115, and "size Property" on page 119

getValue() Method

This method returns a value from the current cursor position in a Buffer object.

Syntax

```
bufferVar.getValue([valueSize][, valueType ])
```

Parameter	Description
valueSize	A positive number indicating the number of bytes to be read; default is 1
valueType	The type of data to be read, expressed as one of the following: ■ signed (the default) ■ unsigned ■ float

Returns The value at the current position in a Buffer object.

Usage To determine where to read from the buffer, use the *bufferVar*.cursor() method.

Acceptable values for *valueSize* are 1, 2, 3, 4, 8, and 10, providing that *valueSize* does not conflict with the optional *valueType* flag. 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

The combination of *valueSize* and *valueType* must match the data to be read.

See Also "putValue() Method" on page 117

offset[] Method

This method provides array-style access to individual bytes in the buffer.

Syntax *bufferVar*[*offset*]

Argument	Description
offset	A number indicating a position in *bufferVar* at which a byte is to be placed in, or read from, a buffer

Returns Not applicable

Usage This is an array-like version of the getValue() and putValue() methods that works only with bytes. You may either get or set these values. The following line assigns the byte at offset 5 in the buffer to the variable `goo`:

```
goo = foo[5]
```

The following line places the value of `goo` (assuming that value is a single byte) to position 5 in the buffer `foo`:

```
foo[5] = goo
```

Every get or put operation uses byte types, that is, eight-bit signed words (SWORD8). If *offset* is less than 0, then 0 is used. If *offset* is greater than the length of the buffer, the size of the buffer is extended with null bytes to accommodate it. If you need to work with character values, you have to convert them to their ANSI or Unicode equivalents.

See Also "getValue() Method" on page 114 and "putValue() Method" on page 117

putString() Method

This method puts a string into a buffer object at the current cursor position.

Syntax *bufferVar*.putString(*string*)

Parameter	Description
string	The string literal to be placed into the buffer object, or the string variable whose value is to be placed into the buffer object

Returns Not applicable

Usage If the unicode flag is set within the buffer object, then the string is put into the buffer object as a Unicode string; otherwise, it is put into the buffer object 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.

A terminating null byte is not added at end of the string.

To put a null terminated string into the buffer object, do the following:

```
buf1.putString("Hello");   // Put the string into the buffer
buf1.putValue( 0 );         // Add terminating null byte
```

Example The following example places the string `language` in the buffer `exclamation` and displays the modified contents of `explanation`, which is the string, `"I love coding with Siebel eScript"`.

```
function eScript_Click ()
{
   var exclamation = new Buffer("I love coding with . . .");
   var language = "Siebel eScript";
   exclamation.cursor = 19;
   exclamation.putString(language);
   TheApplication().RaiseErrorText(exclamation);
}
```

See Also "getString() Method" on page 114

putValue() Method

This method puts the specified value into a buffer at the current file cursor position.

Syntax `*bufferVar*.putValue(*value*[, *valueSize*][, *valueType* ])`

Parameter	Description
value	A number

Parameter	Description
valueSize	A positive number indicating the number of bytes to be used; default is 1
valueType	The type of data to be read, expressed as one of the following: ■ signed (the default) ■ unsigned ■ float

Returns Not applicable

Usage This method puts a specific value into a buffer. Acceptable values for *valueSize* are 1, 2, 3, 4, 8, and 10, providing that this value does not conflict with the optional *valueType* flag.

Combined with *valueSize*, any type of data can be put into a buffer. 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

Any other combination causes an error. The value is put into the buffer at the current cursor position, and the cursor value is automatically incremented by the size of the value to reflect this addition. To explicitly put a value at a specific location while preserving the cursor location, do something similar to the following.

```
var oldCursor = bufferItem.cursor; // Save the cursor location
bufferItem.cursor = 20;            // Set to new location
bufferItem.putValue(foo);          // Put bufferItem at offset 20
bufferItem.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` is converted to something like `1.39999974`. This is sufficiently insignificant to ignore, but note that the following does not hold true:

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

This situation can be prevented by using 8 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, although it may not be enough to affect most calculations.

See Also "getValue() Method" on page 114

size Property

The size of the Buffer object.

Syntax *bufferVar*.size

Parameter	Description
Not applicable	

Returns Not applicable

Usage A value may be assigned to this property; for example,

```
inBuffer.size = 5
```

If a buffer is increased beyond its present size, the additional spaces are filled with null bytes. If the buffer size is reduced such that the cursor is beyond the end of the buffer, the cursor is moved to the end of the modified buffer.

See Also "cursor Property" on page 113

subBuffer() Method

This method returns a new Buffer object consisting of the data between two specified positions.

Syntax

`bufferVar.subBuffer(beginning, end)`

Parameter	Description
beginning	The cursor position at which the new Buffer object should begin
end	The cursor position at which the new Buffer object should end

Returns

A new Buffer object consisting of the data in *bufferVar* between the *beginning* and *end* positions.

Usage

If *beginning* is less than 0, then it is treated as 0, the start of the buffer.

If *end* is beyond the end of the buffer, then the new subbuffer is extended with null bytes, but the original buffer is not altered. The unicode and bigEndian flags are duplicated in the new buffer.

The length of the new buffer is set to *end* - *beginning*. If the cursor in the old buffer is between *beginning* and *end*, then it is converted to a new relative position in the new buffer. If the cursor was before *beginning*, it is set to 0 in the new buffer; if it was past *end*, it is set to the end of the new buffer.

Example

This code fragment creates the new buffer `language` and displays its contents—the string `"Siebel eScript"`.

```
var loveIt= new Buffer("I love coding with Siebel eScript!");
var language = loveIt.subBuffer(19, (loveIt.size - 1))
TheApplication().RaiseErrorText(language);
```

See Also

"getString() Method" on page 114

toString() Method

This method returns a string containing the same data as the buffer.

Syntax `bufferVar.toString()`

Parameter	Description
Not applicable	

Returns A string object that contains the same data as the buffer object.

Usage This method returns a string whose contents are the same as that of *bufferVar*. Any conversion to or from Unicode is done according to the unicode flag of the object.

unicode Property

This property is a Boolean flag specifying whether to use Unicode strings when calling getString() and putString().

Syntax `bufferVar.unicode`

Parameter	Description
Not applicable	

Returns Not applicable

Usage This value is set when the buffer is created, but may be changed at any time. This property defaults to false for Siebel eScript.

Example The following lines of code set the unicode property of a new buffer to true:

```
var aBuffer = new Buffer();
aBuffer.unicode = true;
```

Business Component Objects

A business component defines the structure, the behavior, and the information displayed by a particular subject, such as a product, contact, or account. Siebel business components are logical abstractions of one or more database tables. The information stored in a business component is usually specific to a particular subject and is typically not dependent on other business components. Business components can be used in one or more business objects.

Business component objects have associated data structured as records, they have properties, and they contain data units called *fields*. In Siebel eScript, fields are accessed through business components. The business component object supports getting and setting field values, moving backward and forward through data in a business component object, and filtering changes to data it manages.

Methods of business component objects are documented in the *Siebel Object Interfaces Reference*.

Method or Event	**Description**
ActivateField() Method	ActivateField() allows queries to retrieve data for the field specified in its argument.
ActivateMultipleFields() Method	ActivateMultipleFields() allows users to activate the fields specified in the property set input argument.
Associate() Method	The Associate() method creates a new many-to-many relationship for the parent object through an association business component (read "GetAssocBusComp() Method" on page 125).
BusComp_Associate() Event	The Associate() event is called after a record is added to a business component to create an association.
BusComp_ChangeRecord() Event	The ChangeRecord() event is called after a record becomes the current row in the business component.
BusComp_CopyRecord() Event	The CopyRecord() event is called after a row has been copied in the business component and that row has been made active.

Method or Event	Description
BusComp_DeleteRecord() Event	The DeleteRecord() event is called after a row is deleted. The current context is a different row (the Fields of the just-deleted row are no longer available).
BusComp_InvokeMethod() Event	The InvokeMethod() event is called when the InvokeMethod method is called on a business component.
BusComp_NewRecord() Event	The NewRecord() event is called after a new row has been created in the business component and that row has been made active. The event may be used to set up default values for Fields.
BusComp_PreAssociate() Event	The PreAssociate() event is called before a record is added to a business component to create an association. The semantics are the same as BusComp_PreNewRecord.
BusComp_PreCopyRecord() Event	The PreCopyRecord() event is called before a new row is copied in the business component. The event may be used to perform precopy validation.
BusComp_PreDeleteRecord() Event	The PreDeleteRecord event is called before a row is deleted in the business component. The event may be used to prevent the deletion or to perform any actions in which you need access to the record that is to be deleted.
BusComp_PreGetFieldValue() Event	The PreGetFieldValue() event is called when the value of a business component field is accessed.
BusComp_PreInvokeMethod() Event	The PreInvokeMethod() event is called before a specialized method is invoked on the business component.
BusComp_PreNewRecord() Event	The PreNewRecord event is called before a new row is created in the business component. The event may be used to perform preinsert validation.
BusComp_PreQuery() Event	The PreQuery() event is called before query execution.

Method or Event	Description
BusComp_PreSetFieldValue() Event	The PreSetFieldValue() event is called before a value is pushed down into the business component from the user interface or through a call to SetFieldValue.
BusComp_PreWriteRecord() Event	The PreWriteRecord() event is called before a row is written out to the database. The event may perform any final validation necessary before the actual save occurs.
BusComp_Query() Event	The Query() event is called just after the query is completed and the rows have been retrieved but before the rows are actually displayed.
BusComp_SetFieldValue() Event	The SetFieldValue() event is called when a value is pushed down into the business component from the user interface or through a call to SetFieldValue.
BusComp_WriteRecord() Event	The WriteRecord event is called after a row is written out to the database.
BusObject() Method	The BusObject() method returns the business object that contains the business component.
ClearToQuery() Method	The ClearToQuery() method clears the current query and sort specifications on the business component.
DeactivateFields() Method	DeactivateFields deactivates the fields that are currently active from a business component SQL query statement.
DeleteRecord() Method	DeleteRecord() removes the current record from the business component.
ExecuteQuery() Method	ExecuteQuery() returns a set of business component records using the criteria established with methods such as SetSearchSpec.
ExecuteQuery2() Method	ExecuteQuery2() returns a set of business component records using the criteria established with methods such as SetSearchSpec.

Method or Event	Description
FirstRecord() Method	FirstRecord() moves the record pointer to the first record in a business component, making that record current and invoking any associated script events.
GetAssocBusComp() Method	GetAssocBusComp() returns the association business component. The association business component can be used to operate on the association using the normal business component mechanisms.
GetFieldValue() Method	GetFieldValue() returns the value for the field specified in its argument for the current record of the business component. Use this method to access a field value.
GetFormattedFieldValue() Method	GetFormattedFieldValue returns the value for the field specified in its argument in the current local format; that is, it returns values in the format in which they appear in the Siebel user interface.
GetMultipleFieldValues() Method	GetMultipleFieldValues() allows users to retrieve the field values for a particular record as specified in the property set input argument.
GetMVGBusComp() Method	GetMVGBusComp() returns the MVG business component associated with the business component field specified by *FieldName*. This business component can be used to operate on the Multi-Value Group using the normal business component mechanisms.
GetNamedSearch() Method	GetNamedSearch() returns the named search specification specified by *searchName*.
GetPicklistBusComp() Method	GetPicklistBusComp() returns the pick business component associated with the specified field in the current business component.
GetSearchExpr() Method	GetSearchExpr() returns the current search expression for the business component.
GetSearchSpec() Method	GetSearchSpec() returns the search specification for the field specified by the *fieldName* argument.

Method or Event	Description
GetUserProperty() Method	GetUserProperty() returns the value of a named UserProperty.
GetViewMode() Method	GetViewMode() returns the current visibility mode for the business component. This affects which records are returned by queries according to the visibility rules.
InvokeMethod() Method	InvokeMethod calls the specialized method or user-created method named in its argument.
LastRecord() Method	LastRecord() moves to the last record in the business component.
Name() Method	The Name() method returns the name of the business component.
NewRecord() Method	NewRecord() adds a new record (row) to the business component.
NextRecord() Method	NextRecord() moves the record pointer to the next record in the business component, making that the current record and invoking any associated script events.
ParentBusComp() Method	ParentBusComp() returns the parent (master) business component when given the child (detail) business component of a link.
Pick() Method	The Pick() method picks the currently selected record in a picklist business component (read "GetPicklistBusComp() Method" on page 125) into the appropriate Fields of the parent business component.
PreviousRecord() Method	PreviousRecord() moves to the previous record in the business component, invoking any associated script events.
RefineQuery() Method	This method refines a query after the query has been executed.
SetFieldValue() Method	SetFieldValue() assigns the new value to the named field for the current row of the business component.

Method or Event	Description
SetFormattedFieldValue() Method	SetFormattedFieldValue() assigns the new value to the named field for the current row of the business component. SetFormattedFieldValue accepts the field value in the current local format.
SetMultipleFieldValues() Method	SetMultipleFieldValues() allows users to set the field values for a particular record as specified in the property set input argument.
SetNamedSearch() Method	SetNamedSearch() sets a named search specification on the business component. A named search specification is identified by the *searchName* argument.
SetSearchExpr() Method	SetSearchExpr() sets an entire search expression on the business component, rather than setting one search specification for each field. Syntax is similar to that on the Predefined Queries screen.
SetSearchSpec() Method	SetSearchSpec() sets the search specification for a particular field. This method must be called before ExecuteQuery.
SetSortSpec() Method	SetSortSpec() sets the sorting specification for a query.
SetUserProperty() Method	SetUserProperty() sets the value of a named business component UserProperty. The User Properties are similar to instance variables of a BusComp.
SetViewMode() Method	SetViewMode() sets the visibility type for the business component.
UndoRecord() Method	UndoRecord() reverses any changes made to the record that are not committed. This includes reversing uncommitted modifications to any fields, as well as deleting an active record that has not yet been committed to the database.
WriteRecord() Method	WriteRecord() commits to the database any changes made to the current record.

Business Object Objects

Business objects are highly customizable, object-oriented building blocks of Siebel applications. Business objects define the relationships between different business component objects (BusComps) and contain semantic information about, for example, sales, marketing, and service-related entities. A Siebel business object groups one or more business components into a logical unit of information. Methods of business object objects are documented in the *Siebel Object Interfaces Reference*.

Method	Description
GetBusComp() Method	The GetBusComp() method returns the specified business component.
Name() Method	The Name() method retrieves the name of the business object.

Business Service Objects

Business service objects are objects that can be used to implement reusable business logic within the Object Manager. They include both built-in business services, which may be scripted but not modified, and user-defined objects. Using business services, you can configure standalone *objects* or *modules* with both properties and scripts. Business services may be used for generic code libraries that can be called from any other scripts. The code attached to a menu item or a toolbar button may be implemented as a business service. Methods of business service objects are documented in the *Siebel Object Interfaces Reference*.

Method or Event	Description
GetFirstProperty() Method	GetFirstProperty() retrieves the name of the first property of a business service.
GetNextProperty() Method	Once the name of the first property has been retrieved, the GetNext Property() method retrieves the name of the next property of a business service.
GetProperty() Method	The GetProperty() method returns the value of the property whose name is specified in its argument.
InvokeMethod() Method	The InvokeMethod() method calls a specialized method or a user-created method.
Name() Method	The Name() method returns the name of the service.
PropertyExists() Method	PropertyExists() returns a Boolean value indicating whether a specified property exists.
RemoveProperty() Method	RemoveProperty() removes a property from a business service.
Service_InvokeMethod() Event	The InvokeMethod() event is called after the InvokeMethod method is called on a business service.
Service_PreInvokeMethod() Event	The PreInvokeMethod() event is called before a specialized method is invoked on the business service.

Method or Event	Description
Service_PreCanInvokeMethod() Event	The PreInvokeMethod() event is called before the PreInvokeMethod, allowing the developer to determine whether or not the user has the authority to invoke the business service method.
SetProperty() Method	This method assigns a value to a property of a business service.

The Clib Object

The Clib object contains functions that are a part of the standard C library. Methods to access files, directories, strings, the environment, memory, and characters are part of the Clib object. The Clib object also contains time functions, error functions, sorting functions, and math functions.

Some methods, shown in Table 4, may be considered redundant because their functionality already exists in JavaScript. Where possible, you should use standard ECMAScript methods instead of the equivalent Clib functions.

NOTE: The Clib object is essentially a wrapper for calling functions in the standard C library as implemented for the specific operating system. Therefore these methods may behave differently on different operating systems.

See Also "Redundant Functions in the Clib Object" on page 131, "File I/O Functions" on page 133, "The Time Object" on page 135, "Time Functions" on page 136, "Character Classification" on page 136, "Formatting Data" on page 137

Redundant Functions in the Clib Object

The Clib object includes the functions from the C standard library. As a result, some of the methods in the Clib object overlap methods in JavaScript. In most cases, the newer JavaScript methods should be preferred over the older C functions. However, there are times, such as when working with string routines that expect null terminated strings, that the Clib methods make more sense and are more consistent in a section of a script.

The Clib methods list in Table 4 is paired with the equivalent methods in ECMAScript. Because Siebel eScript and the ECMAScript standard are developing and growing, the ECMAScript methods are always to be preferred over equivalent methods in the Clib object.

Table 4. Correspondence Between Clib and ECMAScript Methods

Clib Method	Description	ECMAScript Method
abs()	Calculates absolute value	Math.abs()
acos()	Calculates the arc cosine	Math.acos()
asin()	Calculates the arc sine	Math.asin()
atan()	Calculates the arc tangent	Math.atan()
atan2()	Calculates the arc tangent of a fraction	Math.atan2()
atof()	Converts a string to a floating-point number	Automatic conversion
atoi()	Converts a string to an integer	Automatic conversion
atol()	Converts a string to a long integer	Automatic conversion
ceil()	Rounds a number up to the nearest integer	Math.ceil()
cos()	Calculates the cosine	Math.cos()
exp()	Computes the exponential function	Math.exp()
fabs()	Computes the absolute value of a floating-point number	Math.abs()
floor()	Rounds a number down to the nearest integer	Math.floor()
fmod()	Calculates the remainder	% operator, modulo
labs()	Returns the absolute value of a long	Math.abs()
log()	Calculates the natural logarithm	Math.log()
max()	Returns the largest of one or more values	Math.max()
min()	Returns the smallest of one or more values	Math.min()
pow()	Calculates *x* to the power of *y*	Math.pow()
sin()	Calculates the sine	Math.sin()

Table 4. Correspondence Between Clib and ECMAScript Methods

Clib Method	Description	ECMAScript Method
sqrt()	Calculates the square root	Math.sqrt()
strcat()	Appends one string to another	+ operator
strcmp()	Compares two strings	= = operator
strcpy()	Copies a string	= operator
strlen()	Gets the length of a string	*string*.length
strlwr()	Converts a string to lowercase	*string*.toLowerCase
strtod()	Converts a string to decimal	Automatic conversion
strtol()	Converts a string to long	Automatic conversion
strupr()	Converts a string to uppercase	*string*.toUpperCase
tan()	Calculates the tangent	Math.tan()
tolower()	Converts a character to lowercase	*string*.toLowerCase
toupper()	Converts a character to uppercase	*string*.toUpperCase

File I/O Functions

Siebel eScript handles file I/O in a manner similar to C and C++. In these languages, files are never read from, or written to, directly. Rather, you must first open a file, most commonly by passing its name to the Clib.fopen() function. (You can also open a file using Clib.tmpfile().) These functions read the file into a buffer in memory and return a *file pointer*—a pointer to the beginning of the buffer. The data in the buffer is often referred to as a *file stream*, or simply a *stream*. Reading and writing occurs relative to the buffer, which is not written to disk unless you explicitly flush the buffer with Clib.fflush() or close the file with Clib.fclose().

Clib supports the following file I/O functions:

- "Clib.fclose() Method" on page 148
- "Clib.feof() Method" on page 148
- "Clib.fflush() Method" on page 149

- “Clib.fgetc() Method and Clib.getc() Method” on page 150
- “Clib.fgetc() Method and Clib.getc() Method” on page 150
- “Clib.fgetpos() Method” on page 151
- “Clib.fgets() Method” on page 152
- “Clib.fopen() Method” on page 154
- “Clib.fprintf() Method” on page 157
- “Clib.fputc() Method and Clib.putc() Method” on page 157
- “Clib.fputs() Method” on page 158
- “Clib.fread() Method” on page 158
- “Clib.freopen() Method” on page 160
- “Clib.fscanf() Method” on page 162
- “Clib.fseek() Method” on page 163
- “Clib.fsetpos() Method” on page 163
- “Clib.ftell() Method” on page 164
- “Clib.fwrite() Method” on page 164
- “Clib.fgetc() Method and Clib.getc() Method” on page 150
- “Clib.fputc() Method and Clib.putc() Method” on page 157
- “Clib.remove() Method” on page 185
- “Clib.rename() Method” on page 185
- “Clib.rewind() Method” on page 186
- “Clib.tmpfile() Method” on page 207

- “Clib.ungetc()Method” on page 209

NOTE: Siebel applications use UTF-16 encoding when writing to a file in Unicode. The first two bytes of the file are always the BOM (Byte Order Mark). When Clib.rewind is called on such a file, it is pointing to the BOM (-257) and not the first valid character. The user can call Clib.fgetc/getc once to skip the BOM.

The Time Object

The Clib object (like the Date object) represents time in two distinct ways: as an integral value (the number of seconds passed since January 1, 1970) and as a Time object with properties for the day, month, year, and so on. This Time object is distinct from the standard JavaScript Date object. You cannot use Date object properties with a Time object or vice versa.

Note that the Time object differs from the Date object, although they contain similar data. The Time object is for use with the other date and time functions in the Clib object. It has the integer properties listed in Table 5.

Table 5. Integer Properties of the Time Object

Value for timeInt	Integer Property
tm_sec	Second after the minute (from 0)
tm_min	Minutes after the hour (from 0)
tm_hour	Hour of the day (from 0)
tm_mday	Day of the month (from 1)
tm_mon	Month of the year (from 0)
tm_year	Years since 1900 (from 0)
tm_wday	Days since Sunday (from 0)
tm_yday	Day of the year (from 0)
tm_isdst	Daylight Savings Time flag

Time Functions

In the methods listed in Table 6, *Time* represents a variable in the Time object format, while *timeInt* represents an integer time value.

The Clib object supports the following time functions.

Table 6. Time Functions and the Objects They Return

Function	Object Returned
"Clib.asctime() Method" on page 141	*Time*
"Clib.clock() Method" on page 144	CPU tick count
"Clib.ctime() Method" on page 145	*timeInt*
"Clib.difftime() Method" on page 146	*timeInt*
"Clib.gmtime() Method" on page 168	*timeInt*
"Clib.localtime() Method" on page 175	*timeInt*
"Clib.mktime() Method" on page 179	*Time*
"Clib.strftime() Method" on page 195	*Time*
"Clib.time() Method" on page 206	*timeInt*

Character Classification

The eScript language does not have a true character type. For the character classification routines, a char is actually a one-character string. Thus, actual programming usage is very much like C. For example, in the following fragment, both .isalnum() statements work properly.

```
var t = Clib.isalnum('a');

var s = 'a';
var t = Clib.isalnum(s);
```

This fragment displays the following:

```
true
true
```

In the following fragment, both Clib.isalnum() statements cause errors because the arguments to them are strings with more than one character:

```
var t = Clib.isalnum('ab');

var s = 'ab';
var t = Clib.isalnum(s);
```

The character classification methods return Booleans: true or false. The following character classification methods are supported in the Clib object:

- "Clib.isalnum() Method" on page 169
- "Clib.isalpha() Method" on page 169
- "Clib.isascii() Method" on page 170
- "Clib.iscntrl() Method" on page 170
- "Clib.isdigit() Method" on page 170
- "Clib.isgraph() Method" on page 171
- "Clib.islower() Method" on page 171
- "Clib.isprint() Method" on page 172
- "Clib.ispunct() Method" on page 173
- "Clib.isspace() Method" on page 173
- "Clib.isupper() Method" on page 174
- "Clib.isxdigit() Method" on page 174

Formatting Data

The print family of functions and scan family of functions both use *format strings* to format the data written and read, respectively.

Formatting Output

Table 7 lists the format strings for use with the print family of functions: fprintf(), rsprintf(), and sprintf(). In these functions, characters are printed as read to standard output until a percent character (%) is reached. The percent symbol (%) indicates that a value is to be printed from the parameters following the format string. The form of the format string is as follows:

```
%[flags][width][.precision]type
```

To include the % character as a character in the format string, use two % characters together (%%).

Table 7. Format Strings for the Print Family of Functions

Formatting Character	Effect
Flag Values	
-	Left justification in the field with space padding, or right justification with zero or space padding
+	Force numbers to begin with a plus (+) or minus (-)
space	Negative values begin with a minus (-); positive values begin with a space
#	Append one of the following symbols to the # character to display the output in the indicated form: ■ o to prepend a zero to nonzero output ■ x or X to prepend 0x or 0X to the output, signifying hexadecimal ■ f to include a decimal point even if no digits follow the decimal point ■ e or E to include a decimal point even if no digits follow the decimal point, display the output in scientific notation, and remove trailing zeros ■ g or G to include a decimal point even if no digits follow the decimal point, display the output in scientific notation, and leave trailing zeros in place

Table 7. Format Strings for the Print Family of Functions

Formatting Character	Effect
Width Values	
n	At least *n* characters are output; if the value is fewer than *n* characters, the output is padded with spaces
0*n*	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 Values	
If precision is specified, then it must begin with a period (.) and must take one of the following forms:	
.0	For floating-point type, no decimal point is output
.*n*	Output is *n* characters, or *n* decimal places if the value is a floating-point number
.*	The next value in the argument list is an integer specifying the precision width
Type Values	
d,i	Signed integer
u	Unsigned integer
o	Octal integer
x	Hexadecimal integer using `0` through `9` and `a, b, c, d, e, f`
X	Hexadecimal integer using `0` through `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 number of f or e type, depending on precision
G	Floating-point number of F or E type, depending on precision
c	Character; for example, `'a', 'b', '8'`
s	String

Formatting Input

Format strings are also used with the scan family of functions: fscanf(), sscanf(), and vfscanf(). The format string contains character combinations that specify the type of data expected. 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 the function. Characters are matched against the input as read and as it matches a portion of the format string until a percent character (%) is reached. The percent character 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 taken from the next parameter in the list following the format string. A parameter specification takes this form:

```
%[*][width]type
```

The *, *width*, and *type* values may be one of the following:

*	Suppresses assigning this value to any parameter
width	Sets the maximum number of characters to read; fewer are read if a white-space or nonconvertible character is encountered

If *width* is specified, the input is an array of characters of the specified length.

Table 8 lists the characters that define the *type.*

Table 8. Type Values for the Scan Family of Functions

Type Value	Effect
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
s	String

Table 8. Type Values for the Scan Family of Functions

Type Value	Effect
[abc]	String consisting of the characters within brackets, where A–Z represents the range A to Z
[^abc]	String consisting of the character *not* within brackets

Clib.asctime() Method

This method returns a string representing the date and time extracted from a Time object.

Syntax `Clib.asctime(`*`Time`*`)`

Parameter	Description
Time	A Time object

Returns A string representing the date and time extracted from a Time object.

Usage For details on the Time object, read "The Time Object" on page 135. The returned string has the format *Day Mon dd hh:mm:ss yyyy;* for example, `Mon Jul 10 13:21:56 2000`.

See Also "Clib.ctime() Method" on page 145, "Clib.gmtime() Method" on page 168, "Clib.localtime() Method" on page 175, "Clib.mktime() Method" on page 179, "GetDate() Method" on page 212, "getTime() Method" on page 219, and "getUTCDate() Method" on page 221

Clib.bsearch() Method

This method looks for an array variable that matches a specified item.

Syntax `Clib.bsearch(`*`key, arrayToSort,`*` [`*`elementCount`*`,] `*`compareFunction`*`)`

Parameter	Description
key	The value to search for
arrayToSort	The name of the array to sort
elementCount	The number of array elements to search; if omitted, the entire array is searched
compareFunction	A user-defined function that can affect the sort order

Returns An array variable that matches *key*, returning the variable if found, null if not.

Usage Clib.bsearch() searches only through array elements with a positive index; array elements with negative indices are ignored.

The *compareFunction* value must receive the key variable as its first argument and a variable from the array as its second argument. If *elementCount* is not supplied, then the function searches the entire array.

Example The following example demonstrates the use of Clib.qsort() and Clib.bsearch() to locate a name and related item in a list:

```
(general) (ListCompareFunction)

function ListCompareFunction(Item1, Item2)
{
   return Clib.strcmpi(Item1[0], Item2[0]);
}

(general) (DoListSearch)

function DoListSearch()

   // create array of names and favorite food
   var list =
   {
      {“Brent”, “salad”},
      {"Laura", "cheese" },
      { "Alby", "sugar" },
      { "Jonathan","pad thai" },
      { "Zaza", "grapefruit" },
      { "Jordan", "pizza" }
   };

   // sort the list
   Clib.qsort(list, ListCompareFunction);
   var Key = "brent";
   // search for the name Brent in the list
   var Found = Clib.bsearch(Key, list, ListCompareFunction);
   // display name, or not found
   if ( Found != null )
      TheApplication().RaiseErrorText(Clib.rsprintf
      ("%s's favorite food is %s\n", Found[0][0],Found[0][1]));
   else
      TheApplication().RaiseErrorText("Could not find name in
list.");
}
```

See Also "Clib.qsort() Method" on page 182

Clib.chdir() Method

This method changes the current directory for the Siebel application.

Syntax `Clib.chdir(`*`dirPath`*`)`

Parameter	Description
dirpath	The path to the directory to make current

Returns 0 if successful; otherwise, -1.

Usage This method changes the current directory for the Siebel application. The default directory for a Siebel application in a Windows environment is always `c:\siebel\bin`. When the script finishes, this directory again becomes the default directory.

dirPath can be an absolute or relative path specification.

Example For an example, read "Clib.getcwd() Method" on page 166.

See Also “Clib.getcwd() Method” on page 166, “Clib.mkdir() Method” on page 178, and “Clib.rmdir() Method” on page 186

Clib.clearerr() Method

This method clears the error status and resets the end-of-file flag for a specified file.

Syntax

```
Clib.clearerr(filePointer)
```

Parameter	Description
filePointer	A pointer to the file to be cleared and reset

Returns Not applicable

Usage This method clears the error status and resets the end-of-file (EOF) flag for the file indicated by *filePointer*.

Clib.clock() Method

This method returns the current processor tick count.

Syntax

```
Clib.clock()
```

Parameter	Description
Not applicable	

Returns The current processor tick count.

Usage The count starts at 0 when the Siebel application starts running and is incremented the number of times per second determined by the operating system.

Clib.cosh() Method

This method returns the hyperbolic cosine of x.

Syntax `Clib.cosh(number)`

Parameter	Description
number	The number whose hyperbolic cosine is to be found

Returns The hyperbolic cosine of x.

See Also "Clib.sinh() Method" on page 188, "Clib.tanh() Method" on page 206, and "Math.cos() Method" on page 280

Clib.ctime() Method

This method returns a date-time value.

Syntax `Clib.ctime(timeInt)`

Parameter	Description
timeInt	A date-time value as returned by the Clib.time() function

Returns A string representing date-time value, adjusted for the local time zone.

Usage This method returns a string representing a date-time value, adjusted for the local time zone. It is equivalent to:

```
Clib.asctime(Clib.localtime(timeInt));
```

where *timeInt* is a date-time value as returned by the Clib.time() function.

Example The following line of code returns the current date and time as a string of the form *Day Mon dd hh:mm:ss yyyy*:

```
TheApplication().RaiseErrorText(Clib.ctime(Clib.time()));
```

See Also "Clib.asctime() Method" on page 141, "Clib.gmtime() Method" on page 168, "Clib.localtime() Method" on page 175, "Clib.time() Method" on page 206, and "toLocaleString() Method and toString() Method" on page 241

Clib.difftime() Method

This method returns the difference in seconds between two times.

Syntax

```
Clib.difftime(timeInt1, timeInt0)
```

Parameter	Description
timeInt0	An integer time value as returned by the Clib.time() function
timInt1	An integer time value as returned by the Clib.time() function

Returns The difference in seconds between *timeInt0* and *timeInt1*.

Example This example displays the difference in time, in seconds, between two times:

```
function difftime_Click ()
{
   var first = Clib.time();
   var second = Clib.time();
TheApplication().RaiseErrorText("Elapsed time is " +
      Clib.difftime(second, first) + " seconds.");
}
```

See Also "Clib.time() Method" on page 206, "Date.toSystem() Method" on page 242

Clib.div() Method and Clib.ldiv() Method

These methods perform integer division and return a quotient and remainder in a structure.

Syntax

```
Clib.div(numerator, denominator)
Clib.ldiv(numerator, denominator)
```

Parameter	Description
numerator	The number to be divided
denominator	The number by which *numerator* is to be divided

Returns A structure with the following elements, which are the result of dividing numerator by denominator:

.quot	quotient
.rem	remainder

Usage Because Siebel eScript does not distinguish between integers and long integers, the Clib.div() and Clib.ldiv() methods are identical.

Example The following example accepts two numbers as input from the user, divides the first by the second, and displays the result:

```
var division = Clib.div(ToNumber(n), ToNumber(d));
   TheApplication().RaiseErrorText("The quotient is " +
division.quot + ".\n\n" + "The remainder is " + division.rem +
".");
```

Clib.errno Property

The errno property stores diagnostic message information when a function fails to execute correctly.

Syntax `Clib.errno`

Parameter	Description
Not applicable	

Returns Not applicable

Usage Many functions in the Clib and SElib objects set errno to nonzero when an error occurs, to provide more specific information about the error. Siebel eScript implements errno as a macro to the internal function _errno(). This property can be accessed with Clib.strerror().

The errno property cannot be modified through eScript code. It is available only for read-only access.

Clib.fclose() Method

This method writes a file's data to disk and closes the file.

Syntax `Clib.fclose(`*`filePointer`*`)`

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns Zero if successful; otherwise, returns EOF.

Usage This method flushes the file's buffers (that is, writes its data to disk) and closes the file. The file pointer ceases to be valid after this call.

Example This example creates and writes to a text file and closes the file, testing for an error condition at the same time. If an error occurs, a message is displayed and the buffer is flushed.

```
function Test_Click ()
{
   var fp = Clib.fopen('c:\\temp000.txt', 'wt');
   Clib.fputs('abcdefg\nABCDEFG\n', fp);
   if (Clib.fclose(fp) != 0)
   {
      TheApplication().RaiseErrorText('Unable to close file.' +
         '\nContents are lost.');
   }
   else
      Clib.remove('c:\\temp000.txt');
}
```

See Also "Clib.fflush() Method" on page 149

Clib.feof() Method

This function determines whether a file cursor is at the end of a file.

Syntax `Clib.feof(`*`filePointer`*`)`

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns A nonzero integer if the file cursor is at the end of the file; 0 if it is not at the end of the file.

Usage This method determines whether the file cursor is at the end of the file indicated by *filePointer*. It returns a nonzero integer (usually 1) if true, 0 if not.

Clib.ferror() Method

This method tests and returns the error indicator for a file.

Syntax `Clib.ferror(`*`filePointer`*`)`

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns 0 if no error; otherwise, the error number.

Usage This method checks whether an error has occurred for a buffer into which a file has been read. If an error occurs, it returns the error number.

See Also “Clib.errno Property” on page 147

Clib.fflush() Method

This function writes the data in a file buffer to disk.

Syntax

```
Clib.fflush(filePointer)
```

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns 0 if successful; otherwise, EOF.

Usage This method causes any unwritten buffered data to be written to the file indicated by *filePointer*. If *filePointer* is null, this method flushes buffers in open files.

Example For an example, read “Clib.fclose() Method” on page 148.

See Also “Clib.fclose() Method” on page 148

Clib.fgetc() Method and Clib.getc() Method

These methods return the next character in a file stream.

Syntax

```
Clib.fgetc(filePointer)
Clib.getc(filePointer)
```

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns The next character in the file indicated by *filePointer* as a byte converted to an integer.

Usage These methods return the next character in a file stream—a buffer into which a file has been read. If there is a read error or the file cursor is at the end of the file, EOF is returned. If there is a read error, Clib.ferror() indicates the error condition.

See Also “Clib.fgets() Method” on page 152 and “Clib.qsort() Method” on page 182

Clib.fgetpos() Method

This method stores the current position of the pointer in a file.

Syntax `Clib.fgetpos(filePointer, position)`

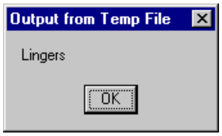
Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()
position	The current position of *filePointer*

Returns 0 if successful; otherwise, nonzero, in which case an error value is stored in the errno property.

Usage This method stores the current position of the file cursor in the file indicated by *filePointer* for future restoration using fsetpos(). The file position is stored in the variable *position*; use it with fsetpos() to restore the cursor to its position.

Example This example writes two strings to a temporary text file, using Clib.fgetpos() to save the position where the second string begins. The program then uses Clib.fsetpos() to set the file cursor to the saved position so as to display the second string, as shown in the illustration.

```
function Test_Click ()
{
   var position;
   var fp = Clib.tmpfile();
   Clib.fputs("Melody\n”, fp);
   Clib.fgetpos(fp, position)
   Clib.fputs("Lingers\n", fp);
   Clib.fsetpos(fp, position);
   TheApplication().RaiseErrorText(Clib.fgets(fp));
   Clib.fclose(fp);
}
```


Output from Temp File

Lingers

See Also "Clib.feof() Method" on page 148, "Clib.fsetpos() Method" on page 163, and "Clib.ftell() Method" on page 164

Clib.fgets() Method

This method returns a string consisting of the characters in a file from the current file cursor to the next newline character.

Syntax

```
Clib.fgets([maxLen,] filePointer)
```

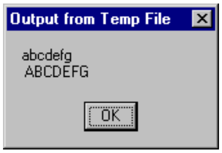
Parameter	Description
maxLen	The maximum length of the string to be returned if no newline character is encountered; if the File Mode is Unicode, the length argument is the length in Unicode characters.
filePointer	A file pointer as returned by Clib.fopen()

Returns A string consisting of the characters in a file from the current file cursor to the next newline character. If an error occurs, or if the end of the file is reached, null is returned.

Usage This method returns a string consisting of the characters in a file from the current file cursor to the next newline character. The newline is returned as part of the string.

Example This example writes a string containing an embedded newline character to a temporary file. It then reads from the file twice to retrieve the output and display it, as shown in the illustration following the example.

```
function Test_Click ()
{
   var x = Clib.tmpfile();
   Clib.fputs("abcdefg\nABCDEFG\n", x);
   Clib.rewind(x);
   TheApplication().RaiseErrorText(Clib.fgets(x) + " " +
Clib.fgets(x));
   Clib.fclose(x);
}
```



abcdefg
ABCDEFG

See Also “Clib.fputs() Method” on page 158

Clib.flock() Method

This method locks or unlocks a file for simultaneous use by multiple processes.

Syntax `Clib.flock(filePointer, mode)`

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen() or Clib.tmpfile()
mode	One of the following: ■ LOCK_EX (lock for exclusive use) ■ LOCK_SH (lock for shared use) ■ LOCK_UN (unlock) ■ LOCK_NB (non-blocking)

Returns 0 if successful; otherwise, a nonzero integer.

Usage The flock() function applies or removes an advisory lock on the file identified by *filePointer*. Advisory locks allow cooperating processes to perform consistent operations on files. However, other processes may still access the files, which can cause inconsistencies.

The locking mechanism allows two types of locks: shared and exclusive. Multiple processes can have shared locks on a file at the same time; however, there cannot be multiple exclusive locks, or shared locks and an exclusive lock, on one file at the same time.

Read permission is required on a file to obtain a shared lock, and write permission is required to obtain an exclusive lock. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect.

If a process requests a lock on an object that is already locked, the request is locked until the file is freed, unless LOCK_NB is used. If LOCK_NB is used, the call fails and the error EWOULDBLOCK is returned.

NOTE: Clib.flock() is not supported in Unicode builds. It always returns 0.

Clib.fopen() Method

This method opens a specified file in a specified mode.

Syntax `Clib.fopen(filename, mode)`

Parameter	Description
filename	Any valid filename that does not include wildcard characters
mode	One of the following characters specifying a file mode, optionally followed by one of the characters listed in Table 9.

Returns A file pointer to the file opened; null if the function fails.

Usage This function opens the file *filename*, in mode *mode*. The *mode* parameter is a string composed of “r”, “w”, or “a” followed by other characters as follows:

Table 9. File Mode Characters

Character	Mode
r	Opens the file for reading; the file must already exist
w	Opens the file for writing; the file must already exist
a	Opens the file in append mode
Optional Characters	
b	Opens the file in binary mode; if b is not specified, the file is opened in text mode (end-of-line translation is performed)
t	Opens the file in text mode
u	Opens the file in Unicode mode; for example, Clib.fopen(“filename.txt”, “rwu”)
+	Opens the file 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.

Example The following code fragment opens the text file `ReadMe` for text-mode reading and displays each line in that file:

```
var fp = Clib.fopen("ReadMe","rt");
if ( fp == null )
   TheApplication().RaiseErrorText("\aError opening file for 
reading.\n")
else
{
   while ( null != (line=Clib.fgets(fp)) )
   {
      Clib.fputs(line, stdout)
   }
}
Clib.fclose(fp);
```

Here is an example which opens a file and reads and writes a string, using the default codepage:

```
var oFile = Clib.fopen("myfile","rw");
if (null != oFile)
{
   var sHello = "Hello";
   var nLen = sHello.length;
   Clib.fputs(sHello, oFile);
   Clib.rewind(oFile);
   Clib.fgets (nLen, sHello);
}
```

Here is an example which opens a file and reads and writes a string in Unicode mode:

```
var oFile = Clib.fopen("myfile","rwu");
if (null != oFile)
{
   var sHello = "Hello";
   var nLen = sHello.length;
   Clib.fputs(sHello, oFile);
   Clib.rewind(oFile);
   Clib.fgets (nLen, sHello);
}
```

The following example specifies a file path:

```
function WebApplet_ShowControl (ControlName, Property, Mode,
&HTML)
{
if (ControlName == "GotoUrl")
   {
      var fp = Clib.fopen("c:\\test.txt","wt+");
      Clib.fputs("property = " + Property + "\n", fp);
      Clib.fputs("mode = " + Mode + "\n",fp);
      Clib.fputs("ORG HTML = " + HTML + "\n",fp);
      Clib.fclose(fp);
      HTML = "<td>New HTML code</td>";
   }
return(ContinueOperation);
```

See Also “Clib.fclose() Method” on page 148 and “Clib.tmpfile() Method” on page 207

Clib.fprintf() Method

This function writes a formatted string to a specified file.

Syntax

```
Clib.fprintf(filePointer, formatString)
```

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()
formatString	A string containing formatting instructions for each data item to be written

Returns Not applicable

Usage This method writes a formatted string to the file indicated by *filePointer*. For information on format strings used with Clib.fprintf(), see Table 7 on page 138.

See Also "Clib.rsprintf() Method" on page 187 and "Clib.sprintf() Method" on page 188

Clib.fputc() Method and Clib.putc() Method

These methods write a character, converted to a byte, to the specified file.

Syntax

```
Clib.fputc(char, filePointer)
Clib.putc(char, filePointer)
```

Parameter	Description
char	A one-character string or a variable holding a single character
filePointer	A file pointer as returned by Clib.fopen()

Returns If successful, *char;* otherwise, EOF.

Usage These methods write a single character to the file indicated by *filePointer*. If *char* is a string, the first character of the string is written to the file indicated by *filePointer*. If *char* is a number, the character corresponding to its Unicode value is written to the file.

See Also "Clib.fgetc() Method and Clib.getc() Method" on page 150 and "Clib.fputs() Method" on page 158

Clib.fputs() Method

This method writes a string to a specified file.

Syntax

```
Clib.fputs(string, filePointer)
```

Parameter	Description
string	A string literal or a variable containing a string
filePointer	A file pointer as returned by Clib.fopen()

Returns EOF if a write error occurs; otherwise, a non-negative value.

Usage This method writes the value of *string* to the file indicated by *filePointer*.

Example For an example, read "Clib.fgets() Method" on page 152.

See Also "Clib.fgets() Method" on page 152 and "Clib.fputc() Method and Clib.putc() Method" on page 157

Clib.fread() Method

This method reads data from an open file and stores it in a variable.

Syntax A

```
Clib.fread(destBuffer, bytelength, filePointer)
```

Syntax B

```
Clib.fread(destVar, varDescription, filePointer)
```

Syntax C

```
Clib.fread(blobVar, blobDescriptor, filePointer)
```

Parameter	Description
destBuffer	A variable indicating the buffer to contain the data read from the file
bytelength	The number of bytes to read
filePointer	A file pointer as returned by Clib.fopen()
destVar	A variable to contain the data read from the file
varDescription	A variable that describes how much data is to be read; must be one of the values in the list in the “Usage” section
blobVar	A variable indicating the BLOB to contain the data read from the file
blobDescriptor	The blobDescriptor for *blobVar*

Returns The number of elements read. For *destBuffer*, the number of bytes read, up to *bytelength*. For *varDescription*, 1 if the data is read, or 0 if there is a read error or EOF is encountered.

Usage This method reads data from the open file *filePointer* and stores it in the specified variable. If it does not yet exist, the variable, buffer, or BLOB is created. The *varDescription* value is a variable that describes how and how much data is to be read: if *destVar* is to hold a single datum, then varDescription must be one of the following:

UWORD8	Stored as an unsigned byte
SWORD8	Stored as a signed byte
UWORD16	Stored as an unsigned 16-bit integer
SWORD16	Stored as a signed 16-bit integer
UWORD24	Stored as an unsigned 24-bit integer
SWORD24	Stored as a signed 24-bit integer
UWORD32	Stored as an unsigned 32-bit integer
SWORD32	Stored as a signed 32-bit integer
FLOAT32	Stored as a floating-point number

FLOAT64	Stored as a double-precision floating-point number

For example, the definition of a structure might be:

```
ClientDef = new blobDescriptor();
ClientDef.Sex = UWORD8;
ClientDef.MaritalStatus = UWORD8;
ClientDef._Unused1 = UWORD16;
ClientDef.FirstName = 30; ClientDef.LastName = 40;
ClientDef.Initial = UWORD8;
```

The Siebel eScript version of 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 are read from the file in a byte-order described by the current value of the BigEndianMode global variable.

Example To read the 16-bit integer `i`, the 32-bit float `f`, and then the 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) )
   TheApplication().RaiseErrorText("Error reading from
file.\n");
}
```

See Also “Clib.fwrite() Method” on page 164

Clib.freopen() Method

This method closes the file associated with a file pointer and then opens a file and associates it with the file pointer of the file that has been closed.

Syntax `Clib.freopen(`*`filename, mode, oldFilePointer`*`)`

Parameter	Description
filename	The name of a file to be opened

Parameter	Description
mode	One of the file modes specified in the Clib.fopen() function; for Unicode, the same “u” flag as in Clib.fopen can be used
oldFilePointer	The file pointer to a file to be closed, and to which *filename* is to be associated

Returns A copy of the old file pointer after reassignment, or `null` if the function fails.

Usage This method closes the file associated with *oldFilePointer* (ignoring any close errors) and then opens *filename* according to *mode* (as in Clib.fopen()) and reassociates *oldFilePointer* to this new file specification. It is commonly used to redirect one of the predefined file handles (stdout, stderr, stdin) to or from a file.

See Also “Clib.fclose() Method” on page 148 and “Clib.fopen() Method” on page 154

Clib.frexp() Method

This method breaks a number into a normalized mantissa between 0.5 and 1.0 and calculates an integer exponent of 2 such that the number is equivalent to the mantissa * 2 ^ exponent.

Syntax `Clib.frexp(`*`number, exponent`*`)`

Parameter	Description
number	The number to be operated on
exponent	The exponent to use

Returns A normalized mantissa between 0.5 and 1.0; otherwise, 0.

Usage This method breaks *number* into a normalized mantissa between 0.5 and 1.0 and calculates an integer exponent of 2 such that *number* = = mantissa * 2 ^ *exponent*. A mantissa is the decimal part of a natural logarithm.

Clib.fscanf() Method

This function reads data from a specified file and stores the data items in a series of parameters.

Syntax `Clib.fscanf(filePointer, formatString, var1, var2, ..., varn)`

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()
formatString	A string containing formatting instructions for each data item to be read
var1, var2, ..., varn	Variables holding the values to be formatted

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.

Usage This function reads input from the file indicated by *filePointer* and stores that input in the *var1, var2, ..., var*n parameters following the *formatString* value according to 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.

Characters from input are matched against the formatting instruction characters of *formatString* until a percent character (%) is reached. The % character indicates that a value is to be read and stored to subsequent parameters following *formatString*. Each subsequent parameter after *formatString* gets the next parsed value taken from the next parameter in the list following *formatString*.

A parameter specification takes this form:

```
%[*][width]type
```

For values for these items, read "Formatting Input" on page 140.

See Also "Clib.sinh() Method" on page 188 and "Clib.sscanf() Method" on page 190

Clib.fseek() Method

This method sets the position of the file cursor of an open file.

Syntax

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

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()
offset	The number of bytes to move the file cursor beyond *mode*
mode	One of the following values: SEEK_CUR: seek is relative to the current position of the file cursor SEEK_END: seek is relative to the end of the file SEEK_SET: seek is relative to the beginning of the file

Returns 0 if successful, or nonzero if it cannot set the file cursor to the indicated position.

Usage This method sets the position of the file cursor in the file indicated by *filePointer*. If *mode* is not supplied, then the absolute offset from the beginning of the file (SEEK_SET) is assumed. For text files (that is, files not opened in binary mode), the file position may not correspond exactly to the byte offset in the file.

See Also "Clib.fgetpos() Method" on page 151, "Clib.ftell() Method" on page 164, and "Clib.rewind() Method" on page 186

Clib.fsetpos() Method

This method sets the current file cursor to a specified position.

Syntax

```
Clib.fsetpos(filePointer, position)
```

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()
position	The value returned by Clib.fgetpos(*filePointer, position*)

Returns 0 if successful; otherwise, nonzero, in which case an error value is stored in errno.

Usage This method sets the current file cursor to a specified position in the file indicated by *filePointer*. It is used to restore the file cursor to a position that has previously been retrieved by Clib.fgetpos() and stored in the *position* variable used by that method.

Example For an example, read "Clib.fgetpos() Method" on page 151.

See Also "Clib.fgetpos() Method" on page 151 and "Clib.ftell() Method" on page 164

Clib.ftell() Method

This method sets the position offset of the file cursor of an open file relative to the beginning of the file.

Syntax

```
Clib.ftell(filePointer)
```

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns The current location of the file cursor, or -1 if there is an error, in which case an error value is stored in Clib.errno.

Usage This method sets the position offset of the file cursor of the open file indicated by *filePointer* relative to the beginning of the file. For text files (that is, files not opened in binary mode), the file position may not correspond exactly to the byte offset in the file.

See Also "Clib.fseek() Method" on page 163 and "Clib.fsetpos() Method" on page 163

Clib.fwrite() Method

This method writes the data in a specified variable to a specified file and returns the number of elements written.

Syntax A `Clib.fwrite(sourceVar, varDescription, filePointer)`

Syntax B `Clib.fwrite(sourceVar, bytelength, filePointer)`

Parameter	Description
bytelength	Number of bytes to write
sourceVar	A variable indicating the source from which data is to be written
varDescription	A value depending on the type of object indicated by *sourceVar*
filePointer	A file pointer as returned by Clib.fopen()

Returns 0 if a write error occurs; use Clib.ferror() to get more information about the error.

Usage This method writes the data in *sourceVar* to the file indicated by *filePointer* and returns the number of elements written.

The varDescription variable describes how much data is to be read from the object indicated by *sourceVar:*

If sourceVar Is	Value of varDescription Is
Buffer	Length of the buffer
Object	Object descriptor
A single datum	One of the values listed in "Clib.fread() Method" on page 158

The Siebel eScript version of 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 eScript.

Example To write the 16-bit integer `i`, the 32-bit float `f`, and the 10-byte buffer `buf` into open file `fp`, use the following code:

```
   if ( !Clib.fwrite(i, SWORD16, fp) || !Clib.fwrite(f, FLOAT32, fp)
               || 10 !=  fwrite(buf, 10, fp))
   {
      TheApplication().RaiseErrorText("Error writing to file.\n");
   }
```

See Also "Clib.fread() Method" on page 158

Clib.getcwd() Method

This method returns the entire path of the current working directory for a script.

Syntax `Clib.getcwd()`

Parameter	Description
Not applicable	

Returns The entire path of the current working directory for a script.

Usage In a Siebel application, the default (current working) directory in a Windows environment is always `C:\Siebel\bin`. When a script finishes running, the default directory returns to `C:\Siebel\bin`, even if the script changes the current working directory.

Example In this example, the current directory is displayed in a message box. The script then makes the root the current directory, creates a new directory, removes that directory, and then attempts to make the removed directory current.

```
function Button_Click ()

{

   var currDir = Clib.getcwd();
   TheApplication().Trace("Current directory is " + Clib.getcwd());
   var msg = Clib.mkdir('C:\\Clib test');
   // Display the error flag created by creating directory;
   // Should be 0, indicating no error.
   TheApplication().Trace(msg);
   // Change the current directory to the new 'Clib test'
   Clib.chdir("C:\\Clib test");
```

```
    TheApplication().Trace("Current directory is " + Clib.getcwd());
    // Delete 'Clib test'
    Clib.chdir("C:\\");
    // Attempting to make a removed directory current yields error
        flag
    Clib.rmdir("Clib test");
    msg = Clib.chdir("C:\\Clib.test");
    TheApplication().Trace(msg);
}
```

The output displayed in the message boxes is as follows:

```
Current directory is C:\SIEBEL\BIN
0
Current directory is C:\Clib test
-1
```

See Also "Clib.chdir() Method" on page 143, "Clib.mkdir() Method" on page 178, and "Clib.rmdir() Method" on page 186

Clib.getenv() Method

This method returns a specified environment-variable string.

Syntax `Clib.getenv(`*`varName`*`)`

Parameter	Description
varName	The name of an environment variable

Returns The value of the named environment variable.

Usage This method returns the value of an environment variable when given its name.

Example The following line of code displays the current path:

```
TheApplication().RaiseErrorText(Clib.getenv("PATH=" + "PATH"));
```

See Also "Clib.putenv() Method" on page 181

Clib.gmtime() Method

This method converts an integer as returned by the Clib.time() function to a Time object representing the current date and time expressed as Greenwich Mean Time (GMT).

Syntax

```
Clib.gmtime(timeInt)
```

Parameter	Description
timeInt	A date-time value as returned by the Clib.time() function

Returns

A Time object representing the current date and time expressed as Greenwich Mean Time.

Usage

This method converts an integer as returned by the Clib.time() function to a Time object representing the current date and time expressed as Greenwich Mean Time (GMT). For details on the Time object, read “The Time Object” on page 135.

NOTE: The line of code

```
      var now = Clib.asctime(Clib.gmtime(Clib.time())) + "GMT";
```

is exactly equivalent to the standard JavaScript construction

```
      var aDate = new Date;
      var now = aDate.toGMTString()
```

Wherever possible, the second form should be used.

Example

The following line of code returns the current GMT date and time as a string in the form *Day Mon dd hh:mm:ss yyyy*.

```
TheApplication().RaiseErrorText(Clib.asctime(Clib.gmtime(Clib.ti
me())));
```

See Also

“Clib.asctime() Method” on page 141, “Clib.ctime() Method” on page 145, “Clib.localtime() Method” on page 175, “Clib.mktime() Method” on page 179, “GetDate() Method” on page 212, “getTime() Method” on page 219, “getUTCDate() Method” on page 221, and “toGMTString() Method” on page 240

Clib.isalnum() Method

This function returns true if a specified character is alphanumeric.

Syntax `Clib.isalnum(char)`

Parameter	Description
char	A single character, or a variable containing a single character

Returns True if *char* is an alphabetic character from A through Z or a through z, or is a digit from 0 through 9; otherwise, false.

Usage This function returns true if *char* is alphanumeric. Otherwise, it returns false.

See Also "Clib.isalpha() Method" on page 169, "Clib.islower() Method" on page 171, "Clib.isprint() Method" on page 172, "Clib.isupper() Method" on page 174, and "Clib.isdigit() Method" on page 170

Clib.isalpha() Method

This function returns true if a specified character is alphabetic.

Syntax `Clib.isalpha(char)`

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is an alphabetic character from A to Z or a to z; otherwise, false.

Usage This function returns true if *char* is alphabetic; otherwise, it returns false.

See Also "Clib.isdigit() Method" on page 170, "Clib.isalnum() Method" on page 169, "Clib.islower() Method" on page 171, "Clib.isprint() Method" on page 172, and "Clib.isupper() Method" on page 174

Clib.isascii() Method

This function returns true if a specified character has an ASCII code from 0 to 127.

Syntax

```
Clib.isascii(char)
```

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is has an ASCII code from 0 through 127; otherwise, false.

Usage This function returns true if *char* is a character in the standard ASCII character set, with codes from 0 through 127; otherwise, it returns false.

See Also "Clib.iscntrl() Method" on page 170 and "Clib.isprint() Method" on page 172

Clib.iscntrl() Method

This function returns true if a specified character is a control character.

Syntax

```
Clib.iscntrl(char)
```

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a control character; otherwise, false.

Usage This function returns true if *char* is a control character, that is, one having an ASCII code from 0 through 31; otherwise, it returns false.

See Also "Clib.isascii() Method" on page 170

Clib.isdigit() Method

This function returns true if a specified character is a decimal digit.

Syntax `Clib.isdigit(`*`char`*`)`

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a decimal digit from 0 through 9; otherwise, false.

Usage This function returns true if *char* is a decimal digit from 0 through 9; otherwise, it returns false.

See Also "Clib.isalnum() Method" on page 169, "Clib.isalpha() Method" on page 169, and "Clib.isupper() Method" on page 174

Clib.isgraph() Method

This function returns true if a specified character is a printable character other than a space.

Syntax `Clib.isgraph(`*`char`*`)`

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a printable character other than the space character; otherwise, false.

Usage This function returns true if *char* is a printable character other than the space character (ASCII code 32); otherwise, it returns false.

See Also "Clib.isprint() Method" on page 172, "Clib.ispunct() Method" on page 173, and "Clib.isspace() Method" on page 173

Clib.islower() Method

This function returns true if a specified character is a lowercase alphabetic character.

Syntax

```
Clib.islower(char)
```

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a lowercase alphabetic character; otherwise, false.

Usage This function returns true if *char* is a lowercase alphabetic character from a through z; otherwise, it returns false.

See Also "Clib.isalnum() Method" on page 169, "Clib.isalpha() Method" on page 169, and "Clib.isupper() Method" on page 174

Clib.isprint() Method

This function returns true if a specified character is printable.

Syntax

```
Clib.isprint(char)
```

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a printable character that can be typed from the keyboard; otherwise, false.

Usage This function returns true if *char* is a printable character that can be typed from the keyboard (ASCII codes 32 through 126); otherwise, it returns false.

See Also "Clib.isalnum() Method" on page 169, "Clib.isascii() Method" on page 170, "Clib.isgraph() Method" on page 171, "Clib.ispunct() Method" on page 173, and "Clib.isspace() Method" on page 173

Clib.ispunct() Method

This function returns true if a specified character is a punctuation mark that can be entered from the keyboard.

Syntax `Clib.ispunct(char)`

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a punctuation mark that can be entered from the keyboard (ASCII codes 33 through 47, 58 through 63, 91 through 96, or 123 through 126); otherwise, it returns false.

See Also "Clib.isgraph() Method" on page 171, "Clib.isprint() Method" on page 172, and "Clib.isspace() Method" on page 173

Clib.isspace() Method

This function returns true if a specified character is a white-space character.

Syntax `Clib.isspace(char)`

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a white-space character; otherwise, false.

Usage This function returns true if *char* is a horizontal tab, newline, vertical tab, form feed, carriage return, or space character (that is, one having an ASCII code of 9, 10, 11, 12, 13, or 32); otherwise, it returns false.

See Also "Clib.isascii() Method" on page 170 and "Clib.isprint() Method" on page 172

Clib.isupper() Method

This function returns true if a specified character is an uppercase alphabetic character.

Syntax

```
Clib.isupper(char)
```

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is an uppercase alphabetic character; otherwise, false.

Usage This function returns true if *char* is an uppercase alphabetic character from A through Z; otherwise, it returns false.

See Also "Clib.isalpha() Method" on page 169 and "Clib.islower() Method" on page 171

Clib.isxdigit() Method

This function returns true if a specified character is a hexadecimal digit.

Syntax

```
Clib.isxdigit(char)
```

Parameter	Description
char	A single character or a variable containing a single character

Returns True if *char* is a hexadecimal digit; otherwise, false.

Usage This function evaluates a single character, returning true if the character is a valid hexadecimal character (that is, a number from 0 through 9 or an alphabetic character from a through f or A through F). If the character is not in one of the legal ranges, it returns false.

See Also "Clib.isdigit() Method" on page 170

Clib.ldexp() Method

This method calculates a floating-point number given a mantissa and exponent.

Syntax `Clib.ldexp(`*`mantissa, exponent`*`)`

Parameter	Description
mantissa	The number to be operated on
exponent	The exponent to use

Returns The result of the calculation.

Usage This method is the inverse of .frexp() and calculates a floating-point number from the following equation:

```
mantissa * 2 ^ exponent
```

A mantissa is the decimal part of a natural logarithm.

See Also "Clib.frexp() Method" on page 161

Clib.localtime() Method

This method returns a value as a Time object.

Syntax `Clib.localtime(`*`timeInt`*`)`

Parameter	Description
timeInt	A date-time value as returned by the Clib.time() function

Returns *The value of timeInt* (as returned by the time() function) as a Time object.

Usage This method returns the value *timeInt* (as returned by the time() function) as a Time object. For details on the Time object, read "The Time Object" on page 135.

NOTE: The line of code

```
var now = Clib.asctime(Clib.localtime(Clib.time()));
```

is exactly equivalent to the standard JavaScript construction

```
var aDate = new Date;
var now = aDate.toLocaleString()
```

Wherever possible, use the second form.

See Also "Clib.asctime() Method" on page 141, "Clib.ctime() Method" on page 145, "Clib.gmtime() Method" on page 168, "Clib.mktime() Method" on page 179, "GetDate() Method" on page 212, "getTime() Method" on page 219, "getUTCDate() Method" on page 221, and "toLocaleString() Method and toString() Method" on page 241

Clib.memchr() Method

This method searches a buffer and returns the first occurrence of a specified character.

Syntax `Clib.memchr(bufferVar, char[, size])`

Parameter	Description
bufferVar	A buffer, or a variable pointing to a buffer
char	The character to find
size	The amount of the buffer to search, in bytes

Returns Null if *char* is not found in *bufferVar*; otherwise, a buffer that begins at the first instance of *char* in *bufferVar.*

Usage This method searches *bufferVar* and returns the first occurrence of *char.* If *size* is not specified, the method searches the entire buffer from element 0.

Clib.memcmp() Method

This method compares the contents of two buffers or the length of two buffers.

Syntax `Clib.memcmp(buf1, buf2[, length])`

Parameter	Description
buf1	A variable containing the name of the first buffer to be compared
buf2	A variable containing the name of the second buffer to be compared
length	The number of bytes to compare

Returns A negative number if *buf1* is less than *buf2*, 0 if *buf1* is the same as *buf2* for *length* bytes, a positive number if *buf1* is greater than *buf2*.

Usage If *length* is not specified, Clib.memcmp() compares the length of the two buffers. It then compares the contents up to the length of the shorter buffer. If *length* is specified and one of the buffers is shorter than *length*, comparison proceeds up to the length of the shorter buffer.

Clib.memcpy() Method and Clib.memmove() Method

These methods copy a specified number of bytes from one buffer to another.

Syntax `Clib.memcpy(destBuf, srcBuf[, length])`

`Clib.memmove(destBuf, srcBuf[, length])`

Parameter	Description
destBuf	The buffer to copy to
srcBuf	The buffer to copy from
length	The number of bytes to copy

Returns Not applicable

Usage These methods copy the number of bytes specified by *length* from *srcBuf* to *destBuf*. If *destBuf* has not already been defined, it is created as a buffer. If the *length* is not supplied, the entire contents of *srcBuf* are copied to d*estBuf*.

Siebel eScript protects data from being overwritten; therefore, in Siebel eScript Clib.memcpy() method is the same as Clib.memmove().

Clib.memset() Method

This method fills a specified number of bytes in a buffer with a specified character.

Syntax

```
Clib.memset(bufferVar, char[, length])
```

Parameter	Description
bufferVar	A buffer or a variable containing a buffer
char	The character to fill the buffer with
length	The number of bytes in which *char* is to be written

Returns Not applicable

Usage This method fills a buffer with *length* bytes of *char*. If the buffer has not already been defined, it is created as a buffer of *length* bytes. If *bufferVar* is shorter than *length,* its size is increased to *length*. If *length* is not supplied, it defaults to the size of *bufferVar*, starting at index 0.

Clib.mkdir() Method

This method creates a directory.

Syntax

```
Clib.mkdir(dirpath)
```

Parameter	Description
dirpath	A string containing a valid directory path

Returns 0 if successful; otherwise, -1.

Usage This method creates a directory. If no path is specified, the directory is created in `C:\siebel\bin`. The specified directory may be an absolute or relative path specification.

Example For an example, read "Clib.getcwd() Method" on page 166.

See Also "Clib.chdir() Method" on page 143, "Clib.getcwd() Method" on page 166, and "Clib.rmdir() Method" on page 186

Clib.mktime() Method

This method converts a Time object to the time format returned by Clib.time().

Syntax `Clib.mktime(Time)`

Parameter	Description
Time	A Time object

Returns An integer representation of the value stored in *Time,* or -1 if *Time* cannot be converted or represented.

Usage Undefined elements of Time are set to 0 before the conversion. This function is the inverse of Clib.localtime(), which converts from a time integer to a Time object. For details on the Time object, read "The Time Object" on page 135.

See Also "Clib.asctime() Method" on page 141, "Clib.ctime() Method" on page 145, "Clib.gmtime() Method" on page 168, "Clib.localtime() Method" on page 175, "GetDate() Method" on page 212, "getTime() Method" on page 219, and "getUTCDate() Method" on page 221

Clib.modf() Method

This method returns the integer part of a decimal number.

Syntax `Clib.modf(`*`number`*`, var `*`intVar`*`)`

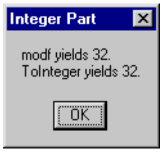
Parameter	Description
number	The floating-point number to be split
intVar	A variable to hold the integer part of *number*

Returns The integer part of *number*, stored in *intVar*.

Usage This method returns the integer part of a decimal number. Its effect is identical to that of ToInteger(*number*).

Example This example passes the same value to Clib.modf() and ToInteger(). As the illustration shows, the result is the same:

```
function eScript_Click ()
{
   Clib.modf(32.154, var x);
   var y = ToInteger(32.154);
   TheApplication().RaiseErrorText("modf yields " + x +
         ".\nToInteger yields " + y + ".");
}
```


Integer Part

modf yields 32.
ToInteger yields 32.

See Also "ToInteger() Method" on page 266

Clib.perror() Method

This method prints and returns an error message that describes the error defined by Clib.errno.

Syntax `Clib.perror([errmsg])`

Parameter	Description
errmsg	A message to describe an error condition

Returns A string containing an error message that describes the error indicated by Clib.errno.

Usage This method is identical to calling Clib.strerror(Clib.errno). If a string variable is supplied, it is set to the string returned.

Clib.putenv() Method

This method creates an environment variable, sets the value of an existing environment variable, or removes an environment variable.

Syntax `Clib.putenv(varName, stringValue)`

Parameter	Description
varName	The name of an environment variable
stringValue	The value to be assigned to the environment variable

Returns 0 if successful; otherwise, -1.

Usage This method sets the environment variable *varName* to the value of *stringValue*. If *stringValue* is null, then *varName* is removed from the environment.

The environment variable change persists only while the Siebel eScript code and its child processes are executing. After execution, the variable is destroyed automatically.

See Also "Clib.getenv() Method" on page 167

Clib.qsort() Method

This method sorts elements in an array.

Syntax `Clib.qsort(`*`array`*`, [`*`elementCount`*`, ]`*`compareFunction`*`)`

Parameter	Description
array	An array to sort
elementCount	The number of elements in the array, up to 65,536
compareFunction	A user-defined function that can affect the sort order

Returns Not applicable

Usage This method sorts elements in an array, starting from index 0 to *elementCount*-1. If *elementCount* is not supplied, the method sorts the entire array. This method differs from the *Array*.sort() method in that it can sort dynamically created arrays, whereas *Array*.sort() works only with arrays explicitly created with a new Array statement.

Example The following example prints a list of colors sorted in reverse alphabetical order, ignoring case:

```
// initialize an array of colors
var colors = { "yellow", "Blue", "GREEN", "purple", "RED",
"BLACK", "white", "orange" };
// sort the list using qsort and our ColorSorter routine
Clib.qsort(colors,"ReverseColorSorter");
// display the sorted colors
for ( var i = 0; i <= getArrayLength(colors); i++ )
   Clib.puts(colors[i]);

function ReverseColorSorter(color1, color2)
// do a simple case insensitive string
// comparison, and reverse the results too
{
   var CompareResult = Clib.stricmp(color1,color2)
   return( _CompareResult );
}
```

The output of the preceding code would be:

```
yellow
white
RED
purple
orange
GREEN
Blue
BLACK
```

See Also “sort() Method” on page 101

quot Method

This method is used to find the quotient after a division operation.

Syntax *intVar*.quot

Placeholder	Description
intVar	Any variable containing an integer

Returns The quotient part of a division operation performed by Clib.div() or Clib.ldiv().

Usage This method is used in conjunction with the Clib.div() or Clib.ldiv() functions. For details, read “Clib.div() Method and Clib.ldiv() Method” on page 146.

Example For an example, read “Clib.div() Method and Clib.ldiv() Method” on page 146.

See Also “Clib.div() Method and Clib.ldiv() Method” on page 146 and “rem Method” on page 184

Clib.rand() Method

This method generates a random number between 0 and RAND_MAX, inclusive.

Syntax

```
Clib.rand()
```

Parameter	Description
Not applicable	

Returns A pseudo-random number between 0 and RAND_MAX, inclusive. The value of RAND_MAX depends upon the operating system, but is typically 32,768.

Usage The sequence of pseudo-random numbers is affected by the initial generator seed and by earlier calls to Clib.rand(). For information about the initial generator seed, read "Clib.srand() Method" on page 189.

See Also "Clib.srand() Method" on page 189 and "Math.random() Method" on page 288

rem Method

This method is used to find the remainder after a division operation.

Syntax

```
intVar.rem
```

Placeholder	Description
intVar	Any variable containing an integer

Returns The remainder part of the result of a division operation performed by Clib.div() or Clib.ldiv().

Usage This method is used in conjunction with the Clib.div() or Clib.ldiv() function. For details, read "Clib.div() Method and Clib.ldiv() Method" on page 146.

Example For an example, read "Clib.div() Method and Clib.ldiv() Method" on page 146.

See Also "Clib.div() Method and Clib.ldiv() Method" on page 146 and "quot Method" on page 183

Clib.remove() Method

This method deletes a specified file.

Syntax `Clib.remove(filename)`

Parameter	Description
filename	A string or string variable containing the name of the file to be deleted

Returns 0 if successful; otherwise, -1.

Usage *The filename* parameter may be either an absolute or a relative filename.

Example For an example, read "Clib.fclose() Method" on page 148.

See Also "Clib.fopen() Method" on page 154

Clib.rename() Method

This method renames a file.

Syntax `Clib.rename(oldName, newName)`

Parameter	Description
oldName	A string representing the name of the file to be renamed
newName	A string representing the new name to give the file

Returns 0 if successful; otherwise, -1.

Usage This method renames a file. The *oldName* parameter may be either an absolute or a relative filename.

Clib.rewind() Method

This method sets the file cursor to the beginning of a file.

Syntax

```
Clib.rewind(filePointer)
```

Parameter	Description
filePointer	A file pointer as returned by Clib.fopen()

Returns Not applicable

Usage This call is identical to Clib.fseek(*filePointer,* 0, SEEK_SET) except that it also clears the error indicator for the file indicated by *filePointer.*

NOTE: Siebel applications use UTF-16 encoding when writing to a file in Unicode. The first two bytes of the file are always the BOM (Byte Order Mark). When Clib.rewind is called on such a file, it is pointing to the BOM (-257) and not the first valid character. The user can call Clib.fgetc/getc once to skip the BOM.

Example For an example, read "Clib.fgets() Method" on page 152.

See Also "Clib.fseek() Method" on page 163

Clib.rmdir() Method

This method removes a specified directory.

Syntax

```
Clib.rmdir(dirpath)
```

Parameter	Description
dirpath	The directory to be removed

Returns 0 if successful; otherwise, -1.

Usage The *dirpath* parameter may be an absolute or relative path specification.

Example For an example, read "Clib.getcwd() Method" on page 166.

See Also "Clib.chdir() Method" on page 143, "Clib.getcwd() Method" on page 166, and "Clib.mkdir() Method" on page 178

Clib.rsprintf() Method

This method returns a formatted string.

Syntax

```
Clib.rsprintf([formatString] [,var1, var2, ..., varn])
```

Parameter	Description
formatString	A string indicating how variable or literal parameters are to be treated
var1, var2, ..., varn	Variables to be printed using the *formatString*

Returns A string formatted according to *formatString*.

Usage Clib.rsprintf() can return string or numeric literals that appear as parameters.

The format string contains character combinations indicating how following parameters are to be treated. For information on format strings used with Clib.rsprintf(), see Table 7 on page 138 in the section "Formatting Data" on page 137. If there are variable parameters, the number of formatting sequences must match the number of variables.

Characters are returned as read until a percent character (%) is reached. The percent character indicates that a value is to be printed from the parameters following the format string.

Example Each of the following lines shows an rsprintf example followed by the resulting string:

```
Clib.rsprintf("I count: %d %d %d.",1,2,3) //"I count: 1 2 3"
var a = 1;
var b = 2;
Clib.rsprintf("%d %d %d",a, b, a+b)       //"1 2 3"
```

See Also “Clib.sprintf() Method” on page 188

Clib.sinh() Method

This method returns the hyperbolic sine of a floating-point number.

Syntax `Clib.sinh(floatNum)`

Parameter	Description
floatNum	A floating-point number, or a variable containing a floating-point number, whose hyperbolic sine is to be found

Returns The hyperbolic sine of *floatNum*.

See Also “Clib.cosh() Method” on page 144 and “Clib.tanh() Method” on page 206

Clib.sprintf() Method

This method writes output to a string variable according to a prescribed format.

Syntax `Clib.sprintf(stringVar, formatString, var1, var2, ..., varn)`

Parameter	Description
stringVar	The string variable to which the output is assigned
formatString	A string indicating how variable or literal parameters are to be treated
var1, var2, ..., varn	Variables to be formatted using the *formatString*

Returns The number of characters written into buffer if successful; otherwise, EOF.

Usage This method formats the values in the variables according to *formatString* and assigns the result to *stringVar*. The *formatString* contains character combinations indicating how following parameters are to be treated. For information on format strings used with Clib.sprintf(), see Table 7 on page 138 in the section “Formatting Data” on page 137. The string value need not be previously defined; it is created large enough to hold the result. Characters are printed as read to standard output until a percent character (%) is reached. The percent character indicates that a value is to be printed from the parameters following the format string.

Example Each of the following lines shows an sprintf example followed by the resulting string:

```
var testString;
Clib.sprintf(testString, "I count: %d %d %d.",1,2,3)
//"I count: 1 2 3"
var  a = 1;
var  b = 2;
Clib.sprintf(testString, "%d %d %d",a, b, a+b)       //"1 2 3"
```

See Also “Clib.rsprintf() Method” on page 187

Clib.srand() Method

This method initializes a random number generator.

Syntax `Clib.srand(`*`seed`*`)`

Parameter	Description
seed	A number for the random number generator to start with

Returns Not applicable

Usage If *seed* is not supplied, then a random seed is generated in a manner that is specific to the operating system in use.

See Also “Clib.rand() Method” on page 183 and “Math.random() Method” on page 288

Clib.sscanf() Method

This method reads input from the standard input device and stores the data in variables provided as parameters.

Syntax

```
Clib.sscanf([formatString] [,var1, var2, ..., varn])
```

Parameter	**Description**
formatString	A string indicating how variable or literal parameters are to be treated
v*ar1, var2, ..., var*n	Variables in which to store the input

Returns EOF if input failure occurs before any conversion occurs; otherwise, the number of variables assigned data.

Usage This method reads input from the standard input stream (the keyboard unless some other file has been redirected as stdin by the Clib.freopen() function) and stores the data read in the variables provided as parameters following *formatString*. The data is stored according to the character combinations in *formatString* which indicate how the input data is to be read and stored.

This method is identical to calling fscanf() with `stdin` as the first parameter. It returns the number of input items assigned; this number may be fewer than the number of parameters requested if there is a matching failure. If there is a conversion failure, EOF is returned.

The *formatString* value 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. The input is not read until the ENTER key is pressed.

Characters from input are matched against the characters of the *formatString* until a percent character (%) is reached. The percent character indicates that a value is to be read and stored to subsequent parameters following *formatString*. Each subsequent parameter after *formatString* gets the next parsed value taken from the next parameter in the list following *formatString*.

A parameter specification takes this form:

```
%[*][width]type
```

For values for these items, read “Formatting Input” on page 140.

See Also “Formatting Data” on page 137, “Clib.fscanf() Method” on page 162, and “Clib.sinh() Method” on page 188

Clib.strchr() Method

This method searches a string for a specified character.

Syntax `Clib.strchr(string, char)`

Parameter	Description
string	A string literal, or string variable, containing the character to be searched for
char	The character to search for

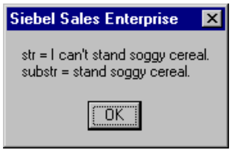
Returns The offset from the beginning of *string* of the first occurrence of *char* in *string*; otherwise, null if *char* is not found in *string*.

Usage This method searches the parameter *string* for the character *char*. When possible, you should use the standard JavaScript method substring() (read “string.replace() Method” on page 311).

Example The following code fragment:

```
var str = "I can't stand soggy cereal."
var substr = Clib.strchr(str, 's');
TheApplication().RaiseErrorText("str = " + str + "\nsubstr = " +
substr);
```

results in the following output.



str = I can't stand soggy cereal.
substr = stand soggy cereal.

See Also “Clib.strcspn() Method” on page 192, “Clib.strpbrk() Method” on page 200, “Clib.strrchr() Method” on page 201, and “string.replace() Method” on page 311

Clib.stricmp() Method and Clib.strcmpi() Method

These methods make a case-insensitive comparison of two strings.

Syntax

```
Clib.stricmp(string1, string2)
Clib.strcmpi(string1, string2)
```

Parameter	Description
string1	A string, or a variable containing a string, to be compared with *string2*
string2	A string, or a variable containing a string, to be compared with *string1*

Returns The result of the comparison, which is 0 if the strings are identical, a negative number if the ASCII code of the first unmatched character in *string1* is less than that of the first unmatched character in *string2*, or a positive number if the ASCII code of the first unmatched character in *string1* is greater than that of the first unmatched character in *string2*.

Usage These methods continue to make a case-insensitive comparison, one byte at a time, of *string1* and *string2* until there is a mismatch or the terminating null byte is reached.

See Also “Clib.strncmp() Method” on page 198 and “Clib.strncmpi() Method and Clib.strnicmp() Method” on page 198

Clib.strcspn() Method

This method searches a string for any of a group of specified characters.

Syntax

```
Clib.strcspn(string, charSet)
```

Parameter	Description
string	A literal string, or a variable containing a string, to be searched
charSet	A literal string, or a variable containing a string, which contains the set of characters to search for

Returns If no matching characters are found, the length of the string; otherwise, the offset of the first matching character from the beginning of *string*.

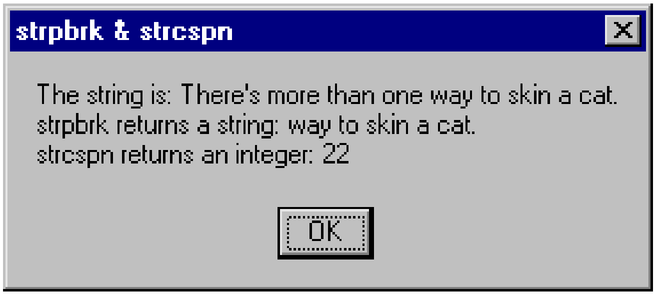
Usage This method searches the parameter string for any of the characters in the string *charSet*, and returns the offset of that character. This method is similar to Clib.strpbrk(), except that Clib.strpbrk() returns the string beginning at the first character found, while Clib.strcspn() returns the offset number for that character.

When possible, you should use the standard JavaScript method substring() (read "string.replace() Method" on page 311).

Example The following fragment demonstrates the difference between Clib.strcspn() and Clib.strpbrk():

```
var string = "There's more than one way to skin a cat.";
var rStrpbrk = Clib.strpbrk(string, "dxb8w9k!");
var rStrcspn = Clib.strcspn(string, "dxb8w9k!");
TheApplication().RaiseErrorText("The string is: " +  string +
   "\nstrpbrk returns a string: " +  rStrpbrk +
   "\nstrcspn returns an integer: " +  rStrcspn);
```

This code results in the following output:



```
The string is: There's more than one way to skin a cat.
strpbrk returns a string: way to skin a cat.
strcspn returns an integer: 22
```

See Also "Clib.strchr() Method" on page 191, "Clib.strpbrk() Method" on page 200, and "string.replace() Method" on page 311

Clib.strerror() Method

This method returns the error message associated with a Clib-defined error number.

Syntax `Clib.strerror(`*`errno`*`)`

Parameter	Description
errno	The error number returned by Clib.errno

Returns The descriptive error message associated with an error number returned by Clib.errno.

Usage When some functions fail to execute properly, they store a number in the Clib.errno property. The number corresponds to the type of error encountered. This method converts the error number to a descriptive string and returns it.

See Also "Clib.errno Property" on page 147

Clib.strftime() Method

This method creates a string that describes the date or the time or both, and stores it in a variable.

Syntax

```
Clib.strftime(stringVar, formatString, Time)
```

Parameter	Description
stringVar	A variable to hold the string representation of the time
formatString	A string that describes how the value stored in *stringVar* is formatted, using the conversion characters listed in the Usage topic
Time	A time object as returned by Clib.localtime()

Returns A formatted string as described by *formatString*.

Usage For details on the Time object, read "The Time Object" on page 135. The following conversion characters are used with Clib.strftime() to indicate time and date output:

%a	Abbreviated weekday name (Sun)
%A	Full weekday name (Sunday)
%b	Abbreviated month name (Dec)
%B	Full month name (December)
%c	Date and time (Dec 2 06:55:15 1979)
%d	Two-digit day of the month (02)
%H	Two-digit hour of the 24-hour day (06)
%I	Two-digit hour of the 12-hour day (06)
%j	Three-digit day of the year from 001 (335)
%m	Two-digit month of the year from 01 (12)
%M	Two-digit minute of the hour (55)
%p	AM or PM (AM)
%S	Two-digit seconds of the minute (15)
%U	Two-digit week of the year where Sunday is the first day of the week (48)

%w	Day of the week where Sunday is 0 (0)
%W	Two-digit week of the year where Monday is the first day of the week (47)
%x	The date (Dec 2 1979)
%X	The time (06:55:15)
%y	Two-digit year of the century (79)
%Y	The year (1979)
%Z	The name of the time zone, if known (EST)
%%	The percent character (%)

Example The following example displays the full day name and month name of the current day:

```
var TimeBuf;
Clib.strftime(TimeBuf,"Today is %A, and the month is %B",
   Clib.localtime(Clib.time()));
TheApplication().RaiseErrorText(TimeBuf);
```

See Also “Clib.asctime() Method” on page 141 and “Clib.localtime() Method” on page 175

Clib.strlwr() Method

This method converts a string to lowercase.

Syntax `Clib.strlwr(`*`str`*`)`

Parameter	Description
str	The string in which to change case of characters to lowercase.

Returns String - the value of *str* after conversion of case.

Usage This method converts uppercase letters in *str* to lowercase, starting at *str*[0] and ending before the terminating null byte. The return is the value of *str*, which is a variable pointing to the start of *str* at *str*[0].

Clib.strncat() Method

This method appends a specified number of characters from one string to another string.

Syntax

```
Clib.strncat(destString, sourceString, maxLen)
```

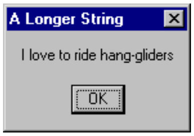
Parameter	Description
destString	The string to which characters are to be added
sourceString	The string from which characters are to be added
maxLen	The maximum number of characters to add

Returns The string in *destString* after the characters have been appended.

Usage This method appends up to *maxLen* characters of *sourceString* onto the end of *destString*. Characters following a null byte in *sourceString* are not copied. The length of *destString* is the lesser of *maxLen* and the length of *sourceString*.

Example This example returns the string "`I love to ride hang-gliders`":

```
var string1 = "I love to ";
var string2 = "ride hang-gliders and motor scooters.";
Clib.strncat(string1, string2, 17);
TheApplication().RaiseErrorText(string1);
```



I love to ride hang-gliders

See Also "Clib.strncpy() Method" on page 199

Clib.strncmp() Method

This method makes a case-sensitive comparison of two strings up to a specified number of bytes until there is a mismatch or it reaches the end of a string.

Syntax

```
Clib.strncmp(string1, string2, maxLen)
```

Parameter	Description
string1	A string, or a variable containing a string, to be compared with *string2*
string2	A string, or a variable containing a string, to be compared with *string1*
maxLen	The number of bytes to compare

Returns

The result of the comparison, which is 0 if the strings are identical, a negative number if the ASCII code of the first unmatched character in *string1* is less than that of the first unmatched character in *string2*, or a positive number if the ASCII code of the first unmatched character in *string1* is greater than that of the first unmatched character in *string2*.

Usage

This method compares up to *maxLen* bytes of *string1* against *string2* until there is a mismatch or it reaches the end of a string. The comparison is case-sensitive. The comparison ends when *maxLen* bytes have been compared or when a terminating null byte has been reached, whichever comes first.

See Also

"Clib.stricmp() Method and Clib.strcmpi() Method" on page 192 and
"Clib.strncmpi() Method and Clib.strnicmp() Method" on page 198

Clib.strncmpi() Method and Clib.strnicmp() Method

These methods make a case-insensitive comparison between two strings, up to a specified number of bytes.

Syntax

```
Clib.strncmpi(string1, string2, maxLen)
Clib.strncmpi(string1, string2, maxLen)
```

Parameter	Description
string1	A string, or a variable containing a string, to be compared with *string2*
string2	A string, or a variable containing a string, to be compared with *string1*
maxLen	The number of bytes to compare

Returns The result of the comparison, which is 0 if the strings are identical, a negative number if the ASCII code of the first unmatched character in *string1* is less than that of the first unmatched character in *string2*, or a positive number if the ASCII code of the first unmatched character in *string1* is greater than that of the first unmatched character in *string2*.

Usage This method compares up to *maxLen* bytes of *string1* against *string2* until there is a mismatch or it reaches the end of a string. This method does a case-insensitive comparison, so that `A` and `a` are considered to be the same. The comparison ends when *maxLen* bytes have been compared or when an end of string has been reached, whichever comes first.

See Also "Clib.stricmp() Method and Clib.strcmpi() Method" on page 192 and "Clib.strncmp() Method" on page 198

Clib.strncpy() Method

This method copies a specified number of characters from one string to another.

Syntax

```
Clib.strncpy(destString, sourceString, maxLen)
```

Parameter	Description
destString	The string to which characters are to be added
sourceString	The string from which characters are to be added
maxLen	The maximum number of characters to add

Returns The ASCII code of the first character of *destString*.

Usage This method copies characters from *sourceString* to *destString*. The number of characters copied is the lesser of *maxLen* and the length of *sourceString*. If *MaxLen* is greater than the length of *sourceString*, the remainder of *destString* is filled with null bytes. A null byte is appended to *destString* if *MaxLen* bytes are copied. If *destString* is not already defined, the function defines it. It is safe to copy from one part of a string to another part of the same string.

See Also "Clib.strncat() Method" on page 197

Clib.strpbrk() Method

This method searches a string for any of several specified characters and returns the string beginning at the first instance of one of the specified characters.

Syntax

```
Clib.strpbrk(string, charSet)
```

Parameter	Description
string	A string variable or literal containing the string from which the substring is to be extracted
charSet	A string variable or literal containing a group of characters, any one of which may be the starting character for the substring

Returns The string beginning at the first instance of one of the specified characters in the *charSet parameter*; otherwise, null, if none is found.

Usage This method searches *string* for any of the characters specified in *charSet.*

When possible, you should use the standard JavaScript method substring() (read "string.replace() Method" on page 311).

Example For an example using this function, read "Clib.strcspn() Method" on page 192. To accomplish the same result using standard JavaScript methods, read "string.replace() Method" on page 311.

See Also “Clib.strchr() Method” on page 191, “Clib.strcspn() Method” on page 192, and “string.replace() Method” on page 311

Clib.strrchr() Method

This method searches a string for the last occurrence of a character in a given string.

Syntax `Clib.strrchr(string, char)`

Parameter	Description
string	A string literal, or string variable, containing the character to be searched for
char	The character to search for

Returns The offset from the beginning of *string* of the first occurrence of *char* in *string*; otherwise, null, if *char* is not found in *string*.

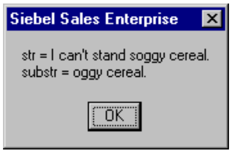
Usage This method searches the parameter *string* for the character *char*. The search is in the reverse direction, from the right, for *char* in *string*.

When possible, you should use the standard JavaScript method substring() (read “string.replace() Method” on page 311).

Example The following code fragment:

```
var str = "I can't stand soggy cereal."
var substr = Clib.strrchr(str, 'o');
TheApplication().RaiseErrorText("str = " + str + "\nsubstr = " +
substr);
```

results in the following output:



```
str = I can't stand soggy cereal.
substr = oggy cereal.
```

See Also "Clib.strchr() Method" on page 191, "Clib.strcspn() Method" on page 192, "Clib.strpbrk() Method" on page 200, and "string.replace() Method" on page 311

Clib.strspn() Method

This method searches a string for characters that are not among a group of specified characters.

Syntax `Clib.strspn(`*`string, charSet`*`)`

Parameter	Description
string	A literal string, or a variable containing a string, to be searched
charSet	A literal string, or a variable containing a string, which contains the set of characters to search for

Returns If all matching characters are found, the length of the string; otherwise, the offset of the first matching character from the beginning of *string*.

Usage This method searches the parameter string for any of the characters in the string *charSet*, and returns the offset of that character. The search is case-sensitive, so you may have to include both uppercase and lowercase instances of the characters to search for.

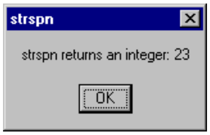
This method is similar to Clib.strpbrk(), except that Clib.strpbrk() returns the string beginning at the first character found, while Clib.strcspn() returns the offset number for that character.

When possible, you should use the standard JavaScript method substring() (read "string.replace() Method" on page 311).

Example The following fragment demonstrates Clib.strcspn(). When searching `string`, it returns the position of the `w`, counting from 0.

```
var string = "There is more than one way to skin a cat.";
var rStrspn = Clib.strspn(string, " aeiouTthrsmn");
TheApplication().RaiseErrorText("strspn returns an integer: ");
```

This results in the following output:



strspn returns an integer: 23

See Also "Clib.strchr() Method" on page 191, "Clib.strcspn() Method" on page 192, "Clib.strpbrk() Method" on page 200, and "string.replace() Method" on page 311

Clib.strstr() Method

This method searches a string for the first occurrence of a second string.

Syntax `Clib.strstr(sourceString, findString)`

Parameter	Description
sourceString	The string within which to search
findString	The string to find

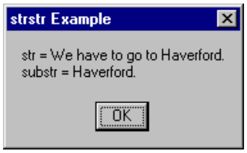
Returns The string beginning at the first occurrence of *findString* in *sourceString,* continuing to the end of *sourceString*; otherwise, null, if *findString* is not found.

Usage This method searches *sourceString*, from its beginning, for the first occurrence of *findString*. The search is case-sensitive. If the desired result can be accomplished with the standard JavaScript substring() method, that method is preferred.

Example The following code:

```
function Test1_Click ()
{
   var str = "We have to go to Haverford."
   var substr = Clib.strstr(str, 'H');
   TheApplication().RaiseErrorText("str = " + str + "\nsubstr = "
+substr);
}
```

results in the following output:



```
str = We have to go to Haverford.
substr = Haverford.
```

See Also "Clib.strstri() Method" on page 204 and "string.replace() Method" on page 311

Clib.strstri() Method

This method performs a case-insensitive search in a string for the first occurrence of a specified substring.

Syntax `Clib.strstri(`*`sourceString, findString`*`)`

Parameter	Description
sourceString	The string within which to search
findString	The string to find

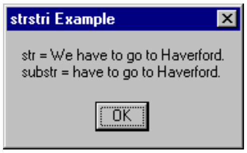
Returns The string beginning at the first occurrence of *findString* in *sourceString*, continuing to the end of *sourceString*; otherwise, null if *findString* is not found.

Usage This is a case-insensitive version of the substring() method. Compare the result with that shown in the "Clib.strstr() Method" on page 203.

Example The following code:

```
function Test_Click ()
{
   var str = "We have to go to Haverford."
   var substr = Clib.strstri(str, 'H');
   TheApplication().RaiseErrorText("str = " + str + "\nsubstr = "
+substr);
}
```

results in the following output:



```
str = We have to go to Haverford.
substr = have to go to Haverford.
```

See Also "Clib.strstr() Method" on page 203 and "string.replace() Method" on page 311

Clib.system() Method

This method passes a command to the command processor.

Syntax `Clib.system(commandString)`

Parameter	Description
commandString	A valid operating system command

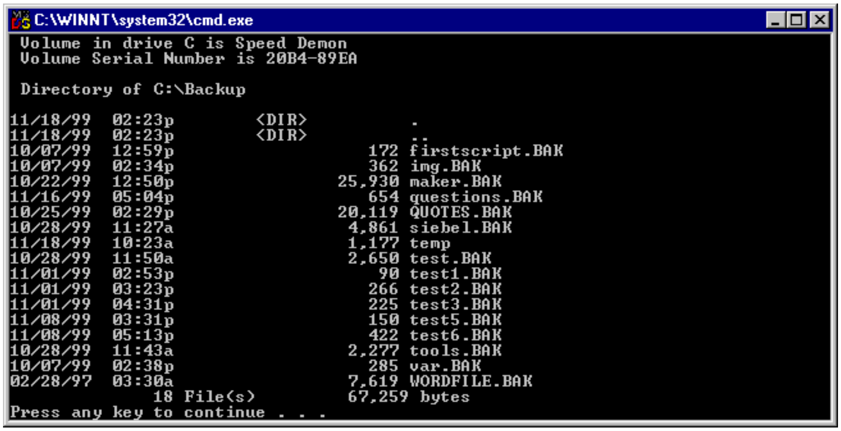
Returns The value returned by the command processor.

Usage This command passes a command to the operating system command processor and opens an operating system window in which it executes. Upon completion of the command, the window closes.

The *commandString* value may be a formatted string followed by variables according to the rules defined in Table 7 on page 138 in the section "Formatting Data."

Example The following code displays a directory in a DOS window, as shown:

```
Clib.system("dir /p C:\\Backup");
```



```
 Volume in drive C is Speed Demon
 Volume Serial Number is 20B4-89EA

 Directory of C:\Backup

11/18/99  02:23p        <DIR>          .
11/18/99  02:23p        <DIR>          ..
10/07/99  12:59p                   172 firstscript.BAK
10/07/99  02:34p                   362 img.BAK
10/22/99  12:50p                25,930 maker.BAK
11/16/99  05:04p                   654 questions.BAK
10/25/99  02:29p                20,119 QUOTES.BAK
10/28/99  11:27a                 4,861 siebel.BAK
11/18/99  10:23a                 1,177 temp
10/28/99  11:50a                 2,650 test.BAK
11/01/99  02:53p                    90 test1.BAK
11/01/99  03:23p                   266 test2.BAK
11/01/99  04:31p                   225 test3.BAK
11/08/99  03:31p                   150 test5.BAK
11/08/99  05:13p                   422 test6.BAK
10/28/99  11:43a                 2,277 tools.BAK
10/07/99  02:38p                   285 var.BAK
02/28/97  03:30a                 7,619 WORDFILE.BAK
              18 File(s)         67,259 bytes
Press any key to continue . . .
```

Clib.tanh() Method

This method calculates and returns the hyperbolic tangent of a floating-point number.

Syntax `Clib.tanh(`*`floatNum`*`)`

Parameter	Description
floatNum	A floating-point number, or a variable containing a floating-point number, whose hyperbolic tangent is to be found

Returns The hyperbolic tangent of *floatNum*.

See Also "Clib.cosh() Method" on page 144 and "Clib.sinh() Method" on page 188

Clib.time() Method

This method returns an integer representation of the current time.

Syntax `Clib.time([[var] timeInt])`

Parameter	Description
timeInt	A variable to hold the returned value, which must be declared if it has not already been declared

Returns An integer representation of the current time.

Usage The format of the time is not specifically defined except that it represents the current time, to the operating system's best approximation, and can be used in many other time-related functions. If *timeInt* is supplied, it is set to equal the returned value.

`Clib.time(timeInt)` and `timeInt = Clib.time()` assign the current local time to `timeInt`.

Example For examples, read "Clib.ctime() Method" on page 145, "Clib.difftime() Method" on page 146, "Clib.gmtime() Method" on page 168, "Clib.localtime() Method" on page 175, and "Clib.strftime() Method" on page 195.

See Also "GetDate() Method" on page 212, "Date.fromSystem() Static Method" on page 213, and "Date.toSystem() Method" on page 242

Clib.tmpfile() Method

This method creates a temporary binary file and returns its file pointer.

Syntax `Clib.tmpfile()`

Parameter	Description
Not applicable	

Returns The file pointer of the file created; null if the function fails.

Usage Clib.tmpfile() creates and opens a temporary binary file and returns its file pointer. The file pointer, and the temporary file, are automatically removed when the file is closed or when the program exits. The location of the temporary file depends on the implementation of Clib on the operating system in use.

Example For an example, read "Clib.fgets() Method" on page 152.

See Also "Clib.fopen() Method" on page 154

Clib.tmpnam() Method

This method creates a string that has a valid file name and is unique among existing files and among filenames returned by this function.

Syntax

```
Clib.tmpnam([str])
```

Parameter	Description
str	A variable to hold the name of a temporary file.

Returns String - a valid and unique filename

Usage This method creates a string that has a valid file name. This string is not the same as the name of any existing file, nor the same as any filename returned by this function during execution of this program. If *str* is supplied, it is set to the string returned by this function.

Clib.toascii() Method

This method translates a character into a seven-bit ASCII representation of the character.

Syntax

```
Clib.toascii(char)
```

Parameter	Description
char	A character literal, or a variable containing a character, to be translated

Returns A seven-bit ASCII representation of *char*.

Usage This method translates a character into a seven-bit ASCII representation of *char*. The translation is done by clearing every bit except for the lowest seven bits. If *char* is already a seven-bit ASCII character, it returns the character.

Example The following line of code returns the close-parenthesis character:

```
TheApplication().RaiseErrorText(Clib.toascii("©"));
```

See Also "Clib.isascii() Method" on page 170

Clib.ungetc()Method

This method pushes a character back into a file.

Syntax `Clib.ungetc(`*`char, filePointer`*`)`

Parameter	Description
char	The character to push back
filePointer	A file pointer as returned by Clb.fopen()

Returns The value of *char* if successful, EOF if unsuccessful.

Usage When *char* is put back, it is converted to a byte and is again in the file for subsequent retrieval. Only one character is pushed back. You might want to use this function to read up to, but not including, a newline character. You would then use Clib.ungetc to push the newline character back into the file buffer.

See Also "Clib.fgetc() Method and Clib.getc() Method" on page 150, "Clib.fputc() Method and Clib.putc() Method" on page 157, and "Clib.putenv() Method" on page 181

The Date Object

Siebel eScript provides two different systems for working with dates. One is the standard Date object of JavaScript; the other is part of the Clib object, which implements powerful routines from C. Two methods, Date.fromSystem() and Date.toSystem(), convert dates in the format of one system to the format of the other. The standard JavaScript Date object is described in this section.

CAUTION: To prevent Y2K problems, avoid using two-digit dates in your eScript code. Siebel eScript follows the ECMAScript standard for two-digit dates, which may be different from the conventions used by other programs, including Siebel applications.

A specific instance of a variable followed by a period should precede the method name to call a method. For example, if you had created the Date object aDate, the call to the .getDate() method would be `aDate.getDate()`. Static methods have "Date." at their beginnings because these methods are called with a literal call, such as Date.parse(). These methods are part of the Date object itself instead of instances of the Date object.

In the examples that follow, *dateVar* stands for the name of a variable that you create to hold a date value.

See Also "The Date Constructor" on page 210

"Universal Time Functions" on page 212

The Date Constructor

The Date constructor instantiates a new Date object.

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

Syntax A

```
var dateVar = new Date;
```

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

Syntax B

```
var dateVar = new Date(milliseconds);
```

Syntax C

```
var dateVar = new Date(dateString);
```

Syntax D

```
var dateVar = new Date(year, month, day);
```

Syntax E

```
var dateVar = new Date(year, month, day, hours, minutes, seconds);
```

Parameter	Description
milliseconds	The number of milliseconds since January 1, 1970.
dateString	A string representing a date and optional time.
year	A year. If the year is between 1950 and 2050, you may supply only the final two digits. Otherwise, four digits must be supplied. However, it's safest to always use four digits to minimize the risk of Y2K problems.
month	A month, specified as a number from 0 to 11. January is 0, and December is 11.
day	A day of the month, specified as a number from 1 to 31. The first day of a month is 1; the last is 28, 29, 30, or 31.
hours	An hour, specified as a number from 0 to 23. Midnight is 0; 11 PM is 23.
minutes	A minute, specified as a number from 0 to 59. The first minute of an hour is 0; the last is 59.
seconds	A second, specified as a number from 0 to 59. The first second of a minute is 0; the last is 59.

Returns If a parameter is specified, a Date object representing the date specified by the parameter.

Usage Syntax B returns a date and time represented by the number of milliseconds since midnight, January 1, 1970. This representation by 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. However, the recommended technique is to convert dates to milliseconds format before doing calculations.

Syntax C accepts a string representing a date and optional time. The format of such a string contains one or more of the following fields, in any order:

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

For example, the following string:

```
"October 13, 1995 13:13:15"
```

specifies the date, October 13, 1995, and the time, one thirteen and 15 seconds PM, which, expressed in 24-hour time, is 13:13 hours and 15 seconds. The time specification is optional; if it is included, the seconds specification is optional.

Syntax forms D and E are self-explanatory. Parameters passed to them are integers.

Example The following line of code:

```
var aDate = new Date(1776, 6, 4)
```

creates a Date object containing the date July 4, 1776.

Universal Time Functions

Universal Coordinated Time (abbreviated as UTC) is what used to be called Greenwich Mean Time (abbreviated GMT). It is also known as World Time and Universal Time. It is a time standard used everywhere in the world. UTC nominally reflects the mean solar time along the Earth's prime meridian (0 degrees longitude, which runs through the Greenwich Observatory outside of London). Siebel eScript includes a selection of Date functions that allow you to work with UTC values:

getUTCDay()
getUTCFullYear()
getUTCHours()
getUTCMilliseconds()
getUTCMinutes()
getUTCMonth()
getUTCSeconds()
setUTCDate()
setUTCFullYear()
setUTCHours()
setUTCMilliseconds
setUTCMinutes()
setUTCMonth()
Date.UTC()

GetDate() Method

This method returns the day of the month of a Date object.

Syntax *dateVar*.getDate()

Parameter	Description
Not applicable	

Returns The day of the month of *dateVar* as an integer from 1 to 31.

Usage This method returns the day of the month of the Date object specified by *dateVar,* as an integer from 1 to 31. The first day of a month is 1; the last is 28, 29, 30, or 31.

Example This example returns 14, the month part of the constructed Date object:

```
function Button2_Click ()
{
   var valentinesDay = new Date("2001", "1", "14");
   TheApplication().RaiseErrorText("Valentine’s Day is on day " +
      valentinesDay.getDate() + ".");
}
```

See Also "getDay() Method" on page 214, "getFullYear() Method" on page 215, "getHours() Method" on page 216, "getMinutes() Method" on page 217, "getMonth() Method" on page 218, "getSeconds() Method" on page 218, "getTime() Method" on page 219, "getYear() Method" on page 226, and "setDate() Method" on page 227

Date.fromSystem() Static Method

This method converts a time in the format returned by the Clib.time() method to a standard JavaScript Date object.

Syntax Date.fromSystem(*time*)

Parameter	Description
time	A variable holding a system date

Returns Not applicable

Usage Date.fromSystem() is a static method, invoked using the Date constructor rather than a variable.

Example To create a Date object from date information obtained using Clib, use code similar to:

```
var SysDate = Clib.time();
var ObjDate = Date.fromSystem(SysDate);
```

See Also "The Time Object" on page 135, "Clib.time() Method" on page 206, "The Date Constructor" on page 210, and "Date.toSystem() Method" on page 242

getDay() Method

This method returns the day of the week of a Date object.

Syntax *dateVar*.getDay()

Parameter	Description
Not applicable	

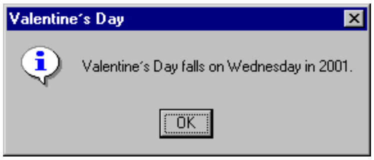
Returns The day of the week of *dateVar* as a number from 0 to 6.

Usage This method returns the day of the week of *dateVar*. Sunday is 0, and Saturday is 6. To get the name of the corresponding weekday, create an array holding the names of the days of the week and compare the return value to the array index, as shown in the following example.

Example This example gets the day of the week on which Valentine's Day occurs and displays the result in a message box, shown in the illustration.

```
function Button1_Click ()
{
   var weekDay = new Array("Sunday", "Monday", "Tuesday",
      "Wednesday", "Thursday", "Friday", "Saturday");
   var valentinesDay = new Date("2001", "1", "14");
   var theYear = valentinesDay.getFullYear()
   var i = 0;
   while (i < valentinesDay.getDay())
```

```
      {
         i++;
         var result = weekDay[i];
      }
      TheApplication().RaiseErrorText("Valentine’s Day falls on " +
result + " in " + theYear + ".");
}
```



Valentine’s Day falls on Wednesday in 2001.

See Also “GetDate() Method” on page 212, “getFullYear() Method” on page 215, “getHours() Method” on page 216, “getMinutes() Method” on page 217, “getMonth() Method” on page 218, “getSeconds() Method” on page 218, “getTime() Method” on page 219, and “getYear() Method” on page 226

getFullYear() Method

This method returns the year of a Date object as a number with four digits.

Syntax `dateVar.getFullYear()`

Parameter	Description
Not applicable	

Returns The year as a four-digit number, of the Date object specified by *dateVar.*

Example For examples, read “getDay() Method” on page 214, “setMilliseconds() Method” on page 229, and “setTime() Method” on page 232.

See Also “GetDate() Method” on page 212, “getDay() Method” on page 214, “getHours() Method” on page 216, “getMinutes() Method” on page 217, “getMonth() Method” on page 218, “getSeconds() Method” on page 218, “getTime() Method” on page 219, “getYear() Method” on page 226, and “setFullYear() Method” on page 228

getHours() Method

This method returns the hour of a Date object.

Syntax `dateVar.getHours()`

Parameter	Description
Not applicable	

Returns The hour portion of *dateVar*, as a number from 0 to 23.

Usage This method returns the hour portion of *dateVar* as a number from 0 to 23. Midnight is 0, and 11 PM is 23.

Example This code fragment returns the number 12, the hours portion of the specified time.

```
var aDate = new Date("October 31, 1986 12:13:14");
TheApplication().RaiseErrorText(aDate.getHours());
```

See Also "GetDate() Method" on page 212, "getDay() Method" on page 214, "getFullYear() Method" on page 215, "getMinutes() Method" on page 217, "getMonth() Method" on page 218, "getSeconds() Method" on page 218, "getTime() Method" on page 219, and "getYear() Method" on page 226

getMilliseconds() Method

This method returns the millisecond of a Date object.

Syntax `dateVar.getMilliseconds()`

Parameter	Description
Not applicable	

Returns The millisecond of *dateVar* as a number from 0 to 999.

Usage This method sets the millisecond of *dateVar* to *millisecond*. When given a date in millisecond form, this method returns the last three digits of the millisecond date; or, if negative, the result of the last three digits subtracted from 1000.

Example This code fragment displays the time on the system clock. The number of milliseconds past the beginning of the second appears at the end of the message.

```
var aDate = new Date;
   TheApplication().RaiseErrorText( aDate.toString() + " " +
      aDate.getMilliseconds() );
```

See Also "GetDate() Method" on page 212, "getDay() Method" on page 214, "getFullYear() Method" on page 215, "getHours() Method" on page 216, "getMinutes() Method" on page 217, "getMonth() Method" on page 218, "getSeconds() Method" on page 218, "getTime() Method" on page 219, and "getYear() Method" on page 226

getMinutes() Method

This method returns the minutes portion of a Date object.

Syntax *dateVar*.getMinutes()

Parameter	Description
Not applicable	

Returns The minutes portion of *dateVar* as a number from 0 to 59.

Usage This method returns the minutes portion of *dateVar* as a number from 0 to 59. The first minute of an hour is 0, and the last is 59.

Example This code fragment returns the number 13, the minutes portion of the specified time.

```
var aDate = new Date("October 31, 1986 12:13:14");
TheApplication().RaiseErrorText(aDate.getMinutes());
```

See Also “GetDate() Method” on page 212, “getDay() Method” on page 214, “getFullYear() Method” on page 215, “getHours() Method” on page 216, “getMonth() Method” on page 218, “getSeconds() Method” on page 218, “getTime() Method” on page 219, and “getYear() Method” on page 226

getMonth() Method

This method returns the month of a Date object.

Syntax

```
dateVar.getMonth()
```

Parameter	Description
Not applicable	

Returns The month portion of *dateVar* as a number from 0 to 11.

Usage This method returns the month, as a number from 0 to 11, of *dateVar*. January is 0, and December is 11.

Example This code fragment returns the number `10`, the result of adding 1 to the month portion of the specified date.

```
var aDate = new Date("October 31, 1986 12:13:14");
TheApplication().RaiseErrorText(aDate.getMonth() + 1);
```

See Also “GetDate() Method” on page 212, “getDay() Method” on page 214, “getFullYear() Method” on page 215, “getHours() Method” on page 216, “getMinutes() Method” on page 217, “getSeconds() Method” on page 218, “getTime() Method” on page 219, and “getYear() Method” on page 226

getSeconds() Method

This method returns the seconds portion of a Date object.

Syntax *dateVar*.getSeconds()

Parameter	Description
Not applicable	

Returns The seconds portion of *dateVar* as a number from 0 to 59.

Usage This method returns the seconds portion of *dateVar*. The first second of a minute is 0, and the last is 59.

Example This code fragment returns the number `14`, the seconds portion of the specified date.

```
var aDate = new Date("October 31, 1986 12:13:14");
TheApplication().RaiseErrorText(aDate.getSeconds() + 1);
```

See Also "GetDate() Method" on page 212, "getDay() Method" on page 214, "getFullYear() Method" on page 215, "getHours() Method" on page 216, "getMinutes() Method" on page 217, "getMonth() Method" on page 218, "getTime() Method" on page 219, and "getYear() Method" on page 226

getTime() Method

This method returns the milliseconds representation of a Date object, in the form of an integer representing the number of seconds between midnight on January 1, 1970, GMT, and the date and time specified by a Date object.

Syntax *dateVar*.getTime()

Parameter	Description
Not applicable	

Returns The milliseconds representation of *dateVar*.

Usage This method returns the milliseconds representation of a Date object, in the form of an integer representing the number of seconds between midnight on January 1, 1970, GMT, and the date and time specified by *dateVar*.

Example This code fragment returns the value `245594000.` To convert this value to something more readily interpreted, use the toLocaleString() method or the toGMTString() method.

```
var aDate = new Date("January 3, 1970 12:13:14");
TheApplication().RaiseErrorText(aDate.getTime());
```

See Also "Clib.asctime() Method" on page 141, "Clib.gmtime() Method" on page 168, "Clib.localtime() Method" on page 175, "Clib.mktime() Method" on page 179, "GetDate() Method" on page 212, "getDay() Method" on page 214, "getFullYear() Method" on page 215, "getHours() Method" on page 216, "getMinutes() Method" on page 217, "getMonth() Method" on page 218, "getSeconds() Method" on page 218, and "getYear() Method" on page 226

getTimezoneOffset() Method

This method returns the difference, in minutes, between Greenwich Mean Time and local time.

Syntax *dateVar*.getTimezoneOffset()

Parameter	Description
Not applicable	

Returns The difference, in minutes, between Greenwich Mean Time (GMT) and local time.

Example This example calculates the difference from Greenwich Mean Time in hours, of your location, based on the setting in the Windows Control Panel.

```
var aDate = new Date();
var hourDifference = Math.round(aDate.getTimezoneOffset() / 60);
TheApplication().RaiseErrorText("Your time zone is " +
      hourDifference + " hours from GMT.");
```

See Also “GetDate() Method” on page 212, “getDay() Method” on page 214, “getFullYear() Method” on page 215, “getHours() Method” on page 216, “getMinutes() Method” on page 217, “getMonth() Method” on page 218, “getSeconds() Method” on page 218, “getTime() Method” on page 219, and “getYear() Method” on page 226

getUTCDate() Method

This method returns the UTC day of the month of a Date object.

Syntax `dateVar.getUTCDate()`

Parameter	Description
Not applicable	

Returns The UTC day of the month of *dateVar*.

Usage This method returns the UTC day of the month of *dateVar* as a number from 1 to 31. The first day of a month is 1; the last is 28, 29, 30, or 31.

Example This code fragment displays 1, the hour portion of the date, followed by the GMT equivalent, which may be the same.

```
var aDate = new Date("May 1, 2001 13:24:35");
TheApplication().RaiseErrorText("Local day of the month is " +
   aDate.getHours() +"\nGMT day of the month is " +
   aDate.getUTCHours());
```

See Also “GetDate() Method” on page 212 and “setUTCDate() Method” on page 234

getUTCDay() Method

This method returns the UTC day of the week of a Date object.

Syntax *dateVar*.getUTCDay()

Parameter	Description
Not applicable	

Returns The UTC day of the week of *dateVar* as a number from 0 to 6.

Usage This method returns the UTC day of the week of *dateVar* as a number from 0 to 6. Sunday is 0, and Saturday is 6.

Example This function displays the day of the week of May 1, 2001, both locally and in universal time.

```
function Button2_Click ()
{
   var localDay;
   var UTCDay;
   var MayDay = new Date("May 1, 2001 13:30:35");
   var weekDay = new Array("Sunday", "Monday", "Tuesday",
      "Wednesday", "Thursday", "Friday", "Saturday");

   for (var i = 0; i <= MayDay.getDay();i++)
      localDay = weekDay[i];
   var msgtext = "May 1, 2001, 1:30 PM falls on " + localDay;

   for  (var j = 0; j <= MayDay.getUTCDay(); j++)
      UTCDay = weekDay[j];
   msgtext = msgtext + " locally, \nand on " + UTCDay + " GMT.";

   TheApplication().RaiseErrorText(msgtext);
}
```

See Also "getDay() Method" on page 214

getUTCFullYear() Method

This method returns the UTC year of a Date object.

Syntax *dateVar*.getUTCFullYear()

Parameter	Description
Not applicable	

Returns The UTC year of *dateVar* as a four-digit number.

Example This code fragment displays 2001, the year portion of the date, followed by the GMT equivalent, which may be the same.

```
var aDate = new Date("January 1, 2001 13:24:35");
TheApplication().RaiseErrorText("Local year is " +
aDate.getYear() +
   "\nGMT year is " + aDate.getUTCFullYear());
```

See Also "getFullYear() Method", "setFullYear() Method" on page 228, and "setUTCFullYear() Method" on page 234

getUTCHours() Method

This method returns the UTC hour of a Date object.

Syntax *dateVar*.getUTCHours()

Parameter	Description
Not applicable	

Returns The UTC hour of *dateVar* as a number from 0 to 23.

Usage This method returns the UTC hour of *dateVar* as a number from 0 through 23. Midnight is 0, and 11 PM is 23.

Example This code fragment displays 13, the hour portion of the date, followed by the GMT equivalent.

```
var aDate = new Date("May 1, 2001 13:24:35");
TheApplication().RaiseErrorText("Local hour is “ +
aDate.getHours() +
   "\nGMT hour is " + aDate.getUTCHours());
```

See Also “getHours() Method” on page 216 and “setUTCHours() Method” on page 235

getUTCMilliseconds() Method

This method returns the UTC millisecond of a Date object.

Syntax *dateVar*.getUTCMilliseconds()

Parameter	Description
Not applicable	

Returns The UTC millisecond of *dateVar* as a number from 0 to 999.

Usage This method returns the UTC millisecond of *dateVar* as a number from 0 through 999. The first millisecond in a second is 0; the last is 999.

See Also “getMilliseconds() Method” on page 216 and “setUTCMilliseconds() Method” on page 236

getUTCMinutes() Method

This method returns the UTC minute of a Date object.

Syntax *dateVar*.getUTCMinutes()

Parameter	Description
Not applicable	

Returns The UTC minute of *dateVar* as a number from 0 to 59.

Usage This method returns the UTC minute of *dateVar* as a number from 0 through 59. The first minute of an hour is 0; the last is 59.

Example This code fragment displays 24, the minutes portion of the date, followed by the GMT equivalent, which is probably the same.

```
var aDate = new Date("May 1, 2001 13:24:35");
TheApplication().RaiseErrorText("Local minutes: " +
aDate.getMinutes() +
   "\nGMT minutes: " + aDate.getUTCMinutes());
```

See Also "getMinutes() Method" on page 217 and "setUTCMinutes() Method" on page 237

getUTCMonth() Method

This method returns the UTC month of a Date object.

Syntax `dateVar.getUTCMonth()`

Parameter	Description
Not applicable	

Returns The UTC month of *dateVar* as a number from 0 to 11.

Usage This method returns the UTC month of *dateVar* as a number from 0 through 11. January is 0, and December is 11.

Example This code fragment displays 5, the month portion of the date (determined by adding 1 to the value returned by getMonth), followed by the GMT equivalent (determined by adding 1 to the value returned by getUTCMonth), which is probably the same.

```
var aDate = new Date("May 1, 2001 13:24:35");
var locMo = aDate.getMonth() + 1;
var GMTMo = aDate.getUTCMonth() + 1
TheApplication().RaiseErrorText("Local month: " + locMo +"\nGMT
month: "
   + GMTMo);
```

See Also "getMonth() Method" on page 218 and "setUTCMonth() Method" on page 238

getUTCSeconds() Method

This method returns the UTC second of a Date object.

Syntax

```
dateVar.getUTCSeconds()
```

Parameter	Description
Not applicable	

Returns The UTC second of *dateVar* as number from 0 to 59.

Usage This method returns the UTC second of *dateVar* as a number from 0 through 59. The first second of a minute is 0, and the last is 59.

See Also "getSeconds() Method" on page 218 and "setUTCSeconds() Method" on page 239

getYear() Method

This method returns the year portion of a Date object.

Syntax

```
dateVar.getYear()
```

Parameter	Description
Not applicable	

Returns The year of the *dateVar* as a three-digit number.

Usage This method returns the year portion of *dateVar* as a three-digit number.

See Also "getFullYear() Method" on page 215, "getUTCFullYear() Method" on page 222, and "setYear() Method" on page 239

Date.parse() Static Method

This method converts a date string to a Date object.

Syntax

```
Date.parse(dateString)
```

Parameter	Description
dateString	A string of the form *ddd, Month dd, yyyy hh:mm:ss*

Returns A Date object representing the date in *dateString*.

Usage Date.parse() is a static method, invoked using the Date constructor rather than a variable. The string must be in the following format:

```
Friday, October 31, 1998 15:30:00 -0800
```

where the last number is the offset from Greenwich Mean Time. This format is used by the *dateVar*.toGMTString() method and by email and Internet applications. The day of the week, time zone, time specification, or seconds field may be omitted. The statement:

```
var aDate = Date.parse(dateString);
```

is equivalent to:

```
var aDate = new Date(dateString);
```

Example The following code fragment yields the result `9098766000`:

```
var aDate = Date.parse("Friday, October 31, 1998 15:30:00 -0220");
TheApplication().RaiseErrorText(aDate);
```

See Also “The Date Constructor” on page 210

setDate() Method

This method sets the day of a Date object to a specified day of the month.

Syntax

```
dateVar.setDate(dayOfMonth)
```

Parameter	Description
dayOfMonth	The day of the month to which to set *dateVar* as an integer from 1 through 31

Returns Not applicable

Usage This method sets the day of *dateVar* to *dayOfMonth* as a number from 1 to 31. The first day of a month is 1; the last is 28, 29, 30, or 31.

See Also "GetDate() Method" on page 212 and "setUTCDate() Method" on page 234

setFullYear() Method

This method sets the year of a Date object to a specified four-digit year.

Syntax

```
dateVar.setFullYear(year[, month[, date]])
```

Parameter	Description
year	The year to which to set *dateVar* as a four-digit integer
month	The month to which to set *year* as an integer from 0 to 11
date	The date of *month* to which to set *dateVar* as an integer from 1 to 31

Returns Not applicable

Usage This method sets the year of *dateVar* to *year*. Optionally, it can set the month of *year* to *month*, and the date of *month* to *date*. The year must be expressed in four digits.

See Also "getFullYear() Method" on page 215, "setDate() Method" on page 227, "setMonth() Method" on page 231, "setUTCFullYear() Method" on page 234, and "setYear() Method" on page 239

setHours() Method

This method sets the hour of a Date object to a specific hour of a 24-hour clock.

Syntax `dateVar.setHours(hour[, minute[, second[, millisecond]]])`

Parameter	Description
hour	The hour to which to set *dateVar* as an integer from 0 through 23
minute	The minute of *hour* to which to set *dateVar* as an integer from 0 through 59
second	The second of *minute* to which to set *dateVar* as an integer from 0 through 59
millisecond	The millisecond of *second* to which to set *dateVar* as an integer from 0 through 999

Returns Not applicable

Usage This method sets the hour of *dateVar* to *hour,* expressed as a number from 0 to 23. It can optionally also set the UTC minute, second, and millisecond. Midnight is expressed as 0, and 11 PM as 23.

See Also "getHours() Method" on page 216, "setMilliseconds() Method" on page 229, "setMinutes() Method" on page 231, "setSeconds() Method" on page 232, and "setUTCHours() Method" on page 235

setMilliseconds() Method

This method sets the millisecond of a Date object to a date expressed in milliseconds relative to the system time.

Syntax `dateVar.setMilliseconds(millisecond)`

Parameter	Description
millisecond	The millisecond to which *dateVar* should be set as a positive or negative integer

Returns Not applicable

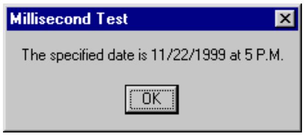
Usage This method sets the millisecond of *dateVar* to *millisecond*. The value of *dateVar* becomes equivalent to the number of milliseconds from the time on the system clock. Use a positive number for later times, a negative number for earlier times.

Example This example accepts a number of milliseconds as input and converts it to the date relative to the date and time on the system clock. The illustration shows the result of entering 0 on November 22, 1999.

```
function test2_Click ()
{
   var aDate = new Date;
   var milli = 20000000;
   aDate.setMilliseconds(milli);
   var aYear = aDate.getFullYear();
   var aMonth = aDate.getMonth() + 1;
   var aDay = aDate.getDate();
   var anHour = aDate.getHours();

   switch(anHour)
   {
      case 0:
         anHour = " 12 midnight.";
         break;
      case 12:
         anHour = " 12 noon.";
         break;
      default:
         if (anHour > 11 )
            anHour = (anHour - 12 ) + " P.M.";
         else
            anHour = anHour + " A.M.";
   }

   TheApplication().RaiseErrorText("The specified date is " +
aMonth + "/" + aDay + "/" + aYear + " at " + anHour);
}
```



The specified date is 11/22/1999 at 5 P.M.

See Also “getMilliseconds() Method” on page 216, “setTime() Method” on page 232, and “setUTCMilliseconds() Method” on page 236

setMinutes() Method

This method sets the minute of a Date object to a specified minute.

Syntax

```
dateVar.setMinutes(minute[, second[, millisecond]])
```

Parameter	Description
minute	The minute to which to set *dateVar* as an integer from 0 through 59
second	The second to which to set *minute* as an integer from 0 through 59
millisecond	The millisecond to which to set *second* as an integer from 0 through 999

Returns Not applicable

Usage This method sets the minute of *dateVar* to *minute* and optionally sets *minute* to a specific *second* and *millisecond*. The first minute of an hour is 0, and the last is 59.

See Also “getMinutes() Method” on page 217, “setMilliseconds() Method” on page 229, “setSeconds() Method” on page 232, and “setUTCMinutes() Method” on page 237

setMonth() Method

This method sets the month of a Date object to a specific month.

Syntax

```
dateVar.setMonth(month[, date])
```

Parameter	Description
month	The month to which to set *dateVar* as an integer from 0 to 11
date	The date of *month* to which to set *dateVar* as an integer from 1 to 31

Returns Not applicable

Usage This method sets the month of *dateVar* to *month* as a number from 0 to 11 and optionally sets the day of *month* to *date*. January is represented by 0, and December by 11.

See Also "getMonth() Method" on page 218, "setDate() Method" on page 227, and "setUTCMonth() Method" on page 238

setSeconds() Method

This method sets the second in a Date object.

Syntax

```
dateVar.setSeconds(second[, millisecond])
```

Parameter	Description
second	The minute to which to set *dateVar* as an integer from 0 through 59
millisecond	The millisecond to which to set *second* as an integer from 0 through 999

Returns Not applicable

Usage This method sets the second of *dateVar* to *second* and optionally sets *second* to a specific *millisecond*. The first second of a minute is 0, and the last is 59.

See Also "getSeconds() Method" on page 218, "setMilliseconds() Method" on page 229, and "setUTCSeconds() Method" on page 239

setTime() Method

This method sets a Date object to a date and time specified by the number of milliseconds before or after January 1, 1970.

Syntax

```
dateVar.setTime(milliseconds)
```

Parameter	Description
milliseconds	The number of milliseconds from midnight on January 1, 1970, GMT

Returns Not applicable

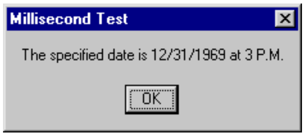
Usage This method sets *dateVar* to a date that is *milliseconds* milliseconds from January 1, 1970, GMT. To set a date earlier than that date, use a negative number.

Example This example accepts a number of milliseconds as input and converts it to a date and hour. To get the result shown in the illustration, a value of -345650 was entered.

```
function dateBtn_Click ()
{
   var aDate = new Date;
   var milli = -4000;
   aDate.setTime(milli);
   var aYear = aDate.getFullYear();
   var aMonth = aDate.getMonth() + 1;
   var aDay = aDate.getDate();
   var anHour = aDate.getHours();

   switch(anHour)
   {
      case 0:
         anHour = " 12 midnight.";
         break;
      case 12:
         anHour = " 12 noon.";
         break;
      default:
         if ( anHour > 11 )
            anHour = (anHour - 12) + " P.M.";
         else
            anHour = anHour + " A.M.";
   }

   TheApplication().RaiseErrorText("The specified date is " +
      aMonth + "/" + aDay + "/" + aYear + " at " + anHour);
}
```



The specified date is 12/31/1969 at 3 P.M.

See Also “getTime() Method” on page 219

setUTCDate() Method

This method sets the UTC day of a Date object to the specified day of a UTC month.

Syntax

```
dateVar.setUTCDate(dayOfMonth)
```

Parameter	Description
dayOfMonth	The day of the UTC month to which to set *dateVar* as an integer from 1 through 31

Returns Not applicable

Usage This method sets the UTC day of *dateVar* to *dayOfMonth* as a number from 1 to 31. The first day of a month is 1; the last is 28, 29, 30, or 31.

See Also "Universal Time Functions" on page 212, "getUTCDate() Method" on page 221, and "setDate() Method" on page 227

setUTCFullYear() Method

This method sets the UTC year of a Date object to a specified four-digit year.

Syntax

```
dateVar.setUTCFullYear(year[, month[, date]])
```

Parameter	Description
year	The UTC year to which to set *dateVar* as a four-digit integer
month	The UTC month to which to set *year* as an integer from 0 to 11
date	The UTC date of *month* to which to set *dateVar* as an integer from 1 to 31

Returns Not applicable

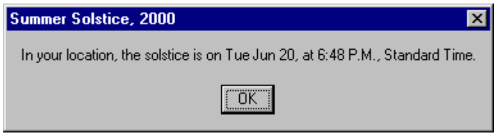
Usage This method sets the UTC year of *dateVar* to *year*. Optionally, it can set the UTC month of *year* to *month*, and the UTC date of *month* to *date*. The year must be expressed in four digits.

Example The following example uses the setUTCFullYear method to assign the date of the 2000 summer solstice and the setUTCHours method to assign its time to a Date object. Then it determines the local date and displays it as shown in the illustration following the example.

```
function dateBtn_Click ()
{
   var Mstring = " A.M., Standard Time.";
   var solstice2K = new Date;
   solstice2K.setUTCFullYear(2000, 5, 21);
   solstice2K.setUTCHours(01, 48);
   var localDate = solstice2K.toLocaleString();
   var pos = localDate.indexOf("2000")
   var localDay = localDate.substring(0, pos - 10);

   var localHr = solstice2K.getHours();
   if (localHr > 11 )
   {
      localHr = (localHr - 12 );
      Mstring =  " P.M., Standard Time.";
   }
   var localMin = solstice2K.getMinutes();

   var msg = "In your location, the solstice is on " + localDay +
      ", at " + localHr + ":" + localMin + Mstring;
   TheApplication().RaiseErrorText(msg);
}
```



Summer Solstice, 2000

In your location, the solstice is on Tue Jun 20, at 6:48 P.M., Standard Time.

See Also “Universal Time Functions” on page 212, “getUTCFullYear() Method” on page 222, “setFullYear() Method” on page 228, and “setYear() Method” on page 239

setUTCHours() Method

This method sets the UTC hour of a Date object to a specific hour of a 24-hour clock.

Syntax

```
dateVar.setUTCHours(hour[, minute[, second[, millisecond]]])
```

Parameter	Description
hour	The UTC hour to which to set *dateVar* as an integer from 0 through 23
minute	The UTC minute of *hour* to which to set *dateVar* as an integer from 0 through 59
second	The UTC second of *minute* to which to set *dateVar* as an integer from 0 through 59
millisecond	The UTC millisecond of *second* to which to set *dateVar* as an integer from 0 through 999

Returns Not applicable

Usage This method sets the UTC hour of *dateVar* to *hour* as a number from 0 to 23. Midnight is expressed as 0, and 11 PM as 23. It can optionally also set the UTC minute, second, and millisecond.

Example For an example, read "setUTCFullYear() Method" on page 234.

See Also "Universal Time Functions" on page 212, "getUTCHours() Method" on page 223, and "setHours() Method" on page 229

setUTCMilliseconds() Method

This method sets the UTC millisecond of a Date object to a date expressed in milliseconds relative to the UTC equivalent of the system time.

Syntax

```
dateVar.setUTCMilliseconds(millisecond)
```

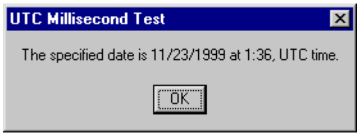
Parameter	Description
millisecond	The UTC millisecond to which *dateVar* should be set as a positive or negative integer

Returns Not applicable

Usage This method sets the UTC millisecond of *dateVar* to *millisecond*. The value of *dateVar* becomes equivalent to the number of milliseconds from the UTC equivalent of time on the system clock. Use a positive number for later times, and a negative number for earlier times.

Example The following example gets a number of milliseconds as input and converts it to a UTC date and time. When run at 5:36 p.m., Pacific Time, on November 22, 1999, it produced the result shown in the illustration.

```
function dateBtn_Click ()
{
   var aDate = new Date;
   var milli = 20000;
   aDate.setUTCMilliseconds(milli);
   var aYear = aDate.getUTCFullYear();
   var aMonth = aDate.getMonth() + 1;
   var aDay = aDate.getUTCDate();
   var anHour = aDate.getUTCHours();
   var aMinute = aDate.getUTCMinutes();
   TheApplication().RaiseErrorText("The specified date is " +
            aMonth +
      "/" + aDay + "/" + aYear + " at " + anHour + ":" +
      aMinute + ", UTC time.");
}
```



The specified date is 11/23/1999 at 1:36, UTC time.

See Also “Universal Time Functions” on page 212, “getUTCMilliseconds() Method” on page 224, and “setMilliseconds() Method” on page 229

setUTCMinutes() Method

This method sets the UTC minute of a Date object to a specified minute.

Syntax

```
dateVar.setUTCMinutes(minute[, second[, millisecond]])
```

Parameter	Description
minute	The UTC minute to which to set *dateVar* as an integer from 0 through 59
second	The UTC second to which to set *minute* as an integer from 0 through 59
millisecond	The UTC millisecond to which to set *second* as an integer from 0 through 999

Returns Not applicable

Usage This method sets the UTC minute of *dateVar* to *minute* and optionally sets *minute* to a specific UTC *second* and UTC *millisecond*. The first minute of an hour is 0, and the last is 59.

See Also "Universal Time Functions" on page 212, "getUTCMinutes() Method" on page 224, and "setMinutes() Method" on page 231

setUTCMonth() Method

This method sets the UTC month of a Date object to a specific month.

Syntax

```
dateVar.setUTCMonth(month[, date])
```

Parameter	Description
month	The UTC month to which to set *dateVar* as an integer from 0 to 11
date	The UTC date of *month* to which to set *dateVar* as an integer from 1 to 31

Returns Not applicable

Usage This method sets the UTC month of *dateVar* to *month* as a number from 0 to 11 and optionally sets the UTC day of *month* to *date*. January is represented by 0, and December by 11.

See Also “Universal Time Functions” on page 212, “getUTCMonth() Method” on page 225, and “setMonth() Method” on page 231

setUTCSeconds() Method

This method sets the UTC second of the minute of a Date object to a specified second and optionally sets the millisecond within the second.

Syntax

```
dateVar.setUTCSeconds(second[, millisecond])
```

Parameter	Description
second	The UTC minute to which to set *dateVar* as an integer from 0 through 59
millisecond	The UTC millisecond to which to set *second* as an integer from 0 through 999

Returns Not applicable

Usage This method sets the UTC second of *dateVar* to *second* and optionally sets *second* to a specific UTC *millisecond*. The first second of a minute is 0, and the last is 59. The first millisecond is 0, and the last is 999.

See Also “Universal Time Functions” on page 212, “getUTCSeconds() Method” on page 226, and “setSeconds() Method” on page 232

setYear() Method

This method sets the year of a Date object as a specified two-digit or four-digit year.

Syntax

```
dateVar.setYear(year)
```

Parameter	Description
year	The year to which to set *dateVar* as a two-digit integer for twentieth-century years, otherwise as a four-digit integer

Returns Not applicable

Usage 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.

See Also "getFullYear() Method" on page 215, "getYear() Method" on page 226, "setFullYear() Method" on page 228, and "setUTCFullYear() Method" on page 234

toGMTString() Method

This method converts a Date object to a string, based on Greenwich Mean Time.

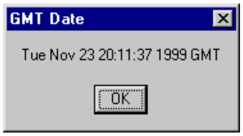
Syntax *dateVar*.toGMTString()

Parameter	Description
Not applicable	

Returns The date to which *dateVar* is set as a string of the form *Day Mon dd hh:mm:ss yyyy* GMT.

Example This example accepts a number of milliseconds as input and converts it to the GMT time represented by the number of milliseconds before or after the time on the system clock.

```
function clickme_Click ()
{
   var aDate = new Date;
   var milli = 200000;
   aDate.setUTCMilliseconds(milli);
   TheApplication().RaiseErrorText(aDate.toGMTString());
}
```



Tue Nov 23 20:11:37 1999 GMT

See Also “Clib.asctime() Method” on page 141, “toLocaleString() Method and toString() Method” on page 241, and “toUTCString() Method” on page 242

toLocaleString() Method and toString() Method

These methods return a string representing the date and time of a Date object based on the time zone of the user.

Syntax

```
dateVar.toLocaleString()
dateVar.toString()
```

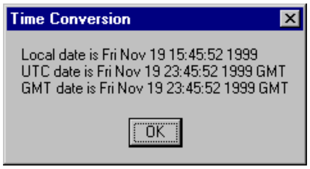
Parameter	Description
Not applicable	

Returns A string representing the date and time of *dateVar* based on the time zone of the user, in the form *Day Mon dd hh:mm:ss yyyy.*

Usage These methods return a string representing the date and time of a Date object based on the local time zone of the user.

Example This example displays the local time from your computer’s clock, the UTC time, and the Greenwich Mean Time. The result appears in the message box that follows the code.

```
var aDate = new Date();
var local = aDate.toLocaleString();
var universal = aDate.toUTCString();
var greenwich = aDate.toGMTString();
TheApplication().RaiseErrorText("Local date is " + local +
   "\nUTC date is " + universal +
   "\nGMT date is " + greenwich);
```



```
Local date is Fri Nov 19 15:45:52 1999
UTC date is Fri Nov 19 23:45:52 1999 GMT
GMT date is Fri Nov 19 23:45:52 1999 GMT
```

See Also “Clib.asctime() Method” on page 141, “Clib.gmtime() Method” on page 168, “Clib.localtime() Method” on page 175, “toGMTString() Method” on page 240, and “toUTCString() Method” on page 242

Date.toSystem() Method

This method converts a Date object to a system time format that is the same as that returned by the Clib.time() method.

Syntax

```
Date.toSystem()
```

Parameter	Description
Not applicable	

Returns A date value in the time format returned by the Clib.time() method.

Usage To create a Date object from a variable in system time format, read “Date.fromSystem() Static Method” on page 213.

Example To convert a Date object to a system format that can be used by the methods of the Clib object, use code similar to:

```
var SysDate = objDate.toSystem();
```

See Also “Date.fromSystem() Static Method” on page 213

toUTCString() Method

This method returns a string that represents the UTC date in a convenient and human-readable form.

Syntax

```
dateVar.toUTCString()
```

Parameter	Description
Not applicable	

Returns A string that represents the UTC date of *dateVar.*

Usage This method returns a string that represents the UTC date in a convenient and human-readable form. The string takes the form *Day Mon dd hh:mm:ss yyyy.*

Example For an example, read "toLocaleString() Method and toString() Method" on page 241.

See Also "Clib.asctime() Method" on page 141, "toGMTString() Method" on page 240, and "toLocaleString() Method and toString() Method" on page 241

Date.UTC() Static Method

This method interprets its parameters as a date and returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified.

Syntax

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

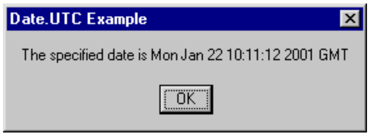
Parameter	Description
year	An integer representing the year (two digits may be used to represent years in the twentieth century; however, use four digits to avoid Y2K problems)
month	An integer from 0 through 11 representing the month
day	An integer from 1 through 31 representing the day of the month
hours	An integer from 0 through 23 representing the hour on a 24-hour clock
minutes	An integer from 0 through 59 representing the minute of *hours*
seconds	An integer from 0 through 59 representing the second of *minutes*

Returns An integer representing the number of milliseconds before or after midnight January 1, 1970, of the specified date and time.

Usage Date.UTC is a static method, invoked using the Date constructor rather than a variable. The parameters are interpreted as referring to Greenwich Mean Time (GMT).

Example This example shows the proper construction of a Date.UTC declaration and demonstrates that the function behaves as specified.

```
function clickme_Click ()
{
   var aDate = new Date(Date.UTC(2001, 1, 22, 10, 11, 12));
   TheApplication().RaiseErrorText("The specified date is " +
      aDate.toUTCString());
}
```


Date.UTC Example

The specified date is Mon Jan 22 10:11:12 2001 GMT

See Also "The Date Constructor" on page 210

The Exception Object

The Exception object contains exceptions being thrown in the case of a failed operation.

Properties

errCode (This property contains the error number.)

errText (This property contains a textual description of the error.)

Methods

"toLocaleString() Method and toString() Method" on page 241

Here is an example of the Exception object:

```
try
}
   var oBO = TheApplication().GetService(“Incorrect name”);
}
catch (e)
}
   var sText = e.errText;
   var nCode = e.errCode;
}
```

Function Objects

A Function object holds the definition of a function defined in eScript. Note that in eScript, procedures are functions.

Syntax A

```
function funcName( [arg1 [, ..., argn]] )
{
     body
}
```

Syntax B

```
var funcName = new Function([arg1 [, ..., argn,]] body );
```

Parameter	Description
funcName	The name of the function to be created
arg1 [, ..., argn]	An optional list of arguments that the function accepts
body	The lines of code that the function executes

Returns Whatever its code is set up to return. For more information, read "return Statement" on page 247.

Usage Syntax A is the standard method for defining a function. Syntax B is an alternative way to create a function and is used to create Function objects explicitly.

Note the difference in case of the keyword Function between Syntax A and Syntax B. Function objects created with Syntax B (that is, the Function constructor) are evaluated each time they are used. This is less efficient than Syntax A—declaring a function and calling it within your code—because declared functions are compiled instead of interpreted.

Example The following fragment of code illustrates creating a function AddTwoNumbers using a declaration:

```
function AddTwoNumbers (a, b)
{
   return (a + b);
}
```

The following fragment illustrates creating the same function using the Function constructor:

```
AddTwoNumbers = new Function ("a", "b", "return (a + b)");
```

The difference between the two is that when AddTwoNumbers is created using a declaration, AddTwoNumbers is the name of a function, whereas when AddTwoNumbers is created using the Function constructor, AddTwoNumbers is the name of a variable whose current value is a reference to the function created using the Function constructor.

length Property

The length property returns the number of arguments expected by the function.

Syntax `funcName.length`

Parameter	Description
funcName	The function whose length property is to be found

Returns The number of arguments expected by *funcName*.

return Statement

The return statement passes a value back to the function that called it.

Syntax `return value`

Parameter	Description
value	The result produced by the function

Returns Not applicable

Usage 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.

Example This function returns a value equal to the number passed to it multiplied by 2 and divided by 5.

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

Here is an example of a script using the preceding function. This script calculates the mathematical expression `n = (10 * 2) / 5 + (20 * 2) / 5.` It then displays the value for `n`, which is 12.

```
function myFunction()
{
   var a = DoubleAndDivideBy5(10);
   var b = DoubleAndDivideBy5(20);
   TheApplication().RaiseErrorText(a + b);
}
```

The Global Object

Global variables are members of the global object. To access global properties, you do not need to use an object name. For example, to access the isNaN() method, which tests to see whether a value is equal to the special value NaN, you can use either of the following syntax forms.

Syntax A

```
globalMethod(value);
```

Syntax B

```
global.globalMethod(value);
```

Placeholder	Description
globalMethod	The method to be applied
value	The value to which the method is to be applied

Usage Syntax A treats *globalMethod* as a function; Syntax B treats it as a method of the global object. You may not use Syntax A 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.

See Also "Global Functions Unique to Siebel eScript" on page 249

"Conversion or Casting Functions" on page 250

Global Functions Unique to Siebel eScript

The global functions described in this section are unique to the Siebel eScript implementation of JavaScript. In other words, they are not part of the ECMAScript standard, but they are useful. Avoid using these functions in a script that may be used with a JavaScript interpreter that does not support these unique functions.

Like other global items, the following functions are actually methods of the global object and can be called with either function or method syntax:

- "COMCreateObject() Method" on page 251
- "CORBACreateObject() Method" on page 252

- "getArrayLength() Method" on page 258
- "setArrayLength() Method" on page 261
- "undefine() Method" on page 271

Conversion or Casting Functions

Though Siebel eScript 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 has one parameter, which is a variable or data item, 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. Because aString had already been created with the value `"123"`, the second line could also have been:

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

Use the following eScript methods when casting or converting between data types:

- "ToBoolean() Method" on page 262
- "ToBuffer() Method" on page 263
- "ToBytes() Method" on page 264
- "ToInt32() Method" on page 265
- "ToInteger() Method" on page 266
- "ToNumber() Method" on page 267
- "ToObject() Method" on page 268
- "ToString() Method" on page 268
- "ToString() Method" on page 268
- "ToUint16() Method" on page 269
- "ToUint32() Method" on page 270

COMCreateObject() Method

COMCreateObject instantiates a COM object.

Syntax

```
COMCreateObject(objectName)
```

Parameter	Description
objectName	The name of the object to be created

Returns A COM object if successful; otherwise, undefined.

Usage You should be able to pass any type of variable to the COM object being called; however, you must ascertain that the variable is of a valid type for the COM object. Valid types are strings, numbers, and object pointers.

NOTE: DLLs instantiated by this method must be Thread-Safe.

Example This example instantiates Microsoft Excel as a COM object and makes it visible:

```
var ExcelApp = COMCreateObject("Excel.Application");
var bb = ExcelApp.visible = true;
//Make Excel visible through the Application object.

// Place some text in the first cell of the sheet
ExcelApp.ActiveSheet.Cells(1,1).Value = "Column A, Row 1";

// Save the sheet
var fileName = "C:\\demo.xls";
ExcelApp.SaveAs (fileName);

// Close Excel with the Quit method on the Application object
ExcelApp.Application.Quit();

// Clear the object from memory
ExcelApp = null;
}
```

See Also “CORBACreateObject() Method” on page 252

CORBACreateObject() Method

CORBACreateObject binds a specified CORBA object and returns its object handle.

Syntax `CORBACreateObject(instanceName[, objectName][, serverName])`

Parameter	Description
instanceName	The name of the interface as declared in the IDL file
objectName	The name given to the CORBA object
serverName	The fully qualified IP address of the server to connect to

Returns The object handle of the CORBA object.

Usage Only the *instanceName* parameter is required. The *serverName* parameter may be specified either as an IP address in *nnn.nnn.nnn.nnn* form or as a fully qualified network name for the host computer. Valid types are strings, numbers, and object pointers.

NOTE: Objects instantiated with CORBACreateObject do not support methods with out or in/out parameters.

The optional parameters, which are valid only with the Visibroker ORB, provide greater specificity regarding the object to connect to. Thus, for example:

```
var cObj = CORBACreateObject("Account")
```

connects to the first account object found. Alternatively:

```
var cObj = CORBACreateObject("Account", "Bus_Server")
```

connects to the first object it finds named Bus_Server that contains an account object. If no object named Bus_Server is found, the method fails.

```
var cObj = CORBACreateObject("Account","", 111.17.2.18)
```

looks for an account object on the server with the IP address `111.17.2.18`. If that server does not contain an account object, the method fails.

If you are using the Visibroker ORB, you must have IREP (an interface repository utility) running. IREP is part of the Visibroker ORB and must be running with the necessary IDL files loaded to allow access to a particular CORBA interface.

If you are using the Orbix ORB, objects that are to be accessed must be registered with the Orbix Naming Service. This requires that the appropriate server changes be made such that the Server carries out the correct registration process.

Objects may be registered in the Naming Service within certain contexts. For example, if you want to register one grid object in the grid.exe executable and this grid object has the human readable name gridObject1 in the server, then it is possible to register this object in the Naming Service under the context of

```
\Siebel Objects\Grids\gridObject1
```

To resolve the object denoted by the name gridObject1, navigate through the appropriate context hierarchy to get the actual Object. Following this convention, use the following to call CORBACreateObject to obtain a reference to the preceding object:

```
var p_corb_gridObj = CORBACreateObject(“Siebel
Objects:Grids:gridObject1”)
```

Separate each of the Naming Context nodes with a colon (:). Note that Orbix ignores the second and third parameters to CORBACreateObject.

NOTE: Siebel eScript has built-in exception support for CORBA objects. Use the try Statement and the throw Statement to build exception handlers. Orbix does not support built-in user exceptions for CORBA objects.

CORBACreateObject and any call to CORBA objects can throw CORBAObjException in addition to exceptions declared in the IDL file.

Exceptions can be caught in the eScript engine using try catch clauses. Exception objects always have a name and are accessible with name data member.

In general, if the exception occurs while executing a CORBA function, the name of the exception object is CORBAObjException. If the error occurs in Siebel code, the exception name is SiebelException.

User exceptions are not supported for Visibroker, where the exceptions declared in the IDL file are mapped to the corresponding eScript objects. The exception name is the one declared in the IDL.

For example, the user can declare a completed exception object in the IDL as follows:

```
exception DataException {
         string moduleId;
         string messageText;
         long schProcRtrnCd;
         long appRtrnCd;
         string addtnlText;

         ErrorCode errCode;
         string errDesc;
         string fieldName;
         long fieldOccurs;

         long fldMsgCd;
         string fldMsg;

         ExpSource expSource;

};
```

Whenever this exception is thrown, the eScript catch clause can access this particular object data member using this syntax:

```
if (obj.name == "DataException ")
{
   TheApplication().MsgBox(obj.moduleId);
   ... any other data members...

return (CancelOpertaion);
}
```

Example This example instantiates a CORBA object and calls several methods on it.

```
var cObj = myCorbaOb.balance () ; //call a method
[check the return value…]

   myCorbaOb.SetBalance (50000); //call another method
   var acctNum = myCorbaOb.accountNumber ; // get the property
value
```

```
      myCorbaOb.accountNumber = accNum ; //set it.
```

For more information on configuring the Siebel Client with the CORBACreateObject() Method, refer to the section on the JSECorbaConnector parameter in *Siebel Web Client Administration Guide*.

defined() Method

This function tests whether a variable or object property has been defined.

Syntax defined(*var*)

Parameter	Description
var	The variable or object property you wish to query

Returns True if the item has been defined; otherwise, false.

Usage This function tests whether a variable or object property has been defined, returning true if it has or false if it has not.

CAUTION: The defined() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

Example The following fragment illustrates two uses of the defined() method. The first use checks a variable, `t`. The second use checks an object `t.t`.

```
var t = 1;
   if (defined(t))
      TheApplication().Trace("t is defined");
   else
      TheApplication().Trace("t is not defined");

   if (!defined(t.t))
      TheApplication().Trace("t.t is not defined"):
   else
      TheApplication().Trace("t.t is defined");
```

See Also “undefine() Method” on page 271

escape() Method

The escape() method receives a string and replaces special characters with escape sequences.

Syntax

```
escape(string)
```

Parameter	Description
string	The string containing characters to be replaced

Returns A string with special characters replaced by Unicode sequences.

Usage The escape() method receives a string and replaces special characters with escape sequences, so that the string may be used with a URL. The escape sequences are Unicode values. For characters in the standard ASCII set (values 0 through 127 decimal), these are the hexadecimal ASCII codes of the characters preceded by percent signs.

Uppercase and lowercase letters, numbers, and the special symbols @ * + _ . / remain in the string. Other characters are replaced by their respective Unicode sequences.

Example The following code provides an example of what occurs once a string has been encoded. Note that the @ and * characters have not been replaced.

```
var str = "@#$*96!";
```

Would result in the following string: `"@#$*%!"`

```
var encodeStr = encode("@#$*%!");
```

Would result in the following string: `"@%23%24*%25%21"`

See Also “unescape(string) Method” on page 272

eval() Method

This method returns the value of its parameter, which is an expression.

Syntax `eval(`*`expression`*`)`

Parameter	Description
expression	The expression to be evaluated

Returns The value of *expression*.

Usage This method evaluates whatever is represented by *expression*. If *expression* is a string, the interpreter tries to interpret the string as if it were JavaScript code. If successful, the method returns the value of *expressio*n. If not successful, it returns the special value `undefined`.

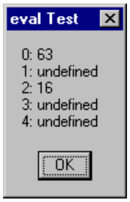
If the expression is not a string, *expression* is returned. For example, calling eval(5) returns the value 5.

Example This example shows the result of using the eval() method on several types of expressions. Note that the string expression in the `test[0]` variable is evaluated because it can be interpreted as a JavaScript statement, but the string expressions in `test[1]` and `test[3]` are undefined.

```
function clickme_Click ()
{
   var msgtext = "";
   var a = 7;
   var b = 9;
   var test = new Array(4);
   var test[0] = "a * b";
   var test[1] = toString(a * b);
   var test[2] = a + b;
   var test[3] = "Strings are undefined.";
   var test[4] = test[1] + test[2];


   for (var i = 0; i < 5; i++)
      msgtext = msgtext + i + ": " + eval(test[i]) + "\n";
   TheApplication().RaiseErrorText(msgtext);
```

Running this code produces the following result.



```
0: 63
1: undefined
2: 16
3: undefined
4: undefined
```

getArrayLength() Method

This function returns the length of a dynamically created array.

Syntax `getArrayLength(array[, minIndex])`

Parameter	Description
array	The name of the array whose length you wish to find
minIndex	The index of the lowest element at which to start counting

Returns The length of a dynamic array, which is one more than the highest index of an array.

Usage Most commonly, the first element of an array is at index 0. If *minIndex* is supplied, then it is used to set to the minimum index, which is zero or less.

This function should be used with dynamically created arrays, that is, with arrays that were not created using the Array() constructor and the new operator. The length property is not available for dynamically created arrays. Dynamically created arrays must use the getArrayLength() and setArrayLength() functions when working with array lengths.

When working with arrays created using the Array() constructor and the new operator, use the length property of the arrays.

CAUTION: The getArrayLength() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also “The Array Constructor” on page 98, “length Property” on page 99, and “setArrayLength() Method” on page 261

isNaN() Method

The isNaN() method determines whether its parameter is or is not a number.

Syntax

```
isNaN(value)
```

Parameter	Description
value	The variable or expression to be evaluated

Returns True if *value* is not a number; otherwise, false.

Usage The isNaN() method determines whether *value* is or is not a number, returning true if it is not or false if it is. *Value* must be in italics.

If *value* is an object reference, IsNan() always returns true, because object references are not numbers.

Example IsNaN("123abc") returns true.

IsNaN("123") returns false.

IsNaN("999888777123") returns false.

IsNaN("The answer is 42") returns true.

See Also “isFinite() Method” on page 259

isFinite() Method

This method determines whether its parameter is a finite number.

Syntax `isFinite(`*`value`*`)`

Parameter	Description
value	The variable or expression to be evaluated

Returns True if *value* is or can be converted to a number; false if *value* evaluates to NaN, POSITIVE_INFINITY, or NEGATIVE_INFINITY.

Usage The isFinite() method returns true if *number* is or can be converted to a number. If the parameter evaluates to NaN, *number*.POSITIVE_INFINITY, or *number*.NEGATIVE_INFINITY, the method returns false. For details on the number object, read "Number Constants" on page 57.

See Also "isNaN() Method" on page 259

parseFloat() Method

This method converts an alphanumeric string to a floating-point decimal number.

Syntax `parseFloat(`*`string`*`)`

Parameter	Description
string	The string to be converted

Returns A floating-point decimal number; if *string* cannot be converted to a number, the special value NaN is returned.

Usage Whitespace characters at the beginning of the string are ignored. The first non-white-space character must be either a digit or a minus sign (-). Numeric characters in *string* are read. The first period (.) in *string* is treated as a decimal point and any following digits as the fractional part of the number. Reading stops at the first non-numeric character after the decimal point. The result is converted into a number. Characters including and following the first non-numeric character are ignored.

Example The following code fragment returns the result `-234.37`:

```
var num = parseFloat(" -234.37 profit");
```

parseInt() Method

This method converts an alphanumeric string to an integer number.

Syntax parseInt(*string*)

Parameter	Description
string	The string to be converted

Returns An integer number; if *string* cannot be converted to a number, the special value NaN is returned.

Usage Whitespace characters at the beginning of the string are ignored. The first non-white-space character must be either a digit or a minus sign (-). Numeric characters in *string* are read. Reading stops at the first non-numeric character. The result is converted into an integer number. Characters including and following the first non-numeric character are ignored.

Example The following code fragment returns the result -234:

```
var num = parseInt(" -234.37 profit");
```

setArrayLength() Method

This function sets the first index and length of an array.

Syntax setArrayLength(*array*[, *minIndex*], *length*])

Parameter	Description
array	The name of the array whose length you wish to find
minIndex	The index of the lowest element at which to start counting; must be 0 or less
length	The length of the array

Returns Not applicable

Usage This function sets the length of *array* to a range bounded by *minIndex* and *length*. If three arguments are supplied, *minIndex*, which must be 0 or less, is the minimum index of the newly sized array, and *length* is the length. Any elements outside the bounds set by *minIndex* and *length* 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.

CAUTION: The setArrayLength() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also "length Property" on page 99 and "getArrayLength() Method" on page 258

ToBoolean() Method

This method converts a value to the Boolean data type.

Syntax `ToBoolean(value)`

Parameter	Description
value	The value to be converted to a Boolean value

Returns A value that depends on *value*'s original data type, according to the following table:

Data Type	Returns
Boolean	*value*
buffer	False if an empty buffer; otherwise, true
null	False
number	False if *value* is 0, +0, -0, or NaN; otherwise, true
object	True

string	False if an empty string, `""`; otherwise, true
undefined	False

Usage This method converts *value* to the Boolean data type. The result depends on the original data type of *value*.

CAUTION: The ToBoolean() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also "ToBuffer() Method" on page 263, "ToObject() Method" on page 268, and "ToString() Method" on page 268

ToBuffer() Method

This function converts its parameter to a buffer.

Syntax

```
ToBuffer(value)
```

Parameter	Description
value	The value to be converted to a buffer

Returns A sequence of ASCII bytes that depends on *value*'s original data type, according to the following table:

Data Type	Returns
Boolean	The string `"false"` if *value* is false; otherwise, `"true"`
null	The string `"null"`
number	If *value* is NaN, `"NaN"`. If *value* is +0 or -0, `"0"`; if *value* is POSITIVE_INFINITY or NEGATIVE_INFINITY, `"Infinity"`; if *value* is a number, a string representing the number
object	The string `"[object Object]"`

string	The text of the string
undefined	The string `"undefined"`

Usage This function converts *value* to a buffer; what is placed in the buffer is a character array of ASCII bytes.

CAUTION: The ToBuffer() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also "ToBytes() Method" on page 264 and "ToString() Method" on page 268

ToBytes() Method

This function places its parameter in a buffer.

Syntax `ToBytes(value)`

Parameter	Description
value	The value to be placed in a buffer

Returns Not applicable

Usage This function transfers the raw data represented by *value* to a buffer. The raw transfer does not convert Unicode values to corresponding ASCII values. Thus, for example, the Unicode string `Hit` would be stored as `\OH\Oi\Ot`, that is, as the hexadecimal sequence `00 48 00 69 00 74`.

CAUTION: The ToBytes() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also “ToBuffer() Method” on page 263 and “ToString() Method” on page 268

ToInt32() Method

This function converts its parameter to an integer in the range of -2^{31} through 2^{31} - 1.

Syntax `ToInt32(value)`

Parameter	Description
value	The value to be converted to an integer

Returns If the result is NaN, +0. If the result is +0 or -0, `0`. If the result is POSITIVE_INFINITY, or NEGATIVE_INFINITY, `Infinity`. Otherwise, the integer part of the number, rounded toward 0.

Usage This function converts *value* to an integer in the range of -2^{31} through 2^{31} - 1 (that is, -2,147,483,648 to 2,147,483,647). To use it without error, first pass *value* to isNaN() or to ToNumber().

To use isNan(), use a statement in the form

```
if (isNaN(value))
.
.   [error-handling statements];
.
else
   ToInt32(value);
```

Because ToInt32() truncates rather than rounds the value it is given, numbers are rounded toward 0. That is, `-12.88` becomes `-12`; `12.88` becomes `12`.

CAUTION: The ToInt32() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also “ToInteger() Method” on page 266, “ToNumber() Method” on page 267, “ToUint16() Method” on page 269, and “ToUint32() Method” on page 270

ToInteger() Method

This function converts its parameter to an integer in the range of -2^{15} to $2^{15} - 1$.

Syntax

```
ToInteger(value)
```

Parameter	Description
value	The value to be converted to an integer

Returns

If the result is NaN, +0. If the result is +0, -0, POSITIVE_INFINITY, or NEGATIVE_INFINITY, the result. Otherwise, the integer part of the number, rounded toward 0.

Usage

This function converts *value* to an integer in the range of -2^{15} to $2^{15} - 1$ (that is, -32,768 to 32,767). To use it without error, first pass *value* to isNaN() or to ToNumber().

To use toNumber(), use a statement of the form

```
var x;
x = toNumber(value);
(if x == 'NaN')
.
.   [error -handling statements];
.
else
   ToInteger(value);
```

Because ToInteger() truncates rather than rounds the value it is given, numbers are rounded toward 0. That is, `-12.88` becomes `-12`; `12.88` becomes `12`.

CAUTION: The ToInteger() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToInt32() Method" on page 265, "ToNumber() Method" on page 267, "ToString() Method" on page 268, "ToUint16() Method" on page 269, "ToUint32() Method" on page 270, and "Math.round() Method" on page 289

ToNumber() Method

This function converts its parameter to a number.

Syntax

```
ToNumber(value)
```

Parameter	Description
value	The value to be converted to a number

Returns

A value that depends on *value*'s original data type, according to the following table:

Data Type	Returns
Boolean	+0 if *value* is false, 1 if *value* is true
buffer	*value* if successful; otherwise, NaN
null	0
number	*value*
object	NaN
string	*value* if successful; otherwise, NaN
undefined	NaN

Usage

This function converts its parameter to a number.

CAUTION: The ToNumber() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToInt32() Method" on page 265, "ToInteger() Method" on page 266, "ToString() Method" on page 268, "ToUint16() Method" on page 269, "ToUint32() Method" on page 270, and "Math.round() Method" on page 289

ToObject() Method

This function converts its parameter to an object.

Syntax

```
ToObject(value)
```

Parameter	Description
value	The value to be converted to an object

Returns A value that depends on *value*’s original data type, according to the following table:

Data Type	Returns
Boolean	A new Boolean object having the value *value*
null	(Generates a run-time error)
number	A new Number object having the value *value*
object	*value*
string	A new string object having the value *value*
undefined	(Generates a run-time error)

Usage This function converts its parameter to an object.

CAUTION: The ToObject() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also “ToString() Method” on page 268

ToString() Method

This method converts its parameter to a string.

Syntax `ToString(value)`

Parameter	Description
value	The value to be converted to a string

Returns A value in the form of a Unicode string, the contents of which depends on *value*'s original data type, according to the following table:

Data Type	Returns
Boolean	`"false"` if *value* is false; otherwise, `"true"`
null	The string `"null"`
number	If *value* is NaN, `"NaN"`. If *value* is +0 or -0, `"0"`; if Infinity, `"Infinity"`; if a number, a string representing the number
object	The string `"[object Object]"`
string	*value*
undefined	The string `"undefined"`

Usage This method converts its parameter to a Unicode string, the contents of which depend on *value*'s original data type.

CAUTION: The ToString() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

Example For an example, read "eval() Method" on page 257.

See Also "ToBuffer() Method" on page 263 and "ToBytes() Method" on page 264

ToUint16() Method

This function converts its parameter to an integer in the range of 0 through 2^{16} -1.

Syntax ToUint16(*value*)

Parameter	Description
value	The value to be converted

Returns If the result is NaN, +0. If the result is +0 , 0. If the result is POSITIVE_INFINITY, it returns Infinity. Otherwise, it returns the unsigned (that is, absolute value of) integer part of the number, rounded toward 0.

Usage This function converts *value* to an integer in the range of 0 to 2^{16} - 1 (65,535). To use it without error, first pass *value* to isNaN() or to ToNumber().

To use toNumber(), use a statement of the form

```
var x;i
x = toNumber(value);
(if x == 'NaN')
.
.   [error -handling statements];
.
else
   ToUint16(value);
```

Because ToUint16() truncates rather than rounds the value it is given, numbers are rounded toward 0. Therefore, 12.88 becomes 12.

CAUTION: The ToUint16() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also "ToInt32() Method" on page 265, "ToInteger() Method" on page 266, "ToNumber() Method" on page 267, "ToUint32() Method" on page 270, and "Math.round() Method" on page 289

ToUint32() Method

This function converts its parameter to an integer in the range of 0 to 2^{32} -1.

Syntax `ToUint32(value)`

Parameter	Description
value	The value to be converted

Returns If the result is NaN, +0. If the result is +0 , 0. If the result is POSITIVE_INFINITY, it returns `Infinity`. Otherwise, it returns the unsigned (that is, absolute value of) integer part of the number, rounded toward 0.

Usage This function converts *value* to an unsigned integer part of *value* in the range of 0 through 2^{32} - 1 (4,294,967,296). To use it without error, first pass *value* to isNaN() or to ToNumber().

To use isNan() without error, use a statement of the form

```
if (isNaN(value))
.
.   [error-handling statements];
.
else
   ToUint32(value);
```

Because ToUint32() truncates rather than rounds the value it is given, numbers are rounded toward 0. Therefore, `12.88` becomes `12`.

CAUTION: The ToUint32() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

See Also "ToInt32() Method" on page 265, "ToInteger() Method" on page 266, "ToNumber() Method" on page 267, "ToUint16() Method" on page 269, and "Math.round() Method" on page 289

undefine() Method

This function undefines a variable, Object property, or value.

Syntax `undefine(value)`

Parameter	Description
value	The variable or object property to be undefined

Returns Not applicable

Usage If a value was previously defined so that its use with the defined() method returns true, then after using undefine() with the value, defined() returns false. Undefining a value is not the same as setting a value to null. In the following fragment, the variable n is defined with the number value of 2, and then undefined.

```
var n = 2;
undefine(n);
```

CAUTION: The undefine() function is unique to Siebel eScript. Avoid using it in a script that may be used with a JavaScript interpreter that does not support it.

Example 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);
```

See Also "CORBACreateObject() Method" on page 252

unescape(string) Method

The unescape() method removes escape sequences from a string and replaces them with the relevant characters.

Syntax `unescape(`*`string`*`)`

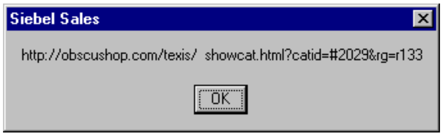
Parameter	Description
string	A string literal or string variable from which escape sequences are to be removed

Returns A string with Unicode sequences replaced by the equivalent ASCII characters.

Usage The unescape() method is the reverse of the escape() method; it removes escape sequences from a string and replaces them with the relevant characters.

Example The following line of code displays the string in its parameter with the escape sequence replaced by printable characters. Note that `%20` is the Unicode representation of the space character. Note also that this example would normally appear on a single line, as strings cannot be broken by a newline.

```
TheApplication().RaiseErrorText(unescape("http://obscushop.com/
texis/%20%20showcat.html?catid=%232029&rg=r133"));
```



```
http://obscushop.com/texis/  showcat.html?catid=#2029&rg=r133
```

See Also “escape() Method” on page 256

The Math Object

The Math object in Siebel eScript has a full and powerful set of methods and properties for mathematical operations. A programmer has a rich set of mathematical tools for the task of doing mathematical calculations in a script.

Properties

- "Math.E Property" on page 281
- "Math.LN10 Property" on page 282
- "Math.LN2 Property" on page 283
- "Math.LOG2E Property" on page 284
- "Math.PI Property" on page 287
- "Math.LOG10E Property" on page 285
- "Math.SQRT2 Property" on page 291
- "Math.SQRT1_2 Property" on page 291

Methods

- "Math.abs() Method" on page 275
- "Math.acos() Method" on page 275
- "Math.asin() Method" on page 276
- "Math.atan() Method" on page 277
- "Math.atan2() Method" on page 278
- "Math.ceil() Method" on page 279
- "Math.cos() Method" on page 280
- "Math.exp() Method" on page 281
- "Math.floor() Method" on page 282
- "Math.log() Method" on page 284

- "Math.max() Method" on page 286
- "Math.min() Method" on page 286
- "Math.pow() Method" on page 287
- "Math.random() Method" on page 288
- "Math.round() Method" on page 289
- "Math.sin() Method" on page 290
- "Math.sqrt() Method" on page 290
- "Math.tan() Method" on page 292

Math.abs() Method

This method returns the absolute value of its parameter; it returns NaN if the parameter cannot be converted to a number.

Syntax

```
Math.abs(number)
```

Parameter	Description
number	A numeric literal or numeric variable

Returns The absolute value of *number*; or NaN if *number* cannot be converted to a number.

Usage This method returns the absolute value of *number*. If *number* cannot be converted to a number, it returns NaN.

Math.acos() Method

This method returns the arc cosine of its parameter, expressed in radians.

Syntax `Math.acos(`*`number`*`)`

Parameter	Description
number	A numeric literal or numeric variable

Returns The arc cosine of *number*, expressed in radians from 0 to pi, or NaN if *number* cannot be converted to a number or is greater than 1 or less than -1.

Usage This method returns the arc cosine of *number*. The return value is expressed in radians and ranges from 0 to pi. It returns NaN if x cannot be converted to a number, is greater than 1, or is less than -1.

To convert radians to degrees, multiply by `180/Math.PI.`

See Also "Math.asin() Method" on page 276, "Math.atan() Method" on page 277, "Math.cos() Method" on page 280, and "Math.sin() Method" on page 290

Math.asin() Method

This method returns an implementation-dependent approximation of the arcsine of its parameter.

Syntax `Math.asin(`*`number`*`)`

Parameter	Description
number	A numeric literal or numeric variable

Returns An implementation-dependent approximation of the arcsine of *number,* expressed in radians and ranging from - pi/2 to +pi/2.

Usage This method returns an implementation-dependent approximation of the arcsine of *number*. The return value is expressed in radians and ranges from -pi/2 to +pi/2. It returns NaN if *number* cannot be converted to a number, is greater than 1, or is less than -1.

To convert radians to degrees, multiply by `180/Math.PI`.

See Also “Math.acos() Method” on page 275, “Math.atan() Method” on page 277, “Math.atan2() Method” on page 278, “Math.cos() Method” on page 280, “Math.sin() Method” on page 290, and “Math.tan() Method” on page 292

Math.atan() Method

This method returns an implementation-dependent approximation of the arctangent of the argument.

Syntax `Math.atan(`*`number`*`)`

Parameter	Description
number	A numeric literal or numeric variable

Returns An implementation-dependent approximation of the arctangent of *number*, expressed in radians.

Usage The Math.atan() function returns an implementation-dependent approximation of the arctangent of the argument. The return value is expressed in radians and ranges from -pi/2 to +pi/2.

The function assumes *number* is the ratio of two sides of a right triangle: the side opposite the angle to find and the side adjacent to the angle. The function returns a value for the ratio.

To convert radians to degrees, multiply by `180/Math.PI`.

Example This example finds the roof angle necessary for a house with an attic ceiling of 8 feet (at the roof peak) and a 16-foot span from the outside wall to the center of the house. The Math.atan() function returns the angle in radians; it is multiplied by 180/PI to convert it to degrees. Compare the example in the discussion of “Math.atan2() Method” on page 278 to understand how the two arctangent functions differ. Both examples return the same value.

```
function RoofBtn_Click ()
{
   var height = 8;
   var span = 16;
   var angle = Math.atan(height/span)*(180/Math.PI);

   TheApplication().RaiseErrorText("The angle is " +
      Clib.rsprintf("%5.2f", angle) + " degrees.")
}
```

See Also "Math.acos() Method" on page 275, "Math.asin() Method" on page 276, "Math.atan2() Method" on page 278, "Math.cos() Method" on page 280, "Math.sin() Method" on page 290, and "Math.tan() Method" on page 292

Math.atan2() Method

This function returns an implementation-dependent approximation to the arctangent of the quotient of its arguments.

Syntax `Math.atan2(y, x)`

Parameter	Description
y	The value on the y axis
x	The value on the x axis

Returns An implementation-dependent approximation of the arctangent of *y*/*x*, in radians.

Usage This function returns an implementation-dependent approximation to the arctangent 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. It is intentional and traditional for the two-argument arctangent 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 This example finds the roof angle necessary for a house with an attic ceiling of 8 feet (at the roof peak) and a 16-foot span from the outside wall to the center of the house. The Math.atan2() function returns the angle in radians; it is multiplied by 180/PI to convert it to degrees. Compare the example in the discussion of "Math.atan() Method" on page 277 to understand how the two arctangent functions differ. Both examples return the same value.

```
function RoofBtn2_Click ()
{
   var height = 8;
   var span = 16;
   var angle = Math.atan2(span, height)*(180/Math.PI);

   TheApplication().RaiseErrorText("The angle is " +
   Clib.rsprintf("%5.2f", angle) + " degrees.")
}
```

See Also "Math.acos() Method" on page 275, "Math.asin() Method" on page 276, "Math.atan() Method" on page 277, "Math.cos() Method" on page 280, "Math.sin() Method" on page 290, and "Math.tan() Method" on page 292

Math.ceil() Method

This method returns the smallest integer that is not less than its parameter.

Syntax `Math.ceil(`*`number`*`)`

Parameter	Description
number	A numeric literal or numeric variable

Returns The smallest integer that is not less than *number*; if *number* is an integer, *number*.

Usage This method returns the smallest integer that is not less than *number*. If the argument is already an integer, the result is the argument itself. It returns NaN if *number* cannot be converted to a number.

Example The following code fragment generates a random number between 0 and 100 and displays the integer range in which the number falls. Each run of this code produces a different result.

```
var x = Math.random() * 100;
   TheApplication().RaiseErrorText("The number is between " +
      Math.floor(x) + " and " + Math.ceil(x) + ".");
```

See Also “Math.floor() Method” on page 282

Math.cos() Method

This method returns an implementation-dependent approximation of the cosine of the argument. The argument is expressed in radians.

Syntax `Math.cos(`*`number`*`)`

Parameter	Description
number	A numeric literal or numeric variable representing an angle in radians

Returns An implementation-dependent approximation of the cosine of *number.*

Usage The return value is between -1 and 1. NaN is returned if *number* cannot be converted to a number.

The angle can be either positive or negative. To convert degrees to radians, multiply by `Math.PI/180`.

Example This example finds the length of a roof, given its pitch and the distance of the house from its center to the outside wall.

```
function RoofBtn3_Click ()
{
   var pitch;
   var width;
   var roof;

   pitch = 35;
   pitch = Math.cos(pitch*(Math.PI/180));
   width = 75;
   width = width / 2;
   roof = width/pitch;
```

```
      TheApplication().RaiseErrorText("The length of the roof is " +
            Clib.rsprintf("%5.2f", roof) + " feet.");
   }
```

See Also "Math.acos() Method" on page 275, "Math.asin() Method" on page 276, "Math.atan() Method" on page 277, "Math.atan2() Method" on page 278, "Math.sin() Method" on page 290, and "Math.tan() Method" on page 292

Math.E Property

This property stores the number value for *e*, the base of natural logarithms.

Syntax `Math.E`

Parameter	Description
Not applicable	

Returns Not applicable

Usage The value of *e* is represented internally as approximately 2.7182818284590452354.

See Also "Math.exp() Method" on page 281, "Math.LN10 Property" on page 282, "Math.LN2 Property" on page 283, "Math.log() Method" on page 284, "Math.LOG2E Property" on page 284, and "Math.LOG10E Property" on page 285

Math.exp() Method

This method returns an implementation-dependent approximation of the exponential function of its parameter.

Syntax `Math.exp(`*`number`*`)`

Parameter	Description
number	The exponent value of *e*

Returns The value of *e* raised to the power *number*.

Usage This method returns an implementation-dependent approximation of the exponential function of its parameter. The argument, that is, returns *e* raised to the power of the *x*, where *e* is the base of the natural logarithms. NaN is returned if *number* cannot be converted to a number. The value of *e* is represented internally as approximately 2.7182818284590452354.

See Also "Math.E Property" on page 281, "Math.LN10 Property" on page 282, "Math.LN2 Property" on page 283, "Math.log() Method" on page 284, "Math.LOG2E Property" on page 284, and "Math.LOG10E Property" on page 285

Math.floor() Method

This method returns the greatest integer that is not greater than its parameter.

Syntax

```
Math.floor(number)
```

Parameter	Description
number	A numeric literal or numeric variable

Returns The greatest integer that is not greater than *number*; if number is an integer, *number*.

Usage This method returns the greatest integer that is not greater than *number*. If the argument is already an integer, the result is the argument itself. It returns NaN if *number* cannot be converted to a number.

Example For an example, read "Math.ceil() Method" on page 279.

See Also "Math.ceil() Method" on page 279

Math.LN10 Property

This property stores the number value for the natural logarithm of 10.

Syntax `Math.LN10`

Parameter	Description
Not applicable	

Returns Not applicable

Usage The value of the natural logarithm of 10 is represented internally as approximately 2.302585092994046.

See Also "Math.exp() Method" on page 281, "Math.LN2 Property" on page 283, "Math.log() Method" on page 284, "Math.LOG2E Property" on page 284, and "Math.LOG10E Property" on page 285

Math.LN2 Property

This property stores the number value for the natural logarithm of 2.

Syntax `Math.LN2`

Parameter	Description
Not applicable	

Returns Not applicable

Usage The value of the natural logarithm of 2 is represented internally as approximately 0.6931471805599453.

See Also "Math.E Property" on page 281, "Math.exp() Method" on page 281, "Math.LN10 Property" on page 282, "Math.log() Method" on page 284, "Math.LOG2E Property" on page 284, and "Math.LOG10E Property" on page 285

Math.log() Method

This function returns an implementation-dependent approximation of the natural logarithm of its parameter.

Syntax `Math.log(number)`

Parameter	Description
number	A numeric literal or numeric variable

Returns An implementation-dependent approximation of the natural logarithm of *number*.

Example This example uses the Math.log() function to determine which number is larger: 999^1000 (999 to the 1000th power) or 1000^999 (1000 to the 999th power). Note that if you attempt to use the Math.pow() function instead of the Math.log() function with numbers this large, the result returned would be `Infinity`.

```
function Test_Click ()
{
   var x = 999;
   var y = 1000;
   var a = y*(Math.log(x));
   var b = x*(Math.log(y))
   if ( a > b )
      TheApplication().
         RaiseErrorText("999^1000 is greater than 1000^999.");
   else
      TheApplication().
         RaiseErrorText("1000^999 is greater than 999^1000.");
}
```

See Also "Math.E Property" on page 281, "Math.exp() Method" on page 281, "Math.LN10 Property" on page 282, "Math.LN2 Property" on page 283, "Math.LOG2E Property" on page 284, "Math.LOG10E Property" on page 285, and "Math.pow() Method" on page 287

Math.LOG2E Property

This property stores the number value for the base 2 logarithm of *e*, the base of the natural logarithms.

Syntax `Math.LOG2E`

Parameter	Description
Not applicable	

Returns Not applicable

Usage The value of the base 2 logarithm of *e* is represented internally as approximately 1.4426950408889634. The value of Math.LOG2E is approximately the reciprocal of the value of Math.LN2.

See Also "Math.E Property" on page 281, "Math.exp() Method" on page 281, "Math.LN10 Property" on page 282, "Math.LN2 Property" on page 283, "Math.log() Method" on page 284, and "Math.LOG10E Property" on page 285

Math.LOG10E Property

The number value for the base 10 logarithm of *e*, the base of the natural logarithms.

Syntax `Math.LOG10E`

Parameter	Description
Not applicable	

Returns Not applicable

Usage The value of the base 10 logarithm of *e* is represented internally as approximately 0.4342944819032518. The value of Math.LOG10E is approximately the reciprocal of the value of Math.LN10.

See Also "Math.E Property" on page 281, "Math.exp() Method" on page 281, "Math.LN10 Property" on page 282, "Math.LN2 Property" on page 283, "Math.log() Method" on page 284, and "Math.LOG2E Property" on page 284

Math.max() Method

This function returns the larger of its parameters.

Syntax

```
Math.max(x, y)
```

Parameter	Description
x	A numeric literal or numeric variable
y	A numeric literal or numeric variable

Returns The larger of *x* and *y*.

Usage This function returns the larger of *x* and *y*, or NaN if either parameter cannot be converted to a number.

See Also "Math.min() Method" on page 286

Math.min() Method

This function returns the smaller of its parameters.

Syntax

```
Math.min(x, y)
```

Parameter	Description
x	A numeric literal or numeric variable
y	A numeric literal or numeric variable

Returns The smaller of *x* and *y*.

Usage This function returns the smaller of *x* and *y*, or NaN if either parameter cannot be converted to a number.

See Also "Math.max() Method" on page 286

Math.PI Property

This property holds the number value for pi.

Syntax

```
Math.PI
```

Parameter	Description
Not applicable	

Returns Not applicable

Usage This property holds the value of pi, which is the ratio of the circumference of a circle to its diameter. This value is represented internally as approximately 3.14159265358979323846.

Example For examples, read “Math.atan() Method” on page 277, “Math.atan2() Method” on page 278, and “Math.cos() Method” on page 280.

Math.pow() Method

This function returns the value of its first parameter raised to the power of its second parameter.

Syntax

```
Math.pow(x, y)
```

Parameter	Description
x	The number to be raised to a power
y	The power to which to raise *x*

Returns The value of *x* to the power of *y*.

Usage This function returns the value of *x* raised to the power of *y*.

Example This example uses the Math.pow() function to determine which number is larger: 99^100 (99 to the 100th power) or 100^99 (100 to the 99th power). Note that if you attempt to use the Math.pow() method with numbers as large as those used in the example in "Math.log() Method" on page 284, the result returned is `Infinity`.

```
function Test_Click ()
{
   var a = Math.pow(99, 100);
   var b = Math.pow(100, 99);
   if ( a > b )
      TheApplication().RaiseErrorText("99^100 is greater than 
100^99.");
   else
      TheApplication().RaiseErrorText("100^99 is greater than 
99^100.");
}
```

See Also "Math.exp() Method" on page 281, "Math.log() Method" on page 284, and "Math.sqrt() Method" on page 290

Math.random() Method

This function returns a pseudo-random number between 0 and 1.

Syntax `Math.random()`

Parameter	Description
Not applicable	

Returns A pseudo-random number between 0 and 1.

Usage This function generates a pseudo-random number between 0 and 1. It takes no arguments. Where possible, it should be used in place of the Clib.rand() method. The Clib.rand() method is to be preferred only when it is necessary to use Clib.srand() to seed the Clib random number generator with a specific value.

Example This example generates a random string of characters within a range. The Math.random() function is used to set the range between lowercase *a* and *z*.

```
function Test_Click ()
{
   var str1 = "";
   var letter;
   var randomvalue;
   var upper = "z";
   var lower = "a";

   upper = upper.charCodeAt(0);
   lower = lower.charCodeAt(0);

   for (var x = 1; x < 26; x++)
   {
      randomvalue = Math.round(((upper - (lower + 1)) *
         Math.random())  + lower);
      letter = String.fromCharCode(randomvalue);
      str1 = str1 + letter;
   }

   TheApplication().RaiseErrorText(str1);
}
```

See Also "Clib.rand() Method" on page 183 and "Clib.srand() Method" on page 189

Math.round() Method

This method rounds a number to its nearest integer.

Syntax `Math.round(`*`number`*`)`

Parameter	Description
number	A numeric literal or numeric variable

Returns The integer closest in value to *number*.

Usage The *number* parameter is rounded up if its fractional part is equal to or greater than 0.5 and is rounded down if less than 0.5. Both positive and negative numbers are rounded to the nearest integer.

Example This code fragment yields the values 124 and -124.

```
var a = Math.round(123.6);
var b = Math.round(-123.6)
TheApplication().RaiseErrorText(a + "\n" + b)
```

See Also "Clib.modf() Method" on page 179, "ToInt32() Method" on page 265, "ToInteger() Method" on page 266, "ToUint16() Method" on page 269, and "ToUint32() Method" on page 270

Math.sin() Method

This method returns the sine of an angle expressed in radians.

Syntax `Math.sin(`*`number`*`)`

Parameter	Description
number	A numeric expression containing a number representing the size of an angle in radians

Returns The sine of *number,* or NaN if *number* cannot be converted to a number.

Usage The return value is between -1 and 1. The angle is specified in radians and can be either positive or negative.

To convert degrees to radians, multiply by `Math.PI/180`.

See Also "Math.acos() Method" on page 275, "Math.asin() Method" on page 276, "Math.atan() Method" on page 277, "Math.atan2() Method" on page 278, "Math.cos() Method" on page 280, and "Math.tan() Method" on page 292

Math.sqrt() Method

This method returns the square root of its parameter; it returns NaN if x is a negative number or cannot be converted to a number.

Syntax `Math.sqrt()`

Parameter	Description
number	A numeric literal or numeric variable

Returns The square root of *number*, or NaN if *number* is negative or cannot be converted to a number.

Usage This method returns the square root of *number*, or Nan if *number* is negative or cannot be converted to a number.

See Also "Math.exp() Method" on page 281, "Math.log() Method" on page 284, and "Math.pow() Method" on page 287

Math.SQRT1_2 Property

This property stores the number value for the square root of ½.

Syntax `Math.SQRT1_2`

Parameter	Description
Not applicable	

Returns Not applicable

Usage This property stores the number value for the square root of ½, which is represented internally as approximately 0.7071067811865476. The value of Math.SQRT1_2 is approximately the reciprocal of the value of Math.SQRT2.

See Also "Math.sqrt() Method" on page 290 and "Math.SQRT2 Property" on page 291

Math.SQRT2 Property

This property stores the number value for the square root of 2.

Syntax `Math.SQRT2`

Parameter	Description
Not applicable	

Returns Not applicable

Usage This property stores the number value for the square root of 2, which is represented internally as approximately 1.4142135623730951.

See Also "Math.sqrt() Method" on page 290 and "Math.SQRT1_2 Property" on page 291

Math.tan() Method

This method returns the tangent of its parameter.

Syntax `Math.tan(`*`number`*`)`

Parameter	Description
number	A numeric expression containing the number of radians in the angle whose tangent is to be returned

Returns The tangent of *number*, or NaN if *number* cannot be converted to a number.

Usage This method returns the tangent of *number*, expressed in radians, or NaN if *number* cannot be converted to a number. To convert degrees to radians, multiply by `Math.PI/180`.

See Also "Math.acos() Method" on page 275, "Math.asin() Method" on page 276, "Math.atan() Method" on page 277, "Math.atan2() Method" on page 278, "Math.cos() Method" on page 280, and "Math.sin() Method" on page 290

User-Defined 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 the name of the property, separated by a period. Any valid variable name may be used as a property name. For example, the code fragment that follows assigns values to the width and height properties of a rectangle object, calculates the area of a rectangle, and displays the result:

```
var Rectangle;

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

TheApplication().RaiseErrorText(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 its own values for width and height.

See Also "Predefining Objects with Constructor Functions" on page 293, "Assigning Functions to Objects" on page 294, and "Object Prototypes" on page 295

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 preceding example creates a Rectangle class and two instances of it. 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 you add the following line:

```
joe.motto = "Be prepared!";
```

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

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 line illustrates:

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

The function now uses the values for height and width that were defined when you 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 value of area2 to 15.

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 make sure that every instance 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 can be avoided by putting the shared function or property in an object's prototype. Then every instance of the object 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, every instance of that object is updated to include the prototype.

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

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

This fragment creates a value, which in this case is a function, for joe.area that supersedes 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.

NOTE: Prototypes cannot be declared inside a function scope.

Property Set Objects

Property set objects are collections of properties that can be used for storing data. They may have child property sets assigned to them. Property sets are used primarily for inputs and outputs to business services. You can assign child property sets to a property set to form a hierarchical data structure. Methods of property set objects are documented in the *Siebel Object Interfaces Reference*.

Method	Description
AddChild() Method	The AddChild() method is used to add subsidiary property sets to a property set, in order to form tree-structured data structures.
Copy() Method	Copy() returns a copy of a property set.
GetChild() Method	GetChild() returns a specified child property set of a property set.
GetChildCount() Method	GetChildCount() returns the number of child property sets attached to a parent property set.
GetFirstProperty() Method	GetFirstProperty() returns the name of the first property in a property set.
GetNextProperty() Method	GetNextProperty() returns the name of the next property in a property set.
GetProperty() Method	GetProperty() returns the value of a property, when given the property name.
GetPropertyCount() Method	GetPropertyCount() returns the number of properties associated with a property set.
GetType() Method	GetType() retrieves the data value stored in the type attribute of a property set.
GetValue() Method	GetValue() retrieves the data value stored in the value attribute of a property set.
InsertChildAt() Method	InsertChildAt() inserts a child property set into a parent property set at a specific location.
PropertyExists() Method	PropertyExists() returns a Boolean value indicating whether a specified property exists in a property set.

Method	Description
RemoveChild() Method	RemoveChild() removes a child property set from a parent property set.
RemoveProperty() Method	RemoveProperty() removes a property from a property set.
Reset() Method	This method removes every property and child property set from a property set.
SetProperty() Method	SetProperty() assigns a data value to a property in a property set.
SetType() Method	SetType() assigns a data value to a type member of a property set.
SetValue() Method	SetValue() assigns a data value to a value member of a property set.

The SElib Object

In Siebel eScript, the SElib object allows calling out to external libraries.

SElib.dynamicLink() Method

Windows Syntax

```
SElib.dynamicLink(Library, Procedure, Convention[, [desc,] arg1,
arg2, arg3, ..., argn])
```

UNIX Syntax

```
SElib.dynamicLink(Library, Procedure[, arg1, arg2, arg3, ...argn])
```

Parameter	Description
Library	Under Windows, the name of the DLL containing the procedure; under UNIX, the name of a shared object; can be specified by fully qualified path name
Procedure	The name or ordinal number of the procedure in the Library dynamic link library
Convention	The calling convention
desc	Used to pass a Unicode string; for example, WCHAR
arg1, arg2, arg3, ..., argn	Arguments to the procedure

Returns Not applicable

Usage The calling convention must be one of the following:

CDECL	Push right parameter first; the caller pops parameters
STDCALL	Push right parameter first; the caller pops parameters (this is almost always the option used in Win32)
PASCAL	Push left parameter first; the callee pops parameters

Values are passed as 32-bit values. If a parameter is undefined when SElib.dynamicLink() is called, then it is assumed that the parameter is a 32-bit value to be filled in; that is, the address of a 32-bit data element is passed to the function, and that function sets the value.

If any parameter is a structure, then it must be a structure that defines the binary data types in memory to represent the following variable. Before calling the function, the structure is copied to a binary buffer as described in "Blob.put() Method" on page 105 and "Clib.fwrite() Method" on page 164.

After calling the function, the binary data are converted back into the data structure according to the rules defined in Blob.get() and Clib.fread(). Data conversion is performed according to the current BigEndianMode setting. The function returns an integer.

Example

The following code example shows a proxy DLL that takes denormalized input values, creates the structure, and invokes a method in the destination DLL. In the process, it calls the SElib dynamicLink.

```
#include <windows.h>
_declspec(dllexport)  int __cdecl
score (
   double AGE,
   double AVGCHECKBALANCE,
   double AVGSAVINGSBALANCE,
   double CHURN_SCORE,
   double CONTACT_LENGTH,
   double HOMEOWNER,
   double *P_CHURN_SCORE,
   double *R_CHURN_SCORE,
   char   _WARN_[5] )
{
   *P_CHURN_SCORE = AGE + AVGCHECKBALANCE + AVGSAVINGSBALANCE;
   *R_CHURN_SCORE =  CHURN_SCORE + CONTACT_LENGTH + HOMEOWNER;
   strcpy(_WARN_, "SFD");
   return(1);
}
```

The following example shows the eScript code required to invoke a DLL. In this code, the Buffer is used for pointers and characters.

```
function TestDLLCall3()
{
   var AGE = 10;
   var AVGCHECKBALANCE = 20;
   var AVGSAVINGSBALANCE = 30;
   var CHURN_SCORE = 40;
   var CONTACT_LENGTH = 50;
   var HOMEOWNER = 60;
```

```
      var P_CHURN_SCORE = Buffer(8);
      var R_CHURN_SCORE = Buffer(8);
      var _WARN_ = Buffer(5);

   SElib.dynamicLink("jddll.dll", "score", CDECL,
      FLOAT64, AGE,
      FLOAT64, AVGCHECKBALANCE,
      FLOAT64, AVGSAVINGSBALANCE,
      FLOAT64, CHURN_SCORE,
      FLOAT64, CONTACT_LENGTH,
      FLOAT64, HOMEOWNER,
      P_CHURN_SCORE,
      R_CHURN_SCORE,
      _WARN_);

      var r_churn_score = R_CHURN_SCORE.getValue(8, "float");
      var p_churn_score = P_CHURN_SCORE.getValue(8, "float");
      var nReturns = r_churn_score + p_churn_score;
      return(nReturns);
      }
```

The following code calls a DLL function in the default codepage.

```
   var sHello = "Hello";
   Selib.dynamicLink("MyLib.dll", "MyFunc", CDECL,  sHello);
```

The following code calls a DLL function that passes Unicode strings.

```
   var sHello = "Hello";
   Selib.dynamicLink("MyLib.dll", "MyFunc", CDECL,  WCHAR, sHello);
```

The following code calls a DLL function that passes both Unicode and non-Unicode strings.

```
   var sHello = "Hello";
   var sWorld = "world";
   Selib.dynamicLink("MyLib.dll", "MyFunc", CDECL,  WCHAR, sHello,
   sWorld);
```

String Objects

The string data type is a hybrid that shares characteristics of primitive data types, Boolean and Number, and of composite data types, Object and Array. The string data type 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.

See Also “The String as Data Type” on page 303, “Escape Sequences for Characters” on page 303, “Single Quote Strings” on page 304, “Back-Quote Strings” on page 304, “The String as Object” on page 305, “charAt() Method” on page 305, “String.fromCharCode() Static Method” on page 306, “indexOf() Method” on page 306, “lastIndexOf() Method” on page 307, “length Property” on page 308, “split() Method” on page 309, “string.replace() Method” on page 311, “substring() Method” on page 312, “toLowerCase() Method” on page 313, and “toUpperCase() Method” on page 314

The 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 puts the string "hello" into the variable word. The second sets the variable word to have the same value as a previous variable hello.

```
var word = "hello";
word = hello;
```

Escape Sequences for Characters

Some characters, such as a quotation mark, have special meaning to the Siebel eScript interpreter and must be indicated with special character combinations when used in strings. This allows the Siebel eScript interpreter to distinguish between, for example, a quotation mark that is part of a string and a quotation mark that indicates the end of the string. The following is a list of the characters indicated by escape sequences:

\a	Audible bell
\b	Backspace

\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\’	Single quote
\”	Double quote
\\	Backslash character
\0###	Octal number (example: `'\033'` is the escape character)
\x##	Hex number (example: `'\x1B'` is the escape character)
\0	Null character (example: `'\0'` is the null character)
\u####	Unicode number (example: `'\u001B'` is the escape character)

Note that these escape sequences cannot be used within strings enclosed by back quotes, which are explained in “Back-Quote Strings” on page 304.

Single Quote Strings

You can declare a string with single quotes instead of double quotes. There is no difference between the two in eScript.

Back-Quote Strings

Siebel eScript 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. Special characters represented by a backslash followed by a letter, such as `\n`, cannot be used in back-quote 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 Siebel eScript method
```

Back-quote strings are not supported in most versions of JavaScript. Therefore, if you plan to port your script to some other JavaScript interpreter, do not use them.

The String as Object

Strings have both properties and methods, and they are listed in this section. These properties and methods are discussed as if strings were pure objects. Although strings are true objects, they do have instance properties and methods.

In the following pages, *stringVar* indicates any string variable. A specific instance of a variable should precede the period to use a property or call a method. The exception to this usage is a static method that actually uses the identifier String instead of a variable created as an instance of a string object.

charAt() Method

This method returns a character at a certain place in a string.

Syntax `stringVar.charAt(position)`

Parameter	Description
position	An integer indicating the position in the string of the character to be returned

Returns A string of length 1 representing the character at *position*.

Usage The character count starts at 0. To get the first character in a string, use index 0, as follows:

```
var string1 = "a string";
string1.charAt(0);
```

To get the last character in a string, use:

```
string1.charAt(string1.length - 1);
```

If position does not fall between 0 and `stringVar.length - 1`, *stringVar*.charAt() returns an empty string.

See Also "String.fromCharCode() Static Method" on page 306, "indexOf() Method" on page 306, and "lastIndexOf() Method" on page 307

String.fromCharCode() Static Method

This method returns a string created from the character codes that are passed to it as parameters.

Syntax `String.fromCharCode(`*`code1, code2, ... code`*`n)`

Parameter	Description
*code1, code2, ... code*n	Integers representing Unicode character codes

Returns A new string containing the characters specified by the codes.

Usage This static method allows you to create a string by specifying the individual Unicode values of the characters in it. The identifier String is used with this static method, instead of a variable name as with instance methods because it is a property of the String constructor. The arguments passed to this method are assumed to be Unicode values. The following line:

```
var string1 = String.fromCharCode(0x0041,0x0042);
```

sets the variable string1 to `"AB"`.

Example The following example uses the decimal Unicode values of the characters to create the string `"Siebel"`. For another example, read "offset[] Method" on page 115.

```
var seblStr = String.fromCharCode(83, 105, 101, 98, 101, 108);
```

See Also "Clib.toascii() Method" on page 208

indexOf() Method

This method returns the position of the first occurrence of a substring in a string.

stringVar.indexOf(*substring* [, *offset*])

Parameter	**Description**
substring	One or more characters to search for
offset	The position in the string at which to start searching, where 0 represents the first character

Returns The position of the first occurrence of a substring in a string variable.

Usage *stringVar*.indexOf() searches the entire substring in a string variable. The *substring* parameter may be a single character. If *offset* is not given, searching starts at position 0. If it is given, searching starts at the specified position.

For example,

```
var string = "what a string";
string.indexOf("a")
```

returns the position of the first a appearing in the string, which in this example is 2. Similarly,

```
var magicWord = "abracadabra";
var secondA = magicWord.indexOf("a", 1);
```

returns 3, the index of the first a to be found in the string when starting from the second character of the string.

NOTE: The indexOf() method is case-sensitive.

See Also "Clib.strchr() Method" on page 191, "Clib.strpbrk() Method" on page 200, "charAt() Method" on page 305, "lastIndexOf() Method" on page 307, and "string.replace() Method" on page 311

lastIndexOf() Method

This method finds the position of the last occurrence of a substring in a string.

Syntax *stringVar*.indexOf(*substring* [, *offset*])

Parameter	Description
substring	One or more characters to search for
offset	The position in the string at which to start searching, where 0 represents the first character

Returns The position of the last occurrence of a substring in a string variable.

Usage The *stringVar*.lastIndexOf() function searches the entire substring in a string variable. The *substring* parameter may be a single character. If *offset* is given, searching starts at the indicated position. If it is not given, searching starts at the end of the string.

For example:

```
var string = "what a string";
string.lastIndexOf("a")
```

returns the position of the last `a` appearing in the string, which in this example is 5. Similarly,

```
var magicWord = "abracadabra";
var firstB = magicWord.lastIndexOf("b", 7);
```

returns 1, the index of the first `b` to be found in the string when starting backward from the eighth character of the string.

See Also "Clib.strchr() Method" on page 191, "Clib.strpbrk() Method" on page 200, "charAt() Method" on page 305, "indexOf() Method" on page 306, and "string.replace() Method" on page 311

length Property

The length property stores an integer indicating the length of the string.

Syntax *stringVar*.length

Parameter	Description
Not applicable	

Returns Not applicable

Usage The length of a string can be obtained by using the length property. For example:

```
var string1 = "No, thank you.";
TheApplication().RaiseErrorText(string1.length);
```

displays the number 14, the number of characters in the string. Note that the index of the last character in the string is equivalent to *stringVar*.length -1, because the index begins at 0, not at 1.

Example This code fragment returns the length of a name entered by the user (including spaces).

```
var userName = "Christopher J. Smith";
TheApplication().RaiseErrorText( "Your name has " +
   userName.length + " characters.");
```

split() Method

This method splits a string into an array of strings based on the delimiters in the parameter substring.

Syntax *stringVar*.split([*delimiter*])

Parameter	Description
delimiter	The character at which the value stored in *stringVar* is to be split

Returns An array of strings, creating by splitting *stringVar* into substrings, each of which begins at an instance of the delimiter character.

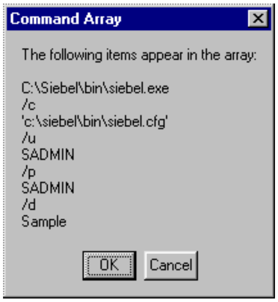
Usage This method splits a string into an array of substrings such that each substring begins at an instance of *delimiter*. The delimiter is not included in any of the strings. If *delimiter* is omitted or is an empty string (""), the method returns an array of one element, which contains the original string.

This method is the inverse of *arrayVar*.join().

Example The following example splits a typical Siebel command line into its elements by creating a separate array element at each space character. Note that the string has to be modified with escape characters to be comprehensible to Siebel eScript. Note also that the cmdLine variable must appear on a single line, which space does not permit in this volume. The result appears in the illustration following the example.

```
function Button3_Click ()
{
   var msgText = "The following items appear in the array:\n\n";
   var cmdLine = "C:\\Siebel\\bin\\siebel.exe /c
\'c:\\siebel\\bin\\siebel.cfg\' /u SADMIN /p SADMIN /d Sample"
   var cmdArray = cmdLine.split(" ");
   for (var i = 0; i < cmdArray.length; i++)
      msgText = msgText + cmdArray[i] + "\n";
   TheApplication().RaiseErrorText(msgText);
}
```

Running this code produces the following result.



```
The following items appear in the array:

C:\Siebel\bin\siebel.exe
/c
'c:\siebel\bin\siebel.cfg'
/u
SADMIN
/p
SADMIN
/d
Sample
```

See Also “join() Method” on page 99

string.replace() Method

This method searches a string using the regular expression pattern defined by *pattern*. If a match is found, it is replaced by the substring defined by *replexp*.

Syntax `string.replace(pattern, replexp)`

Parameter	Description
pattern	Regular expression pattern to find or match in string.
replexp	Replacement expression which may be a string, a string with regular expression elements, or a function.

Returns The original string with replacements according to *pattern* and *replexp*.

Usage The 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 simple string
- A string containing special regular expression replacement elements
- A function that returns a value that may be converted into a string

If any replacements are made, appropriate RegExp object static properties such as RegExp.leftContext, RegExp.rightContext, and RegExp.$n are set. These properties provide more information about the replacements.

The following table shows the special characters that may occur in a replacement expression.

Character	Description
$1, $2 … $9	The text matched by regular expression patterns inside of parentheses. For example, $1 puts the text matched in the first parenthesized group in a regular expression pattern.
$+	The text matched by the last regular expression pattern inside of the last parentheses, that is, the last group.

Character	Description
$&	The text 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.

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;

}
```

See Also “Typographic Conventions” on page 16

substring() Method

This method retrieves a section of a string.

Syntax `stringVar.substring(start[, end])`

Parameter	Description
start	An integer specifying the location of the beginning of the substring to be returned
end	An integer one greater than the location of the last character of the substring to be returned

Returns A new string, of length *end* - *start,* containing the characters that appeared in the positions from *start* to *end* - 1 of *stringVar.*

Usage This method returns a portion of *stringVar*, comprising the characters in *stringVar* at the positions *start* through *end* - 1. The character at the *end* position is not included in the returned string. If the *end* parameter is not used, *stringVar.*substring() returns the characters from *start* to the end of *stringVar.*

Example For an example, read "indexOf() Method" on page 306.

See Also "charAt() Method" on page 305, "indexOf() Method" on page 306, and "lastIndexOf() Method" on page 307

toLowerCase() Method

This method returns a copy of a string with the letters changed to lower case.

Syntax `stringVar.toLowerCase()`

Parameter	Description
Not applicable	

Returns A copy of *stringVar* in lowercase characters.

Usage This method returns a copy of *stringVar* with uppercase letters replaced by their lowercase equivalents.

Example The following code fragment assigns the value `"e. e. cummings"` to the variable poet:

```
var poet = "E. E. Cummings";
poet = poet.toLowerCase();
```

See Also "toUpperCase() Method" on page 314

toUpperCase() Method

This method returns a copy of a string with the letters changed to uppercase.

Syntax *stringVar*.toUpperCase()

Parameter	Description
Not applicable	

Returns A copy of *stringVar* in uppercase characters.

Usage This method returns a copy of *stringVar*, with lowercase letters replaced by their uppercase equivalents.

Example The following fragment accepts a filename as input and displays it in uppercase:

```
   var filename = "c:\\temp\\trace.txt";;
   TheApplication().RaiseErrorText("The filename in uppercase is
" +filename.toUpperCase());
```

See Also "toLowerCase() Method" on page 313

Index

Symbols

; (semicolon) 45
? (question mark) 67

A

absolute value 275
applet object methods 92
application object methods 94
arc cosine 275
arcsine 276
arctangent 277, 278
arguments
  number expected by the function 247
  syntax 16
arguments[] property 71
array
  constructor 98
  element order 100
  elements, sorting 182
  first index and length 261
  join() method 99
  length 258
  length property 99
  methods, list 20
  objects, described 97
  reverse() method 100
  sort() method 101
  sorting into ASCII order 101
array data type 55
ASCII, seven bit representation of a character 208
assignment operator 62

B

back quotes 304
bigEndian byte, using 112
binary large object
  data to a specified location 105
  data, reading 105
BLOB
  Blob.get() method 105
  Blob.put method 105
  Blob.size() method 107
  blobDescriptor 103
  described 103
block comments 44
blocks 45
Boolean data type 50, 53, 70, 303
Boolean variables
  converting from a value 262
  described 53
break statement 74
buffer
  bigEndian property 112
  buffer constructor 109
  comparing lengths and contents of two 177
  copying bytes from one to another 177
  cursor property 113
  data property 113
  file, writing to disk 149
  filling bytes with a character 178
  getString() method 114
  getValue() method 114
  internal data 113
  methods 112
  methods, list 21
  offset[] method 115
  properties 111
  putString() method 116

putValue() method 117
size property 119
subBuffer() method 120
toString() method 120
unicode property 121
business component object methods 122
business object object methods 128
business service object methods 129
byte-array methods, list 35

C

case-insensitivity
comparing strings 192, 198
searching strings for substrings 204
case-sensitivity
comparing two strings 198
described 42
programming guidelines 40
casting methods
list 23
when to use 58
character
alphabetic 169
alphanumeric 169
ASCII 170
characters from current file cursor 152
classification methods, list 22
control 170
decimal digit 170
first occurrence in a buffer 176
hexadecimal digit 174
last occurrence 201
lowercase alphabetic 171
next in a file stream 150
printable 171, 172
punctuation mark 173
pushing back into a file 209
seven-bit ASCII representation 208
uppercase alphabetic 174
white-space 173
writing to a specified file 157
charAt() method 305
Clib object
Clib compared to ECMAScript methods 132
data, formatting 138
file I/O functions 133
format strings 138
formatting data 138
redundant functions 131
time functions 136
Time object 135
Clib.asctime() method 141
Clib.bsearch() method 141
Clib.chdir() method 143
Clib.clearerr() method 144
Clib.clock() method 144
Clib.cosh() method 144
Clib.ctime() method 145
Clib.difftime() method 146
Clib.div() method 146
Clib.errno property 147
Clib.fclose() method 148
Clib.feof() method 148
Clib.ferror() method 149
Clib.fflush() method 149
Clib.fgetc() method 150
Clib.fgetpos() method 151
Clib.fgets() method 152
Clib.flock() method 153
Clib.fopen() method 154
Clib.fprintf() method 157
Clib.fputc() method 157
Clib.fputs() method 158
Clib.fread() method 158
Clib.freopen() method 160
Clib.frexp() method 161
Clib.fscanf() method 162
Clib.fseek() method 163
Clib.fsetpos() method 163
Clib.ftell() method 164
Clib.fwrite() method 164
Clib.getc() method 150
Clib.getcwd() method 166
Clib.getenv() method 167
Clib.gmtime() method 168

Clib.Idexp() method 175
Clib.Idiv() method 146
Clib.isalnum() method 169
Clib.isalpha() method 169
Clib.isascii() method 170
Clib.iscntrl() method 170
Clib.isdigit() method 170
Clib.isgraph() method 171
Clib.islower() method 171
Clib.isprint() method 172
Clib.ispunct() method 173
Clib.isspace() method 173
Clib.isupper() method 174
Clib.isxdigit() method 174
Clib.localtime() method 175
Clib.memchr() method 176
Clib.memcmp() method 177
Clib.memcpy() method 177
Clib.memmove() method 177
Clib.memset() method 178
Clib.mkdir() method 178
Clib.mktime() method 179
Clib.modf() method 179, 180
Clib.putc() method 157
Clib.putenv() method 181
Clib.qsort() method 182
Clib.rand() method 183
Clib.remove() method 185
Clib.rename() method 185
Clib.rewind() method 186
Clib.rmdir() method 186
Clib.rsprintf() method 187
Clib.sinh() method 188
Clib.sprintf() method 188
Clib.srand() method 189
Clib.sscanf() method 190
Clib.strchr() method 191
Clib.strcmpi() method 192
Clib.strcspn() method 192
Clib.strerror() method 194
Clib.strftime() method 195
Clib.stricmp() method 192
Clib.strncat() method 197
Clib.strncmp() method 198
Clib.strncmpi() method 198
Clib.strncpy() method 199
Clib.strnicmp() method 198
Clib.strpbrk() method 200
Clib.strrchr() method 201
Clib.strspn() method 202
Clib.strstr() method 196, 203
Clib.strstri() method 204
Clib.system() method 205
Clib.tanh() method 206
Clib.time() method 206
Clib.tmpfile() method 207, 208
Clib.toascii() method 208
Clib.ungetc() method 209
COMCreateObject() method 251
commands, passing to the command processor 205
comments 44
comparing values 64
conditional expressions 64
constants, numeric 57
continue statement 75
control character 170
conventions, typographic 16
conversion methods
  alphanumeric string to a floating-point decimal number 260, 261
  list 23
  parameter to a buffer 263
  parameter to a number 267
  parameter to a string 268
  parameter to an integer 265, 266, 269, 270
  parameter to an object 268
  value to the Boolean data type 262
copying characters between strings 199
CORBA objects
  exception handling 87
CORBA objects, binding 252
CORBACreateObject() method 252
cosine 280
cursor. *See* file cursor

D

data
- file, writing to disk 148
- formatting 137
- handling methods, list 24
- storing in a series of parameters 162
- storing in variables 158
- writing data in a specified variable to a specified file 164

data types
- array 55
- Boolean, converting value to 262
- composite 54
- decimal floats 52
- described 50
- floating-point numbers 52
- hexadecimal notation 52
- integers 51
- NaN 57
- null 56
- object 55
- octal notation 52
- primitive 51
- properties and methods 58
- special 56
- string 53, 303
- undefined 56

date
- extracted from a Time object 141
- functions, list 25
- stored in variables 195

Date object
- about 210
- Date constructor 210
- universal time functions 212

Date.fromSystem() 210
Date.fromSystem() static method 213
Date.pars() static method 226
Date.toSystem() 210
Date.toSystem() method 242
Date.UTC() static method 243
date-time value 145
decimal digit 170
decimal floats 52
decimal number, integer part 179
defined() method 255
diagnostic messages 147

directory
- changing current 143
- creating 178
- current working, path of 166
- functions, list 28
- removing 186

disk functions, list 28
division 183, 184
do...while statement 76

E

e
- base 10 logarithm 285
- base 2 logarithm 284
- number value of 281

ECMAScript 42
end of line comments 44
end-of-file flag, resetting 144

environment variable
- creating 181
- strings 167

error indicator 149

error messages
- associated with an error number 194

error status 144
error-handling methods, list 30

escape sequences
- back quotes and 304
- list 303
- removing from a string 272
- replacing special characters with 256

escape() method 256
eval() method 257
exponential function 281
expressions 45, 60

F

file
- deleting a specified 185
- functions, list 28
- input/output functions, list 29
- opening in a specified mode 154
- renaming 185
- temporary binary 207

file buffer, data 149

file cursor
- current, setting to a position 163
- locating 148
- position offset, setting 164
- position, current 113
- position, setting 163
- setting to the beginning 186

file mode characters 155

file pointers, associating with other files 160

file-control functions, list 28

floating-point numbers
- converting from alphanumeric 260
- described 52
- hyperbolic sine 188
- hyperbolic tangent 206
- mantissa and exponent as givens 175
- representations of 51

for statement 77

for...in statement 78

formatting data 137

Function objects
- creating 246
- length property 247
- return statement 247

functions
- arguments[] property 71
- described 69
- error checking 73
- passing variables to 70
- recursive 71
- scope 70
- specific location within 79

G

get method, BLOB object 105

getArrayLength() method 258

getDate() method 212

getDay() method 214

getFullYear() method 215

getHours() method 216

getMilliseconds() method 216

getMinutes() method 217

getMonth() method 217, 218

getSeconds() method 218

getTIme() method 219

getTimezoneOffset() method 220

getUTCDate() method 221

getUTCDay() method 221

getUTCFullYear() method 222

getUTCHours() method 223

getUTCMilliseconds() method 224

getUTCMinutes() method 224

getUTCMonth() method 225

getUTCSeconds() method 226

getYear() method 226

Global object
- conversion or casting functions 250
- functions 249

global variables 48

goto statement 79

Greenwich Mean Time (GMT) 212

H

hexadecimal digit 174

hexadecimal notation 52

hyperbolic cosine of x 144

hyperbolic sine 188

hyperbolic tangent 206

I

identifiers
- prohibited 47
- rules 46
- *See also* variables 47

if statement 80

indexOf() method 306
instantiated objects, syntax 16
instantiating 294
integer
  converting to a Time object 168
  described 52
  division 146
  greatest 282
  smallest 279
integer numbers
  converting from alphanumeric 261
isFinite() method 259
isNaN() method 259

J

JavaScript
  common usage 42
  and eScript 39

L

lastIndexOf() method 307
length property
  Array object 99
  Function object 247
  String object 308
line breaks in strings 44
local variables 48
locking files for multiple processes 153
logarithm
  base 10 of *e* 285
  base 2 of *e* 284
  natural 284
  number value for *e* 281
  of 10 282
  of 2 283
loops
  continue statement 75
  do...while statement 76
  for...in statement 78
  new iteration, starting 75
  repeating 88
  terminating 74

M

Math object 274
math properties, list 32
Math.abs() method 275
Math.acos() method 275
Math.asin() method 276
Math.atan() method 277
Math.atan2() method 278
Math.ceil() method 279
Math.cos() method 280
Math.E property 281
Math.exp() method 281
Math.floor() method 282
Math.LN10 property 282
Math.LN2 property 283
Math.log() method 284
Math.LOG10E property 285
Math.LOG2E property 284
Math.max() method 286
Math.min() method 286
Math.PI property 287
Math.pow() method 287
Math.random() method 288
Math.round() method 289
Math.sin() method 290
Math.sqrt() method 290
Math.SQRT1_2 property 291
Math.SQRT2 property 291
Math.tan() method 292
MAX_VALUE constant 57
MIN_VALUE constant 57
miscellaneous methods, list 37

N

NaN constant 57
NaN data type 57
NEGATIVE_INFINITY constant 57
null data type 56
number constants 57
numbers
  calculating integer exponent of 2 161
  pseudo-random 288

random 183
random, generating 189
rounding 289
numeric functions, list 31

O

object data type 55
Object object 293
object property
testing 255
undefining 271
object prototypes 295
objects
assigning functions 294
instantiated, syntax 16
looping through properties 78
templates, creating 293
octal notation 52
operating system interaction methods, list 34
operators
assignment arithmetic 62
auto-decrement 63
auto-increment 63
basic arithmetic 61
bit 64
conditional 67
conditional expressions 64
logical 64
mathematical 61
order of precedence 60
string concatenation 67
typeof 66
output
writing to a string variable 188

P

parameter
converting to a buffer 263
converting to a number 267
converting to a string 268
converting to an integer 265, 266, 269, 270
converting to an object 268
determining if it is a finite number 259
determining if it is a number 259
placing in a buffer 264
raising to a power 287
value, returning 257
parseFloat() method 260, 261
pi, number value 287
point 188
pointer, current position 151
POSITIVE_INFINITY constant 57
printing
format strings 138
processor tick count, current 144
program flow, directing 80, 83
properties, described 293
property set object methods 298
punctuation marks 173
put method, BLOB object 105

Q

question mark (?) 67
quot method 183
quotient, finding 183

R

random number generator 189
random numbers 183
recursive functions 72
rem method 184
return statement 247

S

scientific notation 53
searching in arrays 141
searching in strings
characters not among a group 202
first occurrence of a second string 203
first occurrence of a specified substring 204

group of specified characters 192
several characters 200
specified character 191
SEEK_CUR 163
SEEK_END 163
SEEK_SET 163
SElib object 300
SElib.dynamicLink() method 300
semicolon (;) 45
sequential data 97
setArrayLength() method 261
setDate() method 227
setFullYear() method 228
setHours() method 229
setMilliseconds() method 229
setMinutes() method 231
setMonth() method 231
setSeconds() method 232
setTime() method 232
setUTCDate() method 234
setUTCFullYear() method 234
setUTCHours() method 235
setUTCMilliseconds() method 236
setUTCMinutes() method 237
setUTCMonth() method 238
setUTCSeconds() method 239
setYear() method 239
Siebel eScript
basic concepts 42
and JavaScript 42
programming guidelines 40
this object reference 41
sine 290
size method, BLOB object 107
special characters 303, 304
special data types 56
split() method 309, 311
square root
of 1/2 291
of 2 291
parameter 290
statement blocks
assigning a default object 89
described 45
statements
described 45
repeating a series 77
string concatenation 67
string data type 53
string objects 303
String.fromcharCode() static method 306
strings
appending a specified number of characters 197
back-quote 304
from character codes 306
converting alphanumeric to a floating-point decimal number 260, 261
copying characters between 199
copying to lowercase 313
copying to uppercase 314
creating strings of array elements 99
declaring 304
described 53
escape sequences 303
formatted 187
formatted, writing to a file 157
length stored as an integer 308
methods, list 35
as objects 305
searching for a group of characters 192
searching for characters 191, 200, 202
searching for first occurrence of a second string 203
searching for last occurrence of a character 201
section, retrieving 312
special characters 303
specific place in 305
splitting into arrays 309
substring, first occurrence 306
substring, last occurrence 307
substrings, searching for 204
usage 303
writing to a specified file 158
substring() method 312

switch statement
  controlling the flow 74
  described 83

T

tangent 292
this object reference 294
this object reference in Siebel eScript 41
time
  difference between two times 146
  extracted from a Time object 141
  functions, list 25
  integer representation 206
  stored in variables 195
Time object
  converting 179
  described 135
ToBoolean() method 262
ToBuffer() method 263
ToBytes() method 264
toGMTString() method 240
ToInt32() method 265
ToInteger() method 266
toLocaleString() method 241
toLowerCase() method 313
ToNumber() method 267
ToObjec() method 268
ToString() method 268
toString() method 59, 241
ToUnit16() method 269
ToUnit32() method 270
toUTCString() method 242
trailing parentheses () 40
trigonometric functions, list 32
try statement 86
type conversion, automatic 57

U

undefine() method 271
undefined data type 56
unescape() method 272
Universal Coordinated Time (UTC) 212
unlocking files for multiple processes 153

V

value
  passing back to the function 247
  specifying with object prototypes 295
  undefining 271
valueOf() method 59
variables
  about 47
  array, matching 141
  Boolean 53
  compound 293
  data in, writing to a specified file 164
  declaring 40, 48
  passing by reference 54
  passing by value 51, 70
  passing to the COM object 251
  scope 48
  Siebel eScript 48
  storing data in 190
  testing 255
  undefining 271

W

web applet object methods 92
while statement 45, 88
white-space character 43, 173
with statement 89

Y

Y2K sensitivities 40, 210