Siebel eScript Language Reference

Version 8.0, Rev. A

June 2007

ORACLE®

Contents

1 What's New in This Release

What's New in Siebel eScript Language Reference, Version 8.0, Rev. A

Table 1 lists changes described in this version of the documentation to support release 8.0 of the software.

Table 1. New Product Features in Siebel eScript Language Reference, Version 8.0, Rev. A

Topic	Description
"Data Types in Siebel eScript" on page 27	All primitive and object data types can be strongly typed when you use the ST eScript engine.
"Object Data Types in Siebel eScript" on page 29	The Application object was added to the list of prebuilt object types in Siebel eScript.
"Performance Considerations in Using Object Data Types" on page 31	New topic. To improve performance, it is recommended that you use primitive data types rather than object data types when possible.
"Logical Operators and Conditional Expressions in Siebel eScript" on page 42	Corrected syntax errors in the last code example in this section.
"About Script Libraries" on page 115	Added information on using the Siebel 8.0 script libraries feature with business service objects.

What's New in Siebel eScript Language Reference, Version 8.0

Table 2 lists changes described in this version of the documentation to support release 8.0 of the software.

Table 2. New Product Features in Siebel eScript Language Reference, Version 8.0

Topic	Description
"About the Siebel eScript Script Engines" on page 16	The ST eScript Engine is enabled by default in Siebel 8.0.
"Incompatibilities Between the ST eScript and the T eScript Engines" on page 16	New information on differences between the ST eScript engine and the T eScript engine in performing the following: ■ Processing comparison operations ■ Accessing objects and arrays

Table 2. New Product Features in Siebel eScript Language Reference, Version 8.0

Topic	Description
“About the Fix and Go Feature” on page 18	The Fix and Go option makes script testing more efficient by allowing scripts to be edited and the debugging process to continue without having to recompile the program after each change.
“About Script Assist Repository Introspection” on page 18	Script Assist accesses the definitions of objects in the repository and displays this information allowing developers to choose the relevant methods and properties they want to add to their script.
“Logical Operators and Conditional Expressions in Siebel eScript” on page 42	New information on using the equality comparison operator with strongly typed variables.
“Applet Objects” on page 79	In Siebel 8.0, the applet user property CanInvokeMethod can be used instead of the PreCanInvokeMethod() event to enable and disable applet methods.
“Business Service Objects” on page 114	Business service functions can be called directly from anywhere within the scripting interface after a business service is declared by using Script Assist script libraries.
“String Object Methods and Properties in Siebel eScript” on page 302	Added information on the String search() method.

2 Siebel eScript Language Overview

This chapter provides an overview of the Siebel eScript programming language. It contains the following topics:

About Siebel eScript

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. The Siebel ST eScript engine, the default scripting engine in Siebel 8.0, supports ECMAScript Edition 4.

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

You should regard coding as a last resort. Siebel Tools provides many ways to configure your Siebel application without coding, and these methods should be exhausted before you attempt to write your own code, for the following reasons:

- Using Siebel Tools is easier than writing code.
- More important, your code may not survive an upgrade. Customizations created directly in Siebel Tools are upgraded automatically when you upgrade your Siebel application, but code is not touched, and it may need to be reviewed following an upgrade.
- Finally, declarative configuration through Siebel Tools results in better performance than implementing the same functionality through code.

For more information on implementing the Siebel scripting engine, see *Using Siebel Tools*.

About the Siebel eScript Script Engines

Starting with Oracle's Siebel Business Applications release 7.8, there are two versions of the scripting engine available to you:

- T eScript engine
- ST eScript engine

The T eScript engine is the traditional, previously available engine. The ST eScript engine provides enhancements, including strong typing of variables, and the Script Assist utility and the Fix and Go feature, which allow easier script creation and testing.

NOTE: The ST eScript Engine is the default scripting engine in Siebel 8.0.

For information on implementing the eScript engines, and on using the ST eScript engine enhancements, see *Using Siebel Tools* and the following topics in this guide:

- "About the Fix and Go Feature" on page 18
- "About Script Assist Repository Introspection" on page 18

Except for a few key differences, the ST eScript engine is backward compatible with eScript created with the T eScript engine. For information on these incompatibilities, see "Incompatibilities Between the ST eScript and the T eScript Engines" on page 16. In this document, the engines are referred to by name only in contexts requiring differentiation.

Although it is not recommended, you can, if necessary, revert back to the T eScript engine after compiling with the ST eScript engine. To do so, you must turn off the ST eScript engine in Siebel Tools, back out any script you have typed prior to compiling with the ST eScript engine, then recompile the script with the T eScript engine.

NOTE: If you choose to revert to the T eScript engine, you must contact Siebel Technical Support for assistance and guidance.

Incompatibilities Between the ST eScript and the T eScript Engines

The eScript scripting engines differ in their treatment of scripting elements in several ways. These incompatibilities are described as follows:

- **Variable data typing.** The ST eScript engine supports strong typing, or assigning a variable's data type when the variable is declared, so that the type-binding occurs at compile time. Both engines support typeless variables, whose binding occurs at run time.

  Typically, strongly typed variables provide improved performance, as compared with their typeless counterparts.

  For more information, see "Data Typing in Siebel eScript" on page 35.

- **Comparison operations.** The ST eScript engine compares object *values* when performing equality comparisons for typeless variables but it compares object *identities* when performing equality comparisons for strongly typed variables. If you are not aware of this difference, the results of comparison operations involving strongly typed variables can be misleading.

  For further information on using the equality operator with strongly typed variables, see "Logical Operators and Conditional Expressions in Siebel eScript" on page 42.

- **Accessing objects and arrays.** On the T eScript engine, accessing an array item, or accessing an object function or data, automatically creates a new object. This is not the case on the ST eScript engine; the object to be accessed must be explicitly initialized. The following script examples illustrate these differences between the eScript engines:

  - Accessing Object Functions or Data

    The following script runs correctly using the T eScript engine but fails at runtime using the ST eScript engine:

    ```
    var oArr = new Array ();
    oArr[0].m_Data =1;
    ```

    For the script to run correctly on the ST eScript engine, you must initialize the data object to be accessed, as follows:

    ```
    var oArr = new Array ();
    oArr[0] = new Object ();
    oArr[0].m_Data =1;
    ```

  - Accessing Arrays

    When accessing an array item, the following script runs correctly using the T eScript engine but not on the ST eScript engine:

    ```
    var oArr = new Array ();
    oArr[2][3].m_Data = 2;
    ```

    The script runs correctly on the ST eScript engine if you initialize the array to be accessed as follows:

    ```
    var oArr = new Array ();
    oArr[2] = new Array ();
    oArr[2][3] = new Object ();
    oArr[2][3].m_Data = 2;
    ```

- **Implicit variable type conversion.** There are differences in how implicit type conversions are performed with strongly typed variables and with typeless variables. Implicit conversions happen in mixed type contexts, such as when a string variable is assigned the value of a numerical variable.

  For more information, see "Implicit Type Conversion in Siebel eScript" on page 35.

- **Methods.** The engines restrict the parameters passed with the global.setArrayLength() method differently.

  For more information, see "setArrayLength() Method" on page 241.

  NOTE: If a method (or group of methods) is supported by one engine, and not supported by the other, then the restriction is stated in the documentation for the method (or at a level that covers the group).

- **Properties.** The ST eScript engine does not support the following static properties of the RegExp object:

  - RegExp.$n (including '$_' and '$&')
  - RegExp.input
  - RegExp.lastMatch
  - RegExp.lastParen
  - RegExp.leftContext
  - RegExp.rightContext

  Instead, you must modify your script to use equivalent functions on the target object itself.

- **Commands.** The ST eScript engine does not support #define or #if, preprocessor alternatives that are used at compile time only. An alternative to using #define is to use a var declaration.

  For example, change

  ```
  #define MY_DEFINE "abc"
  ```

  to

  ```
  var MY_DEFINE = "abc";
  ```

About the Fix and Go Feature

In Siebel 8.0, you can enable the ST eScript engine Fix and Go option to facilitate script testing and debugging. When Fix and Go is enabled, you can edit a script and test the change then restart the debugger without having to recompile the entire program. This makes the script testing and debugging process much quicker.

For further information on the ST eScript engine Fix and Go option, see *Using Siebel Tools*.

About Script Assist Repository Introspection

The Script Assist utility, provided by the ST eScript Engine, provides several features that facilitate the scripting process. The repository introspection feature in Siebel 8.0 allows Script Assist to access object definitions in the repository for objects in your script and then display this information in the Script Assist window. You can then select the object methods, property names, user-defined functions or fields available for the object and select the one that you want to add to your script. This helps prevent scripting errors and makes the scripting process simpler.

The ST eScript engine provides repository introspection for the following objects and methods:

- Applet objects
  - BusComp Method
  - BusObject Method
- Application objects
  - ActiveBusObject
  - ActiveViewName
  - GetBusObject
  - GetService
- Business Component objects
  - ActivateField
  - ActivateMultipleFields
  - Associate
  - BusObject
  - DeActivateFields
  - GetAssocBusComp
  - GetFieldValue
  - GetFormattedFieldValue
  - GetMultipleFieldValues
  - GetMvgBusComp
  - GetPickListBusComp
  - GetViewMode
  - ParentBusComp
  - SetFieldValue
  - SetFormattedFieldValue
  - SetMultipleFieldValues
- Business Object objects
  - GetBusComp
- Business Service objects
  - InvokeMethod

Script Assist can determine the object referred to by the *this* object reference key word in a script and displays the appropriate fields for the object. It also shows all the required and optional parameters for Siebel methods, global methods and functions, and user-defined functions, and the methods available for InvokeMethod calls.

To use the repository introspection feature

1. Access Script Assist from the script editor by pressing CTRL+SPACE.
2. In your script, type the name of an object or method, followed by an open parenthesis character.

   The method, property names or fields available for the item are displayed.
3. Select the item you want to add to your script.

For additional information on Script Assist functions and on accessing Script Assist, see *Using Siebel Tools*.

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 declaration 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. Local declaration prevents the accumulation of unwanted values.

Consider case sensitivity. 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 03 to 05 may find that they have incorrectly retrieved orders for the years 1903–1905 (probably an empty list), instead of for 2003–2005, as they had 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().

Use 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 alternative 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.

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 and is interpreted by the Siebel Object Manager at run time. 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. They include:

- "Case Sensitivity in Siebel eScript" on page 21
- "White-Space Characters in Siebel eScript" on page 22
- "Comments in Siebel eScript" on page 23
- "Expressions, Statements, and Blocks in Siebel eScript" on page 24
- "Identifiers in Siebel eScript" on page 24
- "Variables in Siebel eScript" on page 26

Case Sensitivity in Siebel eScript

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 in Siebel eScript

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, see "String Concatenation Operator in Siebel eScript" on page 45.

Special Characters in Siebel eScript

Characters such as the double quote mark ("), the single quote mark ('), the hard return, the semi-colon (;), and the ampersand (&) have special meanings within JavaScript and eScript. But sometimes you want to use them for their traditional values, to have quotation marks appear around a phrase on the screen, to add a hard return to your text file to make it more readable or to specify a file system path. You can escape the character, that is, you can tell JavaScript to skip over it by preceding the character with a back slash.

The backslash (\) character is JavaScript/eScript's escape character. The backslashes in SVB and JavaScript/eScript are used differently. Two backslashes are needed in JavaScript/eScript. The reason for this is that the JavaScript/eScript interpreter sees a single backslash as indicating that the very next character is a character to be "escaped" (to use it literal meaning). For more information, see "Escape Sequences for Characters in Siebel eScript" on page 301.

Comments in Siebel eScript

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 in Siebel eScript

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 in Siebel eScript

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.

eScript 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
What?
=Total()
(Minus)()
Add Both Figures()
```

Prohibited Identifiers in Siebel eScript

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

break	export	super
case	extends	switch
catch	false	this
class	finally	throw
const	for	true
continue	function	try
debugger	if	typeof
default	import	while

delete	in	with
do	new	var
else	null	void
enum	return	

Variables in Siebel eScript

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 require 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. Typically you declare all global variables in a general declarations section. 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 a parameter 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 in Siebel eScript

Data types in Siebel eScript can be classified into primitive data types and object data types. All primitive and object data types can be either:

- **Strongly typed.** The data type is specified when the variable is declared.
- **Typeless.** The data type is determined by the Siebel eScript interpreter at run time.

For further information on strongly typed and typeless variables, see "Data Typing in Siebel eScript" on page 35.

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 happyToo = happyVariable;
```

The following topics provide information on the Siebel eScript data types:

- "Primitive Data Types in Siebel eScript" on page 28
- "Object Data Types in Siebel eScript" on page 29
- "Performance Considerations in Using Object Data Types" on page 31
- "Numbers in Siebel eScript" on page 32
- "Data Typing in Siebel eScript" on page 35
- "Implicit Type Conversion in Siebel eScript" on page 35

Primitive Data Types in Siebel eScript

A *primitive data type* is the set of all possible values of a primitive value. A variable that is of a primitive data type is simply a value. Unlike an object data type, it can have no other properties or functions that are part of its definition.

The primitive data types are:

- **chars.** This primitive type is used for defining and manipulating strings. By convention, a chars value is a sequence of alphanumeric characters. However, it is technically any sequence of 16-bit unsigned integers.
- **float.** This primitive type is used for defining and manipulating floating point numbers.

  NOTE: Integer is not an eScript data type. You can use a variable of type float. Some routines that expect integer arguments do an internal conversion of a float variable.

- **bool.** This primitive type is used for defining and manipulating Boolean objects. A bool value is either true or false.
- **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.

  Following is code that will test whether a variable is undefined:

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

NOTE: When the chars, float, or bool primitive data types are used to declare variables, they must be used as all lowercase.

Object Data Types in Siebel eScript

The ECMAScript standard defines an object as “a member of the type Object. It is an unordered collection of properties, each of which contains a primitive value, object, or function. A function stored in a property of an object is called a method.”

Siebel eScript does not implement a proper class hierarchy. Instead, objects are instantiated as type Object or are instantiated from objects descended from objects of type Object. These instantiated objects act as new object types themselves, from which other objects may be instantiated. Each object has an implicit constructor function that is implemented through the *new* command.

Properties can be added dynamically to any object. An object inherits all the properties of the objects in its ancestral chain.

The Object object type is the generic object type. By declaring a variable of type Object, the variable’s structure is starting new, in a sense, in that it does not inherit properties from any objects descended from the Object type.

Object types that are built into the scripting engine are:

- **String.** A String object is created by using the String constructor in a *new* expression. The string’s value, a chars value, becomes an implicit property of the String object.

  A string is written using a pair of either double or single quotation marks, for example:

  ```
  "I am a string"
  'so am I'
  "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 implicitly converts strings to numbers and numbers to strings, depending on the context. For more information about implicit type conversions, see “Implicit Type Conversion in Siebel eScript” on page 35.

- **Boolean.** A Boolean object is created by using the Boolean constructor in a *new* expression. The Boolean object’s value, a bool value (true or false), is an implicit property of the Boolean object.

  Because Siebel eScript implicitly 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.

- **Number.** A Number object is created by using the Number constructor in a *new* expression. The number’s value, a value of primitive type float, becomes an implicit property of the Number object.

  For more information on numbers in eScript, see “Numbers in Siebel eScript” on page 32.

- **Array.** An array is a series of data stored in a variable. Each datum is associated with an index number or string. The following fragments illustrate the storage of the data in an array variable:

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

The array variable *Test* contains three strings. The array variable can be used as one unit, and the strings can also be accessed individually by appending the bracketed index of the element after the array name.

Arrays and objects in general use grouping similarly. Arrays are objects in Siebel eScript, but they have different notations for accessing properties than other objects. While arrays use indexes, 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 83.

- **Null.** The null object is literally a null pointer. The null object type 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*. The keyword *null* enables comparisons to the null object.

  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.

Table 3 lists the other prebuilt object types.

Table 3. Other Prebuilt Object Types in Siebel eScript

Object	Comment
Application	For more information, see "Application Objects" on page 81.
BLOB	For more information, see "BLOB Objects" on page 91.
BlobDescriptor	For more information, see "blobDescriptor Objects" on page 92.
Buffer	For more information, see "Buffer Objects in Siebel eScript" on page 98.
BusComp	For more information, see "Business Component Objects" on page 109.
BusObject	For more information, see "Business Object Objects" on page 114.
CfgItem	This is a Siebel Product Configurator object.
Clib	For more information, see "Clib Objects" on page 117.
CTIData	For more information, see *Siebel Communications Server Administration Guide*.
CTIService	For more information, see *Siebel Communications Server Administration Guide*.
Date	For more information, see "Date Objects" on page 201.
Exception	For more information, see "Exception Objects" on page 235.

Table 3. Other Prebuilt Object Types in Siebel eScript

Object	Comment
File	For more information, see "Clib.fopen() Method" on page 140.
Math	For more information, see "Math Objects" on page 262.
PropertySet	For more information, see *Siebel Object Interfaces Reference*.
RegExp	For more information, see "RegExp Objects" on page 286 and ECMAScript specifications.
SELib	For more information, see "SElib Objects" on page 294.
Service	For more information, see "Business Service Objects" on page 114.
WebApplet	For more information, see "Applet Objects" on page 79.

Performance Considerations in Using Object Data Types

There is a performance overhead associated with using object data types when compared with using primitive data types and, as a result, primitive data types typically provide superior execution performance. For this reason, it is recommended that you use an object data type only if you need to make use of properties specific to that object type. Otherwise, if an equivalent primitive type exists for the object type, use the primitive data type instead.

Table 4 shows the object data types for which equivalent primitive data types exist.

Table 4. Object Data Type to Primitive Data Type Mappings

Instead of using this object...	You can use this primitive
String	chars
Number	float
Boolean	bool

For example, use the chars primitive data type instead of a String object type in your scripts when you do not need to use String-specific object or conversion methods.

Complex Objects in Siebel eScript

Variables can be passed as parameters to subroutines and functions in two ways:

- **By value.** A variable passed by value retains its value prior to being passed, even though the passed value may change during processing within the subroutine or function. The following fragment illustrates:

```
var a = 5;
var b = ReturnValue(a);

function ReturnValue(c)
{
   c = 2 * c;
   return c ;
}
```

After this script runs, a = 5 and b = 10. However, c has value only during the execution of the function ReturnValue, but not after the function has finished execution. Although a was passed as a parameter to the function, and that value is manipulated as local variable c, a retains the value it had prior to being passed.

- **By reference.** Complex objects are objects that are passed by reference. When a variable is passed by reference, a reference to the object's data is passed. A variable's value may be changed by the subroutine or function to which it is passed, as illustrated by the following fragment:

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

function ReturnName(CurObj)
{
   var c = CurObj.name;
   CurObj.name = “Vijay”;
   return c
}
```

When `AnObj` is passed to the function `ReturnName()`, it is passed by reference. `CurObj` receives a reference to the object, but it 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. During the course of the function executing, `CurObj.name` is changed to “Vijay” within the function, so `AnObj.name` also becomes “Vijay.”

Each method determines whether parameters are passed to it by value or by reference. For the large majority of methods, parameters are passed by value.

Numbers in Siebel eScript

This topic describes the various notations for numeric literals.

NOTE: The notations provided in this section are not data types and should not be used as data types in declarations for strongly typed variables.

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

Integer

Integers are positive and negative whole numbers and zero. 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.

NOTE: A variable cannot be strongly typed as an integer. You can use the primitive type float, and its value can be used as an integer.

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.

NOTE: Floating-point numbers are often referred to as floats. Do not confuse the familiar connotation of float with the eScript float primitive data type.

CAUTION: The assignment of a floating-point number to a variable may cause a loss in precision due to a limit in memory for decimal-to-binary conversion. Numbers that may be stored with a small precision error are decimal numbers that do not convert to a finite binary representation. For example, the statement var x = 142871.45 may result in x being stored as 142871.450000000001. These small precision errors will likely have little effect on precision of subsequent calculations, depending on their context. However, a number's representation may be unexpectedly too large for the field in which it displays, resulting in the error message "Value too long for field %1 (maximum size %2)."

To prevent floating-point precision errors, use the toFixed() Method at appropriate points in calculations or when assigning variable values. For example, use x.toFixed(2) in calculations instead of using variable x as declared above.

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

Scientific notation is useful in expressing very large and very 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 = 5.321e31 + 9.333e-2;
```

The decimal equivalents are:

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

NaN

NaN means "not a number," and NaN is an abbreviation for the phrase. NaN is not a data type, but is instead a value. 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 in Siebel eScript

Several numeric constants, as shown in Table 5, can be accessed as properties of the Number object, though they do not have a literal representation.

Table 5. Numeric Constants in Siebel eScript

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

Table 5. Numeric Constants in Siebel eScript

Constant	Value	Description
`Number.POSITIVE_INFINITY`	Infinity	Number greater than `MAX_VALUE`
`Number.NEGATIVE_INFINITY`	-Infinity	Number less than `MIN_VALUE`

Data Typing in Siebel eScript

You can specify a variable's data type in two ways:

- **Typed (or strongly typed) variables.** You specify the data type in the variable's declaration by appending a colon ":" and the data type after the variable name. For example:

  ```
  var a : Date = new Date ();
  var BO:BusObject;
  var BC:BusComp;
  ```

  Binding and type checking of strongly typed variables occurs at compile time. Typically, a strongly typed variable provides improved execution over its typeless counterpart. It also enables the compilation warning for incorrect methods and properties.

  All of the primitive and object data types can be strongly typed. For a list of these data types, see "Primitive Data Types in Siebel eScript" on page 28 and "Object Data Types in Siebel eScript" on page 29.

  NOTE: To strongly type variables, you must implement the ST eScript engine. For information on implementing the ST eScript engine, see *Using Siebel Tools*.

  For information on implicit type conversions, see "Implicit Type Conversion in Siebel eScript" on page 35.

- **Typeless variables.** You do not specify the data type in the variable's declaration. For example:

  ```
  var count = 0;
  var a = new Date ();
  var BO = new BusObject;
  ```

  At run time, the type is determined by the Siebel eScript interpreter when the variable is first used. The type remains until a later assignment changes the type implicitly. In the preceding examples, the assigning of the value zero types variable *count* as integer. Similarly, variable *a* is of type Date and *BO* is of type BusObject.

Implicit Type Conversion in Siebel eScript

Siebel eScript performs implicit data type conversion in many mixed-type contexts. However, to make sure that your code performs conversions as you expect it to, you should use conversion functions that are provided for that purpose, and you should test your code prior to putting it into production.

For more information on conversion methods, see "Conversion Methods" on page 242.

The rules governing run-time conversion of data types vary and depend on:

- Whether the type mismatch is in the context of a value assignment or the result of concatenation
- Whether the variables involved are typeless or strongly typed

Implicit Type Conversion Resulting from Concatenation in eScript

Data type conversion of typeless variables occurs implicitly 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 or a number to a string 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 implicit conversions:

```
s = "dog" + "house"   // s = "doghouse", two strings are concatenated.
t = "dog" + 4         // t= "dog4", a number is converted to a string
u = 4 + "4"           // u = "44", a number is converted to a string
v = 4 + 4             // v = 8, two numbers are added
w = 23 - "17"          // w = 6, a string is converted to a number
```

To make sure that type conversions are performed when doing addition, subtraction, and other arithmetic, use conversion methods. The following example uses a conversion method to transform string input to numeric to perform arithmetic:

```
var n = "55";
var d = "11";
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 convert data to specific types. Some of these are not part of the ECMAScript standard.

NOTE: There are circumstances under which conversion is not performed implicitly. If you encounter such a circumstance, you must use one of the conversion functions to get the desired result. For an explanation of conversion functions, see "Conversion Methods" on page 242.

Implicit Type Conversion Resulting from Assignment in eScript

Implicit type conversion resulting from assignment differ for typeless and strongly typed variables.

- **Typeless variables.** Conversion occurs implicitly during assignment involving typeless variables only. For example, the following assignments result in variable a assuming the String type.

  ```
  var a = 7.2;
  var b = “seven point 2”
  a = b;
  ```

- **Strongly Typed variables.** Table 6 provides a mapping of types and the results of implicit type conversion during assignment of mixed types. Interpret the table as follows:
    - For the assignment a = b, types for variable a are in the left column, and types for variable b are in the top row. Thus, the assignment attempts to convert a's data type to that of b.
    - For a given data type for each of a and b, the intersection cell in the table specifies whether a's type is implicitly converted to that of b.
        - "Y" indicates that the implicit conversion occurs.
        - "w" indicates that a message may display at compile time warning that the conversion may not occur. Whether a conversion occurs and whether a warning displays depends on the properties of the variables involved in the assignment.
        - "err" indicates that a compilation error occurs.
        - "NA" indicates that there is no conversion needed. Typically, conversion is not required when a variable of Object (a generic object) is converted to a specialized object type.
        - "-" indicates that a and b are of the same type.
    - "value" denotes a typeless variable. Thus, the row and column with "value" headings specify conversions when strongly typed variables are assigned to typeless variables, and vice-versa.
    - "*" denotes objects other than the eScript built-in objects Object, String, Number, and Boolean. These other objects include prebuilt objects and user-defined objects.

Table 6. Implicit Type Conversion of Strongly Typed Variables that Are Assigned

	= b								
a	**value**	**chars**	**bool**	**float**	**Object**	**String**	**Number**	**Boolean**	*****
value	-	Y	Y	Y	Y	Y	Y	Y	Y
chars	Y	-	Y	Y	Y	Y	Y	Y	Y, w[1]
bool	Y	Y	-	Y	Y	Y	Y	Y	Y
float	Y	Y, w	Y	-	Y, w	Y, w	Y	Y	Y, w
Object	Y	err	err	err	-	NA	NA	NA	NA
String	Y	Y	err	err	err	-	err	err	err
Number	Y	err	err	Y	err	err	-	err	err
Boolean	Y	err	Y	err	err	err	err	-	err
*	Y	err	err	err	err	err	err	err	-

1. toString

Properties and Methods of Common Data Types in Siebel eScript

Common 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 these common data types are most often used internally by the Siebel eScript interpreter, but you may use them if you choose. For example, if you have a numeric variable called number and you want to convert it to a string, you can use the .toString() method, as illustrated in the following fragment:

```
var 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. Value is an implicit property of Number and Boolean objects.

valueOf()

This method returns the value of a variable. Value is an implicit property of objects, including Number, String, and Boolean objects.

Expressions in Siebel eScript

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 in Siebel eScript

Operators act on literal and variable values in expressions to generate calculated values. The following topics provide information on various categories of operators.

- "Mathematical Operators in Siebel eScript" on page 39
- "Bit Operators in Siebel eScript" on page 41
- "Logical Operators and Conditional Expressions in Siebel eScript" on page 42
- "Typeof Operator in Siebel eScript" on page 45
- "Conditional Operator in Siebel eScript" on page 45
- "String Concatenation Operator in Siebel eScript" on page 45

Mathematical Operators in Siebel eScript

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. They are described in Table 7.

Table 7. Basic Arithmetic Operators in Siebel eScript

Operator	Purpose	Description
=	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 3.333..., (10 / 3)
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. Table 8 lists these operators, their purposes, and descriptions.

Table 8. Basic Arithmetic Operators in Siebel eScript

Operator	Purpose	Description
=	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 3.333...(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 they are used after, the variable is altered after it is used in the statement.

The lines in Table 9 demonstrate prefix and postfix operations.

Table 9. Auto-Increment and Auto-Decrement Operators in Siebel eScript

Example	Results	Description
`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 in Siebel eScript

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. Table 10 lists these bit operators, their descriptions, and provides examples.

Bit operators available in Siebel eScript are:

Table 10. Bit Operators in Siebel eScript

Operator	Description	Example
<<	Shift left	`i = i << 2`
<<=	Assignment shift left	`i <<= 2`
>>	Signed shift right	`i = i >> 2`

Table 10. Bit Operators in Siebel eScript

Operator	Description	Example
>>=	Assignment signed shift right	`i >>= 2`
>>>	Unsigned shift right	`i = i >>> 2`
>>>=	Assignment unsigned shift right	`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 in Siebel eScript

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 described in Table 11.

Table 11. Logical Operators in Siebel eScript

Operator	Purpose	Description
!	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.

Table 11. Logical Operators in Siebel eScript

Operator	Purpose	Description
==	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 simple guessing 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, either 1, 2, or 3
target_number = 2;
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. Using the assignment operator incorrectly is a common mistake, even among experienced programmers.

Using the Equality Operator with Strongly Typed Variables

The ST eScript engine compares object *values* when performing equality comparisons for typeless variables but it compares object *identities* when performing equality comparisons for strongly typed variables.

The following script examples illustrate the different results produced from eScript comparison operations using the equality operator.

The result of the comparison in the following script, which involves strongly typed variables, is always `"not equal"` because object identities are being compared, not object values:

```
function foo ()
{
  var oStr1 : String = new String ("aa");
  var oStr2 : String = new String ("aa");
  if (oStr1 == oStr2)
     TheApplication ().RaiseErrorText ("equal");
  else
     TheApplication ().RaiseErrorText ("not equal");

}
```

However, the result of the comparison in the following script is also always `"not equal"`, even though the variables are not strongly typed. This is because the String is an object, and object comparisons are used in the if statement.

```
function foo ()
{
  var oStr1 = new String ("aa");
  var oStr2 = new String ("aa");
  if (oStr1 == oStr2)
     TheApplication ().RaiseErrorText ("equal");
  else
     TheApplication ().RaiseErrorText ("no equal");
}
```

To ensure that the *values* of the variables are compared in an equality comparison operation, you can either:

- Use the valueOf () method, as shown in the following script:

```
function foo ()
{
  var oStr1 = new String ("aa");
  var oStr2 = new String ("aa");
  if (oStr1.valueOf () == oStr2.valueOf ())
     TheApplication ().RaiseErrorText ("equal");
  else
     TheApplication ().RaiseErrorText ("no equal");
}
```

- Use primitive data type variables, as shown in the following script:

```
function foo ()
{
  var oStr1 : chars = "aa"
```

```
      var oStr2 : chars = "aa";
      if (oStr1 == oStr2)
         TheApplication ().RaiseErrorText ("equal");
      else
         TheApplication ().RaiseErrorText ("no equal");
   }
```

Typeof Operator in Siebel eScript

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 in Siebel eScript

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, 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 content of the variable.

String Concatenation Operator in Siebel eScript

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 in Siebel eScript

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.

After 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 and are described in this manual. These functions are internal to the interpreter and may be used at any time:

- "Function Scope in Siebel eScript" on page 46
- "Passing Variables to Functions in Siebel eScript" on page 47
- "The Function Arguments[] Property in Siebel eScript" on page 47
- "Function Recursion in Siebel eScript" on page 48
- "Error Checking for Functions in Siebel eScript" on page 48

Function Scope in Siebel eScript

Functions are global in scope and can be called from anywhere in a script within the object in which it has been declared. 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 parameters 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 in Siebel eScript

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, see "return Statement" on page 237.

The Function Arguments[] Property in Siebel eScript

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 arguments. 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 in Siebel eScript

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 in Siebel eScript

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 Clib.fopen() method is called and is unable to open a file, then the method returns null.

If you then try to read from or write to the file that is assumed to be open, you receive errors. To prevent these errors, check whether Clib.fopen() returns 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("Error with fopen as returned null " +
   "in the following object: " + this.Name() + " " + e.toString() + e.errText());
}
```

You may abort a script in such a case, and the error text indicates why the script failed. See "Clib Objects" on page 117.

Siebel eScript Statements

This section describes statements your program uses to make decisions and to direct the flow based on those decisions:

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

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, see "switch Statement" on page 57.

See Also

"do...while Statement" on page 51
"for Statement" on page 52
"if Statement" on page 56
"while Statement" on page 61

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

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.

Example

The following example writes the numbers 1 through 6 and 8 through 10, followed by the string ".Test". The use of the continue statement after "if (i==7)" prevents the write statement for 7, but keeps executing the loop.

```
var i = 0;
while (i < 10)
{
   i++;
   if (i==7)
      continue;
   document.write(i + ".Test");
}
```

See Also

"do...while Statement" on page 51
"for Statement" on page 52
"goto Statement" on page 55
"while Statement" on page 61

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

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 one time 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 52
"while Statement" on page 61

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

Usage

The counter variable must be declared with var if it has not already been declared. Even though it is declared in the for statement, its scope is local to the whole function that contains the for 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 in ways other than being incremented on each pass through the loop. Changing the counter in other ways makes your script difficult to maintain and debug.

A for statement can control multiple counters in a loop. 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, see "eval() Method" on page 244.

See Also

"do...while Statement" on page 51
"while Statement" on page 61

for...in Statement

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

NOTE: The for...in statement can be used with associative arrays, which are arrays that use strings as index elements. The for...in statement is not for use with nonassociative arrays. For more information, see "Associative Arrays in Siebel eScript" on page 85.

Syntax

```
for (LoopVar in object)
{
   statement_block;
}
```

Placeholder	Description
object	A previously defined associative array or object
LoopVar	A variable that iterates over every element in the associative array or property of the object

Usage

An object must have at least one defined property or it cannot be used in a for...in statement. Associative arrays must have at least one defined element.

The statement block executes one time for every element in the associative array or property of the object. For each iteration of the loop, the variable *LoopVar* contains the name of one of the elements of the array or the name of a property of the object and may be accessed with a statement of the form `array_name[LoopVar]` or *`object`*`[`*`LoopVar`*`]`.

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.

```
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);
}
```

Running this code produces the following results:

```
   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.
```

For an example of the for...in statement used with an associative array, see "Associative Arrays in Siebel eScript" on page 85.

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

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

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 less than 10.");
}
```

The brackets are not required if only a single statement is to be executed if *condition* is true. However, 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 the following example, if only one action is to be taken in either circumstance:

```
if ( i < 10 )
   TheApplication().RaiseErrorText("i is less than 10.");
else
   TheApplication().RaiseErrorText("i is not less 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 )
{
   i += 10;
   TheApplication().RaiseErrorText ("Original i was less than 10, and has now been
   incremented by 10.");
}
else
{
   i -= 5;
   TheApplication().RaiseErrorText ("Original i was at least 10, and has now been
   decremented by 5.");
}
```

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, see "setTime() Method" on page 220.

```
if ( i < 10 )
{
      TheApplication().RaiseErrorText("i is 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 57

switch Statement

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

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
valuen	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

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 executes.

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 executes 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 56

throw Statement

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

Syntax

throw *exception*

Parameter	Description
exception	An object in a named error class

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 example, 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 60

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

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 59).

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.

See Also

"throw Statement" on page 59

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

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

Usage

The with statement is used to save time when working with objects. It prefixes 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");
```

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

The portion in the with block is equivalent to:

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

3 Quick Reference: Methods and Properties in Siebel eScript

This chapter describes the Siebel eScript functions, methods, and properties by functional group, rather than by object. It includes the following topics:

NOTE: In this chapter, properties can be distinguished from methods by the fact that they do not end with a pair of parentheses.

Array Methods and Properties in Siebel eScript

Table 12 provides a list of array methods and properties.

Table 12. Array Methods in Siebel eScript

Method or Property	Purpose
getArrayLength() Method	Determines size of an array
Array join() Method	Creates a string from array elements
Array length Property	Returns the length of an array
Array pop() Method	Returns the last element of an array, then removes that element from the array
Array push() Method	Appends new elements to the end of an array.

Table 12. Array Methods in Siebel eScript

Method or Property	Purpose
Array reverse() Method	Reverses the order of elements of an array
setArrayLength() Method	Sets the size of an array
Array sort() Method	Sorts array elements
Array splice() Method	Splices new elements into an array
Clib.bsearch() Method	Does a binary search for a member of a sorted array
Clib.qsort() Method	Sorts an array; may use comparison function

Buffer Methods and Properties in Siebel eScript

Table 13 provides a list of buffer methods.

Table 13. Buffer Methods in Siebel eScript

Method or Property	Purpose
bigEndian Property	Stores a Boolean flag for bigEndian byte ordering
cursor Property	Stores the current position of the buffer cursor
data Property	Refers to the internal data of a buffer
getString() Method	Returns a string starting from the current cursor position
getValue() Method	Returns a value from a specified position
offset[] Method	Provides array-style access to individual bytes in the buffer
putString() Method	Puts a string into a buffer
putValue() Method	Puts a specified value into a buffer
subBuffer() Method	Returns a section of a buffer
SElib.pointer() Method	Gets the address in memory of a Buffer variable
subBuffer() Method	Stores the size of a Buffer object
toString() Method	Returns a string equivalent of the current state of a buffer
unicode Property	Stores a Boolean flag that specifies whether to use Unicode strings when calling getString() and putString()

Character Classification Methods in Siebel eScript

Table 14 provides a list of character classification methods.

Table 14. Character Classification Methods in Siebel eScript

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

Conversion Methods in Siebel eScript

Table 15 provides a list of conversion methods.

Table 15. Conversion Methods in Siebel eScript

Method	Purpose
escape() Method	Escapes special characters in a string
eval() Method	Converts an expression to its value
parseFloat() Method	Converts a string to a float
parseInt() Method	Converts a string to an integer
ToBoolean() Method	Converts a value to a Boolean
ToBuffer() Method	Converts a value to a buffer
ToBytes() Method	Converts a value to a buffer (raw transfer)
toExponential() Method	Converts a number to exponential notation

Table 15. Conversion Methods in Siebel eScript

Method	Purpose
toFixed() Method	Converts a number to a specific number of decimal places
ToInteger() Method	Converts a value to an integer
ToNumber() Method	Converts a value to a number
ToObject() Method	Converts a value to an object
toPrecision() Method	Converts a number to a specific number of significant digits
ToString() Method	Converts a value to a string
ToUint16() Method	Converts a value to an unsigned integer
ToUint32() Method	Converts a value to an unsigned large integer
unescape(string) Method	Removes escape sequences in a string

Data Handling Methods in Siebel eScript

Table 16 provides a list of data handling methods.

Table 16. Data Handling Methods in Siebel eScript

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

Date and Time Methods in Siebel eScript

Table 17 provides a list of date and time methods.

Table 17. Date and Time Methods in Siebel eScript

Method	Purpose
Clib.asctime() Method	Converts a date-time to an ASCII string
Clib.clock() Method	Gets the processor time
Clib.ctime() Method	Converts a date-time to an ASCII string
Clib.difftime() Method	Computes the difference between two times
Clib.gmtime() Method	Converts a date-time to GMT
Clib.localtime() Method	Converts a date-time to a structure
Clib.mktime() Method	Converts a time structure to calendar time
Clib.strftime() Method	Writes a formatted date-time to a string
Clib.time() Method	Gets the current time
Date.fromSystem() Static Method	Converts system time to Date object time
Date.parse() Static Method	Converts a Date string to a Date object
Date.toSystem() Method	Converts a Date object to a system time
Date.UTC() Static Method	Returns the date-time, in milliseconds, from January 1, 1970 of its parameters
getDate() Method	Returns the day of the month
getDay() Method	Returns the day of the week
getFullYear() Method	Returns the year as a four-digit number
getHours() Method	Returns the hour
getMilliseconds() Method	Returns the millisecond
getMinutes() Method	Returns the minute
getMonth() Method	Returns the month
getSeconds() Method	Returns the second
getTime() Method	Returns the date-time, in milliseconds, of a Date object
getTimezoneOffset() Method	Returns the difference, in minutes, from GMT
getUTCDate() Method	Returns the UTC day of the month
getUTCDay() Method	Returns the UTC day of the week
getUTCFullYear() Method	Returns the UTC year as a four-digit number
getUTCHours() Method	Returns the UTC hour
getUTCMilliseconds() Method	Returns the UTC millisecond

Table 17. Date and Time Methods in Siebel eScript

Method	Purpose
getUTCMinutes() Method	Returns the UTC minute
getUTCMonth() Method	Returns the UTC month
getUTCSeconds() Method	Returns the UTC second
getYear() Method	Returns the year as a two-digit number
setDate() Method	Sets the day of the month
setFullYear() Method	Sets the year as a four-digit number
setHours() Method	Sets the hour
setMilliseconds() Method	Sets the millisecond
setMinutes() Method	Sets the minute
setMonth() Method	Sets the month
setSeconds() Method	Sets the second
setTime() Method	Sets the date-time in a Date object, in milliseconds
setUTCDate() Method	Sets the UTC day of the month
setUTCFullYear() Method	Sets the UTC year as a four-digit number
setUTCHours() Method	Sets the UTC hour
setUTCMilliseconds() Method	Sets the UTC millisecond
setUTCMinutes() Method	Sets the UTC minute
setUTCMonth() Method	Sets the UTC month
setUTCSeconds() Method	Sets the UTC second
setYear() Method	Sets the year as a two-digit number
toGMTString() Method	Converts a Date object to a string
toLocaleString() Method and toString() Method	Returns a string representing the date and time of a Date object based on the time zone of the computer running the script
toUTCString() Method	Returns a string that represents the UTC date

Disk and File Methods in Siebel eScript

Siebel eScript provides the following disk and file methods:

- "Disk and Directory Methods in Siebel eScript" on page 71
- "File-Control Methods in Siebel eScript" on page 71
- "File-Manipulation Methods in Siebel eScript" on page 72

Disk and Directory Methods in Siebel eScript

Table 18 provides a list of disk and directory methods.

Table 18. Disk and Directory Methods in Siebel eScript

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

Backslashes (\) can be interpreted as escape characters. When forming Windows path names, double each back slash to prevent this interpretation. For example, to change the working directory to C:\Applications\Myfolder, use the following command:

```
Clib.chdir(“C:\\Applications\\Myfolder”);
```

Similarly, when using UNC paths to access a computer on your network, use four backslashes (\\\\) before the computer name.

```
Clib.system("copy \\\\server01\\share\\SR.txt D:\\SR.txt ");
```

File-Control Methods in Siebel eScript

Table 19 provides a list of file-control methods.

Table 19. File-Control Methods in Siebel eScript

Method	Purpose
Clib.fclose() Method	Closes an open file
Clib.fopen() Method	Opens a file
Clib.freopen() Method	Assigns a new file spec to a file handle
Clib.remove() Method	Deletes a file
Clib.rename() Method	Renames a file
Clib.tmpfile() Method	Creates a temporary binary file
Clib.tmpnam() Method	Gets a temporary filename

File-Manipulation Methods in Siebel eScript

Table 20 provides a list of file-manipulation methods.

Table 20. File-Manipulation Methods in Siebel eScript

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

Error Handling Methods in Siebel eScript

Table 21 provides a list of error handling methods.

Table 21. Error Handling Methods in Siebel eScript

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

Table 21. Error Handling Methods in Siebel eScript

Method	Purpose
Clib.strerror() Method	Gets a string describing an error number
throw Statement	Makes sure that script execution halts when an error occurs

Mathematical Methods and Properties in Siebel eScript

The eScript language provides the following mathematical methods and properties:

- "Numeric Methods in Siebel eScript" on page 73
- "Trigonometric Methods in Siebel eScript" on page 74
- "Mathematical Properties in Siebel eScript" on page 74

Numeric Methods in Siebel eScript

Table 22 provides a list of numeric methods.

Table 22. Numeric Methods in Siebel eScript

Method	Purpose
Clib.div() Method and Clib.ldiv() Method	Performs integer division and returns an object with the quotient and remainder
Clib.frexp() Method	Breaks a real number into a mantissa and an exponent as a power of 2
Clib.ldexp() Method	Calculates mantissa * 2 ^ exponent
Clib.modf() Method	Splits a value into integer and fractional parts
Clib.rand() Method	Returns a random real number between 0 and 1
Clib.srand() Method	Seeds the random number generator
Math.abs() Method	Returns the absolute value of an integer
Math.ceil() Method	Rounds a real number up to the next highest integer
Math.exp() Method	Computes the exponential function
Math.floor() Method	Rounds a real number down to the next lowest integer
Math.log() Method	Calculates the natural logarithm
Math.max() Method	Returns the largest of one or more values
Math.min() Method	Returns the smallest of one or more values

Table 22. Numeric Methods in Siebel eScript

Method	Purpose
Math.pow() Method	Calculates *x* to the power of *y*
Math.random() Method	Returns a random real number between 0 and 1
Math.round() Method	Rounds a value up or down
Math.sqrt() Method	Calculates the square root

Trigonometric Methods in Siebel eScript

Table 23 provides a list of trigonometric methods.

Table 23. Trigonometric Methods in Siebel eScript

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

Mathematical Properties in Siebel eScript

Table 24 provides a list of mathematical properties, each of which is a numeric constant.

Table 24. Mathematical Properties in Siebel eScript

Property	Value
Math.E Property	Value of *e*, the base for natural logarithms
Math.LN10 Property	Value of the natural logarithm of 10
Math.LN2 Property	Value of the natural logarithm of 2
Math.LOG10E Property	Value of the base 10 logarithm of *e*
Math.LOG2E Property	Value of the base 2 logarithm of *e*

Table 24. Mathematical Properties in Siebel eScript

Property	Value
Math.PI Property	Value of pi
Math.SQRT1_2 Property	Value of the square root of ½
Math.SQRT2 Property	Value of the square root of 2

Memory Manipulation Methods in Siebel eScript

Table 28 provides a list of methods with which to manipulate data at specific memory locations.

Table 25. Uncategorized Methods in Siebel eScript

Method	Purpose
SElib.peek() Method	Reads data from a specific position in memory
SElib.pointer() Method	Gets the address in memory of a Buffer variable
SElib.poke() Method	Writes data to a specific position in memory

Operating System Interaction Methods in Siebel eScript

Table 26 provides a list of operating system interaction methods.

Table 26. Operating System Interaction Methods in Siebel eScript

Method	Purpose
Clib.getenv() Method	Returns the value of an environment variable as a string
Clib.putenv() Method	Assigns a value to a specified environment variable
Clib.system() Method	Instructs the operating system to run the specified Command

String and Byte-Array Methods in Siebel eScript

Table 27 provides a list of string and byte-array methods.

Table 27. String and Byte-Array Methods in Siebel eScript

Method	Purpose
Clib.memchr() Method	Searches a byte array
Clib.memcmp() Method	Compares two byte arrays
Clib.memcpy() Method and Clib.memmove() Method	Copies or moves from one byte array to another
Clib.memset() Method	Copies from one byte array to another
Clib.rsprintf() Method	Returns a formatted string
Clib.sprintf() Method	Writes formatted output to a string
Clib.sscanf() Method	Reads and formats input from a string
Clib.strchr() Method	Searches a string for a character
Clib.strcspn() Method	Searches a string for the first character in a set of characters
Clib.stricmp() Method and Clib.strcmpi() Method	Makes a case-sensitive comparison of two strings
Clib.strlwr() Method	Converts a string to lowercase
Clib.strncat() Method	Concatenates a portion of one string to another
Clib.strncmp() Method	Makes a case-sensitive comparison of parts of two strings
Clib.strncmpi() Method and Clib.strnicmp() Method	Makes a case-insensitive comparison of parts of two strings
Clib.strncpy() Method	Copies a portion of one string to another
Clib.strpbrk() Method	Searches string for a character from a set of characters
Clib.strrchr() Method	Searches a string for the last occurrence of a character
Clib.strspn() Method	Searches a string for a character not in a set of characters
Clib.strstr() Method	Searches a string for a substring (case sensitive)
Clib.strstri() Method	Searches a string for a substring (case insensitive)
String charAt() Method	Returns the character at a specified location in a string
String indexOf() Method	Returns the index of the first instance of a specified substring in a string
String lastIndexOf() Method	Returns the index of the last instance of a specified substring in a string

Table 27. String and Byte-Array Methods in Siebel eScript

Method	Purpose
String match() Method	Returns an array of strings that are matches within the string against a target regular expression.
String search () Method	Searches a string for a regular expression pattern. If the pattern exists, this method returns the index of the regular expression.
RegExp compile() Method	Changes the pattern and attributes to use with the current instance of a RegExp object.
RegExp exec() Method	Returns an array of strings that are matches of the regular expression on the target string.
RegExp test() Method	Indicates whether a target string contains a regular expression pattern.
String split() Method	Parses a string and returns an array of strings based on a specified separator
String.fromCharCode() Static Method	Returns the character associated with a specified character code
substring() Method	Retrieves a section of a string
toLowerCase() Method	Converts a string to lowercase
toUpperCase() Method	Converts a string to uppercase

Uncategorized Methods in Siebel eScript

Table 28 provides a list of uncategorized methods.

Table 28. Uncategorized Methods in Siebel eScript

Method	Purpose
SElib.dynamicLink() Method	Calls a procedure from a dynamic link library (Windows) or shared object (UNIX)

4 Siebel eScript Commands

This chapter presents the eScript commands sorted alphabetically by object type and then by command name. It includes information on the following objects:

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. Methods and events of the Web Applet object are listed in Table 29.

Table 29. Web Applet Object Methods and Events

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.
FindActiveXControl() Method	FindActiveXControl returns a reference to a DOM element based upon the name specified in the name parameter.
FindControl() Method	FindControl returns the control whose name is specified in the parameter. This applet must be part of the displayed view.
InvokeMethod() Method	The InvokeMethod() method calls a parameter-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.

Table 29. Web Applet Object Methods and Events

Method or Event	Description
WebApplet_PreCanInvokeMethod() Event	The PreCanInvokeMethod() event is called before the PreInvokeMethod, allowing the developer to specify whether or not the user has the authority to invoke a specified WebApplet method. You can also use the applet user property CanInvokeMethod to enable and disable methods at the applet level. This is the recommended method of enabling applet controls because it is simpler than using the PreCanInvokeMethod() scripting event. For further information on the CanInvokeMethod user property, see *Siebel Developer's Reference*.
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.

Application Objects

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*. Table 30 provides a list of Application object methods and events.

Table 30. Application Object Methods and Events

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.

Table 30. Application Object Methods and Events

Method or Event	Description
Application_Close() Event	The Close() event is called before the application exits. This event 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.
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 parameter.
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.

Table 30. Application Object Methods and Events

Method or Event	Description
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 position 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 position name 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.
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 parameter. 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 parameter. 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 can use numbers as indices, arrays provide an easy way to work with sequential data. For example, to keep track of how many jelly beans you ate each day, you could graph your jelly bean 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 jelly beans 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

"Array Constructors in Siebel eScript" on page 84
"Array join() Method" on page 86
"Array length Property" on page 87
"Array reverse() Method" on page 88
"Array sort() Method" on page 89

Array Constructors in Siebel eScript

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 parameters. 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);
```

You can pass 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.

Associative Arrays in Siebel eScript

Siebel eScript supports associative arrays, where the array index can be a string instead of a number. This capability is useful when you want to associate values with specific names. For example you may want to have a month's array where the elements are the names of the months and the values are the number of days in the month.

To access items in an associative array, you use a string as an index. For example:

```
array_name["color"] = "red";
array_name["size"] = 15;
```

An advantage of associative arrays is that they are the only arrays that can be used with the for...in statement. This statement loops through every element in an associative array or object, regardless of how many or how few elements it may contain. For more information, see "for...in Statement" on page 54.

The following example creates an associative array of months and days, and totals the number of days.

```
// open file
var fp = Clib.fopen("c:\\months.log", "at");

// populate associative array
var months = new Array();
months["November"] = 30;
months["December"] = 31;
months["January"] = 31;
months["February"] = 28;

// iterate through array items
var x;
var total = 0;
```

```
for (x in months)
   {
    // write array items name and value to file
    Clib.fputs(x + " = " + months[x] + "\n",fp);
    // Add this month’s value to the total
    total = total + months[x];
   }
Clib.fputs ("Total = " + total + "\n",fp);

//close file
Clib.fclose(fp);
```

The output of this example is:

```
November = 30
December = 31
January = 31
February = 28
Total = 120
```

Array 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("*/*");
```

Array 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.

Array pop() Method

This method returns the last element of the current Array object, then removes the element from the array.

Syntax

arrayName.pop()

Returns

The last element of the current Array object.

Usage

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

Example

```
var a = new Array( "four" );
TheApplication().RaiseErrorText("First pop: " + a.pop() + ", Second pop: " + a.pop());
// First displays the last (and only) element, the string "four".
// Then displays "undefined" because the array is empty after
// the first call removes the only element.
```

Array push() Method

This method appends new elements to the end of an array.

Syntax

arrayName.push([*element1*,*element2*, ..., *elementn*])

Parameter	Description
element1, *element2*, . . . *elementn*	A list of elements to append to the array in the order given

Returns

The length of the array after the new elements are appended

Usage

This method appends the elements provided as arguments to the end of the array, in the order that they appear. The length of the current Array object is adjusted to reflect the change.

Example

```
    var a = new Array(1,2);
    TheApplication().RaiseErrorText(a.push(5,6) + "    " + a);
    // Displays 4  1,2,5,6, the length and the new array.
```

Array 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()

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"
```

Array 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* allows 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.

```
function compareNumbers(a, b)
{
   return a - b;
}
var a = new Array(5, 3, 2, 512);
var fp = Clib.fopen("C:\\log\\Trace.log", "a");
Clib.fprintf(fp, "Before sort: " + a.join() + "\n");
a.sort(compareNumbers);
Clib.fprintf(fp, "After sort: " + a.join() + "\n");
Clib.fclose(fp);
```

Array splice() Method

This method removes a specified number of elements from the array, starting at a given index, and returns an array of those removed elements. The method then rearranges the remaining elements as necessary to insert a specified number of new elements at the start index of the removed elements. The entire process effectively splices new elements into the array.

Syntax

`arrayName.splice(start, deleteCount[, element1, element2, . . . elementn])`

Parameter	Description
start	The index at which to splice in the new elements. ■ If *start* is negative, then (length+*start*) is used instead; that is, *start* indicates to splice at an index counting back from the end of the array. For example, *start* = -1 indicates to splice from the last element in the array. ■ If *start* is larger than the index of the last element, then the length of the array is used, effectively appending to the end of the array.
deleteCount	The number of elements to remove from the array. All of the available elements to remove are removed if deleteCount is larger than the number of elements available to remove.
element1, element2, . . . elementn	A list of elements to insert into the array in place of the ones that were removed.

Returns

An array consisting of the elements that are removed from the original array

Usage

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

Example

```
var a = new Array( 1, 2, 3, 4, 5 );
TheApplication().RaiseErrorText(a.splice(1,3,6,7) + "    " + a);
// Displays 2,3,4   1,6,7,5
// Beginning at element in position 1, three elements (a[1], a[2], a[3] = 2,3,4)
// are replaced with 6,7.
```

BLOB Objects

The following topics describe binary large objects (BLOBs).

blobDescriptor Objects

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:

Value	Description
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

Value	Description
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*)

Parameter	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

Parameter	Description
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 "blobDescriptor Objects" on page 92.

Example

This example shows how to get values from a Blob object.

```
function GetBlobVal()
{
   var a, b, c;
   a = "";
   b = 1234;
   c = 12345678;
   // Call a function to build the Blob
   var blob = BuildBlob(a, b, c);
   TheApplication().TraceOn("c:\\temp\\blob.txt","Allocation","All");
   // Get the values from the blob object
   // The first variable is string
   var resultA = Blob.get(blob,0,1000);
   // The second variable is an integer
   var resultB = Blob.get(blob,1000,UWORD16);
   // The third variable has a type of float
   var resultC = Blob.get(blob,1002,FLOAT64);
   TheApplication().Trace(resultA);
   TheApplication().Trace(resultB);
   TheApplication().Trace(resultC);
}

function BuildBlob(a, b, c)
{

   var blob;
   a = "Blob Test Value From Function";
   var offset = Blob.put(blob, 0, a, 1000);
   offset = Blob.put(blob, offset, b*2, UWORD16);
```

```
      Blob.put(blob, offset, c*2, FLOAT64);
      return blob;
   }
```

See Also

"blobDescriptor Objects" on page 92
"Blob.put() Method" on page 95

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(b`*`lobVar`*`[, `*`offse`*`t], `*`buffer, bufferLen`*`)`

Syntax C

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

Parameter	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 "blobDescriptor Objects" on page 92.

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

"blobDescriptor Objects" on page 92
"Blob.get() Method" on page 93

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)
```

Parameter	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* parameter must have one of the values listed for blobDescriptors in "blobDescriptor Objects" on page 92.

See Also

"blobDescriptor Objects" on page 92

Buffer Objects in Siebel eScript

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.

For an understanding of the Buffer Objects, see the following topics:

- "Buffer Constructors in Siebel eScript" on page 98
- "Buffer Object Methods" on page 100
- "Buffer Object Properties" on page 107

Buffer Constructors in Siebel eScript

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

Syntax A

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

Parameter	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*]);

Parameter	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*);

Parameter	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.

Example
The following example shows creation of new Buffer objects.

```
function BufferConstruct()
{
   TheApplication().TraceOn("c:\\temp\\BufferTrace.doc","Allocation","All");
   // Create empty buffer with size 100
   var buff1 = new Buffer(100 , true , true);
   // Create a buffer from string
   var buff2 = new Buffer("This is a buffer String constructor example", true);
   // Create buffer from buffer
   var buff3 = new Buffer(buff2,false);
   try
   {
      with(buff1)
      {
         // Add values from 0-99 to the buffer
         for(var i=0;i<size;i++)
         {
            putValue(i);
         }
         var val = "";
         cursor=0;
         // Read the buffer values into variable
         for(var i=0;i<size;i++)
         {
            val += getValue(1)+" ";
         }
         // Trace the buffer value
         TheApplication().Trace("Buffer 1 value: "+val);
      }
      with(buff2)
      {
         // Trace buffer 2
         TheApplication().Trace("Buffer 2 value: "+getString());
      }
      // Trace buffer 3
      with(buff3)
      {
         TheApplication().Trace("Buffer 3 value: "+getString());
      }
   }
   catch(e)
   {
      TheApplication().Trace(e.toString());
   }
}
```

Buffer Object Methods

Siebel eScript supports the following Buffer object methods.

- "getString() Method" on page 101
- "getValue() Method" on page 101

- "offset[] Method" on page 102
- "putString() Method" on page 103
- "putValue() Method" on page 104
- "subBuffer() Method" on page 105
- "toString() Method" on page 106

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 101
"offset[] Method" on page 102
"subBuffer() Method" on page 105

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 104

offset[] Method

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

Syntax

bufferVar[*offset*]

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

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 101
"putValue() Method" on page 104

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

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, add 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 enjoy coding with Siebel eScript."

```
function eScript_Click ()
{
   var exclamation = new Buffer("I enjoy coding with . . .");
   var language = "Siebel eScript.";
   exclamation.cursor = 20;
   exclamation.putString(language);
   TheApplication().RaiseErrorText(exclamation);
}
```

See Also

"getString() Method" on page 101

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
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

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, add code 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 conversion is sufficiently insignificant to ignore, but note that the following does not hold true:

```
bufferItem.putValue(1.4,8,"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 101

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 101

toString() Method

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

Syntax

bufferVar.toString()

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.

Example

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

Buffer Object Properties

Siebel eScript supports the following Buffer object properties.

- "bigEndian Property" on page 107
- "cursor Property" on page 108
- "data Property" on page 108
- "size Property" on page 108
- "unicode Property" on page 109

bigEndian Property

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

Syntax

`*bufferVar*.bigEndian`

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

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, see "getString() Method" on page 101 and "subBuffer() Method" on page 105.

See Also

"subBuffer() Method" on page 105

data Property

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

Syntax

bufferVar.data

Usage

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

size Property

The size of the Buffer object.

Syntax

bufferVar.size

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 108

unicode Property

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

Syntax

`*bufferVar*.unicode`

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*.

Table 31 provides a list of Business Component object methods and events.

Table 31. Business Component Object Methods and Events

Method or Event	Description
ActivateField() Method	ActivateField() allows setting search specifications and queries to retrieve data for the field specified in its parameter.
ActivateMultipleFields() Method	ActivateMultipleFields() allows the script to do ActivateField() for many fields at one time. These fields are listed in a property set.
Associate() Method	The Associate() method creates a new many-to-many relationship for the parent object through an association business component (see "GetAssocBusComp() Method" on page 112).
BusComp_Associate() Event	The Associate() event is called after a record has been added to a business component to create an association.
BusComp_ChangeRecord() Event	The ChangeRecord() event is called when a business component changes its current record from one record to another, for example when a user changes the record focus in an applet or when a script calls the NextRecord() method.
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.
BusComp_DeleteRecord() Event	The DeleteRecord() event is called after a row is deleted. The current context moves to a different row because 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.

Table 31. Business Component Object Methods and Events

Method or Event	Description
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.
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() deletes the current business component record from the database.
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. ExecuteQuery2() is an SQL-Server specific version of ExecuteQuery().

Table 31. Business Component Object Methods and Events

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 parameter 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 parameter 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() is used in scripts and effectively performs many GetFieldValue() calls using a list of fields specified in a property set.
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* parameter.
GetUserProperty() Method	GetUserProperty() returns the value of a named UserProperty.
GetViewMode() Method	GetViewMode() returns the current visibility mode for the business component. This method 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 parameter.

Table 31. Business Component Object Methods and Events

Method or Event	Description
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 places the currently selected record in a picklist business component into the appropriate fields of the parent business component. See also "GetPicklistBusComp() Method" on page 112.
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.
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() is used in scripts and effectively performs many SetFieldValue() calls using a list of fields specified in a property set.
SetNamedSearch() Method	SetNamedSearch() sets a named search specification on the business component. A named search specification is identified by the *searchName* parameter.
SetSearchExpr() Method	SetSearchExpr() sets one search expression with many fields for the whole business component, rather than setting one search specification for each field.
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.

Table 31. Business Component Object Methods and Events

Method or Event	Description
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 reversal 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

A Siebel business object groups one or more business components into a logical unit of information. 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. 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.

Do not store Siebel objects, such as business objects and business components, as properties of custom objects, such as shown in the following example:

```
var oParms = new Object;
oParms.bo = TheApplication().GetBusObject("List Of Values");
```

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 stand-alone 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 existing Siebel business service objects are documented in the *Siebel Object Interfaces Reference*.

Table 32 provides a list of Business Service object methods and events.

Table 32. Business Service Object Methods and Events

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 parameter.
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_PreCanInvokeMethod() Event	The PreCanInvokeMethod() event is called before the PreInvokeMethod, so the developer can determine whether or not the user has the authority to invoke the business service method.
Service_PreInvokeMethod() Event	The PreInvokeMethod() event is called before a specialized method is invoked on the business service.
SetProperty() Method	This method assigns a value to a property of a business service.

About Script Libraries

In Siebel 8.0, the ST eScript engine provides business services script libraries. Script libraries assist in the development of reusable, modular components that can be upgraded and that are easy to maintain. Script libraries provide a framework for invoking global scripts.

The script libraries feature provides the following functions:

- Allows you to call a business service function directly from anywhere within the scripting interface after you have declared the business service, without having to declare property sets or issue InvokeMethod calls
- Allows you to write strongly typed methods for built-in business services, which can then be called using the Script Assist utility

NOTE: Using script libraries is optional. All code written before Siebel 8.0 is still supported.

Example of Calling a Business Service Function Directly

The following code illustrates how to call a method on the Data Transfer Service directly, without declaring property sets. Calling business service methods this way results in shorter, more readable scripts.

```
var oBS : Service = TheApplication ().GetService ("Data Transfer Service");
oBS.SendData ("Name", "John Doe");
```

Example of a Custom Method for a Business Service

You can write custom methods for business services and make these available from within Script Assist. For example, the following code creates a custom wrapper method, SendData, on the Data Transfer Service. You can intercept and change the calls to the Data Transfer Service in a centralized location in the SendData method.

```
function SendData (sTag : String, sValue : String)
{
  var oPS1 = TheApplication ().NewPropertySet ();
  var oPS2 = TheApplication ().NewPropertySet ();

  oPS1.SetProperty ("Tag", sTag);
  oPS1.SetProperty ("Value", sValue);

  this.InvokeMethod ("SendData", oPS1, oPS2)
}
```

Displaying a Custom Method Within Script Assist

To make a custom method available to the script libraries so that it can be called from within Script Assist, you must do the following:

- Save the business service method script.
- Ensure the script does not contain compile errors.

  For functions called using script libraries, the compiler checks that argument types are valid and do not contain incompatibilities.
- Check the External Use flag for the business service object in Siebel Tools.

The custom method for the service is added to the script libraries and will display from the Script Assist utility.

To display custom business service methods within Script Assist, perform the following procedure.

To display custom methods in Script Assist

1. Access Script Assist from the script editor by pressing CTRL+SPACE.
2. In your script, type the name of a business service object followed by a period (.).

   All the default and custom scripted methods available for the business service object are displayed.
3. Select the method that you want to add to your script.

For detailed information on Script Assist, on setting object properties, and on using the Server Script Editor to create, save, and compile scripts, see *Using Siebel Tools*.

Clib Objects

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 37, may be considered redundant because their functionality already exists in eScript. Where possible, you should use standard eScript methods instead of the equivalent Clib functions. The Clib library is supported in Unix and Windows application servers. It is not supported for client-side scripting (Browser script).

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.

For an understanding of the Clib object, see the following topics:

- "Clib Object Buffer Methods in Siebel eScript" on page 118
- "Clib Object Character Classification in Siebel eScript" on page 120
- "Clib Object Error Methods" on page 129
- "File I/O Methods in eScript" on page 130
- "Formatting Data in eScript" on page 159
- "Clib Object Math Methods" on page 163
- "Redundant Functions in the Clib Object" on page 171
- "Clib Object String Methods" on page 172
- "Time Objects" on page 186
- "Clib Object Time Methods" on page 186
- "Clib Object Uncategorized Methods" on page 196

Clib Object Buffer Methods in Siebel eScript

The eScript language has the following commands for buffer manipulation:

- "Clib.memchr() Method" on page 118
- "Clib.memcmp() Method" on page 118
- "Clib.memcpy() Method and Clib.memmove() Method" on page 119
- "Clib.memset() Method" on page 120

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

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 *destBuf.*

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

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 Object Character Classification in Siebel eScript

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 parameters 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 121

- "Clib.isalpha() Method" on page 122
- "Clib.isascii() Method" on page 122
- "Clib.iscntrl() Method" on page 123
- "Clib.isdigit() Method" on page 123
- "Clib.isgraph() Method" on page 124
- "Clib.islower() Method" on page 124
- "Clib.isprint() Method" on page 125
- "Clib.ispunct() Method" on page 126
- "Clib.isspace() Method" on page 126
- "Clib.isupper() Method" on page 127
- "Clib.isxdigit() Method" on page 127
- "Clib.toascii() Method" on page 128

Clib.isalnum() Method

This function returns true if a specified character is alphanumeric.

Syntax

Clib.isalnum(*char*)

Parameter	Description
char	Either 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 122
"Clib.isdigit() Method" on page 123
"Clib.islower() Method" on page 124
"Clib.isprint() Method" on page 125
"Clib.isupper() Method" on page 127

Clib.isalpha() Method

This function returns true if a specified character is alphabetic.

Syntax

`Clib.isalpha(`*char*`)`

Parameter	Description
char	Either 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.isalnum() Method" on page 121
"Clib.isdigit() Method" on page 123
"Clib.islower() Method" on page 124
"Clib.isprint() Method" on page 125
"Clib.isupper() Method" on page 127

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	Either 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 123
"Clib.isprint() Method" on page 125

Clib.iscntrl() Method

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

Syntax

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

Parameter	Description
char	Either 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 122

Clib.isdigit() Method

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

Syntax

```
Clib.isdigit(char)
```

Parameter	Description
char	Either 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 121
"Clib.isalpha() Method" on page 122
"Clib.isupper() Method" on page 127

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	Either 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 125
"Clib.ispunct() Method" on page 126
"Clib.isspace() Method" on page 126

Clib.islower() Method

This function returns true if a specified character is a lowercase alphabetic character.

Syntax

`Clib.islower(`*`char`*`)`

Parameter	Description
char	Either 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 121
"Clib.isalpha() Method" on page 122
"Clib.isupper() Method" on page 127

Clib.isprint() Method

This function returns true if a specified character is printable.

Syntax

`Clib.isprint(`*`char`*`)`

Parameter	Description
char	Either 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 121
“Clib.isascii() Method” on page 122
“Clib.isgraph() Method” on page 124
“Clib.ispunct() Method” on page 126
“Clib.isspace() Method” on page 126

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	Either 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 124
“Clib.isprint() Method” on page 125
“Clib.isspace() Method” on page 126

Clib.isspace() Method

This function returns true if a specified character is a white-space character.

Syntax

```
Clib.isspace(char)
```

Parameter	Description
char	Either 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 122
"Clib.isprint() Method" on page 125

Clib.isupper() Method

This function returns true if a specified character is an uppercase alphabetic character.

Syntax

`Clib.isupper(`*`char`*`)`

Parameter	Description
char	Either 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 122
"Clib.islower() Method" on page 124

Clib.isxdigit() Method

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

Syntax

`Clib.isxdigit(`*`char`*`)`

Parameter	Description
char	Either 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 123

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 122

Clib Object Error Methods

Siebel eScript has the following Clib methods for handling errors:

- "Clib.clearerr() Method" on page 133
- "Clib.errno Property" on page 129
- "Clib.perror() Method" on page 129
- "Clib.strerror() Method" on page 130

Clib.errno Property

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

Syntax

```
Clib.errno
```

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.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.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 129

File I/O Methods in eScript

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() method. You can also open a file using Clib.tmpfile().

These methods 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.chdir() Method" on page 132

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. To skip the BOM, call Clib.fgetc/getc once.

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. The path can be absolute or relative.

Returns

0 if successful; otherwise, -1.

Usage

This method changes the current directory for the Siebel application. If the Siebel Server is restarted, the current directory is automatically reset as one of the following:

- The current directory recognized by the Windows operating system on the Siebel Server
- The home directory of the administrator who restarts the Siebel Server on UNIX

Example

The following example shows the use of Clib.chdir() to change the current working directory of the Siebel application. The default Siebel working directory is the *SIEBEL_ROOT*\bin directory. For example, if you installed the Siebel client in C:\sea752\client, then the default working directory is C:\sea752\client\bin.

```
function Application_Start (CommandLine)
{

   // Start Tracing
   TheApplication().TraceOn("c:\\temp\\SiebTrace.txt","Allocation","All");

   var currDir = Clib.getcwd();
   TheApplication().Trace("Current directory is " + Clib.getcwd());

   // Create a new directory
   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 gives an
      // error
      Clib.rmdir("Clib test");
      msg = Clib.chdir("C:\\Clib test");
      TheApplication().Trace(msg);
   }
```

Here are the trace results from the script:

```
   Current directory is D:\sea752\client\BIN
   0
   Current directory is C:\Clib test
   -1
```

See Also

"Clib.getenv() Method" on page 198
"Clib.mkdir() Method" on page 154
"Clib.rmdir() Method" on page 156

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

Usage

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

Clib.getcwd() Method

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

Syntax

Clib.getcwd()

Returns

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

Usage

In a Siebel application, the default current working directory is the directory in which the application has been installed. If a script changes the current working directory (using `Clib.chdir()` or similar command), the current working directory returns to its original value when the script finishes.

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.clearerr() Method” on page 133
“Clib.mkdir() Method” on page 154
“Clib.rmdir() Method” on page 156

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 136

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 129

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.

See Also

"Clib.getenv() Method" on page 198

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 139
"Clib.qsort() Method" on page 199

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.

```
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);
   var msg = Clib.fgets(fp));
   Clib.fclose(fp);
   TheApplication().RaiseErrorText(msg);
}
```

See Also

“Clib.feof() Method” on page 135
“Clib.fsetpos() Method” on page 151
“Clib.ftell() Method” on page 152

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 parameter is the length in Unicode characters. If you do not specify *maxLen*, then eScript uses the default limit of 999 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.

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

Running this code produces the following result.

```
abcdefg
ABCDEFG
```

See Also

"Clib.fputs() Method" on page 145

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 (nonblocking)

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 required characters specifying a file mode, followed by optional characters, as described in Table 33

Returns

This method returns a file pointer to the file opened or null, if the function fails. This return value is on object of type File.

NOTE: Several Clib methods require an argument denoted as *filePointer* in this document. These input arguments are of type File and are often the return value of a Clib.fopen() call.

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 shown in Table 33.

Table 33. Clib.fopen() Mode Parameters

Parameter	Required?	Mode
r	Yes, only one of these parameters is required.	Opens the file for reading; the file must already exist.
w		Opens the file for writing. If the file does not exist, eScript creates the file.
a		Opens the file in append mode.
b	No	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	No	Opens the file in text mode. Do not use for non-ASCII characters, use “u” instead.
u	No	Opens the file in Unicode mode; for example, Clib.fopen(“filename.txt”, “rwu”). Use this mode for both ASCII and non-ASCII characters.
+	No	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:File = 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 that opens a file, writes a string to the file, then reads the string from the file, 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 that opens a file, writes a string to the file, then reads the string from the file, 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.getenv() Method" on page 198
"Clib.tmpfile() Method" on page 158

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

Usage

This method writes a formatted string to the file indicated by *filePointer*. For information on format strings used with Clib.fprintf(), see Table 34 on page 160.

Example

The following example shows uses of Clib.fprintf() with various format string parameters.

```
function Service_PreInvokeMethod (MethodName, Inputs, Outputs)
{
   if (MethodName == "fprintfsamples")
   {
      var intgr = 123456789;
      var flt = 12345.6789;
      var hour = 1;
      var min = 7;
      var sec = 0;
      var str = "Hello World";
      var file = Clib.fopen("c:\\temp\\fprintf.txt", "w");

      // Simple formatting:
      Clib.fprintf(file, "(1) %s, it is now %i:%i:%i pm.\n", str, hour, min, sec);
      Clib.fprintf(file, "(2) The number %i is the same as %x.\n", intgr, intgr);
      Clib.fprintf(file, "(3) The result is %f.\n", flt);

      // Flag values:
      // "+" forces a + or - sign; "#" modifies the type flag "x"

      // to prepend "0x" to the output. (Compare with the simple
      // formatting example.)
      Clib.fprintf(file, "(4) The number %+i is the same as %#x.\n", intgr, intgr);
```

```
         // Width values:
         // Note that the width is a minimal width, thus longer values
         // are not truncated.
         // "2" fills with spaces, "02" fills with zeros.
         var myWidth = 2;
         Clib.fprintf(file, "(5) %5s, it is now %2i:%02i:%02i pm.\n", str, hour, min,
         sec);

         // Precision values:
         // ".2" restricts to 2 decimals after the decimal separator.
         // Note that the number will be rounded appropriately.
         Clib.fprintf(file, "(6) The result is %.2f.\n", flt);

         // A combined example:
         // <space> displays either space or minus;
         // "+" displays either plus or minus;
         // "020" uses a minimal width of 20, padded with zeros;
         // ".2" displays 2 digits after the decimal separator;
         // "*" uses the next argument in the list to specify the width.
         Clib.fprintf(file, "(7) The values are:\n%+020.2f\n% 020.2f\n% *.2f", flt,
         intgr, 20, intgr);

         Clib.fclose(file);

         return (CancelOperation);
      }
      return (ContinueOperation);
   }
```

The script produces the following output:

```
(1) Hello World, it is now 1:7:0 pm.
(2) The number 123456789 is the same as 75bcd15.
(3) The result is 12345.678900.
(4) The number +123456789 is the same as 0x75bcd15.
(5) Hello World, it is now  1:07:00 pm.
(6) The result is 12345.68.
(7) The values are:
+0000000000012345.68
0000000123456789.00
123456789.00
```

See Also

"Clib.rsprintf() Method" on page 173
"Clib.sprintf() Method" on page 173

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 137
"Clib.fputs() Method" on page 145

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, see "Clib.fgets() Method" on page 139.

See Also

"Clib.fgets() Method" on page 139
"Clib.fputc() Method and Clib.putc() Method" on page 144

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 shown in the following table.

Value	Description
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 152

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
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.

Example

The following sample script writes to two different files using the same file pointer.

```
var oFile = Clib.fopen("c:\\temp\\firstfile","w");
if (oFile == null)
{
   TheApplication().RaiseErrorText("File not found.");
}
Clib.fprintf(oFile, "Writing to first file\n");
Clib.freopen("c:\\temp\\secondfile", "w", oFile);
if (oFile == null)
{
   TheApplication().RaiseErrorText("File not found.");
}
Clib.fprintf(oFile, "Writing to second file\n");
Clib.fclose(oFile);
```

See Also

"Clib.getenv() Method" on page 198
"Clib.fopen() Method" on page 140

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, ..., varn* 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, see "Formatting Input in eScript" on page 162.

Example

The following example shows uses of Clib.fscanf() with various options on the parameters.

```
var int1;
var int2;
var hour;
var min;
var sec;
var str;

var file = Clib.fopen("c:\\temp\\fscanf.txt", "r");
TheApplication().TraceOn("c:\\temp\\testoutput.txt", "allocation", "all");

// Simple scanf:
// input line e.g.: "Monday 10:18:00"
Clib.fscanf(file, "%s %i:%i:%i\n", str, hour, min, sec);
TheApplication().Trace(str + ", " + hour + ", " + min + ", " + sec);

// Using width specifier:
// input line e.g.: "1234567890"
Clib.fscanf(file, "%5i%5i\n", int1, int2);
TheApplication().Trace(int1 + ", " + int2);

// Reading hexadecimal integers and suppressing assignment to a variable:
// input line e.g.: "AB3F 456A 7B44"
Clib.fscanf(file, "%x %*x %x\n", int1, int2);
TheApplication().Trace(int1 + ", " + int2);

// Using character ranges:
// input line e.g.: "helloHELLO"
Clib.fscanf(file, "%[a-z]\n", str);
TheApplication().Trace(str);

Clib.fclose(file);
```

The script produces the following trace output:

```
COMMENT,"Monday, 10, 18, 0"
COMMENT,"12345, 67890"
COMMENT,"43839, 31556"
COMMENT,hello
```

See Also

"Clib.sinh() Method" on page 168
"Clib.sscanf() Method" on page 157

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 138
"Clib.ftell() Method" on page 152
"Clib.rewind() Method" on page 155

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, see "Clib.fgetpos() Method" on page 138.

See Also

"Clib.fgetpos() Method" on page 138
"Clib.ftell() Method" on page 152

Clib.ftell() Method

This method gets 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 gets 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 150
"Clib.fsetpos() Method" on page 151

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...	Then, the 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 146

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 146

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.

See Also

"Clib.clearerr() Method" on page 133
"Clib.getenv() Method" on page 198
"Clib.rmdir() Method" on page 156

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.

See Also

"Clib.fopen() Method" on page 140

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()

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, see "Clib.fgets() Method" on page 139.

See Also

"Clib.fseek() Method" on page 150

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.

See Also

“Clib.clearerr() Method” on page 133
“Clib.getenv() Method” on page 198
“Clib.mkdir() Method” on page 154

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
var1, var2, ..., varn	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 parameters 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, see “Formatting Input in eScript” on page 162.

See Also

"Clib.fscanf() Method" on page 149
"Clib.sinh() Method" on page 168
"Clib.asctime() Method" on page 187

Clib.tmpfile() Method

This method creates a temporary binary file and returns its file pointer.

Syntax

```
Clib.tmpfile()
```

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, see "Clib.fgets() Method" on page 139.

See Also

"Clib.fopen() Method" on page 140

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.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 137
"Clib.fputc() Method and Clib.putc() Method" on page 144
"Clib.putenv() Method" on page 198

Formatting Data in eScript

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

Formatting Output in eScript

Table 34 lists the format strings for use with the print functions: fprintf() (see "Clib.fprintf() Method" on page 143), rsprintf(), and sprintf() (see "Clib.rsprintf() Method" on page 173). 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 percent characters together (%%).

Table 34. Format Strings for the Print Functions

Formatting Character	Effect	Example Statement and Output
Flag Values		

-	Left justification in the field with space padding or right justification with zero or space padding	`fprintf(file, "[%-8i]", 26);`
		`[26      ]`

+	Force numbers to begin with a plus (+) or minus (-)	`fprintf(file, "%+i", 26);`
		`+26`

space	Negative values begin with a minus (-); positive values begin with a space	`fprintf(file, "[% i]", 26);`
		`[ 26]`

#	Append one of the following symbols to the # character to display the output in the indicated form:
	■ o to prefix a zero to nonzero octal output
	■ x or X to prefix 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, and display the output in scientific notation
	■ g or G to include a decimal point even if no digits follow the decimal point, display the output in scientific notation (depending on precision), and leave trailing zeros in place
		`fprintf(file, "%#o", 26);`
		`032`
		
		`fprintf(file, "%#x", 26);`
		`0x1A`
		
		`fprintf(file, "%#.f", 26);`
		`26.`
		
		`fprintf(file, "%#e", 26);`
		`2.600000e+001`
		
		`fprintf(file, "%#g", 26);`
		`26.0000`

Table 34. Format Strings for the Print Functions

Formatting Character	Effect	Example Statement and Output
Width Values		

n	At least *n* characters are output; if the value is fewer than *n* characters, the output is padded on the left with spaces.	`fprintf(file, "[%8s]", "Test");`
		`[    Test]`

0*n*	At least *n* characters are output, padded on the left with zeros.	`fprintf(file, "%08i", 26);`
		`00000026`

*	The next value in the parameter list is an integer specifying the output width.	`fprintf(file, "[%*s]", 8, "Test");`
		`[    Test]`

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.	`fprintf(file, "%.0f", 26.735);`
		`26`

.*n*	Output is *n* characters, or *n* decimal places if the value is a floating-point number.	`fprintf(file, "%.2f", 26.735);`
		`26.73`

.*	The next value in the parameter list is an integer specifying the precision width.	`fprintf(file, "%.*f", 1, 26.735);`
		`26.7`

Type Values		

d,i	Signed integer	`fprintf(file, "%i", 26);`
		`26`

u	Unsigned integer	`fprintf(file, "%u", -1);`
		`4294967295`

o	Octal integer	`fprintf(file, "%o", 26);`
		`32`

x	Hexadecimal integer using `0` through 9 and `a, b, c, d, e, f`	`fprintf(file, "%x", 26);`
		`1a`

X	Hexadecimal integer using `0` through 9 and `A, B, C, D, E, F`	`fprintf(file, "%X", 26);`
		`1A`

f	Floating-point of the form `[-]dddd.dddd`	`fprintf(file, "%f", 26.735);`
		`26.735000`

e	Floating-point of the form `[-]`*`d.ddd`*e+*`dd`* or `[-]`*`d.ddd`*e-*`dd`*	`fprintf(file, "%e", 26.735);`
		`2.673500e+001`

Table 34. Format Strings for the Print Functions

Formatting Character	Effect	Example Statement and Output
E	Floating-point of the form `[-]`*d.ddd*`E+`*dd* or `[-]`*d.ddd*`E-`*dd*	`fprintf(file, "%E", 26.735);` `2.673500E+001`
g	Floating-point number of f or e type, depending on precision	`fprintf(file, "%g", 26.735);` `26.735`
G	Floating-point number of F or E type, depending on precision	`fprintf(file, "%G", 26.735);` `26.735`
c	Character; for example, `'a'`, `'b'`, `'8'`	`fprintf(file, "%c", 'a');` `a`
s	String	`fprintf(file, "%s", "Test");` `Test`

Formatting Input in eScript

Format strings are also used with the scan functions: fscanf() (see "Clib.fscanf() Method" on page 149), sscanf() (see "Clib.sscanf() Method" on page 157), 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 parameters 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 * and *width* values may be as shown on Table 35.

Table 35. Scan Functions Formatting Parameters * and width

Parameter	Description
*	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 36 lists the characters that define the *type.*

Table 36. Type Values for the Scan 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
[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

Example

This sample script creates a file called myfile.txt and stores a float number and a string. Then the stream is rewound and both values are read with fscanf.

```
function WebApplet_Load()
{
   var f;
   var str;
   var pFile = Clib.fopen ("c:\\myfile.txt","w+");
   Clib.fprintf (pFile, "%f %s", 3.1416, "PI");
   Clib.rewind (pFile);
   Clib.fscanf (pFile, "%f", f);
   Clib.fscanf (pFile, "%s", str);
   Clib.fclose (pFile);
   Clib.printf ("I have read: %f and %s \n",f,str);
}
```

Here are the trace results from the script:

```
I have read: 3.141600 and PI
```

Clib Object Math Methods

Siebel eScript has the following Clib math methods.

- "Clib.cosh() Method" on page 164
- "Clib.div() Method and Clib.ldiv() Method" on page 164
- "Clib.frexp() Method" on page 165
- "Clib.ldexp() Method" on page 166

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 168
"Clib.tanh() Method" on page 169
"Math.cos() Method" on page 268

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 elements shown in the following table, which are the result of dividing numerator by denominator.

Return Element	Description
.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 + ".");
```

When run with the values of n=9 and d=4, this example produces this result.

```
The quotient is 2.

The remainder is 1.
```

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 so 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.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 165

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(). The result is the same for both.

```
function eScript_Click ()
{
   Clib.modf(32.154, var x);
   var y = ToInteger(32.154);
   TheApplication().RaiseErrorText("modf yields " + x +
         ".\nToInteger yields " + y + ".");
}
```

When run, this example produces the following result:

```
modf yields 32
ToInteger yields 32.
```

See Also

"ToInteger() Method" on page 252

Clib.rand() Method

This method generates a random number between 0 and RAND_MAX, inclusive.

Syntax

`Clib.rand()`

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(). If the 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.srand() Method" on page 168
"Math.random() Method" on page 274

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 164
"Clib.tanh() Method" on page 169

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

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 167
"Math.random() Method" on page 274

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 164
"Clib.sinh() Method" on page 168

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, see "Clib.div() Method and Clib.ldiv() Method" on page 164.

Example

For an example, see "Clib.div() Method and Clib.ldiv() Method" on page 164.

See Also

"Clib.div() Method and Clib.ldiv() Method" on page 164
"rem Method" on page 170

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, see "Clib.div() Method and Clib.ldiv() Method" on page 164.

Example

For an example, see "Clib.div() Method and Clib.ldiv() Method" on page 164.

See Also

"Clib.div() Method and Clib.ldiv() Method" on page 164
"quot Method" on page 169

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 eScript. In most cases, the newer eScript 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.

Each Clib method listed in Table 37 is paired with the equivalent methods in eScript. Because Siebel eScript and the ECMAScript standard are developing and growing, the eScript methods are always to be preferred over equivalent methods in the Clib object.

Table 37. Correspondence Between Clib and ECMAScript Methods

Clib Method	Description	eScript 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()
sqrt()	Calculates the square root	Math.sqrt()

Table 37. Correspondence Between Clib and ECMAScript Methods

Clib Method	Description	eScript Method
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

Clib Object String Methods

Siebel eScript has the following string methods for the Clib object.

- “Clib.rsprintf() Method” on page 173
- “Clib.sprintf() Method” on page 173
- “Clib.strchr() Method” on page 175
- “Clib.strcspn() Method” on page 176
- “Clib.stricmp() Method and Clib.strcmpi() Method” on page 177
- “Clib.strlwr() Method” on page 178
- “Clib.strncat() Method” on page 178
- “Clib.strncmp() Method” on page 179
- “Clib.strncmpi() Method and Clib.strnicmp() Method” on page 180
- “Clib.strncpy() Method” on page 180
- “Clib.strpbrk() Method” on page 181
- “Clib.strrchr() Method” on page 182
- “Clib.strspn() Method” on page 183
- “Clib.strstr() Method” on page 184
- “Clib.strstri() Method” on page 185

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 34 on page 160 in the section "Clib.asctime() Method" on page 187. 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:

```
var TempStr = Clib.rsprintf("I count: %d %d %d.",1,2,3) //"I count: 1 2 3"
var a = 1;
var b = 2;
TempStr = Clib.rsprintf("%d %d %d",a, b, a+b) //"1 2 3"
```

See Also

"Clib.sprintf() Method" on page 173

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 34 on page 160 in the section "Clib.asctime() Method" on page 187. 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

The following examples show Clib.sprintf() used with various format string parameters. Trace file output follows after the script.

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");

var a, b, c;
a = 5;
b = 2;

Clib.sprintf(c, "First # %d + Second # %d is equal to %03d",a,b, a+b);
TheApplication().Trace("Output : " + c);

Clib.sprintf(c, "\n First # %d \n Second # %d \n => %d",12,16, 12+16)
TheApplication().Trace("Output : " + c);

var x, y, z, n;
var x = "Ali is 25 years old";
var y = "he lives in Ireland.";
var n = Clib.sprintf(z, "\n %s and %s",x,y)  ;

TheApplication().Trace("Output : " + z);
TheApplication().Trace("Total characters: " + n);

var  a = 16.51;
var  b = 5.79;
var c;
```

```
Clib.sprintf(c, "%.3f / %.3f is equal to %0.3f",a,b, parseFloat(a/b));
TheApplication().Trace("Output : " + c);

TheApplication().TraceOff();
```

The script produces the following trace output.

```
02/18/04,18:37:35,START,7.5.3 [16157] LANG_INDEPENDENT,SADMIN,3964,3836
02/18/04,18:37:35,COMMENT,Output : First # 5 + Second # 2 is equal to 007
02/18/04,18:37:35,COMMENT,"Output :
 First # 12
 Second # 16
 => 28"
02/18/04,18:37:35,COMMENT,"Output :
 Ali is 25 years old and he lives in Ireland."
02/18/04,18:37:35,COMMENT,Total characters: 46
02/18/04,18:37:35,COMMENT,Output : 16.510 + 5.790 is equal to 2.851
02/18/04,18:37:35,STOP
```

See Also

"Clib.rsprintf() Method" on page 173

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 which is to be searched
char	The character to be searched

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*. The return value is zero-based. The first character is zero, the second is 1, and so on.

Usage

This method searches the parameter *string* for the character *char*. When possible, you should use the standard JavaScript method substring(). For more information see "String replace() Method" on page 310.

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.

```
I can't stand soggy cereal.
stand soggy cereal.
```

See Also

"Clib.strcspn() Method" on page 176
"Clib.strpbrk() Method" on page 181
"Clib.strrchr() Method" on page 182
"String replace() Method" on page 310

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 be searched

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*. The return value is zero-based. The first character is zero, the second is 1, and so on.

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() (see "String replace() Method" on page 310).

Example

The following fragment demonstrates the difference between Clib.strcspn() and Clib.strpbrk():

```
var string = "There's more than one way to climb a mountain.";
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 climb a mountain.
strpbrk returns a string: way to climb a mountain.
strcspn returns an integer: 22
```

See Also

“Clib.strchr() Method” on page 175
“Clib.strpbrk() Method” on page 181
“String replace() Method” on page 310

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 179

"Clib.strncmpi() Method and Clib.strnicmp() Method" on page 180

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);
```

See Also

"Clib.strncpy() Method" on page 180

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 177
“Clib.strncmpi() Method and Clib.strnicmp() Method” on page 180

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 177
“Clib.strncmp() Method” on page 179

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 read
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 178

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(). For more information, see "String replace() Method" on page 310.

Example

For an example using this function, see "Clib.strcspn() Method" on page 176. To accomplish the same result using standard JavaScript methods, see "String replace() Method" on page 310.

See Also

"Clib.strchr() Method" on page 175
"Clib.strcspn() Method" on page 176
"String replace() Method" on page 310

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

This function returns the substring of *string* beginning at the rightmost occurrence of *char* and ending with the rightmost character in *string*. If *char* is not found in *string*, the function returns null.

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() (see "String replace() Method" on page 310).

Example

The following code fragment:

```
    var str = "I don’t like soggy cereal."
    var substr = Clib.strrchr(str, 'o');
    TheApplication().RaiseErrorText("str = " + str + "\nsubstr = " + substr);
```

results in the following output:

```
    str = I don’t like soggy cereal.
    substr = oggy cereal.
```

See Also

“Clib.strchr() Method” on page 175
“Clib.strcspn() Method” on page 176|
“Clib.strpbrk() Method” on page 181
“String replace() Method” on page 310

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 character in *string* that is not a member of *charSet.*

Usage

This method searches the characters from the beginning of *string*, then returns the offset of the first character that is not a member of *charSet*. The search is case sensitive, so you may have to include both uppercase and lowercase instances of characters in *charSet*.

This method is similar to Clib.strpbrk(), except that Clib.strpbrk() returns the string beginning at the first character found, while Clib.strspn() returns the offset number for that character.

When possible, you should use the standard JavaScript method substring() (see “String replace() Method” on page 310).

Example

The following fragment demonstrates Clib.strspn(). When searching string, it returns the position of the w, counting from 0.

```
var string = "There is more than one way to swim.";
var rStrspn = Clib.strspn(string, " aeiouTthrsmn");
TheApplication().RaiseErrorText("strspn returns an integer: " + rStrspn);
```

This results in the following output:

```
strspn returns an integer: 23
```

See Also

"Clib.strchr() Method" on page 175
"Clib.strcspn() Method" on page 176
"Clib.strpbrk() Method" on page 181
"String replace() Method" on page 310

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 185
"String replace() Method" on page 310

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 184.

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 184
"String replace() Method" on page 310

Time Objects

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 38.

Table 38. 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

Clib Object Time Methods

In the methods listed in Table 39, *Time* represents a variable in the Time object format, while *timeInt* represents an integer time value.

The Clib object supports the following time methods.

Table 39. Time Methods and the Objects They Return

Method	Object Returned
Clib.asctime() Method	*Time*
Clib.clock() Method	CPU tick count
Clib.div() Method and Clib.ldiv() Method	*timeInt*
Clib.difftime() Method	*timeInt*
Clib.gmtime() Method	*timeInt*
Clib.localtime() Method	*timeInt*
Clib.mktime() Method	*Time*
Clib.strftime() Method	*Time*
Clib.tmpnam() Method	*timeInt*

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, see "Time Objects" on page 186. The returned string has the format *Day Mon dd hh:mm:ss yyyy;* for example, `Wed Aug 10 13:21:56 2005`.

Example

This script shows the difference between asctime() and mkdir() formats for time.

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");

var tm = Clib.localtime(Clib.time());
var tmStr = Clib.asctime(tm);
```

```
   var tmVal = Clib.mktime(tm);

   TheApplication().Trace("Time String : " + tmStr);
   TheApplication().Trace("Time Value  : " + tmVal);

   TheApplication().TraceOff();
```

The script produces trace output similar to the following.

```
   03/05/04,12:26:30,START,7.5.3 [16157] LANG_INDEPENDENT,SADMIN,6532,6584
   03/05/04,12:26:30,COMMENT,"Time String : Fri Mar 05 12:26:30 2004"
   03/05/04,12:26:30,COMMENT,Time Value  : 1078489590
   03/05/04,12:26:30,STOP
```

See Also

"Clib.div() Method and Clib.ldiv() Method" on page 164
"Clib.gmtime() Method" on page 190
"Clib.localtime() Method" on page 191
"Clib.mktime() Method" on page 192
"getDate() Method" on page 206
"getTime() Method" on page 213
"getUTCDate() Method" on page 225

Clib.clock() Method

This method returns the current processor tick count.

Syntax

`Clib.clock()`

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.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 187
"Clib.gmtime() Method" on page 190
"Clib.localtime() Method" on page 191
"Clib.tmpnam() Method" on page 158
"toLocaleString() Method and toString() Method" on page 222

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
timeInt1	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.tmpnam() Method" on page 158
"Date.toSystem() Method" on page 206

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, see "Time Objects" on page 186.

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.time())));
```

See Also

"Clib.asctime() Method" on page 187
"Clib.div() Method and Clib.ldiv() Method" on page 164
"Clib.localtime() Method" on page 191
"Clib.mktime() Method" on page 192
"getDate() Method" on page 206
"getTime() Method" on page 213
"getUTCDate() Method" on page 225
"toGMTString() Method" on page 221

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 a Time object, as returned by the time() function.

Usage

This method returns the value *timeInt* (as returned by the time() function) as a Time object. For details on the Time object, see "Time Objects" on page 186.

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 187
"Clib.div() Method and Clib.ldiv() Method" on page 164
"Clib.gmtime() Method" on page 190
"Clib.mktime() Method" on page 192
"getDate() Method" on page 206
"getTime() Method" on page 213
"getUTCDate() Method" on page 225
"toLocaleString() Method and toString() Method" on page 222

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, see "Time Objects" on page 186.

Example

This example shows a use of Clib.mktime in order to format a time so that it can be used with Clib.difftime.

```
// create time object and set time to midnight:
var midnightObject = Clib.localtime(Clib.time());
midnightObject.tm_hour = 0;
midnightObject.tm_min = 0;
midnightObject.tm_sec = 0;

// use mktime to convert Time object to integer:
var midnight = Clib.mktime(midnightObject);
```

```
    // difftime can now use this value:
    var diff = Clib.difftime(Clib.time(), midnight);
    TheApplication().Trace("Seconds since midnight: " + diff);
```

The script produces a trace output similar to this:

```
    COMMENT,Seconds since midnight: 59627
```

See "Clib.asctime() Method" on page 187 for an example that shows the difference between asctime() and mktime() formatting.

See Also

"Clib.asctime() Method" on page 187
"Clib.div() Method and Clib.ldiv() Method" on page 164
"Clib.gmtime() Method" on page 190
"Clib.localtime() Method" on page 191
"getDate() Method" on page 206
"getTime() Method" on page 213
"getUTCDate() Method" on page 225

Clib.strftime() Method

This method creates a string that describes the date, 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, see "Time Objects" on page 186. The conversion characters in the following table are used with Clib.strftime() to indicate time and date output.

Character	Description	Example
%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);
```

The display would be similar to:

```
Today is Friday, and the month is July
```

The following example shows the use of different conversion characters to format the value returned by Clib.strftime.

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");

var tm, tmStrFmt;
tm = Clib.localtime(Clib.time());

Clib.strftime(tmStrFmt, "%m/%d/%Y",tm);
TheApplication().Trace("Time String Format: " + tmStrFmt);

Clib.strftime(tmStrFmt, "%A %B %d, %Y",tm);
TheApplication().Trace("Time String Format: " + tmStrFmt);

TheApplication().TraceOff();
```

The script produces trace output similar to the following.

```
03/05/04,12:44:01,START,7.5.3 [16157] LANG_INDEPENDENT,SADMIN,6848,6708
03/05/04,12:44:01,COMMENT,Time String Format: 03/05/2004
03/05/04,12:44:01,COMMENT,"Time String Format: Friday March 05, 2004"
03/05/04,12:44:01,STOP
```

See Also

"Clib.asctime() Method" on page 187
"Clib.localtime() Method" on page 191

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, see "Clib.div() Method and Clib.ldiv() Method" on page 164, "Clib.difftime() Method" on page 189, "Clib.gmtime() Method" on page 190, "Clib.localtime() Method" on page 191, and "Clib.strftime() Method" on page 193.

See Also

"getDay() Method" on page 207
"Date.toSystem() Method" on page 206
"getDate() Method" on page 206

Clib Object Uncategorized Methods

The following methods are uncategorized:

- "Clib.bsearch() Method" on page 196
- "Clib.getenv() Method" on page 198
- "Clib.putenv() Method" on page 198
- "Clib.qsort() Method" on page 199
- "Clib.system() Method" on page 200

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 be searched
arrayToSort	The name of the array to search
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 parameter and a variable from the array as its second parameter. 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 199

Clib.getenv() Method

This method returns a specified environment variable string.

Syntax

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

Parameter	Description
varName	The name of an environment variable, enclosed in quotes

Returns

The value of the named environment variable.

Usage

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

Example

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

See Also

"Clib.putenv() Method" on page 198

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, enclosed in quotes
stringValue	The value to be assigned to the environment variable, enclosed in quotes

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, a previously existing variable reverts to its pre-script value. A variable created by Clib.putenv() is destroyed automatically.

Example

The following script creates an environment variable and assigns it a value. It then traces the return value to confirm that the variable was created.

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");
var a = Clib.putenv("TEST","test value");
TheApplication().Trace("TEST        : " + a);
TheApplication().Trace("TEST= " + Clib.getenv("TEST");
TheApplication().TraceOff();
```

The script produces the following trace output.

```
03/05/04,16:56:28,START,7.5.3 [16157] LANG_INDEPENDENT,SADMIN,3388,7448
03/05/04,16:56:28,COMMENT,TEST        : 0
03/05/04,16:56:28,COMMENT,TEST= test value
03/05/04,16:56:28,STOP
```

See Also

"Clib.getenv() Method" on page 198

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

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

"Array sort() Method" on page 89

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. An alternative that does not open a window is "SElib.dynamicLink() Method" on page 294.

The *commandString* value may be a formatted string followed by variables according to the rules defined in Table 34 on page 160.

Example

The following code displays a directory in a DOS window.

```
Clib.system("dir /p C:\\Backup");
```

Date Objects

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 this topic, *dateVar* stands for the name of a variable that you create to hold a date value.

See Also

"The Date Constructor in Siebel eScript" on page 202
"Universal Time Methods" on page 223

The Date Constructor in Siebel eScript

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 is 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 simple 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(1802, 6, 23)
```

creates a Date object containing the date July 23, 1802.

Date and Time Methods

Siebel eScript provides the following date and time methods. In addition, there are special date and time methods for working with Universal Time (UTC). For more information, see "Universal Time Methods" on page 223.

- "Date.fromSystem() Static Method" on page 204
- "Date.parse() Static Method" on page 205
- "Date.toSystem() Method" on page 206
- "getDate() Method" on page 206
- "getDay() Method" on page 207
- "getFullYear() Method" on page 208
- "getHours() Method" on page 209
- "getMilliseconds() Method" on page 210
- "getMinutes() Method" on page 210

- “getMonth() Method” on page 211
- “getSeconds() Method” on page 212
- “getTime() Method” on page 213
- “getTimezoneOffset() Method” on page 213
- “getYear() Method” on page 214
- “setDate() Method” on page 215
- “setFullYear() Method” on page 215
- “setHours() Method” on page 216
- “setMilliseconds() Method” on page 216
- “setMinutes() Method” on page 218
- “setMonth() Method” on page 218
- “setSeconds() Method” on page 219
- “setTime() Method” on page 220
- “setYear() Method” on page 221
- “toGMTString() Method” on page 221
- “toLocaleString() Method and toString() Method” on page 222

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

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

“Clib.tmpnam() Method” on page 158
“The Date Constructor in Siebel eScript” on page 202
“Date.toSystem() Method” on page 206
“Time Objects” on page 186

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 *weekday, 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 in Siebel eScript” on page 202

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()`

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, see "getDay() Method" on page 207.

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

"getDay() Method" on page 207

getDate() Method

This method returns the day of the month of a Date object.

Syntax

dateVar`.getDate()`

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 7, the day part of the constructed Date object:

```
function Button2_Click ()
{
   var MyBirthdayDay = new Date("1958", "11", "7");
   TheApplication().RaiseErrorText("My birthday is on day " +
      MyBirthdayDay.getDate() + ".");
}
```

See Also

“getDay() Method” on page 207
“getFullYear() Method” on page 208
“getHours() Method” on page 209
“getMinutes() Method” on page 210
“getMonth() Method” on page 211
“getSeconds() Method” on page 212
“getTime() Method” on page 213
“getYear() Method” on page 214
“setDate() Method” on page 215

getDay() Method

This method returns the day of the week of a Date object.

Syntax

dateVar.getDay()

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 New Year’s Day occurs and displays the result in a message box.

```
function Button1_Click ()
{
   var weekDay = new Array("Sunday", "Monday", "Tuesday",
      "Wednesday", "Thursday", "Friday", "Saturday");
   var NewYearsDay = new Date("2004", "1", "1");
   var theYear = NewYearsDay.getFullYear()
   var i = 0;
   while (i < NewYearsDay.getDay())
```

```
      {
         i++;
         var result = weekDay[i];
      }
      TheApplication().RaiseErrorText("New Year’s Day falls on " + result + " in " +
   theYear + ".");
   }
```

The result displayed in the message box is:

```
New Year’s Day falls on Thursday in 2004.
```

See Also

"getDate() Method" on page 206
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMinutes() Method" on page 210
"getMonth() Method" on page 211
"getSeconds() Method" on page 212
"getTime() Method" on page 213
"getYear() Method" on page 214

getFullYear() Method

This method returns the year of a Date object as a number with four digits.

Syntax

dateVar.getFullYear()

Returns

The year as a four-digit number, of the Date object specified by *dateVar.*

Example

For examples, see "getDay() Method" on page 207, "setMilliseconds() Method" on page 216, and "setTime() Method" on page 220.

See Also

“getDate() Method” on page 206
“getDay() Method” on page 207
“getHours() Method” on page 209
“getMinutes() Method” on page 210
“getMonth() Method” on page 211
“getSeconds() Method” on page 212
“getTime() Method” on page 213
“getYear() Method” on page 214
“setFullYear() Method” on page 215

getHours() Method

This method returns the hour of a Date object.

Syntax

dateVar.getHours()

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 206
“getDay() Method” on page 207
“getFullYear() Method” on page 208
“getMinutes() Method” on page 210
“getMonth() Method” on page 211
“getSeconds() Method” on page 212
“getTime() Method” on page 213
“getYear() Method” on page 214

getMilliseconds() Method

This method returns the milliseconds part of a Date object.

Syntax

dateVar.getMilliseconds()

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 206
"getDay() Method" on page 207
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMinutes() Method" on page 210
"getMonth() Method" on page 211
"getSeconds() Method" on page 212
"getTime() Method" on page 213
"getYear() Method" on page 214

getMinutes() Method

This method returns the minutes portion of a Date object.

Syntax

dateVar.getMinutes()

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 206
"getDay() Method" on page 207
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMonth() Method" on page 211
"getSeconds() Method" on page 212
"getTime() Method" on page 213
"getYear() Method" on page 214

getMonth() Method

This method returns the month of a Date object.

Syntax

dateVar.getMonth()

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 206
"getDay() Method" on page 207
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMinutes() Method" on page 210
"getSeconds() Method" on page 212
"getTime() Method" on page 213
"getYear() Method" on page 214

getSeconds() Method

This method returns the seconds portion of a Date object.

Syntax

dateVar.getSeconds()

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());
```

See Also

"getDate() Method" on page 206
"getDay() Method" on page 207
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMinutes() Method" on page 210
"getMonth() Method" on page 211
"getTime() Method" on page 213
"getYear() Method" on page 214

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()

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 187
"Clib.gmtime() Method" on page 190
"Clib.localtime() Method" on page 191
"Clib.mktime() Method" on page 192
"getDate() Method" on page 206
"getDay() Method" on page 207
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMinutes() Method" on page 210
"getMonth() Method" on page 211
"getSeconds() Method" on page 212
"getYear() Method" on page 214

getTimezoneOffset() Method

This method returns the difference, in minutes, between Greenwich mean time and local time.

Syntax

dateVar.getTimezoneOffset()

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 206
"getDay() Method" on page 207
"getFullYear() Method" on page 208
"getHours() Method" on page 209
"getMinutes() Method" on page 210
"getMonth() Method" on page 211
"getSeconds() Method" on page 212
"getTime() Method" on page 213
"getYear() Method" on page 214

getYear() Method

This method returns the year portion of a Date object as the offset from the base year 1900.

Syntax

dateVar.getYear()

Returns

This method returns the year portion of *dateVar* as the offset from the base year 1900. The offset is positive for years after 1900 and is negative for years before 1900.

Usage

This method returns the year portion of *dateVar* as the offset from the base year 1900. For example, if the value of *dateVar* is a date in the year 2004, then *dateVar*.getYear() = 104.

See Also

"getFullYear() Method" on page 208
"getUTCFullYear() Method" on page 226
"setYear() Method" on page 221

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

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 206
"setUTCDate() Method" on page 230

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

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 208
"setDate() Method" on page 215
"setMonth() Method" on page 218
"setUTCFullYear() Method" on page 230
"setYear() Method" on page 221

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

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 209
"setMilliseconds() Method" on page 216
"setMinutes() Method" on page 218
"setSeconds() Method" on page 219
"setUTCHours() Method" on page 231

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

A date

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 in the computer's clock.

```
function test2_Click ()
{
   var aDate = new Date;
   var milli = 7200000;
   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);
}
```

7200000 milliseconds is two hours, so if you run this routine on November 22, 2005 sometime between 3 and 4 P.M., you get the following result:

```
   The specified date is 11/22/2005 at 5 P.M.
```

See Also

"getMilliseconds() Method" on page 210
"setTime() Method" on page 220
"setUTCMilliseconds() Method" on page 232

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

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 210
"setMilliseconds() Method" on page 216
"setSeconds() Method" on page 219
"setUTCMinutes() Method" on page 233

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

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 211
"setDate() Method" on page 215
"setUTCMonth() Method" on page 234

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

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 212
"setMilliseconds() Method" on page 216
"setUTCSeconds() Method" on page 234

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

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.

```
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);
}
```

Example, if you enter a value of -345650, the result is:

```
   The specified date is 12/31/1969 at 3 P.M.
```

See Also

"getTime() Method" on page 213

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

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 208
"getYear() Method" on page 214
"setFullYear() Method" on page 215
"setUTCFullYear() Method" on page 230

toGMTString() Method

This method converts a Date object to a string, based on Greenwich mean time.

Syntax

dateVar.toGMTString()

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 computer's clock.

```
function clickme_Click ()
{
   var aDate = new Date;
   var milli = 200000;
   aDate.setUTCMilliseconds(milli);
   TheApplication().RaiseErrorText(aDate.toGMTString());
}
```

See Also

"Clib.asctime() Method" on page 187
"toLocaleString() Method and toString() Method" on page 222
"toUTCString() Method" on page 235

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 computer running the script.

Syntax

dateVar.toLocaleString()
dateVar.toString()

Returns

A string representing the date and time of *dateVar* based on the time zone of the computer running the script, 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 computer running the script. If the code is implemented in eScript, then the code runs on a server. The server may or may not be in the same time zone as the user. If the code is implemented in JavaScript, then the code runs on the user's computer and uses that computer's time zone.

Example

This example displays the local time from the computer's clock, the Universal time (UTC), and the Greenwich mean time (GMT).

```
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);
```

The result appears in a message box similar to the following:

```
Local date is Fri Aug 12 15:45:52 2005
UTC date is Fri Aug 12 23:45:52 2005 GMT
GMT date is Fri Aug 12 23:45:52 2005 GMT
```

See Also

“Clib.asctime() Method” on page 187
“Clib.gmtime() Method” on page 190
“Clib.localtime() Method” on page 191
“toGMTString() Method” on page 221
“toUTCString() Method” on page 235

Universal Time Methods

Siebel eScript has methods for both Greenwich mean time (abbreviated GMT) date and time, and for Universal Coordinated Time (abbreviated as UTC). GMT dates and times observe daylight savings time, whereas UTC dates and times do not. 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). UTC is also known as World Time and Universal Time. It is a time standard used everywhere in the world.

Siebel eScript includes the following Date and time functions for working with UTC values.

- “Date.UTC() Static Method” on page 224
- “getUTCDate() Method” on page 225
- “getUTCDay() Method” on page 225
- “getUTCFullYear() Method” on page 226
- “getUTCHours() Method” on page 227
- “getUTCMilliseconds() Method” on page 227
- “getUTCMinutes() Method” on page 228
- “getUTCMonth() Method” on page 228
- “getUTCSeconds() Method” on page 229
- “setUTCDate() Method” on page 230
- “setUTCFullYear() Method” on page 230
- “setUTCHours() Method” on page 231
- “setUTCMilliseconds() Method” on page 232
- “setUTCMinutes() Method” on page 233
- “setUTCMonth() Method” on page 234
- “setUTCSeconds() Method” on page 234
- “toUTCString() Method” on page 235

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, day,* [*, 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(2005, 1, 22, 10, 11, 12));
   TheApplication().RaiseErrorText("The specified date is " +
      aDate.toUTCString());
}
```

A sample run of this code produced the following result.

```
The specified date is Sat Jan 22 10:11:12 2005 GMT
```

See Also

"The Date Constructor in Siebel eScript" on page 202

getUTCDate() Method

This method returns the UTC day of the month of a Date object.

Syntax

dateVar.getUTCDate()

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, 2005 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 206
"setUTCDate() Method" on page 230

getUTCDay() Method

This method returns the UTC day of the week of a Date object.

Syntax

dateVar.getUTCDay()

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, 2005, both locally and in universal time.

```
function Button2_Click ()
{
   var localDay;
   var UTCDay;
   var MayDay = new Date("May 1, 2005 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, 2005, 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 207

getUTCFullYear() Method

This method returns the UTC year of a Date object.

Syntax

dateVar.getUTCFullYear()

Returns

The UTC year of *dateVar* as a four-digit number.

Example

This code fragment displays 2005, the year portion of the date, followed by the GMT equivalent, which may be the same.

```
var aDate = new Date("January 1, 2005 13:24:35");
TheApplication().RaiseErrorText("Local year is " + aDate.getYear() +
   "\nGMT year is " + aDate.getUTCFullYear());
```

See Also

"getFullYear() Method"
"setFullYear() Method" on page 215
"setUTCFullYear() Method" on page 230

getUTCHours() Method

This method returns the UTC hour of a Date object.

Syntax

dateVar.getUTCHours()

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, 2005 13:24:35");
TheApplication().RaiseErrorText("Local hour is " + aDate.getHours() +
   "\nGMT hour is " + aDate.getUTCHours());
```

See Also

"getHours() Method" on page 209
"setUTCHours() Method" on page 231

getUTCMilliseconds() Method

This method returns the UTC millisecond of a Date object.

Syntax

dateVar.getUTCMilliseconds()

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 210
"setUTCMilliseconds() Method" on page 232

getUTCMinutes() Method

This method returns the UTC minute of a Date object.

Syntax

dateVar.getUTCMinutes()

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, 2005 13:24:35");
TheApplication().RaiseErrorText("Local minutes: " + aDate.getMinutes() +
   "\nGMT minutes: " + aDate.getUTCMinutes());
```

See Also

"getMinutes() Method" on page 210
"setUTCMinutes() Method" on page 233

getUTCMonth() Method

This method returns the UTC month of a Date object.

Syntax

dateVar.getUTCMonth()

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, 2005 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 211
"setUTCMonth() Method" on page 234

getUTCSeconds() Method

This method returns the UTC second of a Date object.

Syntax

dateVar.getUTCSeconds()

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 212
"setUTCSeconds() Method" on page 234

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

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

"getUTCDate() Method" on page 225
"setDate() Method" on page 215
"Universal Time Methods" on page 223

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

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.

```
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);
}
```

A sample run of this code produced the following result:

```
In your location, the solstice is on Tue Jun 20, at 6:48 P.M., Standard Time.
```

See Also

"getUTCFullYear() Method" on page 226
"setFullYear() Method" on page 215
"setYear() Method" on page 221
"Universal Time Methods" on page 223

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

Parameter	Description
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

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, see "setUTCFullYear() Method" on page 230.

See Also
"getUTCHours() Method" on page 227
"setHours() Method" on page 216
"Universal Time Methods" on page 223

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

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:

```
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.");
   }
```

When run at 5:36 P.M., Pacific time, on August 22, 2005, it produced the following result:

```
   The specified date is 8/23/2005 at 1:36 UTC time.
```

See Also

"getUTCMilliseconds() Method" on page 227
"setMilliseconds() Method" on page 216
"Universal Time Methods" on page 223

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

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

"getUTCMinutes() Method" on page 228
"setMinutes() Method" on page 218
"Universal Time Methods" on page 223

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

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

"getUTCMonth() Method" on page 228
"setMonth() Method" on page 218
"Universal Time Methods" on page 223

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

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

“getUTCSeconds() Method” on page 229
“setSeconds() Method” on page 219
“Universal Time Methods” on page 223

toUTCString() Method

This method returns a string that represents the UTC date in a convenient and human-readable form.

Syntax

dateVar.toUTCString()

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, see “toLocaleString() Method and toString() Method” on page 222.

See Also

“Clib.asctime() Method” on page 187
“toGMTString() Method” on page 221
“toLocaleString() Method and toString() Method” on page 222

Exception Objects

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.)

The following example shows 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. 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 parameters 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, see "return Statement" on page 237.

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 parameters expected by the function.

Syntax

funcName.length

Parameter	Description
funcName	The function whose length property is to be found

Returns

The number of parameters 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

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
}
```

The following code fragment show 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);
}
```

Global Objects

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

"Conversion Methods" on page 242
"Global Functions Unique to Siebel eScript" on page 238

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 239
- "getArrayLength() Method" on page 240
- "setArrayLength() Method" on page 241
- "undefine() Method" on page 242

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. This method can be executed in server script only; it does not apply to browser script.

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");

// Make Excel visible through the Application object.
ExcelApp.Visible = true;
ExcelApp.WorkBooks.Add();

// 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.ActiveWorkbook.SaveAs (fileName);
```

```
    // Close Excel with the Quit method on the Application object
    ExcelApp.Application.Quit();

    // Clear the object from memory
    ExcelApp = null;
    return (CancelOperation);
```

NOTE: Applications, such as Excel, may change from version to version, requiring you to change your code to match. This example code was tested on Excel 2002.

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 ST eScript engine does not support negative array indices. If you defined arrays with negative indices using the T eScript engine in Siebel Business Applications releases prior to 7.8, then you must redefine their index ranges and any references based on index values.

CAUTION: The getArrayLength() function is unique to Siebel eScript. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"Array Constructors in Siebel eScript" on page 84
"Array length Property" on page 87
"setArrayLength() Method" on page 241

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. **NOTE:** This parameter can be used, but is not meaningful, if you use the ST eScript engine. When using this engine, the minimum index is restricted to zero only, and is assigned by default.
length	The length of the array

Usage

This function sets the length of *array* to a range bounded by *minIndex* and *length*. If three parameters are supplied, *minIndex* 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 parameters are passed to setArrayLength(), the second parameter is length and the minimum index of the newly sized array is 0 by default.

CAUTION: The ST eScript engine does not support negative array indices. If you defined arrays with negative indices using the T eScript engine in Siebel Business Applications releases prior to 7.8, then you must redefine their index ranges and any references based on index values. An alternative to using setArrayLength is to set array lengths with the length property of the Array object.

NOTE: The setArrayLength() function is unique to Siebel eScript. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

For more information on implementing the Siebel scripting engine, see *Using Siebel Tools*.

See Also

"getArrayLength() Method" on page 240
"Array length Property" on page 87

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

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function 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);
```

Conversion Methods

There are times when the types of variables or data should be specified and controlled. Several of the following conversion methods have one parameter, which is a variable or data item, to be converted to the data type specified in the name of the method. For example, the following fragment creates two variables:

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

The first variable, aString, is created by converting the number 123 to a string. The second variable, aNumber, is created by converting the string value "123" to a number. Because aString had already been created with the value "123", the second line could also have been:

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

The remainder of the following methods are used to convert numerical values to various forms or to interpret characters of strings in different ways.

- "escape() Method" on page 243
- "eval() Method" on page 244
- "parseFloat() Method" on page 245
- "parseInt() Method" on page 246
- "ToBoolean() Method" on page 247
- "ToBuffer() Method" on page 248
- "ToBytes() Method" on page 248
- "toExponential() Method" on page 249
- "toFixed() Method" on page 250
- "ToInteger() Method" on page 252
- "ToNumber() Method" on page 253
- "ToObject() Method" on page 254
- "toPrecision() Method" on page 255
- "ToString() Method" on page 256
- "ToUint16() Method" on page 257
- "ToUint32() Method" on page 258
- "unescape(string) Method" on page 259

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 when a string is encoded. Note that the @ and * characters have not been replaced.

```
var str = escape("@#$*96!");
```

Results in the following string: `"@%23%24*96%21"`

```
var encodeStr = escape("@#$*%!");
```

Results in the following string: `"@%23%24*%25%21"`

See Also

"unescape(string) Method" on page 259

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 *expression*. 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. 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
```

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 nonwhite-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 [,radix]*)

Parameter	Description
string	The string to be converted
radix	The radix, or base of the number system, in which the integer return value is expressed; for example, if radix is 8, then the return value is expressed as an octal number.

Returns

An integer number; if *string* cannot be converted to a number, the special value NaN is returned. If *radix* is not provided or is zero, then radix is assumed to be 10, with the following exceptions:

- If *string* begins with the character pairs 0x or 0X, a radix of 16 is assumed.
- If *string* begins with zero and a valid octal digit (0-7), a radix of 8 is assumed.

Usage

White-space characters at the beginning of the string are ignored. The first nonwhite-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.

CAUTION: When the passed string contains a leading zero, such as in “05,” the number is interpreted as on octal, as it is in other eScript contexts. Parameters that are interpreted as invalid octals, such as “08” and “09,” will generate a return value of zero.

Example

The following code fragment returns the result -234:

```
var num = parseInt(" -234.37 profit");
```

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToBuffer() Method" on page 248
"ToObject() Method" on page 254
"ToString() Method" on page 256

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToBytes() Method" on page 248
"ToString() Method" on page 256

ToBytes() Method

This function places its parameter in a buffer.

Syntax

`ToBytes(`*`value`*`)`

Parameter	Description
value	The value to be placed in a buffer

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToBuffer() Method" on page 248
"ToString() Method" on page 256

toExponential() Method

This function returns a number converted to exponential notation with a specified number of decimal places in its mantissa.

Syntax

`numberVar``.toExponential(`*`len`*`)`

Parameter	Description
len	The number of decimal places in the mantissa of the exponential notation to which the number contained in variable *numberVar* is to be converted

Returns

This function returns the number contained in variable *numberVar*, expressed in exponential notation to *len* decimal places. If *len* is less than the number of significant decimal places of *numberVar*, then the function applies standard rounding (round up for 5 or greater, else round down). If *len* is greater than the number of significant decimal places of *numberVar*, then the function pads the extra places with zeroes. If *len* is negative, an error is thrown.

Usage

This function allows you to express numbers in exponential notation with a desired number of decimal places. Exponential notation is generally used to express very large or very small numbers. Because the mantissa of a number expressed in exponential notation is always exactly one digit, controlling the number of decimal places is also a means of controlling the number of significant digits in the number. The justified accuracy of the number may limit the number of significant digits.

Example

The following uses of toExponential() yield the results shown.

```
var num = 1234.567
var num3 = num.toExponential(3) //returns 1.235e+3
var num2 = num.toExponential(0) //returns 1e+3
var num9 = num.toExponential(9) //returns 1.234567000e+3

var smallnum = 0.0001234
var smallnum2 = smallnum.toExponential(2) //returns 1.2e-4
var smallnumerr = smallnum.toExponential(-1) //throws error
```

See Also

“toFixed() Method” on page 250
“toPrecision() Method” on page 255

toFixed() Method

This function returns a number converted to a specified number of decimal places.

Syntax

numberVar.toFixed(*len*)

Parameter	Description
len	The number of decimal places to which the number contained in variable *numberVar* is to be converted

Returns

This function returns the number contained in variable *numberVar*, expressed to *len* decimal places. If *len* is less than the number of significant decimal places of *numberVar*, then the function applies standard rounding (round up for 5 or greater, else round down). If *len* is greater than the number of significant decimal places of *numberVar*, then the function pads the extra places with zeroes. If *len* is negative, an error is thrown.

Usage

This function allows you to express numbers with a desired number of decimal places; for example, to express results of currency calculations with exactly two decimal places.

Example

The following uses of toFixed() yield the results shown.

```
var profits=2487.8235
var profits3 = profits.toFixed(3) //returns 2487.824
var profits2 = profits.toFixed(2) //returns 2487.82
var profits7 = profits.toFixed(7) //returns 2487.8235000
var profits0 = profits.toFixed(0) //returns 2488
var profitserr = profits.toFixed(-1) //throws error
```

See Also

"toExponential() Method" on page 249
"toPrecision() Method" on page 255

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 following 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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToInteger() Method" on page 252
"ToNumber() Method" on page 253
"ToUint16() Method" on page 257
"ToUint32() Method" on page 258

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"Math.round() Method" on page 275
"toFixed() Method" on page 250
"ToNumber() Method" on page 253
"ToString() Method" on page 256
"ToUint16() Method" on page 257
"ToUint32() Method" on page 258

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"Math.round() Method" on page 275
"toFixed() Method" on page 250
"ToInteger() Method" on page 252
"ToString() Method" on page 256
"ToUint16() Method" on page 257
"ToUint32() Method" on page 258

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"ToString() Method" on page 256

toPrecision() Method

This function returns a number converted to a specified number of significant digits.

Syntax

numberVar.toPrecision(*len*)

Parameter	Description
len	The number of significant digits to which the number contained in variable *numberVar* is to be converted

Returns

This function returns the number contained in variable *numberVar*, expressed to *len* significant digits. If *len* is less than the number of significant digits of *numberVar*, then the function applies standard rounding (round up for 5 or greater, else round down) and expression in scientific notation, if necessary. If *len* is greater than the number of significant decimal places of *numberVar*, then the function pads the extra digits with zeroes and adds a decimal point, if necessary.

Usage

This function allows you to express numbers at a desired length; for example, the result of a scientific calculation may only justify accuracy to a specific number of significant digits.

Example

The following uses of toPrecision() yield the results shown.

```
var anumber = 123.45
var a6 = anumber.toPrecision(6) //returns 123.450
var a4 = anumber.toPrecision(4) //returns 123.5
var a2 = anumber.toPrecision(2) //returns 1.2e+2
```

See Also

"toExponential() Method" on page 249
"toFixed() Method" on page 250

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

Example

For an example, see "eval() Method" on page 244.

See Also

"ToBuffer() Method" on page 248
"ToBytes() Method" on page 248

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 in the following 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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

"Math.round() Method" on page 275
"toFixed() Method" on page 250
"ToInteger() Method" on page 252
"ToNumber() Method" on page 253
"ToUint32() Method" on page 258

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 in the following 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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function in a script that may be used with a JavaScript interpreter that does not support it.

See Also

“Math.round() Method” on page 275
“toFixed() Method” on page 250
“ToInteger() Method” on page 252
“ToNumber() Method” on page 253
“ToUint16() Method” on page 257

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"));
```

The code produces the following result.

```
http://obscushop.com/texis/  showcat.html?catid=#2029
rg=r133
```

See Also

“escape() Method” on page 243

Data Handling Methods in Siebel eScript

Use the following eScript methods for various data handling:

- "Blob.get() Method" on page 93
- "Blob.put() Method" on page 95
- "Blob.size() Method" on page 97
- "Blob.get() Method" on page 93
- "escape() Method" on page 243
- "isFinite() Method" on page 262
- "isNaN() Method" on page 261
- "ToString() Method" on page 256
- "undefine() Method" on page 242

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. Before using it, confirm that the JavaScript interpreter that will run the script supports Siebel eScript functions. Avoid using this function 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 242

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 262

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, see "NaN" on page 34.

See Also

"isNaN() Method" on page 261

Math Objects

The Math object in Siebel eScript has a full and powerful set of methods and properties for mathematical operations. A programmer has a set of mathematical tools for the task of doing mathematical calculations in a script.

Methods Supported by the Math Object

The Math Object supports the following methods:

- "Math.abs() Method" on page 263
- "Math.acos() Method" on page 264
- "Math.asin() Method" on page 265
- "Math.atan() Method" on page 265
- "Math.atan2() Method" on page 266

- "Math.ceil() Method" on page 268
- "Math.cos() Method" on page 268
- "Math.exp() Method" on page 269
- "Math.floor() Method" on page 270
- "Math.log() Method" on page 271
- "Math.max() Method" on page 272
- "Math.min() Method" on page 272
- "Math.pow() Method" on page 273
- "Math.random() Method" on page 274
- "Math.round() Method" on page 275
- "Math.sin() Method" on page 276
- "Math.sqrt() Method" on page 277
- "Math.tan() Method" on page 277

Properties of the Math Object

The Math Object has the following properties:

- "Math.E Property" on page 278
- "Math.LN10 Property" on page 279
- "Math.LN2 Property" on page 279
- "Math.LOG10E Property" on page 279
- "Math.LOG2E Property" on page 280
- "Math.PI Property" on page 280
- "Math.SQRT1_2 Property" on page 281
- "Math.SQRT2 Property" on page 281

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 265
"Math.atan() Method" on page 265
"Math.cos() Method" on page 268
"Math.sin() Method" on page 276

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 264
"Math.atan() Method" on page 265
"Math.atan2() Method" on page 266
"Math.cos() Method" on page 268
"Math.sin() Method" on page 276
"Math.tan() Method" on page 277

Math.atan() Method

This method returns an implementation-dependent approximation of the arctangent of the parameter.

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 parameter. 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 266 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 264
"Math.asin() Method" on page 265
"Math.atan2() Method" on page 266
"Math.cos() Method" on page 268
"Math.sin() Method" on page 276
"Math.tan() Method" on page 277

Math.atan2() Method

This function returns an implementation-dependent approximation to the arctangent of the quotient of its parameters.

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 parameters *y* and *x*, where the signs of the parameters are used to determine the quadrant of the result. It is intentional and traditional for the two-parameter arctangent function that the parameter named *y* be first and the parameter 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 265 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 264
"Math.asin() Method" on page 265
"Math.atan() Method" on page 265
"Math.cos() Method" on page 268
"Math.sin() Method" on page 276
"Math.tan() Method" on page 277

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 parameter is already an integer, the result is the parameter 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 270

Math.cos() Method

This method returns an implementation-dependent approximation of the cosine of the parameter. The parameter 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 264
"Math.asin() Method" on page 265
"Math.atan() Method" on page 265
"Math.atan2() Method" on page 266
"Math.sin() Method" on page 276
"Math.tan() Method" on page 277

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 parameter, 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 278
"Math.LN10 Property" on page 279
"Math.LN2 Property" on page 279
"Math.log() Method" on page 271
"Math.LOG2E Property" on page 280
"Math.LOG10E Property" on page 279

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 parameter is already an integer, the result is the parameter itself. It returns NaN if *number* cannot be converted to a number.

Example

For an example, see "Math.ceil() Method" on page 268.

See Also

"Math.ceil() Method" on page 268

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("999^1000 is not greater than 1000^999.");
}
```

See Also

“Math.E Property” on page 278
“Math.exp() Method” on page 269
“Math.LN10 Property” on page 279
“Math.LN2 Property” on page 279
“Math.LOG2E Property” on page 280
“Math.LOG10E Property” on page 279
“Math.pow() Method” on page 273

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 272

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 272

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 271, 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("99^100 is not greater than 100^99.");
}
```

See Also

"Math.exp() Method" on page 269
"Math.log() Method" on page 271
"Math.sqrt() Method" on page 277

Math.random() Method

This function returns a pseudo-random number between 0 and 1.

Syntax

Math.random()

Returns

A pseudo-random number between 0 and 1.

Usage

This function generates a pseudo-random number between 0 and 1. It takes no parameters. 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 167
"Clib.srand() Method" on page 168

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)
```

NOTE: Rounding may not be precise if you multiply or divide a value and then round it. Multiplication and division lead to precision loss.

Example

This code fragment illustrates precision loss due to multiplication.

```
var n = 34.855;
n = n* 100;
var r = Math.round(n)
```

The value of n is 3485.4999999999995 instead of 3485.5. When rounded this results in 3485 instead of 3486.

Example

This code fragment provides a workaround for the loss of precision due to multiplication.

```
var n = parseFloat(34.855);
n = parseFloat(n1b*100.0);
var r = Math.round(n);
```

See Also

"Clib.modf() Method" on page 166
"toFixed() Method" on page 250
"ToInteger() Method" on page 252
"ToUint16() Method" on page 257
"ToUint32() Method" on page 258

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 264
"Math.asin() Method" on page 265
"Math.atan() Method" on page 265
"Math.atan2() Method" on page 266
"Math.cos() Method" on page 268
"Math.tan() Method" on page 277

Math.sqrt() Method

This method returns the square root of its parameter; it returns NaN if x is a negative number or is a value that 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 is a value that cannot be converted to a number.

Usage

This method returns the square root of *number*, or Nan if *number* is negative or is a value that cannot be converted to a number.

See Also

"Math.exp() Method" on page 269
"Math.log() Method" on page 271
"Math.pow() Method" on page 273

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* is a value that 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 264
"Math.asin() Method" on page 265
"Math.atan() Method" on page 265
"Math.atan2() Method" on page 266
"Math.cos() Method" on page 268
"Math.sin() Method" on page 276

Math.E Property

This property stores the number value for *e*, the base of natural logarithms.

Syntax

```
Math.E
```

Usage

The value of *e* is represented internally as approximately 2.7182818284590452354.

See Also

"Math.exp() Method" on page 269
"Math.LN10 Property" on page 279
"Math.LN2 Property" on page 279
"Math.log() Method" on page 271
"Math.LOG2E Property" on page 280
"Math.LOG10E Property" on page 279

Math.LN10 Property

This property stores the number value for the natural logarithm of 10.

Syntax

```
Math.LN10
```

Usage

The value of the natural logarithm of 10 is represented internally as approximately 2.302585092994046.

See Also

"Math.exp() Method" on page 269
"Math.LN2 Property" on page 279
"Math.log() Method" on page 271
"Math.LOG2E Property" on page 280
"Math.LOG10E Property" on page 279

Math.LN2 Property

This property stores the number value for the natural logarithm of 2.

Syntax

```
Math.LN2
```

Usage

The value of the natural logarithm of 2 is represented internally as approximately 0.6931471805599453.

See Also

"Math.E Property" on page 278
"Math.exp() Method" on page 269
"Math.LN10 Property" on page 279
"Math.log() Method" on page 271
"Math.LOG2E Property" on page 280
"Math.LOG10E Property" on page 279

Math.LOG10E Property

The number value for the base 10 logarithm of *e*, the base of the natural logarithms.

Syntax

```
Math.LOG10E
```

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 278
"Math.exp() Method" on page 269
"Math.LN10 Property" on page 279
"Math.LN2 Property" on page 279
"Math.log() Method" on page 271
"Math.LOG2E Property" on page 280

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
```

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 278
"Math.exp() Method" on page 269
"Math.LN10 Property" on page 279
"Math.LN2 Property" on page 279
"Math.log() Method" on page 271
"Math.LOG10E Property" on page 279

Math.PI Property

This property holds the number value for pi.

Syntax

```
Math.PI
```

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, see "Math.atan() Method" on page 265, "Math.atan2() Method" on page 266, and "Math.cos() Method" on page 268.

Math.SQRT1_2 Property

This property stores the number value for the square root of ½.

Syntax

```
Math.SQRT1_2
```

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 277
"Math.SQRT2 Property" on page 281

Math.SQRT2 Property

This property stores the number value for the square root of 2.

Syntax

```
Math.SQRT2
```

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 277
"Math.tan() Method" on page 277

User-Defined Objects in Siebel eScript

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

"Assigning Functions to Objects in Siebel eScript" on page 283
"Object Prototypes in Siebel eScript" on page 284
"Predefining Objects with Constructor Functions in Siebel eScript" on page 282

Predefining Objects with Constructor Functions in Siebel eScript

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. However, the rectangle sally has no motto property.

Assigning Functions to Objects in Siebel eScript

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 in Siebel eScript

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 to be sure that every instance of an object use the same default values and that these instances 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*. Table 40 describes the Property Set object methods.

Table 40. Property Set Object Methods

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.

Table 40. Property Set Object Methods

Method	Description
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.
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.

RegExp Objects

RegExp, or regular expression, object instances are definitions of character patterns and associated attributes that are used to perform character pattern searches of target strings.

RegExp Object Methods

The Siebel ST eScript engine does not support the following static methods of the RegExp object: RegExp.$n (including '$_' and '$&'), RegExp.input, RegExp.lastMatch, RegExp.lastParen, RegExp.leftContext, and RegExp.rightContext. The Siebel T engine does support these methods.

Both the Siebel ST and T eScript engines support the following methods that are documented in this section:

- "RegExp compile() Method" on page 287
- "RegExp exec() Method" on page 288
- "RegExp test() Method" on page 290

Throughout this section, *regexp* is used to represent a RegExp object instance.

RegExp compile() Method

This method changes the pattern and attributes to use with the current instance of a RegExp object.

Syntax

regexp.compile(*pattern*[, *attributes*])

Parameter	Description
pattern	A string with a new regular expression pattern to use with this RegExp object
attributes	A string with the new attributes for this RegExp object. If included, this string must contain one or more of the following characters or be an empty string "": i - sets the ignoreCase property to true g - sets the global property to true m - sets the multiline property to true

Usage

This method allows use of a RegExp instance multiple times with changes to its characteristics.

Use the compile() method with a regular expression that is created with the constructor function, not the literal notation.

Example

```
var regobj = new RegExp("now");
// use this RegExp object
regobj.compile("r*t");
// use it some more
regobj.compile("t.+o", "ig");
// use it some more
```

See also

"RegExp global Property" on page 291

"RegExp ignoreCase Property" on page 292

"RegExp multiline Property" on page 292

"RegExp source Property" on page 293

RegExp exec() Method

This method returns an array of strings that are matches of the regular expression on the target string.

Syntax

regexp.exec(*str*)

Parameter	Description
str	A string on which to perform a regular expression match

Returns

This method returns an array with various elements (the matched strings that are found), and their property sets. The elements returned depend on the attributes of the regular expression. The method returns null if no match is found.

Usage

Of all the RegExp and String methods, RegExp exec() is one of the most powerful because it includes all information about each match in its returned array.

When exec() is executed without the global attribute, "g", being set on the RegExp instance, and a match is found, then:

- Element 0 of the returned array is the first text in the string that matches the primary RegExp pattern.
- Element 1 is the text matched by the first subpattern (in parentheses) of the RegExp instance.
- Element 2 is the text matched by the second subpattern of the RegExp instance, and so forth.

These elements and their numbers correspond to groups in regular expression patterns and replacement expressions.

The returned array includes the following properties:

- The length property is the number of text matches in the returned array.
- The index property is the start position of the first text that matches the primary RegExp pattern.
- The input property is the target string that was searched.

The return values, and the index and input properties are the same as those of the returned array from the String match() Method when match() is used on a regular expression whose global attribute is not set.

When exec() is executed with the global attribute, "g", set on the RegExp instance, and a match is found, then:

- The same results are returned as when the global attribute is not set, but the behavior is more complex, which allows further operations.

  NOTE: Although exec() and the String match() method provide the same return arrays when the global attribute is not set on the regular expression, exec() and match() return different arrays when the global attribute is set on the regular expression.

- Searching begins at the position in the target string specified by this.lastIndex. After a match is found, this.lastIndex is set to the position after the last character in the text matched. The property this.lastIndex is read/write and may be set at anytime, so you can loop through a string and find all matches of a pattern by setting this.lastIndex to the start position of the previous match found + 1. When no match is found, this.lastIndex is reset to 0.

If you use the T eScript engine and any matches are found, appropriate RegExp object static properties, such as RegExp.leftContext, RegExp.rightContext, RegExp.$n, and so forth are set, providing more information about the matches.

NOTE: The ST eScript engine does not support the following static properties of the RegExp object: RegExp.$n (including '$_' and '$&'), RegExp.input, RegExp.lastMatch, RegExp.lastParen, RegExp.leftContext, RegExp.rightContext.

Examples

The following example calls exec() from a regular expression whose global attribute is not set. The output is commented.

```
function fn ()
{
   var myString = new String("Better internet");
   var myRE = new RegExp(/(.).(.er)/i);
   var results = myRE.exec(myString);
   var resultmsg = "";
   for(var i =0; i < results.length; i++)
   {
      resultmsg = resultmsg + "return[" + i + "] = " + results[i] + "\n";
   }
   TheApplication().RaiseErrorText(resultmsg);
}
fn ();
```

Output is:

```
return[0] = etter   \\First text containing primary pattern ...er (any three
                    \\characters followed by "er")
return[1] = e        \\First text matching the first subpattern (.) (any single
                    \\character) within the first text matching the primary pattern
return[2] = ter      \\First text matching the second subpattern (.er) (any single
                     \\character followed by "er") within the first text matching
                     \\the primary pattern
```

The following example calls exec() from a regular expression whose global attribute is set. The method returns all matches of the regular expression’s primary pattern in a string, including matches that overlap.

```
function fn ()
{
   var str = "ttttot tto";
   var pat = new RegExp("t.t", "g");
   var resultmsg = "";
   while ((rtn = pat.exec(str)) != null)
   {
      resultmsg = resultmsg + "Text = " + rtn[0] + " Pos = " + rtn.index
      + " End = " + (pat.lastIndex - 1) + "\n";
      pat.lastIndex = rtn.index + 1;
   }
   TheApplication().RaiseErrorText(resultmsg)
}
fn ();
```

Output is:

```
Text = ttt Pos = 0 End = 2
Text = ttt Pos = 1 End = 3
Text = tot Pos = 3 End = 5
Text = t t Pos = 5 End = 7
```

See also

"RegExp test() Method" on page 290

"String match() Method" on page 306

RegExp test() Method

This method indicates whether a target string contains a regular expression pattern.

Syntax

regexp.test(*str*)

Parameter	Description
str	A string on which to perform a regular expression match

Returns

This method returns true if the target string contains the regular expression pattern, else it returns false.

Usage

This method is equivalent to *regexp*.exec(*str*)!=null.

If you use the T eScript engine and there is a match, then appropriate RegExp object static properties, such as RegExp.leftContext, RegExp.rightContext, RegExp.$n, and so forth are set, providing more information about the matches.

NOTE: The ST eScript engine does not support the following static properties of the RegExp object: RegExp.$n (including '$_' and '$&'), RegExp.input, RegExp.lastMatch, RegExp.lastParen, RegExp.leftContext, RegExp.rightContext.

Although not common, test() may be used in a special way when the global attribute, "g", is set on the RegExp instance. As with RegExp exec(), when a match is found, the lastIndex property of the RegExp instance is set to the character position after the found text match. Thus, test() may be used repeatedly on a string, for instance, to determine whether a string has more than one match or to count the number of matches.

For information about using the RegExp lastIndex property repeatedly on a string, see "RegExp exec() Method" on page 288.

Example

```
var str = "one two three tio one";
var pat = /t.o/;
rtn = pat.test(str);
// Then rtn == true.
```

See also

"RegExp exec() Method" on page 288

"String match() Method" on page 306

RegExp Object Properties

The Siebel ST and Siebel T eScript engines support the following RegExp Object properties.

Throughout this section, *regexp* is used to represent a RegExp object instance.

- "RegExp global Property" on page 291
- "RegExp ignoreCase Property" on page 292
- "RegExp multiline Property" on page 292
- "RegExp source Property" on page 293

RegExp global Property

This read-only property indicates the value of the global attribute of an instance of the RegExp object.

Syntax

regexp.global

Usage

This property has a value of true if "g" is an attribute of the regular expression pattern being used, else its value is false.

NOTE: The global attribute of a RegExp instance can be changed with the RegExp compile() method.

Example

```
    // Create RegExp instance with global attribute.
    var pat = /^Begin/g;
    //or
    var pat = new RegExp("^Begin", "g");
    //Then pat.global == true.
```

See also

"RegExp compile() Method" on page 287

RegExp ignoreCase Property

This read-only property indicates the value of the ignoreCase attribute of an instance of the RegExp object.

Syntax

regexp.ignoreCase

Usage

This property has a value of true if "i" is an attribute of the regular expression pattern being used, else its value is false.

NOTE: The ignoreCase attribute of a RegExp instance can be changed with the RegExp compile() method.

Example

```
    // Create RegExp instance with ignoreCase attribute.
    var pat = /^Begin/i;
    //or
    var pat = new RegExp("^Begin", "i");
    //Then pat.ignoreCase == true.
```

See also

"RegExp compile() Method" on page 287

RegExp multiline Property

This read-only property indicates the value of the multiline attribute of an instance of the RegExp object.

Syntax

regexp.multiline

Usage

This property has a value of true if "m" is an attribute of the regular expression pattern being used, else its value is false. The multiline property determines whether a pattern search is done in a multiline mode.

NOTE: The multiline attribute of a RegExp instance can be changed with the RegExp compile() method.

Example

```
// Create RegExp instance with multiline attribute.
var pat = /^Begin/m;
//or
var pat = new RegExp("^Begin", "i");
//Then pat.multiline == true.
```

See also

"RegExp compile() Method" on page 287

RegExp source Property

This read-only property stores the regular expression pattern being used to find matches in a string, not including the attributes.

Syntax

regexp.source

Usage

This read-only property stores the regular expression pattern being used to find matches in a string, not including the attributes.

NOTE: The source attribute of a RegExp instance can be changed with the RegExp compile() method.

Example

```
var pat = /t.o/g;
// Then pat.source == "t.o"
```

See also

"RegExp compile() Method" on page 287

SElib Objects

In Siebel eScript, the SElib object allows calling out to external libraries and applications.

SElib.dynamicLink() Method

This method calls a procedure from a dynamic link library (Windows) or shared object (UNIX).

Windows Syntax

`SElib.dynamicLink(`*`Library, Procedure, Convention`*`[`*`, [desc,] arg1, arg2, arg3, ..., arg`*`n])`

UNIX Syntax

`SElib.dynamicLink(`*`Library, Procedure`*`[`*`, arg1, arg2, arg3, ...arg`*`n])`

NOTE: On UNIX, the total number of parameters passed with SElib.dynamicLink() must not exceed 22. These 22 parameters include the shared library name and the procedure name, so you can pass up to 20 additional parameters.

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	Parameters to the procedure

Usage

The calling convention must be one of the following:

Value	Description
CDECL	Push right parameter first; the caller pops parameters
STDCALL	Push right parameter first; the caller pops parameters (this value 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 95 and "Clib.fwrite() Method" on page 152.

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. The defined method *score* is called by SElib dynamicLink in the subsequent example code.

```
#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);
```

The following example shows how to call an external application and pass it arguments (0, 0, and 5):

```
SElib.dynamicLink("shell32", "ShellExecuteA", STDCALL, 0, "open",
"c:\\Grabdata.exe", 0, 0, 5).
```

See also

“Clib.system() Method” on page 200

SElib.peek() Method

This method reads data from a specific position in memory.

Syntax

`SElib.peek(`*`address`*`[, `*`dataType`*`])`

Parameter	Description
address	The address in memory from which to get data, that is, a pointer to the data in memory.
dataType	The type of data to get, from among the following types: UWORD8, SWORD8, UWORD16, SWORD16, UWORD24, SWORD24, UWORD32, SWORD32, FLOAT32, FLOAT64, FLOAT80 (not available in Win32) For each type, the numerical suffix, for example 8 or 16, specifies the number of bytes to get. The “S” or “U” prefix on some types designates “signed” or “unsigned,” respectively. The default value is UWORD8.

Returns

This method returns the data specified by *dataType*.

Usage

This method reads (or gets) data from the position in memory to which the *address* argument points. The *dataType* parameter specifies how many bytes to read and how to interpret the data.

CAUTION: Routines that work with memory directly should be used with caution. To avoid destroying or moving data unexpectedly, you should clearly understand memory and the operations of these methods before using them.

Example

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");
var v = new Buffer("Now");
// Collect "Now", the original value, for display.
TheApplication().Trace(v);
// Get the address of the first byte of v, "N"
var vPtr = SElib.pointer(v);
// Get the "N"
var p = SElib.peek(vPtr);
// Convert "N" to "P"
SElib.poke(vPtr,p+2);
// Display "Pow"
TheApplication().Trace(v);
TheApplication().TraceOff();
```

The script produces the following trace output:

```
COMMENT,Now
COMMENT,Pow
```

See also

"SElib.poke() Method" on page 299

"Blob.get() Method" on page 93

"Clib.memchr() Method" on page 118

"Clib.fread() Method" on page 146

SElib.pointer() Method

This method gets the address in memory of a Buffer variable.

Syntax

SElib.pointer(*bufferVar*])

Parameter	Description
bufferVar	The name or identifier of a Buffer variable

Returns

This method returns the address of (a pointer to) the Buffer variable identified by *varName*.

Usage

This method gets the address in memory of the first byte of data in a Buffer variable. For information on the Buffer object, see "Buffer Objects in Siebel eScript" on page 98.

CAUTION: A pointer is valid only until a script modifies the variable identified by *bufferVar* or until the variable goes out of scope in a script. Putting data in the memory occupied by *bufferVar* after such a change is dangerous. When data is put into the memory occupied by *bufferVar*, be careful not to put more data than will fit in the memory that the variable actually occupies.

Example

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");
var v = new Buffer("Now");
// Collect "Now", the original value, for display.
TheApplication().Trace(v);
// Get the address of the first byte of v, "N"
var vPtr = SElib.pointer(v);
// Get the "N"
var p = SElib.peek(vPtr);
// Convert "N" to "P"
SElib.poke(vPtr,p+2);
// Display "Pow"
TheApplication().Trace(v);
TheApplication().TraceOff();
```

The script produces the following trace output:

```
COMMENT,Now
COMMENT,Pow
```

See also

"SElib.peek() Method" on page 296
"SElib.poke() Method" on page 299
"BLOB Objects" on page 91
"Clib.memchr() Method" on page 118

SElib.poke() Method

This method writes data to a specific position in memory.

Syntax

SElib.poke(*address*, *data*[, *dataType*])

Parameter	Description
address	The address in memory to which to write data, that is, a pointer to the position in memory in which to start writing the data.
data	The data to write directly to memory. The data should match the type given by *dataType*.
dataType	The type of data to write, from among the following types: UWORD8, SWORD8, UWORD16, SWORD16, UWORD24, SWORD24, UWORD32, SWORD32, FLOAT32, FLOAT64, FLOAT80 (not available in Win32) For each type, the numerical suffix, for example 8 or 16, specifies the number of bytes to write. The "S" or "U" prefix on some types designates "signed" or "unsigned," respectively. The default value is UWORD8.

Returns

This method returns the address of the byte that follows the data that is written to memory.

Usage

This method writes data to the position in memory to which the *address* argument points. The data to be written must match the type given by the *dataType* argument, or its default value if not provided. The *dataType* argument specifies how many bytes to write and how to interpret the data.

CAUTION: Routines that work with memory directly should be used with caution. To avoid destroying or moving data unexpectedly, you should clearly understand memory and the operations of these methods before using them.

Example

```
TheApplication().TraceOn("c:\\eScript_trace.txt","allocation","all");
var v = new Buffer("Now");
// Collect "Now", the original value, for display.
TheApplication().Trace(v);
// Get the address of the first byte of v, "N"
var vPtr = SElib.pointer(v);
// Get the "N"
var p = SElib.peek(vPtr);
// Convert "N" to "P"
SElib.poke(vPtr,p+2);
// Display "Pow"
TheApplication().Trace(v);
TheApplication().TraceOff();
```

The script produces the following trace output:

```
COMMENT,Now
COMMENT,Pow
```

See also

"SElib.peek() Method" on page 296
"Blob.put() Method" on page 95
"Clib.memchr() Method" on page 118
"Clib.fread() Method" on page 146

String Objects

One of the properties of the String object is its value, a sequence of text characters. Like other objects, the String object has other properties and methods.

Throughout this section, "string" is used to represent the value of an instance of the String object, that is, a sequence of characters. Typically, other properties of the String object are attributes that describe the string value, and methods of the String object manipulate the string value.

To indicate that a text literal is a string, it is enclosed in quotation marks. In the following example, the first statement puts the string "hello" into the variable word. The second sets the variable word to have the same value as the variable hello.

```
var word = "hello";
word = hello;
```

You can declare a string with single quotes instead of double quotes. There is no difference between the two in eScript.

See Also

"Escape Sequences for Characters in Siebel eScript" on page 301
"String Object Methods and Properties in Siebel eScript" on page 302

Escape Sequences for Characters in Siebel eScript

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 indication 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 table lists the characters indicated by escape sequences.

Escape Sequence	Description
\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)

These escape sequences cannot be used within strings enclosed by back quotes, which are explained in “Back-Quote Strings in Siebel eScript” on page 301.

Back-Quote Strings in Siebel eScript

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; that is, the escape characters are part of a string. Special characters represented by a back slash 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.

String Object Methods and Properties in Siebel eScript

The following conventions are used in the methods and properties in this topic:

- *stringVar* indicates any string variable. A specific instance of a variable should precede the period to use a property or call a method.
- The identifier *String* indicates a static method of the String object. It does not apply to a specific instance of the String object.

String 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, where the position of the first character in the string is 0.

Returns

A string of length 1 representing the character at *position*.

Usage

To get the first character in a string, use index 0, as follows:

```
var string1 = "a string";
var firstchar = string1.charAt(0);
```

To get the last character in a string, use:

```
var lastchar = 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 indexOf() Method” on page 303
“String lastIndexOf() Method” on page 304
“String length Property” on page 305
“String.fromCharCode() Static Method” on page 302

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, ... coden*)

Parameter	Description
code1, code2, ... coden	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 parameters 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, see "offset[] Method" on page 102.

```
var seblStr = String.fromCharCode(83, 105, 101, 98, 101, 108);
```

See Also

"Clib.toascii() Method" on page 128

String 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 be searched
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 for 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";
var firsta = 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

"String charAt() Method" on page 302
"Clib.strchr() Method" on page 175
"Clib.strpbrk() Method" on page 181
"String lastIndexOf() Method" on page 304
"String replace() Method" on page 310

String lastIndexOf() Method

This method finds the position of the last occurrence of a substring in a string.

Syntax

`stringVar.lastIndexOf(substring [, offset])`

Parameter	Description
substring	One or more characters to search for
offset	The rightmost position in the string at which to start searching. If *offset* is not provided, the entire string is searched.

Returns

If *offset* is provided, the function returns the rightmost position, not greater than *offset*, at which *substring* begins in the string contained in the variable *stringVar*. If *offset* is not provided, the function returns the rightmost position in the entire string at which *substring* begins.

If *substring* is not found, or if *offset* is outside the range of valid positions in the string, then the function returns -1.

Usage

The *stringVar*.lastIndexOf() function is used to determine the last position within a string that a substring occurs. By setting the offset parameter, the search can be limited to a substring of leftmost characters of the string.

Substring is not required to occur entirely within the substring of the string bounded by *offset*. Its first character is required to occur at a position no greater than *offset*.

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 lastabr = magicWord.lastIndexOf("abr", 8);
```

returns 7, the position of the last “abr” beginning at a position no greater than 8.

See Also

“String charAt() Method” on page 302
“Clib.strchr() Method” on page 175
“Clib.strpbrk() Method” on page 181
“String indexOf() Method” on page 303
“String replace() Method” on page 310

String length Property

The length property stores an integer indicating the length of the string.

Syntax

`*stringVar*.length`

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.");
```

String match() Method

This method returns an array of strings that are matches within the string against a target regular expression.

Syntax

stringVar.match(*regexp*)

Parameter	Description
regexp	A regular expression, provided as a literal or as a variable

Returns

This method returns an array with various elements (the matched strings that are found), and their property sets. The elements returned depend on the attributes of the regular expression. The method returns null if no match is found.

Usage

When match() is executed with the global attribute, "g", not set on the regular expression, then the return array and its properties are equivalent to those returned under the same circumstances using the RegExp exec() method. If a match is found, then:

- Element 0 of the returned array is the first text in the string that matches the primary RegExp pattern.
- Element 1 is the text matched by the first subpattern (in parentheses) of the RegExp instance.
- Element 2 is the text matched by the second subpattern of the RegExp instance, and so forth.

These elements and their numbers correspond to groups in regular expression patterns and replacement expressions.

The returned array includes the following properties:

- The length property is the number of text matches in the returned array.
- The index property is the start position of the first text that matches the primary RegExp pattern.
- The input property is the target string that was searched.

The return values, and the index and input properties are the same as those of the returned array from the RegExp exec() method when exec() is used with a regular expression whose global attribute is not set.

When match() is executed with the global attribute, "g", set on the regular expression, and a match is found, then:

- Element 0 of the return array is the first text in the string that matches the primary pattern of the regular expression.

- Each subsequent element of the return array is the next text in the string that matches the primary pattern of the regular expression, and that starts after the last character of the previous match. Thus matches that overlap other matches are not returned. For example, if the regular expression's primary pattern is a.. ("a" followed by any two characters) and the string is "abacadda", then the return array includes "aba" and "add", but not "aca".

  NOTE: Although match() resembles the RegExp exec() method when the global attribute is not set on the regular expression, match() and exec() are very different when the global attribute is set on the regular expression.

Examples

The following example calls match() against a regular expression whose global attribute is not set. The output is commented.

```
function fn ()
{
   var myString = new String("Better internet");
   var myRE = new RegExp(/(.).(.er)/i);
   var results = myString.match(myRE);
   var resultmsg = "";
   for(var i =0; i < results.length; i++)
   {
      resultmsg = resultmsg + "return[" + i + "] = " + results[i] + "\n";
   }
   TheApplication().RaiseErrorText(resultmsg);
}
fn ();
```

Output is:

```
return[0] = etter   \\First text containing primary pattern ...er (any three
                    \\characters followed by "er")
return[1] = e        \\First text matching the first subpattern (.) (any single
                    \\character) within the first text matching the primary pattern
return[2] = ter      \\First text matching the second subpattern (.er) (any single
                     \\character followed by "er") within the first text matching
                     \\the primary pattern
```

The following example calls match() against a regular expression whose global attribute is set. The method returns matches of the regular expression's primary pattern that do not overlap.

```
function fn ()
{
   var str = "ttttot tto";
   var pat = new RegExp("t.t", "g");
   var rtn = str.match(pat);
   var resultmsg = "";
   for(var i =0; i < rtn.length; i++)
   {
      resultmsg = resultmsg + "match [" + i + "] = " + rtn[i] + "\n";
   TheApplication().RaiseErrorText(resultmsg);
   }
}
fn ();
```

Output is:

```
match [0] = ttt
match [1] = tot
```

The output does not include the “ttt” instance that starts at position 1 or “t t” because these instances start within other strings that are returned.

See also

“RegExp exec() Method” on page 288

String search () Method

This method searches a specified string for a specified regular expression pattern.

Syntax

stringVar.search(*regexp*)

Parameter	Description
regexp	A regular expression, provided as a literal or as a variable

Returns

The index of the regular expression (*regexp*) inside the specified string. If the regular expression is not found, this method returns -1.

Usage

This method can be executed in server script and browser script.

Example

```
function Test(sValue)
{
   //Validate for 5 digit numbers
     var sCheck = /^\d{5}$/;  //regular expression defining a 5 digit number
     if(sValue.search(sCheck)==0)
  {
        return("Valid");
     }
     else
     {
        return("Invalid");
  }
}
```

String 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. The string has to be modified with escape characters to be comprehensible to Siebel eScript. Also, the cmdLine variable must appear on a single line, which space does not permit in this volume.

```
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

"Array join() Method" on page 86

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

stringVar.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 you are using the T eScript engine and 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.

NOTE: The ST eScript engine does not support the following static properties of the RegExp object: RegExp.$n (including '$_' and '$&'), RegExp.input, RegExp.lastMatch, RegExp.lastParen, RegExp.leftContext, RegExp.rightContext.

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.
$&	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;

}
```

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, see "String indexOf() Method" on page 303.

See Also

"String charAt() Method" on page 302
"String indexOf() Method" on page 303
"String lastIndexOf() Method" on page 304

toLowerCase() Method

This method returns a copy of a string with the letters changed to lowercase.

Syntax

stringVar.toLowerCase()

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 313

toUpperCase() Method

This method returns a copy of a string with the letters changed to uppercase.

Syntax

stringVar.toUpperCase()

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 312

A Compilation Error Messages in Siebel eScript

This appendix provides explanations and examples of error messages generated by Siebel eScript when a script is compiled with the ST eScript engine. This appendix contains the following topics:

Conventions Used in This Appendix

The following conventions are used in this appendix:

- The error prefix is the text that appears for all of a group of errors; for example, "Syntax error at Line *line#* position *character#*:".
- The message is the unique part of an error message that applies only to a single error. The message may be text appended after an error prefix, or it may be the entire error message.
- In each example, comment text explains the flawed script that it follows. Not all errors have associated examples.
- A cause is provided for some, but not all, errors. Typically, the example suffices to explain causes of an error.

Syntax Error Messages in eScript

Table 41 on page 316 contains error messages that can result from incorrect script syntax when the script is compiled with the ST eScript engine.

Syntax error messages start with the error prefix “Syntax error at line *line#* position *character#*:”.

Table 41. Syntax Error Messages in Siebel eScript

Message	Examples	Cause

Expected ':'	`1.`
	`function main ()`
	`{`
	` var a = false;`
	` var b = a ? 1, 2;`
	` //expect : after 1`
	`}`
	
	`2.`
	`function main ()`
	`{`
	` var a = {prop1:1, prop2};`
	` //expect : after prop2`
	`}`
	
	`3.`
	`function main ()`
	`{`
	` var a = 1;`
	` var b;`
	` switch (a)`
	` {`
	`  case 1`
	`  //expect : after 1`
	`   b =a;`
	`  default`
	`  //expect : after default`
	`   b = 0;`
	` }`
	`}`	

Expected ';'	`function main ()`
	`{`
	` for (i=1; i<10)`
	` //miss ; after i<10`
	` {`
	` ...`
	` }`
	`}`	

Expected '('	`function main <>`
	`//expect ( after main`
	`{`
	` ...`
	`}`	

Table 41. Syntax Error Messages in Siebel eScript

Message	Examples	Cause
Expected ')'.		(and) do not pair up.

Expected ']'.	`function main ()`
	`{`
	` var a = new Array (10);`
	` a[10 = 1;`
	` //expect ] after a[10 = 1`
	`}`	

Expected '{'.	`function main ()`
	`var a = new Array (1);`
	`//expect { before var`	

Expected '}'.		{ and } do not pair up.

Expected identifier.	`function ()`
	`// expect an identifier after`
	`// function */`
	`{`
	` ...`
	`}`
	
	`function main ()`
	`{`
	` var;`
	`//expect an identifier after var`
	`}`	

Invalid token.	`function main ()`
	`{`
	` var a = "\u000G";`
	` // '\u000G' is an invalid`
	` // unicode escape sequence`
	`}`
	
	`function main ()`
	`{`
	` var a = "\u0G";`
	` // '\u0G' is an invalid hex`
	` // escape sequence`
	`}`
		There is an invalid unicode escape sequence or an invalid hex escape sequence.

Expected while.	`function main ()`
	`{`
	` do`
	` {`
	`  ...`
	` }`
	` //expect while on this line`
	`}`	

Table 41. Syntax Error Messages in Siebel eScript

Message	Examples	Cause
Throw must be followed by an expression on the same line.		

Invalid continue statement.	`function main ()`
	`{`
	` continue;`
	` // continue is not within a loop`
	`}`
		The continue statement is not within the body of:
		■ do...while
		■ while
		■ for
		■ for...in

Invalid break statement.	`1.`
	`function BreakError()`
	`{`
	` break;`
	`// break is not within a valid`
	`// loop`
	`}`
		The break statement is outside of the body of:
		■ do...while
		■ while
		■ for
		■ for...in

Invalid return statement. Return statement cannot be used outside the function body.	`function fn ()`
	`{`
	` ….`
	`}`
	`return;`
	`//Return is outside the function`
	`//body.`	

Invalid left-hand side value.	`function main ()`
	`{`
	` new Object () = 1;`
	` // new Object () is not a valid`
	` // left value`
	`}`	

Invalid regular expression.	`var oRegExp:RegExp;`
	`oRegExp = /[a-c*/;`
	`// The regular expression is`
	`// missing the closing ]. It`
	`// should be [a-c]*.`	

Semantic Error Messages in eScript

Table 42 on page 319 contains error messages that can result from semantic errors when the script is compiled with the ST eScript engine.

See also "Semantic Warnings in eScript" on page 322.

Semantic error messages start with the error prefix “Semantic Error around line *line#*:”.

Table 42. Semantic Error Messages in Siebel eScript

Message	Examples	Cause

Argument *argument_label* either type does not correct or is not defined.	`function main ()`
	`{`
	`fn (new Date(), new Date());`
	`// type of the second parameter`
	`// mismatches with function`
	`// definition and cannot be`
	`// implicitly converted to`
	`// 'Number' type`
	`}`
	
	`function fn (arg1: chars, arg2:`
	`Number)`
	`{`
	` TheApplication().RaiseErrorText`
	` ("fn");`
	`}`
	`main ();`	

No such predefined property *property_label* in class *object_type*.	`function main ()`
	`{`
	` delete "123".prop1;`
	` // prop1 is not a property of`
	` // String object. Also, because`
	` // the String object is`
	` // constructed here by implicitly`
	` // converting "123", prop1`
	` // cannot be created dynamically.`
	`}`	

[] operator can only apply to Object, Buffer or Array class.		The script is trying to use [] accessor to types other than Object, Buffer or Array

Table 42. Semantic Error Messages in Siebel eScript

Message	Examples	Cause

Type mismatch: L: *left_type*; R: *right_type.*	`1.`
	`function TypeMismatch()`
	`{`
	` var BC:BusComp;`
	` var MyDate:Date = new Date();`
	` BC =MyDate;`
	`// MyDate is not the same data type`
	`// as strongly typed variable BC`
	`}`
	
	`2.`
	`function fn ()`
	`{`
	` var a: String;`
	` a = new Date ();`
	` //Type mismatch: strongly typed`
	` //String is assigned a Date.`
	`}`
		A value which belongs to one data type is assigned to a strongly typed variable of another data type.

Return type is wrong. Defined return type is *return_type*.	`function fn (): Array`
	`{`
	` return new Date ();`
	`}`
	
	`fn ();`
		The actual return type is different form the defined return type, and the actual return type cannot be implicitly converted to the defined type.

No such label *label* defined.	`function fn ()`
	`{`
	` break labl;`
	` // where labl is not a valid label`
	`}`
	
	`fn ();`	

Continue out of loop.	`function ContinueOut()`
	`{`
	` var i =0`
	` while (i<3)`
	` {`
	`  i++;`
	`  continue Mylabel;`
	`  // Mylabel label is defined`
	`  //outside of the while loop.`
	` }`
	` Mylabel:`
	` var a=1;`
	`}`
		A continue command attempts to branch to a label that is outside of a loop.

Table 42. Semantic Error Messages in Siebel eScript

Message	Examples	Cause

Label redefined.	`function LabelError()`
	`{`
	` Outer:`
	`  for (var i = 0; i < 5; i++)`
	`  {`
	`   var j = 0;`
	`   Inner:`
	`    while (j!=5)`
	`    {`
	`     j++;`
	`     continue Inner;`
	`     Inner: //Label Inner is`
	`            //redefined.`
	`      var b=1;`
	`    }`
	`  }`
	`}`
		There is already an existing label with the same name.

function *function_label* is double defined.	`function fn ()`
	`{`
	` TheApplication().RaiseErrorText`
	` ("fn");`
	`}`
	`function fn ()`
	`// second declaration of function`
	`// fn is not allowed`
	`{`
	` TheApplication().RaiseErrorText`
	` ("fn again");`
	`}`	

Calling function *function_label* with insufficient number of arguments.	`function main ()`
	`{`
	` fn ();`
	` // does not provide enough`
	` //parameters`
	`}`
	
	`function fn (arg1: chars, arg2:`
	`chars)`
	`{`
	` ...`
	`}`	

Table 42. Semantic Error Messages in Siebel eScript

Message	Examples	Cause

Cannot access property *property_name* on native type.	`function main ()`
	`{`
	` var a:chars = "123";`
	` a.m_prop = "123";`
	` // chars is a primitive type, so it`
	` // has no properties`
	`}`
	`main ();`	

Object_name is an invalid object type.	`function main ()`
	`{`
	` var a: Obj1 = "123";`
	` // where Obj1 is an invalid object`
	` // type`
	
	`}`
	`main ();`	

Indiscriminate usage of goto.	`function main ()`
	`{`
	` var obj = new Object();`
	` with (obj)`
	` {`
	`  labl:`
	`  TheApplication().RaiseErrorText`
	`  ("in with");`
	` }`
	`goto labl;`
	`}`
	`main ();`
		Script uses goto to attempt a branch into a with block from outside of the with block.

Semantic Warnings in eScript

Typically, semantic warnings make you aware of script that will run, but may produce unexpected results or may be inefficient. Semantic warnings do not display during compilation. Instead, view them in Oracle’s Siebel Tools product by choosing Debug > Check Syntax.

Table 43 on page 323 contains semantic warnings in eScript when the script is compiled with the ST eScript engine. See also “Semantic Error Messages in eScript” on page 318.

Semantic warnings start with the prefix “Semantic Warning around line *line#*:”.

Table 43. Semantic Warnings in Siebel eScript

Message	Example	Cause

Undefined identifier *identifier*. Global object will be used to locate the identifier.	`function main ()`
	`{`
	` obj = new Object();`
	` // obj is created without being`
	` // declared with var.`
	`}`
	`main ();`
		An undeclared variable created within a function is not locally defined. Instead, it is created as a property of the Global object.

Variable *variable* might not be initialized.	`function main ()`
	`{`
	` var a;`
	` TheApplication().RaiseErrorText`
	` (a);`
	`}`
	`main ();`	

Label '*label*' is unused and can be removed.	`function main ()`
	`{`
	` var a = 1;`
	` labl:`
	` // labl is unused`
	` TheApplication().RaiseErrorText`
	` (a);`
	`}`
	`main ();`	

Calling function *function_label* with insufficient number of arguments.	`function main ()`
	`{`
	`// It is a warning condition`
	`// instead of an error if the`
	`// missing argument is not`
	`// strongly typed.*/`
	` var c = fn ();`
	`}`
	
	`function fn (a, b)`
	`{`
	` return a+b;`
	`}`
	`main ();`	

Type conversion from *data_type1* to *data_type2* may not succeed.	`function main ()`
	`{`
	`var n: float = "123";`
	`}`	

Table 43. Semantic Warnings in Siebel eScript

Message	Example	Cause

No such method *method_name*.	`function main ()`
	`{`
	`fn ();`
	`}`
	`main ();`	

variable *variable* is double declared.	`1.`
	`function fn ()`
	`{`
	` for (var n = 0 ; n < 3 ; n++)`
	` {`
	`  ...;`
	` }`
	` for (var n = 0 ; n < 3 ; n++)`
	` // n is double declared within  /`
	`/ the scope of fn.`
	` {`
	`  ...;`
	`}`
	`fn ();`
	
	`2.`
	`function main ()`
	`{`
	` var string1 = "a string";`
	` var string1 = “another string”;`
	` // string1 should not be`
	`redeclared.`
	`}`
	`main ();`
		A local variable is declared more than once.
		
		To avoid this warning for the common case in Example 1, declare the counter variable outside of the for definition and use the counter variable without var in the for definition. For example:
		
		`function fn ()`
		`{`
		`var n;`
		`for (n = 0 ; n < 3 ; n++)`
		`{`
		`...`
		`}`
		`for (n = 0 ; n < 3 ; n++)`
		`{`
		`...`
		
		The multiple declarations like that in Example 2 have the net effect of all declarations after the first declaration being interpreted as simple assignments, but with the unnecessary overhead of variable declarations. Instead, use simple assignments after the first declaration; for example:
		
		`string1 = “another string”.`

Preprocessing Error Messages in eScript

Preprocessing error messages typically indicate compatibility issues when script created with the T eScript engine is compiled with the ST eScript engine.

Table 44 on page 325 contains preprocessing error messages when the script is compiled with the ST eScript engine.

Preprocessing error messages start with the error prefix “PreProcess Error:”.

Table 44. Preprocessing Error Codes in Siebel eScript

Message	Example	Cause
Cannot open include file *file_path*.	`#include "mystuff.js"` `//where mystuff.js does not exist`	The path to the file in an include statement is not valid.

Index