ScriptEase 4.20

Copyright Notice

Technical documentation by Ronald Terry Constant

Table of Contents

Table of Contents 3
Introduction 9
    ScriptEase Desktop 9
    ScriptEase package 10
ScriptEase JavaScript 13
    Basics of ScriptEase 14
    Identifiers 17
    Data types 21
    Automatic type conversion 27
    Properties and methods of basic data types 27
    Operators 28
    Flow decisions statements 34
    Exception handling 40
    Functions 42
    Objects 49
    Dynamic objects 55
Preprocessor 59
    Variables in the environment 59
    Preprocessor Directives 61
    Executing a script 66
    ScriptEase shell command-line 66
    Running a script 68
    Command-line switches 72
Predefined Values 75
    Preprocessor values 75
    Predefined constants and values 77
Quick Start Tutorial 81
    ScriptEase Shell 81
    Simple script 82
    Date and time display 82
    Function with parameters 84
    Terminology 86
    Function with a return 87
    Screen.write improved 88
    Library file 91
    Library and sample files 94

Using library files 95
Integrated Debugger 99
Using the ScriptEase Debugger 100
Main menu bar 106
ScriptEase versus C language 113
Data types in C and SE 114
Automatic type declaration 114
Array representation 115
Automatic array allocation 115
Literal strings 116
Structures 119
Passing variables by reference 120
Pointer operator * and address operator & 122
Case statements 122
Initialization code which is external to functions 123
Unnecessary tokens 123
Macros 124
Token replacement macros 124
Back quote strings 125
Converting existing C code to ScriptEase 125
Array object 127
Creating arrays 128
Array object instance properties 130
Array object instance methods 131
Blob Object 141
Blob object static methods 141
blobDescriptor object 145
Number Object 147
Number object instance methods 147
SElib Object 149
SElib object static methods 149
Buffer Object 187
Buffer object instance properties 187
Buffer object instance methods 189
Screen Object 197
Screen object static methods 197
String Object 205
String as data type 205
String as object 207

String object instance properties ... 208
String object instance methods ... 208
String object static methods ... 216
RegExp Object ... 217
RegExp object instance methods ... 217
Object Object ... 219
Object object instance methods ... 219
Math Object ... 221
Math object static properties ... 221
Math object static methods ... 223
Global object ... 233
Conversion or casting ... 233
global object properties ... 234
global object methods/functions ... 235
Function Object ... 249
Function object instance methods ... 249
Dos Object ... 253
Dos object static methods ... 253
Clib Object ... 259
Console I/O functions ... 259
Time functions ... 268
Script execution ... 274
Error ... 277
File I/O ... 279
Directory ... 293
Sorting ... 295
Environment variables ... 298
Character classification ... 299
String manipulation ... 303
Memory manipulation ... 322
Math ... 324
Variable argument lists ... 335
Unix Object ... 341
Unix object static methods ... 341
Boolean Object ... 347
Boolean object instance methods ... 347
Date Object ... 349
Date object instance methods ... 351
Date object static methods ... 364

Link Libraries 367
UUCode Link Library 369
  UU object static methods 369
DSP Link Library 371
  DSP Object 371
  Creating a DSP object 371
  Using a DSP object 372
  DSP object instance methods 374
  DSP object static properties 380
GD Link Library 381
  GD Object 381
MD5 Checksum Link Library 405
  md5 Object 405
  md5 object instance methods 405
SEDBC Link Library 407
  Cursor Object 407
  Database Object 416
  SimpleDataset Object 427
  Stproc Object 440
Socket Link Library 447
  Socket Object 447
  Socket object instance methods 447
  Socket object static methods 454
Com Object Link Library 459
  Com Object 459
Script Libraries 461
  Common script libraries 462
  Common utility and sample scripts 464
  Win32 script libraries 464
  Win32 utility and sample scripts 466
Appendix B Instance and Static Notation 467
  String instance properties 468
  String instance methods 468
  String static properties 468
  String static methods 469
  Prototype property 469
Appendix A Grouped Functions 471
  Routines for arrays 471
  Routines for Buffers 471

Routines for character classification 472
Routines for console I/O 472
Routines for conversion/casting 473
Routines for data/variables 473
Routines for date/time 474
Routines for diagnostic/error 475
Routines for directory, file, and OS 475
Routines for display control 476
Routines for execution control 476
Routines for file/stream I/O 477
Routines for math 477
Routines for memory manipulation 479
Routines for miscellaneous 479
Routines for strings/byte arrays 480
Routines for variable argument lists 481
Index 482

Introduction

Welcome to **ScriptEase**, the exciting world of scripting. ScriptEase is built on **JavaScript**, which is perhaps the most popular scripting language in the world, and is supported by C script, which is based on the most popular and powerful programming language in the world. The guiding principles for the development of ScriptEase are: **simplicity**, **power**, and **safety**. Prepare to take control of your computer and to guide your computer destiny through the use of simple and powerful scripts written in ScriptEase.

The foundation of ScriptEase is JavaScript which emphasizes simplicity and flexibility. But what about the C scripting? Do you have to know the C language? No, you do not need to know C to use ScriptEase, but the power of C is available to you anytime you want to take advantage of it. Many scripts are written without using C at all. Some programmers, especially those from a C background, write scripts using C script also. The choice is yours.

ScriptEase Desktop

ScriptEase Desktop, which is part of the Nombas product line of scripting solutions, is designed to be used on individual computers, whether alone or in networks. ScriptEase not only allows you to control your computer, but you may control computers on a network through powerful networking capabilities. Even more, ScriptEase Distributed Scripting Protocol allows a user on one computer in a network, including the Internet, to run scripts on and control another computer on the network. But, enough of previews into the awesome power available to you in ScriptEase Desktop. The emphasis here is on simplicity for average computer users, though experienced and professional programmers appreciate the power of ScriptEase that allows them to accomplish more tasks in less time.

Most users want simple scripting ability so that they may control their computers instead of their computers controlling them. ScriptEase allows you to take control of your computer life and not be completely at the mercy of the whims and mistakes of commercial programmers. Consider WordPad, the standard word processor that comes with Windows95/98. Every time a user clicks the WordPad icon, another instance of WordPad starts. Some users become frustrated. They start an instance of WordPad to work on a document, not remembering that they already have WordPad running with that same document in it. They lose work as they save new changes over the other document. The result is lost work and frustration.

Some people like WordPad's ability to run multiple instances, but others dislike it. For those who do not like to run multiple instances, ScriptEase provides a solution. A simple script of a few lines can make sure that only one instance of WordPad is run at a time. You are in control.

Perhaps you have important data that you want backed up in a special way. A ScriptEase script allows you to accomplish the task easily. Are these the only ways to use ScriptEase. Of course not! The only real limitation is your imagination and desire.

ScriptEase package

The ScriptEase Desktop package that you have received may be broken down into the following categories.

The interpreter

The interpreter is comprised of the executable files that are often as the interpreter. The interpreter is the main program that interprets script or program files. Scripts are plain text files that are normally written using any text editor. For versions that support the integrated debugger, scripts may be written and debugged in the debugger.

The debugger

Some versions of ScriptEase support the use of the Integrated Debugging Environment (IDE). The IDE is a powerful source code debugger that allows you to execute code using up-to-date features such as trace, stepping, break points, and watches. These features are implemented through menus, shortcut keys, and a multiple document interface (MDI), which means you can debug a script using several windows to view the debugging process.

Library files

ScriptEase comes with many library files that have useful and powerful routines for use in your own scripts. These library files can be easily included in your scripts giving you instant access to routines such as the dialog routines in dialog.jsh. When ScriptEase is installed, these library files are installed with appropriate information provided to your computer system to use them immediately and easily.

Prewritten routines

ScriptEase comes with many scripts that are complete and useful programs. An example of such a file is deltree.jsh which allows you to delete an entire directory tree, including files. Another example is filecomp.jse which allows you to compare two files. Like library files, these prewritten routines put into place during installation with appropriate information provided to your computer system to use them immediately and easily.

Sample files

Many sample files are installed with ScriptEase. These sample files range from being examples and tutorials for writing scripts to being complete and use programs.

Documentation

Up to the minute information and documentation about ScriptEase and installed files is kept in documentation files that come in various formats such as text, rtf, and HTML. Further documentation, information about platform specific issues, and information about installation are included as printed material accompanying the main manual.

Installation

The installation procedure is self-explanatory and prompts for any information that it needs. However, printed installation instructions are found in the printed material that accompanies the main manual.

ScriptEase JavaScript

ScriptEase is a scripting or programming language that allows a computer user or programmer to write simple scripts with tremendous power. The guiding principles for ScriptEase are **simplicity** and **power** which add up to easy elegance in scripting. Scripts are much easier to write and use than the source code for compiled languages such as C++.

ScriptEase uses JavaScript, one of the most popular scripting language in today's world, as its core language. In fact, ScriptEase uses the ECMAScript standard for JavaScript. ECMAScript is the core version of JavaScript which has been standardized by the European Computer Manufacturers Association and is the only standardization of JavaScript. ScriptEase closely follows and will follow this standardized JavaScript.

ScriptEase is not limited to JavaScript, as good as it may be. ScriptEase has enhanced the power of JavaScript by adding two objects, Clib and SElib, that have the power of the C programming language. Indeed, ScriptEase implements a scripting version of C which has the power of C in a simple scripting language. With the power of C readily available, computer users or programmers are able to accomplish any tasks that they pursue. Both JavaScript and C script can be intermingled in ScriptEase code, which allows scripters flexibility, power, and simplicity.

The following line is a complete script which could be saved as a script file and run as a program. The program simply displays a message, "A simple one line script," on a computer screen

```
Screen.writeln("A simple one line script")
```

The following code fragment uses a more structured approach to accomplish the same task. JavaScript and C share similar programming styles, such as the main() function shown in this fragment.

```
function main()
{
   Clib.puts("A simple one line script");
}
```

A ScriptEase script may be written using a very straightforward scripting approach as shown in the first example above, which is similar to the simple scripting of a DOS batch file. A second line could be added to the single line as shown in the following fragment.

```
Screen.writeln("A simple one line script")
Clib.puts("Now there are two lines")
```

The example using the main() function could be expanded as follows.

```
function main()
{
   Clib.puts("A simple one line script");
   Screen.writeln("Now there are two lines");
}
```

These examples illustrate how easily ScriptEase can be used in a simple scripting mode and how easily the power of functions can be put in a script, and not just the power of functions, but the power of C. They show how easily JavaScript and C script can be intermingled, since C is implemented as a JavaScript object. Functions and other programming concepts are explained in the following descriptions of the ScriptEase language. A tutorial section provides illustrations of scripts in addition to the example code fragments in the text.

Most JavaScript, other than ScriptEase, is part of web browsers and is used while users are connected to the Internet. Usually people are unaware that JavaScript is commonly being executed on their computers when they are connected to various Internet sites. Not only are they unaware, they are unable to write and execute scripts on their computers for their own uses. ScriptEase steps in at this point. ScriptEase Desktop is designed for users to control their own computers in a stand alone mode. Users do not have to be connected to the Internet to use ScriptEase, as they must be with other JavaScript interpreters.

Whether the desire is to write a simple script to copy a document to a backup folder or to write an entire data processing program, ScriptEase can do the job or any other job desired. ScriptEase has joined JavaScript and C. Further, ScriptEase adds commands and functions not available in standard implementations of either. In short, ScriptEase is the most powerful and advanced scripting language available today, and it achieves its power while still being simple to use.

The following sections of this manual will help you to start enjoying the power of ScriptEase.

Basics of ScriptEase

Case sensitivity

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

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

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

White space characters

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

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

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

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

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

Comments

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

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

The following code fragments are examples of valid comments:

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

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

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

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

Expressions, statements, and blocks

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

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

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

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

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

Statements within blocks are often indented for easier reading.

Identifiers

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

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

The following identifiers, variables and functions, are valid:

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

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

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

Prohibited identifiers

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

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

Variables

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

```
Screen.writeln("Rumpelstiltskin Henry Constantinople")
```

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

```
var Name = "Rumpelstiltskin Henry Constantinople"
Screen.write(Name)
```

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

Variable scope

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

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

```
var perfectNumber;
```

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

```
var perfectNumber = 28;
```

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

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

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

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

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

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

Function identifier

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

Function scope

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

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

Screen.writeln(SumTwo(3, 4))

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

```
Screen.writeln(global.SumTwo(3, 4))
```

Data types

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

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

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

In the fragment above which defines and uses the function SumTwo(), the literals, 3 and 4, are passed as arguments to the function SumTwo() which has corresponding parameters, a and b. The parameters, a and b, are variables for the function the hold the literal values that were passed to it.

Data types need to be understood in terms of their literal representations in a script and of their characteristics as variables.

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

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

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

Primitive data types

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

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

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

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

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

Number type

Integer

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

Decimal

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

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

Hexadecimal

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

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

Octal

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

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

Floating point

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

Decimal

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

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

Scientific

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

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

Boolean type

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

String type

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

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

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

Composite data types

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

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

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

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

The composite data types are: Object and Array.

Object type

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

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

Array type

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

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

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

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

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

Special values

undefined

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

```
var test;
if (test == undefined)
   Screen.writeln("test is undefined")
```

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

null

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

```
var test;
if (test == null)
   Screen.write("test is undefined")
```

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

```
var test = null;
```

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

NaN

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

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

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

Number constants

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

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

`Number.NEGATIVE_INFINITY`	`- Infinity`	Number below `MIN_VALUE`

Automatic type conversion

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

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

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

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

Properties and methods of basic data types

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

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

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

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

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

toString()

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

valueOf()

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

Operators

Object operator

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

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

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

The `Math.abs()` method is a static method, that is, it is used directly with the Math object instead of an instance of the object. Many methods are instance methods, that is, they are used with instances of an object instead of the object itself. The `substring()` method is an instance method of the String object. An instance method is not used with an object itself but only with instances of an object. The `substring()` method is never used with the String object as

`String.substring()`. The following fragment declares and initializes a string variable, which is an instance of the string object, and then uses the substring() method with this instance by using the object operator.

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

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

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

Mathematical operators

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

Basic arithmetic

The arithmetic operators in ScriptEase are pretty standard.

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

The following are examples using variables and arithmetic operators.

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

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

Notice that:

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

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

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

but has a different meaning than:

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

which is still different from:

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

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

Assignment arithmetic

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

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

The following lines are examples using assignment arithmetic.

```
var i;
i  = 2;          i is now  2
```

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

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

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

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

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

Bit operators

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

<<	shift left	`i = i << 2;`
<<=	assignment shift left	`i <<= 2;`
>>	shift right	`i = i >> 2;`
>>=	assignment shift right	`i >>= 2;`
>>>	shift left with zeros	`i = i >>> 2`
>>>=	assignment shift left with zeros	`i >>>= 2`
&	bitwise and	`i = i & 1`
&=	assignment bitwise and	`i &= 1;`
|	bitwise or	`i = i | 1`

`|=`	assignment bitwise or	`i |= 1;`
`^`	bitwise xor, exclusive or	`i = i ^ 1`
`^=`	assignment bitwise xor, exclusive or	`i ^= 1`
`~`	Bitwise not, complement	`i = ~i;`

Logical operators and conditional expressions

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

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

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

```
var guess = GetTheGuess(); //get the user input
if (guess > target_number)
{
   ...guess is too high...
}

if (guess < target_number)
{
   ...guess is too low...
}

if (guess == target_number)
```

```
{
   ...you guessed the number!...
}
```

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

The logical operators are:

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

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

instanceof operator

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

```
var s = new String("abcde");
Screen.writeln(s instanceof String);   // Displays true
```

The second line could also be written as:

```
Screen.writeln(s instanceof(String));
```

typeof operator

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

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

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

Flow decisions statements

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

if

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

```
if ( goo < 10 )
```

```
{
   Screen.write("goo is smaller than 10\n");
}
```

else

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

```
if ( goo < 10 )
{
   Screen.write("goo is smaller than 10\n");
}
else
{
   Screen.write("goo is not smaller than 10\n");
}
```

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

```
if ( goo < 10 )
{
   Screen.write("goo is less than 10\n");
   if ( goo < 0 )
   {
      Screen.write("goo is negative; so it's less than 10\n");
   }
}
else if ( goo > 10 )
{
   Screen.write("goo is greater than 10\n");
}
else
{
   Screen.write("goo is 10\n");
}
```

while

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

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

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

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

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

do {...} while

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

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

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

```
do
{
   var name = GetNameFromTheList();
   SendEmail(name)
} while (name != TheLastNameOnTheList());
```

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

for

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

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

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

```
for(var x=1; x<11; x++)
{
   Screen.write(x);
}
```

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

```
for( ; ThereAreUncalledNamesOnTheList() ; )
{
   var name=GetNameFromTheList();
   SendEmail(name)
}
```

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

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

break

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

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

The break statement is also used at the close of a case statement, as shown below.

continue

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

switch, case, and default

The switch statement makes a decision based on the value of a variable or statement. The switch statement follows the following format:

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

...

default:
   default_statement
}
```

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

For example, suppose you had a series of account numbers, each beginning with a letter that determines what type of account it is. You could use a switch statement to carry out actions depending on that first letter. The same task could be accomplished with a series of nested if statements, but they require much more typing and are harder to read.

```
switch ( key[0] )
{
case 'A':
   Screen.write("A"); //handle 'A' accounts...
   break;
case 'B':
   Screen.write("B"); //handle 'B' accounts...
   break;
case 'C':
   Screen.write("C"); //handle 'C' accounts...
   break;
default:
   Screen.write("Invalid account number.\n");
   break;
}
```

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

Normally, if a switch and series of case statements reference array variables, then a comparison is performed whether or not the reference the same array data. But if either the switch variable or one of the case values is a literal string, then the comparison of the strings is done using the values of the strings in a .strcmp() type comparison.

goto and labels

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

```
goto LABEL;
```

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

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

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

Conditional operator

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

```
test_expression ? expression_if_true : expression_if_false
```

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

The following fragment illustrates the use of the conditional operator.

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

Exception handling

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

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

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

Remember these execution guides:

- When a `throw` statement executes, the rest of the code in a function is ignored, and the function does not return a value.
- A program continues in the next `catch` statement block after the `try` statement block in which an exception occurred., and any value thrown is caught in a parameter in the catch statement.
- A program executes a `finally` statement block if all exceptions, that have been thrown, have been caught and handled.

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

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

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

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

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

```
function main(argc, argv)
{
   var rtn;

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

   Screen.write("Paused..."); Clib.getch();
} //main

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

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

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

Functions

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

Screen.write() is an example of a function which provides an easy way to display formatted text. It receives a string from the function that called it and displays the string on the screen. Screen.write is a void function, meaning it has no return value.

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

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

Several sets of built- in functions are included as part of the ScriptEase interpreter. These functions are described in this manual. They are internal to the interpreter and may be used at any time. In addition, ScriptEase ships with a number of external libraries or .jsh files. External libraries must be explicitly included in your script to use the functions in them. See the description of the #include preprocessor directive.

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

Function return statement

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

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

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

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

This script displays12.

Passing information to functions

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

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

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

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

```
var num1 = 4;
var num2 = 4;
var num3;
SetNumbers(num1, num2, num3, 6)

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

After executing this code, the values of variables is:

```
n1 == 5
n2 == 4
n3 == 5
```

The variable num1 was passed by reference to parameter n1. When n1 was set to 5, num1 was actually set to 5 since n1 merely pointed to num1. The variable num2 was passed by value to parameter n2. When n2, which received an actual value of 4, was set to 5, num2 remained unchanged. The variable num3 was undefined when passed by reference to parameter n3. When n3, which pointed to num3, was set to 5, num3 was actually set to 5 and defined as an integer type. The literal value 6 was passed to parameter n4, but not by reference since 6 is not a variable that can be changed. Though n4 has an ampersand, the literal value 6 was passed by value to n4 which, in this example, becomes merely a local variable for the function SetNumbers().

Passing information to cfunctions

All variables passed as arguments to the parameters of cfunctions are passed by reference. If the cfunction called alters a parameter variable, the original variable from where the cfunctions was called is actually altered. Since cfunction parameters receive values by reference, they only point to the original variables, and thus, any changes made to them are made to the original variables.

Function property arguments[]

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

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

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

Function recursion

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

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

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

Error checking for functions

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

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

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

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

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

You may abort a script in such a case, but at least you will know why. See the section on the Clib object.

main() function

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

The `main()` function automatically receives two parameters, which, by convention, are called argc and argv. The parameter argc, argument count, is the number of parameters passed to the script and the parameter argv is an array of strings, with each element being one of the parameters. The first element, argv[0], of this array is always the name of the script, thus if argc == 1, then no variables were passed to a script.

Arguments are passed to a script as parameters when it is called from a command line as illustrated in the following line.

```
sewin32.exe jseedit.jse document.txt
```

In the example above, argc == 2, argv[0] == "jseedit.jse" and argv[1] == "document.txt".

cfunction keyword

The cfunction keyword defines a function whose behavior is somewhat different than that of standard functions. In a cfunction, variables and operators behave more as they would in C, specifically in the ScriptEase implementation of C as a scripting language. The cfunction is provided for the convenience of C programmers who are used to the way the C language handles functions and variables and for those situations in which the underlying logic of C is more efficient for a particular procedure.

Strings are treated as null- terminated character arrays. The first character of a string is assigned to string[0], the second to string[1], and so on until the end of the string. The last character of a string is always '\0', which defines the end of the string. If you assign a variable to a string, using double quotes, the '\0' character is automatically appended to the end of the string. To assign a string to a variable without appending the '\0' character, put the string in single quotes. Single quotes are most often used with single characters. For example, if

```
var boy = "m";
var girl = 'm';
```

the variable boy is a character array in which: boy[0] = 'm' and boy[1] = '\0'. The variable girl is a character, containing the letter 'm'. Internally, characters are converted to numbers according to the ASCII standard.

You can change the contents of strings or parts of them by assigning a new character value to a element of a character array. For example:

```
var string = "file"
string[0] = 'b'
```

This fragment creates a string containing the word "bile".

Array arithmetic

If you try to add a number to a string, instead of converting the number to a string and concatenating the two, the starting point of the string will be shifted forward by the number of characters in number. For example, the statement:

```
"This is a test" + 3
```

evaluates to "This is a test3", in a standard JavaScript. In a cfunction, however, this statement evaluates to "s is a test". The starting point of the string has been shifted by three, so that string[0] is now 's' instead of 'T'. The 'T', 'h', and 'i' of the original string are at indices [- 3], [- 2], and [- 1], respectively.

Variables passed to cfunctions are passed by reference. In other words, if you have two variables:

```
var George = "one"
var Martha = "one"
```

and you compare them with the == operator, the comparison evaluates to false and not to true, as you might expect. The reason is that while George and Martha have the same value, they are not the same variable since they point to different memory locations, and therefore are not equal to each other. In functions declared with the function keyword, string variables are compared by value, so the actual values of George and Martha are compared. In such cases the result of comparing identical strings with == comparison is true.

Objects

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

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

```
var Rectangle;

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

Screen.write(Rectangle.height * Rectangle.width);
```

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

Predefining objects with constructor functions

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

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

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

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

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

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

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

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

Initializers for objects and arrays

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

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

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

Arrays may initialized using a syntax similar to the following:

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

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

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

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

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

The following code fragment shows the differences.

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

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

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

The display from this code is:

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

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

Methods - assigning functions to objects

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

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

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

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

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

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

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

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

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

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

creates an object class Rectangle with the rectangle_area method included as one of its properties. The method is available to any instance of the class:

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

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

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

Object prototypes

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

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

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

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

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

Rectangle.prototype.area = rectangle_area;
```

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

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

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

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

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

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

for/in

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

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

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

with

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

```
with (Clib)
{
   printf("I am a camera");
   srand();
   xxx = rand() % 5;
   putchar(xxx);
}
```

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

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

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

Dynamic objects

ScriptEase allows for direct access to the interior workings of how object properties are called. If you wish, you may specify how an object accesses its data by replacing one of the following routines which are internal to ScriptEase. The following methods are available for modifying how an object calls its members. In all cases, the parameter, property, is the name of the property being called.

_get(property)

Whenever the value of a property is accessed, the `_get()` method is called. By defining a new `_get()` method for an object, you modify the way it accesses property values. If the new `_get()` method has no return value, the value that the function would normally return is returned.

The example below modifies the Rectangle object created earlier with a new `_get()` method. Whenever you access the value of one of the object's properties, it will inform you if the Rectangle is a square. After the object is initialized, the `main()` function creates an instance of the object with the width and height properties both set to 3. When the value of the `Rectangle.area()` method is retrieved, used in a `Clib.printf()` statement, the dynamic `_get()` function is called, which displays, "The rectangle is a square," since width and height are equal. Since no value is returned from the dynamic `_get()` function, the value normally returned, 9 in this case, is returned to the `main()` function.

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

function rectangle_get()
{
   if (this.width == this.height)
      Clib.printf("The rectangle is a square.");
}

function Rectangle(width, height)
```

```
{
   this.width = width;
   this.height = height;
   this._get = rectangle_get;
}

Rectangle.prototype.area = rectangle_area;

main()
{
   var rect = new Rectangle(3, 3);
   Clib.printf("The area of the rectangle is %d.", rect.area());
   Clib.getch();
}
```

_put(property, value)

This method controls the way that new data is assigned to a property.

_canPut(property)

This method returns a boolean value indicating whether the property can be written to or not, that is, whether it is read- only or not. For example, you could modify this property to notify users when they try to change read- only values.

_hasProperty(property)

This method returns a boolean value indicating whether or not a property exists.

_delete(property)

This method is called whenever a property is deleted with the delete operator.

_defaultValue(hint)

This method returns the primitive value of a variable.

The parameter hint should be either a string or a number that indicates the preferred data type to return. If hint is a string, the method will return a string if possible, otherwise a different type. The actual value of hint is ignored.

_construct(...)

This method is called whenever a new object is created with the new operator. The object will have been already created and passed as the this variable to the .construct() method.

_call(...)

The call function is called whenever an object method is called. Whatever parameters are passed to the original function will be passed to the call() function.

The following example creates an Annoying object that beeps whenever it retrieves the value of a property.

```
function myget(prop)
{
   System.beep();
   return this[property];
}

var Annoying = new Object;

Annoying.get = myget;
```

Note that the System.beep() method is used only for this example and must be explicitly created for actual use.

Preprocessor

ScriptEase programs, such as sewin32.exe, read, preprocess, interpret, and execute scripts. A ScriptEase executable program is sometimes referred to as the processor, interpreter, or engine. There is a different version of the processor for each operating system that ScriptEase supports. Each version interprets ScriptEase code in a manner appropriate to its operating system. For example, the .directory() function parses a DOS directory differently than a Unix directory because of differences in the operating systems. But, the end result for the user is the same.

The term ScriptEase is used generically for all versions of ScriptEase. The names of executable programs for each operating system are different, for example:

- **Win32**
  Sewin32.exe
  Secon32.exe
- **Win16**
  Sewin16.exe
- **DOS**
  Sedos.exe
  Sedos32.exe
- **OS/2**
  Seos2.exe

Many of the scripts that ship with ScriptEase work with any version, but some scripts work only with specific versions or operating systems. When ScriptEase is installed, the scripts are placed into directories indicating which versions they work with.

This section describes environment variables and preprocessor directives that affect the processing of a ScriptEase script prior to finally compiling, tokenizing, and executing the script. The description then covers the sequence of events when a script is executed, the ScriptEase shell, stand, finally, command-line switches.

Variables in the environment

The environment variables covered in this section are those which are important when a script is being preprocessed. Other environment variables that ScriptEase uses are covered in the sections most appropriate to them.

SEDESKPATH

SEDESKPATH is an environment variable that the processor uses to search for scripts and libraries. It functions like a PATH variable for the processor. When looking for scripts and libraries, the processor first searches the current directory and then the directory that has the ScriptEase executable. If the file needed is not found, it searches through the directories specified by the SEDESKPATH variable. If the needed files are still not found, the processor looks for them in the regular PATH variable.

In Windows systems, the processor searches the SEDESKPATH profile value in win.ini (in Windows 3.x) or the Registry (in Windows 95/98 and NT) before searching the SEDESKPATH environment variable.

PATH

The PATH variable is used to find ScriptEase files if they are not found in the current directory or in the directories of SEDESKPATH.

SEDESKPREFS

SEDESKPREFS is an environment variable that should be set before loading and executing a script. SEDESKPREFS allows you to specify one or more files that the interpreter will automatically load and run every time a script is started. The files specified may be files with the extensions, .jsh, .jse, or any other file extensions that a programmer chooses to use. SEDESKPREFS takes a form similar to a PATH environment variable. Multiple files are separated by semicolons. An example setting for SEDESKPREFS is:

```
SEDESKPREFS=C:\SEDESK\JSH\GLBLS.JSH;C:\SEDESK\GENERAL.JSE
```

The two files in this example are files that a programmer might choose to create and are not standard files distributed with ScriptEase. Glbls.jsh might have various definitions, globals variables, and so forth that a programmer uses regularly, and General.jse might have several functions that are regularly used. Every time these files are needed by a script, they must be included with code similar to the following.

```
#include "Glbls.jsh"
#include "General.jse"
```

If SEDESKPREFS, as shown above, exists in the environment, then no scripts need to contain the include the statements shown immediately above.

SE_ESET

Two Clib methods `Clib.getenv()` and `Clib.putenv()` allow you to retrieve and set values or system environment variables. Operating systems that do not allow direct modification of environment variables (such as 32-bit Windows and OS/2) use the environment variable, SE_ESET, to hold the name of a file used to change the system=s environment variables indirectly.

Preprocessor Directives

The following ScriptEase statements that begin with a # character are collectively called preprocessor directives, since they are processed before a script is actually executed and direct the way the script commands are interpreted. Preprocessor directives can only be used with the ScriptEase interpreter. Other JavaScript interpreters will not recognize them.

define

The #define directive is used to replace a token or almost any identifier with other characters. The #define directive is executed while the script is being read into the interpreter, before the script itself is executed. The #define directive causes one string to be replaced by another in the script that goes to the interpreter. All substitutions are made before the code is interpreted. A #define directive has the following structure.

```
#define token replacement
```

This line results in all subsequent occurrences of "token" being replaced by "replacement". Consider the following line.

```
#define NumberOfCountriesInSouthAmerica 13
```

The define statement increases program legibility and makes it easier to change code later. If Bolivia and Peru decide someday to unite, you only have to change the #define statement to update your program. Otherwise, you would have to go through your script looking for all occurrences of the number 13, decide when they refer to the number of countries in South America, and change them to the number 12.

Likewise, if you write screen routines for a 25 line monitor, and then later decide to make it a 50 line monitor, you are better off altering the following #define directive from:

```
#define ROW_COUNT 25
```

to

```
#define ROW_COUNT 50
```

and using ROW_COUNT in your code. You only have to make one change in your script instead of many.

The ScriptEase interpreter has default tokens that are defined, such as true and false.

include

The #include directive lets you include other scripts, and all of the functions contained therein, as a part of the code you are writing. Usually #include lines are placed at the beginning of the script and consist only of the #include statement and the name of the file to be included, as in the following.

```
#include <gdi.jsh>
#include "gdi.jsh"
#include 'gdi.jsh'
```

Any one of these lines makes all of the functions in the library file gdi.jsh available to the script that has the line. If the file to be included is in one of the directories in SEDESKPATH, you do not need to specify anything more than the name and extension of the file. If it is not, you must supply a full path so the interpreter can find the file, as shown next.

```
#include <c:\CMM\LIB.JSH>
```

The quote characters, ' or ", may be used in place of the angled brackets < and >.

To include several files in one program simply use multiple #include directives as shown.

```
#include <screen.jsh>
#include <keyboard.jsh>
#include <init.jsh>
#include <comm.jsh>
```

The ScriptEase interpreter will not include a file more than once, so if a file has already been included, a second or subsequent #include directive, with the same file specification, has no effect. ScriptEase ships with a large number of libraries of pre-written functions that you can use. Library files are plain text files, as are all ScriptEase scripts, and have the extension .jsh as a default. See the section on ScriptEase versus the C language for more information about library files and including them in scripts. See the tutorial section about writing and including the write.jsh library file.

if, ifdef, elif, else, endif

These directives are all preprocessor conditionals and allow you to specify a different set of global variables and constants based on different conditions at load and tokenize time. Conditional directives are frequently used in scripts designed to run on different operating systems by ensuring that scripts include files that are appropriate for the operating system being used.

#if is used like an if statement. #else corresponds to an else statement. #elif corresponds to an else if statement. These directives define which block of code is actually used when a script is interpreted and executed. You must use them with terminating #endif directives to mark the ends of code blocks.

For example, suppose you have a script that builds long path names from directories supplied to it in different variables. If you are working in a DOS based environment, the backslash character is used to separate d irectories, so you could indicate the full path of a file in DOS as follows:

```
var fullPathOfFile = Clib.rsprintf("%s\\%s\\%s\\%s",
rootdirectory, subdirectory1, subdirectory2, filename);
```

If you ported this script to a UNIX machine, however, you would run into problems since UNIX uses forward slashes to separate directories.

You can get around this problem by defining the separator character differently for each operating system:

```
#if defined(_UNIX_)
   #define PathChar '/'
#elif defined(_MAC_)
   #define PathChar ':'
#else
   #define PathChar '\\'
#endif
```

By putting the separator character in a variable, you can make the script work on any operating system:

```
var fullPathOfFile = Clib.rsprintf("%s%c%s%c%s%c%s",
rootdirectory,
PathChar, subdirectory1,
PathChar, subdirectory2,
PathChar, filename);
```

The #ifdef directive is a limited form of the #if statement and is equivalent to #if defined(var). The example above could be rewritten with #ifdef statements like this:

```
#ifdef (_UNIX_)
   #define PathChar '/'
#ifdef (_MAC_)
   #define PathChar ':'
#else
   #define PathChar '\\'
#endif
```

link

The #link command incorporates pre compiled libraries, dynamic link library (.dll) files, into the ScriptEase interpreter. The #link directive is similar to the #include statement with no parameters. For example, the directive

```
#link <oleautoc>
```

lets the interpreter use the functions for OLE automation. #link takes no parameters other than the name of the library being linked.

Although you could write these functions in JavaScript, the functions in the #link libraries are processor intensive and run much more quickly from a compiled source.

Nombas currently supplies the following #link libraries:

- **GD**
  for generating .gif files and other graphics functions
- **ODBC**
  for working with ODBC databases
- **OLEAUTOC**
  for doing OLE automation
- **REGEXPSN**
  to perform complex searches
- **SESOCK**
  for working with sockets

Please contact Nombas for more information on the #link developer's kit, which lets users to create customized #link libraries. The most recent versions of #link libraries are listed on the Nombas downloads page at the following web site:

http://www.sedesk.com/

http://www.nombas.com/us/

option

The #option directive has four useful options that are available when a file is being parsed before it begins executing. Many programmers will appreciate the help that these options provide while developing scripts. Each of the following options may be preceded by the not operator, "!", to turn an option off.

- **`DefaultLocalVars`**
  With this option set, all variables declared inside functions are local variables. The default is that variables declared in functions without the var keyword or variables declared outside functions are global. Thus, with this option set, only variables declared outside of functions are global.
- **`MathErrorWarnings`**
  With this option set, ScriptEase provides warning messages on division by zero, operations on NaN, and invalid automatic type conversions to numbers.
- **`RequireFunctionKeyword`**
  This option requires that the "function" or "cfunction" keywords precede functions. This option is similar to requiring the var keyword for variables.
- **`RequireVarKeyword`**
  With this option set, all variables, both global and local, must be declared with the var keyword. This option is useful while developing a script. It helps to insure that variable names are typed correctly and to avoid common mistakes when undefined variables are expected to be defined.

The default behavior for ScriptEase Desktop is consistent with normal JavaScript and is represented by the following list of #option settings.

```
#option !DefaultLocalVars
#option MathErrorWarning
#option !RequireFunctionKeyword
#option !RequireVarKeyword
```

Remember that the **#option directive must begin in the first column** of the line on which it appears in a script. This directive may be used multiple times but must always begin in the first column.

Executing a script

The sequence of events when a script is executed are:

- When a script is run, the interpreter first checks for the SEDESKPREFS environment variable and then executes preprocessor directives. It locates any files that are included or linked, assigns values to any #defined tokens, and the statements between #if and #endif directives are executed, if the directives evaluate to true. Settings for the #option directive are observed when encountered.
- Then any code that is not included as part of a function is executed. Any variables referenced are global variables and are available to all functions in the script.
- Finally, the main() function is executed, if there is one. If there is no function main(), the program will end after running through all of the steps in the initialization. Code may be set to execute when the program exits using the Clib.atexit() method.

ScriptEase shell command-line

Except for the Unix version, when any version of ScriptEase is run without command line arguments, a user is put in a ScriptEase shell. In a shell, ScriptEase scripts can be run from the command line. Other programs may also be run from a shell command line. In Windows versions, though the shell resembles a DOS command prompt, Windows applications may be run from the text command prompt. To exit any ScriptEase shell, simply type "exit" at the command prompt.

File redirection

The input and output of commands executed from a ScriptEase shell may be redirected from or to a file with redirection operators.

- **<**
  This command line operator redirect standard input from a file to a file so that a program gets input from the redirected file instead of the keyboard.
  ```
  sort < list.txt
  ```

- **>**
  This command line operator redirect standard output from a file to a file instead of to the screen. The file receiving the redirected output is created new every time.
  ```
  dir > dir.txt
  ```
- **>>**
  This command line operator is similar to the > operator, except if the file receiving the redirected output exists, then the output is appended to the file. If the file does not exist, it is created new.
  ```
  dir >> dir.txt
  ```

In the first example, the sort program receives the lines of text from list.txt file as its input, and it displays those lines to the screen in alphabetical order.

In the second example, the directory listing that the dir command would normally display to the screen is saved to the file dir.txt. If the file dir.txt already exists, it is over written by the new directory listing.

In the third example, the directory listing is appended to the file dir.txt, unless the file does not exist, in which case, the file is created.

Auto files

When a ScriptEase shell starts, three files are executed automatically if they exist: autoload.jse, autoexec.jse, and shellchr.jse. These files modify and extend the functionality of the ScriptEase shell. Various extensions to a ScriptEase shell are implemented through the hooks: ShellFilterCharacter() and ShellFilterCommand().

The following list has descriptions of some of the features implemented by the autoload.jse file that ships with ScriptEase. To see a complete list of shell commands, type "help" at a shell command prompt. For help with a specific command, type "help command". The word "command" should be replaced by the name of the actual command for which you want help.

- **CD -** this command changes the directory.
- **CD implicit -** a ScriptEase shell has the ability to automatically change directory. If the name of an existing directory is entered at a shell prompt, the current directory is changed to it.
- **ChDir -** this command changes directory.
- **Cls -** this command clears a ScriptEase screen.

- **History -** a ScriptEase shell maintains a history of the commands that have been entered in it. The up and down arrow keys may be used to scroll through this list of commands.
- **Start -** this command is used in operating systems that support multitasking. When a program is started with this command, a ScriptEase shell does not wait till the program finishes executing before returning to the command prompt. Thus, another application can be launched while the previous program is still running. For example, the following command line launches the program notepad.exe and immediately returns to the shell prompt.

  ```
  start notepad.exe
  ```

  When launching a program with the start command, any arguments needed by the program follow the program name as normally done.
- **Tab -** the tab key functions as a speed key for entering directory or program names. A user can enter the first letters of an existing directory or file name and then press the tab key. The initial letters are filled out to the name of the first directory or file that fits these letters. For example, suppose the directory documents and the file dinosaurs.txt exist in the current directory. If a user enters "do" and presses the tab key, the entry is filled out to "documents". But, if a user enters "di" and presses the tab key, the entry is filled out to "dinosaurs.txt".
- **Type -** this command displays the contents of a text file to the screen.

Running a script

There are several ways to run a ScriptEase script: from an operating system command prompt, a ScriptEase shell prompt, a GUI interface, or a batch file. All examples in this section assume that files are either in the same directory or can be found in the directories specified by either SEDESKPATH or PATH. See the description of the SElib.compileScript() method, on page ?, for more information about executing scripts as text, object, or executable files.

Operating system command prompt

At the command prompt of most operating systems, the ScriptEase interpreter program is the first program entered on the command line. Short fragments of ScriptEase code may be passed to the interpreter directly. The following command line displays "hello" to screen.

```
Secon32.exe "Screen.write('hello')"
```

The quotes are required around script commands when they are passed directly to an interpreter. If quotation marks are required in a script command, then one of or a combination of the following must be used: single quotes, back quotes, or escape sequences. In this example, single quotes are used.

Passing commands directly to a ScriptEase interpreter is seldom done. Usually, a script is contained in a text file created with a text editor. The following command line illustrates using a script file.

```
Secon32.exe Myscript.jse Myarg1 Myarg2 . . .
```

The ScriptEase interpreter secon32.exe receives the script myscript.jse and its arguments as parameters. When the script itself executes, it receives the arguments after it as its parameters. If a script does not require arguments, none need to be specified. The file myscript.jse may be put on the command line without its extension .jse, since the interpreter automatically adds the default extension .jse if it is absent. Thus, the above command line could look like the following.

```
Secon32.exe Myscript Myarg1 Myarg2 . . .
```

Some operating system command processors, such as 4Dos.com which replaces Command.com in a DOS environment, allow extensions to be defined as executable extensions. If the extension .jse is defined as an executable extension, then the above two lines may be shortened to one of following lines.

```
Myscript.jse Myarg1 Myarg2 . . .
Myscript Myarg1 Myarg2 . . .
```

ScriptEase shell command prompt

A ScriptEase shell command prompt accepts every form of a command line shown above in the section about an operating system command prompt. Unless a user has an enhanced command processor as mentioned above, a ScriptEase shell provides, perhaps, the most flexible command prompt environment from which to execute scripts.

GUI environment

Graphic User Interfaces are the most popular operating environments for most computer users in today's world. Most people are familiar with the process of double-clicking an icon to launch an application. ScriptEase scripts may be launched in the same way. When ScriptEase installs, it puts appropriate information in the settings of an operating environment, such as in the registry of Windows. As with all applications in a Graphic User Interface, parameters are not automatically passed to an application when it is launched by clicking on it.

DOS batch files

ScriptEase scripts may be imbedded into batch files by putting them between special marker statements. These special marker statements are coordinated with how an operating system processes batch files so that ScriptEase statements are ignored. There are two special marker statements for a DOS batch file: "GOTO SE_EXIT" and ":SE_EXIT". The statement, GOTO SE_EXIT, is put before ScriptEase code, and the statement, :SE_EXIT, is put after. When the operating system is processing a batch file, the goto statement simply instructs the batch processor to skip over the ScriptEase code. The ScriptEase interpreter knows to process lines of text between the statements as ScriptEase code and to ignore other lines of text in the batch file. The following example, mybatch.bat, is a batch file using special statements for ScriptEase.

```
@Echo off

Secon32.exe Mybatch.bat
GOTO SE_EXIT
Screen.writeln("ScriptEase: line one")
Screen.writeln("ScriptEase: line two")
:SE_EXIT
```

This batch file may be called from a command prompt into ways. The first way is:

```
Mybatch.bat
```

and the second way is:

```
Secon32.exe Mybatch.bat
```

Both ways result in the following output.

```
ScriptEase: line one
ScriptEase: line two
```

When the batch file is called as a batch file, it calls a ScriptEase interpreter with the batch file as a parameter. The ScriptEase interpreter knows to process only lines of text between the two special statements. After the ScriptEase interpreter has finished and control has returned to the batch processor, the next statement to be executed is the statement, GOTO SE_EXIT, with skips the ScriptEase statements.

When the batch file is called as a parameter for a ScriptEase interpreter, the interpreter simply executes the code between the special statements.

Mybatch.bat may be altered as follows to demonstrate more fully how ScriptEase script may be embedded in a batch file. In this altered file, one line of normal batch code has been put before any lines concerned with ScriptEase and one line after them. The batch file may be called by both methods shown above.

```
@Echo off

echo Mybatch.bat has started.

Secon32.exe Mybatch.bat
GOTO SE_EXIT
Screen.writeln("ScriptEase: line one")
Screen.writeln("ScriptEase: line two")
:SE_EXIT

echo Mybatch.bat has finished.
```

When this batch file is called as a batch file, it results in the following output.

```
Mybatch.bat has started.
ScriptEase: line one
ScriptEase: line two
Mybatch.bat has finished.
```

When this batch file is called as a parameter for a ScriptEase interpreter, it results in the following output.

```
ScriptEase: line one
ScriptEase: line two
```

This output is identical with the output of the original batch file, since the ScriptEase interpreter processes only the ScriptEase code, which is identical in both batch files.

OS/2 batch file

An OS/2 .cmd file has a command, EXTPROC, which allows and external processor to be called to process a batch or source file. The statement, EXTPROC, must be the first statement in the file and must be followed by a single space and the name of the external processor. For ScriptEase, the processor is SEOS2. The following file, mysource.cmd, displays the arguments passed to it from a command line and illustrates the use of the statement EXTPROC.

```
EXTPROC SEOS2

function main(argc, argv)
{
   for (var i=0; i < argc; i++)
   Clib.printf("Input argument %d = \%s\n", i, argv[i]);
}
```

OS/2 REXX file

Running ScriptEase script from a REXX file is similar to the process described for DOS batch files above. The main difference is that the two special statements surrounding ScriptEase code are different. For a REXX file, the two statements are: "SIGNAL SE_EXIT" and "SE_EXIT:". The following example file may be called in ways similar to the process described for a DOS batch file. The behavior, in regards to the ScriptEase code between the special statements, is also similar.

```
`call SD.bat %0.cmd %1 %2 %3 %4 %5 %6 %7 %8 %9`
SIGNAL SE_EXIT

function main(argc, argv)
{
   var SUM = 0;
   for (var i=1; i < argc; i++)
   SUM += Clib.atoi(argv[i]);
}

SE_EXIT:
```

Unix

Unix also allows the specification of an external processor. To specify an external processor, use the statement #! followed by the full and name of an external processor program. The following example is a simple illustration.

```
#! /usr/local/bin/se
Screen.writeln("Hello");
```

Command-line switches

Bind

The professional versions of ScriptEase processors have a /bind option which allows scripts to be compiled into stand-alone executable programs. These executable programs are completely independent and do not require any ScriptEase programs to run. Assume that the script myscript.jse exists and executes properly. The script may be compiled into a stand-alone executable program using a command line similar to the following.

```
Secon32.exe /Bind=Myscript.exe Myscript.jse
```

Such a command line instructs a ScriptEase processor to compile the script myscript.jse to the executable program myscript.exe. The name for the executable program specified after the /bind option does not have to be the same as the name of the script.

OS/2 and seos2pm.exe

Some scripts in the OS/2 operating system might require the presence of the file seos2pm.exe for an executable program that has been created using the /bind option. Some methods, such as .pmDynamicLink() and other .pm*() methods, always require the presence of seos2pm.exe. The file seos2pm.exe may be distributed free of royalty.

Predefined Values

ScriptEase has many predefined values that are useful when writing scripts. Predefined values are available to the shell, the preprocessor, and various methods and functions. Some values are available at all levels of script interpretation.

Preprocessor values

The following preprocessor values are defined and available during the preprocessing phase of script interpretation and are not available as values during the running of a script if they apply. For example, `_WIN32_` and `_95CON_` are defined if *SEcon32.exe* is the current interpreter. If a preprocessor value is defined, it has a value of 1. Normally, these preprocessor values are used with the preprocessor directives: `#if`, `#ifdef`, and `#if defined()`. See the section on the Processor for detailed information.

The distinction between compile time and run time is important if a script is being compiled into a jsb file. If decisions are being made based on preprocessor directives, then those decisions are made at compile time. Thus, if certain behavior depends on being run by a particular interpreter, then the script must be run by the same program with which it was compiled. An example is in order.

Consider the following fragment:

```
#ifdef _95WIN_
   var n = 10;
#else
   var n = 20;
#endif
```

If this fragment is compiled with SEwin32.exe and run with SEwin32.exe, the variable n is initialized to 10 as expected. But if this fragment is compiled with SEcon32.exe and run with SEwin32.exe, the variable n is still initialized to 10, though the intent might have been for it to be initialized to 20 under any platform but Windows 95/98. This difference in behavior does not occur when a text script is being interpreted and only applies to scripts that have been compiled. If a script is being interpreted as a text file, the results are consistent.

If you want consistent behavior in a compiled script, do the test at run time. The above fragment could be written as:

```
if (defined(_95WIN_)
   var n = 10
else
   var n = 20;
```

In this case, a compiled script always makes the expected decision. One warning. The `#option RequireVarKeyword` preprocessor directive is important here. If the `var` keyword is being required, the fragment above will cause an error since `_95WIN_` is not declared using `var`. If you are requiring that variables be declared using the `var` keyword, rewrite the above fragment to be:

```
#option !RequireVarKeyword
if (defined(_95WIN_)
   var n = 10
else
   var n = 20;
#option RequireVarKeyword
```

Platform

The following values indicate which ScriptEase interpreter is currently running. These values are useful when writing scripts to be used on multiple platforms that perform operations that are specific to different platforms.

`_DOS_`	MS-DOS
`_DOS32_`	MS-DOS with extended memory
`_MAC_`	Macintosh
`_NWNLM_`	Netware Network Lan Manager
`_OS2_`	OS/2
`_UNIX_`	UNIX
`_WINDOWS_`	Windows
`_WIN32_`	Windows 32 bit 95/98/NT
	If Win32 is defined, one of the following is defined also, which allows more precision in determining the operating environment being used.
`_95CON_`	Windows 95/98 in a DOS or console session
`_95WIN_`	Windows 95/98 as a window

`_NTCON_`	Windows NT in a DOS or console session
`_NTWIN_`	Windows NT as a window

Shell

`_SHELL_`	Defined if any ScriptEase interpreter is running as a shell

Strictness of interpretation

The following values are used with the #option preprocessor directive

`DefaultLocalVars`	Variables inside functions default to being local
`MathErrorWarnings`	Ensures that warning messages are displayed for various invalid math operations
`RequireVarKeyword`	All variables must be declared with the keyword var
`RequireFunctionKeyword`	All functions must be declared with the keywords function or cfunction

Predefined constants and values

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

`false`	Boolean false
`null`	A null value with multiple uses
`true`	Boolean true
`BIG_ENDIAN`	Indicates whether a processor stores a multi-byte value as Big_Endian or Little_Endian.
`0`	0, false, indicating that the processor stores

	the low byte of a value in low memory, such as Intel.
`1`	1, true, indicating that the processor stores the low byte of a value in high memory, such as Motorola.
`CLOCKS_PER_SEC`	
`EOF`	In file operations, indicates that the end of a file has been reached
`EXIT_FAILURE`	Indicates an error when exiting a script, 1-255
`EXIT_SUCCESS`	Indicates success when exiting a script, 0
`FATTR_NORMAL`	Attribute for a normal file
`FATTR_RDONLY`	Attribute for a read only file
`FATTR_HIDDEN`	Attribute for a hidden file
`FATTR_SYSTEM`	Attribute for a system file
`FATTR_SUBDIR`	Attribute for a directory
`FATTR_ARCHIVE`	Attribute for a changed file
`INTERP_COMPILED_SCRIPT`	Run a script compiled with the compileScript() method. This flag only works with the INTERP_TEXT flag.
`INTERP_LOAD`	Load code into same function and variable space as the code that is calling .interpret(). All functions, and variables are supplied to the code being called, which can modify and use them. If the code being called has similarly named functions or variables as the calling code, the functions in the called code replace those in the calling code.
`INTERP_NOINHERIT_GLOBAL`	Global variables will not be inherited as global variables in the interpreted code.

`INTERP_NOINHERIT_LOCAL`	Local variables will not be inherited by the interpreted code.
`INTERP_FILE`	Code must be the name of a ScriptEase source file, followed by any arguments.
`INTERP_TEXT`	Code is ScriptEase source code with no arguments.
`LOCK_EX`	File lock exclusive (in Windows equivalent to LOCK_SH)
`LOCK_SH`	File lock share (in Windows equivalent to LOCK_EX)
`LOCK_NB`	File lock non-blocking (use in a bitwise or with LOCK_EX or LOCK_SH)
`LOCK_UN`	File unlock
`NaN`	Not a Number
`Number.MAX_VALUE`	Largest positive number that can be represented in ScriptEase
`Number.MIN_VALUE`	Small negative number that can be represented in ScriptEase
`Number.NaN`	Not a Number
`Number.POSITIVE_INFINITY`	Any number greater than MAX_VALUE
`Number.NEGATIVE_INFINITY`	Any number smaller than MIN_VALUE
`P_NOWAIT`	Do not wait for a child process, started by SElib.spawn(), to finish before continuing script execution.
`P_OVERLAY`	Current program exit and new process executes in its place.
`P_SWAP`	Swap ScriptEase to expanded or extended memory and then execute a child process

	as P_WAIT does.
`P_WAIT`	Wait for a child process, started by SElib.spawn(), to finish before continuing script execution.
`RAND_MAX`	Maximum value that can be returned by the Clib.rand() method.
`_SEDESKTOP_`	Defined if the current interpreter is ScriptEase Desktop, in distinction from other ScriptEase interpreters.
`SEEK_CUR`	Position in a file relative to the current position in a file
`SEEK_END`	Position in a file relative to the end of the file
`SEEK_SET`	Position in a file relative to the beginning of the file
`stderr`	FILE stream for standard error output
`stdin`	FILE stream for standard input
`stdout`	FILE stream fro standard output
`VERSION_MAJOR`	The major version number of ScriptEase, for example, 4 in 4.10b
`VERSION_MINOR`	The minor version number of ScriptEase, for example, 10 in 4.10b
`VERSION_STRING`	The revision letter of ScriptEase, for example, b in 4.10b

Quick Start Tutorial

This tutorial section provides examples and information to get started with ScriptEase and is not intended to be a complete tutorial on ScriptEase or JavaScript. Keep in mind that ScriptEase scripts may be written as simple scripts, much like simple batch files, in which lines of code execute sequentially, or they may be written as structured programs. The examples in this tutorial illustrate both kinds of scripts. While in this tutorial section, the two kinds of scripts are sometimes referred to as: batch scripts and program scripts. When a script has code outside of functions and code inside of functions, it shares characteristics of both batch and program scripts. For example, the following fragment:

```
Screen.writeln("first ");

function main()
{
   Screen.writeln("third.");
}

Screen.writeln("second ");
```

results in the following output:

```
first second third.
```

ScriptEase Shell

When a ScriptEase interpreter program, such as sewin32.exe or secon32.exe, is run by itself with no script as a parameter, it starts a ScriptEase shell. A ScriptEase shell provides an interface for a user to use ScriptEase. The interface is a command line interface using text commands. Most users will not use ScriptEase as a shell but will use it to execute, as illustrated later, various scripts that they have written. As examples, both of the following lines start a ScriptEase shell.

```
Sewin32.exe

Secon32.exe
```

These lines, like all examples in this section, assume that ScriptEase executables are in the current directory or that their paths are in a PATH variable. When you are in a shell, you may exit by typing "exit" at the prompt. For simplicity, the

ScriptEase interpreters, Secon32.exe and Sewin32.exe, are used in examples. If a different platform is being used, substitute the name of the appropriate ScriptEase interpreter.

Simple script

The following line is a simple and complete script.

```
Screen.write('A simple script')
```

The following command line will execute the script.

```
Secon32.exe "Screen.write('A simple script')"
```

However, executing script fragments that fit on a command line is of limited value. The more common way to handle scripts is to save the them to a file and execute the file. Any text editor may be used to work with script files. Assume that the single line of this example has been saved to a file named simple.jse. The script could then be executed with the following command line.

```
Secon32.exe simple.jse
```

This line, like all examples in this section, assumes that a script file is in the same directory as a ScriptEase interpreter program or is visible to the program through the use of SEDESKPATH or PATH variables.

The most common way to display information on a computer screen is to use the statements: `Screen.write()` and `Screen.writeln()`. The statements are methods in the Screen object provided by ScriptEase. The Screen object provides multiple methods for working with ScriptEase screens or windows. The `write()` method displays a value, which is not limited to being a string, on the screen. The `writeln()` method does the same thing but automatically adds carriage return and new line characters to the end of a displayed value. Both of these methods display information to the default screen or window of a ScriptEase interpreter. ScriptEase ships with library files which have routines for displaying information in other ways, such as in windows on a Windows platform.

Date and time display

The following fragment:

```
var d = new Date
Screen.writeln(d.toLocaleString())
```

produces output similar to the following.

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

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

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

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

```
var d = new Date;

function main()
{
   Screen.writeln(d.toLocaleString());
}
```

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

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

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

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

Function with parameters

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

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

```
var dat = new Date();
   // Sun == 0 . . . Sat == 6
if (dat.getDay() == 6)
{
   var FirstLine = "The host is closed on Saturday.";
   var SecondLine = "Terminating program.";
   ExitOnError(FirstLine, SecondLine, EXIT_FAILURE);
} //if

function ExitOnError(LineOne, LineTwo, ExitCode)
{
   Screen.writeln(LineOne);
   Screen.writeln(LineTwo);
   Clib.exit(ExitCode);
} //ExitOnError

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

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

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

The third line tests, with an if statement, whether the current day is day number 6, Saturday. If the day is 6, then two variables, FirstLine and SecondLine, are created with string information in them. Then the function ExitOnError() is called with the two variables as the first two parameters of the function. The parameter is `EXIT_FAILURE`, which is a predefined value in ScriptEase, is passed as the third parameter.

The function ExitOnError() uses the information passed to it in its parameters: LineOne, LineTwo, and ExitCode. Notice that the variables, FirstLine and SecondLine, do not have the same names as the parameters, LineOne and LineTwo. Arguments, such as FirstLine and SecondLine, do not have to have the same names as the parameters to which they are passed, in this case, LineOne and LineTwo. Further, arguments do not have to be variables as parameters are. In this example, `EXIT_FAILURE` is not a variable but is a predefined value. The variables FirstLine and SecondLine did not have to be created at all. The function ExitOnError() could have been called with two literal strings instead of two variable names. But such a line could become too long. The use of variables in the if statement makes the code easier to read and to alter. Without the variables, the call to ExitOnError() would have been:

```
ExitOnError("The host is closed on Saturday.", "Terminating
program.", EXIT_FAILURE);
```

Specifically, what does the function ExitOnError() do? It displays the parameter LineOne on a line by itself, displays the parameter LineTwo on the next line, and exits the script with an error code. The rest of the script, no matter how long, is not executed on Saturday. The display to the screen is:

```
The host is closed on Saturday.
Terminating program.
```

The exit from the script is accomplished in line 14 with the `Clib.exit()` method. The identifier Clib is the Clib object and gets its name from "C library" since the Clib object has almost all of the functions of the standard C library. These C functions are implemented as methods of the Clib object. Though documentation with ScriptEase covers this library of methods, other books and documentation that cover the standard C library are useful, especially for users new to the C language. Experienced C programmers will be able to use the Clib object quickly and easily. See the section on the Clib object for more information.

If the day of the week is not 6, then the statements in the if statement block of the code are ignored. The script is not terminated, and all code after the if statement is executed. In this script, the string "The program is continuing." is displayed to the screen by the last line in the script.

Terminology

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

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

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

```
var n = parseFloat("3.21");
Screen.writeln(typeof n);
Screen.writeln(n);
```

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

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

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

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

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

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

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

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

Function with a return

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

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

var CubedNumber = Cubed(3);
Screen.writeln(CubedNumber);
```

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

Screen.write improved

The methods, `Screen.write()` and `Screen.writeln()`, are useful and easy to use, but they can be improved. For example, the first line in the following fragment works, but the second line causes an error since these methods can have only one parameter.

```
Screen.writeln(1)
Screen.writeln(1,2)
```

The first step in writing our improved display functions is to write a function that uses `Screen.write()` and that accepts a variable number of arguments. The `Write()` function in the following fragment accomplishes our goal.

```
function Write()
{
   for (var i=0; i < arguments.length; i++)
   Screen.write(arguments[i]);
} //Write
```

This function uses a for loop to display, using the Screen.write method, all arguments passed to it. The key element in the loop is the special property, arguments. Every function has an arguments property which may be used as an Object or an Array. In the initialization section of the for loop, the arguments property behaves like an object since arguments.length is a property of arguments that returns the number of arguments which have been passed to a function. The line that uses `Screen.write()` to display the values that have been passed to the function uses the arguments property like an array. The first argument in the array is indexed by the number 0, that is, the first argument is arguments[0]. The last argument is indexed by arguments.length minus 1, that is, arguments.length - 1. See the section on function properties for more information about the arguments property of functions.

Our second step is to write a similar function that automatically adds end of line characters to the display. The following fragment accomplishes our goal by writing a blank line at the end of the display of all values passed to the function.

```
function Writeln()
{
   for (var i=0; i < arguments.length; i++)
      Screen.write(arguments[i]);
   Screen.writeln();
} //Writeln
```

What has been accomplished? We can now display multiple values with a single call to a function. The following fragment:

```
Write(1,2,3)
Write(4,5)
```

produces the following display.

```
12345
```

And the following fragment:

```
Writeln(1,2,3)
Writeln(4,5)
```

produces the following display.

```
123
45
```

We can add a few touches to make such routines more useful, especially when developing and debugging a script. First, we can write routines that allow us to put characters or strings between displayed values.

```
function WriteSep(Sep)
{
   for (var i=1; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write(Sep);
   } //for
} //WriteSep

function WritelnSep(Sep)
{
   for (var i=1; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write(Sep);
   } //for
   Screen.writeln();
} //WritelnSep
```

Now we can display multiple values separated by characters of our choice. The following fragment:

```
WriteSep('--',1,2,3)
WriteSep('--',4,5)
```

produces the following display.

```
1--2--3--4--5--
```

And the following fragment:

```
WritelnSep('--',1,2,3)
Writeln('--',4,5)
```

produces the following display.

```
1--2--3--
4--5--
```

The separator character or characters must be a string which is passed as the first parameter to the functions. A modified form of these two routines makes them especially useful when developing and debugging scripts that depend on precise strings. The following fragment defines the modified routines.

```
function WriteBar()
{
   for (var i=0; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write('|');
   } //for
} //WriteBar

function WritelnBar()
{
   for (var i=0; i < arguments.length; i++)
```

```
    {
       Screen.write(arguments[i]);
       Screen.write('|');
    } //for
    Screen.writeln();
 } //WritelnBar
```

These two routines simply use the pipe bar, |, as a separator between values. The following fragment:

```
 var s1 = "one";
 var s2 = "one ";
 WriteBar(s1);
 WriteBar(s2);
```

produces the following display.

```
 one|one |
```

And the following fragment:

```
 var s1 = "one";
 var s2 = " one ";
 WritelnBar(s1);
 WritelnBar(s2);
```

produces the following display.

```
 one|
 one |
```

If these variables were displayed using Screen.write() or Screen.writeln(), the difference between them would not be apparent on the screen. But, by using the pipe bar separator, the difference between the two strings is obvious.

Library file

We have now written six useful functions. If we want to use them over and over again, we can put them into a single library file that we include in other scripts. The following fragment is an entire script to be saved as a library file with the name write.jsh. Since these are general use routines, we can save write.jsh in a directory for general library routines. The likely choice is \sedesk\jsh which is created when ScriptEase is installed.

```
 // Write routines library file

 function Write()
```

```
{
   for (var i=0; i < arguments.length; i++)
   Screen.write(arguments[i]);
} //Write

function Writeln()
{
   for (var i=0; i < arguments.length; i++)
      Screen.write(arguments[i]);
   Screen.writeln();
} //Writeln

function WriteSep(Sep)
{
   for (var i=1; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write(Sep);
   } //for
} //WriteSep

function WritelnSep(Sep)
{
   for (var i=1; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write(Sep);
   } //for
   Screen.writeln();
} //WritelnSep

function WriteBar()
{
   for (var i=0; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write('|');
      } //for
} //WriteBar

function WritelnBar()
{
   for (var i=0; i < arguments.length; i++)
   {
      Screen.write(arguments[i]);
      Screen.write('|');
   } //for
   Screen.writeln();
} //WritelnBar
```

```
var wb  = WriteBar;
var wlb = WritelnBar;
```

Notice the last two lines that assign the functions, `WriteBar()` and `WritelnBar()`, to the variables, wb and wlb. These assignments illustrate how JavaScript treats almost all identifiers, whether variables, arrays, objects, or functions, like variables. Both wb and wlb may be used in place of the functions assigned to them. In the following example, both lines of code do the same thing.

```
WriteBar("Line one")
wb("Line one")
```

Likewise, both lines of code in the following fragment do the same thing.

```
WritelnBar("Line two");
wlb("Line two");
```

The only reason for adding the two assignments at the end of write.jsh is for the ease of a programmer. The functions `WriteBar()` and `WritelnBar()` are mainly useful while developing and debugging a script. Typing `wb()` is simply faster than typing `WriteBar()`. Plus, the assignments illustrate how JavaScript handles many identifiers as variables. Though not officially accurate, the variables, wb and wlb, may be thought of as aliases, if that metaphor is helpful.

Two more observations should be made concerning write.jsh. First, the functions defined do not list a specific number of parameters to receive, which is the most common way to define functions. The following fragment illustrates the more common method of defining functions.

```
function MoreCommonWrite(Str1, Str2)
{
Screen.write(Str1);
Screen.write(Str2);
}
```

The function MoreCommonWrite() explicitly defines, expects, and uses two parameters. Whereas, the functions defined in write.jsh were written to expect and use a variable number of parameters. A programmer has the choice of which way to define and use parameters in a function.

Second, each function end with a comment that is the function name. For example, the `Write()` function ends with "} //Write". The function name in a comment at this point is not required nor particularly useful in very short functions such as the ones in write.jsh. However, they are useful when writing functions of many lines and are shown here mainly because of programming habits, hopefully good ones.

The following fragment illustrates how to include our newly created library file in another script.

```
#include 'write.jsh'

var s1 = "This fragment illustrates "
var s2 = "how to include "
var s3 = "our library file "
var s4 = "in a script."

Writeln(s1,s2,s3,s4)
```

The #include statement is a preprocessor directive which instructs a ScriptEase interpreter to include the text of a file in a script before the file is executed. In effect, the text of the included file replaces the include directive as if the text had been originally typed in the script at that point. Thus, the six functions and two variables used as aliases that we defined in the library file are available to us in the current script.

Library and sample files

ScriptEase Desktop ships with a number of files that are sample scripts. Many of these scripts are useful programs that perform both useful and powerful tasks. When ScriptEase Desktop is installed, it puts these sample scripts in various subdirectories of the installation directory. The default installation directory is C:\Sedesk, and all other ScriptEase Desktop directories are subdirectories. Of course, a user may specify a different installation directory, but for now, SEdesk is assumed. Sample files are put into the following directories.

- SEdesk\Utility
- SEdesk\Sample

In addition, scripts that are specific to a particular operating system or platform are put into the directory with the files for that platform. For example, scripts for Win32 are put into the following directory.

- Sedesk\Win32

Many library files are provided in the following directory.

- Sedesk\Include

These library files provide very powerful routines that are not part of the actual JavaScript or ScriptEase languages. Some of the library files are specific to a particular platform, such as the dialog library files for Windows.

ScriptEase users and programmers will find that many of the routines and programs that they want have already been written and provided as sample and library files. Before writing a script, it is usually beneficial to check these files. Sample and library files are self documented, that is, the documentation explaining the files and routines in them is included in the files as comments. These comments are explicitly written to be similar to a reference manual. In addition, files that document these sample and library files are provided in the directories with them. These documentation files may be found in text (txt), rich text (rtf), or hyper text markup (htm) formats.

The reason for documenting library and sample files in this manner is that they are continually being improved and updated. The latest library, sample, and documentation files may be found in the ScriptEase Desktop pages of the Nombas web site at:

http://www.sedesk.com/
http://www.nombas.com/us/

Feel free to visit this site often and download updated files and documentation.

Using library files

Whether users write their own library files, as illustrated above with the improved Screen.write routines, or use the library files provided by Nombas, they need to include them with their scripts. As shown above, the #include preprocessor directive is a commonly used way, as it is in other programming languages. See the section on preprocessor directives for more information on using the include directive.

ScriptEase provides another, simple and powerful, way to include files that is not standard in other programming languages. The SEDESKPREFS environment variable is an innovation that many ScriptEase users will learn to appreciate. To use this variable, simply set it to the files, usually library files, that should be included with all scripts. In DOS and Windows platforms, appropriate lines may be added to the autoexec.bat file.

To include the write.jsh library file, as shown above, add the following line to the autoexec.bat file.

```
set SEDESKPREFS=write.jsh
```

With this setting, every script that is run by a ScriptEase interpreter will include write.jsh at the beginning of the script. More than one file may be included as a SEDESKPREFS setting. If more than one file is to be included, separate the file names with a semicolon, as is done with the PATH variable. The following line illustrates.

```
set SEDESKPREFS=write.jsh;file.jsh
```

With this setting, both write.jsh and file.jsh will be included at the beginning of the interpretation of scripts. The SEDESKPREFS setting only affects scripts, such as .jse files, that are being interpreted. Scripts that have been compiled into a stand alone executable file, using the /Bind option of ScriptEase:Desktop Pro, are not affected by the settings of SEDESKPREFS. The SEDESKPREFS will be used when a script is compiled but ignored when it is run as an executable program.

Both of the example lines shown above assume that the SEDESKPATH environment variable or registry key has been set to point to the directories holding library files, or any files to be included in a script. SEDESKPATH is set when ScriptEase is installed. SEDESKPATH may be altered by a user, but normally will not need to be changed.

Path names may be included in the SEDESKPREFS setting. For example, assume that a user wants to keep certain library files in a directory not included in SEDESKPATH. Then SEDESKPREFS could be set with a full specification of the file, such as in the following.

```
set SEDESKPREFS=C:\Sedesk\MyLibraries\write.jsh;file.jsh
```

With this setting, the file, C:\Sedesk\MyLibraries\write.jsh, will be included with all scripts, even if a file named write.jsh exists in the standard library directories. The file, file.jsh, which is in the Sedesk\Jsh library directory will also be included. The library file, file.jsh, is a useful file when working with files on a disk.

A user might have certain routines and settings that he wants included with all scripts. The settings and include statements could be put into every script written, or they could be put into one library file that is included with every script. For example, the following script could be saved as general.jsh in the Sedesk\Include directory.

```
#option RequireVarKeyword
#option !DefaultLocalVars
#option RequireFunctionKeyword
#option MathErrorWarnings

#include 'write.jsh'
```

Then set SEDESKPREFS as follows.

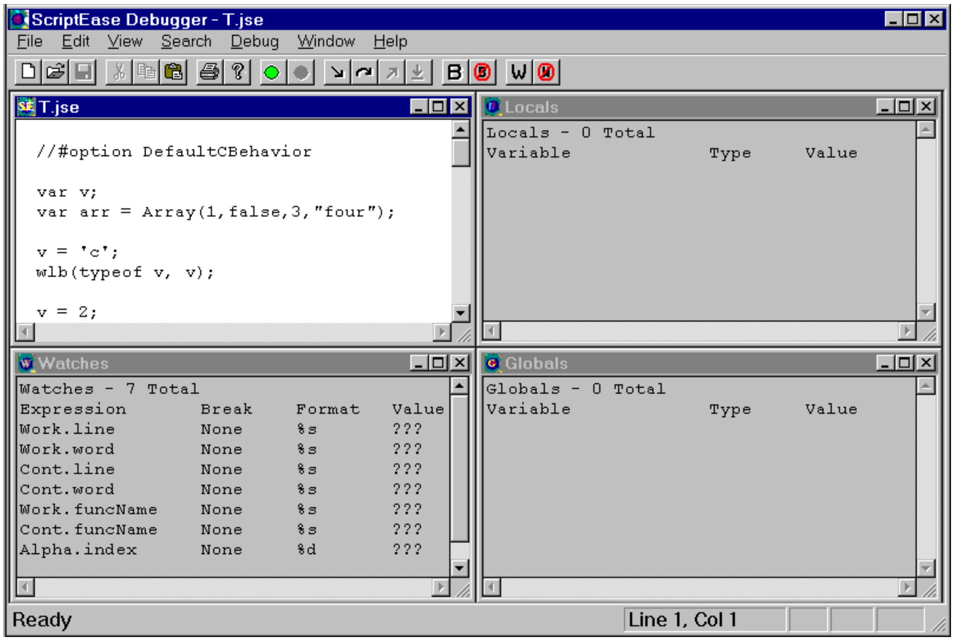
```
set SEDESKPREFS=general.jsh
```

Now what has been accomplished? Every interpreted script will include general.jsh. The library file general.jsh sets the strictest options for scripts. These option settings facilitate the writing and development of scripts by reducing the number of bugs that enter a script by common errors. Further, general.jsh includes write.jsh that has the improved write routines developed above. Thus, write.jsh will also be included with all interpreted scripts.

There is no absolutely right or best way to use the flexibility provided by SEDESKPREFS, but the flexibility is there to meet differing needs and styles.

Integrated Debugger

ScriptEase comes with a source debugger that provides a complete Integrated Debugging Environment, which means you can edit a script while you are debugging it.


ScriptEase Debugger - T.jse
File Edit View Search Debug Window Help
T.jse
Locals
Watches
Globals
Ready
Line 1, Col 1

```
//#option DefaultCBehavior

var v;
var arr = Array(1,false,3,"four");

v = 'c';
wlb(typeof v, v);

v = 2;
```

```
Locals - 0 Total
Variable                  Type      Value
```

```
Watches - 7 Total
Expression          Break     Format    Value
Work.line           None      %s        ???
Work.word           None      %s        ???
Cont.line           None      %s        ???
Cont.word           None      %s        ???
Work.funcName       None      %s        ???
Cont.funcName       None      %s        ???
Alpha.index         None      %d        ???
```

```
Globals - 0 Total
Variable                  Type      Value
```

The debugger is a Windows application with a standard Multiple Document Interface (MDI) like many other applications. The image above has four windows showing: the script, Watches, Locals, and the Globals window. The specifics about windows are explained later. The script window is explained in the section about the File menu options, and the other three in the section about the window menu options. For now, just understand that the tiled arrangement shown above is just one out of many ways to display windows in the debugger. You may have multiple script window or only one. You may have only one window showing or any combination of windows. Like any MDI application, you may maximize, minimize, tile, and cascade windows. In short, the user interface of the ScriptEase debugger is a standard windows interface.

ScriptEase debuggers are available only for Windows operating environments. There are debuggers for Windows 95/98, Windows NT, and Windows 3.x.

Using the ScriptEase Debugger

The ScriptEase debugger is a source code debugger which means that you may debug programs while watching the execution of a program line by line in the original source code. You may set breakpoints, trace lines of code as they execute, step into and over functions, watch variables that you choose, keep up with global and local variables, and other powerful options that you expect in a good source code debugger.

The main window of the ScriptEase debugger consists of the following components, listed in top to bottom order.

Components of main MDI window

Menu bar

All commands in the ScriptEase debugger may be accessed through menus. The menu bar is described completely in the following section, "Main menu bar."

Tool bar

The toolbar has buttons for the common and useful debugger commands. Instead of clicking menus, you may click a button on the toolbar as a shortcut. The commands that are available on the toolbar are exactly the same as the corresponding commands in menus. In the section, "Main menu bar," commands that are available on the toolbar are indicated by the notation: "In toolbar."

Document window

The document window is a standard Windows Multiple Document Interface (MDI) window. You may open four kinds of windows within the document window: Source, Watches, Locals, and Globals.

Status bar

The status bar at the bottom of the window provides useful information concerning the currently active window. The current cursor position in a script window is displayed as line and column numbers. The status of the Caps, Num, and Scroll lock keys is displayed. When the mouse cursor is over menu and toolbar items, help or hint information displays in the status bar. The general state of the IDE is also displayed, such as "Ready" or "Program Terminated."

MDI windows

Source

Source windows may be called script windows since they display the source code of a script file. These script windows are actually text editing windows in which scripts may be viewed, edited, or used for source line debugging.

When used for editing, the editor is capable of writing an entire script, but the editing features of a script window are basic and adequate for simple scripts. Normally, you will use a more powerful editor for most writing and editing of sophisticated scripts, an editor such as the ScriptEase Editor that accompanies ScriptEase products. The ScriptEase Editor has features that allow you to coordinate your work effectively with the ScriptEase debugger. Currently, when you change text in a script while it is still loaded in a script window in the debugger, you must manually reload the file in the debugger. However, when you make changes in a script while in a script window, the ScriptEase Editor can automatically detect the changes and reload the file. Thus, for most editing of scripts use the ScriptEase Editor for major writing and script windows in the debugger for minor changes while debugging a script.

The current position in a source file is indicated by a special marker, icon, that can be chosen from several options. In addition, breakpoints may be set in a script window. Breakpoints display as small red hexagons at the beginning of the lines of scripts to which they apply.

You may open multiple script windows at the same time. Remember, that various debugging commands apply to the currently active script window. For example, a command such as "Debug | Run in Debugger" runs the script in the currently active source window, not any other scripts that might be open in source windows.

Source windows have gray backgrounds when in debugging, as opposed to editing, mode. You may not edit scripts while in debugging mode. When script windows have gray backgrounds, remember that you may only use debugging commands, such as "Debug | Step Into."

Globals

The Globals window displays all global variables that are available to the point in a script. The source marker indicates in a script where execution is currently occurring. The information for each variable displayed is the variable name, type, and value.

Locals

The Locals window displays all local variables that are available at the point in a script where execution is occurring. The source marker indicates in a script where execution is currently occurring. The variables in a local window constantly change as functions that have local variables are entered and debugged. The information for each variable displayed is the variable name, type, and value.

Watches

The Watches window is a place where you can view variables and expressions that you want to see. You may put plain variables here, and when they are active, these variables will show as in other windows. In addition you may set variables to be watched and used as breakpoints. You may set execution to break if a variable changes or is equal to true or false. But the watch window may be used with more than just variables, it may be used with expressions. For example,

the following code:

```
var arr = Array(false,1, 2, 3, "four");
```

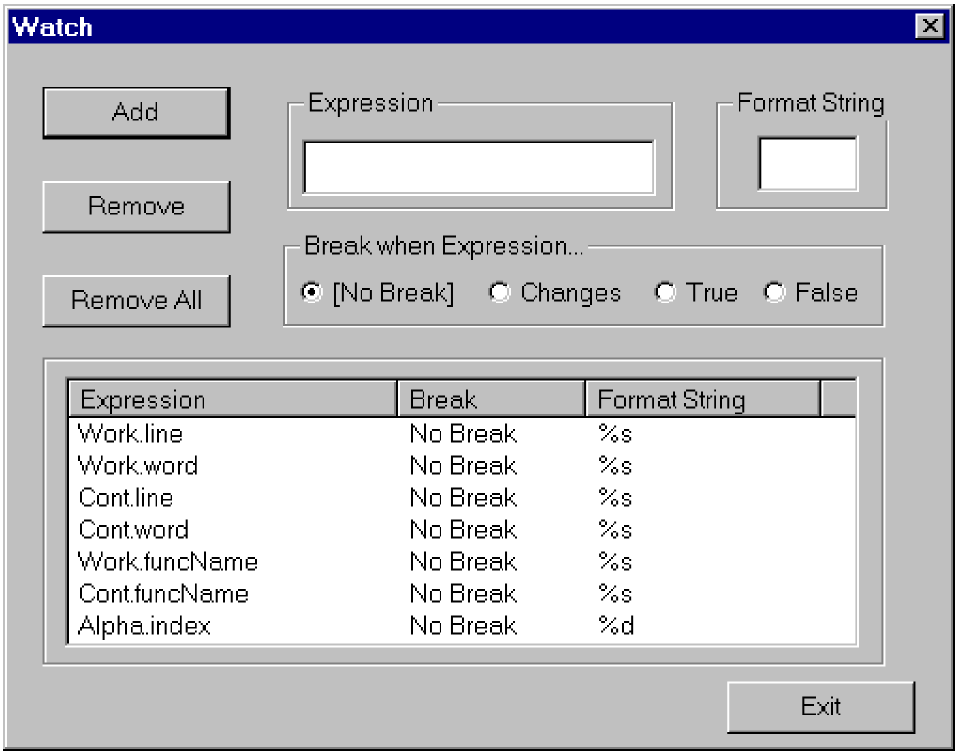
creates an array with four elements. In the Locals and Globals windows, the array `arr` is shown as type object with no value shown.

You might want to keep up with one or more elements in the array. To keep up with the second element in the array `arr`, set a watch for `arr[1]` and it will appear as an expression to be watched with its format type and value, which in this case is 1. Perhaps you want to keep up with the addition or concatenation of the fourth and fifth elements. If so, set a watch or `arr[3] + arr[4]`, which in this case would display a value of "four3".

In fact, the watch window is designed to watch expressions rather than variables. When a variable by itself is watched, the debugger simply considers it to be an expression. Notice that the second column in the watch window provides format information instead of the type of a variable.

Setting watches

The Watch dialog, Figure 2, is the main window used to set watch information.



Break when Expression...: [● [No Break]] Changes True False

Expression	Break	Format String
Work.line	No Break	%s
Work.word	No Break	%s
Cont.line	No Break	%s
Cont.word	No Break	%s
Work.funcName	No Break	%s
Cont.funcName	No Break	%s
Alpha.index	No Break	%d

Add

The Add button adds the current expression, in the Expression edit box, to the list of expressions to be watched in the Watches window.

Remove

The Remove button removes the expression which is currently highlighted in the list of expressions to be watched.

Remove All

The Remove All button removes all expressions to be watched.

Expression

The Expression edit box allows entry of expressions and variables to be watched in the Watches window.

Format String

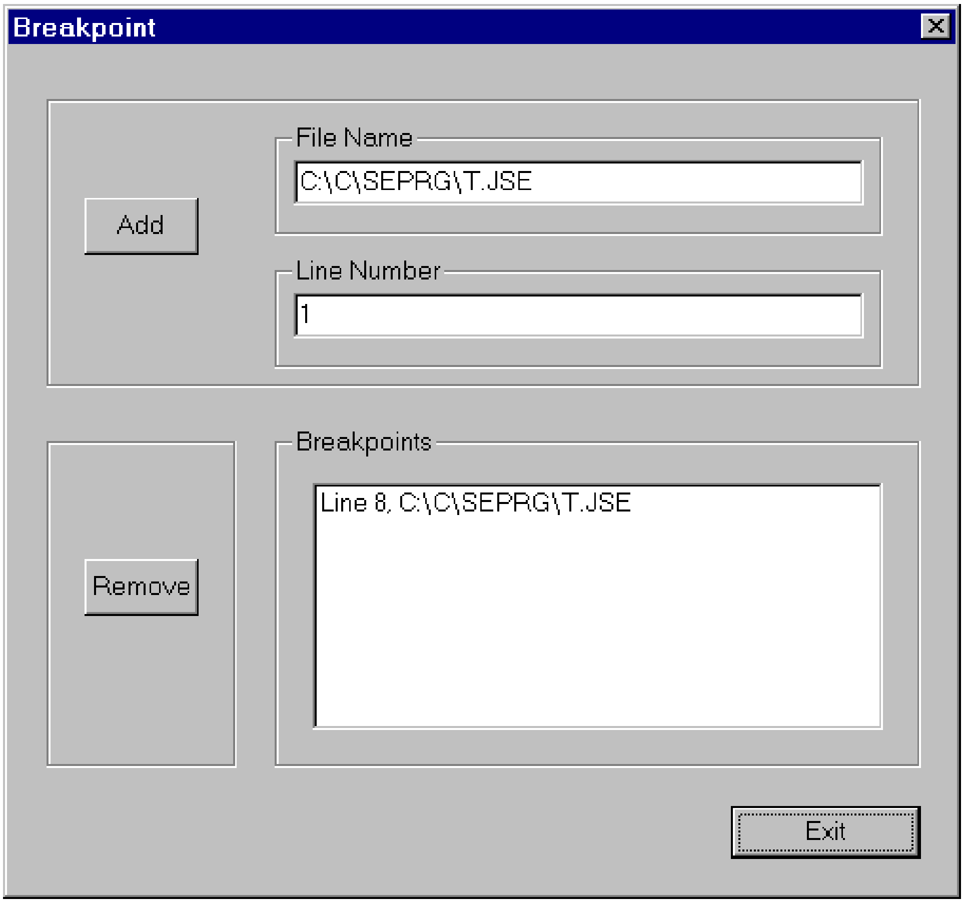
The Format String edit box allows some control over the format of expression, that is, how an expression value will appear.

Break when Expression

The four options in this group allow watches to serve as conditional breakpoints. To simply watch an expression or variable, set [No Break], which is the default. Set Changes if you want program execution to pause when the expression or variable changes value. Set True or False if you want program execution to pause when an expression becomes true or false. You may use "Debug | Change Variables..." to set a variable to a different value and watch execution with the changed variable.

Setting breakpoints

The Breakpoint dialog, Figure 3, is the main window used to set breakpoints.



File Name: C:\C\SEPRG\T.JSE

Line Number: 1

Breakpoints: Line 8, C:\C\SEPRG\T.JSE

Add breakpoint

The Add button adds a breakpoint at the line specified, in the Line Number edit box, to the script specified in the File Name edit box. Of course, the script itself is not altered since scripts are plain text files. Breakpoints are retained as settings within the ScriptEase debugger.

Remove breakpoint

The Remove button removes the breakpoint which is currently highlighted in the Breakpoints list box.

File Name for breakpoint

The File Name edit box indicates which script is presently being used for add and remove operations

Line Number for breakpoint

The Line Number edit box indicates which line in a script is affected by add and remove operations

Breakpoints listing

The Breakpoints list box shows all breakpoints currently active in a script.

Main menu bar

The main menu bar consists of the seven menus across the top of the windows just below the title of bar. The seven menus are: File, Edit, View, Search, Debug, Window, and Help. Some menu commands may be accessed from the toolbar or by shortcut keys, and those that can are indicated by the notations: "In toolbar" and a keystroke description.

File menu

The file menu has options for starting, opening, closing, saving, and printing script files. Plus, an exit option to exit the debugger. All of the commands concerning files operate on script or source files. These files are opened in the integrated editor which allows the use of all debugging options in the integrated debugger. The editor is also a standard editor that can be used to do plain text editing in any text file, such as one created by Notepad.

The editor can be used to write complete scripts. Normally, however, scripters use their favored editors to write and edit most scripts and use the integrated editor while debugging a script.

New — In toolbar and Ctrl+N

Start a new script or source file. The file is opened in the editor which is integrated with all debugging features.

Open... — In toolbar and Ctrl+O

Open dialog to open a script file.

Close — Ctrl+W

Close the currently active script file.

Save — In toolbar and Ctrl+S

Save the currently active script file.

Save As...

Save the currently active script file to a new filename. The title of the currently active script will change to the new filename. Immediately after a script is saved to a new filename, the script will exist in two separate files with the old and new filenames. But, the new filename will be the active script. To edit the previous file, it must be opened again.

Print... — In toolbar and Ctrl+P

Print the currently active script file using straightforward print settings. The print dialog that opens is a standard Windows print dialog.

Print Preview

Preview how the printed script file will look before actually printing the file. When previewing a page, there are various options to page through the pre-printed document, examine pages one or two at a time, zoom in and out, print the document, or close the preview window without printing.

Print Setup...

Change printer settings. These settings are for the printer and are not a page setup. The print setup dialog that opens is a standard Windows print dialog.

(Recent files list)

List up to four of the most recent script files that have been opened in the editor.

Exit

Exit the entire ScriptEase debugger program. Some settings, such as the size and location of open windows is saved. Thus, when the ScriptEase debugger is started again, it is easier to restore various windows to their previous state.

Edit menu

Undo — Ctrl+Z

Undo the last editor operation in the script window.

Cut — In toolbar and Ctrl+X

Cut selected text from the script window.

Copy — In toolbar and Ctrl+C

Copy selected text from the script window.

Paste — In toolbar and Ctrl+V

Paste text at the insertion point, where the cursor is, or into the selection in the script window.

Options

Font...

Display a dialog to set the style, size, and color of the font used in the debugger windows.

Tabs...

Set how many spaces should be used when displaying a tab character in the debugger windows.

Trace On

When a script is run using the Debug | Run in Debugger menu item, the active script runs until it encounters a breakpoint or the script ends. If the Edit | Options | Trace On option is checked, then when a script is run in the debugger, the lines executed are traced. The source marker visibly moves from source line to source line as the script is run. The effect is similar to choosing the Debug | Step Into and Step Over menu items. The difference is that with Trace On checked, the stepping is done automatically.

Trace Speed

When the Trace On menu item is checked, the Trace Speed options determine how fast the trace operation executes each line of a script. The options are: Fast, Normal, Slow, and Slowest.

Trace over

When the Trace On menu item is checked, the Trace Over menu item determines if the tracing steps over functions that are called or steps into them. When Trace Over is checked, the tracer steps over functions, and when it is not checked, the tracer steps into functions.

Source Mark

When debugging a script, the current position in a script is visibly marked by an icon or graphic. The Source Mark option allows a choice of the appearance of the marker.

Default Interpreter...

The default interpreter is the ScriptEase executable that the debugger uses when executing a script. In Win32, the two valid programs are SEwin32.exe and SEcon32.exe. There are differences between a windowed application and a console application. You may want to set the default interpreter to be the same interpreter that you will use to execute a script.

View menu

Toolbar view

View the push button toolbar, just below the menu bar, if checked.

Status Bar view

View the status bar at the bottom of the debugger window. The status bar displays various helpful messages and the position of the cursor or insertion point in the editor in terms of line and column.

Search menu

Find... Ctrl+F

Find text in the script window using a find dialog.

Replace... Ctrl+R

Find text in the script window and replace it with other text using a find and replace dialog.

Debug menu

Start Debug Session

Start executing the active script in a debug session. The source marker is positioned at the first executable line in the script awaiting further commands.

Restart

Restart a debugging session. The source marker is positioned at the first executable line in the script awaiting further commands.

Run in Debugger **In toolbar and F5**

Run the current script in the debugger. The source mark appears. The script executes until a break point is reached or the script is finished.

Go **Ctrl+F5**

Execute the current script as a program, that is, not in the debugger.

Stop **In toolbar**

Stop the execution of a script that is running in the debugger. The script may be actively executing or paused at a source line or breakpoint.

Step Into **In toolbar and F9**

Steps into any user defined functions in the current source line and begins displaying source lines in the function as they are executed. Does not step into built in functions. If a script has not begun execution in the debugger, then the first line of executable code is executed.

Step Over **In toolbar and F10**

Steps over any user defined functions in the current source line and simply executes the line and pauses at the next line in the current script. If a script has not begun execution in the debugger, then the first line of executable code is executed.

Step to Cursor **In toolbar and F11**

Executes all lines of executable code till reaching the line where the cursor is located. In effect, the cursor behaves like a temporary breakpoint.

Step Out **In toolbar and F12**

Executes lines of code in the current function until the function is finished.

Parameters...

Opens a dialog box to set command line parameters to be sent to a script when it is executed in the debugger. The parameters are handled by a script in the same way as they are when part of a command line.

Breakpoint

Toggle current **In toolbar and F8**

Toggle the breakpoint at the current line, off or on.

Add/Remove...

Opens a dialog box to add or remove breakpoints on any line in the current script.

Remove all — In toolbar

Removes all breakpoints in the current script.

Watch

Add/Remove... — Toolbar

Opens a dialog box for adding variables and expressions to the watch window or removing them.

Remove all — Toolbar

Remove all watches from the current script and debugging session.

Change Variables

The menu item allows a variable to be changed while a script is executing.

Window menu

Cascade

Display the open windows in the debugger in a cascaded fashion.

Tile

Tile open windows horizontally. If two or three windows are open, they are all tiled horizontally extending the entire width of the main debugger window. If four or more windows are open, then two columns of windows are begun, and all windows are tiled horizontally in the two columns. For example, if a script window, the global, the local, and the watch window are opened, the resulting window is quartered. Each window will be in the four corners of the main window. The screen shot, Figure 1, at the beginning of this section is an example of four tiled windows.

Arrange Icons

As in all MDI applications, open windows may be minimized inside the main window. The Arrange Icons menu item arranges these minimized icons at the bottom of the main debugger window.

Global... — Ctrl+Shft+G

Open the Globals window to view global variables while debugging a script.

Local... Ctrl+Shft+L

Open the Locals window to view local variables while debugging a script.

Watch... Ctrl+Shft+W

Open the Watches window to view variables and expressions that have been defined by a user.

(Open windows list)

A list of the currently open windows in the debugger.

Help menu

Help Topics... F1

Display a help file for the debugger.

About ScriptEase Debugger... In toolbar

Displays program information, version number, and copyright notice for the debugger.

ScriptEase versus C language

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

The assumption here is that readers of this section already know C. Thus, only those aspects of the C portion of ScriptEase that differ from C are described. If something is not mentioned here, ScriptEase follows standard C behavior. While in this section on the differences from C, the term ScriptEase is used for the portion of ScriptEase that implements the standard C library and ScriptEase additions to that library. Almost all of the implementation of C in ScriptEase involves the use of Clib objects, SElib objects, or cfunctions. Thus, references to ScriptEase as the C portion of ScriptEase usually involve Clib, SElib, or cfunction. The differences between a function and a cfunction are also discussed in the section on ScriptEase JavaScript.

Deviations from C result from following several principles:

- simplicity
- power
- safety

The C portion of ScriptEase is different from C where changes make ScriptEase more convenient for scripting, writing small programs, and entering command line code or where unaltered C rules encourage coding that is potentially unsafe. Keep in mind, that most issues involved in this section involve the use of Clib, SElib, and cfunction.

The C portion of ScriptEase is C without type declarations and pointers. If you already know C and can forget these two aspects of C while using ScriptEase, then you already know the C portion of ScriptEase. If you were to take C code and delete all the lines, code words, and symbols that either declare data types or explicitly point to data, then you would be left with code that would work with Clib, SElib, and cfunction. Though you would be altering source code, you would not be removing capabilities.

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

Data types in C and SE

ScriptEase uses the same data types as JavaScript.

Automatic type declaration

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

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

could be converted to the following ScriptEase code:

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

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

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

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

Array representation

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

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

The following line creates an automatic array in ScriptEase.

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

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

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

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

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

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

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

Automatic array allocation

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

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

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

Literal strings

A literal string in ScriptEase is any array of characters, that is, a string, appearing in source code within double, single, or back quotes. Back quotes are sometimes referred to as back-ticks. The following lines show examples of literal strings in ScriptEase:

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

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

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

```
TestStr == "MYLONGPASSWORD"
```

is simpler than :

```
Clib.strcmp(TestStr, "MYLONGPASSWORD").
```

In general, literal strings adhere to the two following rules.

- Comparisons are intrinsically handled by Clib.strcmp()

- Assignment and passing of literal strings is done by making copies of the literal string

Literal strings and assignments

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

```
for (var i = 0; i < 3; i++)
{
   var str = "dog";
   Clib.strcat(str, "house");
   Clib.puts(str);
}
```

results in the following output:

```
doghouse
doghouse
doghouse
```

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

```
doghouse
doghousehouse
doghousehousehouse
```

Literal strings and comparisons

If both sides of a comparison operator are strings, and at least one of them is a literal string, then the comparison is performed as if `Clib.strcmp()` were being used. If one or both variables are literal strings, then the following translation of the comparison operation is performed.

```
lvar operator rvar    Clib.strcmp(lvar, rvar) operator 0
```

The following examples demonstrate how literal strings follow the logic of `Clib.strcmp()`.

```
if (animal == "dog")  // if (Clib.strcmp(animal, "dog") == 0)
if (animal <  "dog")  // if (Clib.strcmp(animal, "dog") <  0)
if ("dog" <= animal)  // if (Clib.strcmp("dog", animal) <= 0)
```

In ScriptEase, the following fragment:

```
var animal = "dog";
if (animal == "dog")
Clib.puts("hush puppy");
```

displays:

```
"hush puppy"
```

Literal strings and parameters

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

```
for (var i = 0; i < 3; i++)
{
   var str = Clib.strcat("dog", "house");
   Clib.puts(str)
}
```

results in the following output:

```
doghouse
doghouse
doghouse
```

Literal strings and returns

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

```
for (var i = 0; i < 3; i++)
{
   var str = Clib.strcat(dog(),"house");
   Clib.puts(str)
}

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

results in the following output:

```
doghouse
doghouse
doghouse
```

Literal Strings and switch statements

If either a switch expression or a case expression is a literal string, then the case statement match is based on a string comparison using `Clib.strcmp()` logic. The following fragment illustrates.

```
switch(Clib.strlwr(temp, argv[1]))
{
case "add":
{
   DoTheAddThing();
   break;
}
case "remove":
{
   DoTheRemoveThing();
   break;
}
default:
{
   Clib.puts("Whaddya want?");
}
}
```

Structures

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

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

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

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

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

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

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

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

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

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

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

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

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

Passing variables by reference

By default, lvalues in ScriptEase are passed to cfunctions by reference. If a cfunction alters a variable, then the variable passed as an argument by the calling routine is altered also, if it is an lvalue. So instead of the following C code which uses address and pointer operators:

```
main()
{
   CQuadrupleInPlace(&i);
   ...
}

void CQuadrupleInPlace(int *j)
{
   *j += 4;
}
```

a ScriptEase conversion could be:

```
function main()
{
   ...
   QuadrupleInPlace(i);
   ...
}

cfunction QuadrupleInPlace(j)
{
   j += 4;
}
```

The following calls to QuadrupleInPlace() are valid in ScriptEase, but the values passed as arguments are not changed after QuadrupleInPlace() is called. Why? None of the arguments being passed are lvalues.

```
QuadrupleInPlace(8);
QuadrupleInPlace(i+1);
QuadrupleInPlace(8+1);
```

Variables may not be passed by value to **cfunctions**. However, general ScriptEase allows **functions** to have primitive types passed values by value or by reference, though composite data types can be passed only by reference. See the sections on data types, passing information to functions, and passing information to cfunctions for more information.

Pointer operator * and address operator &

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

Further, in cfunctions, address arithmetic may be used to simulate some to the functionality of pointers. The following function displays the string in the variable s. In the first display line shows:

```
abcde
```

The second display line, which uses address arithmetic "s+2" shows:

```
cde

cfunction main(argc, argv)
{
   var s = "abcde";
   Screen.writeln(s);
   Screen.writeln(s+2);
}
```

Remember that in cfunctions, variables are passed by reference. In functions (not cfunctions), all variables, except primitive data types, are passed by reference. ScriptEase adds the address operator `&` for primitive data types. If you want to pass a primitive data type by reference in a JavaScript function, use the address operator in the parameter list. For example,

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

Remember, the address operator `&` is for functions, not cfunctions.

Case statements

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

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

As described in the section on literal strings above, if either a switch expression or a case expression is a literal string, then any comparisons are based on the logic of `Clib.strcmp()`, that is, as if the comparisons were `!Clib.strcmp(switch_expr, case_expr)`.

Initialization code which is external to functions

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

```
Clib.printf("first ");

cfunction main()
{
   Clib.printf("third.");
}

Clib.printf("second ");
```

results in the following output:

```
first second third.
```

Unnecessary tokens

If symbols are redundant, they are usually unnecessary in ScriptEase which allows more flexibility in writing scripts and is less onerous for users not trained in C. Semicolons that end statements are usually redundant and do not do anything extra when a script is interpreted. C programmers are trained to use semicolons to end statements, a practice that can be followed in ScriptEase.

Indeed, some programmers think that the use of semicolons in ScriptEase and JavaScript is a good to be pursued. Many people who are not trained in C wonder at the use of redundant semicolons and are sometimes confused by their use. The use of semicolons is personal. If a programmer wants to use them, then he should, but if he does not want to, then he should not.

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

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

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

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

The fragments could be rewritten to be:

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

and:

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

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

Macros

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

Token replacement macros

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

```
#define NULL 0
```

Back quote strings

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

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

Converting existing C code to ScriptEase

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

C **ScriptEase**

```
int i;                              var i; // or nothing
int foo = 3;                        var foo = 3;
struct                              var st; // no struct type
{                                      // Simply use st.row
   int row;                            // and st.col
   int col;                            // when needed.
}
char name[] = "George";             var name = "George";
int goo(int a, char *s, int c);     var goo(a, buf, c);
int zoo[] = {1, 2, 3};              var zoo = {1, 2, 3};
```

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

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

Array object

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

An Array is a special class of object that refers to its properties with numbers rather than with variable names. Properties of an Array object are called elements of the array. The number used to identify an element is called an index in brackets which follows an array name. Array indices must be either numbers or strings.

Array elements can be of any data type. The elements in an array do not all need to be of the same type, and there is no limit to the number of elements an array may have.

The following statements demonstrate assigning values to arrays.

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

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

Since arrays use a number to identify the data they contain, they provide an easy way to work with sequential data. For example, suppose you wanted to keep track of how many jelly beans you ate each day, so you can 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 all your data stored conveniently 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++)
   Screen.write("On April " + x + " I ate " + April[x] +
      " jellybeans.\n");
```

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

Creating arrays

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

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

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

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

In this case, an array with length 31 is created.

Finally, you can pass a list of elements to the `Array()`function, which creates an array containing all of the parameters passed. For example:

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

creates an array with a length of 6. c[0] is set to 5, c[1] is set to 4, and so on up to c[5], which is set to the string "blast off". Note that the first element of the array is array[0], not array[1].

Arrays may also be created dynamically. By referring to a variable with an index in brackets, a variable is created as or converted to an array. The array that is created is an automatic or dynamic array which is different than an instance of an Array object created as described in this section. Automatic arrays, created as described in this paragraph, are unable to use the methods and properties described below, so it is recommended that you use, in most circumstances, the `new Array()` constructor function to create arrays.

Initializers for arrays and objects

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

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

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

Arrays may initialized using a syntax similar to the following:

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

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

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

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

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

The following code fragment shows the differences.

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

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

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

The display from this code is:

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

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

Array object instance properties

Array length

SYNTAX: array.length

DESCRIPTION: The length property returns one more than the largest index of the array. Note that this value does not necessarily represent the actual number of elements in an array, since elements do not have to be contiguous.

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

SEE: Array(), getArrayLength(), setArrayLength()

EXAMPLE:

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

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

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

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

Array object instance methods

Array() with length

SYNTAX:	new Array(length)
WHERE:	length - If this is a number, then it is the length of the array to be created. Otherwise, it is the element of a single-element array to be created.
RETURN:	object - an Array object of the length specified.
DESCRIPTION:	The array returned from this function is an empty array whose length is equal to the `length` parameter. If `length` is not a number, then the length of the new array is set to 1, and the first element is set to the `length` parameter. Note that this can also be called as a function, without the new operator.
SEE:	Automatic arrays
EXAMPLE:	`var a = new Array(5);`

Array() with list

SYNTAX:	new Array([element1, ...])
WHERE:	elementN - list of elements to be in the new Array object being created.
RETURN:	object - an Array object with the elements specified.
DESCRIPTION:	This function is an alternate form of the Array constructor which initializes the elements of the new array with the arguments passed to the function. The arguments are inserted in order into the array, starting with element 0. The length of the new array is set to the total number of arguments. If no arguments are supplied, then an empty array of length 0 is created.
SEE:	See Array()
EXAMPLE:	`var a = new Array(1,"two",three);`

Array concat()

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

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

Array join()

SYNTAX:	array.join([separator])
WHERE:	separator - a value to be converted to a string and used to separate the list of array elements. The default is an empty string.
RETURN:	string - string consisting of the elements, delimited by separator, of an array.
DESCRIPTION:	The elements of the current object, from 0 to the length of the object, are sequentially converted to strings and appended to the return string. In between each element, the separator is added. If `separator` is not supplied, then the single-character string "," is used. The string conversion is the standard conversion, except the undefined and null elements are converted to the empty

string "".

The join() method creates a string of all of array elements. The join() method has an optional parameter, a string which represents the character or characters that will separate the array elements. By default, the array elements will be separated by a comma. For example:

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

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

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

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

SEE: Array.toString()

EXAMPLE:

```
// The following code:

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

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

Array pop()

SYNTAX: array.pop()

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

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

SEE: Array.push()

EXAMPLE:

```
// The following code:
```

```
var array = new Array( "four" );
Screen.writeln( array.pop() );
Screen.writeln( array.pop() );

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

Array push()

SYNTAX: array.push([element1, ...])

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

RETURN: number - the length of the new array.

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

SEE: Array.pop()

EXAMPLE:

```
// The following code:

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

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

Array reverse()

SYNTAX: array.reverse()

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

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

EXAMPLE:

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

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

//produces the following array:

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

Array shift()

SYNTAX: array.shift()

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

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

SEE: Array.unshift(), Array.pop()

EXAMPLE:

```
//The following code:

var array = new Array( 1, 2, 3 );
Screen.writeln( array.shift() );
Screen.writeln( array );

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

Array slice()

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

WHERE: start - the element offset to start from.

end - the element offset to end at.

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

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

SEE: String.substring()

EXAMPLE:

```
// The following code:

var array = new Array( 1, 2, 3, 4 );
Screen.writeln( array.slice( 1, -1 ) );

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

Array sort()

SYNTAX: array.sort([compareFunction])

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

RETURN: object - this Array object after being sorted.

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

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

- If compareFunction(a, b) is less than zero, sort b to a lower index than a.
- If compareFunction(a, b) returns zero, leave a and b unchanged to each other.
- If compareFunction(a, b) is greater than zero, sort b to a higher index than a.

By specifying the following function as a sort function, you will get the desired result when comparing numbers:

```
 function compareNumbers(a, b)
 {
    return a   b
 }
```

SEE: Clib.strcmp()

EXAMPLE:

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

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

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

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

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

Array splice()

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

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

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

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

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

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

SEE: Array.push()

EXAMPLE:

```
// The following code:

var array = new Array( 1, 2, 3, 4, 5 );
Screen.writeln( array.splice( 1, 2, 6, 7, 8 );
Screen.writeln( array );

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

Array toString()

SYNTAX: array.toString()

RETURN: string - string representation of an Array object.

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

SEE: Array.join()

EXAMPLE:

```
// The following code:

var array = new Array( 1, "two", , null, false );
Screen.writeln( array.toString() );

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

Array unshift()

SYNTAX: array.unshift([element1, ...])

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

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

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

SEE: Array.shift(), Array.push()

EXAMPLE:

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

Blob Object

This section describes Blobs, Binary Large Objects.

The methods in this section are preceded with the Object name Blob, since individual instances of the Blob Object are not created. For example, `Blob.get()` is the syntax to use to get data from a Blob. Blob and Buffer variables overlap. The Buffer is the newer construct, and the Blob is retained mostly for compatibility with previous versions of ScriptEase. When necessary to work with data in memory, use a Buffer object if possible.

Blob object static methods

Blob.get()

SYNTAX:	Blob.get(BlobVar, offset, DataType)
	Blob.get(BlobVar, offset, bufferLen)
	Blob.get(BlobVar, offset, DataStructureDefinition)

WHERE:	BlobVar - binary large object variable to use. offset - the offset or position in the Blob from which to work. DataType - the type of data with which to work. bufferLen - the length of data to work with as a buffer or byte array. DataStructureDefinition - definition of a structure (object) variable.
RETURN:	value - the data retrieved according to the defining parameters.
DESCRIPTION:	This method reads data from a specified location of a Binary Large Object, a Blob and is the companion function to `Blob.put()`. The parameter BlobVar specifies the Blob to use. The parameter offset specifies where, in the Blob, to get data. The last parameter specifies the format of the data in the Blob and, hence, determines the type of the value returned which is the data read from the Blob.

Valid values for DataType are:

```
UWORD8,  SWORD8,  UWORD16, SWORD16, UWORD24, SWORD24,
UWORD32, SWORD32, FLOAT32, FLOAT64, FLOAT80
```

See `Clib.fread()` or blobDescriptor object, below, for more information on these DataType values.

SEE: Blob.put(), Blob.size(), _BigEndianMode, Buffer object

Blob.put()

SYNTAX: Blob.put(BlobVar[, offset], variable, DataType)

Blob.put(BlobVar[, offset], buffer, bufferLen)

Blob.put(BlobVar[, offset], SrcStruct, DataStructureDefinition)

WHERE: BlobVar - binary large object variable to use.

offset - the offset or position in the Blob from which to work.

variable - variable with data to put into a Blob.

buffer - buffer with data to put into a Blob.

SrcStruct - structure (object) with data to put into a Blob.

DataType - the type of data with which to work.

bufferLen - the length of data to work with as a buffer or byte array.

DataStructureDefinition - definition of an object (structure) variable.

RETURN: number - the byte offset to the next byte following the data that was just inserted into a Blob. If at the end of a Blob, then return the value that equals `Blob.size(Blob)`.

DESCRIPTION: 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 variable. The contents of such a variable may be viewed as a packed structure. Data can be placed at any location within a Blob. The parameter BlobVar specifies the Blob to use. The parameter offset specifies where, in the Blob, to write data. The third parameter is the data to write. The last parameter specifies the format of the data in the Blob.

`Blob.put()` returns the byte offset for the next byte following the section where data was just put. If the data is put at the end of the Blob, then the return is equivalent to the size of the Blob.

If 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 in v is converted to the specified DataType and then copied into the bytes specified by offset.

If DataType is not the length of a byte buffer, then it must be one of these types:

```
UWORD8,  SWORD8,  UWORD16, SWORD16, UWORD24, SWORD24,
UWORD32, SWORD32, FLOAT32, FLOAT64, FLOAT80
```

See `Clib.fread()` or blobDescriptor object, below, for more information on these DataType values.

SEE: Blob.get(), Blob.size(), _BigEndianMode, Buffer object

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 DOS 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:

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:

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

Blob.size()

SYNTAX: Blob.size(BlobVar[, SetSize])

Blob.size(DataType)

Blob.size(bufferLen)

Blob.size(DataStructureDefinition)

WHERE: BlobVar - binary large object variable to use.

SetSize - size to which to set BlobVar.

DataType - the type of data with which to work.

bufferLen - the length of data to work with as a buffer or byte array.

DataStructureDefinition - definition of a structure (object) variable.

RETURN: number - bytes in a Blob variable. If SetSize is passed, then that value is returned.

DESCRIPTION: This method determines the size of a Binary Large Object, Blob. 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 DataStructureDefinition are used, Blob.size() returns the size of a Blob that would contain the type of data item used in by Blob.get() or Blob.put(). In these cases, these parameters specify the type to be used for converting ScriptEase data to and from a Blob.

Blob.size returns the size of a Blob which is the number of bytes in BlobVar. If SetSize is supplied, then the return is SetSize.

SEE: Blob.get(), Blob.put(), _BigEndianMode, Buffer object

blobDescriptor object

When an object (structure) needs to be sent to a process other than the ScriptEase 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 to be sent. This description tells how the properties of the object are stored in memory and is used with functions, such as `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. 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 defined above:

```
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 example above, 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`.

The following values may be used for blobDescriptors.

`UWORD8`	Stored as a byte
`SWORD8`	Stored as an integer
`UWORD16`	Stored as an integer

`SWORD16`	Stored as an integer
`UWORD24`	Stored as an integer
`SWORD24`	Stored as an integer
`UWORD32`	Stored as an integer
`SWORD32`	Stored as an integer
`FLOAT32`	Stored as a float
`FLOAT64`	Stored as a float
`FLOAT80`	Stored as a float (not available in Win32)

If a 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.

Number Object

```
platform: All OS, All version of SE
```

Number object instance methods

Number toLocaleString()

SYNTAX:	number.toLocaleString()
RETURN:	string - a string representation of this number.
DESCRIPTION:	This method behaves like Number.toString() and converts a number to a string in a manner specific to the current locale. Such things as placement of decimals and comma separators are affected.
SEE:	Number.toString()

EXAMPLE:	`var n = 8.9;`
	`var s = n.toLocaleString();`

Number toString()

SYNTAX:	number.toString()
RETURN:	string - a string representation of this number.
DESCRIPTION:	This method behaves similarly to Number.toLocaleString() and converts a number to a string using a standard format for numbers.
SEE:	Number.toLocaleString()

EXAMPLE:	`var n = 8.9;`
	`var s = n.toString();`

SElib Object

The methods in the SElib object extend the functionality of JavaScript. Whereas the Clib object extends the power of JavaScript by providing functions from the standard C library, the SElib extends power by allowing programmers to work with such things as directories, files, memory, windows, messages, system operations, and script execution. The methods in the SElib object are more like the C functions in the Clib object than JavaScript functions.

When using the methods in this section, they are preceded with the Object name SElib, since individual instances of the SElib Object are not created. For example, SElib.directory() is the syntax to use to get directory information in a script.

SElib object static methods

SElib.baseWindowFunction()

SYNTAX:	SElib.baseWindowFunction(hWnd, message, param1, param2)
WHERE:	hWnd - a number, a handle of the window receiving the message. message - a number, a Windows message ID. param1 - the first parameter of the message ID. param2 - the second parameter of the message ID.
RETURN:	value - the value returned by the base window function. If the parameter handle is not a window with a windowFunction created with SElib.makeWindow() or is not a window subclassed with SElib.subclassWindow(), then the return is 0.
DESCRIPTION:	Calls the base procedure of a window created with a windowFunction in SElib.makeWindow() or subclassed with SElib.subclassWindow(). This method is normally used within a ScriptEase window function to pass the window parameter to the base procedure before handling it in your own code. Remember that if your window function returns no value, ScriptEase will

	call the base procedure automatically which is the preferred method.
SEE:	SElib.makeWindow(), SElib.subclassWindow(), Window object in winobj.jsh

SElib.breakWindow()

SYNTAX:	SElib.breakWindow(hWnd)
WHERE:	hWnd - a number, the handle of the window being released or destroyed.
RETURN:	boolean - true on success and the window is successfully destroyed, released, or subclassed, else false on failure.
DESCRIPTION:	For Win32 and Win16 Releases control of a window controlled by SElib.subclassWindow() or destroys a window previously created with SElib.makeWindow(). No other windows are affected. If hWnd is not a valid window handle, no action is taken and true is returned. When a window is destroyed all appropriate *`DestroyWindow()`* functions, internal to the Windows API, are called. Any child windows of a main window are destroyed before the main window. If hWnd is a window controlled by SElib.subclassWindow(), then this method removes the WindowFunction for a window from the message function loop. If hWnd is not supplied, then all windows created with SElib.makeWindow() are destroyed and all subclassing ends.
SEE:	SElib.makeWindow()

SElib.compileScript()

SYNTAX:	SElib.compileScript(codeToCompile[, isFile])
WHERE:	codeToCompile - a string with ScriptEase statements or a

filename of a script file.

isFile - a boolean telling whether or not codeToCompile is a filename or a string with statements. The default is false indicating that codeToCompile is a string consisting of ScriptEase statements.

RETURN: buffer - the compiled code in a ScriptEase buffer. Normally, this buffer of compiled code is saved to a file.

DESCRIPTION: Compiles a ScriptEase script into executable code which is normally written to a file with an extension of ".jsb" and referred to as a ScriptEase binary file. This compiled code is the same code that is created when the /bind option is used with the Pro version of ScriptEase Desktop and the code is bound in an executable ".exe" file.

Compiled code may be executed in two ways. First, the compiled code may be passed to the SElib.interpret() method as the Code parameter. The SElib.interpret() method executes compiled code in the same way that it does text script. Second, a ScriptEase binary file may be executed by a ScriptEase interpreter, such as sewin32.exe. This second way is the most common way to execute compiled code. There are three basic ways that a ScriptEase script file may be run:

- A text script, as typed by a programmer, may be called using an interpreter program, such as sewin32.exe. The interpreter reads the text and performs all the statements in it. Running a script in this way results in the slowest overall execution speed since the interpreter must preprocess, tokenize, and run the file.
- A text script may be compiled using the SElib.compileScript() method and written to a ScriptEase binary file. A ScriptEase binary file may also be called by an interpreter program, such as sewin32.exe. But overall execution time is faster since the first two steps, preprocessing and tokenizing, are already done by SElib.compileScript(). The compiled code of a script is the same as the compiled code of an executable file produced using the /bind option of the Pro version.
- A text script can be compiled using the /bind option of the

Pro version. The script is compiled, into the same form as when using SElib.compileScript() but is physically attached to the pertinent executable part of an interpreter, such as sewin32.exe. The compiled file is an executable file with an extension of ".exe" and can be run as a stand alone program.

See the section on "Running a script" in the manual or help file for more information on executing ScriptEase scripts.

ScriptEase binary files are called in the same way as text scripts, either ".jse" or ".jsh" files. Assume that a file named testobj.jse has been compiled with SElib.compileScript() to testobj.jsb. The invocations of either file by an interpreter do the same thing. For example, both lines below accomplish the same thing when run as a command line.

```
sewin32.exe testobj.jse sewin32.exe testobj.jsb
```

The second line using ".jsb" executes faster, in overall time, that is, it begins executing more quickly.

In a like manner, assume that a file named testinc.jsh has been compiled with SElib.compileScript() to testinc.jsb. Either file may be included in a script using the preprocessor directive `#include`. Both lines of script below accomplish the same thing.

```
#include "testinc.jsh" #include "testinc.jsb"
```

The second line executes faster since the code in that file is precompiled. This include example points to another difference between the /bind option and the SElib.compileScript() method. The /bind option results in a stand alone executable file. The SElib.compileScript() method allows the flexibility of precompiling sections of code that may be used in other scripts or of having a complete precompiled program. Complete programs compiled by either method execute at the same speed, at actual run time.

A compiled ScriptEase binary file may also be run from a script by using the SElib.interpret() method, using the INTERP_COMPILED_SCRIPT flag.

A ScriptEase binary file has 4 bits that identify it as a compiled script and 16 bytes for a checksum to make sure that the file has not been altered. Compiled scripts are implemented at a very low level which allows ScriptEase binary files to be included in a script, as already described. But, there is another benefit. A programmer may use file extensions other than the default ".jsb".

ScriptEase comes with a script, compile.jse, which automates the process of compiling a text script to a ScriptEase binary file.

SEE: SElib.interpret(), SElib.interpretInNewThread(), compile.jse

EXAMPLE:

```
   // Compile the script file, myscript.jse,
   // to the ScriptEase
   // binary file, myscript.jsb.
function main(argc, argv)
{
      // Filename of the script to compile
   var infile  = "Myscript.jse";
      // Filename for the compiled code
   var outfile = "Myscript.jsb";

      // Compile the script file
      // into compiled code.
      // Argument true indicates that infile is a
filename
   var compiledScript = SElib.compileScript(infile,
true);

      // If the returned buffer has code in it,
      // save it to a file.
   if( compiledScript != null )
   {
      var outfp = Clib.fopen(outfile, "w");
      if( outfp == null )
      {
         Clib.fprintf(stderr,
            "Could not open file \"%s\"\n",
            outfile);
         Clib.fclose(outfp);
      }
      else
      {
         Clib.fwrite(compiledScript,
            getArrayLength(compiledScript), outfp);
         Clib.fclose(outfp);
      }
   }
}
```

SElib.directory()

SYNTAX: SElib.directory([filespec[, subdirs[, includeAttr[, requireAttr]]]])

WHERE: filespec - string specification for files to find. The specification must be consistent with the operating system being used and may include wildcard characters. A file specification may include path specifications, both full and partial.

subdirs - a boolean as to whether or not to include subdirectories in file search. The default is false, which limits the search for filespec to the current directory.

includeAttr - specify the file attributes to include in the file search. Only files with one of the attributes specified will be included in the array of file names and information retrieved. Attribute flags that do not apply to an operating system are ignored. If includeAttr is 0, only files with no attributes are included. The default value is:

```
FATTR_RDONLY|FATTR_SUBDIR|
FATTR_ARCHIVE|FATTR_NORMAL
```

File attributes are set using the following values:

```
FATTR_RDONLY   Read-only file
FATTR_HIDDEN   Hidden file
FATTR_SYSTEM   System file
FATTR_SUBDIR   Directory
FATTR_ARCHIVE  Archive file
```

More than one file attribute can be specified by using the bitwise or operator, "|". For example, to find files with the hidden or system attributes set, use the following expression:

```
FATTR_HIDDEN | FATTR_SYSTEM
```

A file attribute may be excluded from array of files returned by using the bitwise not operator, "~". For example, to exclude subdirectories, use the following expression:

```
~FATTR_SUBDIR
```

requireAttr - specify attributes that files are required to have to be included in the array of file names and information retrieved. Files must have at least these attributes. The difference between the two file attributes specifications is that files must have at least one of the attributes specified by includeAttr but must have all the attributes specified by requireAttr. The default value is 0.

RETURN: object - an array of objects with information about the file names retrieved. If no files or directories match the specifications of the parameters, a null is returned. Each element of the array has the following properties:

```
.name    Full file name, including the filespec
path.
.attrib  File flags, as defined above in IncAttr.
.size    Size of file, in bytes.
.access  Date and time of last file access.
.write   Date and time of last write to file.
.create  Date and time of file creation.
```

For example, if you use the following line of code:

```
var FileList = SElib.directory("*.*");
```

The information for the first file retrieved is accessed using:

```
FileList[0].name
FileList[0].attrib
FileList[0].size
FileList[0].access
FileList[0].write
FileList[0].create
```

The information for the second file is accessed using:

```
FileList[1].name
...
```

DESCRIPTION: Find files in a directory or subtree that match path and file specifications and have specified file attributes set. Remember the directory names are treated like file names and have the FATTR_SUBDIR attribute set. Matching files and information about them are retrieved and returned in an array of objects. These objects are also structures.

This method may be used in many ways. One way, besides the obvious way of getting information about files, is to test for the existence of a file or file specification. If the file specified does not exist, the return is null.

SEE: SElib.fullpath(), SElib.splitFilename, File object in fileobj.jsh

EXAMPLE:

```
   // The following routine lists
   // all files matching FileSpec,
   // except subdirectory entries,
   // in the current directory of a script.
function ListDirectory(FileSpec)
{
   var FileList = SElib.directory(FileSpec, False,
      ~FATTR_SUBDIR)
   if (null == FileList)
      Clib.printf(
        "No files found for search spec \"%s\".\n",
        FileSpec)
   else
   {
      var FileCount = getArrayLength(FileList);
      for (var i = 0; i < FileCount; i++)
         Clib.printf(
          "%s\tsize = %d\tCreate date/time = %s\n",
          FileList[i].name, FileList[i].size,
          Clib.ctime(FileList[i].Create));
   }
}
```

SElib.doWindows()

SYNTAX: SElib.doWindows(immediateReturn)

WHERE: immediateReturn - if true return immediately, regardless of messages. Default is false.

RETURN: boolean - true if any of the windows created with SElib.makeWindow() or subclassed with SElib.subclassWindow() are still open, that is, have not received a WM_NCDESTROY message. Returns false if there are no valid windows registered with the ScriptEase Window Manager.

DESCRIPTION: For Win32 and Win16

Starts the ScriptEase Window Manager to activate whatever windows have been created or subclassed with SElib.makeWindow() or SElib.subclassWindow. All such windows are registered with the Window Manager. The Window Manager controls the messages sent to the windows in its registry and routes them to their respective window functions.

There should not be more than one copy of the Window Manager running at a time. Generally, SElib.doWindows() is called only once with a succession of windows. All windows created or subclassed after a call to SElib.doWindows() are automatically registered with the Window Manager.

The flags that define window messages are kept in the library file, message.jsh.

If the optional parameter immediateReturn is true, the method returns immediately, regardless of whether there are messages for this application or not. Otherwise this method yields control to other applications until a message has been processed, subject to filtering by SElib.messageFilter(), for this application or for any window subclassed by this application.

The example below displays a standard Windows window. If you click anywhere in the window, the string "You clicked me!" is displayed briefly in the middle of the window. When the window is closed, the script terminates.

SEE: SElib.makeWindow(), SElib.subclassWindow(), Window object in winobj.jsh

EXAMPLE:

```
#include <message.jsh>
#include <window.jsh>
function main()
{
   var hWnd = SElib.makeWindow(null, null,
      WindowFunction, "Display Windows' messages",
      WS_OVERLAPPEDWINDOW | WS_VISIBLE,
      CW_USEDEFAULT, CW_USEDEFAULT,
      500, 350, null, 0);
   SElib.messageFilter(hWnd, WM_LBUTTONDOWN);
   while(SElib.doWindows()) ;
}

function WindowFunction(hWnd, msg, param1,
                        param2, counter)
{
   if (msg == WM_LBUTTONDOWN)
   {
      var msgHwnd = SElib.makeWindow(hWnd,
         "static", null, "You clicked me!",
         WS_CHILD | WS_VISIBLE,
         200, 150, 100, 50, null, 0);
         SElib.suspend(1000);
         SElib.breakWindow(msgHwnd);
   }
}
```

SElib.fullpath()

SYNTAX: SElib.fullpath(pathspec)

WHERE: pathspec - a partial path specification.

RETURN: string - the pathspec filled out to its full path specification or null if the path specification is invalid.

DESCRIPTION: Converts pathspec to a full and absolute path specification. The file name part of the path specification is not affected and may have wildcards. The drive and directory part of the path specification is converted or fleshed out to a full and absolute path.

The exact behavior of SElib.fullpath() depends on the underlying operating system. Some results can vary when using system specific path specifications.

SEE: SElib.directory(), SElib.splitFilename(), File object in fileobj.jsh

EXAMPLE:

```
   // The following returns the full spec
   // of current dir
function CurDir()
{
   return SElib.fullpath(".")
}
   // The following returns the full spec
   // of a parent dir
function CurDir()
{
   return SElib.fullpath("..\")
}
   // The following works in DOS or OS/2
   // to test whether a drive
   // letter is valid
function ValidDrive(DriveLetter)
{
   Clib.sprintf(CurdirSpec, "%c:.", DriveLetter)
   return (null != SElib.fullpath(CurdirSpec) )
}
```

SElib.getObjectProperties()

SYNTAX: SElib.getObjectProperties(object[, includeUndefined])

WHERE: object - an object from which to get its properties.

includeUndefined - a boolean, determines whether or not to include properties with undefined values. The default is false, that is, do not include properties with undefined values.

RETURN: object - an array of strings which are the names of the properties of the object. The array is terminated with a null, that is, the last element is always null.

DESCRIPTION: Get the names of the properties of an object in an array of strings in which each element is a property name and the last element is null.

The parameter includeUndefined must be true to return properties that are not defined. If includeUndefined is false, then only properties that have defined data are included. The default for includeUndefined is false.

The final member of the returned array returned is always null. If the parameter object is not defined or contains no properties, then the return is an array with a single element set to null.

SEE: For/in statement

EXAMPLE:

```
var Point;
Point.row = 5;
Point.col = 8;
Point.height;
DisplayAllStructureMembers(Point);

function DisplayAllStructureMembers(ObjectVar)
{
   Screen.writeln("Object Properties:");
   var MemberList =
SElib.getObjectProperties(ObjectVar);
   for (var i = 0; MemberList[i]; i++)
   Clib.printf("  %s\n", MemberList[i]);
}
```

```
// This fragment produces the following output.
// Object Properties:
//   row
//   col
```

SElib.inSecurity()

SYNTAX: SElib.inSecurity(infoVar)

WHERE: infoVar - variable to be passed to the ScriptEase security filter. Your application and its security filter may use it however you choose.

RETURN: boolean - true if there is a security filter, else false.

DESCRIPTION: Calls the security manager's initialization routine and is the only way your application can directly interact with the security filter. It is provided so you can reinitialize the security system, probably to change the security level of a script.

Typically, you use this method when executing a particularly insecure piece of code, such as a script received over a network, to downgrade the security level, restoring it when the script completes.

SElib.instance()

SYNTAX: SElib.instance()

RETURN: number - instance handle of the current ScriptEase session, that is, for the current script.

DESCRIPTION: For Win32

Get the instance handle of the currently executing script. This handle may be used with Windows API functions that use an instance handle.

SEE: Screen.handle(), SElib.makeWindow(), icon.jsh, pickfile.jsh, dropper.jse, iconmany.jse

EXAMPLE:

```
var hScript = SElib.instance()
```

SElib.interpret()

SYNTAX: SElib.interpret(codeToInterpret[,howToInterpret[,security]])

WHERE: codeToInterpret - a string with ScriptEase code statements to be interpreted as script statements or the file specification, path and file name, of a script file. If the interpreted code receives arguments, they are put at the end of the codeToInterpret string--somewhat like a command line string.

howToInterpret - tells how to handle the interpreted code. The following flag values may be combined using the bitwise or operator, "`|`". The value must be 0 or one of the following choices:

- `INTERP_FILE`
  CodeToInterpret is the file name of a script, followed by any arguments.
- `INTERP_TEXT`
  CodeToInterpret is a string of source code with no arguments attached.
- `INTERP_LOAD`
  Load code into same function and variable space as the script that is calling `SElib.interpret()`. All functions, and variables are supplied to the code being called, which can modify and use them. If the code being called has similarly named functions or variables as the calling code, functions in the called code replace those in the calling code.
- `INTERP_NOINHERIT_LOCAL`
  Local variables are not inherited by the interpreted code.
- `INTERP_NOINHERIT_GLOBAL`
  Global variables are not inherited by the interpreted code as globals.
- `INTERP_COMPILED_SCRIPT`
  Run a script compiled with SElib.compileScript().This flag only works with the `INTERP_TEXT` flag.

`INTERP_FILE` and `INTERP_TEXT` are mutually exclusive. If neither is supplied the interpreter decides whether codeToInterpret is a file or string of code.

These flags tell the computer how to interpret the parameter codeToInterpret. If one is not supplied, the computer parses the string and determines the most appropriate way to interpret it.

security – the filename of the security script to run this interpreted script using. This is exactly like the security script passed to SE:Desk using the '/secure=' option, except it applies only to the script you are about to interpret. Remember that security is additive; any existing security is still in effect for the interpreted script as well.

RETURN: value - the return of the interpreted code.

DESCRIPTION: Interprets a string as if it were script. More flexible than the JavaScript `eval()` function since it interprets a file as well as a string and allows more control over how interpreted code inherits variables from the script that calls `SElib.interpret()`. By default, all variables in a script are inherited as global variables.

There is no specific return for an error. To trap an error use the `try/catch` error trapping statements.

The `SElib.interpret()` method may not be used with scripts that have been compiled into executable files using the */bind* option of the Pro version of ScriptEase Desktop.

SEE: SElib.interpretInNewThread(), SElib.spawn()

EXAMPLE:

```
   // The following interpreted code displays "Hello
world"
SElib.interpret('Screen.writeln("Hello world")',
INTERP_TEXT);
   // The following interprets
   // the file jseedit.jse with
   // autoexec.bat as an argument to the script
SElib.interpret("jseedit.jse c:\\autoexec.bat",
                INTERP_FILE);
```

SElib.interpretInNewThread()

SYNTAX: SElib.interpretInNewThread(filename, codeToInterpret)

WHERE: filename - the name of a script file with ScriptEase code. Use null if not interpreting a file.

codeToInterpret - a string variable with one or more ScriptEase statements to interpret, if not using a file. If a file is being interpreted, the string is used as command line arguments for the script file being interpreted.

RETURN: number - the ID of the thread containing the new instance of ScriptEase. Depending on the operating system, returns 0 or -1 on an error.

DESCRIPTION: For Win32 and OS/2, that is, for operating systems that support multithreading. Not supported for operating systems that do not support multithreading, such as DOS and 16-bit Windows.

This method creates a new thread within the current ScriptEase process and interprets a script within that new thread. The new script runs independently of the currently executing thread. This method differs from SElib.interpret() in that the calling thread does not wait for the interpretation to finish and differs from SElib.spawn() in that the new thread runs in the same memory and process space as the currently running thread.

A script writer must ensure any synchronization among threads. ScriptEase data and globals are on a per-thread basis.

If the parameter filename is not null, then it is the name of a file to interpret, and the parameters, filename and codeToInterpret are parsed as if being command-line parameters to a main() function.

If the parameter filename is null, then codeToInterpret is treated as JavaScript code, a string with ScriptEase statements, and is interpreted directly.

SEE: SElib.interpret(), SElib.spawn()

EXAMPLE:
```
// See usage in threads.jse and httpd.jse
```

SElib.makeWindow()

SYNTAX: SElib.makeWindow(parent, class, windowFunction, text, style,
col, row, width, height, createParam, utilityVar)

WHERE: parent - window handle of the parent window of this window, which would mean that this window is a subwindow. Pass null if this window is being created on the desktop, without a specific window being its parent. If null, the desktop is the parent.

class - a string or an object. If this parameter is a string, it must be one of the pre-existing Windows classes:

```
button
combobox
edit
listbox
scrollbar
static
```

If this parameter is an object or structure it may have the following properties:

```
.style          Windows class style
.icon           icon bitmap for minimized window
.cursor         appearance when over this window
.background     window background color
```

Properties that are not assigned values receive default values. In general, the class defines the behavior of a window.

windowFunction - an identifier, the function that is called whenever Windows sends a message to this window. Use null if no function is to be called to intercept windows messages. In the case of null, default functions for Windows are called. If specified, the windowFunction should return a number or nothing. Use the actual identifier of the function and not a string with its name. For example, use MyWinFunction instead of "MyWinFunction". The windowFunction is described in greater detail in the description section.

text - the window title or caption that appears in the title bar. Use null or "" if the window has no title.

style - the style of the window. Windows has many predefined styles that may be joined into one style by using the bitwise or operator, "|". Windows styles are defined with "WS_" at the beginning. For example, `WS_MAXIMIZEBOX | WS_THICKFRAME` would define a window that has a thick frame and a maximize box. The "WS_" windows styles are standard definitions used in Windows programming and may be found in `winobj.jsh` or `window.jsh`.

col - the left most column of the window, expressed in pixels.

row - the top most row of the window, expressed in pixels. Together, col and row define the top left corner of the window. Use CW_USEDEFAULT for col and row to let Windows set the position.

width - the total width of the window, expressed in pixels.

height - the total height of the window, expressed in pixels. By using col, row, width, and height, a window can be place precisely on a screen.

createParam - normally set to null. If used, it may be a number or object that is passed with the Windows WM_CREATE message when creating a window.

utilityVar - any variable that a scripter chooses. This variable is passed to the windowFunction when it receives a Windows message. The windowFunction may alter the utilityVar. An object or structure may be used, in which case many values may be passed and altered as properties of the object. One practice is to use an object to keep up with the properties of a window, sometimes including its subwindows. This object is a good vehicle for passing information.

RETURN: number - the handle of the window created on success, else null.

DESCRIPTION: For Win32 and Win16

This method is the basic function for creating windows that will be opened and managed by ScriptEase. This function provides the basis for normal windows operations when windows created by it are opened. This function registers the created window with ScriptEase, so that when the .doWindows() method is executed, this window will be properly managed.

If the class of the Window is unknown, it is registered as a new class.

The windowFunction, a parameter of SElib.makeWindow(), is a function that is specified to intercept and handle all Windows messages that are posted to this window, the window just created by SElib.makeWindow(). The windowFunction will intercept all messages sent its associated window which slows execution of a script. Use SElib.messageFilter() to limit the messages that are actually intercepted by the windowFunction. If the windowFunction has a return value, it must be a number, which seems limiting. But remember, that you may use utilityVar as a variable for receiving information and for passing information.

The definition of a windowFunction must follow the following format:

```
function MyWinFunction(hWnd, Message, Param1,
                       Param2 [, utilityVar])
{
// Body of the window function
}
```

hWnd - a number, Window handle for the window which receives these Windows messages. It is the handle of the window created by SElib.makeWindow() that specified this function to receive messages.

Message - a number, a message ID. Windows defines message IDs and posts them to windows.

Param1 - a parameter that may accompany a message.

Param2 - a second parameter that may accompany a message.

utilityVar - an optional variable that is specified in the SElib.makeWindow() call that created this window. This variable is often an object/structure with several pieces of information which may be altered. If it is, the changes are available to other functions that may use the variable while SElib.doWindows() is active and is showing and managing the windows under its control.

SEE: SElib.doWindows()

EXAMPLE:

```
var InfoStruct;
InfoStruct.width = 400;
InfoStruct.height = 300;

var hWnd  = SElib.makeWindow
            (
               0, null, MyWinFunction,
                     "My Window", WS_MAXIMIZEBOX,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     InfoStruct.width, InfoStruct.height,
                     null, InfoStruct
                 );

function MyWinFunction(hWnd, Msg, Param1,
                       Param2, UtilVar)
{
   // Body of function to process messages.
   // Notice that UtilVar receives InfoStruct
}
```

SElib.messageFilter()

SYNTAX: SElib.messageFilter(hWnd[, message[, ...]])

WHERE: hWnd - a number, the handle of a window created by SElib.makeWindow() or subclassed with SElib.subclassWindow().

message - one or more messages to be processed by the window to which hWnd points.

RETURN: object - an array of messages being filtered prior to this call to SElib.messageFilter(). Returns null if no messages are in the filter, that is, all messages are passed through to ScriptEase functions or if hWnd is not a handle for a window processed by SElib.makeWindow() or SElib.subclassWindow().

DESCRIPTION: For Win32 and Win16

Restricts the messages being processed by windows created with SElib.makeWindow() or subclassed with SElib.subclassWindow(). Scripts run much faster if windows only process the messages that they act on, that is, just the messages that they need. Initially, there are no message filters so all messages are processed.

Calling this method with no parameters removes all message filtering.

SEE: SElib.makeWindow(), SElib.subclassWindow()

SElib.multiTask()

SYNTAX: SElib.multiTask(on)

WHERE: on - a boolean determining whether multitasking is on or off. Default is true.

RETURN: void.

DESCRIPTION: For Win16

Turns multitasking of programs on or off. Normally, multitasking is enabled and should be turned off only for very brief and critical sections of code. No messages are received by the current program or any other program while multitasking is off.

`SElib.multiTask()` is additive, meaning that if you call `SElib.multiTask(false)` twice, then you must call `SElib.multiTask(true)` twice before multitasking is resumed.

The example below empties the clipboard. Multitasking is turned off during this brief interval to ensure that no other program tries to open the clipboard while this program is accessing it.

SEE: SElib.suspend()

EXAMPLE:

```
SElib.multiTask(false);
SElib.dynamicLink("USER", "OPENCLIPBOARD", SWORD16,
                  PASCAL, Screen.handle());
SElib.dynamicLink("USER", "EMPTYCLIPBOARD", SWORD16,
PASCAL);
SElib.dynamicLink("USER", "CLOSECLIPBOARD", SWORD16,
PASCAL);
SElib.multiTask(true);
```

SElib.peek()

SYNTAX: SElib.peek(address[, dataType])

WHERE: address - the address in memory from which to get data, that is, a pointer to data in memory.

dataType - the type of data to get, or thought of in another way, the number of bytes of data to get. `UWORD8` is the default.

RETURN: value - returns the data specified by dataType

DESCRIPTION: Reads or gets data from the position in memory to which the parameter address points. The parameter dataType may have the following values:

```
UWORD8   SWORD8   UWORD16   SWORD16   UWORD24
SWORD24  UWORD32  SWORD32   FLOAT32   FLOAT64
FLOAT80  (FLOAT80 is not available in Win32)
```

These values specify the number of bytes to be read and returned.

Caution. Routines that work with memory directly, such as this one, should be used with caution. A programmer should clearly understand memory and the operations of these methods before using them. ScriptEase does not trap errors caused by this routine.

SEE: SElib.poke(), Blob.get(), Clib.memchr(), Clib.fread() for more information on the dataType values

EXAMPLE:

```
var v = "Now";
   // Display "Now"
Screen.writeln(v);
   // Get the "N"
var vPtr = SElib.pointer(v);
   // Get the address of the first byte of v, "N"
var p = SElib.peek(vPtr);
   // Convert "N" to "P"
SElib.poke(vPtr,p+2);
   // Display "Pow"
Screen.writeln(v);

// See usage in clipbrd.jsh, com.jsh,
// dde.jsh, ddesrv.jsh, and winsock.jsh
```

SElib.pointer()

SYNTAX: SElib.pointer(varName)

WHERE: varName - the name or identifier of a variable

RETURN: number - the address of, a pointer to, the variable identified by varName.

DESCRIPTION: Gets the address in memory of a variable. The pointer points to the first byte of data in a variable. The variable may be a primitive data type: byte, integer, or float, or it may be a single dimension array of bytes, integers, or floats, which includes a string. If the variable is an array, then the address returned points to the first byte of the first element of the array. The parameter varName may also identify a Blob variable since Blobs are actually byte arrays. Other types of data are not allowed.

For computer architectures that distinguish between near and far memory addresses, the value returned by SElib.pointer() is a far address or pointer.

ScriptEase data is guaranteed to remain fixed at its memory location only as long as that memory is not modified by a script. Thus, a pointer is valid only until a script modifies the variable identified by varName or until the variable goes out of scope in a script. Putting data in the memory occupied by varName after such a change is dangerous. When data is put into the memory occupied by varName, be careful not to put more data than will fit in the memory that the variable actually occupies.

Caution. Routines that work with memory directly, such as this one, should be used with caution. A programmer should clearly understand memory and the operations of these methods before using them. ScriptEase does not trap errors caused by this routine.

SEE: SElib.peek(), SElib.poke(), Clib.memchr(), Blob object

EXAMPLE:

```
var v = "Now";
   // Display "Now"
Screen.writeln(v);
   // Get the "N"
var vPtr = SElib.pointer(v);
   // Get the address of the first byte of v, "N"
var p = SElib.peek(vPtr);
   // Convert "N" to "P"
SElib.poke(vPtr,p+2);
   // Display "Pow"
Screen.writeln(v);

// See usage in fileobj.jsh, batch.jsh,
// memsrch.jsh, touch.jsh, and pickfile.jsh
```

SElib.poke()

SYNTAX: SElib.poke(address, data[, dataType])

WHERE: address - the address in memory from which to get data, that is, a pointer to data in memory.

data - data to write directly to memory. The data should match the dataType.

dataType - the type of data to get, or thought of in another way, the number of bytes of data to get. UWORD8 is the default.

RETURN: number - the address of the byte after the data just written to memory.

DESCRIPTION: Writes data to the position in memory to which the parameter address points. The data to be written must match the dataType. The parameter dataType may have the following values:

```
UWORD8   SWORD8   UWORD16   SWORD16   UWORD24
SWORD24  UWORD32  SWORD32   FLOAT32   FLOAT64
FLOAT80  (FLOAT80 is not available in Win32)
```

These values specify the number of bytes to be written to memory.

Caution. Routines that work with memory directly, such as this one, should be used with caution. A programmer should clearly understand memory and the operations of these methods before using them. ScriptEase does not trap errors caused by this routine.

SEE: SElib.peek(), Blob.put(), Clib.memchr(), Clib.fread for more information on the dataType values

EXAMPLE:

```
var v = "Now";
   // Display "Now"
Screen.writeln(v);
   // Get the "N"
var vPtr = SElib.pointer(v);
   // Get the address of the first byte of v, "N"
var p = SElib.peek(vPtr);
   // Convert "N" to "P"
SElib.poke(vPtr,p+2);
   // Display "Pow"
```

```
Screen.writeln(v);

// See usage in bmp.jsh, clipbrd.jsh,
// dde.jsh, ddecli.jsh, and dropsrc.jsh
```

SElib.ShellFilterCharacter()

SYNTAX: SElib.ShellFilterCharacter(functionFilterCharacter, allKeys)

WHERE: functionFilterCharacter - identifier, the name of a ScriptEase function to use to filter characters.

allKeys - boolean, specifies whether the functionFilterCharacter is called for every keystroke or just for keys that are not ordinary printable characters, such as function keys. The return of the method Clib.isprint() corresponds to the difference in keys that allKeys affects.

RETURN: void.

DESCRIPTION: Adds a character filter function to a ScriptEase shell. When ScriptEase is running as a command shell, that is, when a ScriptEase interpreter is executed with no arguments, this methods allows the installation of a function to be called when keystrokes are pressed. For example, the autoload.jse script that ships with ScriptEase uses this method to implement command line history and filename completion.

The function, functionFilterCharacter, must conform to the following:

```
function functionFilterCharacter(command,
   position, key, extended, alphaNumeric)
```

command - string, the current string on the shell command line. This string is read/write and may be changed by this function.

position - number, the current cursor position within the command string. This position may be altered by this function.

key - number, the key being pressed. This parameter may be altered by the function. Set key to zero, 0, to ignore keyboard input.

extended - boolean, true if the current keystroke is an extended keyboard character, that is, a function key, a keyboard combination, and so forth.

alphaNumeric - true if the current keystroke is an alphabetic or numeric key. The return of the method Clib.isalnum() corresponds to alphaNumeric.

return - boolean, true if the command line must be redrawn or the cursor position moved, based on the actions in this function.

SEE: SElib.ShellFilterCommand(), autoload.jse, Clib.isalnum()

SElib.ShellFilterCommand()

SYNTAX: SElib.ShellFilterCommand(functionFilterCommand)

WHERE: functionFilterCommand - identifier, the name of a function to use to filter commands to a ScriptEase shell.

RETURN: void.

DESCRIPTION: Adds a command filter function to a ScriptEase shell. When ScriptEase is running as a command shell, that is, when a ScriptEase interpreter is executed with no arguments, this method allows a function to be installed which is called when commands are entered in a shell. For example, the autoload.jse script that ships with ScriptEase uses this method to implement commands, such as CD and TYPE.

The function, functionFilterCommand, must conform to the following:

```
function functionFilterCommand(command)
```

command - a string, the current string on a shell command line. This string is read/write and may be changed by the function. A ScriptEase shell executes the command after returning from this function. To prevent ScriptEase from executing any command set command to a zero-length string, for example, command[0]='\0', but not command="".

Before passing a command line to a filter function, ScriptEase strips leading white space from the beginning and end of the command string. Also, any redirection on a command line is not seen by this function, since redirection is handled internally by ScriptEase. For example, if a command line string is "dir>dir.txt", then this function only sees the string "dir".

SEE: SElib.ShellFilterCommand(), autoload.jse

SElib.spawn()

SYNTAX: SElib.spawn(mode, execSpec[, arg[, ...]])

WHERE: mode - a number indicating how to spawn or execute the file named by execSpec. The parameter mode may be one of the following values though not all values are valid on all operating systems:

- `P_WAIT` Wait for a child program to complete before continuing. (All platforms)
- `P_NOWAIT` A script continues to run while a child program runs. In windows, a successful call with mode `P_NOWAIT` returns the window handle of the spawned process. (Windows and OS/2)
- `P_SWAP` Like P_WAIT, but swap out ScriptEase to create more room for the child process. `P_SWAP` will free up as much memory as possible by swapping ScriptEase to *`EMS/XMS/INT15`* memory or to disk (in *`TMP`* or *`TEMP`* or else current directory) before executing the child process (thanks to Ralf Brown for his excellent spawn library). (DOS only)
- `P_OVERLAY` The script exits and the child program is executed in its place. (DOS 16-bit)

execSpec - a string with the path and filename of an executable file or a ScriptEase script.

arg - one or more values to passed as parameters to the file to be executed.

RETURN: void - if the mode is `P_OVERLAY`.

number - if the mode is `P_WAIT`, the return is the exit code of the child process, else it is -1.

number - if the mode is `P_NOWAIT` or `P_SWAP`, the return is the identifier of the child process, else it is `-1`.

DESCRIPTION: Launches another application. The parameter mode determines the behavior of the script after the spawn call, while execSpec is the name of the process being spawned. Any arguments to the spawned process follow execSpec.

The parameter execSpec may be the path and filename of an executable file or the name of a ScriptEase script. If it is a script, the spawned script runs from the same instance of ScriptEase as the calling script. A spawned script does not cause another instance of the interpreter to be launched. A script that has been bound with the ScriptEase /bind function cannot be spawned from the same instance as the calling script.

The parameter execSpec is automatically passed as argument 0. ScriptEase implicitly converts all arguments to strings before passing them to the child process.

`SElib.spawn()` searches for execSpec in the current directory and then in the directories of the PATH environment variable. If there is no extension in execSpec, `SElib.spawn()` searches for file extensions in the following order: com, exe, bat, and cmd.

If a batch file is being spawned in 16-bit DOS and the environment variable `COMSPEC_ENV_SIZE` exists, the command processor is provided the amount of memory as indicated by `COMSPEC_ENV_SIZE`. If `COMSPEC_ENV_SIZE` does not exist, the command processor receives only enough memory for existing environment variables.

A return value of -1 results when `Clib.errno` is set to identify why the function failed.

SEE: SElib.interpret(), SElib.interpretInNewThread(), winexec.jsh

EXAMPLE:

```
   // The following fragment
   // calls a mortgage program,
   // mortgage.exe, which takes
   // three parameters, initial debt,
   // rate, and monthly payment, and
   // returns, in its exit code,
   // the number of months needed to pay the debt.
var months = SElib.spawn(P_WAIT,
   "MORTGAGE.EXE 300000 10.5 1000");
```

```
if (months < 0)
   Screen.writeln( "Error spawning MORTGAGE");
else
   Clib.printf(
    "It takes %d months to pay off the mortgage\n",
    months);

   // The arguments could also
   // be passed to mortgage.exe as
   // separate variables, as in the following.
var months = SElib.spawn(P_WAIT,
   "MORTGAGE.EXE",300000,10.5,1000);

   // The arguments could be passed
   // to mortgage.exe in a
   // variable array, provided that
   // they are all of the same
   // data type, in this case strings.
var MortgageData;
MortgageData[0] = "300000";
MortgageData[1] = "10.5";
MortgageData[2] = "1000";
var ths = spawn(P_WAIT,
   "MORTGAGE.EXE", MortgageData);
```

SElib.splitFilename()

SYNTAX: SElib.splitFilename(filespec)

WHERE: filespec - string specification for a file. May be a full or partial path specification.

RETURN: object - structure containing the drive and directory, file, and extension information contained in filespec. The structure returned has the following properties:

```
.dir   directory name including leading drive
       spec and trailing slash (d:\dir1\dir2\)
.name  root name of file only (filename)
.ext   file extension with leading period (.ext)
```

The three properties returned are guaranteed not to be null.

The actual characters used, such as the slash, depend on the operating system.

DESCRIPTION: Break up a file specification, full or partial path specification, into its component parts: drive and directory, filename, and extension. The filespec does not have to actually exist. This method merely divides up the filespec, as passed, according to the conventions of the operating system without checking to see if a drive, directory, or filename actually exists.

SEE: SElib.fullpath(), SElib.splitFilename(), File object in fileobj.jsh

EXAMPLE:

```
   // After splitting a filespec,
   // the following statement will
   // reconstruct it
var parts = SElib.splitFilename(MySpec);
var FileSpec = MySpec.dir + MySpec.name + MySpec.ext;
```

SElib.subclassWindow()

SYNTAX: SElib.subclassWindow(hWnd, windowFunction, utilityVar)

WHERE: hWnd - a number, the handle of an existing window to subclass.

windowFunction - an identifier, the function that is called whenever Windows sends a message to this window. The parameter windowFunction is the same as for SElib.makeWindow().

utilityVar - any variable that a scripter chooses. This variable is passed to the windowFunction when it receives a Windows message. The parameter utilityVar is the same as for SElib.makeWindow().

RETURN: boolean - true on success, else false if hWnd is invalid, was created with SElib.makeWindow(), or is already subclassed.

DESCRIPTION: For Win32 and Win16

This method hooks the specified windowFunction into the message loop for a window such that the function is called before the window's default or previously-defined function.

The parameter hWnd is the window handle of an already existing window to subclass.

The parameter windowFunction is the same as in the SElib.makeWindow() method. Note that, as in the SElib.makeWindow() method, if this method returns a value, then the default or subclassed function is not called. If this method returns no value, the call is passed on to the previous function. This method may be used to subclass any Window that is not already being managed by a windowFunction for this ScriptEase instance. If a window was created with SElib.makeWindow() or is already subclassed then this method fails.

Note that this method may be used, only once, with the window handle returned by Screen.handle(). If you want to subclass the main ScriptEase window, it is best to open another instance of ScriptEase and subclass it rather than to subclass the instance that is powering your script. Although it is possible to subclass that window, if you try to do anything with it, you will likely get caught in an infinite loop and hang. To undo the window subclassing or remove a WindowFunction from the message loop, use SElib.breakWindow().

A WindowFunction may modify UtilityVar.

In your function that handles messages for another process, certain limits are set as to what you can do with system resources. For example, an open file handle is invalid while processing a message for another program, because Windows maps file handles into a table for programs. To work around this problem, you may send a message to one of your ScriptEase windows to handle the processing. This action switches Windows' tables to your program while handling that SendMessage.

SEE:	SElib.makeWindow(), Window object in winobj.jsh

SElib.suspend()

SYNTAX:	SElib.suspend(milliSeconds)
WHERE:	milliSeconds - a number, the time in thousandths of a second to suspend program execution.
RETURN:	void.
DESCRIPTION:	Suspends script or program execution for the time interval

specified in milliSeconds. The next statement in a script will execute at the end of the delay.

True accuracy to the exact millisecond is not guaranteed and is only closely approximated according to the accuracy provided by the underlying operating system. This method allows a computer to devote more time to other processes and can be used to give the processor time to complete other tasks before calling the next line in a script.

The example below spawns a copy of Windows Notepad, puts the date and time into the document by simulating the selection of Time/Date from the Edit menu, and then displays the line "You asked for the time?". The SElib.suspend() method gives the processor time to finish completing the menu command before entering the text into Notepad. If Keystroke() were called immediately after the call to MenuCommand(), the text would be sent to Notepad while the menu item was still being selected and would be garbled.

SEE: SElib.spawn(), Clib.ctime(), Date object

EXAMPLE:

```
#include <menuctrl.jsh>
#include <keypush.jsh>
var hWnd = SElib.spawn(P_NOWAIT, "notepad.exe");
MenuCommand(hWnd, "Edit|Time");
SElib.suspend(300);
KeyStroke("\nYou asked for the time?");
```

SElib.windowList()

SYNTAX: SElib.windowList(hWnd)

WHERE: hWnd - a number, the handle of the window for which to find its child windows.

RETURN: object - an array of window handles for all the child windows of hWnd.

DESCRIPTION: For Win32 and Win16

Get the handles of all child windows of the window designated by hWnd. I hWnd is not passed, then get the handles of the windows on the desktop which amount to all the parent windows.

SEE: SElib.makeWindow(), Window object in winobj.jsh

Dynamic links

For Win32, Win16, and OS/2

The dynamic link method, which varies in usage among the three platforms that support it, allows flexibility when making calls to dynamic link libraries, DLLs, and allows access to operating-system functions, API calls, not explicitly provided by ScriptEase. If you know the proper conventions for a call, then you can make an `SElib.dynamicLink()` call in a ScriptEase function to be used for making a system call. Such a function is referred to as a wrapper, a function in which a system call becomes available as a function call.

There are three versions of `SElib.dynamicLink()`: Win32, Win16, and OS/2. These three versions differ slightly in the way they are called. So, if you wish to use one function in a script that will be run on different platforms, you must create an operating system filter using preprocessor directives: `#if`, `#ifdef`, `#elif`, `#else`, and `#endif`.

Since these versions are different in the way that they call `SElib.dynamicLink()`, they will be treated separately.

SElib.dynamicLink() - for Win32

SYNTAX: SElib.dynamicLink(library, procedure, convention)

WHERE: library - a string, the name of the dynamic link library, DLL, being used, the one having the procedure being called.

procedure - a string or number, the name or ordinal number of a routine in a dynamic link library to be used.

convention - the calling convention to use when invoking or using the procedure being called.

```
CDECL    Push right parameter first.
         Caller pops parameters.
STDCALL  Push right parameter first.
         Caller pops parameters.
PASCAL   Push left parameter first.
         Callee pops parameters.
```

RETURN: value - the value returned by the procedure being called, else void if the procedure does not return a value.

DESCRIPTION: For Win32

Calls a routine in a dynamic link library, DLL. The most common use is to use various functions in the Windows API.

All values are passed as 32-bit values. If a parameter is undefined when 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 will set the value.

If a 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 DLL function, the structure is copied to a binary buffer as described in Blob.put() and Clib.fwrite(). When calling the DLL function, a descriptor argument must precede the structured parameter, and this descriptor argument is in addition to the parameter list for the procedure being called. After calling the DLL function, the binary data will be 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.

SEE: Clib.fread(), Blob object

EXAMPLE:

```
   // The following calls
   // the Windows MessageBeep() function:
#define  MESSAGE_BEEP_ORDINAL 104
SElib.dynamicLink("USER.EXE", MESSAGE_BEEP_ORDINAL,
   SWORD16, PASCAL,0);

   // The following displays a simple message box
   // and waits for user to press <Enter>.
#define MESSAGE_BOX_ORDINAL 1
#define MB_OK  0x0000
// Message box contains one push button: OK.
#define MB_TASKMODAL 0x2000
// Must respond to this message
SElib.dynamicLink("USER.EXE", MESSAGE_BOX_ORDINAL,
   SWORD16, PASCAL, null,
   "This is a simple message box",
   "Title of box", MB_OK | MB_TASKMODAL);
   // The following accomplishes
   // the same thing as above.
#define MB_OK 0x0000
// Message box contains one push button: OK.
#define MB_TASKMODAL  0x2000
// Must respond to message
SElib.dynamicLink("USER", "MESSAGEBOX", SWORD16,
   PASCAL, null,
   "This is a simple message box",
   "Title of box", MB_OK | MB_TASKMODAL);
```

SElib.dynamicLink() - for Win16

SYNTAX: SElib.dynamicLink(library, procedure, returnType, convention)

WHERE: library - a string, the name of the dynamic link library, DLL, being used, the one having the procedure being called.

procedure - a string or number, the name or ordinal number of a routine in a dynamic link library to be used.

returnType - a number which tells ScriptEase what type of value the procedure returns, so that it can be properly converted into an integer. The be one of the following:

```
UWORD8   SWORD8   UWORD16   SWORD16   UWORD24
SWORD24  UWORD32  SWORD32   FLOAT32   FLOAT64
FLOAT80  (FLOAT80 is not available in Win32)
```

convention - the calling convention to use when invoking or using the procedure being called.

```
CDECL    Push right parameter first.
         Caller pops parameters.
STDCALL  Push right parameter first.
         Caller pops parameters.
PASCAL   Push left parameter first.
         Callee pops parameters.
```

RETURN: value - the value returned by the procedure being called, else void if the procedure does not return a value.

DESCRIPTION: For Win16

Calls a routine in a dynamic link library, DLL. The most common use is to use various functions in the Windows API.

If a parameter is a Blob, a byte-array, or an undefined value, it is passed as a far pointer. All other numeric values are passed as 16-bit values. If 32-bits are needed, the parameter must be passed in parts, with the low word first and the high word second for CDECL calls but the high word first and low word second for PASCAL calls.

If a parameter is undefined when SElib.dynamicLink() is called, then it is assumed that the parameter is a far pointer to be filled in, that is, that the far address of a data element is passed to the function and that function will set 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 DLL function, the structure will be copied to a binary buffer as described in Blob.put() and Clib.fwrite(). After calling the DLL function, the binary data is 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.

SEE: Blob object, Clib.fread()

SElib.dynamicLink() - for OS/2

SYNTAX: SElib.dynamicLink(library, procedure, bitSize, convention, ...)

WHERE: library - a string, the name of the dynamic link library, DLL, being used, the one having the procedure being called.

procedure - a string or number, the name or ordinal number of a routine in a dynamic link library to be used.

bitSize - indicates whether this call is 16-bit or 32-bit and may be either of two defined values: BIT16 or BIT32.

convention - the calling convention to use when invoking or using the procedure being called.

```
CDECL    Push right parameter first.
         Caller pops parameters.
STDCALL  Push right parameter first.
         Caller pops parameters.
PASCAL   Push left parameter first.
         Callee pops parameters.
```

RETURN: value - the value returned by the procedure being called, else void if the procedure does not return a value.

DESCRIPTION: For OS/2

Calls a routine in a dynamic link library, DLL.

Any parameters required by a dynamically linked function should be passed at the end of the parameters listed above, as indicated by the ellipsis at the end of the parameter list. These variables are interpreted as follows, depending on the operating system.

For 32-bit functions, all values are passed as 32-bit values. For 16-bit functions, if the parameter is a Blob, a byte-array, or an undefined value, then it is passed as a 16:16 segment:offset pointer, otherwise all numeric values are passed as 16-bit values, so if 32-bits are needed they must be passed in parts, with the low word first and the high word second.

If a parameter is undefined when SElib.dynamicLink() is called, then it is assumed that parameter is a 32-bit value to be filled in, that is, that the address of a 32-bit data element is passed to the function and that function will set 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 DLL function, the structure is copied to a binary buffer as described in Blob.put() and Clib.fwrite(). After calling the DLL function, the binary data is 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.

An alternative syntax:

The OS/2 processor also allows you to call a function via a call gate with the following syntax:

```
SElib.dynamicLink(callGate, bitSize, convention,
    ...)
```

Where callGate is the gate selector for a routine referenced through a call gate.

SEE: Blob object

Buffer Object

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

Buffer object instance properties

Buffer bigEndian

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

Buffer cursor

SYNTAX:	buffer.cursor
DESCRIPTION:	The current position within a buffer. This value is always between 0 and .size. It can be assigned to as well. If a user attempts to move the cursor beyond the end of a buffer, than the buffer is extended to accommodate the new position, and filled with null bytes. If a user attempts to set the cursor to less than 0, then it is set to the beginning of the buffer, to position 0.
SEE:	Buffer.bigEndian
EXAMPLE:	`var p = buffer.cursor;`

Buffer data

SYNTAX: buffer.data

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

SEE: Buffer.size

Buffer size

SYNTAX: buffer.size

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

SEE: Buffer.cursor

EXAMPLE:
```
var n = buffer.size;
```

Buffer unicode

SYNTAX: buffer.unicode

DESCRIPTION: This property is a boolean flag specifying whether to use unicode strings when calling getString() and putString(). This value is set when the buffer is created, but may be changed at any time. This property defaults to the unicode status of the underlying ScriptEase engine.

SEE: Buffer.bigEndian

EXAMPLE:
```
buffer.bigEndian = false;
```

Buffer[] Array

SYNTAX: Buffer[offset]

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

SEE: Buffer.getValue(), Buffer.putValue()

EXAMPLE:

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

Buffer object instance methods

Buffer()

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

WHERE: size - size of buffer to be created.

string - string of characters from which to create a buffer.

buffer - buffer of characters from which to create another buffer.

bufferObject - buffer to be duplicated.

unicode - boolean flag for the initial state of the unicode property of the buffer

bigEndian - numeric description of the initial state of the bigEndian property of the buffer.

RETURN: object - the new buffer created.

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

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

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

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

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

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

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

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

A line of code following this syntax creates a new buffer object from another buffer object. Everything is duplicated exactly from the other bufferObject, including the cursor location, size, and data.

All of the above calls have an equivalent call form (such as "Buffer(15)"), except that this simply returns the buffer part (equivalent to the data member), rather than the entire Buffer object.

SEE:

EXAMPLE:

Buffer getString()

SYNTAX: buffer.getString([length])

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

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

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

SEE: Buffer.putString()

EXAMPLE:

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

Buffer getValue()

SYNTAX: Buffer.getValue([ValueSize[, ValueType]])

WHERE: ValueSize - a positive number describing the number of bytes to be used and defaults to 1. The following are acceptable values: 1,2,3,4,8, and 10 ValueType - One of the following types: "`signed`", "`unsigned`", or "`float`". The default type is: "`signed`."

ValueType - either `signed`, `unsigned` or `float.`

RETURN: value - from the specified position in a buffer object.

DESCRIPTION: This call is similar to the `putValue()` function, except that it gets a value instead of puts a value.

SEE: Buffer.putValue(), Buffer[]

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

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

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

Buffer putString()

SYNTAX: buffer.putString(String)

WHERE: string - Any string.

RETURN: void.

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

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

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

Buffer putValue()

SYNTAX: buffer.putValue(Value[, ValueSize[, ValueType]])

WHERE: Value - value to be put into the buffer.

ValueSize - a positive number describing the number of bytes to be used and defaults to 1. The following are acceptable values: 1,2,3,4,8, and 10 ValueType - One of the following types: "`signed`", "`unsigned`", or "`float`". The default type is: "`signed`."

ValueType - either `signed`, `unsigned` or `float.`

RETURN: The value is put into buffer at the current cursor position, and the cursor value is automatically incremented by the size of the value to reflect this addition.

DESCRIPTION: This method puts the specified value into a buffer. The value must be a number. `ValueSize` or both `valueSize` and `valueType` may be passed as additional parameters. ValueSize is a positive number describing the number of bytes to be used and defaults to 1. Acceptable values for `valueSize` are 1,2,3,4,8, and 10, providing that it does not conflict with the optional `valueType` flag. (See listing below.)

The parameter valueType must be one of the following: "`signed`", "`unsigned`", or "`float`". It defaults to "`signed`." The `valueType` parameter describes the type of data to be read. Combined with valueSize, any type of data can be put. The following list describes the acceptable combinations of valueSize and valueType:

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

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

SEE: Buffer.getValue(), Buffer[]

EXAMPLE:

```
/*
```

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

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

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

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

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

Buffer subBuffer()

SYNTAX: buffer.subBuffer(Beginning, End)

WHERE: Beginning - start of offset

End - end of offset (up to but not including this point)

RETURN: object - another Buffer object consisting of the data between the positions specified by the parameters: beginning and end.

DESCRIPTION: If the parameter beginning is less than 0, then it is treated as 0, the start of the buffer. If the parameter end is beyond the end of the buffer, then the new sub-buffer is extended with null bytes, but the original buffer is not altered.

SEE: String.subString()

EXAMPLE:

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

Buffer toString()

SYNTAX: buffer.toString()

RETURN: string - a string equivalent of the current state of the buffer, with all characters, including `"\0"`.

DESCRIPTION: Any conversion to or from unicode is done according to the `.unicode` flag of the object.

SEE: Buffer.getString()

EXAMPLE:

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

Screen Object

The methods in this section are preceded with the Object name Screen, since individual instances of the Screen Object are not created. For example, Screen.clear() is the syntax to use to clear a ScriptEase text screen.

The methods documented in this section are the internal methods of the Screen object. The script library file *screen.jsh* adds methods and properties to the Screen object. See the documentation for *screen.jsh* for more information about useful Screen object methods.

Screen object static methods

Screen.clear()

SYNTAX:	Screen.clear()
RETURN:	void.
DESCRIPTION:	This method clears the portion of a computer screen that is controlled by a script. On some platforms, such as DOS, the entire screen may be cleared, but on others, such as Win32, only a window will be cleared.
SEE:	Screen.setBackground(), Screen.setForeground()
EXAMPLE:	Screen.clear();

Screen.cursor()

SYNTAX:	Screen.cursor([col[, row])
WHERE:	col - the column or x coordinate of a character on a text screen or window. The unit of measurement is a character position.
	row - the row or y coordinate of a character on a text screen or window. The unit of measurement is a character position.
RETURN:	object - a structure with two numeric properties, col and row, which represent the current cursor position on a text screen or

window. The properties of the structure are:

```
 .col
 .row
```

DESCRIPTION: Gets and sets the cursor position in a text screen or window. If no parameters are passed, the only action is to return the current cursor position. If the parameters, col and row, are passed, the cursor is set to the position specified. If the parameter row is omitted, the cursor is moved to the column specified by col on the current row.

When parameters are passed, the cursor position returned is the position after the cursor has been placed at the new coordinates.

Text screen coordinates begin with 0, that is, the first column or row is 0. The first position on a text screen/window is at col == 0 and row == 0.

SEE: Screen.size()

EXAMPLE:

```
   // Get the cursor position as:
   // CurPos.col and CurPos.row
var CurPos = Screen.cursor()
   // Place the cursor at column 3, the 4th column,
   // of the current row
var CurPos = Screen.cursor(3)
   // Place the cursor at column 3, the 4th column,
and
   // at row 4, the 5th row, of the current text
screen/window.
var CurPos = Screen.cursor(3, 4)
   // Place the cursor at the first position
Screen.cursor(0, 0);
```

Screen.handle()

SYNTAX: Screen.handle()

RETURN: number - the window handle of the current ScriptEase screen or window.

DESCRIPTION: This method returns the window handle of a ScriptEase screen, such as the text screen produced by normal text output from *Sewin32.exe*. This handle may be used with other windows routines that manipulate windows. A ScriptEase screen is a window like other windows in the Windows API.

SEE: Window object in *winobj.jsh*, *winutils.jsh*

EXAMPLE:
```
var ScreenHandle = Screen.handle()
```

Screen.setBackground()

SYNTAX: Screen.setBackground(color | r, g, b)

WHERE: color - a number that represents a color. A single number may be used. The file *colors.jsh* has defines, such as color_cyan, for many popular colors. If more than one parameter is passed, then the method assumes that a three number system based on red, green, and blue elements is being used.

r - a number that represents the red element of a color.

g - a number that represents the green element of a color.

b - a number that represents the blue element of a color.

RETURN: void.

DESCRIPTION: Sets the background color of a ScriptEase screen or window, such as the window that is created when ScriptEase is running as a shell. The method may receive either one or three arguments. If there is only one parameter color, it must be one of the colors defined in colors.jsh. If there are three parameters, they define a color based on a combination of r(ed), g(reen), and b(lue).

The background color is the color of screen or window on which characters are displayed. The foreground color is the color of the characters. The colors set are for the entire screen/window, not just the current text being written.

SEE: Screen.setForeground()

EXAMPLE:
```
   // Needed for color_cyan below
include "colors.jsh"
   // Set screen background to cyan
Screen.setBackground(color_cyan)
   // Set to white
Screen.setBackground(255, 255, 255)
   // Set to white
Screen.setBackground(0xFF, 0xFF, 0xFF)
   // Set to black
```

```
Screen.setBackground(0, 0, 0);
```

Screen.setForeground()

SYNTAX: Screen.setForeground(color | r, g, b)

WHERE: color - a number that represents a color. A single number may be used. The file colors.jsh has defines, such as color_cyan, for many popular colors. If more than one parameter is passed, then the method assumes that a three number system based on red, green, and blue elements is being used.

r - a number that represents the red element of a color.

g - a number that represents the green element of a color.

b - a number that represents the blue element of a color.

RETURN: void.

DESCRIPTION: Sets the foreground color of a ScriptEase screen or window, such as the window that is created when ScriptEase is running as a shell. The method may receive either one or three arguments. If there is only one parameter color, it must be one of the colors defined in colors.jsh. If there are three parameters, they define a color based on a combination of r(ed), g(reen), and b(lue).

The foreground color is the color of the characters. The background color is the color of screen or window on which characters are displayed. The colors set are for the entire screen/window, not just the current text being written.

SEE: Screen.setBackground()

EXAMPLE:

```
   // Needed for color_cyan below
include "colors.jsh"
   // Set screen foreground to cyan
Screen.setForeground(color_cyan)
   // Set to white
Screen.setForeground(255, 255, 255)
   // Set to white
Screen.setForeground(0xFF, 0xFF, 0xFF)
   // Set to black
Screen.setForeground(0, 0, 0);
```

Screen.size()

SYNTAX: Screen.size([col[, row])

WHERE: col - a number representing the number of columns, the x coordinate, of the current text screen or window. The unit of measurement is a character position.

row - a number representing the number of rows, the y coordinate, of the current text screen or window. The unit of measurement is a character position.

RETURN: object - a structure with two numeric properties, col and row, which represent the current width and height of a text screen or window. The properties of the structure are:

```
 .col
 .row
```

DESCRIPTION: Gets and sets the size, width and height, of the current text screen or window. If no parameters are passed, the only action is to return a structure with the width and height. If two arguments are passed, the screen/window is set to the width and height specified by the two parameters. If only one argument is passed, an error occurs.

SEE: Screen.cursor()

EXAMPLE:

```
   // Get the current screen/window size
var CurSiz = Screen.size()
   // Set text screen/window size to 40 columns
   // and 12 rows
Screen.size(40, 12)
```

Screen.write()

SYNTAX: Screen.write(data)

WHERE: data - any data type in JavaScript.

RETURN: void.

DESCRIPTION: The method Screen.write is the most basic way of displaying data in text form. It will display any JavaScript data type as a string by using the automatic data conversion abilities of JavaScript. For example, look at the following fragment:

```
var s = "123";
var n = 123;
var a = new Array(1, "2", 3);
Screen.writeln(s);
Screen.writeln(n);
Screen.writeln(a);
```

The display is:

```
123
123
1,2,3
```

Automatic conversion allows the variables n and s to display the same and converts the array a to a suitable string in which the elements of the array are separated by commas.

Screen.write does not put end of line characters on a string, that is, the cursor is positioned at the end of the string displayed. Use Screen.writeln to display a string and position the cursor on the next line. For example, the following two lines of code produce the same display:

```
Screen.write("456");
Clib.printf("456");
```

The following two lines of code produce the same display and illustrate the difference between Screen.write and Screen.writeln:

```
Screen.write("This is a line.\n");
Screen.writeln("This is a line.");
```

The Screen.write statement uses the escape character "\" to put end of line (EOL) characters at the end of the string. The Screen.writeln statement automatically puts EOL characters at the end of a string.

As an example of displaying data, consider the following fragment:

```
var FirstName = "John ";
var LastName = "Doe ";
var CityStateZip = new Array("Medford", "MA",
02155);

Screen.write(FirstName + LastName + "in ");
Screen.write(CityStateZip);
```

produces the following display:

```
John Doe in Medford,MA,1133
```

This code fragment illustrates how easy it is to work with, concatenate, combine, and display different data types.

SEE: Screen.writeln(), write.jsh which has many extensions to the Screen.write method.

EXAMPLE: `Screen.write("Using Screen.write is simple.");`

Screen.writeln()

SYNTAX: Screen.writeln(data)

WHERE: data - any data type in JavaScript.

RETURN: void.

DESCRIPTION: Screen.writeln is the same as Screen.write except that Screen.writeln automatically puts end of line (EOL) characters at the end of data that it displays. See Screen.write for a full description, including the differences between Screen.write and Screen.writeln.

SEE: Screen.writeln(), write.jsh which has many extensions to the Screen.write method.

EXAMPLE: `Screen.writeln("Using Screen.write is simple.");`

String Object

The String object is a data type, a hybrid that shares characteristics of primitive data types and of composite data types. The String is presented in this section under two main headings in which the first describes its characteristics as a primitive data type and the second describes its characteristics as an object.

String as data type

A string is an ordered series of characters. The most common use for strings is to represent text. To indicate that text is a string, it is enclosed in quotation marks. For example, the first statement below puts the string "hello" into the variable word. The second sets the variable word to have the same value as a previous variable hello:

```
var word = "hello";
word = hello;
```

Escape sequences for characters

Some characters, such as a quotation mark, have special meaning to the interpreter and must be indicated with special character combinations when used in strings. This allows the interpreter to distinguish between a quotation mark that is part of a string and a quotation mark that indicates the end of the string. The table below lists the characters indicated by escape sequences:

`\a`	Audible bell
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash character

`\0###`	Octal number	(example: '\033' is the escape character)
`\x##`	Hex number	(example: '\x1B' is the escape character)
`\0`	Null character	(example: '\0' is the null character)
`\u####`	Unicode number	(example: '\u001B' is the escape character)

Note that these escape sequences cannot be used within strings enclosed by back quotes, which are explained below.

Single quote

You can declare a string with single quotes instead of double quotes. There is no difference between the two in JavaScript, except that double quote strings are used less commonly by many scripters. In functions declared with the cfunction keyword, the difference is more important. For more information, see the section on cfunctions.

Back quote

ScriptEase provides the back quote "`", also known as the back-tick or grave accent, as an alternative quote character to indicate that escape sequences are not to be translated. Any special characters represented with a backslash followed by a letter, such as "\n", cannot be used in back tick strings.

For example, the following lines show different ways to describe a single file name:

```
"c:\\autoexec.bat" // traditional C method
'c:\\autoexec.bat' // traditional C method
`c:\autoexec.bat`  // alternative ScriptEase method
```

Back quote strings are not supported in most versions of JavaScript. So if you are planning to port your script to some other JavaScript interpreter, you should not use them.

Long Strings

You can use the + operator to concatenate strings. The following line:

```
var proverb = "A rolling stone " + "gathers no moss."
```

creates the variable proverb and assigns it the string "A rolling stone gathers no moss." If you try to concatenate a string with a number, the number is converted to a string.

```
var newstring = 4 + "get it";
```

This bit of code creates newstring as a string variable and assigns it the string "4get it".

The use of the + operator is the standard way of creating long strings in JavaScript. In ScriptEase, the + operator is optional. For example, the following:

```
var badJoke =
   "I was standing in front of an Italian "
   "restaurant waiting to get in when this guy "
   "came up and asked me, \"Why did the "
   "Italians lose the war?\" I told him I had "
   "no idea. \"Because they ordered ziti"
   "instead of shells,\" he replied."
```

creates a long string containing the entire bad joke.

String as object

Strictly speaking, the String object is not truly an object. It is a hybrid of a primitive data type and of an object. As an example of its hybrid nature, when strings are assigned using the assignment operator, the equal sign, the assignment is by value, that is, a copy of a string is actually transferred to a variable. Further, when strings are passed as arguments to the parameters of functions, they are passed by value. Objects, on the other hand, are assigned to variables and passed to parameters by reference, that is, a variable or parameter points to or references the original object.

Strings have both properties and methods which are listed in this section. These properties and methods are discussed as if strings were pure objects. Strings have instance properties and methods and are shown with a period, ".", at their beginnings. A specific instance of a variable should be put in front of a period to use a property or call a method. The exception to this usage is a static method which actually uses the identifier String, instead of a variable created as an instance of String. The following code fragment shows how to access the .length property, as an example for calling a String property or method:.

```
var TestStr = "123";
var TestLen = TestStr.length;
```

String object instance properties

String length

SYNTAX:	string.length
DESCRIPTION:	The length of a string, that is, the number of characters in a string. JavaScript strings may contain the `"\0"` character.
SEE:	String.lastIndexOf()

EXAMPLE:	`var s = "a string";`
	`var n = s.length;`

String object instance methods

String()

SYNTAX:	new String([value])
WHERE:	value - value to be converted to a string as this string object.
RETURN:	This method returns a new string object whose value is the supplied value.
DESCRIPTION:	If `value` is not supplied, then the empty string "" is used instead. Otherwise, the value ToString(`value`) is used. Note that if this function is called directly, without the new operator, then the same construction is done, but the returned variable is converted to a string, rather than being returned as an object.
EXAMPLE:	`var s = new String(123);`

String charAt()

SYNTAX:	string.charAt(Position)
WHERE:	Position - offset within a string.
RETURN:	string - character at position `Position`

DESCRIPTION: This method acter at the specified position. If no character exists at location `nPosition`, or if `nPosition` is less than 0, then `NaN` is returned.

SEE: String.charCodeAt()

EXAMPLE:

```
// To get the first character in a string,
// use as follows:

var string = "a string";
string.charAt(0);

// To get the last character in a string, use:
string.charAt(string.length - 1);
```

String charCodeAt(index)

SYNTAX: string.charCodeAt(index)

WHERE: index - index of the character whose encoding is to be returned.

RETURN: number - representing the unicode value of the character at position index of a string. Returns `NaN` if there is no character at the position.

DESCRIPTION: This method gets the nth character code from a string.

SEE: String.charAt(), String.fromCharCode()

String concat()

SYNTAX: string.concat([string1, ...])

WHERE: stringN - A list of strings to append to the end of the current object.

RETURN: This method returns a string value (not a string object) consisting of the current object and any subsequent arguments append to it.

DESCRIPTION: This method creates a new string whose contents are equal to the current object. Each argument is then converted to a string using ToString() and appended to the newly created string. This value is then returned. Note that the original object remains unaltered. The '+' operator performs the same function.

SEE: Array.concat()

EXAMPLE:

```
// The following line:

var proverb = "A rolling stone " + "gathers no moss."

// creates the variable proverb and
// assigns it the string
// "A rolling stone gathers no moss."
// If you try to concatenate a string with a number,
// the number is converted to a string.

 var newstring = 4 + "get it";

// This bit of code creates newstring as a string
// variable and assigns it the string
// "4get it".

// The use of the + operator is the standard way of
// creating long strings in JavaScript.
// In ScriptEase, the + operator is optional.
// For example, the following:

var badJoke = "I was in front of an Italian "
    "restaurant waiting to get in when this guy "
    "came up and asked me, \"Why did the "
    "Italians lose the war?\" I told him I had "
    "no idea. \"Because they ordered ziti"
    "instead of shells,\" he replied."

// creates a long string containing
// the entire bad joke.
```

String indexOf()

SYNTAX: string.indexOf(substring[, offset])

WHERE: substring - substring to search for within string.

offset - optional integer argument which specifies the position within string at which the search is to start.

RETURN: number - index of the first appearance of a substring in a string. If position is undefined or not supplied, 0 is returned.

DESCRIPTION: String.indexOf() searches the string for the string specified in `substring`. The search begins at `offset` if `offset` is specified, otherwise the search begins at the beginning of the string. If `substring` is found, String.indexOf() returns the position of its first occurance. Character positions within string are numbered in increments of one beginning with zero.

SEE: String.charAt(), String.lastIndexOf(), String.substring()

EXAMPLE:

```
var string = "what a string";
string.indexOf("a")

// returns the position, which is 2 in this example,
// of the first "a" appearing in the string.
// The method indexOf()may take an optional second
// parameter which is an integer indicating the index
// into a string where the method starts searching
// the string. For example:

var magicWord = "abracadabra";
var secondA = magicWord.indexOf("a", 1);

// returns 3, index of the first "a" to be found in
// the string when starting from the second letter of
// the string.
// Since the index of the first character is 0, the
// index of second character is 1.
```

String lastIndexOf()

SYNTAX: string.lastIndexOf(substring[, offset])

WHERE: substring - The substring that is to be searched for within string

offset - An optional integer argument which specifies the position within string at which the search is to start.

RETURN: number - position of the last occurence of the `substring` specified

DESCRIPTION: This method is similar to `String.indexOf()`, except that it finds the last occurrence of a character in a string instead of the first.

SEE: String.indexOf()

String localeCompare()

SYNTAX:	string.localeCompare()
WHERE:	
RETURN:	
DESCRIPTION:	
SEE:	
EXAMPLE:	

String slice()

SYNTAX:	string.slice(start[, end])
WHERE:	start - index to start from. end - index at which to end.
RETURN:	string - a substring (not a String object) consisting of the characters.
DESCRIPTION:	This method is very similar to String.substring(), in that it returns a substring from one index to another. The only difference is that if either `start` or `end` is negative, then it is treated as `length+start` or `length+end`. If either exceeds the bounds of the string, then either 0 or the length of the string is used instead.
SEE:	String.substring()

String split()

SYNTAX:	string.split([substring])
WHERE:	substring - character, string or regular expression where the string is split. If `substring` is not specified, an array will be returned with the name of the `string` specified. Essentially this will mean that the string is split character by character.
RETURN:	object - if no delimiters are specified, returns an array with one element which is the original string.
DESCRIPTION:	This method splits a string into an array of strings based on the

delimiters in the parameter substring. The parameter substring is optional and if supplied, determines where the string is split.

SEE: Array.join()

EXAMPLE:

```
// For example, to create an array of al
// use code similar to the following fra
var sentence = "I am not a crook";
var wordArray = sentence.split(' ');
```

String substring()

SYNTAX: string.substring(Start, End)

WHERE: Start - integer specifing the position within the string to begin the desired substring.

End - integer specifing the position within the string to end the desired substring. This integer must be one greater than the desired end position to allow for the terminating `null` byte.

RETURN: string - substring of the result of converting this object to a string, starting from character position start and running to the end of the string. The result is a string value, not a string object.

DESCRIPTION: This method retrieves a section of a string. The Start parameter is the index or position of the first character to include. The End parameter marks the end of the string. The End position is the index or position after the last character to be included. The length of the substring retrieved is defined by End minus Start. Another way to think about the Start and End positions is that End equals Start plus the length of the substring desired.

SEE: String.charAt(), String.indexOf(), String.lastIndexOf(), String.slice()

EXAMPLE:

```
// For example, to get the first nine characters
// in string, use a Start position
// of 0 and add 9 to it, that is,
// "0 + 9", to get the End position
// which is 9. The following fragment illustrates.

var str = "1234567890 - 10 digits begin this string";
var substr = str.substring(0,9);

//The variable substr will equal "123456789".
```

```
// The characters from 0 to 8, a total of 9,
// are included, but the tenth character,
// "0", at position 9 is not included.
```

String toLocaleLowerCase()

SYNTAX:	string.toLocaleLowerCase()
RETURN:	string - a copy of a string with each character converted to lower case.
DESCRIPTION:	This method behaves exactly the same as String.prototype.toLowerCase(). It is designed to convert the string to lower case in a locale sensitive manner, though this functionality is currently unavailable. Once it is implemented, this function may behave differently is some locales (such as Turkish), though for the majority it will be identical to toLowerCase()
SEE:	String.toLowerCase()

String toLocaleUpperCase()

SYNTAX:	string.toLocaleUpperCase()
RETURN:	string - a copy of a string with each character converted to lower case.
DESCRIPTION:	This method behaves exactly the same as String.prototype.toUpperCase(). It is designed to convert the string to lower case in a locale sensitive manner, though this functionality is currently unavailable. Once it is implemented, this function may behave differently is some locales (such as Turkish), though for the majority it will be identical to toUpperCase()
SEE:	String.toUpperCase()

String toLowerCase()

SYNTAX:	string.toLowerCase()

RETURN: string - copy of a string with all of the letters changed to lower case.

DESCRIPTION: This method changes the case of a string.

SEE: String.toUpperCase

EXAMPLE:

```
var string = new String("Hello, World!");
string.toLowerCase()

// This will return the string "hello, world!".
```

String toUpperCase())

SYNTAX: string.toLowerCase()

RETURN: string - a copy of a string with all of the letters changed to upper case.

DESCRIPTION: This method changes the case of a string.

SEE: String.toLowerCase()

EXAMPLE:

```
var string = new String("Hello, World!");
string.toUpperCase()

// This will return the string
//  "HELLO, WORLD!".
```

String valueOf()

SYNTAX: string.valueOf()

RETURN: string - the value of a string.

DESCRIPTION: The value returned is equivalent to String.toString and is not generally called in code but rather internally by JavaScript.

SEE: String.toString(), Object.valueOf()

EXAMPLE:

```
foo = new String("This is a string");
Screen.writeln(foo.valueOf())

// The result, "This is a string",
// will be printed.
```

String object static methods

String.fromCharCode()

SYNTAX:	string.fromCharCode(CharCode[, ...])
WHERE:	CharCode - character code, or list of codes, to be converted.
RETURN:	string - string created from the character codes that are passed to it as parameters.
DESCRIPTION:	The identifier String is used with this static method, instead of a variable name as with instance methods. The arguments passed to this method are assumed to be unicode characters.
SEE:	String()

EXAMPLE:	`// The following code:`
	`var string = String.fromCharCode(0x0041,0x0042)`
	`// will set the variable string to be "AB".`

RegExp Object

The RegExp object allows the use of regular expression parsing in searches. The syntax follows the ECMAScript standard.

RegExp object instance methods

RegExp()

SYNTAX:	new RegExp() new RegExp(pattern) new RegExp(pattern, flags)
WHERE:	pattern - a string containing the regular expression search pattern. flags - A string containing the options for this regular expression object .
RETURN:	object - a new regular expression object, or null on error.
DESCRIPTION:	Creates a new regular expression object using the search pattern and options if they are specified. The flag string must contain any of the following characters, or it must be the empty string: i - sets the ignoreCase property to true g - sets the global property to true

EXAMPLE:

```
// no options
var regobj = new RegExp( "r*t", "" );
// ignore case
var regobj = new RegExp( "r*t", "i" );
// global search
var regobj = new RegExp( "r*t", "g" );
// set both to be true
var regobj = new RegExp( "r*t", "ig" );
```

RegExp exec()

SYNTAX:	regexp.exec(string)
WHERE:	string - the string on which to perform a regular expression match.

RETURN:	object - an array object containing the results of the match, or null if there was no match.
DESCRIPTION:	This method performs a regular expression search on the specified string using the regular expression pattern for this object.

EXAMPLE:	`var regobj = new RegExp( "r*t", "" );`
	`var result = regobj.exec( "rat" );`

RegExp test()

SYNTAX:	regexp.test(string)
WHERE:	string - the string on which to perform a regular expression match.
RETURN:	boolean - true if there is a match, false otherwise.
DESCRIPTION:	This function is equivalent to `regexp.exec(string)!=null`

RegExp compile()

SYNTAX:	regexp.compile(pattern[, flags])
WHERE:	pattern - A string containing the regular expression search pattern. flags - A string containing the options for this regular expression object.
RETURN:	void.
DESCRIPTION:	This function sets the regular expression for this object to the specified pattern. If the flag string is supplied, it must contain any of the following characters, or it must be the empty string: i - sets the ignoreCase property to true g - sets the global property to true

EXAMPLE:	`var regobj = new RegExp();`
	`regobj.compile( "r*t" );`

Object Object

`platform: All OS, All version of SE`

Object object instance methods

Object hasOwnProperty()

SYNTAX:	object.hasOwnProperty(propertyName)
WHERE:	property - name of the property about which to query.
RETURN:	boolean - indicating whether or not the current object has a property of the specified name.
DESCRIPTION:	This method simply determines if the object has a property with the name propertyName. This is almost the same as testing `defined(object[propertyName])`, except that undefined values are different from non-existent values, and the internal `_hasProperty()` method of the object may be called.

Object isPrototypeOf()

SYNTAX:	object.isPrototypeOf(variable)
WHERE:	variable - the object to test.
RETURN:	boolean - true if variable is an object and the current object is present in the prototype chain of the object, otherwise it returns false.
DESCRIPTION:	If variable is not an object, then this method immediately returns false. Otherwise, the method recursively searches the internal _prototype property of the object and if at any point the current object is equal to one of these prototype properties, then the method returns true.

Object propertyIsEnumerable()

SYNTAX:	object.propertyIsEnumerable(propertyName)
WHERE:	property - name of the property about which to query.
RETURN:	boolean - true if the current object has an enumerable property of the specified name, otherwise false.
DESCRIPTION:	If the current object has no property of the specified name, then false is immediately returned. If the property has the `DontEnum` attribute set, then false is returned. Otherwise, true is returned.

Object toLocaleString()

SYNTAX:	object.toLocaleString()
RETURN:	string - a string representation of this object.
DESCRIPTION:	This method is intended to provide a default .toLocaleString method for all objects. It behaves exactly if .toString() had been called on the original object.
SEE:	Object.toString()

Object toString()

SYNTAX:	object.toString()
RETURN:	string - a string representation of this object.
DESCRIPTION:	When this method is called, the internal class property (_class) is retrieved from the current object. A string is then constructed whose contents are "[object classname]", where `classname` is the value of the property from the current object. Note that this function is rarely called directly, rather it is called implicitly through such functions as ToString().
SEE:	Object.toLocaleString

Math Object

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

The methods in this section are preceded with the Object name Math, since individual instances of the Math Object are not created. For example, Math.abs() is the syntax to use to get the absolute value of a number.

Math object static properties

Math.E

SYNTAX:	Math.E
DESCRIPTION:	The number value for e, the base of natural logarithms. This value is represented internally as approximately 2.7182818284590452354.
EXAMPLE:	`var n = Math.E;`

Math.LN10

SYNTAX:	Math.LN10
DESCRIPTION:	The number value for the natural logarithm of 10. This value is represented internally as approximately 2.302585092994046.
EXAMPLE:	`var n = Math.LN10;`

Math.LN2

SYNTAX:	Math.LN2
DESCRIPTION:	The number value for the natural logarithm of 2. This value is represented internally as approximately 0.6931471805599453.
EXAMPLE:	`var n = Math.LN2;`

Math.LOG2E

SYNTAX:	Math.LOG2E
DESCRIPTION:	The number value for the base 2 logarithm of e, the base of the natural logarithms. This value is represented internally as approximately 1.4426950408889634. The value of Math.LOG2E is approximately the reciprocal of the value of Math.LN2.
EXAMPLE:	`var n = Math.LOG2E;`

Math.LOG10E

SYNTAX:	Math.LOG10E
DESCRIPTION:	The number value for the base 10 logarithm of e, the base of the natural logarithms. This value is represented internally as approximately 0.4342944819032518. The value of Math.LOG10E is approximately the reciprocal of the value of Math.LN10
EXAMPLE:	`var n = Math.LOG10E`

Math.PI

SYNTAX:	Math.PI
DESCRIPTION:	The number value for pi, the ratio of the circumference of a circle to its diameter. This value is represented internally as approximately 3.14159265358979323846.
EXAMPLE:	`var n = Math.PI;`

Math.SQRT1_2

SYNTAX:	Math.SQRT1_2
DESCRIPTION:	The number value for the square root of 2, which is represented internally as approximately 0.7071067811865476. The value of Math.SQRT1_2 is approximately the reciprocal of the value of Math.SQRT2.

EXAMPLE: `var n = Math.SQRT1_2;`

Math.SQRT2

SYNTAX: Math.SQRT2

DESCRIPTION: The number value for the square root of 2, which is represented internally as approximately 1.4142135623730951.

EXAMPLE: `var n = Math.SQRT2;`

Math object static methods

Math.abs()

SYNTAX: Math.abs(X)

WHERE: X - a number.

RETURN: number - the absolute value of `x`. Returns `NaN` if `x` cannot be converted to a number.

DESCRIPTION: Computes the absolute value of a number.

EXAMPLE:

```
//The function returns the absolute value
// of the number -2 (i.e.
//the return value is 2):
var n = Math.abs(-2);
```

Math.acos()

SYNTAX: Math.acos(X)

WHERE: X - a number between 1 and -1.

RETURN: number - the arc cosine of `x`.

DESCRIPTION: The return value is expressed in radians and ranges from 0 to `pi`. Returns `NaN` if `x` cannot be converted to a number, is greater than 1, or is less than -1.

EXAMPLE:

```
function compute_acos(x)
{
   return Math.acos(x)
}
```

```
// If you pass -1 to the function compute_acos(),
// the return is the
// value of pi (approximately 3.1415...),
// if you pass 3 the
// return is NaN since 3 is out
// of the range of Math.acos.
```

Math.asin()

SYNTAX: Math.asin(X)

WHERE: X - a number between 1.0 and -1.0

RETURN: number - implementation-dependent approximation of the arc sine of the argument.

DESCRIPTION: The return value is expressed in radians and ranges from `-pi/2` to `+pi/2`. Returns `NaN` if `x` cannot be converted to a number, is greater than 1, or less than -1.

EXAMPLE:

```
function compute_asin(x)
{
   return Math.asin(x)
}
//If you pass -1 to the function compute_acos(),
//the return is the
//value of -pi/2 , if you pass 3 the return is
//NaN since 3 is out of Math.acos's range.
```

Math.atan()

SYNTAX: Math.atan(X)

WHERE: X - any number.

RETURN: number - an implementation-dependent approximation of the arctangent of the argument.

DESCRIPTION: The return value is expressed in radians and ranges from `-pi/2` to `+pi/2`.

EXAMPLE:

```
//The arctangent of x is returned
//in the following function:
function compute_arctangent(x)
{
   return Math.arctangent(x)
}
```

Math.atan2()

SYNTAX: Math.atan2(X, Y)

WHERE: X - x coordinate of the point.

Y - y coordinate of the point.

RETURN: number - an implementation-dependent approximation to the arc tangent of the quotient, `y/x`, of the arguments `y` and `x`, where the signs of the arguments are used to determine the quadrant of the result.

DESCRIPTION: It is intentional and traditional for the two-argument arc tangent function that the argument named `y` be first and the argument named `x` be second. The return value is expressed in radians and ranges from `-pi` to `+pi`.

EXAMPLE:

```
//The arctangent of the quotient y/x
//is returned in the
//following function:
function compute_arctangent_of_quotient(x, y)
{
   return Math.arctangent2(x, y)
}
```

Math.ceil()

SYNTAX: Math.ceil(X)

WHERE: X - any number or numeric expression.

RETURN: number - the smallest number that is not less than the argument and is equal to a mathematical integer.

DESCRIPTION: If the argument is already an integer, the result is the argument itself. Returns `NaN` if `x` cannot be converted to a number.

EXAMPLE:

```
//The smallest number that is
//not less than the argument and is
//equal to a mathematical integer is returned
//in the following function:
function compute_small_arg_eq_to_int(x)
{
   return Math.ceil(x)
}
```

Math.cos()

SYNTAX:	Math.cos(X)
WHERE:	X - an angle, measured in radians.
RETURN:	number - an implementation-dependent approximation of the cosine of the argument
DESCRIPTION:	The argument is expressed in radians. Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by 2pi/360.

EXAMPLE:

```
//The cosine of x is returned
//in the following function:
function compute_cos(x)
{
   return Math.cos(x)
}
```

Math.exp(X)

SYNTAX:	Math.exp(X)
WHERE:	X - either a number or a numeric expression to be used as an exponent
RETURN:	number - an implementation-dependent approximation of the exponential function of the argument.
DESCRIPTION:	For example returns e raised to the power of the `x`, where e is the base of the natural logarithms. Returns `NaN` if `x` cannot be converted to a number.

EXAMPLE:

```
//The exponent of x is returned
//in the following function:
function compute_exp(x)
{
   return Math.exp(x)
}
```

Math.floor()

SYNTAX:	Math.floor(X)
WHERE:	X - a number.
RETURN:	number - the greatest number value that is not greater than the argument and is equal to a mathematical integer.
DESCRIPTION:	If the argument is already an integer, the return value is the argument itself.
EXAMPLE:	see below

EXAMPLE:

```
//The floor of x is returned
//in the following function:
function compute_floor(x)
{
   return Math.floor(x)
}
//If 6.78 is passed to compute_floor,
//7 will be returned. If 89.1
//is passed, 90 will be returned.
```

Math.log()

SYNTAX: Math.log(X)

WHERE: X - a number.greater than zero.

RETURN: number - an implementation-dependent approximation of the natural logarithm of `x`.

DESCRIPTION: If a negative number is passed to Math.log, the return is `NaN`

EXAMPLE:

```
//The natural log of x is returned
//in the following function:
function compute_log(x)
{
   return Math.log(x)
}
//If the argument is less than 0 or NaN,
//the result is NaN
//If the argument is +0 or -0,
//the result is -infinity
//If the argument is 1, the result is +0
//If the argument is +infinity,
//the result is +infinity
```

Math.max()

SYNTAX: Math.max(X, Y)

WHERE: X - a number.

Y - a number.

RETURN: number - the larger of `x` and `y`.

DESCRIPTION: Returns `NaN` if either argument cannot be converted to a number.

EXAMPLE:

```
//The larger of x and y is returned
//in the following function:
function compute_max(x, y)
{
   return Math.max(x, y)
}
//If x = a and y = 4 the return is NaN
//If x > y the return is x
//If y > x the return is y
```

Math.min()

SYNTAX: Math.min(X, Y)

WHERE: X - a number.

Y - a number.

RETURN: number - the smaller of `x` and `y`. Returns `NaN` if either argument cannot be converted to a number.

DESCRIPTION: Returns `NaN` if either argument cannot be converted to a number.

EXAMPLE:

```
//The smaller of x and y is returned
//in the following function:
function compute_min(x, y)
{
   return Math.min(x, y)
}
//If x = a and y = 4 the return is NaN
//If x > y the return is y
//If y > x the return is x
```

Math.pow()

SYNTAX: Math.pow(X, Y)

WHERE: X - The number which will be raised to the power of `Y`

Y - The number which `X` will be raised to

RETURN: number - the value of `x` to the power of `y`.

DESCRIPTION: If the result of `Math.pow` is an imaginary or complex number, `NaN` will be returned. Please note that if `Math.pow` unexpectedly returns infinity, it may be because the floating-point value has experienced overflow.

EXAMPLE:

```
//x to the power of y is returned
//in the following function:
function compute_x_to_power_of_y(x, y)
{
   return Math.pow(x, y)
}
//If the result of Math.pow is
//an imaginary or complex number,
//the return is NaN
//If y is NaN, the result is NaN
//If y is +0 or -0, the result is 1,
//even if x is NaN
//If x = 2 and y = 3 the return value is 8
```

Math.random()

SYNTAX: Math.random()

RETURN: number - a number which is positive and pseudo-random and which is greater than or equal to 0 but less than 1.

DESCRIPTION: Calling this method numerous times will result in an established pattern (the sequence of numbers will be the same each time. This method takes no arguments. Seeding is not yet possible.

SEE: Clib.rand()

EXAMPLE:

```
//Return a random number:
function compute_rand_numb()
{
   return Math.rand()
}
```

Math.round()

SYNTAX: Math.round(X)

WHERE: X - a number.

RETURN: number - value that is closest to the argument and is equal to a mathematical integer. `X` is rounded up if its fractional part is equal to or greater than 0.5 and is rounded down if less than 0.5.

DESCRIPTION: The value of Math.round(x) is the same as the value of Math.floor(x+0.5), except when x is *0 or is less than 0 but greater than or equal to -0.5; for these cases Math.round(x) returns *0, but Math.floor(x+0.5) returns +0.

SEE: Math.floor()

EXAMPLE:

```
//Return a mathematical integer:
function compute_int(x)
{
   return Math.round(x)
}
//If the argument is NaN, the result is NaN
//If the argument is already an integer
//such as any of the
//following values: -0, +0, 4, 9, 8;
//then the result is the
//argument itself.
//If the argument is .2, then the result is 0.
//If the argument is 3.5, then the result is 4
//Note: Math.round(3.5) returns 4,
//but Math.round(-3.5) returns -3.
```

Math.sin()

SYNTAX: Math.sin(X)

WHERE: X - an angle in radians.

RETURN: number - the sine of `x`, expressed in radians.

DESCRIPTION: Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by 2pi/360.

EXAMPLE:

```
//Return the sine of x:
function compute_sin(x)
{
   return Math.sin(x)
}
//If the argument is NaN, the result is NaN
```

```
//If the argument is +0, the result is +0
//If the argument is -0, the result is -0
//If the argument is +infinity or -infinity,
//the result is NaN
```

Math.sqrt()

SYNTAX: Math.sqrt(X)

WHERE: X - a number or numeric expression greater than or equal to zero.

RETURN: number - the square root of `x`.

DESCRIPTION: Returns `NaN` if `x` is a negative number or cannot be converted to a number.

SEE: Math.exp()

EXAMPLE:

```
//Return the square root of x:
function compute_square_root(x)
{
   return Math.sqrt(x)
}
//If the argument is NaN, the result is NaN
//If the argument is less than 0,
//the result is NaN
//If the argument is +0, the result is +0
//If the argument is -0, the result is -0
//If the argument is +infinity,
//the result is +infinity
```

Math.tan()

SYNTAX: Math.tan(X)

WHERE: X - an angle measured in radians.

RETURN: number - the tangent of `x`, expressed in radians.

DESCRIPTION: Returns `NaN` if `x` cannot be converted to a number. In order to convert degrees to radians you must multiply by 2pi/360.

EXAMPLE:

```
//Return the tangent of x:
function compute_tan(x)
{
   return Math.tan(x)
}
//If the argument is NaN, the result is NaN
```

```
//If the argument is +0, the result is +0
//If the argument is -0, the result is -0
//If the argument is +infinity or -infinity,
//the result is NaN
```

Global object

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

```
if (defined(name))
   Screen.writeln("name is defined");

if (global.defined(name))
   Screen.writeln("name is defined");
```

The following two lines of code are also equivalent.

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

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

Most of the `global` methods, functions, described in this section are defined in the ECMAScript standards. A few are unique additions to ScriptEase. In other words, they are not part of the ECMAScript standard, but they are useful. Avoid using the unique functions in a script if it will be used with a JavaScript interpreter that does not support these few unique functions.

Conversion or casting

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

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

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

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

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

global object properties

global._argc

SYNTAX: _argc

DESCRIPTION: This property refers to the number of parameters passed to the `main()` function of a script. The name of the script is always the first parameter, so if `_argc == 1`, then the script received no arguments. See the `main()` function for more information on `argc` and the `main()` function. General programming practice uses `argc`, a parameter to the `main()`function rather than `_argc`.

SEE: function main(), _argv

EXAMPLE:

```
function main(argc, argv)
{
   // At this point, unless deliberately changed
   // by special programming, _argc == argc
}
```

global._argv

SYNTAX: _argv

DESCRIPTION: This property is an array of strings. Each string is a parameter passed to the `main()` function. The value of `argv[0]` is always the name of the script being called. The first parameter passed to the script is in `argv[1]`. See the `main()` function for more information on `argc`, `argv`, and the `main()` function. General programming practice uses `argv`, a parameter to the `main()`function rather than `_argv`.

SEE: function main(), _argc

global object methods/functions

global.defined()

SYNTAX: defined(value)

WHERE: value - a value or variable to check to see if it is defined.

RETURN: boolean - true if the value has been defined, else false

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

SEE: global.undefine()

EXAMPLE:

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

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

global.escape()

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

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

global.eval()

SYNTAX:	eval(expression)
WHERE:	expression - a valid expression to be parsed and treated as if it were code or script.
RETURN:	value - the result of the evaluation of expression as code.
DESCRIPTION:	Evaluates whatever is represented by the parameter expression. If expression is not a string, it will be returned. For example, calling eval(5) returns the value 5. If expression is a string, the interpreter tries to interpret the string as if it were JavaScript code. If successful, the method returns the last variable with which was working, for example, the return variable. If the method is not successful, it returns the special value, undefined.
SEE:	SElib.interpret()

EXAMPLE:	`var a = "who";`
	`   // Displays the string as is`

```
Screen.writeln('a == "who"');
   // Evaluates the contents of the string as code,
   // and displays "true",
   // the result of the evaluation
Screen.writeln(eval('a == "who"'));
```

global.isFinite()

SYNTAX: isFinite(number)

WHERE: number - to check if it is a finite number.

RETURN: boolean - if the parameter is or can be converted to a number, else false.

DESCRIPTION: This method returns true if the parameter, number, is or can be converted to a number. If the parameter evaluates as NaN, Number.POSITIVE_INFINITY, or Number.NEGATIVE_INFINITY, the method returns false.

SEE: global.isNaN()

EXAMPLE:
```
if (isFinite(99)) Screen.writeln("A number");
```

global.isNaN()

SYNTAX: isNaN(number)

WHERE: number - a value to if it is not a number.

RETURN: boolean - true if number is not a number, else false.

DESCRIPTION: This method returns true if the parameter, number, evaluates to NaN, "Not a Number". Otherwise it returns false.

SEE: global.isFinite()

EXAMPLE:
```
if (isNan(99)) Screen.writeln("Not a number");
```

global.getArrayLength()

SYNTAX: getArrayLength(array[, minIndex])

WHERE: array - an automatic array.

minIndex - the minimum index to use.

RETURN: number - the length of an array.

DESCRIPTION: This function should be used with dynamically created arrays, that is, with arrays that were **not** created using the `new Array()` operator and constructor. When working with arrays created using the `new Array()` operator and constructor, use the `length` property of the Array object. The `length` property is not available for dynamically created arrays which must use the functions, `getArrayLength()` and `setArrayLength()`, when working with array lengths.

The `getArrayLength()` function returns the length of a dynamic array, which is one more than the highest index of an array, if the first element of the array is at index 0, which is most common. If the parameter minIndex is passed, then it is used to set to the minimum index, which will be zero or less. You can use this function to get the length of an array that was not created with the `Array()` constructor function.

This function and its counterpart, `setArrayLength()`, are intended for use with dynamically created arrays, that is, arrays not created with the `Array()` constructor function. Use the `length` property to get the length of arrays created with the constructor function and not `getArrayLength()`.

SEE: setArrayLength(), Array.length

EXAMPLE:
```
var arr = {4,5,6,7};
Screen.writeln(getArrayLength(arr));
```

global.getAttributes()

SYNTAX: getAttributes(variable)

WHERE: variable - a variable identifier, name.

RETURN: number - representing the attributes set for a variable. If no attributes are set, the return is 0. See `setAttributes()` for a list of predefined constants for the attributes that a variable may have.

DESCRIPTION: Gets and returns the variable attributes for the parameter variable. Variable attributes may be set using the function setAttributes(). See `setAttributes()` for more information and descriptions of the attributes of variables that can be set.

SEE: setAttributes()

global.parseFloat()

SYNTAX: parseFloat(string)

WHERE: string - to be converted to a decimal float.

RETURN: number - the float to which the string converts, else NaN.

DESCRIPTION: This method is similar to `parseInt()` except that it reads decimal numbers with fractional parts. In other words, the first period, ".", in the parameter string is considered to be a decimal point, and any following digits are the fractional part of the number. The method `parseFloat()` does not take a second parameter.

SEE: global.parseInt()

EXAMPLE:
```
var i = parseInt("9.3");
```

global.parseInt()

SYNTAX: parseInt(string[, radix])

WHERE: string - to be converted to an integer.

radix - the number base to use, default is 10.

RETURN: number - the integer to which string converts, else NaN.

DESCRIPTION: This method converts an alphanumeric string to an integer number. The first parameter, string, is the string to be converted, and the second parameter, radix, is an optional number indicating which base to use for the number. If the radix parameter is not supplied, the method defaults to base 10 which is decimal. If the first digit of string is a zero, radix defaults to base 8 which is octal. If the first digit is zero followed by an "x", that is, "0x", radix defaults to base 16 which is hexadecimal.

White space characters at the beginning of the string are ignored.

The first non-white space character must be either a digit or a minus sign (-). All numeric characters following the string will be read, up to the first non-numeric character, and the result will be converted into a number, expressed in the base specified by the radix variable. All characters including and following the first non-numeric character are ignored. If the string is unable to be converted to a number, the special value NaN is returned.

SEE: global.parseFloat()

EXAMPLE:

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

global.setArrayLength()

SYNTAX: setArrayLength(array[, minIndex[, length]])

WHERE: array - an automatic array.

minIndex - the minimum index to use. Default is 0.

length - the length of the array to set.

RETURN: void.

DESCRIPTION: This function sets the first index and length of a array. Any elements outside the bounds set by MinIndex and length are lost, that is, become undefined. If only two arguments are passed to setArrayLength(), the second argument is length and the minimum index of the newly sized array is 0. If three arguments are passed to setArrayLength(), the second argument, which must be 0 or less, is the minimum index of the newly sized array, and the third argument is the length.

SEE: getArrayLength(), Array.length

EXAMPLE:

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

global.setAttributes()

SYNTAX: setAttributes(variable, attributes)

WHERE: variable - a variable identifier, name.

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

RETURN: void.

DESCRIPTION: This function sets the variable attributes for the parameter variable using the parameter attributes. Variables in ScriptEase may have various attributes set that affect the behavior of variables. This function has no return.

The following list describes the attributes that may be set for variables. Multiple attributes may be set for variables by combining them with the or operator. For example, the flag setting `READ_ONLY | DONT_ENUM` sets both of these attributes for one variable.

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

this variable fails.

SEE: getAttributes()

EXAMPLE:

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

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

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

global.undefine

SYNTAX: undefine(value)

WHERE: value - value, variable, or property to be undefined.

RETURN: void.

DESCRIPTION: This function undefines a variable, Object property, or value. If a value was previously defined so that its use with the function defined() returns true, then after using undefine() with the value, defined() returns false. Undefining a value is different than setting a value to null.

SEE: defined()

EXAMPLE:

```
// In the following fragment, the variable n
// is defined with the number value of 2 and
// then undefined.
var n = 2;
undefine(n);

// In the following fragment an object o
// is created and a property o.one is defined.
// The property is then undefined but
// the object o remains defined.
```

```
var o = new Object;
o.one = 1;
undefine(o.one);
```

global.ToBoolean()

SYNTAX: ToBoolean(value)

WHERE: value - to be cast as a boolean.

RETURN: boolean - conversion of value.

DESCRIPTION: The following list indicates how different data types are converted by this function.

- Boolean
  same as value
- Buffer
  same as for String
- null
  false
- Number
  false, if value is 0, +0, -0 or NaN, else true
- Object
  true
- String
  false if empty string, "", else true
- undefined
  false

global.ToBuffer()

SYNTAX: ToBuffer(value)

WHERE: value - to be cast as a buffer.

RETURN: buffer - conversion of value.

DESCRIPTION: This function converts value to a buffer in a manner similar to ToString() except that the resulting array of characters is a sequence of ASCII bytes and not a unicode string.

SEE: ToBytes()

global.ToBytes()

SYNTAX:	ToBytes(value)
WHERE:	value - to be cast as a buffer.
RETURN:	buffer - conversion of value.
DESCRIPTION:	This function converts value to a buffer and differs from ToBuffer() in that the conversion is actually a raw transfer of data to a buffer. The raw transfer does not convert unicode values to corresponding ASCII values. For example, the unicode string `"Hit"` is stored in a buffer as `"\0H\0\i\0t"`, that is, as the hexadecimal sequence: 00 48 00 69 00 74.
SEE:	ToBuffer()

global.ToInt32()

SYNTAX:	ToInt32(value)
WHERE:	value - to be cast as a signed 32 bit integer.
RETURN:	number - conversion of value.
DESCRIPTION:	This function is the same as ToInteger() except that if the return is an integer, it is in the range of - 2^{31} through 2^{31} - 1.
SEE:	ToInteger(), ToNumber()

global.ToInteger()

SYNTAX:	ToInteger(value)
WHERE:	value - to be cast as an integer.
RETURN:	number - conversion of value.
DESCRIPTION:	This function converts value to an integer type. First, call ToNumber(). If result is NaN, return +0. If result is +0, -0, +Infinity or -Infinity, return result. Else return floor(abs(result)) with the appropriate sign. For example, the value -4.8 is

converted to -4.

SEE: ToInt32(), ToNumber()

global.ToNumber()

SYNTAX: ToNumber(value)

WHERE: value - to be cast as a number.

RETURN: number - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by this function.

- Boolean
  +0, if value is false, else 1
- Buffer
  same as for String
- null
  +0
- Number
  same as value
- Object
  first, call ToPrimitive(), then call ToNumber() and return result
- String
  number, if successful, else NaN
- undefined
  NaN

SEE: ToInteger(), ToInt32()

global.ToObject()

SYNTAX: ToObject(value)

WHERE: value - to be cast as an object.

RETURN: object - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by this function.

- Boolean
  new Boolean object with value
- null
  generate runtime error
- Number
  new Number object with value
- Object
  same as parameter
- String
  new String object with value
- undefined
  generate runtime error

SEE: ToPrimitive()

---

global.ToPrimitive

SYNTAX: ToPrimitive(value)

WHERE: value - to be cast as a primitive.

RETURN: value - conversion of value to one of the primitive data types.

DESCRIPTION: This function does conversions only for parameters of type Object. An internal default value of the Object is returned.

SEE: ToObject()

---

global.ToString()

SYNTAX: ToString(value)

WHERE: value - to be cast ass a string.

RETURN: string - conversion of value.

DESCRIPTION: The following table lists how different data types are converted by is this function.

- Boolean

"false", if value is false, else "true"

- null
  "null"
- Number
  if value is NaN, return "NaN". If +0 or -0, return "0". If Infinity, return "Infinity". If a number, return a string representing the number. If a number is negative, return "-" concatenated with the string representation of the number.
- Object
  first, call ToPrimitive(), then call ToString() and return result
- String
  same as value
- undefined
  "undefined"

SEE: ToPrimitive(), ToNumber()

global.unescape()

SYNTAX: unescape(string)

WHERE: string - holding escape characters.

RETURN: string - with escape characters replaced by appropriate characters.

DESCRIPTION: This method is the reverse of the `escape()` method and removes escape sequences from a string and replaces them with the relevant characters. That is, an encoded string is decoded.

SEE: global.escape()

EXAMPLE:
```
unescape("Hello%20there%21");
// Returns "Hello there!"
```

global.Uint16()

SYNTAX: ToUint16(value)

WHERE: value - to be cast as a 16 bit unsigned integer.

RETURN: number - conversion of value.

DESCRIPTION:	This function is the same as ToInteger() except that if the return is an integer, it is in the range of 0 through 2^{16} - 1.
SEE:	ToUint32(), ToInteger()

global.Uint32()

SYNTAX:	ToUint32(value)
WHERE:	value - to be cast as a 32 bit unsigned integer.
RETURN:	number - conversion of value.
DESCRIPTION:	This function is the same as ToInteger() except that if the return is an integer, it is in the range of 232 - 1.
SEE:	ToInt32(), ToInteger()

Function Object

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

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

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

Function object instance methods

Function()

SYNTAX: new Function(params[, ...], body)

WHERE: params - one or a list of parameters for the function.

body - the body of the function as a string.

RETURN: object - a new function object with the specified parameters and body that can later be executed just like any other function.

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

EXAMPLE:

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

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

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

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

Function apply()

SYNTAX: function.apply([thisObj[, arguments])

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

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

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

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

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

SEE: Function(), Function.prototype.call()

EXAMPLE:

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

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

Function call()

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

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

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

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

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

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

SEE: Function(), Function.apply()

EXAMPLE:
```
// The following code:

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

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

Function toString()

SYNTAX: function.toString()

RETURN: string - a representation of the function.

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

EXAMPLE:
```
var myFunction =  new Function("a", "b",
    "this.value = a + b");
Screen.writeln( myFunction );

// This fragment will print the following
// to the screen:

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

Dos Object

`platform: DOS, Win16`

The methods in this section are specific to the DOS or WIN16 versions of ScriptEase. Most of these routines allow a programmer to have more power than is generally acknowledged as safe under the scripting guidelines of general ScriptEase. Be cautious when you use these commands. They allow much latitude in what may be done at a very low programming level with little or no built-in protections.

The methods in this section are preceded with the Object name Dos, since individual instances of the Dos Object are not created. In other words, the Dos object has only static methods. For example, `Dos.inport(portid)` is the syntax to use to read a byte from a hardware port. Remember to prepend "Dos." to the method names as shown in this section.

Dos object static methods

Dos.address()

SYNTAX:	Dos.address(segment, offset)
WHERE:	segment - segment portion of memory address.
	offset - offset portion of memory address.
RETURN:	number - memory address, a segment:offset address suitable for use in calls such as peek() and poke().
DESCRIPTION:	Convert segment:offset pointer into memory address.
SEE:	Dos.offset(), Dos.segment()

Dos.asm()

SYNTAX:	Dos.asm(buf[, ax[, bx[, cx[, dx[, si[, di[, ds[, es]]]]]]]])
WHERE:	buf - a byte buffer.
	ax, bx, cx, dx, si, di, ds, es - registers.

RETURN: number - long value for whatever is in DX:AX when buf returns.

DESCRIPTION: Make a far call to the routine that you have coded into buf. ax, bx, cx, dx, si, di, ds, and es are optional; if some or all are supplied, then the ax, bx, cx, etc... will be set to these values when the code at buf is called. The code in buf will be executed with a far call to that address, and is responsible for returning via retf or other means. The ScriptEase calling code will restore ALL registers except ss, sp, ax, bx, cx, and dx. If es or ds are supplied, then they must be valid values or 0, if 0 then the current value will be used.

EXAMPLE:

```
// The following example uses 80x86 assembly code
// to rotate memory bits:

   // return value of byte b rotate count byte
function RotateByteRight(b, count)
{
   assert( 0 <= b && b <= 0xFF );
   assert( 0 <= count && count <= 8 )
   return asm(`\xD2\xC8\xCB',b,0,count,0);

   // assembly code for would look as follows:
   // ror al, cl D2C8
   // retf CB
}
```

Dos.inport()

SYNTAX: Dos.inport(portid)

WHERE: portid - port from which to read.

RETURN: number - byte of data from a hardware port.

DESCRIPTION: Read byte from a hardware port: portid.

SEE: Dos.inportw()

Dos.inportw()

SYNTAX: Dos.inportw(portid)

WHERE: portid - port from which to read.

RETURN: number - 16 bit word of data from a hardware port.

DESCRIPTION: Read a word (16 bit) from hardware port: portid. Value read is unsigned (not negative).

SEE: Dos.inport()

Dos.interrupt()

SYNTAX: Dos.interrupt(interrupt, regIn[, regOut]

WHERE: interrupt - DOS interrupt number.

regIn -

regOut -

RETURN: boolean - since many interrupts set the carry flag for error, this function returns false if the carry flag is set, else true.

DESCRIPTION: Executes an 8086 interrupt. Set registers, call 8086 interrupt function, and then get the return values of the registers. The parameters regIn and regOut are structures containing the elements corresponding to the registers on an 8086. On input, those structure members that are defined will be set, and those that are not defined will be set to zero, with the exception of the segment registers (es & ds) which retain their current values if not explicitly specified. The possible defined input values are ax, ah, al, bx, bh, bl, cx, ch, cl, dx, dh, dl, bp, si, di, ds, and es. All Fields of the output reg structure are the same, with the addition of the *FLAGS* member, and all are set before returning. If regOut is not supplied, then the return registers and *FLAGS* register will be set for regIn on return from the interrupt call.

The parameter regOut is set to the register values upon return from Interrupt. If regOut is not supplied then regIn is set to contain the register values upon return from Interrupt.

EXAMPLE:
```
// The following example calls the DOS interrupt
// service 0x2C to read the clock:

   // display DOS time as accurately as it is read
PrintDOStime()
{
   reg.ah = 0x2C;
   interrupt(0x21,reg);
   printf("%2d:%02d:%02d",reg.ch,reg.cl,reg.dh);
}
```

Dos.offset()

SYNTAX:	Dos.offset(buf)
	Dos.offset(address)
WHERE:	buf - a byte buffer.
	address - address in memory.
RETURN:	number - offset of buffer such that 8086 would recognize the address segment::buffer as pointing to the first byte of buf.
DESCRIPTION:	`Dos.segment()` and `Dos.offset()` return the segment and offset of the data at index 0 of buf, which must be a byte array. The buffer must be big enough for whatever purpose it is used, and no changes may be made to the size of buf after these values are determined since changing the size of buf might change its absolute address. If the address versions are used, then address is assumed to be a far pointer to data, and segment will be the high word while address will be the low word. See `Dos.address()` for converting segment and offset into a single address.
SEE:	Dos.offset(), Dos.address()

Dos.outport()

SYNTAX:	Dos.outport(portid, value)
WHERE:	portid - port to which to send value.
	value - a byte of data to send to the port identified by portid.
RETURN:	void.
DESCRIPTION:	Write a byte value to hardware port: portid.

Dos.outportw()

SYNTAX:	Dos.outportw(portid, value)
WHERE:	portid - port to which to send value.
	value - a 16 bit word of data to send to the port identified by portid.
RETURN:	void.
DESCRIPTION:	Write a 16 bit word value to hardware port: portid.

Dos.segment()

SYNTAX:	Dos.segment(buf)
	Dos.segment(address)
WHERE:	buf - a byte buffer.
	address - address in memory.
RETURN:	number - segment of buffer such that 8086 would recognize the address segment::buffer as pointing to the first byte of buf.
DESCRIPTION:	`Dos.segment()` and `Dos.offset()` return the segment and offset of the data at index 0 of buf, which must be a byte array. The buffer must be big enough for whatever purpose it is used, and no changes may be made to the size of buf after these values are determined since changing the size of buf might change its absolute address. If the address versions are used, then address is assumed to be a far pointer to data, and segment will be the high word while address will be the low word. See `Dos.address()` for converting segment and offset into a single address.
SEE:	Dos.offset(), Dos.address()

Clib Object

```
platform: All operating systems; all versions of SE
```

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

Some of the functions in the Clib Object overlap the methods in JavaScript. In most cases, the newer JavaScript methods should be preferred over the older C functions. However, there are times, such as when working with many cfunctions or 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.

Clib functions with equivalent methods in JavaScript are noted as such. Since ScriptEase, JavaScript and the ECMAScript standard are developing and growing, generally, a programmer should favor the JavaScript methods over equivalent methods in the Clib object.

The methods in this section are preceded with the Object name Clib, since individual instances of the Clib Object are not created. For example, Clib.exit() is the syntax to use to exit a script.

Console I/O functions

Console I/0 functions are not available for ScriptEase WebServer Edition

Clib.printf()

SYNTAX:	Clib.printf(formatString[, variables ...])
WHERE:	formatString - string that specifies the final format.
	variables - values to be converted to and formatted as a string.
RETURN:	number - characters written, or a negative number if there is an error.
DESCRIPTION:	This method writes output to the standard output device according to the format string and returns a number equal to the number of characters written, or a negative number if there is an error. The format string can contain character combinations indicating how following parameters are to be treated. Characters

are printed as read to standard output until a percent character, %, is reached. % indicates that a value is to be printed from the parameters following the format string. Each subsequent parameter specification takes from the next parameter in the list following format. A parameter specification has the following form in which square brackets indicate optional fields and angled brackets indicate required fields:

%[flags][width][.precision]<type>

flags may be:

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

width may be:

- `n`
  (n is a number e.g., 14) At least n characters are output, padded with blanks
- `0n`

At least n characters are output, padded on the left with zeros

- `*`
  The next value in the argument list is an integer specifying the output width
- `.precision`
  If precision is specified, then it must begin with a period (.), and may be as follows:

  - `0`
    For floating point type, no decimal point is output
  - `n`
    n characters or n decimal places (floating point) are output
  - `*`
    The next value in the argument list is an integer specifying the precision width

type may be:

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

- `c`
  character (e.g. 'a', 'b', '8')
- `s`
  string

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

SEE: Clib.sprintf()

EXAMPLE:

```
//Each of the following lines shows
// a printf example followed by what would show
// on the output in boldface:

Clib.printf("Hello world!")
// Hello world!
Clib.printf("I count: %d %d %d.",1,2,3)
// I count: 1 2 3
var a = 1;
var b = 2;
Clib.printf("%d %d %d", a, b, a +b)
// 1 2 3
```

Clib.getch()

SYNTAX: Clib.getch()

RETURN: number - character value of the key pressed.

DESCRIPTION: This method works exactly like getche(), but does not echo the returned key to the screen. For example, the following code has you enter a password; each time you enter a letter an asterisk is written to the screen:

SEE: Clib.getchar()

EXAMPLE:

```
var password;
for (var gg = 0; ;gg++)
{
var letter = Clib.getch();
if (letter == '\n') continue;
Clib.putc('*').
password[gg] = letter;
}
```

Clib.getchar()

SYNTAX:	Clib.getchar()
RETURN:	number - character value of the key pressed.
DESCRIPTION:	This method returns the next character from stdin. Usually, this is the keyboard, but you may redefine it to something else. This method will wait for "enter" to be pressed after the key, and will then return two values: the key pressed, and then the value of the enter key.
SEE:	Clib.getche()

Clib.getche()

SYNTAX:	Clib.getche()
RETURN:	number - character value of the key pressed.
DESCRIPTION:	This method waits until a key is pressed and returns the character value of that key. The character will be printed (echoed) to the screen. Some key presses, such as extended keys and function keys, may generate multiple getche() return values. If a key was pressed before calling the function but never cleared from the keyboard buffer, that value will be returned instead of the next pressed key. This is not a common occurrence but can happen. To see whether there are any key values pending in the keyboard buffer, use .kbhit().
SEE:	Clib.getch()

Clib.gets()

SYNTAX:	Clib.gets()
RETURN:	string - an entire string from the keyboard, or `null` if there was an error.
DESCRIPTION:	This method reads an entire string from the keyboard and returns it (or null if there was an error). The function will read all characters up to a newline character or EOF. If a newline

	character is read, it will not be included in the string.
SEE:	Clib.getchar()
EXAMPLE:	`var s = Clib.gets()`

Clib.kbhit()

SYNTAX:	Clib.kbhit()
RETURN:	boolean - `true` if there are any keystrokes waiting, `false` if not.
DESCRIPTION:	This method checks to see whether there are any keystrokes waiting to be processed, returning true if there are and false if there are not.
SEE:	Clib.getche()

Clib.putchar()

SYNTAX:	Clib.putchar(chr)
WHERE:	chr - character to write to the stream `stdout`.
RETURN:	number - character written on success, else `EOF`.
DESCRIPTION:	This method writes chr to the stream defined by `stdout` (usually the screen). If successful, it will return the character it just wrote; if not, it will return `EOF`. This method is identical to `Clib.fputc(chr, stdout)`.
SEE:	Clib.puts()

Clib.puts()

SYNTAX:	Clib.puts(str)
WHERE:	str - string to write to the stream `stdout`.
RETURN:	number - a positive number on success, else `EOF`.
DESCRIPTION:	Writes a string to `stdout`, followed by a newline character. Will not write the final null character of null terminated strings.

	Returns `EOF` if there is an error writing the string; otherwise it returns a positive number.
	This method is the same as `Clib.fputs(str, stdout)` except that a newline character is written after the string.
SEE:	Clib.putchar()

Clib.scanf()

SYNTAX:	Clib.scanf(formatString, variables[, ...])
WHERE:	formatString - specifies how to read and store data in variables.
	variables - list of variables to hold data input according to formatString.
RETURN:	number - input items assigned.
DESCRIPTION:	This flexible method reads input from the screen, extracts data from it by matching the string to a format string (as described below), and stores the data in the variables which follow the format string. It returns the number of input items assigned; this number may be fewer than the number of parameters requested if there was a matching failure. The format string contains character combinations that specify the type of data expected. The format string specifies the admissible input sequences, and how the input is to be converted to be assigned to the variable number of arguments passed to this function.

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

`%[*][width]<type>`

`*`, width, and type may be:

- `*`

suppress assigning this value to any parameter
- width
  maximum number of characters to read; fewer will be read if white space or nonconvertible character
- type
  may be one of the following:

  - `d, D, i, I`
    signed integer
  - `u, U`
    unsigned integer
  - `o,  O`
    octal integer
  - `x, X`
    hexadecimal integer
  - `f, e, E, g, G`
    floating point number
  - `c`
    character; if width was specified then this will be an array of characters of the specified length
  - `s`
    string
  - `[abc]`
    string consisting of all characters within brackets; where A- Z represents range "A" to "Z"
  - `[^abc]`
    string consisting of all character NOT within brackets.

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

SEE: Clib.vscanf()

Clib.vprintf()

SYNTAX: Clib.vprintf(formatString, valist)

WHERE: formatString - string that specifies the final format.

valist - a variable list of arguments to be used according to formatString.

RETURN: number - number of characters written on success, else a negative number.

DESCRIPTION: This method displays formatted output on the standard output stream, screen, using a variable number of arguments. This method is similar to .printf() except that it takes a variable argument list using valist.

See printf() and va_start() for more information. The method .vprintf() returns the number of characters written on success, else a negative number on error.

The example function acts just like a printf() statement except that it beeps, displays a message, beeps again, and waits a second before returning. This method could be a wrapper for the printf() method to display urgent messages.

SEE: Clib.printf(), Clib.va_start()

EXAMPLE:
```
function UrgentPrintf(FormatString[ arg1 ...])
{
   // create variable arg list
   Clib.va_start(valist, FormatString);
   Screen.write("\a"); // audible beep
   // printf original statement
   var ret = Clib.vprintf(FormatString, valist);
   Screen.write("\a"); // beep again
   SElib.suspend(1000); // wait before returning
   Clib.va_end(valist); // end using valist
   return(ret);       // return as printf would }
}
```

Clib.vscanf()

SYNTAX: Clib.vscanf(formatString, valist)

WHERE: formatString - string that specifies the final format.

valist - a variable list of arguments to be used according to formatString.

RETURN: number - input items assigned. This number may be fewer than the number of parameters requested if there is a matching failure during input.

DESCRIPTION: This method gets formatted input from the standard input stream, the keyboard, using a variable number of arguments. This method is similar to scanf() except that it takes a variable argument list. See scanf() and va_start() for more information.

The method vscanf() modifies any number of parameters following formatString, setting the parameters to data according to the specifications of the format string.

This method returns the number of input items assigned. This number may be fewer than the number of parameters requested if there is a matching failure during input.

The example function behaves like scanf(), including taking a variable number of input arguments, except that it beeps and tries again if there are zero matches:

SEE: Clib.scanf()

EXAMPLE:

```
function Must_scanf(FormatString[,arg1 ...)
{
   Clib.va_start(valist, FormatString);
      // creates variable arg list
   do
   {  // mimic original scanf() call
      var count = Clib.vscanf(FormatString,
                              valist);
      if ( 0 == count ) // if no match, beep
         Screen.write("\a");
   } while( 0 == count );
      // if not match, try again
   Clib.va_end(valist);
      // end using valist (optional)
   return(count);
      // return as scanf() would
}
```

Time functions

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, etc. This Time object is distinct from the standard JavaScript Date object. You cannot use Date object properties with a Time object or vice versa.

In the methods below, timeObj represents a variable in the Time object format, while timeInt represents an integral time value.

Clib.asctime()

SYNTAX:	Clib.asctime(timeObj)
WHERE:	timeObj - time variable in the Time object format.
RETURN:	string - the date and time extracted from a Time object, as returned by `Clib.localtime()`.
DESCRIPTION:	Returns a string representing the date and time extracted from a Time object, as returned by `Clib.localtime()`. The string will have this format: `Mon Jul 19 09:14:22 1993`

Clib.clock()

SYNTAX:	Clib.clock()
RETURN:	number - the current processor tick count.
DESCRIPTION:	Returns the current processor tick count. Clock value starts at 0 when ScriptEase program begins and is incremented `CLOCKS_PER_SEC` times per second.

Clib.ctime(timeInt)

SYNTAX:	Clib.ctime(timeInt)
WHERE:	timeInt - an integer time value.
RETURN:	string - the date and time extracted from a Time object, as returned by `Clib.localtime()`.
DESCRIPTION:	This method is equivalent to: `Clib.asctime( Clib.localtime(time) )`, where timeInt is a date_time value as returned by the `Clib.time()` function.

Clib.difftime()

SYNTAX:	Clib.difftime(timeInt0, timeInt1)
WHERE:	timeInt0 - an integer time value.
	timeInt1 - an integer time value.
RETURN:	number - difference between two times, in seconds.
DESCRIPTION:	This method returns the difference in seconds between two times. timeInt0 and timeInt1 are integral time values as returned by the time() function.

Clib.gmtime()

SYNTAX:	Clib.gmtime(timeInt)
WHERE:	timeInt - an integer time value.
RETURN:	number - the value timeInt (as returned by the time() function) as a Time object.
DESCRIPTION:	Takes the integer timeInt (as returned by the time() function) and converts it to a Time object representing the current date and time expressed as Greenwich mean time. See localtime() for a description of the returned object.

Clib.localtime()

SYNTAX:	Clib.localtime(timeInt)
WHERE:	timeInt - an integer time value.
RETURN:	number - the value timeInt (as returned by the time() function) as a Time object.
DESCRIPTION:	This method returns the value timeInt (as returned by the time() function) as a Time object. Note that the Time object differs from the Date object, although they contain the same data. The Time object is for use with the other date and time functions in the Clib object. It has the following integer properties:

- `.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

The following function prints the current date and time on the screen and returns the day of the year, where Jan 1 is the 1st day of the year.

EXAMPLE:

```
// Show today's date
// Return day of the year in USA format
ShowToday()
{
      // get current time structure
   var tm = localtime(time());
      // display the date in USA format
   Clib.printf("Date: %02d/%02d/%02d   ",
               tm.tm_mon+1,
   tm.tm_mday, tm.tm_year % 100);
      // hour to run from 12 to 11, not 0 to 23
   var hour = tm.tm_hour % 12;
   if ( hour == 0 )
      hour = 12;
      // print current time
   Clib.printf("Time: % 2d:%02d:%02d\n", hour,
               tm.tm_min,
   tm.tm_sec);
      // return day of year, Jan. 1 is day 1
   return( tm.tm_yday + 1 );
}
```

Clib.mktime()

SYNTAX: Clib.mktime(timeObj)

WHERE: timeObj - time variable in the Time object format.

RETURN: number - time integer, or -1 if time cannot be converted or represented.

DESCRIPTION: This method converts timeObj (an object as returned by .localtime()) to the time format returned by time() (an integer). All undefined elements of timeObj will be set to 0 before the conversion. It returns 1 if time cannot be converted or represented.

In other words, while localtime() converts from a time integer to a Time object, mktime() converts from a Time object to a time integer.

Clib.strftime()

SYNTAX: Clib.strftime(string, formatString, timeObj)

WHERE: string - a variable to receive the formatted time string.

formatString - string that specifies the final format.

timeObj - time variable in the Time object format.

RETURN: string - a string that describes the date and/or time and stores it in the variable string.

DESCRIPTION: This method creates a string that describes the date and or time and stores it in the variable string. The parameter formatString describes what the string will look like, and timeObj is a time object as returned by localtime().

These following conversion characters are used with `Clib.strftime()` to indicate time and date output:

- `%a`
  abbreviated weekday name (Sun)
- `%A`

full weekday name (Sunday)

- `%b`
  abbreviated month name (Dec)
- `%B`
  full month name (December)
- `%c`
  date and time (Dec 2 06:55:15 1979)
- `%d`
  two- digit day of the month (02)
- `%H`
  two- digit hour of the 24- hour day (06)
- `%I`
  two- digit hour of the 12- hour day (06)
- `%j`
  three- digit day of the year from 001 (335)
- `%m`
  two- digit month of the year from 01 (12)
- `%M`
  two- digit minute of the hour (55)
- `%p`
  AM or PM (AM)
- `%S`
  two- digit seconds of the minute (15)
- `%U`
  two- digit week of year, Sunday is first day of week (48)
- `%w`
  day of the week where Sunday is 0 (0)
- `%W`
  two- digit week of year, Monday is first day of 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`
  name of the time zone, if known (EST)
- `%%`
  the per cent character (%)

EXAMPLE:

```
// displays the full day name and month name
// of the current day
Clib.strftime(TimeBuf,
              "Today is: %A, the month is: %B",
              Clib.localtime(time()));
Clib.puts(TimeBuf);
```

Clib.time()

SYNTAX: Clib.time([t])

WHERE: t - variable to receive the time returned.

RETURN: number - integer representation of the current time.

DESCRIPTION: Returns an integer representation of the current time. The format of the time is not specifically defined except that it represents the current time, to the system's best approximation, and can be used in many other time related functions. If t is supplied then it will be set to equal the returned value.

Script execution

Clib.abort()

SYNTAX: Clib.abort([AbortAll])

WHERE: AbortAll - boolean flag as to whether to abort all levels of ScriptEase execution.

RETURN: number - `EXIT_FAILURE` to the operating system.

DESCRIPTION: This method terminates a program, usually when a specified error occurs. This method causes abnormal program termination and should only be called on a fatal error. This method exits, without returning to the caller, and returns `EXIT_FAILURE` to the operating system.

If the boolean AbortAll is true, this method aborts through all levels of ScriptEase interpretation. If you are in multiple levels of SElib.interpret(), .abort() aborts through all SElib.interpret() levels.

SEE: Clib.assert()

Clib.assert()

SYNTAX: Clib.assert(test)

WHERE: test - boolean flag to determine if the current file name and line number will be displayed and if the script will abort.

RETURN: void.

DESCRIPTION: If boolean evaluates to false this function will print the file name and line number to stderr and abort. If the assertion evaluates to true then the program continues. .assert() is typically used as a debugging technique to test assumptions before executing code based on those assumptions. Unlike C, the ScriptEase implementation of assert does not depend upon NDEBUG being defined or undefined; it is always active.

SEE: Clib.abort()

EXAMPLE:

```
// The Inverse() function below returns
// the inverse of the input number (1/x):
function Inverse(x)
{
   assert(0 != x);
   return 1 / x;
}
```

Clib.atexit()

SYNTAX: Clib.atexit(functionId)

WHERE: functionId - a function to be called when a script is exited.

RETURN: void.

DESCRIPTION: This method registers a function to be called when the script ends. The variable string passed to this function is the name of a function to be called.

Clib.exit()

SYNTAX:	Clib.exit(code)
WHERE:	code - status number to return to the operating system.
RETURN:	number - the status code of the exit is returned to the operating system from which a script was called.
DESCRIPTION:	This method causes normal program termination. It calls all functions registered with .atexit(), flushes and closes all open file streams, updates environment variables if applicable to this version of ScriptEase, and returns control to the OS environment with the return code of status.
SEE:	Clib.atexit()

Clib.system()

SYNTAX: Clib.system([P_SWAP,] commandString)

WHERE: P_SWAP - in DOS version, determines whether the ScriptEase interpreter is swapped out of normal memory.

commandString - the command string to be executed, a command as would be entered at a command prompt.

RETURN: value - the value returned by a command processor.

DESCRIPTION: Passes commandString to the command processor and returns whatever value was returned by the command processor. commandString may be a formatted string followed by variables according to the rules defined in .sprintf().

- DOS
  In the DOS version of ScriptEase, if the special argument P_SWAP is used then SeDos.exe is swapped to EMS/XMS/INT15 memory or disk while the system command is executed. This leaves almost all available memory for executing the command. See SElib.spawn() for a discussion of P_SWAP.

- DOS32
  The 32 bit protected mode version of DOS ignores the first parameter if it is an not a string; in other words, P_SWAP is ignored.

SEE: SElib.spawn()

Error

Clib.errno

SYNTAX: Clib.errno

DESCRIPTION: The property errno stores diagnostic message information when a function fails to execute correctly. Many functions in the Clib and SElib objects set errno to non-zero in case of error to provide more specific information about the error. ScriptEase implements errno as a macro to the internal function _errno(). This property can be accessed with perror() or strerror().

SEE: Clib.perror()

Clib.clearerr()

SYNTAX: Clib.clearerr(filePointer)

WHERE: filePointer - pointer to file for which error information is to be cleared.

RETURN: void.

DESCRIPTION: This method clears the error status and resents the end-of-file flags for the file associated with filePointer. There is no return value.

SEE: Clib.ferror()

Clib.ferror()

SYNTAX: Clib.ferror(filePointer)

WHERE: filePointer - pointer to file for which error information is to be retrieved.

RETURN: number - 0 on no file error, else the current error value associated with a file operation.

DESCRIPTION: The parameter filePointer is a file pointer as returned by fopen(). This method tests and returns the error indicator for stream file. Returns 0 if no error, otherwise returns the error value.

SEE: Clib.clearerr()

Clib.perror()

SYNTAX: Clib.perror([errmsg])

WHERE: errmsg - a message to describe an error condition.

RETURN: string - error message that describes the error indicated by Clib.errno.

DESCRIPTION: Prints and returns an error message that describes the error defined by Clib.errno. This method is identical to calling `Clib.strerror (Clib.errno)`. If a string variable is supplied it will be set to the string returned.

SEE: Clib.ferror()

Clib.strerror()

SYNTAX: Clib.strerror(errno)

WHERE: errno - an error number to convert to a descriptive string.

RETURN: string - an error number converted to a descriptive string.

DESCRIPTION: When some functions fail to execute properly, they store a number in the .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: Clib.perror()

EXAMPLE:
```
// Opens a file for reading, and if it cannot
// open the file then it prints a descriptive
// message and exits the program.
```

```
function MustOpen(filename)
{
   var fh = fopen(filename, "r");
   if ( fh == null )
   {
      Clib.printf("Error:%s\n",
                  Clib.strerror(errno));
      Clib.exit(EXIT_FAILURE);
   }
   return(fh);
}
```

File I/O

Clib.fopen()

SYNTAX: Clib.fopen(filename, mode)

WHERE: filename - a string with a filename to open.

mode - how or for what operations the file will opened.

RETURN: number - a file pointer to the file opened, `null` in case of failure.

DESCRIPTION: This method opens the file specified by filename for file operations specified by mode, returning a file pointer to the file opened. null is returned in case of failure.

The parameter filename is a string. It may be any valid file name, excluding wildcard characters.

The parameter mode is a string composed of one or more of the following characters. For example, "`r`" or "`rt`"

- `r`
  open file for reading; file must already exist
- `w`
  open file for writing; create if doesn't exist; if file exists then truncate to zero length
- `a`
  open file for append; create if doesn't exist; set for writing at end- of- file
- `b`

binary mode; if b is not specified then open file in text mode (end- of- line translation)

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

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

SEE: Clib.fclose(), Clib.flock()

EXAMPLE:

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

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

Clib.fclose()

SYNTAX: Clib.fclose(filePointer)

WHERE: filePointer - pointer to file to close.

RETURN: number - 0 on success, else `EOF`.

DESCRIPTION: The parameter filePointer is a file pointer as returned by Clib.fopen(). This method flushes the file buffers of a stream and closes the file. The file pointer ceases to be valid after this call. Returns zero if successful, otherwise returns `EOF`.

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

Clib.feof()

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

Clib.fflush()

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

Clib.fgetc()

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

Clib.fgetpos()

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

Clib.fgets()

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

Clib.fprintf()

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

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

Clib.fputc()

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

Clib.fputs()

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

SEE: Clib.fputc()

Clib.fread()

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

WHERE: dstVar - variable to hold data read from file.

varDescription - description of the data to read, that is, how and how much.

filePointer - pointer to file to use.

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

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

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

- `FLOAT64`
  Stored as a float in dstVar
- `FLOAT80`
  Stored as a float in dstVar (not available in Win32)

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

For example, the definition of an object might be:

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

The ScriptEase version of `Clib.fread()` differs from the standard C version in that the standard C library is set up for reading arrays of numeric values or structures into consecutive bytes in memory. In JavaScript this is not necessarily the case.

`Data types will be read from the file in a byte-order described by the current value of the _BigEndianMode global variable.`

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

EXAMPLE:

```
// To read the 16 bit integer "i",
// the 32 bit float "f", and
// then 10 byte buffer "buf"
// from the open file "fp"
// use code like the following.

if ( !Clib.fread(i,SWORD16,fp) ||
     !Clib.fread(f,FLOAT32,fp) ||
     (10 != Clib.fread(buf,10,fp)) )
{
   Clib.printf("Error reading from file.\n");
   Clib.abort();
}
```

Clib.freopen()

SYNTAX:	Clib.freopen(filename, mode, filePointer)
WHERE:	filename - a string with a filename to open.
	mode - how or for what operations the file will opened.
	filePointer - pointer to file to use.
RETURN:	number - file pointer on success, else `null`.
DESCRIPTION:	This method closes the file associated with filePointer, ignoring any close errors, opens filename according to mode, as with `Clib.fopen()` , and reassociates filePointer with the new file specification. This method is commonly used to redirect one of the pre-defined file handles (`stdout`, `stderr`, or `stdin`) to or from a file.
	The method returns a copy of the modified filePointer, or `null` if it fails.
	The example code calls ScriptEase for DOS with no parameters, which causes a help screen to be printed, and redirects `stdout` to a file *cenvi.out* so that *cenvi.out* will contain the text of the ScriptEase help screens.
SEE:	Clib.fopen()

EXAMPLE:

```
if ( null == Clib.freopen("cenvi.out", "w", stdout) )
   Clib.printf("Error redirecting stdout\a\n")
else
   Clib.system("SEDOS");
```

Clib.fscanf()

SYNTAX:	Clib.fscanf(filePointer, formatString[, variables ...])
WHERE:	filePointer - pointer to file to use.
	formatString - string that specifies the final format.
	variables - values to be converted to and formatted as a string.
RETURN:	number - input items assigned on success, else `EOF`.
DESCRIPTION:	This flexible function reads input from the file indicated by filePointer and stores in parameters following formatString according the character combinations in the format string, which

indicate how the file data is to be read and stored. The file must be open, with read access. It returns the number of input items assigned. This number may be fewer than the number of parameters requested if there was a matching failure. If there is an input failure, before the conversion occurs, this function returns `EOF`.

See `Clib.scanf()` for a description of this format string. The parameters following the format string will be set to data according to the specifications of the format string.

SEE: Clib.scanf()

EXAMPLE:

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

var fp = Clib.fopen("weight.dat", "r");
var FormatString = "%[,] %*c %s %d\n";
while (3 == Clib.fscanf(fp, FormatString,
       LastName, Firstame, weight))
   Clib.printf("%s %s weighs %d pounds.\n",
               FirstName, LastName, weight);
Clib.fclose(fp);

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

Clib.fseek()

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

WHERE: filePointer - pointer to file to use.

offset - number of bytes past or offset from the point indicated by mode.

mode - file position to use as a starting point. Default is `SEEK_SET` and may be one of the following:

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

- `SEEK_END`
  position is relative from the end of the file
- `SEEK_SET`
  position is relative to the beginning of the file

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

DESCRIPTION: Set the position of the file pointer of the open file stream filePointer. The parameter offset is a number indicating how many bytes the new position will be past the starting point indicated by mode.

If mode is not supplied then absolute offset from the beginning of file, `SEEK_SET`, is assumed. For text files, not opened in binary mode, the file position may not correspond exactly to the byte offset in the file.

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

Clib.fsetpos()

SYNTAX: Clib.fsetpos(filePointer, pos)

WHERE: filePointer - pointer to file to use.

pos - position in file to set.

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

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

SEE: Clib.fseek()

Clib.ftell()

SYNTAX: Clib.ftell(filePointer)

WHERE: filePointer - pointer to file to use.

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

Clib.fwrite()

SYNTAX:	Clib.fwrite(srcVar, varDescription, filePointer)
WHERE:	srcVar - variable to hold data read from file.
	varDescription - description of the data to read, that is, how and how much.
	filePointer - pointer to file to use.
RETURN:	number - elements written on success, else 0 if a write error occurs.
DESCRIPTION:	This method writes the data in srcVar to the file indicated by filePointer and returns the number of elements written. 0 will be returned if a write error occurs. Use `Clib.ferror()` to get more information about the error. varDescription is a variable that describes the how and how much data is to be read. If srcVar is a buffer, it will be the length of the buffer. If srcVar is an object, varDescription must be an object descriptor. If srcVar is to hold a single datum then varDescription must be one of the values listed in the description for `Clib.fread()`.
	The ScriptEase 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 JavaScript.
SEE:	Clib.fread()

EXAMPLE:

```
// To write the 16_bit integer "i",
// the 32_bit float "f", and
// then 10_byte buffer "buf" into open file "fp",
// use the following code.

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

Clib.getc()

SYNTAX: Clib.getc(filePointer)

WHERE: filePointer - pointer to file to use.

RETURN: number - on success, the next character, as an unsigned byte converted to an integer, in a file. Else `EOF` if a read error or at the end of file.

DESCRIPTION: This method is identical to `Clib.fgetc()`. It returns the next character in a file as an unsigned byte converted to an integer. Returns EOF if there is a read error or if at the end of the file. If there is a read error then `Clib.ferror()` will indicate the error condition.

SEE: Clib.gets()

Clib.putc()

SYNTAX: Clib.putc(chr, filePointer)

WHERE: chr - character to write to file.

filePointer - pointer to file to use.

RETURN: number - character written on success, else `EOF` on write error.

DESCRIPTION: This method writes the character chr, converted to a byte, to an output file stream. This method is identical to `Clib.fputc()`. It returns chr on success and `EOF` on a write error.

SEE:	Clib.fputc()

Clib.remove()

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

Clib.rename()

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

Clib.rewind()

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

Clib.tmpfile()

SYNTAX:	Clib.tmpfile()
RETURN:	number - on success, a file pointer to a temporary binary file that will automatically be removed when it is closed or when the program exits, else `null` on failure.
DESCRIPTION:	This method returns the file pointer of a temporary binary file that will automatically be removed when it is closed or when the program exits. Returns `null` if the function fails.
SEE:	Clib.tmpnam()

Clib.tmpnam()

SYNTAX:	Clib.tmpnam([str])
WHERE:	str - a variable to hold the name of a temporary file.
RETURN:	string - a valid and unique filename.
DESCRIPTION:	This method creates a string that is a valid file name that is not the same as the name of any existing file and not the same as any filename returned by this function during execution of this program. If str is supplied it will be set to the string returned by this function.
SEE:	Clib.tmpfile()

Clib.ungetc(chr, filePointer)

SYNTAX:	Clib.ungetc(chr, stream)
WHERE:	chr - character to write to file.
	filePointer - pointer to file to use.
RETURN:	number - on success, the character put back into a file stream, else `EOF`.
DESCRIPTION:	This method pushes character chr back into an input stream.

	When chr is put back, it is converted to a byte and is again in an input stream for subsequent retrieval. Only one character is guaranteed to be pushed back. The method returns chr on success, else `EOF` on failure.
SEE:	Clib.getc()

Directory

Clib.chdir()

SYNTAX:	Clib.chdir(dirpath)
WHERE:	dirpath - directory specification to which to change.
RETURN:	number - 0 on success, else -1.
DESCRIPTION:	This method changes the directory for a script from its current directory to the directory specified in the parameter dirpath. The specified directory may be an absolute or relative path specification.
SEE:	Clib.getcwd()

Clib.getcwd()

SYNTAX:	Clib.getcwd()
RETURN:	string - complete path of the current working directory for a script.
DESCRIPTION:	This method returns the complete path of the current working directory for a script.
SEE:	Clib.chdir()

Clib.flock()

SYNTAX:	Clib.flock(filePointer, lockFlag)
WHERE:	filePointer - pointer to file to use.

lockFlag - determines which locking operation to perform on a file. The flags are:

- `LOCK_EX`
  File lock exclusive (equivalent to `LOCK_SH` in Windows)
- `LOCK_SH`
  File lock share (equivalent to `LOCK_EX` in Windows)
- `LOCK_NB`
  File lock non-blocking (bitwise or with `LOCK_EX` or `LOCK_SH`)
- `LOCK_UN`
  File unlock

RETURN: number - 0 on success, else -1 on failure.

DESCRIPTION: This method allows a file to be locked or unlocked, which is a capability that is often important in a multi-tasking operating system.

The ability to lock and unlock access to a file varies among operating systems. For normal usage on most systems, the operating system handles all necessary locking and administration of sharing privileges for files. However, if a scripter needs extra control over files, ScriptEase provides the ability. For example, a script might use files to hold data while it is running but does not need to keep the files open during all phases of script execution. By locking and unlocking such files, a scripter ensures that these files are not altered while a script is running.

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

EXAMPLE:

```
// The following fragment opens a file and
// then locks it for exclusive use without blocking
// further execution of the script.

var fp = Clib.fopen("myfile", "r");
Clib.flock(fp, LOCK_EX | LOCK_NB);
   // Use the file
Clib.flock(fp, LOCK_UN);
Clib.fclose(fp);
```

Clib.mkdir()

SYNTAX:	Clib.mkdir(dirpath)
WHERE:	dirpath - directory specification to make.
RETURN:	number - 0 on success, else -1.
DESCRIPTION:	This method creates the directory specified in the parameter dirpath. The specified directory may be an absolute or relative path specification.
SEE:	Clib.rmdir(), Clib.chdir()

Clib.rmdir()

SYNTAX:	Clib.rmdir(dirpath)
WHERE:	dirpath - directory specification to delete.
RETURN:	number - 0 on success, else -1.
DESCRIPTION:	This method deletes the directory specified by the parameter dirpath.
SEE:	Clib.mkdir(), Clib.remove()

Sorting

Clib.bsearch()

SYNTAX:	Clib.bsearch(key, array[, elementCount], compareFunction)
WHERE:	key - value for which to search.
	array - beginning of array to search.
	elementCount - number of elements to search. Default is the entire array.
	compareFunction - function used to compare key with each element searched in the array.
RETURN:	value - the element in an array if found, else `null` if not found.

DESCRIPTION: This method looks for an array variable that matches key, returning it if found and null if not. It will only search through positive array members (array members with negative indices will be ignored). The compareFunction must receive the key variable as its first argument and a variable from the array as its second argument. If elementCount is not supplied then will search the entire array. The elementCount is limited to 64K for 16 bit version of ScriptEase.

SEE: Clib.qsort()

EXAMPLE:
```
// This example creates a two dimensional array
// that pairs a name with a favorite food.
// A name is searched for. The name and paired
// food is displayed.

var Found;
var Key;
var list;

   // create array of names and favorite food
var list =
{
   {"Marge",    "salad"},
   {"Lisa",     "tofu"},
   {"Homer",    "sugar"},
   {"Bart",     "anything"},
   {"Itchy",    "cats"},
   {"Scratchy", "anything from the garbage"}
};
   // sort the list
Clib.qsort(list, ListCompareFunction);

Key[0] = "marge";
   // search for the name Marge in the list
Found = Clib.bsearch(Key, list, ListCompareFunction);
   // display name, or not found

if (Found != null)
   Clib.printf("%s's favorite food is %s\n",
               Found[0], Found[1])
else
   Clib.puts("Could not find name in list.");

   // This compare function is used to sort
   // the array and to find a name.
   // The sort and search are case insensitive.
function ListCompareFunction(Item1, Item2)
```

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

Clib.qsort()

SYNTAX: Clib.qsort(array[, elementCount], CompareFunction)

WHERE: array - array to sort.

elementCount - number of elements to sort. Default is the entire array.

compareFunction - function used to compare key with each element searched in the array.

RETURN: void.

DESCRIPTION: This method sorts elements in an array, starting from index 0 to elementCount- 1. If elementCount is not supplied then will sort the entire array. This method differs from the `Array.sort()` method in that it can sort automatically created arrays, whereas `Array.sort()` only works with arrays explicitly created with a `new Array` statement.

The value of elementCount is limited to 64K

SEE: Clib.bsearch(), Array()

EXAMPLE:

```
// Create a list of color names,
// sort the list in reverse alphabetical order,
// case insensitive, and display the list.

   // initialize an array of colors
var colors = {"yellow", "Blue", "GREEN", "purple",
              "RED", "BLACK", "white", "orange"};

   // sort the list ReverseColorSorter function
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 is:
//  yellow
//  white
//  RED
//  purple
//  orange
//  GREEN
//  Blue
//  BLACK
```

Environment variables

Clib.getenv()

SYNTAX: Clib.getenv([variableName])

WHERE: variableName - the name of an environment variable.

RETURN: value - a string representation of the value of an environment variable on success. If no variableName is passed, an array of all environment variable names. On failure, returns `null`.

DESCRIPTION: If the parameter variableName is supplied, this method returns the value of a similarly named environment variable as a string, if the variable exists, and `null` if VariableName does not exist. If no name is supplied then returns an array of all environment variable names, ending with a `null` element.

SEE: Clib.putenv()

EXAMPLE:

```
// Print the existing environment variables,
// in "EVAR=Value" format,
// sorted alphabetically.

   // get array of all environment variable names
var EnvList = Clib.getenv();
   // sort array alphabetically
Clib.qsort(EnvList, getArrayLength(EnvList),
           Clib.stricmp);
   // display each element in ENV=VALUE format
for ( var lIdx = 0; EnvList[lIdx]; lIdx++ )
```

```
                         Clib.printf("%s=%s\n", EnvList[lIdx],
                                     Clib.getenv(EnvList[lIdx]));
```

Clib.putenv()

SYNTAX:	Clib.putenv(variableName, stringValue)
WHERE:	variableName - the name of an environment variable.
	stringValue - new value for environment variable variableName.
RETURN:	number - 0 on success, else -1.
DESCRIPTION:	This method sets the environment variable variableName to the value of stringValue. If stringValue is `null` then variableName is removed from the environment. For those operating systems in which ScriptEase can alter the parent environment (DOS or OS/2 when invoked with *SD.bat* or *SEset.cmd*) the variable setting will still be valid when ScriptEase exits; otherwise the variable change applies only to the ScriptEase code and to child processes of the ScriptEase program. Returns - 1 if there is an error, else 0.
SEE:	Clib.getenv()

Character classification

JavaScript does not have a true character type. For the character classification routines, a chr is actually a single 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');
Screen.writeln(t);

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

This fragment displays the following.

```
true
true
```

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

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

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

All character classification methods return booleans: true or false.

Clib.isalnum()

SYNTAX:	Clib.isalnum(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is in: A-Z, a-z, or 0-9. Else false.
DESCRIPTION:	Returns true if chr is a character in one of the following sets: A-Z, a-z, or 0-9.

Clib.isalpha()

SYNTAX:	Clib.isalpha(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is in: A-Z or a-z. Else false.
DESCRIPTION:	Returns true if chr is a alphabetic character in one of the following sets of characters: A-Z or a-z.

Clib.isascii()

SYNTAX:	Clib.isascii(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is in ASCII: 0-127.
DESCRIPTION:	Returns true if chr is an ASCII character in the following set of codes: 0-127.

Clib.iscntrl()

SYNTAX:	Clib.iscntrl(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is in ASCII: 0-31 or 127.
DESCRIPTION:	Returns true if chr is a control character in the set of ASCII characters. Control characters are in one of the following sets of codes: 0-31 or 127.

Clib.isdigit()

SYNTAX:	Clib.isdigit(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is in: 0-9.
DESCRIPTION:	Returns true if chr is a decimal digit in the following set of characters: 0-9.

Clib.isgraph()

SYNTAX:	Clib.isgraph(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is a printable character.
DESCRIPTION:	Returns true if chr is a printable character excluding the space character " ", code 32.

Clib.islower()

SYNTAX:	Clib.islower(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is in: a-z.
DESCRIPTION:	Returns true if chr is a lowercase character in the following set of

	characters: a- z

Clib.isprint()

SYNTAX:	Clib.isprint(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr a printable ASCII code in: 32-126.
DESCRIPTION:	Returns true if chr is a printable character in the following set of codes: 32-126.

Clib.ispunct()

SYNTAX:	Clib.ispunct(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - if chr is a punctuation character code in: 32-47, 58-63, 91-96, or 123-126.
DESCRIPTION:	Returns true if chr is a punctuation character in one of the following sets of codes: 32-47, 58-63, 91-96, or 123-126.

Clib.isspace()

SYNTAX:	Clib.isspace(chr)
WHERE:	chr - a character, a single character string.
RETURN:	boolean - true if chr is a white space in ASCII: 9, 10, 11, 12, 13, or 32.
DESCRIPTION:	Returns true if chr is a white space character, that is, one of the following codes: 9, 10, 11, 12, 13, or 32 (horizontal tab, new line, vertical tab, form feed, carriage return, or space).

Clib.isupper()

SYNTAX:	Clib.isupper(chr)

WHERE: chr - a character, a single character string.

RETURN: boolean - true if chr is in: A-Z.

DESCRIPTION: Returns true if chr is an uppercase character in the following set of characters: A- Z.

Clib.isxdigit()

SYNTAX: Clib.isxdigit(chr)

WHERE: chr - a character, a single character string.

RETURN: boolean - true if chr is in: 0-9, A-F, or a-f.

DESCRIPTION: Returns true if chr is a hexadecimal digit in one of the following sets of characters: 0-9, A-F, or a-f.

String manipulation

Clib.rsprintf()

SYNTAX: Clib.rsprintf(formatString[, variables ...])

WHERE: formatString - string that specifies the final format.

variables - values to be converted to and formatted as a string.

RETURN: string - formatted according to formatString using any variables passed.

DESCRIPTION: This method returns a `formatted` string. It is similar to `Clib.printf()`, except that a string is returned instead of printed.

SEE: Clib.printf()

EXAMPLE:

```
// If in a script you had a line:

Clib.printf("%s has seen %s %d times.\n", name,
            movie, timesSeen);

// and you wanted to pass the resulting string
// as a parameter to a function, you could do it
// as follows.
```

```
func(Clib.rsprintf("%s has seen %s %d times.\n",
                    name, movie, timesSeen));

// The following lines of code achieve
// the same result, that is, create
// a string variable named word that contains
// the string "Who is #1?".

var word
word = Clib.rsprintf("Who is #%d?", 3-2);
Clib.sprintf(word, "Who is #%d?", 3-2);
```

Clib.rvsprintf()

SYNTAX:	Clib.rvsprintf(formatString, valist)
WHERE:	formatString - string that specifies the final format.
	valist - a variable list of arguments to be used according to formatString.
RETURN:	string - specified by formatString on success, else `EOF` on error.
DESCRIPTION:	This method returns formatted output using the variable argument list represented by the parameter valist, a Blob. This method is similar to `Clib.sprintf()` except that it takes a variable argu ment list and returns a formatted string based on the arguments, rather than storing it in a string buffer. See `Clib.sprintf()` and `Clib.va_start()` for more information. The method `Clib.rvsprintf()` returns a string specified by formatString on success, else `EOF` on error.
SEE:	Clib.sprintf(), Clib.vprintf()

Clib.sscanf()

SYNTAX:	Clib.sscanf(str, formatString[, variables ...])
WHERE:	str - string holding the data to read into variables according to formatString.
	formatString - specifies how to read and store data in variables.
	variables - list of variables to hold data input according to

	formatString.
RETURN:	number - input items assigned. May be lower than the number of items requested if there is a matching failure.
DESCRIPTION:	This flexible method reads data from a string and stores it in variables passed as parameters following formatString. The parameter formatString specifies how data is read and stored in variables. See `Clib.scanf()` for details about formatString. `Clib.scanf()` reads data from the standard input stream, whereas this method, `Clib.sscanf()` reads data from a string.
SEE:	Clib.scanf(), Clib.fscanf(), Clib.vscanf()

Clib.sprintf()

SYNTAX:	Clib.sprintf(str, formatString[, variables ...])
WHERE:	str - to hold the formatted output. formatString - string that specifies the final format. variables - values to be converted to and formatted as a string.
RETURN:	number - characters written to string on success, else `EOF` on failure.
DESCRIPTION:	This method writes output to the string variable specified by str according to formatString, and returns the number of characters written or `EOF` if there was an error. The parameter formatString may contain character combinations indicating how following parameters are to be written. The parameter str need not be previously defined. It will be created large enough to hold the result. The format string may contain character combinations indicating how following parameters are to be treated. Characters are handled normally until a percent character, `%`, is reached. The percent `%` indicates that a value is to be written from the variables following the format string. See `Clib.printf()` for a complete description of formatString.
SEE:	Clib.printf()

EXAMPLE:

```
// Each of the following lines shows
// a sprintf example followed
// by the resulting string.

Clib.sprintf(testString, "I count: %d %d %d.",1,2,3)

//     "I count: 1 2 3"

var a = 1;
var b = 2;
Clib.sprintf(testString, "%d %d %d", a, b, a+b)

//      "1 2 3"
```

Clib.strcat()

SYNTAX: Clib.strcat(dstStr, srcStr)

WHERE: dstStr - destination string to which to add srcStr and to hold the final result.

srcStr - source string to append to dstStr.

RETURN: string - the resulting string from concatenating dstStr and srcStr.

DESCRIPTION: This method appends srcStr string onto the end of dstStr string. The dstStr string is made big enough to hold srcStr, and a terminating `null` byte. In ScriptEase, a string copy is safe, so that you can copy from one part of a string to another part of itself.

The return is the value of dstStr, that is, a variable pointing to the dstStr array starting at dstStr[0].

SEE: Clib.strcpy(), Clib.memcpy()

EXAMPLE:

```
// The result of the following code is:
//  Giant == "Fee Fie Foe Fum"

var Giant = "Fee";
   // add Fie
Clib.strcat(Giant, " Fie");
   // add Foe
Clib.strcat(Giant, " Foe");
   // add Fum
Clib.strcat(Giant, " Fum");
```

Clib.strchr()

SYNTAX: Clib.strchr(str, chr)

WHERE: str - string to search for a character.

chr - character to search for.

RETURN: string - beginning at the point in string where chr is found, else `null` if is not found..

DESCRIPTION: This method searches the parameter str for the character chr. It returns a variable indicating the first occurrence of chr in str, else it returns `null` if chr is not found in str.

SEE: Clib.strstr(), String indexOf()

EXAMPLE:

```
// The following code fragment:

var str = "I can't stand soggy cereal."
var substr = Clib.strchr(str, 's');
Clib.printf("str = %s\n", str);
Screen.writeln("substr = " + substr);

// results in the following output.
//  str = I can't stand soggy cereal.
//  substr = stand soggy cereal.
```

Clib.strcmp()

SYNTAX: Clib.strcmp(str1, str2)

WHERE: str1 - first string to compare.

str2 - second string to compare

RETURN: number - negative, zero, or positive according to the following rules:

- **`< 0`** if str1 is less than str2
- **`= 0`** if str1 is the same as str2
- **`> 0`** if str1 is greater than str2

DESCRIPTION: This method does a case- sensitive comparison of the characters of str1 with str2 until there is a mismatch or a terminating null byte is reached.

SEE:	Clib.strcmpi(), Clib.stricmp(), ==, ===

Clib.strcmpi()

SYNTAX:	Clib.strcmpi(str1, str2)
WHERE:	str1 - first string to compare. str2 - second string to compare
RETURN:	• **`< 0`** if str1 is less than str2 • **`= 0`** if str1 is the same as str2 • **`> 0`** if str1 is greater than str2
DESCRIPTION:	This method does a case- insensitive comparison of the characters of str1 with str2 until there is a mismatch or a terminating null byte is reached.
SEE:	Clib.strcmp(), Clib.stricmp(), ==, ===

Clib.strcpy()

SYNTAX:	Clib.strcpy(dstStr, srcStr)
WHERE:	dstStr - destination string to which the source string will be copied. srcStr - source string to copy to destination string.
RETURN:	string - the value of dstStr after the copy process.
DESCRIPTION:	This method copies bytes from srcStr to dstStr, up to and including the terminat ing `null` character. If dstStr is not already defined, then it is defined as a string. It is safe to copy from one part of a string to another part of the same string. The return is the value of dstStr, that is, a variable pointing to the dstStr array starting at dstStr[0].
SEE:	Clib.strncpy(), =

Clib.strcspn()

SYNTAX:	Clib.strcspn(str, chrSet)
WHERE:	str - string to be searched. chrSet - set of characters to search for.
RETURN:	number - offset into str to a found character on success, else the length of str.
DESCRIPTION:	This method searches the parameter string for any of the characters in the string chrSet and returns the offset of that character. If no matching characters are found, it returns the length of the string. This method is similar to `Clib.strpbrk()`, except that `Clib.strcspn()` returns the offset number, or index, for the first character found, while `Clib.strpbrk.()` returns the string beginning at that character.
SEE:	Clib.strpbrk()

EXAMPLE:

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

var string =
    "There's more than one way to skin a cat.";
var rStrpbrk = Clib.strpbrk(string, "dxb8w9k!");
var rStrcspn = Clib.strcspn(string, "dxb8w9k!");
Clib.printf("The string is: %s\n", string);
Clib.printf("\nstrpbrk returns a string: %s\n",
            rStrpbrk);
Clib.printf("\nstrcspn returns an integer: %d\n",
            rStrcspn);
Clib.printf("string +strcspn = %s\n", string +
            rStrcspn); Clib.getch();

// And results in the following output:
//  The string is:
//  There's more than one way to skin a cat.
//  strpbrk returns a string: way to skin a cat.
//  strcspn returns an integer: 22
//  string +strcspn = way to skin a cat
```

Clib.stricmp()

SYNTAX: Clib.stricmp(str1, str2)

WHERE: str1 - first string to compare.

str2 - second string to compare

RETURN:

- **`< 0`** if str1 is less than str2
- **`= 0`** if str1 is the same as str2
- **`> 0`** if str1 is greater than str2

DESCRIPTION: This method does a case- insensitive comparison of the characters of str1 with str2 until there is a mismatch or a terminating null byte is reached.

SEE: Clib.strcmp(), Clib.strcmpi(), ==, ===

Clib.strlen()

SYNTAX: Clib.strlen(str)

WHERE: str - string to find length of.

RETURN: number - the number of characters in str, not including the terminating `null` character.

DESCRIPTION: This method returns the length of parameter str. The length property of JavaScript strings is similar. The difference between `Clib.strlen(str)` and `str.length` is that `str.length` counts `null` characters as part of a string, whereas `Clib.strlen()` considers them to be markers indicating the end of the string and does not include them or any characters which follow them as part of a string.

The return is the number of characters, bytes, in str, starting from the character at str[0] and ending before the terminating null-byte.

SEE: String length

Clib.strlwr()

SYNTAX: Clib.strlwr(str)

WHERE: str - string in which to change case of characters to lowercase.

RETURN:	string - the value of str after conversion of case.
DESCRIPTION:	This method converts all uppercase letters in str to lowercase, starting at str[0] and ending before the terminating null byte. The return is the value of str, that is, a variable pointing to the start of str at str[0].
SEE:	Clib.strupr(), String toLowerCase()

Clib.strncat()

SYNTAX: Clib.strncat(dstStr, srcStr, maxLen)

WHERE: dstStr - destination string to which to add srcStr and to hold the final result.

srcStr - source string to append to dstStr.

maxLen - maximum number of characters to append from srcStr.

RETURN: string - the value of the destination string after the source string characters have been appended.

DESCRIPTION: This method appends up to maxLen bytes of srcStr onto the end of dstStr. Characters following a null- byte in srcStr are not copied. The dstStr array is made big enough to hold:

```
Clib.min( Clib.strlen(srcStr),maxLen)
```

characters and a terminating `null` character. The final value of dstStr is returned.

SEE: Clib.strcat()

Clib.strncmp()

SYNTAX: Clib.strncmp(str1, str2, maxLen)

WHERE: str1 - first string to compare.

str2 - second string to compare

maxLen - maximum number of characters to use for comparison.

RETURN: number - negative, zero, or positive according to the following rules:

- **`< 0`** if str1 is less than str2
- **`= 0`** if str1 is the same as str2
- **`> 0`** if str1 is greater than str2

DESCRIPTION: This method compares up to maxLen bytes of str1 against str2 until there is a mismatch or reach the terminating `null` byte. The comparison is case-sensitive. The comparison ends when maxLen bytes have been compared or when a terminating `null` byte has been compared, whichever comes first.

SEE: Clib.strncmpi(), Clib.strnicmp(), ==, ===

Clib.strncmpi()

SYNTAX: Clib.strncmpi(str1, str2, maxLen)

WHERE: str1 - first string to compare.

str2 - second string to compare

maxLen - maximum number of characters to use for comparison.

RETURN: number - negative, zero, or positive according to the following rules:

- **`< 0`** if str1 is less than str2
- **`= 0`** if str1 is the same as str2
- **`> 0`** if str1 is greater than str2

DESCRIPTION: This method compares up to maxLen bytes of str1 against str2 until there is a mismatch or reach the terminating `null` byte. The comparison is case-insensitive. The comparison ends when maxLen bytes have been compared or when a terminating `null` byte has been compared, whichever comes first.

SEE: Clib.strncmp(), Clib.strnicmp(), ==, ===

Clib.strncpy()

SYNTAX: Clib.strncpy(dstStr, srcStr, maxLen)

WHERE: dstStr - destination string to which the source string will be copied.

srcStr - source string to copy to destination string.

maxLen - maximum number of characters to copy.

RETURN: string - the value of dstStr after the copy process.

DESCRIPTION: This method copies:

```
Clib.min(Clib.strlen(srcStr)+1, MaxLen)
```

characters from srcStr to dstStr. If dstStr is not already defined then this method defines it as a string. The destination string is padded with `null` characters, if maxLen is greater than the length of srcStr, and a `null` character is appended to dstStr if maxLen characters are copied. It is safe to copy from one part of a string to another part of the same string. Returns the value of dstStr; that is, a variable into the destination array based at dstStr[0].

SEE: Clib.strcpy()

Clib.strnicmp()

SYNTAX: Clib.strnicmp(str1, str2, maxLen)

WHERE: str1 - first string to compare.

str2 - second string to compare

maxLen - maximum number of characters to use for comparison.

RETURN: number - negative, zero, or positive according to the following rules:

- **`< 0`** if str1 is less than str2
- **`= 0`** if str1 is the same as str2
- **`> 0`** if str1 is greater than str2

DESCRIPTION: This method compares up to maxLen bytes of str1 against str2 until there is a mismatch or reach the terminating `null` byte. The comparison is case-insensitive. The comparison ends when maxLen bytes have been compared or when a terminating `null` byte has been compared, whichever comes first.

SEE: Clib.strncmp(), Clib.strncmpi(), ==, ===

Clib.strpbrk()

SYNTAX: Clib.strpbrk(str, chrSet)

WHERE: str - string to be searched.

chrSet - set of characters to search for.

RETURN: string - beginning with the character in chrSet that was found, else `null`.

DESCRIPTION: This method searches str for any of the characters in chrSet, and returns the string based at the found character. Returns `null` if no character from chrSet is found.

`Clib.strcspn()` returns a number and `Clib.strpbrk()` returns a string.

SEE: Clib.strcspn()

EXAMPLE:

```
// See Clib.strcspn() for an example
// using this function.
```

Clib.strrchr()

SYNTAX: Clib.strrchr(str, chr)

WHERE: str - string to search.

chr - character to search for.

RETURN: string - beginning with the first character found from the right, else `null`.

DESCRIPTION: This method searches a string for the last occurrence of chr. The search is in the reverse direction, from the right, for chr in a string. The method returns a variable indicating the last occurrence of chr in a string, else it returns `null` if chr is not found in str.

SEE: Clib.strchr()

EXAMPLE:

```
// The following code:
```

```
var str = "I can't stand soggy cereal."
var substr = Clib.strrchr(str, 's');
Clib.printf("str = %s\n", str);
Screen.writeln("substr = " + substr);

// Results in the following output.
//  str = I can't stand soggy cereal.
//  substr = soggy cereal.
```

Clib.strspn()

SYNTAX:	Clib.strspn(str, chrSet)
WHERE:	str - string to be searched.
	chrSet - set of characters to search for.
RETURN:	number - the offset or index into str of the first character that is not in chrSet.
DESCRIPTION:	This method searches a string for any characters that are not in chrSet, and returns the offset of the first instance of such a character. If all characters in str are also in chrSet, the return is the length of string.
SEE:	Clib.strcspn()

Clib.strstr()

SYNTAX:	Clib.strstr(srcStr, findStr)
WHERE:	srcStr - a string to search.
	findStr - a string to find.
RETURN:	string - beginning in srcStr with the first character in findStr that was found, else `null`.
DESCRIPTION:	This method searches srcStr, starting at srcStr[0], for the first occurrence of findStr. The search is case-sensitive. The method returns a variable indicating the beginning of the first occurrence of findStr in srcStr, else it returns `null` if findStr is not found in srcStr.
SEE:	Clib.strchr(), Clib.strstri()

EXAMPLE:

```
// The following code fragment:

cfunction main()
{
   var Phrase = "To be or not to be? Beep beep!";
   do
   {
      Screen.writeln(Phrase);
      Phrase = Clib.strstr(Phrase + 1, "be");
   } while (Phrase != null);
}
// results in the following output.
//  To be or not to be? Beep beep!
//  be or not to be? Beep beep!
//  be? Beep beep!
//  beep!
```

Clib.strstri()

SYNTAX: Clib.strstri(srcStr, findStr)

WHERE: srcStr - a string to search.

findStr - a string to find.

RETURN: string - beginning in srcStr with the first character in findStr that was found, else `null`.

DESCRIPTION: This method searches srcStr, starting at srcStr[0], for the first occurrence of findStr. The search is case-insensitive. The method returns a variable indicating the beginning of the first occurrence of findStr in srcStr, else it returns `null` if findStr is not found in srcStr.

SEE: Clib.strstr()

Clib.strtod()

SYNTAX: Clib.strtod(str[, endStr])

WHERE: str - string to be converted to a number.

endStr - the part of str after the characters that were actually parsed.

RETURN: number - the first part of str converted to a double precision number.

DESCRIPTION: This method converts the string str into a number and optionally returns a partial string that begins beyond the characters parsed by this method. White space characters are skipped at the start of str, and the string characters are converted to a float as long as they match the following format.

[sign][digits][.][digits][format[sign]digits]

The parameter endStr is not compared against `null`, as it is in standard C implementations, and is optional. If the parameter endStr is supplied, then endStr is set to a string beginning at the first character that was not used in converting.

The return is the first part of str, converted to a floating-point num ber.

SEE: Clib.strtok()

EXAMPLE:

```
// The following strings, are examples
// that can be converted.
//  "1"
//  "1.8"
//  "-400.456e-20"
//  ".67e50"
//  "2.5E+50"
```

Clib.strtok()

SYNTAX: Clib.strtok(srcStr, delimiterStr)

WHERE: srcStr - source string consisting of delimited tokens.

delimiterStr - string of delimiter characters that separate tokens.

RETURN: string - a token, a substring, in srcStr, else `null` if there is not a token or if there are no more tokens.

DESCRIPTION: This method is unusual. The parameter srcStr is a string that consists of text tokens, substrings, separated by delimiter characters found in delimiterStr. The parameter srcStr may be altered during the first and subsequent calls to `Clib.strtok()`.

On the first call to `Clib.strtok()`, srcStr points to the string to tokenize and delimiterStr is a set of characters which are used to separate tokens in the source string. The first call, such as:

```
token = Clib.strtok(srcStr, delimiterStr)
```

returns a variable pointing to the srcStr array and based at the first character of the first token in srcStr. On subsequent calls, such as

```
token = Clib.strtok(null, delimiterStr)
```

the first argument is null and `Clib.strtok()` will continue through srcStr returning subsequent tokens.

The initial variable receiving tokens must remain valid throughout following calls that use `null`. If the variable is changed in any way, a subsequent use of `Clib.strtok()` must first use the syntax form in which the new string, not `null`, is passed as a first parameter.

This method returns `null` if there are no more tokens; otherwise returns srcStr array variable based at the next token in srcStr.

SEE: Clib.strstr()

EXAMPLE:

```
// The following code:

var source =
   " Little   John,,,Eats ?? crackers;;;! ";
var token = Clib.strtok(source,", ");
while(null != token)
{
   Clib.puts(token);
   token = Clib.strtok(null,";?, ");
}

// produces the following list of tokens.
//  Little
//  John
//  Eats
//  crackers
//  !
```

Clib.strtol()

SYNTAX: Clib.strtol(str[, endStr[, radix]])

WHERE: str - string to be converted to a number.

endStr - the part of str after the characters that were actually parsed.

radix - the number base for the conversion.

RETURN: number - the first part of str converted to a long integer number.

DESCRIPTION: This method converts the string str into a number and optionally returns a string starting beyond the characters parsed in the method. White space characters are skipped at the start of str, and the string characters are converted to an integer as long as they match the following format.

[sign][0][x][digits]

The parameter endStr is not compared against `null`, as it is in standard C implementations and is optional. The parameter radix specifies the base for conversion. For example, base 10 would use decimal digits zero through nine, 0 - 9, and base 16 would use hexadecimal digits zero through nine, 0 - 9, uppercase letters "A" through "F", A - F, or lowercase letters "a" through "f", a - f. If radix is zero or is not supplied, then the radix is automatically determined based on the first characters of str.

If the parameter endStr is supplied, then endStr is set to a string beginning at the first character that was not used in converting. The return is the first part of str, converted to a floating-point number.

SEE: Clib.strtod()

EXAMPLE:

```
// As examples, the following strings//
/ can be converted.
//  "1"
//  "12"
//  "-400"
//  "0xFACE"
```

Clib.strupr()

SYNTAX: Clib.strlwr(str)

WHERE: str - string in which to change case of characters to uppercase.

RETURN:	string - the value of str after conversion of case.
DESCRIPTION:	This method converts all lowercase letters in str to uppercase, starting at str[0] and ending before the terminating null byte. The return is the value of str, that is, a variable pointing to the start of str at str[0].
SEE:	Clib.strlwr(), String toUpperCase()

Clib.toascii()

SYNTAX:	Clib.toascii(chr)
WHERE:	chr - character to be converted.
RETURN:	
DESCRIPTION:	This method translates chr to ASCII format, to seven bits. The translation is done by clearing all but the lowest 7 bits. The return is chr converted to ASCII. Remember that JavaScript has no true character type, thus, this method considers a single character string to be a chr.
SEE:	
EXAMPLE:	

Clib.tolower()

SYNTAX:	Clib.tolower(chr)
WHERE:	chr - character to be converted.
RETURN:	
DESCRIPTION:	If chr is an uppercase alphabetic character, then this method returns chr converted to lowercase alphabetic, otherwise it returns chr unaltered. Remember that JavaScript has no true character type, thus, this method considers a single character string to be a chr.
SEE:	
EXAMPLE:	

Clib.toupper()

SYNTAX:	Clib.toupper(chr)
WHERE:	chr - character to be converted.
RETURN:	
DESCRIPTION:	If chr is a lowercase alphabetic character, then this method returns chr converted to uppercase alphabetic, otherwise it returns chr unaltered. Remember that JavaScript has no true character type, thus, this method considers a single character string to be a chr.
SEE:	
EXAMPLE:	

Clib.vsprintf()

SYNTAX:	Clib.vsprintf(str, formatString, valist)
WHERE:	str - to hold the formatted output. formatString - string that specifies the final format. valist - a variable list of arguments to be used according to formatString.
RETURN:	number - characters written to str, not including the terminating `null` character, on success, else `EOF` on error.
DESCRIPTION:	This method puts formatted output into str, a string, using a variable number of arguments, specified by valist. The parameter formatString specifies the format of the data put into the string. This method is similar to `Clib.sprintf()` except that it takes a variable argu ment list. The method returns the number of characters written to buffer, not including the terminating null byte, on success, else EOF on error.
SEE:	Clib.sprintf(), Clib.va_start()

Memory manipulation

Clib.memchr()

SYNTAX:	Clib.memchr(buf, chr[, maxLen])
WHERE:	buf - buffer or byte array to search.
	chr - character to search for.
	maxLen - maximum number of bytes to search.
RETURN:	buffer - beginning in array with the character found, else `null` if not found.
DESCRIPTION:	This method searches a buffer, a byte array, or a Blob, and returns a variable indicating or beginning with the first occurrence of chr. If the parameter maxLen is not specified, the method searches the entire array from element zero.
SEE:	Clib.strchr()

Clib.memcmp()

SYNTAX: Clib.memcmp(buf1, buf2[, maxLen])

WHERE: buf1 - first buffer or byte array to use in comparison.

buf2 - second buffer or byte array to use in comparison.

maxLen - maximum number of characters to compare.

RETURN: number - negative, zero, or positive according to the following rules:

- **`< 0`** if str1 is less than str2
- **`= 0`** if str1 is the same as str2
- **`> 0`** if str1 is greater than str2

DESCRIPTION: This method compares the first maxLen bytes of buf1 and buf2. If the parameter maxLen is not specified, then maxLen is the smaller of the lengths of buf1 and buf2. If maxLen is specified and one of the arrays is shorter than the specified length, then ScriptEase treats length of the shorter array as being maxLen.

The example function checks to see if the shorter string is the same as the beginning of the longer string. This method differs from `Clib.strcmp()` in that this function returns true if passed the strings "foo" and "foobar", since it only compares characters up to the end of the shorter string.

SEE: Clib.strcmp()

EXAMPLE:

```
function MyStrCmp(string1, string2)
{
   var len = Clib.min(string1.length,
                      string2.length);
   return(Clib.memcmp(string1, string2, len) == 0);
}
```

Clib.memcpy()

SYNTAX: Clib.memcpy(dstBuf, srcBuf[, maxLen])

WHERE: dstBuf - destination buffer to which the source buffer will be copied.

srcBuf - source buffer to copy to destination buffer.

maxLen - maximum number of characters to copy.

RETURN: buffer - the final destination buffer.

DESCRIPTION: This method copies the number of bytes specified by maxLen from srcBuf to dstBuf. If dstBuf is not already defined, then it is defined as a buffer. If the parameter maxLen is not supplied, then all of the bytes in srcBuf are copied to dstBuf.

ScriptEase insures protection from data overwrite, so in ScriptEase the `Clib.memcpy()` method is the same as `Clib.memmove()`.

SEE: Clib.strncpy(), Clib.memmove()

Clib.memmove()

SYNTAX: Clib.memmove(dstBuf, srcBuf[, maxLen])

WHERE: dstBuf - destination buffer to which the source buffer will be copied.

	srcBuf - source buffer to copy to destination buffer.
	maxLen - maximum number of characters to copy.
RETURN:	buffer - the final destination buffer.
DESCRIPTION:	This method copies the number of bytes specified by maxLen from srcBuf to dstBuf. If dstBuf is not already defined, then it is defined as a buffer. If the parameter maxLen is not supplied, then all of the bytes in srcBuf are copied to dstBuf.
	ScriptEase insures protection from data overwrite, so in ScriptEase the `Clib.memcpy()` method is the same as `Clib.memmove()`.
SEE:	Clib.strncpy(), Clib.memcpy()

Clib.memset()

SYNTAX:	Clib.memset(buf, chr[, maxLen])
WHERE:	buf - a byte array or buffer.
	chr - character to set in buf.
	maxLen - number of bytes in buf to set to chr.
RETURN:	buffer - buf with the appropriate number of bytes set to chr.
DESCRIPTION:	This method sets the first number, as specified by maxLen, of bytes of buf to character chr. If buf is not already defined, then it is defined as a buffer of size maxLen. If the length of buf is less than the number of bytes specified by maxLen, then buf is grown to be big enough for maxLen bytes. If the parameter maxLen is not supplied, then maxLen is the size of buf, starting at index 0.
SEE:	Clib.memchr()

Math

Clib.abs()

SYNTAX:	Clib.abs(x)

WHERE:	x - number to work with.
RETURN:	number - absolute value of x.
DESCRIPTION:	This method returns the absolute, non-negative, value of x.
SEE:	Clib.labs(), Clib.fabs()

Clib.acos()

SYNTAX:	Clib.acos(x)
WHERE:	x - number to work with.
RETURN:	number - arc cosine of x.
DESCRIPTION:	This method returns the arc cosine of x in the range of 0 to pi radians.
SEE:	Clib.cos()

Clib.asin()

SYNTAX:	Clib.asin(x)
WHERE:	x - number to work with.
RETURN:	number - arc sine of x.
DESCRIPTION:	This method returns the arc sine of x in the range of -pi/2 to pi/2 radians.
SEE:	Clib.sin()

Clib.atan()

SYNTAX:	Clib.atan(x)
WHERE:	x - number to work with.
RETURN:	number - arc tangent of x.
DESCRIPTION:	This method returns the arc tangent of x in the range of -pi/2 to pi/2 radians.

SEE:	Clib.tan()

Clib.atan2()

SYNTAX:	Clib.atan2(x, y)
WHERE:	x - number to work with, numerator.
	y - number to work with, denominator.
RETURN:	number - arc tangent of x/y.
DESCRIPTION:	This method returns the arc tangent of x/y, in the range of -pi to +pi radians.
SEE:	Clib.atan()

Clib.atof()

SYNTAX:	Clib.atof(str)
WHERE:	str - string to convert to a number.
RETURN:	number - str converted.
DESCRIPTION:	This method converts the ASCII string str to a floating-point value, if str can be converted.
SEE:	Clbib.atol()

Clib.atoi()

SYNTAX:	Clib.atoi(str)
WHERE:	str - string to convert to a number.
RETURN:	number - str converted.
DESCRIPTION:	This method converts the ASCII string str to an integer, if str can be converted.
SEE:	Clib.atol()

Clib.atol()

SYNTAX:	Clib.atol(str)
WHERE:	str - string to convert to a number.
RETURN:	number - str converted.
DESCRIPTION:	This method converts the ASCII string str to a long integer, if str can be converted. This method is the same as the `Clib.atoi()` method, since longs and integers are the same in ScriptEase.
SEE:	Clib.atoi()

Clib.ceil()

SYNTAX:	Clib.ceil(x)
WHERE:	x - number to work with.
RETURN:	number - smallest integer greater than x.
DESCRIPTION:	This method returns the smallest integer value not less than x.
SEE:	Clib.floor()

Clib.cos()

SYNTAX:	Clib.cos(x)
WHERE:	x - number to work with.
RETURN:	number - cosine of x.
DESCRIPTION:	This method returns the cosine of x in radians.
SEE:	Clib.acos(), Clib.cosh()

Clib.cosh()

SYNTAX:	Clib.cosh(x)
WHERE:	x - number to work with.

RETURN:	number - hyperbolic cosine of x.
DESCRIPTION:	This method returns the hyperbolic cosine of x.
SEE:	Clib.cos()

Clib.div()

SYNTAX: Clib.div(x, y)

WHERE: x - number to work with, numerator.

y - number to work with, denominator.

RETURN: object - a structure with the results of division in the following two properties:

```
.quot   quotient
.rem    remainder
```

DESCRIPTION: This method performs integer division and returns a quotient and remainder in an object, a structure. Since integers and long integers are the same in ScriptEase, `Clib.div()` is the same as `Clib.ldiv()`. The value returned is a structure with two elements or properties.

SEE: Clib.ldiv()

Clib.exp()

SYNTAX: Clib.exp(x)

WHERE: x - number to work with.

RETURN: x - exponential value of x.

DESCRIPTION: This method returns the exponential value of x.

SEE: Clib.frexp(), Clib.ldexp(), Clib.pow()

Clib.fabs()

SYNTAX: Clib.fabs(x)

WHERE:	x - number to work with.
RETURN:	number - absolute value of x, a float.
DESCRIPTION:	This method returns the absolute, non-negative, value of a float x.
SEE:	Clib.abs()

Clib.floor()

SYNTAX:	Clib.floor(x)
WHERE:	x - number to work with.
RETURN:	number - largest integer not greater than x.
DESCRIPTION:	This method returns the largest integer value not greater than x.
SEE:	Clib.ceil()

Clib.fmod()

SYNTAX:	Clib.fmod(x, y)
WHERE:	x - number to work with, numerator.
	y - number to work with, denominator.
RETURN:	This method returns the remainder of x/y.
DESCRIPTION:	This method returns the remainder of x/y, that is, the modulus of two floats..
SEE:	Clib.modf(), Clib.div()
EXAMPLE:	

Clib.frexp()

SYNTAX:	Clib.frexp(x, exp)
WHERE:	x - number to work with.
	exp - exponent used with a mantissa.

RETURN:	number - mantissa with and absolute value between 0.5 and 1.0. If x is 0, return 0.
DESCRIPTION:	This method breaks x into a normalized mantissa between 0.5 and 1.0 and calculates an integer exponent of 2 such that `x == mantissa * 2 ^ exponent`. The return is normalized mantissa between 0.5 and 1.0, or 0. The exponent used is in x.
SEE:	Clib.exp(), Clib.ldexp(), Clib.pow()

Clib.labs()

SYNTAX:	Clib.labs(x)
WHERE:	x - number to work with.
RETURN:	number - absolute value of a long integer.
DESCRIPTION:	This method returns the absolute, non-negative, value of an integer.
	Since integers and long integers are the same in ScriptEase, `Clib.labs()` is the same as `Clib.abs()`.
SEE:	Clib.abs(), Clib.fabs()

Clib.ldexp()

SYNTAX:	Clib.ldexp(man, exp)
WHERE:	man - mantissa to work with
	exp - exponent used with a mantissa.
RETURN:	number - mantissa * 2 ^ exp.
DESCRIPTION:	This method is the inverse of `Clib.frexp()` and calculates a floating point number using the following equation:
	mantissa * 2 raised to the power of exp.
SEE:	Clib.frexp(), Clib.exp()

Clib.ldiv()

SYNTAX:	Clib.ldiv(x, y)
WHERE:	x - number to work with, numerator.
	y - number to work with, denominator.

RETURN:	object - a structure with the results of division in the following two properties:
	`.quot   quotient`
	`.rem     remainder`

DESCRIPTION:	This method performs integer division and returns a quotient and remainder in an object, a structure. Since integers and long integers are the same in ScriptEase, `Clib.div()` is the same as `Clib.ldiv()`. The value returned is a structure with two elements or properties.
SEE:	Clib.div()

Clib.log()

SYNTAX:	Clib.log(x)
WHERE:	x - number to work with.
RETURN:	number - natural logarithm of x.
DESCRIPTION:	This method returns the natural logarithm of x.
SEE:	Clib.exp(), Clib.log10(), Clib.pow()

Clib.log10()

SYNTAX:	Clib.log10(x)
WHERE:	x - number to work with.
RETURN:	number - base ten logarithm of x.
DESCRIPTION:	This method returns the base ten logarithm of x.
SEE:	Clib.log()

Clib.max()

SYNTAX:	Clib.max(x[, ...])
WHERE:	x - number or list of numbers to work with.
RETURN:	number - maximum number passed.
DESCRIPTION:	This method is similar to the standard C macro, max(), with the differences that only one variable must be supplied and any number of other variables may be supplied for the comparison.
SEE:	Clib.min()

Clib.min()

SYNTAX:	Clib.min(x[, ...])
WHERE:	x - number or list of numbers to work with.
RETURN:	number - minimum number passed.
DESCRIPTION:	This method is similar to the standard C macro, min(), with the differences that only one variable must be supplied and any number of other vari ables may be supplied for comparison.
SEE:	Clib.max()

Clib.modf()

SYNTAX:	Clib.modf(x, i)
WHERE:	x - float to work with.
	i - variable to receive the integral part of x.
RETURN:	number - signed fractional part of x.
DESCRIPTION:	This method splits a floating point number x into integer and fractional parts, where the integer and frac tion both have the same sign as x. The method sets the parameter i to the integer part of x and returns the fractional part of x.
SEE:	Clib.fmod(), Clib.ldiv()

Clib.pow()

SYNTAX:	Clib.pow(x, exp)
WHERE:	x - number to raise to a power.
	exp - exponent of x, power to which to raise x.
RETURN:	number - x ^ exp.
DESCRIPTION:	This method returns x to the power of y.
SEE:	Clib.exp()

Clib.rand()

SYNTAX:	Clib.rand()
RETURN:	number - random number between 0 and `RAND_MAX`, inclusive.
DESCRIPTION:	This method returns pseudo-random number between 0 and `RAND_MAX`, inclusive. The sequence of pseudo-random numbers is affected by the initial generator seed and by earlier calls to `Clib.rand()`. See `Clib.srand()` for information about the initial generator seed.
SEE:	Clib.srand(), RAND_MAX

Clib.sin()

SYNTAX:	Clib.sin(x)
WHERE:	x - number to work with.
RETURN:	number - sine of x.
DESCRIPTION:	This method returns the sine of x in radians.
SEE:	Clib.asin(), Clib.sinh()

Clib.sinh()

SYNTAX:	Clib.sinh(x)
WHERE:	x - number to work with.
RETURN:	number - hyperbolic sine of x.
DESCRIPTION:	This method returns the hyperbolic sine of the float x.
SEE:	Clib.sin()

Clib.sqrt()

SYNTAX:	Clib.sqrt(x)
WHERE:	x - number to work with.
RETURN:	number - square root of x.
DESCRIPTION:	This method returns the square root of x.
SEE:	Clib.exp(), Clib.pow()

Clib.srand()

SYNTAX:	Clib.srand(seed)
WHERE:	seed - number with which to seed a random number generator.
RETURN:	void.
DESCRIPTION:	This method initializes a random number generator using the parameter seed. If seed is not supplied, then a random seed is generated in an a manner that is specific to different operating systems. Use this method first when generating a sequence of random numbers.
SEE:	Clib.rand()

Clib.tan()

SYNTAX:	Clib.tan(x)
WHERE:	x - number to work with.
RETURN:	number - tangent of x.

DESCRIPTION:	This method returns the tangent of x in radians.
SEE:	Clib.atan(), Clib.tanh()

Clib.tanh()

SYNTAX:	Clib.tanh(x)
WHERE:	x - number to work with.
RETURN:	number - hyperbolic tangent of x.
DESCRIPTION:	This method calculates and returns the hyperbolic tangent of the parameter x, a float.
SEE:	Clib.tan()

Variable argument lists

Clib.va_arg()

SYNTAX:	Clib.va_arg([valist[, offset])
	Clib.va_arg(offset)
	Clib.va_arg()
WHERE:	valist - a variable list of arguments passed to a function.
	offset - index of a particular argument.
RETURN:	value - parameter being retrieved. If no parameters, the number of parameters.
DESCRIPTION:	The method `Clib.va_arg()` provides an alternate way to retrieve a function's parameters. It's most often used when the number of parameters passed to the function is not constant. This method covers the same territory as the `Function.arguments[]` property and is provided for those who prefer C functions for handling variable arguments.
	When called with no parameters, it returns the number of parameters passed to the current function. If an offset is supplied, it returns the input variable at index: offset. `Clib.va_arg(0)` is

the first parameter passed, `Clib.va_arg(1)` the second, etc. It is a fatal error to retrieve an argument offset beyond the number of parameters in the function or the valist.

The valist form, with an optional offset, uses a valist variable that has been previously initialized with `Clib.va_start()`. Each call to `Clib.va_arg(valist)` returns the next parameter passed to a function. If an offset is passed in the variable at that offset from the original starting place of the valist will be returned.

SEE: Clib.va_start(), Clib.va_end(), Clib.vfprintf(), Clib.vfscanf(), Clib.vprintf(), Clib.vscanf(), Clib.vsprintf(), Clib.vsscanf()

EXAMPLE:

```
// The following script:

function main()
{
   lips(0, 1, 2, 3, 4)
}

lips()
{
   Clib.va_start(valist)
   Clib.printf("va_arg(0) = %d\n", va_arg(0));
   Clib.printf("va_arg(1) = %d\n", va_arg(1));
   Clib.printf("va_arg(valist) = %d\n",
               va_arg(valist));
   Clib.printf("va_arg(valist, 2) = %d\n",
               va_arg(valist, 2));
   Clib.printf("va_arg(valist, 2) = %d\n",
               va_arg(valist, 2));
   Clib.printf("va_arg(valist) = %d\n",
               va_arg(valist));
   Clib.getch()
}

// produces the following output:
//  va_arg(0) = 0
//  va_arg(1) = 1
//  va_arg(valist) = 0
//  va_arg(valist, 2) = 3
//  va_arg(valist, 2) = 3
//  va_arg(valist) = 1
```

Clib.va_end()

SYNTAX: Clib.va_end(valist)

WHERE: valist - a variable list of arguments passed to a function.

RETURN: void.

DESCRIPTION: Terminates a variable arguments list. This method makes valist invalid. Many implementations of C require the calling of this function. ScriptEase does not. But, since people may expect it, ScriptEase provides it.

SEE: Clib.va_arg(), Clib.va_start(), Clib.vfprintf(), Clib.vfscanf(), Clib.vprintf(), Clib.vscanf(), Clib.vsprintf(), Clib.vsscanf()

Clib.va_start()

SYNTAX: Clib.va_start(valist[, inputVar])

WHERE: valist - a variable list of arguments passed to a function.

RETURN: number - calls to Clib.va_arg(), that is, the number of variables in valist.

inputVar -

DESCRIPTION: This method initializes valist for a function with a variable number of arguments. After the first call to this function, subsequent calls to Clib.va_arg() may be used to get the rest of the parameters in sequence.

The parameter inputVar must be one of the parameters defined on the function line of a function. The first argument returned by the first call to `Clib.va_arg()` will be the variable passed after inputVar. If inputVar is not provided, then the first parameter passed to a function will be the first one returned by `Clib.va_arg(valist)`.

SEE: Clib.va_end(), Clib.va_start(), Clib.vfprintf(), Clib.vfscanf(), Clib.vprintf(), Clib.vscanf(), Clib.vsprintf(), Clib.vsscanf()

EXAMPLE:

```
// The following example uses and accepts
// a variable number of strings and
// concatenates them all together.

function MultiStrcat(Result, InitialString);
```

```
   // Append any number of strings to InitialString.
   // e.g., MultiStrcat(Result,
   // "C:\\","FOO",".","CMD")
{
   Clib.strcpy(Result,""); // initialize result;
   var Count = Clib.va_start(ArgList, InitialString);
   for (var i = 0; i < Count; i++)
      Result, va_arg(ArgList));
}
```

Clib.vfprintf()

SYNTAX:	Clib.vfprintf(filePointer, formatString[, valist])
WHERE:	filePointer - pointer to file to use.
	formatString - string that specifies the final format.
	valist - a variable list of arguments to be formatted according to formatString..
RETURN:	number - characters written, else a negative number on error.
DESCRIPTION:	This method formats a string with a variable number of arguments and prints it to the file specified by filePointer. It returns the number of characters written, or a negative number if there was an output error.
SEE:	Clib.fprintf(), Clib.sprintf()

Clib.vfscanf()

SYNTAX:	Clib.vfscanf(filePointer, formatString[, valist])
WHERE:	filePointer - pointer to file to use.
	formatString - string that specifies the final format.
	valist - a variable list of variables to hold data input according to formatString.
RETURN:	number - input fields successfully scanned, converted, and stored, else `EOF`.
DESCRIPTION:	This method is similar to `Clib.fscanf()` except that it takes a

	variable argument list. See `Clib.fscanf()` for more details.
SEE:	Clib.va_arg(), Clib.fscanf()
EXAMPLE:	

Clib.vsscanf()

SYNTAX:	Clib.vsscanf(str, formatString, valist)
WHERE:	str - string holding the data to read into variables according to formatString.
	formatString - specifies how to read and store data in variables.
	valist - a variable list of variables to hold data according to formatString.
RETURN:	number - input fields successfully scanned, converted, and stored, else `EOF`.
DESCRIPTION:	This method is similar to `Clib.sscanf()` except that it takes a variable argument list. The parameters following the format string will be assigned values according to the specifications of the format string.
	The function returns the number of input items assigned. This number may be fewer than the number of parameters requested if there was a matching failure.
SEE:	Clib.va_arg(), Clib.sscanf()

Unix Object

```
platform: Unix OS, all versions of SE
```

Unix object static methods

Unix.fork()

SYNTAX:	Unix.fork()
RETURN:	number - 0 or a child process id. 0 is returned to the child process, the id of the child process is returned to the parent.
DESCRIPTION:	A call to this function creates two duplicate processes. The processes are exact copies of the currently running process, so both pick up execution from the next statement. Because these processes are duplicates, they share identical all resources the original one had at the time of fork()ing, but not any allocated later. For instance, any open file handles or sockets are shared. If both processes write to them, the output will be intermixed since each write from either process advances the file pointer for both. `Unix.wait()` allows you to wait for completion of a Child. Using `Unix.wait()` or `Unix.waitpid()` is important to prevent annoying zombie processes from building up.
SEE:	Unix.kill(), Unix.wait(), Unix.waitpid()

EXAMPLE:

```
// Here is a simple example:

function main()
{
   var id = Unix.fork();

   if( id==0 )
   {
      Clib.printf("Child here!\n");
      Clib.exit(0);
   }
   else
   {
      Clib.printf("started child process %d\n", id);
   }
}
```

Unix.kill()

SYNTAX:	Unix.kill(pid, signal)
WHERE:	pid - process to kill.
	signal - the signal to send the process.
RETURN:	number - 0 for success, -1 for error.
DESCRIPTION:	This is simply a direct wrapper for the Unix kill command. To get documentation on it for your particular Unix system, just type 'man 2 kill'
SEE:	Unix.fork()

EXAMPLE:

```
// Typically you would use this to kill a child,
// for instance:

if( var id = Unix.fork() )
{
   while(1)
      Clib.printf("I am an annoying child.\n");
}
else
{
   /* child would be too annoying, so kill it */
   Unix.kill(id,9);       //9 is SIGKILL
   Unix.wait(var status); //wait until child is dead
   Clib.printf(
      "I hope DSS doesn't here about this...\n");
}
```

Unix.setgid()

SYNTAX:	Unix.setgid(id)
WHERE:	id - group id to set.
RETURN:	number - 0 for success, -1 for error.
DESCRIPTION:	Changes the group ID to the given ID, if allowed. I used it in the mini web-server to make sure not running as root (it changes to nobody.)
SEE:	Unix.setuid()

Unix.setsid()

SYNTAX:	Unix.setsid()
RETURN:	number - 0 for success, -1 for error.
DESCRIPTION:	Creates a new session with no terminal, must useful for having commands that when run immediately have the terminal prompt reappear, but continue to run in the background.
SEE:	Unix.fork()

EXAMPLE:

```
// A typical daemon program has a line like this:

#if defined(_UNIX_)
   Unix.setsid(); if( Unix.fork() ) Clib.exit(0);
#endif

// which detaches the program from the terminal and
// continues. Notice, this for line means that
// only the child is running. Because the parent
// has exited and the child does not have the
// original file handles, the shell thinks
// the program is done and goes back to the prompt.
```

Unix.setuid()

SYNTAX:	Unix.setuid(id)
WHERE:	id - user id to set.
RETURN:	number - 0 for success, -1 for error.
DESCRIPTION:	Changes the user ID to the given ID, if allowed. I used it in the mini web-server to make sure not running as root (it changes to nobody.)
SEE:	Unix.setgid()

Unix.wait()

SYNTAX:	Unix.wait(status)
WHERE:	status - status of the process.

RETURN: number - process id of the exiting child, else -1 for error.

DESCRIPTION: A call to wait() will suspend execution until a child process terminates, then return the id of the particular child that exited. The status parameter is a filled in with the status code for the process (this is the raw data exactly as returned by the underlying C wait() call provided for Unix gurus who find this information useful.) Any resources used by the Child are cleaned up.

SEE: Unix.kill(), Unix.waitpid()

EXAMPLE:

```
// Here is a simple example:

function main()
{
   var id = Unix.fork();

   if( id==0 )
   {
      Clib.printf("Child here!\n");
      Clib.exit(0);
   }
   else
   {
      Clib.printf("started child process %d\n", id);
      Clib.assert( Unix.wait(var dontcare)==id );
      Clib.printf("child process is dead meat.\n");
   }
}
```

Unix.waitpid()

SYNTAX: Unix.waitpid(pid, status, flags)

WHERE: pid - child process interested in or -1 for any.

status - status of the process.

flags - WNOHANG or 0.

RETURN: number - process id of the exiting child, else -1 for error.

DESCRIPTION: Very similar to `Unix.wait()`, except you can specify which child process you care about as well as some flags. The only flag currently given a name is WNOHANG, which means that if no child is ready to exit, the call returns immediately. Unix gurus who need the full functionality can put the other possible flag values here.

SEE: Unix.kill(), Unix.waitpid()

EXAMPLE:

```
// This function is most useful in the main loop
// of a server daemon
// (see inn.jse, unix/daemon.jse samples.)
// By calling it each time through the loop such as:

Unix.waitpid(-1,var status, WNOHANG);

// Child processes will get cleaned up and
// zombie processes will not stick around
// wasting resources.
```

Boolean Object

Boolean object instance methods

Boolean()

SYNTAX: new Boolean(value)

WHERE: value - a value to be converted to a boolean.

RETURN: object - a Boolean object with the parameter value converted to a boolean value.

DESCRIPTION: This function creates a boolean object that has the parameter value converted to a boolean value. If the function is called without the `new` constructor, then the return is simply the parameter value converted to a boolean.

SEE: Boolean.toString()

EXAMPLE:

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

Boolean.toString()

SYNTAX: **boolean.toString()**

RETURN: string - "true" or "false" according to the value of the Boolean object.

DESCRIPTION: This `toString()` returns a string corresponding to the value of a Boolean object or primitive data type.

SEE: Boolean.toString(), boolean data type

EXAMPLE:

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

Date Object

ScriptEase shines in its ability to work with dates and provides two different systems for working with them. One is the standard Date object of JavaScript and 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.

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

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

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

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

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

The second syntax accepts a string representing a date and optional time. The format of such a datestring is:

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

For example, the following string:

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

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

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

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

For example, the following line of code:

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

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

ScriptEase has a rich and full set of methods to work with dates and times. A programmer has a very complete set of tools to use when including date and time routines in a script. The Clib object also has methods for working with date and times that extend the power of ScriptEase beyond standard JavaScript.

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

Date object instance methods

Date getDate()

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

Date getDay()

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

Date getFullYear()

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

Date getHours()

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

Date getMilliseconds()

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

Date getMinutes()

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

Date getMonth()

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

Date getSeconds()

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

Date getTime()

SYNTAX:	date.getTime()
RETURN:	number - the milliseconds representation of a Date object.
DESCRIPTION:	Gets time information in the form of an integer representing the

number of seconds from midnight on January 1, 1970, GMT, to the date and time specified by a Date object.

Date getTimezoneOffset()

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

Date getUTCDate()

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

Date getUTCDay()

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

Date getUTCFullYear()

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

Date getUTCHours()

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

Date getUTCMilliseconds()

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

Date getUTCMinutes()

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

Date getUTCMonth()

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

Date getUTCSeconds()

SYNTAX:	date.getUTCSeconds()

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

Date getYear()

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

Date setDate()

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

Date setFullYear()

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

The parameter day is the same as for `setDate()`.

Date setHours()

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

Date setMilliseconds()

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

Date setMinutes()

SYNTAX:	date.setMinutes(minute[, second[, millisecond]])
WHERE:	minute - a minute in an hour.

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

Date setMonth()

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

Date setSeconds()

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

`setMilliseconds()`.

Date setTime()

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

Date setUTCDate()

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

Date setUTCFullYear()

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

The parameter day is the same as for `setUTCDate()`.

Date setUTCHours()

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

Date setUTCMilliseconds()

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

Date setUTCMinutes()

SYNTAX:	date.setUTCMinutes(minute[, second[, millisecond]])

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

Date setUTCMonth()

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

Date setUTCSeconds()

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

is 0, and the last is 59.

The parameter milliseconds is the same as for `setUTCMilliseconds()`.

Date setYear()

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

Date toGMTString()

SYNTAX: date.toGMTString()

RETURN: string - string representation of the GMT date and time.

DESCRIPTION: This method converts a Date object to a string, based on Greenwich Mean Time.

EXAMPLE:

```
var d = new Date();
Screen.writeln(d.toGMTString());

// The fragment above would produce something like:
// Mon May 1 15:48:38 2000 GMT
```

Date toDateString()

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

SEE: Date.toString(), Date.toTimeString(), Date.toLocaleDateString()

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

Date toLocaleDateString()

SYNTAX: date.toLocaleDateString()

RETURN: string - locale-sensitive string representation of the date portion of the current date.

DESCRIPTION: This function behaves in exactly the same manner as Date.toDateString(). This function is designed to take in the current locale when formatting the string. Locale reflects the time zone of a user.

SEE: Date.toString(), Date.toLocaleTimeString(), Date.toLocaleString()

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

Date toLocaleString()

SYNTAX: date.toLocaleString()

RETURN: string - locale-sensitive string representation of the current date.

DESCRIPTION: This function behaves in exactly the same manner as Date.toString(). This function is designed to take in the current locale when formatting the string, though this functionality is currently unimplemented. Locale reflects the time zone of a user.

SEE: Date.toString(), Date.toLocaleTimeString(), Date.toLocaleDateString()

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

Date toLocaleTimeString()

SYNTAX: date.toLocaleTimeString()

RETURN: string - locale-sensitive string representation of the time portion of the current date.

DESCRIPTION: This function behaves in exactly the same manner as Date.toTimeString(). This function is designed to take in the current locale when formatting the string. Locale reflects the time zone of a user.

Date toString()

SYNTAX: date.toString()

RETURN: string - representation of the date and time data in a Date object.

DESCRIPTION: Converts the date and time information in a Date object to a string in a form such as: "Mon May 1 09:24:38 2000"

SEE: Date.toDateString(), Date.toLocaleString(), Date.toTimeString()

EXAMPLE:

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

Date toSystem()

SYNTAX: date.toSystem()

RETURN: number - the Date object date and time value converted to the system date and time.

DESCRIPTION: This method converts a Date object to a system time format which is the same as that returned by the Clib.time() method. To create a Date object from a variable in system time format, see the Date.fromSystem() method.

Date toTimeString()

SYNTAX: date.toTimeString()

RETURN: string - representation of the Time portion of the current object.

DESCRIPTION: This function returns the time portion of the current date as a string. This string is formatted to read "Hours:Minutes:Seconds", as in "16:43:23". This function uses the local time, rather than

	the UTC time.
SEE:	Date.toString(), Date.toDateString(), Date.toLocaleDateString()

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

Date toUTCString()

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

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

Date valueOf()

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

Date object static methods

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

Date.fromSystem()

SYNTAX:	Date.fromSystem(time)
WHERE:	time - time in system data format, the same format as returned by

```
Clib.time()
```

RETURN: object - Date object with the time passed.

DESCRIPTION: This method converts the parameter time, which is in the same format as returned by the <code>Clib.time()</code>, to a standard JavaScript Date object.

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);

// To convert a Date object to system format
// that can be used by
// the methods of the Clib object,
// use code similar to:

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

Date.parse()

SYNTAX: Date.parse(datestring)

WHERE: datestring - A string representing the date and time to be passed

RETURN: number - milliseconds between the `Datestring` and midnight , January 1, 1970 GMT.

DESCRIPTION: This method converts the string datestring to a Date object. The string must be in the following format: `Friday, October 31, 1998 15:30:00 -0500` This format is used by the .toGMTString() method and by email and Internet applications. The day of the week, time zone, time specification or seconds field may be omitted.

SEE: Date(), Date.setTime(), Date.toGMTString(), Date.UTC

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

Date.UTC()

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

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

Link Libraries

Link libraries are dynamic link library files (.dll files) developed specifically to work with ScriptEase. ScriptEase can use any DLL, but the calling conventions needed to call routines in a DLL are necessarily more cumbersome than calling internal routines. ScriptEase extends the power and ease of using its link libraries by tying them to the internal data structures of ScriptEase. In this way, the data and routines in a link library are available with the same calling conventions of internal routines. Consider the two following code fragment:

```
   // Using dynamicLink
var v1 = SElib.dynamicLink("YourDll.dll", "FunctionOne", STDCALL,
args ...);
var v2 = SElib.dynamicLink("YourDll.dll", "FunctionTwo", STDCALL,
args ...);

   // Using a link library
#link <SElink.dll>
var v1 = FunctionOne(args ...);
var v2 = FunctionTwo(args ...);
```

As you can see in the first three lines, every time you want to call a routine from a general DLL, you must use the more cumbersome `SElib.dynamicLink()` method and its cumbersome calling conventions. Cumbersome calling conventions exist in any language that allows general DLL files to be called. But, notice the difference after a ScriptEase link DLL is linked into a script, as illustrated by the line, `#link <SElink.dll>`. The routines and data in the DLL are accessible in the same way as internal routines such as `Screen.writeln()`.

Script libraries, scripts that end with `jsh`, can be used to define objects, methods, properties, functions, and data. The advantage of script libraries is that you may develop them quickly and alter them at any time. An advantage of link libraries is that they execute faster since they compiled executables.

The following sections explain various link libraries. To use these link libraries, they must be included in a script with the `#link <>` preprocessor directive. Most link libraries have a corresponding script library, `jsh` file, that simplifies their use even more.

UUCode Link Library

The Unix-To-Unix encoding library provides two functions for encoding and decoding data in a text format.

```
 platform: Mac, OS2, Windows;  All versions of SE
   source: #link <uucode.dll>
```

UU object static methods

UU.encode()

SYNTAX:	UU.encode(infile[, outfile])
WHERE:	infile - Name of input file
	outfile - Name of output file
RETURN:	boolean - Whether or not the operation was successful
DESCRIPTION:	This method uses the Unix-to-Unix encoding mechanism, still popular in newsgroups, as a way of translating binary data into printable text data. If <code>outfile</code> is not supplied, then an appropriate filename is generated by either adding or replacing the extension with ".uue". The file "foo.c" would become "foo.uue". This file later can be decoded with any popular UUdecoding program, or a call to UU.decode();
SEE:	UU.decode()

UU.decode()

SYNTAX:	UU.decode(infile[, outfile])
WHERE:	infile - Name of input file
	outfile - Name of output file
RETURN:	boolean - Whether or not the operation was successful
DESCRIPTION:	This method decodes a file stored using the Unix-to-Unix encoding mechanism. If <code>outfile</code> is not supplied,

then the filename that is stored in the infile (the original name of the file) is used instead.

SEE: UU.encode()

DSP Link Library

Distributed Scripting Protocol is implemented by the ScriptEase DSP link library as the DSP object.

DSP Object

```
platform: All platforms except Dos; All versions of SE
  source: #link <sedsp.dll>
```

The DSP object provides a framework for implementing distributed scripting across a variety of computers and networks.

Creating a DSP object

The Distributed Scripting Protocol provides no internal method for managing a connection or transporting packets. It is simply a framework, with the physical transport method being supplied by the user. As such, it is impossible to simply create a DSP object, because it is incapable of doing anything by itself. The user must supply a set of functions to manage the connection with the server. To create a DSP object, you call `new DSP(myOpenFunction, myParameters)`. The function that you supply must open the connection and return a reference to it. It is possible in some instances that you do not need to open anything special, and so you can ignore this parameter. Here is an example of an open function for a DSP connection, using internet sockets:

```
function idspOpen( host, port )
{
   return new Socket( host, port );
}
```

We will see this function passed to the DSP constructor in a moment. First, to accomplish sending/receiving packets, the user needs to define two functions, `dspSend` and `dspReceive`. These functions must be inherited through the prototype chain, because otherwise when DSP objects are copied implicitly through reference construction (see below), the functions will not get passed. Because we want to keep the DSP functions (such as dspService), we need to preserve the original DSP prototype, and a constructor looks like the following:

```
function iDSP( host, port )
{
   var ret = new DSP( idspOpen, host, port );
   // Now we override the ._prototype to insert our functions
   if( ret != null )
      ret._prototype = iDSP.prototype;
   return ret;
}
// Here we set up the iDSP.prototype to keep the DSP functions
// in the chain
iDSP.prototype._prototype = DSP.prototype;
```

Once this constructor is called, we have a valid DSP object, assuming we add the transport functions. To do this, we must add `dspSend` and `dspReceive` to the prototype. The actual syntax of these functions is similar to `Clib.fread` and `Clib.fwrite`, and a description can be found in the function reference. For our iDSP example, they would look something like this:

```
function iDSP.prototype.dspSend( conn, buffer, timeout )
{  // Ignore timeout
   return conn.write(buffer);
}
function iDSP.prototype.dspReceive( conn, &buffer, length,
timeout)
{
   return conn.read( buffer, length );
}
```

Note that both these functions ignore the timeout parameter and do not correctly handle errors. A full-featured version of these functions can be found in the file *idsp.jsh*. The final function that we must provide is the `dspCloseConnection` function, which is responsible for closing the connection. This function looks like the following:

```
function iDSP.prototype.dspCloseConnection( conn )
{
   conn.close();
}
```

Once all of these transport functions have been defined, new iDSP objects can be instantiated with a call to `new iDSP` and used as any other DSP object. Because the transport level of DSP is separate from the core library, DSP can be adapted to communicate between any servers in any way. In addition, communication can be done during the call to the open function. This allows for password authentication or any other information to be shared.

Using a DSP object

Once a DSP object is created using the method described above, every DSP object behaves in exactly the same way. Once the functions are set up, the transport layer of the protocol is hidden.

The basic idea is that all DSP objects are in fact references to objects on the remote side, and they will remain so except under certain circumstances (described below). When a connection is first established, it is a reference to the global object. Members of the remote global object can be accessed as members of the connection. But they remain references, so `var print = connection.Clib.printf` will not actually make a remote call to the server. At the appropriate time, `print` will be resolved into `Clib.printf` and sent to the server in the appropriate manner. The circumstances which can trigger a de-referencing and remote call are:

Calling functions - When a DSP reference is called as a function, it gets resolved into the appropriate path and the function is called on the remote server. All parameters are converted to source with `ToSource()` and passed to the server, and set back afterwards (in case any were passed by reference). The client waits for the return value from the server and returns that as the result of the function call. This makes calling functions transparent to the client, so `connection.Screen.writeln("hi")` will actually call `Screen.writeln` on the server and print out "hi".

Setting a value - When a value is put to a DSP reference, such as `connection.globalCount = 5`, a remote call to the server is generated, and the remote value is updated. The above case acts just as if `globalCount = 5` was executed on the server.

Implicitly - When a DSP reference is converted to a primitive, then it gets de-referenced. This implicit conversion happens mostly in operator expressions, in which both values are converted to primitives first. So `var myCount = connection.globalCount + 1` will get the value of `globalCount` from the server and add one to it. This can also be accomplished explicitly with `ToPrimitive()`, but the method below is more straightforward and understandable. The explicit use of ToPrimitive() on DSP references is discouraged.

Explicitly - Any DSP reference can be explicitly de-referenced with a call to`.dspGetValue`. Once an object has been de-referenced this way, any subsequent accesses will not cause a remote call, and changes will only affect the local copy. Note that calling a function in this way will result in the function being called on the local client, not the server.

DSP object instance methods

DSP()

SYNTAX:	new DSP([openFunction[, param1[, ...]]])
WHERE:	openFunction - The function to call to initialize the connection.
	paramN - Additional parameters to pass to the open function
RETURN:	object - A new DSP object, or null on error
DESCRIPTION:	This function creates a new DSP object, or returns null on error. Note that calling this function itself accomplishes very little unless you build up an appropriate DSP object by adding open, close, and transport functions. A new DSP object can be created with just `new DSP()`, but it will be unusable without transport functions. See the introduction for more information about setting up a proper DSP object. The first optional parameter is the open function to use. Once the object has been created, this function is called with any additional parameters passed to DSP(). The result of this call is set the `dspConnection` member of the newly created object, and is only used to pass as the first parameters to the `dspSend`, `dspReceive`, and `dspCloseConnection` methods. If `openFunction` is supplied and returns null, then it is considered an error and the DSP construction fails.

EXAMPLE:

```
function fileOpen( filename )
{
   return Clib.fopen( filename, "wb" );
}
var connection = new DSP( fileOpen, "c:\tempfile.dat"
    );
// This will call fileOpen and assign the result to
// connection.dspConnection.  If it was null,
// then the DSP connection will fail
```

DSP dspCloseConnection()

SYNTAX:	dsp.closeConnection(connection)
WHERE:	connection - The original connection that was created with the openFunction passed to `new DSP()`
RETURN:	void.
DESCRIPTION:	This function is responsible for terminating the connection that was opened at the time the DSP object was created. This is an optional function, and if not supplied then nothing will be done with the connection. See the introduction for an example of how to implement this function.
SEE:	DSP()

DSP dspReceive()

SYNTAX:	dsp.dspReceive(connection, buffer, bufferLength, timeout)
WHERE:	connection - The original connection that was returned from the openFunction passed to `new DSP()` buffer - A buffer which is to be filled with data. This variable must be passed by reference (with the & operator). bufferLength - The maximum amount of data to read timeout - The maximum amount of time to wait (in milliseconds) for data to be ready for reading on the connection
RETURN:	number - The number of bytes read, or -1 on error
DESCRIPTION:	This function is responsible for getting data from the connection. This function should wait up to `timeout` milliseconds for data to be available on the connection. If there is no data available, then this function should return 0. Otherwise, the function should read up to `bufferLength` bytes from the connection and put the data into `buffer`. Note that this means that `buffer` must be passed by reference. If there is some sort of error, then this function should either throw an error, or return -1. See introduction for an example of how to implement this function. Note that the function need not wait for the entire buffer to be filled, it should read only as much data as is available to be read.

SEE:	dspSend()

DSP dspSend()

SYNTAX:	dsp.dspSend(connection, buffer, timeout)
WHERE:	connection - The original connection that was returned from the openFunction passed to `new DSP()` buffer - The buffer to send timeout - The maximum amount of time to wait (in milliseconds) for data to be ready for writing on the connection
RETURN:	number - The number of bytes written, or -1 on error
DESCRIPTION:	This function is responsible for sending data across the connection (the one returned by the openFunction passed to the DSP constructor). It's behavior is similar to that of dspReceive(). It should wait up until `timeout` for data to be ready, and then send as much as possible along the connection (up to the length of `buffer`). If the timeout expires, the function should return 0. If there was some sort of error, then an error should be thrown, or -1 returned. Otherwise, the number of bytes written should be returned. Throwing an error is often more descriptive than the generic failure message. See introduction for an example of how to implement this function.
SEE:	dspReceive()

DSP dspLoad()

SYNTAX:	dsp.dspLoad(code)
WHERE:	code - String of code to load on the remote server
RETURN:	void.
DESCRIPTION:	This function loads the specified code into the global context on the remote server. Any code that you execute will remain on the remote server. This function is designed to load functions on the remote server so that they may be called by the client. This

function does not wait for a return value from the host. As a consequence, remote errors will not be immediately reported. They will be reported next time a client routine (calling a function, getting/putting a value) queries the server. Note that if you wish to execute remote code and get a return value, the global eval() method for the server should be used, although the changes will not be permanent.

EXAMPLE:

```
function foo() { Screen.writeln("Hello!"); }
// This code will make "foo = new Function(...)"
// to set up the function on the remote server.
connection.dspLoad( "foo = " + ToSource(foo) );
connection.foo();
// foo is now a global function on the server
```

DSP dspService()

SYNTAX: dsp.dspService()

RETURN: boolean - A value indicating whether the connection is still open.

DESCRIPTION: This is the main server-side function. Although it can be used by any DSP object, it is intended to be the server side of the client-server model. When called, it will wait until an incoming packet is received and then service that packet appropriately. The method will return false if the packet received was a close command, in which case the connection has been closed, and an explicit call to `dspClose` is not necessary. It is designed to be called repeatedly until the connection is closed.

EXAMPLE:

```
// Assume 'connection' is a valid connection
while( connection.dspService() )
   ;
// At this point, the connection has been
// successfully closed
```

DSP dspClose()

SYNTAX: dsp.dspClose()

RETURN: void.

DESCRIPTION: This function closes the DSP connection. First, it sends a close command to the remote host, signaling that the connection is closing. It then calls the `dspCloseConnection` method if it exists, passing the original connection variable returned by the open function when this connection was created.

EXAMPLE:

```
connection.dspClose();
```

DSP dspGetValue()

SYNTAX: dsp.dspGetValue()

RETURN: variable - remote value of the current DSP reference.

DESCRIPTION: This function provides an explicit way to convert a DSP reference into a value. Such conversion is done automatically when the reference is converted to a primitive, or a value is assigned to a reference. See the introductory section for more information on DSP references and getting remote values.

EXAMPLE:

```
var reference = connection.globalValue;
var value = connection.globalValue.dspGetValue();
reference = 5;  // This will change the remote value
value = 6;
// This will change the local copy, not the remote
```

DSP dspSecurityInit()

SYNTAX: dsp.dspSecurityInit(secureVar)

WHERE: secureVar - private storage for the DSP security. The member 'dsp' is preset to the DSP object. Remember, the DSP object can be seen by the running script, but not the secure variable itself.

RETURN: void.

DESCRIPTION: The dspSecurityInit function turns on security for a DSP object. This means when the remote client tries to run a script on your machine using DSP, it will be run with your security manager in effect. See the security document for a complete description of how it works. In the case of DSP, each security function (jseSecurityInit, jseSecurityTerm, and jseSecurityGuard) has an exactly corresponding function, i.e. dspSecurityInit, dspSecurityTerm, and dspSecurityGuard. In the security initialization function, you'll typically select some functions to be allowed, and let all others be vetoed.

SEE: DSP.dspSecurityTerm, DSP.dspSecurityGuard

EXAMPLE:

```
function iDSP.dspSecurityGuard( conn )
{
   myfunc.setSecurity(jseSecureAllow);
   myotherfunc.setSecurity(jseSecureGuard);
}
```

DSP dspSecurityTerm()

SYNTAX: dsp.dspSecurityTerm(secureVar)

WHERE: secureVar - private storage for the DSP security.

RETURN: void.

DESCRIPTION: This function is typically not needed, but you can use it to cleanup anything you initialized in the DSP security initialization function.

SEE: DSP.dspSecurityInit, DSP.dspSecurityGuard

DSP dspSecurityGuard()

SYNTAX: dsp.dspSecurityGuard(secureVar, function, params)

WHERE: function - the function being called

secureVar - private storage for the DSP security.

params - whatever parameters are passed to the function

RETURN: void.

DESCRIPTION: If a DSP object is given a dspSecurityGuard function (exactly like any of the other DSP callback functions), when it tries to call any function not part of the script (i.e. one of your functions or a wrapper function), the security guard is called for approval. See the security document for a description on how this all works. You must provide a dspSecurityInit for security to be activated. Only those functions the security initialization function marks as guarded will use this function.

SEE: dsp.dspSecurityInit, dsp.dspSecurityTerm

DSP object static properties

DSP.remote

SYNTAX: DSP.remote

DESCRIPTION: This global property of the DSP object is used to make calls back to the remote client from within a function. When the first DSP object in a script is created, this gets assigned to that value. From then on, whenever a packet needs to be serviced, this value is set (and later restored) to the object representing the incoming connection. This allows for multiple connections, and lets the function easily call back the appropriate client. Note that within a `dspLoad` call, the client does not wait for a response, and so trying to call on the client will yield no result until the server is queried again.

EXAMPLE:

```
// Assume the client calls this:
serverConn.printRemote("hi");
// And the server side looks like this:
function printRemote( string )
{
   DSP.remote.Screen.write( string );
}
// This will print out "hi" on the client machine
```

GD Link Library

GD Object

```
    title: GD Object
 platform: All OS except Dos; All versions of SE
   source: #link <gd.dll>
```

The GD object provides a set of routines for manipulating GIF images.

Point specifications

A number of GD routines expect a Point Specification as one of the parameters. This is a pseudo-type that can take one of several forms. It is either an object with two members, 'x' and 'y', representing the two coordinates of the point, or an array with two members, element 0 being the x coordinate and element 1 being the y coordinate. All of the following are equivalent:

```
var point1 = {x:1, y:2};
var point2 = [1, 2];
```

Note also that every routine can also have the x and y coordinates passed as separate parameters, so these are equivalent:

```
gd.getPixel( 1, 2 );
gd.getPixel( [1,2] );
```

As such, the Point object is really just a matter of convenience to help distinguish points as a unit.

Font specifications

The character drawing routines expect a font parameter which describes the font to use. The font selection, though limited, should be enough for the basic purposes for which this library is used. Valid font types are the strings *"tiny"*, *"small"*, *"mediumBold"*, *"large"*, or *"giant"*. Each one is a different size. *fontTiny* is 5x8, *fontSmall* is 6x12, *fontMediumBold* is 7x13, *fontLarge* is 8x16, and *fontGiant* is 9x15.

Color styles

In addition to simple color indexes, all drawing routines can also take a color stype, which is a special string value that allows for more complex fills and shapes. The valid types are:

"styled" - Use the style specified with `GD.setStyle()`. A style is a sequence of colors to be used when drawing lines. It is only valid for line-drawing routines, and is used to make dashed lines.

"brushed" - Use the brush specified with `GD.setBrush()`. A brush is another GD image which is drawn instead of a regular pixel. Using transparent colors, it is possible to create a brush of any size.

"styledBrushed" - A combination of both "styled" and "brushed". The brush is used, but is only drawn when non-transparent pixels are encountered in the style.

"tiled" - Use the tile specified with `GD.setTile()`. This style can only be used with fill routines. It uses the current tile, which can be any GD image, and fills the region with that tile, laying the images side-by-side sequentially.

GD object instance methods

GD()

SYNTAX:	new GD(x, y)
WHERE:	x - Horizontal size, in pixels.
	y - Vertical size, in pixels.
RETURN:	object - a new GD object of the specified size.
DESCRIPTION:	The `x` and `y` parameters determine the horizontal and vertical size of the image, respectively. The object returned is a GD object.

GD arc()

SYNTAX:	gd.arc(centerX, centerY, width, height, startDegree, endDegree, color)
	gd.arc(centerPoint, width, height, startDegree, endDegree, color)
WHERE:	centerX - horizontal position of center.
	centerY - vertical position of center.

centerPoint - center point specification.

height - height of arc.

startDegree - degree value of starting position in standard coordinate plane. Values greater than 360 are interpreted as modulo 360.

endDegree - degree value of ending position in standard coordinate plane. Values greater than 360 are interpreted as modulo 360.

color - color index to use for arc, or one of the strings "styled", "brushed", "styledBrushed".

RETURN: void.

DESCRIPTION: This method draws an arc in the specified format. The center position is specified, along with the width and the height. The arc is then draw between the two given degree values. A full ellipse can be drawn from degree 0 to degree 360, and a circle can be drawn in the same manner while setting `width` and `height` to be the same. If there is a out-of-bounds error or some other error, then the arc is not drawn at all.

EXAMPLE:

```
// Draw a circle with a diameter of 16 pixels
// in the middle of the image
var gd = new GD(65, 65);
gd.arc( [32,32], 16, 16, 0, 360, 0 );
```

GD blue()

SYNTAX: gd.blue(index)

WHERE: index - color index to look up.

RETURN: number - blue component of the specified color index.

DESCRIPTION: This method looks up the color indicated by `index` and returns the blue component of that color.

SEE: GD.red(), GD.green()

EXAMPLE:

```
var index = gd.colorAllocate(0,100,200);
gd.blue(index);  // This will return 200
```

GD boundsSafe(

SYNTAX: gd.boundsSafe(x, y)

gd.boundsSafe(point)

WHERE: x - horizontal pixel location.

y - vertical pixel location.

point - Point specification. See GD.getPixel() for a description.

RETURN: boolean - whether the specified coordinates are within bounds.

DESCRIPTION: This method sees if the specified pixel location is within the bounds of the image. If so, then true is returned, false otherwise.

EXAMPLE:

```
var gd = new GD(5,5);
gd.boundsSafe( 4, 3 );     // True
gd.boundsSafe( [4,5] );    // False
gd.boundsSafe( {x:6,y:2} ) // False
```

GD drawChar()

SYNTAX: gd.drawChar(font, x, y, char, color)

gd.drawChar(font, point, char, color)

WHERE: font - Font specification

x - horizontal position of upper-left corner of character

y - vertical position of upper-left corner of character

point - Point specification.

char - The specified character to draw

color - color index or style to use

RETURN: void.

DESCRIPTION: This method draws a character in the image at the specified location in the appropriate font. If the coordinates are out of bounds, then no drawing is done. The reason that it is named 'drawChar' and not simply 'char' is that 'char' is a reserved keyword and an invalid variable name.

SEE: GD.charUp(), GD.string()

EXAMPLE:

```
// Write "hi" at the starting position
```

```
var gd = new GD(50,50);
gd.drawChar( GD.fontSmall, 5, 5, "h", 0 );
gd.drawChar( GD.fontSmall, [11,5], "i", 0 );
// This is the equivalent of GD.string()
// with the string "hi"
```

GD charUp()

SYNTAX:	gd.charUp(font, x, y, char, color)
	gd.charUp(font, point, char, color)
WHERE:	font - Font specification
	x - horizontal position of upper-left corner of character
	y - vertical position of upper-left corner of character
	point - point specification. See GD.getPixel() for a description.
	char - specified character to draw
	color - color index or style to use
RETURN:	void.
DESCRIPTION:	This method is exactly the same as GD.drawChar(), except that the character is drawn vertically, pointing upwards.
SEE:	GD.drawChar(), GD.stringUp()

GD colorAllocate()

SYNTAX:	gd.colorAllocate(red, green, blue)
WHERE:	red - Red value, from 0 to 255
	green - Green value, from 0 to 255
	blue - Blue value, from 0 to 255
RETURN:	number - Color index of allocated color, or -1 if none available.
DESCRIPTION:	This method searches through the color table for the next available color index, and sets it to be the supplied RGB color. If no color indexes are available, then -1 is returned. If the supplied RGB colors are invalid, a runtime error is generated. When creating a new image, the first time you call this function,

you set the background color for the image.

SEE: GD.colorExact(), GD.colorClosest(), GD.colorDeallocate()

EXAMPLE:

```
var gd = new GD(10,10);
var index = gd.colorAllocate(255,255,255);
// index now points to white, and the background
// of the image is also white
```

GD colorClosest()

SYNTAX: gd.colorClosest(red, green, blue)

WHERE: red - Red value, from 0 to 255

green - Green value, from 0 to 255

blue - Blue value, from 0 to 255

RETURN: number - index of the closest color to the one supplied.

DESCRIPTION: This method searches through the color table and finds the closest color to the one supplied. The algorithm uses Euclidian distance to calculate closeness. This function is most useful when unable to allocate a new color, and the closest must be used instead.

SEE: GD.colorAllocate()

EXAMPLE:

```
/* Attempt to allocate a specific color,
 * but if unable to (the image
 * has the maximum number of colors),
 * then attempt to find the closest
 * color as a suitable replacement
 */
var gd = GD.fromGif("test.gif");
var index;
if( -1 == (index = gd.colorAllocate(234,12,107)) )
  index = gd.colorClosest(234,12,107);
```

GD colorDeallocate()

SYNTAX: gd.colorDeallocate(color)

WHERE: color - color index to deallocate.

RETURN: void.

DESCRIPTION: This method frees up the color at index `color` for later use. The color index will remain the same, but it may be re-allocated at any point and changed. Note that this function simply marks the color for reuse, so that the total colors allocated in the image still remains the same. If a call to colorAllocate() immediately follows this call, then the old index will be re-used for the new color, and all pixels within the image with that index will be altered as well.

SEE: GD.colorAllocate()

GD colorExact()

SYNTAX: gd.colorExact(, green, blue)

WHERE: red - Red value, from 0 to 255

green - Green value, from 0 to 255

blue - Blue value, from 0 to 255

RETURN: number - The first index matching the supplied color, or -1 if it doesn't exist.

DESCRIPTION: This method searches through the color table and tries to find the first index whose red, green, and blue values are exactly equal to the supplied values. If no index is found, then -1 is returned.

SEE: GD.colorClosest(), GD.colorAllocate()

EXAMPLE:

```
// Attempt to get the color,
// and create it if it does not exist
var gd = GD.fromGif("test.gif");
var index;
if( -1 == (index = gd.colorExact(1,1,1)) )
   index = gd.colorAllocate(1,1,1);
```

GD colorsTotal()

SYNTAX: gd.colorsTotal()

RETURN: void.

DESCRIPTION: This method returns the total number of colors allocated in the current GD image. Note that colors deallocated with colorDeallocate() are still considered 'allocated', because they have simply been marked for reuse.

GD colorTransparent()

SYNTAX:	gd.colorTransparent(color)
WHERE:	color - color index to make transparent.
RETURN:	void.
DESCRIPTION:	This method sets the specified color index to be the transparent index. To indicate that there is to be no transparent color, the value -1 should be passed as the color index.

EXAMPLE:

```
var gd = new GD(64,64);
var index = gd.colorAllocate(0,0,0);
gd.colorTransparent(index);
// The background (and all black pixels)
// is transparent
```

GD copy()

SYNTAX:	gd.copy(source, dstX, dstY, srcX, srcY, width, height)
	gd.copy(source, dstPoint, srcPoint, width, height)
WHERE:	source - A gd object to copy from
	dstX - Horizontal destination pixel in current object
	dstY - Vertical destination pixel in current object
	dstPoint - Destination pixel in current object.
	srcX - Horizontal source pixel in source object
	srcY - Vertical source pixel in source object
	srcPoint - Source pixel in source object.
	width - Width of section to copy
	height - Height of section to copy
RETURN:	void.
DESCRIPTION:	This method copies a section from one GD image to another. The portion of `source`, starting at the specified point (which is the upper-left corner of the region) and extending `width` and

`height` in either direction. This region is then copied to the current GD object at the specified location (which is again the upper-left corner of the region). In copying the region, this method attempts to preserve the colors of the original source as best as possible. The method first tries calling colorExact() on the current image, and if that doesn't work then colorAllocate(), and finally if that fails, then colorClosest(). If you specify the same source image as the current image, then the method will work appropriately as long as the regions to not overlap. If they do, then the result is undefined.

SEE: GD.copyResized()

EXAMPLE:

```
// Copy top-left 16x16 from "test.gif"
// while attempting to preserve
// necessary colors.
var source = GD.fromGif("test.gif");
var dest = new GD( 16, 16 );
dest.copy( source, [0,0], [0,0], 16, 16 );
```

GD copyResized()

SYNTAX: gd.copyResized(source, dstX, dstY, srcX, srcY, dstW, dstH, srcW, srcH)

gd.copyResized(source, dstPoint, srcPoint, dstW, dstH, srcW, srcH)

WHERE: source - A gd object to copy from

dstX - Horizontal destination pixel in current object

dstY - Vertical destination pixel in current object

dstPoint - Destination pixel in current object.

srcX - Horizontal source pixel in source object

srcY - Vertical source pixel in source object

srcPoint - Source pixel in source object.

dstW - Width of region in current object

dstH - Height of region in current object

srcW - Width of region in source object

	srcH - Height of region in source object
RETURN:	void.
DESCRIPTION:	This method is very similar to GD.copy(), except that it has the additional option of resizing the image in the process of copying. This method will stretch or shrink the region as appropriate in order to fit in the destination area. Specifying the same destination and source sizes is the equivalent of calling GD.copy(). See GD.copy() for more description.
SEE:	GD.copy()

EXAMPLE:

```
// Copy top-left 4x4 square from "test.gif"
// and magnify it four times
// to a size of 16x16 in the destination image
var source = GD.fromGif("test.gif");
var dest = new GD( 16, 16 );
dest.copyResized(source, [0,0], [0,0], 16, 16, 4, 4);
```

GD dashedLine()

SYNTAX:	gd.dashedLine(x1, y1, x2, y2, color)
	gd.dashedLine(point1, point2, color)
WHERE:	x1 - horizontal pixel location of starting point
	y1 - vertical pixel location of starting point
	x2 - horizontal pixel location of ending point
	y2 - vertical pixel location of ending point
	point1 - First point specification.
	point2 - Second point specification.
	color - color index or style to use for drawing line
RETURN:	void.
DESCRIPTION:	This method is exactly the same as GD.line(), except that a dashed line is drawn. This function is only for backwards compatibility, as much greater control is achieved by using the combination of GD.setStyle() and GD.line().
SEE:	GD.line(), GD.setStyle()

EXAMPLE:

```
var gd = new GD(10,10);
gd.dashedLine( [2,3], [9,7], 0 );

// The above code has been replaced by the following
var gd = new GD(10,10);
// Four pixel wide dash
gd.setStyle( [0, 0, 0, 0, -1, -1, -1, -1 );
gd.line( [2,3], [9,6], "styled" );
```

GD destroy()

SYNTAX: gd.destroy()

RETURN: void.

DESCRIPTION: This method cleans up all the memory associated with this GD object. Once it has been called, the object is no longer valid.

GD fill()

SYNTAX: gd.fill(x, y, color)

gd.fill(point, color)

WHERE: x - Horizontal position of starting pixel

y - Vertical position of starting pixel

point - Point of starting pixel. See GD.getPixel() for a description

color - Fill color index or style

RETURN: void.

DESCRIPTION: This method is very similar to GD.fillToBorder(), except that instead of filling until another color is hit, this method fills all pixels that are the same color as the original, until it hits any other color pixel. The pixels are changed to the color indicated by `color`.

SEE: GD.fillToBorder()

EXAMPLE:

```
/* Draw a circle with color index 1 and
 * a smaller one with color
 * index 3.  The call to GD.fill() will fill
 * the inner circle with color
 * index 2.  The fill will stop at the first circle,
 * since it is not the
 * same color as the starting pixel.
 */
var gd = new GD(65, 65);
gd.arc( [32,32], 16, 16, 0, 360, 1 );
gd.arc( [32,32], 14, 14, 0, 360, 3 );  // will be
erased
gd.fill( [33,34], 2 );
```

GD filledPolygon()

SYNTAX:	gd.filledPolygon(point1[, x2, y2[, ...], color)
WHERE:	pointN - Point specification for Nth point
	xN - x coordinate of Nth point
	yN - y coordinate of Nth point
	color - color index or style to use for fill
RETURN:	void.
DESCRIPTION:	This method is exactly the same as GD.polygon(), except that it fills in the polygon, managing intersections in the process.
SEE:	GD.polygon()

GD filledRectangle()

SYNTAX:	gd.filledRectangle(x1, y1, x2, y2, color)
	gd.filledRectangle(point1, point2, color)
WHERE:	x1 - horizontal pixel location of first corner
	y1 - vertical pixel location of first corner
	x2 - horizontal pixel location of second corner
	y2 - horizontal pixel location of second corner
	point1 - First point specification.
	point2 - Second point specification.
	color - color index or style to use for fill
RETURN:	void.

DESCRIPTION: This method is exactly the same as GD.rectangle(), except that it fills the rectangle, instead of drawing an outline. As with GD.rectangle(), if either point is out of bounds, then no drawing is done.

SEE: GD.rectangle(), GD.filledPolygon()

GD fillToBorder()

SYNTAX: gd.fillToBorder(x, y, border, color)

gd.fillToBorder(point, border, color)

WHERE: x - Horizontal position of starting pixel

y - Vertical position of starting pixel

point - Point specification of starting pixel.

border - Index border color to stop at

color - Fill color or style index

RETURN: void.

DESCRIPTION: This method fills the image with the selected color, until it hits a border with the color specified by `border`. `border` must be a color index, not one of the styled colors. `color` can be anything.

SEE: GD.fill()

EXAMPLE:

```
/* Will draw a circle with color index 1,
 * and then fill it with color
 * index 2.  The fill will stop
 * at the specified border, which means that
 * the second circle drawn, using color index 3,
 * will be erased as the
 * outer circle is filled.
 */
var gd = new GD(65, 65);
gd.arc( [32,32], 16, 16, 0, 360, 1 );
gd.arc( [32,32], 14, 14, 0, 360, 3 );
   // will be erased
gd.fillToBorder( [33,34], 1, 2 );
```

GD getInterlaced()

SYNTAX:	gd.getInterlaced()
RETURN:	boolean - Whether this image is interlaced.
DESCRIPTION:	If the current image has the interlace flag set, then this method returns true. Otherwise, it returns false.
SEE:	GD.interlace()

GD getPixel()

SYNTAX:	gd.getPixel(x, y)
	gd.getPixel(point)
WHERE:	x - horizontal position of pixel, measured from left
	y - vertical position of pixel, measured from top
	point - A point specification.
RETURN:	number - a color index indicating the color at the selected pixel.
DESCRIPTION:	This method accesses the pixel at position (x, y), and returns the color of that pixel. If the pixel coordinates are out of bounds, then zero is returned.
SEE:	GD.SetPixel()
EXAMPLE:	`var gd = GD.fromGif("test.gif");` `gd.getPixel(0,0);` `gd.getPixel( [0,0] );` `gd.getPixel( {x:0,y:0} );`

GD getTransparent()

SYNTAX:	gd.getTransparent()
RETURN:	number - The color index of the current transparent color for this image.
DESCRIPTION:	This method looks up the transparent color that was set by GD.transparent() or read from the file originally.
SEE:	GD.transparent()

GD green()

SYNTAX:	gd.green(index)
WHERE:	index - The color index to look up
RETURN:	number - The green component of the specified color index
DESCRIPTION:	This method looks up the color indicated by `index` and returns the green component of that color.
SEE:	GD.blue(), GD.red()

GD height()

SYNTAX:	gd.height()
RETURN:	number - The height of the image
DESCRIPTION:	This method returns the height of the current GD image

GD interlace()

SYNTAX:	gd.interlace(flag)
WHERE:	flag - A boolean value indicating whether this image is interlaced or not
RETURN:	void.
DESCRIPTION:	This method sets the interlace flag for the current image. If the `flag` parameter is true, then the image is interlaced, otherwise it is not. Interlaced GIF images allow views to gradually fade in the image, rather than having to read in the whole file and then display it. This flag only affects the image once it is saved as a GIF file. It has no affect on any other methods. Viewers which don't support interlacing will still be able to display the image, it will just appear all at once like any other image.
SEE:	GD.getInterlaced()

GD line()

SYNTAX:	gd.line(x1, y1, x2, y2, color)
	gd.line(point1, point2, color)

WHERE:	x1 - horizontal pixel location of starting point
	y1 - vertical pixel location of starting point
	x2 - horizontal pixel location of ending point
	y2 - vertical pixel location of ending point
	point1 - First point specification.
	point2 - Second point specification.
	color - color index or style to use for drawing line
RETURN:	void.
DESCRIPTION:	This method draws a line using color index `color`, starting from position (`x1, y1`) and going to position (`x2, y2`). Alternatively, the line is drawn from point1 to point2, if the coordinates are given in this manner. If either coordinate is out of bounds, then no drawing is done.
SEE:	GD.dashedLine()

GD polygon()

SYNTAX:	gd.polygon(point1[, x2, y2, ...], color)
WHERE:	pointN - Point specification for Nth point
	xN - x coordinate of Nth point
	yN - y coordinate of Nth point
	color - color index or style to use for line
RETURN:	void.
DESCRIPTION:	This method draws a polygon by connecting sequential points with lines. The parameters are either a pair of parameters indicating the two coordinates of the point, or a point specification type. A point type can either be an array with two elements, element 0 being the x coordinate and element 1 being the y coordinate, or an object with members 'x' and 'y', representing the x and y coordinates.
SEE:	GD.filledPolygon()

EXAMPLE:

```
// Draw a rectangle
function myRectangle(gd,x1,y1,x2,y2,color)
{
   gd.polygon([x1,y1], x1, y2, {x:x2,y:y2}, [x2,y1],
               [x1,y1], color);
}
```

GD rectangle()

SYNTAX: gd.rectangle(x1, y1, x2, y2, color)

gd.rectangle(point1, point2, color)

WHERE: x1 - horizontal pixel location of first corner

y1 - vertical pixel location of first corner

x2 - horizontal pixel location of second corner

y2 - horizontal pixel location of second corner

point1 - First point specification.

point2 - Second point specification.

color - color index or style to use for drawing line

RETURN: void.

DESCRIPTION: This method draws a rectangle with one corner located at position (`x1, y1`) and the other at position (`x2, y2`). The color used is specified by the `color` parameter. Alternatively, the coordinates can be specified with the point format. If either corner is out of bounds, then no drawing is done. Note that this is a shorthand function, as this can be accomplished in several other ways.

SEE: GD.filledRectangle(), GD.polygon()

EXAMPLE:

```
var gd = new GD(10,10);
gd.rectangle( 4, 5, 8, 9, 0 );

// is equivalent to:
var gd = new GD(10,10);
gd.line( [4,5], [8,5],  0 );
gd.line( [4,9], [8,9],  0 );
gd.line( [4,5], [4,9],  0 );
gd.line( [8,5], [8,9],  0 );
```

```
// which is also equivalent to:
var gd = new GD(10,10);
gd.polygon( [ [4,5], [8,5], [4,9], [8,9] ], 0 );
```

GD red()

SYNTAX: gd.red(index)

WHERE: index - The color index to look up

RETURN: number - The red component of the specified color index

DESCRIPTION: This method looks up the color indicated by `index` and returns the red component of that color.

SEE: GD.blue(), GD.green()

GD setBrush()

SYNTAX: gd.setBrush(brush)

WHERE: brush - A GD image to use as the current brush in this image

RETURN: void.

DESCRIPTION: This method sets the current brush for this image to be the image specified by `brush`. This image is then used for drawing when the "brushed" string is used as a color parameter to a drawing function. This method attempts to preserve the colors of the brush in the current image, including the transparent color. Transparent pixels are not draw when using the brush, allowing for brushes of any shape. The original brush must remain a valid image. Once destroy() has been called on the supplied brush, the style "brushed" can no longer be used until another brush is set. Note that because this can allocate colors in the image, do not set the brush if you won't be using it, because the color table could fill up quickly.

SEE: GD.setTile()

EXAMPLE:

```
var brush = GD.fromGif("brush.gif");
var gd = new GD(64,64);
gd.setBrush( brush );
gd.line( [16,3], [52,45], "brushed" );
brush.destroy();
```

GD setPixel()

SYNTAX: gd.setPixel(x, y, color)

gd.setPixel(point, color)

WHERE: x - horizontal position of pixel, measured from left

y - vertical position of pixel, measured from top

point - A point specification.

color - index into color table

RETURN: void.

DESCRIPTION: This method sets the designated pixel to the appropriate color. If either `x` or `y` is out of bounds, or if `color` is not a valid color index, then nothing is done.

SEE: GD.getPixel(), GD.colorAllocate()

EXAMPLE:

```
var gd = new GD(4,4);
var black = gd.colorAllocate(0,0,0);
gd.setPixel(0,0,black);
```

GD setStyle()

SYNTAX: gd.setStyle(style)

WHERE: style - style to set for current image.

RETURN: void.

DESCRIPTION: This method sets the current style for this image, which is used whenever the string "styled" is passed as a color index parameter to a drawing function. The parameters to the method is a list of pixels, which are color indexes or the special value -1, which indicates a transparent pixel. When drawing lines or a series of pixels, the drawing methods cycle through the sequence defined in the current style and applies the color to each successive pixel. If the value of -1 is used, then no color is applied and the background remains.

SEE: GD.setBrush(), GD.setTile()

EXAMPLE:

```
/* Create a Red, Green, Blue,
 * dashed line from the upper left
 * corner of the image to the lower right corner.
 * Each dash will be 3
 * pixels wide, and there will be 3 pixels
 * of space in between.
 */

var gd = new GD(64,64);
var red = gd.colorAllocate( 255, 0, 0 );
var green = gd.colorAllocate( 0, 255, 0 );
var blue = gd.colorAllocate( 0, 0, 255 );
gd.setStyle(red, red, red,
            -1, -1, -1,
            green, green, green,
            -1, -1, -1,
            blue, blue, blue,
            -1, -1, -1 );
gd.line( [0,0], [63,63], "styled" );
```

GD setTile()

SYNTAX: gd.setTile(tile)

WHERE: tile - A GD image to use as the repeating tile for this image

RETURN: void.

DESCRIPTION: This method sets the current tile for this image in a manner similar to GD.setBrush(). This tile image is then used whenever the style "tiled" is used as a color parameter in a function. The "tiled" style only works when calling a filling function, such as GD.fill() or GD.filledPolygon(). This method attempts to preserve the colors of the original tile, by either finding exact colors, allocating new colors, or finding the closest color if necessary. Transparent pixels in the image allow the underlying image to shown through. Once the tile is set with setTile(), the original tile must be retained as long as the image is being used. Otherwise, the result is undefined.

SEE: GD.setBrush()

EXAMPLE:
```
var tile = GD.fromGif("tile.gif");
var gd = new GD(64,64);
gd.setTile( tile );
gd.filledRectangle( [0,0], [63,63], "tiled" );
tile.destroy();
```

GD string()

SYNTAX:	gd.string(font, x, y, char, color)
	gd.string(font, point, char, color)
WHERE:	font - Font specification to use
	x - horizontal position of upper-left corner of character
	y - vertical position of upper-left corner of character
	point - Point specification. See GD.getPixel() for a description.
	string - The string to draw.
	color - color index or style to use for string
RETURN:	void.
DESCRIPTION:	This method draws a string on the current image, at the specified location and in the appropriate color. If the coordinates are out of bounds, then no drawing is done.
SEE:	GD.drawChar(), GD.stringUp()

GD stringUp()

SYNTAX:	gd.stringUp(font, x, y, char, color)
	gd.stringUp(font, point, char, color)
WHERE:	font - Font specification to use
	x - horizontal position of upper-left corner of character
	y - vertical position of upper-left corner of character
	point - Point specification. See GD.getPixel() for a description.
	string - The string to draw.
	color - color index or style to use for string
RETURN:	void.
DESCRIPTION:	This method is exactly the same as GD.string(), except that this method draws the string vertically, facing upwards.
SEE:	GD.charUp(), GD.string()

GD toGd()

SYNTAX:	gd.toGd(filename)
WHERE:	filename - Name of file to output to
RETURN:	boolean - Whether the operation was successful
DESCRIPTION:	This method outputs the gd object to the file in the native format of the library, which is unreadable by any other program, but can be read and written quickly. It is mostly used to store a commonly used base-image in native format, which can then be worked with from there.
SEE:	GD.toGd(), GD.fromGif()

GD toGif()

SYNTAX:	gd.toGif(filename)
WHERE:	filename - Name of file to output to
RETURN:	boolean - Whether the operation was successful.
DESCRIPTION:	This method compresses the GIF data in the appropriate manner, and outputs the contents of the image to the specified file in GIF form.
SEE:	GD.toGd(), GD.fromGif()

GD width()

SYNTAX:	gd.width()
RETURN:	number - The width of the image
DESCRIPTION:	This method returns the width of the current GD image
SEE:	GD.height()

GD object static methods

GD.fromGd()

SYNTAX:	GD.fromGd(filename)

WHERE: filename - name of GD file to open.

RETURN: object - new GD object with the contents of the specified file, or null if there was an error.

DESCRIPTION: This method attempt to open the specified GD file, and then reads in the data. A GD file is one created with the toGd() method, and is written in the library's native format. If there is an error opening the file or reading the data, then null is returned.

SEE: GD.fromGd(), GD.toGd()

GD.fromGif()

SYNTAX: GD.fromGif(filename)

WHERE: filename - name of GIF file to open.

RETURN: object - new GD object with the contents of the file, or null if there was an error.

DESCRIPTION: This method attempts to open the specified file, and then attempts to read in the GIF data. If there is an error opening the file or reading the data, then null is returned. Otherwise, the method constructs a new GIF object whose contents is the GIF read from the file.

SEE: GD.fromGd(), GD.toGif()

GD.fromXbm()

SYNTAX: GD.fromXbm(filename)

WHERE: filename - name of XBm file to open.

RETURN: object - new GD object with the contents of the specified file, or null if there was an error.

DESCRIPTION: This method attempts to open the specified XBM file, and then reads in the data. If there is an error opening the file or reading the data, then null is returned.

SEE: GD.fromGif()

MD5 Checksum Link Library

The md5 object provides a simple means of calculating checksums based on the md5 algorithm, a well-known and accepted method.

md5 Object

```
 platform: Mac, OS2, Windows, all versions of SE
   source: #link <md5.dll>
```

md5 object instance methods

md5()

SYNTAX:	new md5()
RETURN:	object - a new md5 checksum object.
DESCRIPTION:	This method creates a new object, and initializes it to be used for md5 sum computation. MD5 is an old, well-established checksum calculation formula that is still used for File download verification. The checksum verifies the integrity of the data, because if any bit is changes in the source, then the checksum will be drastically different.

term()

SYNTAX:	md5.term()
RETURN:	buffer - The computed checksum for this md5 object
DESCRIPTION:	This method MUST be called in order to correctly dispose of the md5 object. It returns a buffer, 16 bytes long, representing the md5 checksum for this object. It also frees up any memory being used by the object.

EXAMPLE:

```
var md5sum = new md5();
md5.update("hi");
var digest = md5.term();
// digest is now equal to the checksum of "hi"
```

update()

SYNTAX:	md5.update(buffer[, length])
WHERE:	buffer - A string or buffer of data to add into this checksum length - Length of data to be added. If not supplied, then the length of buffer is used.
RETURN:	void.
DESCRIPTION:	This method adds the supplied buffer into the running md5 checksum. If `length` is greater than the length of `buffer`, then the buffer is expanded as if filled with null bytes.

EXAMPLE:	`var md5sum = new md5();`
	`md5sum.update("hello");`
	`md5sum.update(", world!",4);`
	`md5sum.term();  // Return the checksum of "hello, wo"`

SEDBC Link Library

The link library, *sedbc.dll*, has methods and properties for working with a database in ScriptEase. These methods and properties provide a high-level interface for working with ODBC databases. The Database object allows the user to create a connection to a database which can then be queried, manipulated, and so forth through direct SQL statements or by the Cursor object. SQL statements stored inside the database are known as stored procedures and can be called using the Stproc object, allowing for the use of complex database-specific procedures from a script. Finally, true ease of use is provided by the SimpleDataset object, which is a combination of a Database object and a Cursor object. As a package, *sedbc.dll* allows a script to have detailed, low-level control of an ODBC database through SQL statements and easy to use, high-level routines at the same time.

Cursor Object

```
 platform:   Win32; all versions of SE
   source:   #link <sedbc.dll>
```

A Cursor object represents a database cursor for a specified SQL SELECT statement or specified database table.

Description of the Cursor object

A Cursor is a structure, created from a database table, which represents a subset of that table. When performing a query on a database, the results of the query are returned as a Cursor.

A Cursor object can be used to perform the following operations:

- Modify data in a database table.
- Navigate in a database table.
- Customize the display of the virtual table returned by a database query.

A Cursor object can be constructed in the following manners:

- The cursor method of a database object.
- The table method of a database object.
- The cursor method of a Stproc object.

There is no need to call a Cursor constructor.

A Cursor object has the notion of a "current" row. When operations are performed on a Cursor, they usually affect this row. The current row can be moved forward and backward through a Cursor using the `next` and `previous` methods, respectively. Similarly, the `first` and `last` methods set the current row to the first or last row in the cursor. Each of these methods will return `false` if the desired row does not exist within the Cursor. Thus, if the Cursor does not have any rows in it (perhaps because the SELECT statement used to create the cursor did not return any results), each of these methods will return false. Don't forget to check for this condition!

Important - A Cursor does not guarantee the order or positioning of its rows. For example, if a row is added to a Cursor, there is no way of knowing where that row will actually appear within in the cursor. Thus, do not make any assumptions about the ordering of rows within the Cursor. When finished with a Cursor object, use the `close` method to close it and release the memory it uses. If a database connection that has an open Cursor is released, the runtime engine waits until that Cursor is closed before actually releasing the connection to the database, so it is important to remember to close Cursors. If a Cursor has not been not explicitly closed using the close method by the time the associated Database or DbPool object goes out of scope, the runtime engine will try to close it. This may tie up system resources unnecessarily and/or lead to unpredictable results. Use the `prototype` property of the Cursor class to add a property to all Cursor instances. The addition applies to all Cursor instances running in all applications on the server, not just the application that made the change. This allows the capabilities of the object to be expanded for the entire server.

Cursor Instance Properties

The properties of Cursor objects vary from instance to instance. Each Cursor object has a property for each named column in the Cursor. Thus, when a Cursor is created, it acquires a property for each column in the virtual table, as determined by the SELECT statement.

Note - Unlike other properties in JavaScript, cursor properties corresponding to column names are not case sensitive, because SQL is not case sensitive and some databases are not case sensitive.

Properties of a Cursor object can be referred to as elements of an array. The 0-index array element corresponds to the first column, the 1-index array element corresponds to the second column, and so on.

SELECT statements can retrieve values that are not columns in the database, such as aggregate values and SQL expressions. Display these values by using the Cursor's property array index for the value.

Cursor filter

SYNTAX: cursor.filter

DESCRIPTION: A property containing a conditional expression that determines which subset of rows are retrieved by a cursor. This expression is a string containing the WHERE clause of an SQL statement describing the rows to be included. The string does not include the reserved word WHERE, however. Initially, the filter property value is set to the empty string, indicating that all of the Cursor rows are to be retrieved. Call `reload` after changing the filter to update the contents of the Cursor.

SEE: Cursor.reload()

EXAMPLE:

```
// assume 'database' is a valid Database object
var curs = database.table("customer")

// Set cursor filter so that the Cursor only
retrieves objects
// whose 'City' field is set to 'Berlin'
curs.filter = "City = 'Berlin'";

// Reload the cursor
err = curs.reload();
```

Cursor sort

SYNTAX: cursor.sort

DESCRIPTION: A property containing the sort order of a cursor. The Cursor sort order will determine the order that the rows are returned in when iterating the Cursor. The sort property is a string that contains the ORDER BY clause of an SQL statement. It does not include the reserved word ORDER BY, however. Initially, the sort property is set to the empty string, and, therefore, no item sort order is guaranteed. Call `reload` after changing the sort order to update the contents of the Cursor.

SEE: Cursor.reload()

EXAMPLE:

```
// assume 'database' is a valid Database object
```

```
var curs = database.table("customer")

// Set sort order so that the Cursor is sorted first
by the
// 'city' field, and, for records with the same
'city' value,
// descending by the field 'name'.
curs.sort = "city, name DESC";

// Reload the cursor
err = curs.reload();
```

Cursor Instance Methods

Cursor close()

SYNTAX:	cursor.close()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.
DESCRIPTION:	The `close` method closes a cursor or result set and releases the memory it uses. If a cursor is not explicitly closed using the `close` method, it will automatically be closed by the runtime engine when the corresponding client object goes out of scope.
SEE:	Database.majorErrorCode(), Database.minorErrorCode()
EXAMPLE:	`err = curs.close()`

Cursor columnName()

SYNTAX: cursor.columnName(n)

WHERE: n - zero-based integer corresponding to the column in the query. The first column in the result set is 0, the second is 1, and so on.

RETURN: string - the name of column number n in the cursor.

DESCRIPTION: Given a column number, `columnName()` returns the name of the column.

When using SELECT statements with wildcards (*) to select all the columns in a table, the columnName method does not guarantee the order in which it assigns numbers to the columns. Thus, use `columnName` to find which name corresponds to which column number.

SEE: Cursor.columns()

EXAMPLE:
```
// assume 'database' is a valid, open Database object
var curs = database.cursor(SELECT * FROM customer);

// get the name of the first column in the cursor
header = customerSet.columnName(0);
```

Cursor columns()

SYNTAX: cursor.columns()

RETURN: number - columns in a Cursor object.

DESCRIPTION: This function returns the number of named and unnamed columns that are present in the given Cursor.

SEE: Cursor.columnName()

EXAMPLE:
```
numCols = curs.columns();
```

Cursor deleteRow()

SYNTAX: cursor.deleteRow()

RETURN: number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database's `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION: This function, only available on up datable cursors, deletes the current row from the Database object.

SEE: Database.commitTransaction(), Database.rollbackTransaction()

EXAMPLE:
```
// assume 'database' is a valid Database object
var curs = database.table("customer");

// delete all rows from the Database where City is
"Medford"
while (curs.next())
{
  if(curs.City == "Medford")
    err = curs.deleteRow();
}
database.commitTransaction();
```

Cursor first()

SYNTAX: cursor.first()

RETURN: boolean - false if the cursor is empty or if cursor is forward-only cursor and the current row is not the first row, otherwise true.

DESCRIPTION: This method moves the current row to the first row in the Cursor and returns true so long as there is a first row. Note that if the cursor is empty, this method always returns false. Also note that, if the cursor does not allow backwards movement of the current row, false will be returned.

SEE: Cursor.next(), Cursor.previous(), Cursor.last()

EXAMPLE:
```
// assume 'database' is a valid Database object
var curs = database.table("customer");

// set the current row to the first row in the Cursor
curs.first();
```

Cursor insertRow()

SYNTAX: cursor.insertRow()

RETURN: number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database's `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION: This function, only available on undatable cursors, inserts a row into the associated database table. The location of the inserted row may vary depending on the database vendor's implementation, and thus row ordering is not guaranteed. There are several ways to specify values for the row being inserted:

Explicitly assigning values to each column in the cursor and then calling `insertRow`.

Choosing to a row using the `next` or `previous` methods, changing the values of some of the columns and then calling `insertRow`. Columns that were not explicitly assigned values will receive values from that initially chosen row.

Do not choose a row with `next` or `previous` and call `insertRow`. Since there is no current row in this case, all of the columns for the new row will be null.

Any columns in the cursor that contain unassigned values when `insertRow` is called will be null in the new row.

SEE: Cursor.next(), Cursor.previous(), Database.commitTransaction(), Database.rollbackTransaction()

EXAMPLE:

```
// assume 'database' is a valid Database object
var curs = database.table("customer");

// choose the first row to act as a "template" for
the new row
curs.next();

// plug in some values for the new row
curs.Name = "Fred Flintstone";
curs.City = "Bedrock";

// add the row to the database
err = curs.insertRow();
database.commitTransaction();
```

Cursor last()

SYNTAX: cursor.last()

RETURN: boolean - false if the cursor is empty; otherwise true.

DESCRIPTION: This method moves the current row to the last row in the Cursor and returns true so long as there is a last row. Note that if the cursor is empty, this method always returns false.

SEE: Cursor.next(), Cursor.previous, Cursor.first()

EXAMPLE:

```
// assume 'database' is a valid Database object
var curs = database.table("customer");

// set the current row to the last row in the Cursor
curs.last();
```

Cursor next()

SYNTAX:	cursor.next()
RETURN:	boolean - false if the current row is the last row; otherwise true.
DESCRIPTION:	The current row of a Cursor is initially positioned "before" the first row. Using the `next` method, the current row can be moved forwards through the records in the Cursor. The `next` method moves the pointer and returns true as long as there is another row available. When the current row has reached the last row of the Cursor, `next` returns false. Note that, in the event of an empty Cursor, this method will always return false.
SEE:	Cursor.previous(), Cursor.first(), Cursor.last()

EXAMPLE:

```
// assume 'database' is a valid Database object
var curs = database.cursor("select * from customer",
true);

// visit each object in the cursor
while (curs.next())
  ;
```

Cursor previous()

SYNTAX:	cursor.previous()
RETURN:	boolean - false if the current row is the first row; otherwise true.
DESCRIPTION:	Using the `previous` method, the current row can be moved backwards through the records in the Cursor. The `previous` method moves the pointer and returns true as long as there is another row available. When the current row has reached the first row of the Cursor, `next` returns false. Note that, in the event of an empty Cursor, this method will always return false.
SEE:	Cursor.next(), Cursor.first(), Cursor.last()

EXAMPLE:

```
// assume 'database' is a valid Database object
var curs = database.cursor("select * from customer",
true);

// set the current row to the last row in the cursor
curs.last();

// visit each object in the cursor, backwards
while (curs.previous())
```

```
;
```

Cursor reload()

SYNTAX:	cursor.reload()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database's `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.
DESCRIPTION:	Requeries the database and recreates the rows of the cursor, taking into account the filter and sort properties of the Cursor.
SEE:	Database.majorErrorCode(), Database.minorErrorCode(), Cursor.sort, Cursor.filter

EXAMPLE:

```
// assume 'curs' is a valid Cursor object
// Change sort order of the cursor rows
curs.sort = "Year";

// reload the cursor's contents
err = cursor.reload();
```

Cursor updateRow()

SYNTAX:	cursor.updateRow()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database's `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION:

This method uses the values in the current row of an undatable cursor to modify a row in a table. Before an `updateRow` can be performed, make sure the `next` method has been called at least once, so that the current row of the Cursor is assigned.

To update a row in a database table, assign values to columns in the current row of the cursor, and call `updateRow`. Column values that are not explicitly assigned are not changed by the updateRow method.

SEE: Cursor.next(), Cursor.previous(), Database.commitTransaction(), Database.rollbackTransaction()

EXAMPLE:
```
// assume 'database' is a valid Database object
var curs = database.table("customer");

// choose the first row to be updated
curs.next();

// update the values for the new row
curs.Paid = False;

// update the row in the Cursor
curs.updateRow();
database.commitTransaction();
```

Database Object

```
  platform:  Win32; all versions of SE
    source:  #link <sedbc>
  location:  link
```

The Database object allows an application to access and interact with a relational database.

Description of the Database object

Use the database object to connect to a remotely stored relational database stored on a server.

The database object can be used to perform the following tasks on a relational database:

- Execute SQL statements and queries on the database server
- Iterate the results of a query in order to process or display them
- Manage database transactions
- Run stored procedures

When closing down a database, be sure to close any associated open cursors, result sets, and stored-procedure objects, or else unpredictable results may occur.

Transactions

A transaction is a group of database actions that are performed together. Either all the actions succeed together or they all fail together. When a group of database actions is made permanent, it is called committing a transaction. Rolling back a transaction cancels all of the actions of a non-committed transaction.

Explicit transaction control is available for any set of actions using the `beginTransaction`, `commitTransaction`, and `rollbackTransaction` methods. If transactions are not controlled explicitly, the runtime engine uses the underlying database's autocommit feature to treat each database modification as a separate transaction. Each statement is either committed or rolled back immediately, based on the success or failure of the individual statement. Explicitly managing transactions overrides this default behavior.

NOTE: When making changes to a database, it is recommended that explicit transaction control be used. If not, the database may report errors. However, even if errors are not specifically reported, data integrity cannot be guaranteed unless explicit transactions are used. In addition, any time a Cursors object is used to update a database, it is also recommended that explicit transactions be used to ensure the consistency of the data.

For the database object, the scope of a transaction is limited to lifetime of the connection. If the database object is disconnected before calling `commitTransaction` or `rollbackTransaction` method, then the transaction is automatically rolled back.

Database beginTransaction()

SYNTAX:	database.beginTransaction()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.
DESCRIPTION:	After calling `beginTransaction`, all subsequent actions that modify the database are grouped within this transaction, known as the current transaction. Nested transactions are not supported. If `beginTransaction` is called when a transaction is already open, an error message will be returned.
SEE:	Database.commitTransaction(), Database.rollbackTransaction()
EXAMPLE:	`var err = db.beginTransaction();`

Database commitTransaction()

SYNTAX:	database.commitTransaction()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.
DESCRIPTION:	This method commits all of the actions performed since the last call to `beginTransaction`. If there is no current transaction (for instance, the application has not called `beginTransaction`), calls to `commitTransaction` are ignored.
SEE:	Database.beginTransaction(), Database.rollbackTransaction()
EXAMPLE:	`var err = db.commitTransaction();`

Database connect()

SYNTAX:	database.connect(dbtype, server, username, password)
WHERE:	dbtype - A string representing the database type. Currently only "ODBC" is supported.
	server - Data source name. On Windows systems using ODBC, this is specified in the ODBC Administrator Control Panel; on UNIX, in the .odbc.ini file. See your database or system administrator for more information.
	username - Name of the user to connect to the database. Some relational database management systems (RDBMS) require that this be the same as your operating system login name; others maintain their own collections of valid user names. If in doubt, see your system administrator.
	password - User's password. If the database does not require a password, use an empty string.
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the `majorErrorCode` and `majorErrorMessage` methods to

interpret the meaning of the error.

DESCRIPTION: Creates and caches a database connection to the specified database of the given type, using the username and password passed-in. When the connection goes out of scope, any pending transactions are rolled back. If any database connections are open when `connect` is called, they are closed and released before the new connection is opened.

SEE: Database.disconnect(), Database.connected()

EXAMPLE:
```
// This example creates a new database and then
connects it to
// the database named "CLIENTS" using the username
"ADMIN" and
// the password "admin-password"
var db = new database();
var err = db.connect("ODBC", "CLIENTS", "ADMIN",
"admin-passwd");
```

Database connected()

SYNTAX: database.connected()

RETURN: boolean - true if the Database object is currently connected to a data source, false otherwise.

DESCRIPTION: This method returns true if the Database object is currently connected to a database. If `connected` returns false, reconnect the database before performing any further database actions, otherwise the actions will result in errors.

SEE: Database.connect(), Database.disconnect()

EXAMPLE:
```
// This example first checks to see if the database
is
// connected to a data source.  If not, it connects
it to the
// database named "CLIENTS" using the username
"ADMIN" and the
// password "admin-password"
if (!db.connected())
  err = db.connect( "ODBC", "CLIENTS",
                    "ADMIN", "admin-passwd" );
```

Database cursor()

SYNTAX:	database.cursor(sqlstatement[, updateable])
WHERE:	sqlstatement - String containing a SQL SELECT statement supported by the database server. updateable - Boolean parameter indicating whether the cursor can be modified.
RETURN:	object - a new Cursor object, representing the results of the specified SQL statement.
DESCRIPTION:	This method creates a Cursor object that contains the rows returned by the specified SQL SELECT statement in the `sqlstatement` parameter. If the SELECT statement does not return any rows, the resulting Cursor object also has no rows. The optional `updateable` parameter specifies whether the Cursor object created can be modified. If no value is specified in the `updateable` parameter, the cursor is created non-updateable. If an updateable Cursor object is desired, the virtual table returned by the `sqlstatement` parameter must be updateable. For example, the SELECT statement passed as the `sqlstatement` parameter cannot contain a GROUP BY clause. In addition, the query usually must retrieve key values from a table. For more information on constructing updateable queries, consult your database vendor's documentation.
SEE:	Cursor object

EXAMPLE:

```
// This example creates the updateable cursor 'custs'
and
// returns the columns 'ID', 'CUST_NAME', and 'CITY'
from the
// customer table:
custs = db.cursor( "select id, cust_name, city from
customer",
                   true );
```

Database disconnect()

SYNTAX:	database.disconnect()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the

`majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION: Disconnects Database object from its data source.

SEE: Database.connect(), Database.connected()

EXAMPLE:

```
// The example checks to see if the Database object
is
// connected to a data source, and, if so,
disconnects it.
if (db.connected())
  err = db.disconnect();
```

Database execute()

SYNTAX: database.execute(sqlstatement)

WHERE: sqlstatement - string representing the SQL statement to execute.

RETURN: number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION: This method allows execution of any data definition language (DDL) or data manipulation language (DML) SQL statement supported by the database server that does not return a cursor (such as `CREATE`, `ALTER`, or `DROP`). Each database supports a standard core of DDL and DML statements. In addition, a database may support DDL and DML statements specific to that database vendor. Use `execute` to call any of those statements. However, a database vendor may provide functions that are not DDL or DML statements. Do not use `execute` to call those functions. For example, do not call the Oracle `describe` function or the Informix `load` function from the `execute` method.

Although the `execute` method can be used to perform data modification (`INSERT`, `UPDATE`, or `DELETE` statements), it is recommended that Cursor objects be used instead to achieve the same functionality. Using the Cursor object for these sorts of actions allows better database-type independence and also allows the use of binary large object (BLOb) data.

When using the `execute` method, the SQL statement must strictly conform to the syntax requirements of the database server. For example, some servers require each SQL statement be terminated with a semicolon. See the server documentation for more information. If a transaction has not been started with `beginTransaction`, the single statement is automatically immediately committed when `execute` is called.

SEE: Database.cursor(), Database.beginTransaction(), Database.commitTransaction(), Database.rollbackTransaction()

EXAMPLE:

```
// This example deletes all records from the database
// whose ID is 'requestedID'. It is recommended,
// however, that the Cursor object be used to perform
this action.
err = db.execute("delete from customer where
customer.ID = " + requestedID );
```

Database majorErrorCode()

SYNTAX: database.majorErrorCode()

RETURN: variable - the result returned by this method varies depending on the database server being used, but contains an error code indicating why the most recent database activity failed.

DESCRIPTION: SQL statements can fail for a variety of reasons, including referential integrity constraints, lack of user privileges, record or table locking in a multiuser database, and so on. When an action fails, the database server returns an error code indicating the reason for failure. Use this method to fetch that error code.

SEE: Database.majorErrorMessage(), Database.minorErrorCode(), Database.minorErrorMessage()

EXAMPLE:

```
errCode = db.majorErrorCode();
```

Database majorErrorMessage()

SYNTAX: database.majorErrorMessage()

RETURN: variable - the result returned by this method varies depending on the database server being used, but contains an error message explaining why the most recent database activity failed.

DESCRIPTION: SQL statements can fail for a variety of reasons, including referential integrity constraints, lack of user privileges, record or table locking in a multiuser database, and so on. When an action fails, the database server returns an error message indicating the reason for failure. Use this method to fetch that error message.

SEE: Database.majorErrorCode(), Database.minorErrorCode(), Database.minorErrorMessage()

EXAMPLE: `errMessage = db.majorErrorMessage();`

Database minorErrorCode()

SYNTAX: database.minorErrorCode()

RETURN: variable - the result returned by this method varies depending on the database server being used. In general, the method returns a secondary error code indicating a condition where the last database activity may not have completed as expected.

DESCRIPTION: The result returned by this method varies depending on the database server being used. In general, the method returns a secondary error code indicating a condition where the last database activity may not have completed as expected.

SEE: Database.majorErrorCode, Database.majorErrorMessage, Database.minorErrorMessage()

EXAMPLE: `errCode = db.minorErrorCode();`

Database minorErrorMessage()

SYNTAX: database.minorErrorMessage()

RETURN: variable - the result returned by this method varies depending on the database server being used. In general, the method returns a secondary error message indicating a condition where the last database activity may not have completed as expected.

DESCRIPTION: This method returns the secondary error message returned by database vendor library.

SEE: Database.majorErrorCode, Database.majorErrorMessage, Database.minorErrorCode()

EXAMPLE: `errCode = db.minorErrorMessage();`

Database procedureName()

SYNTAX: database.procedureName(n)

WHERE: n - Zero-based integer corresponding to the stored procedure in the database.

RETURN: The name of the stored procedure with index `n`.

DESCRIPTION: This method returns the name of the stored procedure corresponding to the specified index, `n`.

SEE: Stproc object, Database.storedProc(), Database.prodecureName()

EXAMPLE:

```
// fetch the name of stored procedure 0
procName = db.procedureName(0);
```

Database procedures()

SYNTAX: database.procedures()

RETURN: number - number of stored procedures in the database.

DESCRIPTION: This method returns the number of procedures stored in the database.

SEE: Stproc object, Database.storedProc(), Database.procedureName()

EXAMPLE:

```
// get the number of stored procedures in 'db'
procCount = db.procedures();
```

Database rollbackTransaction()

SYNTAX: database.rollbackTransaction()

RETURN: number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION: This method undoes all actions performed since the last call to `beginTransaction`. If there is no current transaction (for instance, the application has not called `beginTransaction`), calls to `rollbackTransaction` are ignored.

SEE: Database.beginTransaction, Database.commitTransaction()

EXAMPLE:
```
err = db.rollbackTransaction()
```

Database storedProc()

SYNTAX: database.storedProc(procName)

WHERE: procName - String specifying the name of a stored procedure or SQL statements with parameters.

RETURN: object - new `Stproc` object.

DESCRIPTION: This method creates a stored procedure object (`Stproc`) from either the named stored procedure contained within the Database object, or from the passed-in SQL statement.

SEE: Stproc object, Database.procedures(), Database.procedureName()

EXAMPLE:
```
// this example create a new database,
//and then executes
// a stored procedure contained within
var db = new Database;
db.connect(DBEngine, DataSource, User, Password);
var sp = db.storedProc("SomeProc");
sp.ItemID = 123;
sp.execute();

// now, execute an SQL Stproc
sp = db.storedProc("delete from Items where Weight =
?" );
sp[0] = 1000;
sp.execute();

// clean up
sp.close();
db.close();
```

Database table()

SYNTAX: database.table(tableName[, updateable])

WHERE: tableName - The name of an existing table in the database.

	updateable - Boolean flag indicating if the created cursor should be able to be modified (is updateable).
RETURN:	object - Cursor object representing the specified database table.
DESCRIPTION:	This method creates a new `Cursor` object from the specified table stored in the database. The resulting `Cursor` has one row for each row in the database table and will be empty if the database table has no rows. The optional `updateable` parameter specifies whether the created Cursor object can be modified. If no value is specified for the `updateable` parameter, it is false by default. To create an updateable `Cursor` object, the table specified in parameter must also be updateable.
SEE:	Cursor object, Database.tables(), Database.tableName()

EXAMPLE:

```
// create a new Cursor object from the "clients"
database table
clientsCurs = db.table( "clients", false );
```

Database tableName()

SYNTAX:	database.tableName(n)
WHERE:	n - Zero-based integer corresponding to the table in the database.
RETURN:	string - name of the table in the database with index `n`.
DESCRIPTION:	This method returns the name of the database table corresponding to the specified index, `n`.
SEE:	Database.table(), Database.tables()

EXAMPLE:

```
// fetch the name of database table 0
tableName = db.tableName( 0 );
```

Database tables()

SYNTAX:	database.tables()
RETURN:	number - number of tables in the database.
DESCRIPTION:	This method returns the number of tables stored in the database.
SEE:	Database.table(), Database.tableName()

```
EXAMPLE:        // get the number of tables in 'db'
                tableCount = db.tables();
```

SimpleDataset Object

```
    title: SimpleDataset object
 platform: Win32; all versions of SE
   source: #include <smdtset.jsh>
```

A SimpleDataset object is a easy-to-use database-access object that combines database and cursor functionality into a single object.

Description of the SimpleDataset object

SimpleDataset is a JavaScript class that combines the concept of a table and a cursor into a single, easy-to-use object. No more than one table may be represented by a SimpleDataset, so inserting items into the dataset doesn't require a target table to be specified. SQL is not needed to use a SimpleDataset and all operations can be performed through simple method calls.

When a SimpleDataset is created, it initially contains all of the rows ("records") in the specified table. The find() method allows this set to be reduced to only those records that match specified templates.

A SimpleDataset has the notion of the "current record". This is the record that SimpleDataset operations will affect. When the SimpleDataset is first created, the current record is the record "before" the first record, and is thus undefined.

Use the firstRecord(), lastRecord(), nextRecord(), and prevRecord() methods, to step through the records in the SimpleDataset. The current record is returned by currentRecord(). The objects returned by these routines have one property for each of the current record's fields.

The current record can be deleted using deleteRecord(). All items in the dataset can be deleted by deleteAll().

Records can be inserted to the SimpleDataset's table by insertRecord(). The "current" record can be replaced by a specified record using replaceRecord().

A Cursor object representing the SimpleDataset can be obtained by using the cursor() method. It may be necessary to use this to perform more powerful operations on the dataset.

Although the SimpleDataset can be closed through its close() method, it is automatically closed when the object goes out of scope.

Using the SimpleDataset object, the following five-line script can be used to print out the contents of a database:

```
function print_all(db, table, user, passwd)
{
   var ds = new SimpleDataset(db, table, user, passwd);

   while(var rec = ds.nextRecord())
     for(var prop in rec)
         Clib.printf(prop + " = " + rec[prop] + "\n");

   ds.close();
}
```

SimpleDataset instance methods

SimpleDataset()

SYNTAX:	new SimpleDataset(database, table, username, password)
WHERE:	database - The name of the ODBC database to open. On Windows systems using ODBC, this is specified in the ODBC Administrator Control Panel; on UNIX, in the .odbc.ini file. See your database or system administrator for more information. table - the name of the database table to use. username - name of the user to connect to the database. Some relational database management systems (RDBMS) require that this be the same as your operating system login name; others maintain their own collections of valid user names. If in doubt, see your system administrator. password - user's password. If the database does not require a password, use an empty string.
RETURN:	object - a new `SimpleDataset`, or `null` on error.
DESCRIPTION:	Constructor for the `SimpleDataset` object. When the `SimpleDataset` is created, it contains all of the elements in the table. The current element is set to the one "before" the first element in the dataset (and thus is "out of range").
EXAMPLE:	`// create a SimpleDataset connected to the database`

```
// named "corporate", table named "clients" using
// the username "ADMIN" and the password
// "admin-password"
var ds = new SimpleDataset("corporate", "clients",
                              "ADMIN", "admin-password");
```

SimpleDataset close()

SYNTAX: simpledataset.close()

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method closes the `SimpleDataset` object, freeing up the system resources being used by it. It also closes the associated, hidden, `database` and `Cursor` objects.

EXAMPLE:
```
var success = ds.close();
```

SimpleDataset currentRecord()

SYNTAX: simpledataset.currentRecord()

RETURN: object - the SimpleDataset's current record, or null if the current record is out of range.

DESCRIPTION: This method returns the record for the current record in the `SimpleDataset`. If the current element is undefined, `null` is returned. The returned object has one property for each field of the SimpleDataset's current record.

SEE: SimpleDataset.nextRecord(), SimpleDataset.prevRecord(), SimpleDataset.firstRecord(), SimpleDataset.lastRecord()

EXAMPLE:
```
// get the current record
var cr = ds.currentRecord();

// print out all of the fields of the object
for( var prop in cr )
   Clib.printf(prop + " = " + cr[prop] + "\n");
```

SimpleDataset nextRecord()

SYNTAX: simpledataset.nextRecord()

RETURN: object - the next record in the `SimpleDataset`. If there is no next record, `null` is returned.

DESCRIPTION: This method moves the current record forward in the `SimpleDataset` and returns the new current record. If the previous current record was the last record or the `SimpleDataset` is empty, `null` is returned.

SEE: SimpleDataset.currentRecord(), SimpleDataset.prevRecord(), SimpleDataset.firstRecord(), SimpleDataset.lastRecord()

EXAMPLE:
```
// get the next record
var rec = ds.nextRecord();

// so long as the record isn't null, print out all
// of the fields of the object
if( null != rec )
   for( var prop in rec )
      Clib.printf(prop + " = " + rec[prop] + "\n");
```

SimpleDataset prevRecord()

SYNTAX: simpledataset.prevRecord()

RETURN: object - the previous record in the `SimpleDataset`. If there is no previous record, `null` is returned.

DESCRIPTION: object - the previous record in the `SimpleDataset`. If there is no previous record, `null` is returned.

SEE: see: SimpleDataset.currentRecord(), SimpleDataset.nextRecord(), SimpleDataset.firstRecord(), SimpleDataset.lastRecord()

EXAMPLE:
```
// get the previous record
var rec = ds.prevRecord();

// so long as the record isn't null, print out all
// of the fields of the object
if( null != rec )
   for( var prop in rec )
      Clib.printf(prop + " = " + rec[prop] + "\n");
```

SimpleDataset firstRecord()

SYNTAX: simpledataset.firstRecord()

RETURN: object - the first record in the `SimpleDataset`. If the

`SimpleDataset` is empty, `null` is returned.

DESCRIPTION: This method moves the current record to the first record in the `SimpleDataset` and returns the new current record. If the `SimpleDataset` is empty, `null` is returned.

SEE: SimpleDataset.currentRecord(), SimpleDataset.nextRecord(), SimpleDataset.prevRecord(), SimpleDataset.lastRecord()

EXAMPLE:
```
// get the first record
var rec = ds.firstRecord();

// so long as the record isn't null, print out all
// of the fields of the object
if( null != rec )
   for( var prop in rec )
      Clib.printf(prop + " = " + rec[prop] + "\n");
```

SimpleDataset lastRecord()

SYNTAX: simpledataset.lastRecord()

RETURN: object - the last record in the `SimpleDataset`. If the `SimpleDataset` is empty, `null` is returned.

DESCRIPTION: This method moves the current record to the last record in the `SimpleDataset` and returns the new current record. If the `SimpleDataset` is empty, `null` is returned.

SEE: SimpleDataset.currentRecord(), SimpleDataset.nextRecord(), SimpleDataset.prevRecord(), SimpleDataset.firstRecord()

EXAMPLE:
```
// get the last record
var rec = ds.lastRecord();

// so long as the record isn't null, print out all of
the
// of the fields of the object
if( null != rec )
   for( var prop in rec )
      Clib.printf(prop + " = " + rec[prop] + "\n");
```

SimpleDataset find() with template

SYNTAX: simpledataset.find(template1[, template2[, ...]])

WHERE: templateN - Item template to search for. When more than one template is present, the templates are OR'd together.

Templates contain properties to match. Only those records which have properties that match those values will be included in the result set.

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method searches the `SimpleDataset`'s database table for all items that match the given templates. The contents of the `SimpleDataset` are changed to reflect the results of the search. The previous contents of the `SimpleDataset` are cleared and the complete database table is searched to create the new contents.

After the `find` has completed, the current record is set to the record "before" the first record. Fill out the properties in the template to indicate which items to find. For instance, to find all records whose 'city' field equals "Metropolis", set the value of the 'city' property to "Metropolis". If a template has more than one property, the properties will be combined with an AND to form the search term.

More than one template can be used. If multiple templates are used, the template values will be combined using an OR to form the search term.

SEE: SimpleDataset.findAll(), SimpleDataset.findDistinct(), SimpleDataset.caseSensitive

EXAMPLE:

```
// the following function will print out the fields
// of each of the records that have either Boston,
// USA or Paris, France as
// their city, country values
function print_BostonParis(db, table, user, passwd)
{
   // create the SimpleDataset
   var ds = new SimpleDataset(db, table,  user,
                              passwd);

   var template1, template2;

   template1.city = "Boston";
   template1.country = "USA";

   template2.city = "Paris";
   template2.country = "France";

   ds.find( template1, template2 );

   while( var rec = ds.nextRecord() )
      for( var prop in rec )
         Clib.printf(prop + " = " + rec[prop] +
                     "\n");

   ds.close();
}
```

SimpleDataset find() with clause

SYNTAX: simpledataset.find(whereClause)

WHERE: whereClause - A string containing the `WHERE` clause of an SQL statement (without the word `WHERE`) indicating which items to find.

RETURN: Boolean value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method searches the `SimpleDataset`'s database table for all items that match the given SQL `WHERE` clause. The contents of the `SimpleDataset` are changed to reflect the results of the search. The previous contents of the `SimpleDataset` are cleared and the complete database table is searched to create the new contents.

After the `find` has completed, the current record is set to the record "before" the first record.

The string passed into `find` contains a SQL `WHERE` clause. This allows more elaborate searches to be performed.

SEE: SimpleDataset.findAll(), SimpleDataset.findDistinct()

EXAMPLE:

```
// the following function will print out the fields
// of each of the records that have either Boston
// or Paris as their city values
function print_BostonParis(db, table, user, passwd)
{
   // create the SimpleDataset
   var ds = new SimpleDataset(db, table, user,
                              passwd);

   var whereClause;

   whereClause = "(City = \'Boston\') OR
                 (City = \'Paris\')";

   ds.find(template1, template2);

   while(var rec = ds.nextRecord())
      for(var prop in rec)
         Clib.printf(prop + " = " + rec[prop] +
                     "\n");

   ds.close();
}
```

SimpleDataset findAll()

SYNTAX: simpledataset.findAll()

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method clears the contents of the `SimpleDataset` and replaces them with the entire contents of the database table. Effectively, this resets the `SimpleDataset` to its initial state.

After setting the new contents, the current record is set to the record "before" the first record.

SEE: SimpleDataset.find(), SimpleDataset.findDistinct()

EXAMPLE:
```
// reset the contents of the SimpleDataset
err = ds.findAll();
```

SimpleDataset findDistinct()

SYNTAX: simpledataset.findDistinct(field)

WHERE: field - string indicating for which field duplicate values should be filtered out.

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method removes all records from the `SimpleDataset` that have duplicate values for the indicated fields. In other words, for the given field, only one record with each value is left in the `SimpleDataset`.

Note that no guarantees are made as to which records are left in the `SimpleDataset` for each value of the field.

SEE: SimpleDataset.find(), SimpleDataset.findAll()

EXAMPLE:

```
// print the unique country values
// in a SimpleDataset
function unique_countries(db, table, user, passwd)
{
   var ds = new SimpleDataset(db, table, user,
                              passwd );

   // find the distinct country values
   ds.findDistinct( "country" );

   while( var rec = ds.nextRecord() )
      Clib.printf(var.country + "\n");
}
```

SimpleDataset addRecord()

SYNTAX: simpledataset.addRecord(record)

WHERE: record - Object whose properties contain the values of the fields of the record to be added to the database table.

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method inserts the specified record into the `SimpleDataset` and its associated database table. The record to be inserted will have the field values indicated by the properties of the object passed into `addRecord`. After inserting a new record, the current record is left unchanged.

Note that no guarantees are made about the position of the inserted record within the `SimpleDataset`.

SEE: SimpleDataset.deleteRecord(), SimpleDataset.deleteAll()

EXAMPLE:

```
// The following function opens a SimpleDataset,
// adds the city
// Boston, Massachusetts to it,
// and then closes it down
function add_city(db, table, user, passwd)
{
   var ds = new SimpleDataset(db, table, user,
                              passwd );

   // set up the field values
   // of the item to be added
   var record;
   record.city = "Boston";
   record.country = "USA";
   record.state = "Massachusetts";
   record.population = 500000;

   // add the item and clean up
   ds.addRecord( record );
   ds.close();
}
```

SimpleDataset deleteRecord()

SYNTAX: simpledataset.deleteRecord()

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method removes the current record from the `SimpleDataset` and its associated database table. After deleting the record, the current record is set to the record "before" the first record.

SEE: SimpleDataset.deleteAll()

EXAMPLE:

```
// This function will delete all records
// with USA as their country
function delete_USA(db, table, user, passwd)
{
   var ds = new SimpleDataset(db, table, user,
                              passwd );

   // find the entries whose country is USA
   var template;
   template.country = "USA";
   ds.find( template );

   // delete the records from the SimpleDataset
   // and clean up
   while( ds.next() )
      ds.deleteRecord();
   ds.close();
}
```

SimpleDataset deleteAll()

SYNTAX: simpledataset.deleteAll()

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method removes all records from the `SimpleDataset`. The corresponding rows will also be deleted from the associated database table.

SEE: SimpleDataset.deleteRecord()

EXAMPLE:

```
// This function will delete all records
// with USA as their country
function delete_USA(db, table, user, passwd)
{
   var ds = new SimpleDataset(db, table, user,
                              passwd );

   // find the entries whose country is USA
   var template;
   template.country = "USA";
   ds.find(template);

   // delete the records from the SimpleDataset
   // and clean up
   ds.deleteAll();
   ds.close();
}
```

SimpleDataset replaceRecord()

SYNTAX: simpledataset.replaceRecord(record)

WHERE: record - object whose properties contain the values of the fields of the record to replace the current record with.

RETURN: boolean - value indicating success. In the case that the operation failed, use the `getLastErrorCode()` and `getLastError()` methods to determine the reason for the failure.

DESCRIPTION: This method replaces the current record in the `SimpleDataset` with the specified record. The record that the current record will be replaced with will have the field values indicated by the properties of the object passed into `addRecord`. After inserting a new record, the current record remains unchanged; that is, the current record is the record that replaced the previous current record.

SEE: SimpleDataset.addRecord(), SimpleDataset.deleteRecord()

EXAMPLE:

```
// This function will set the population
// of the first record
// with USA as its country to 100,000
function replace_population(db, table, user,
                            passwd)
{
   var ds = new SimpleDataset(db, table, user,
                              passwd );

   // find the entries whose country is USA
   var template;
   template.country = "USA";
   ds.find(template);

   // advance to first record in the result set
   var rec = ds.nextRecord();

   if( null != rec )
   {
      // set the new population value
      rec.population = 100000;

      // replace the record and clean up
      ds.replaceRecord(rec);
   }
   ds.close();
}
```

SimpleDataset cursor()

SYNTAX:	simpledataset.cursor()
RETURN:	object - the `Cursor` object that represents the current contents of the `SimpleDataset`
DESCRIPTION:	This method returns the Cursor object that represents the SimpleDataset. This may be useful if functionality beyond that of the SimpleDataset is required.
SEE:	Cursor object

EXAMPLE:	`// get the SimpleDataset as a Cursor`
	`var curs = ds.cursor();`

SimpleDataset getLastErrorCode()

SYNTAX:	simpledataset.getLastErrorCode()
RETURN:	number - integer specifying error code
DESCRIPTION:	This method returns an integer containing the code of any error encountered by the last SimpleDataset method call. The error codes/strings are reset whenever a SimpleDataset method is called (excluding getLastErrorCode() and getLastError()).
SEE:	SimpleDataset.getLastError()

EXAMPLE:	`// get the error code`
	`errCode = ds.getLastErrorCode();`

SimpleDataset getLastError()

SYNTAX:	simpledataset.getLastError()
RETURN:	string - message describing the last error encountered.
DESCRIPTION:	This method returns a string explaining the error encountered by the last SimpleDataset method call. The error codes/strings are reset whenever a SimpleDataset method is called (excluding getLastErrorCode() and getLastError()).
	SimpleDataset.getLastErrorCode()

EXAMPLE:	`// get a string describing the error`
	`error = ds.getLastError()`

SimpleDataset static properties

SimpleDataset.caseSensitive

SYNTAX:	SimpleDataset.caseSensitive
DESCRIPTION:	Boolean value indicating whether or not `SimpleDataset`'s `find` calls are case sensitive. By default, searches are not case sensitive.
SEE:	SimpleDataset.find()

EXAMPLE:	`// turn on case sensitivity`
	`// for SimpleDataset searches`
	`ds.caseSensitive = true;`

Stproc Object

```
    title: Stproc Object
 platform: Win32; all versions of SE
   source: #link <sedbc>
```

A Stproc object represents a call to a database stored procedure or SQL statement with parameters.

Description of the Stproc object

The Stproc object represents a stored procedure. A stored procedure is an SQL statement or other procedure that can be saved in a database object. The procedure object can be recalled and executed, if necessary returning its results as a Cursor object.

Stproc instance properties

The properties of Stproc objects vary from instance to instance. Each Stproc object has a property for each parameter in the stored procedure or SQL statement. Thus, when a Stproc object is created, it acquires a property for each of its parameters.

Parameters of a Stproc object may also be referred to as elements of an array. The 0-index array element corresponds to the first parameter, the 1-index array element corresponds to the second parameter, and so forth.

The following example demonstrates how to call a stored procedure using named parameter properties. A `GetCityArea` procedure might be defined in a MS Access database as follows:

```
PARAMETERS AreaParam Text, CityParam Text;
```

```
SELECT Table3.* FROM Table3
WHERE ((Table3.Area=[AreaParam]) AND
       (Table3.City=[CityParam]));

// Recall the Stproc object 'GetCityArea' from the database
sp = db.storedProc("GetCityArea");

// Set the parameter values
sp.AreaParam = ÔEuropeÕ;
sp.CityParam = ÔParisÕ;

// Execute the stored procedure
citySet = sp.cursor();

// Clean up
citySet.close();
sp.close();

This example uses the same procedure, but accesses the
parameters through array indexes.

// Recall the Stproc object 'GetCityArea' from the database
sp = database.storedProc("GetCityArea");

// Set the parameter values
sp[0] = ÔEuropeÕ;
sp[1] = ÔParisÕ;

// Execute the stored procedure
citySet = sp.cursor();

// Clean up
citySet.close();
sp.close();
```

Stproc instance methods

Stproc close()

SYNTAX:	stproc.close()
RETURN:	number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database's `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.
DESCRIPTION:	This method closes a `Stproc` object and releases the memory it

uses. If a `Stproc` object is not explicitly closed with the `close` method, the runtime engine automatically it when the corresponding `database` object goes out of scope.

SEE: Database.majorErrorCode(), Database.minorErrorCode()

EXAMPLE:
```
// Close down the Stproc
err = sp.close();
```

Stproc parameterName()

SYNTAX: stproc.parameterName(n)

WHERE: n - zero-based integer corresponding to the parameter in the Stproc object. The first parameter is 0, the second is 1, etc. return:

RETURN: string - the name of parameter n.

DESCRIPTION: This method returns the name of the parameter corresponding to the given index.

SEE: Stproc.parameters()

EXAMPLE:
```
// fetch the second parameter name
// of the Stproc 'sp'
paramName = sp.parameterName( 1 );
```

Stproc parameters()

SYNTAX: stproc.parameters()

RETURN: number - parameters in the Stproc.

DESCRIPTION: This method returns the number of named and unnamed parameters for the stored procedure or SQL statement.

SEE: Stproc.parameterName()

EXAMPLE:
```
// create an array of parameter names
// for Stproc 'sp'
for( i=0; i<sp.parameters(); i++ )
    pNames[i] = sp.parameterName( i );
```

Stproc cursor()

SYNTAX: stproc.cursor([updateable])

WHERE: updateable - Boolean parameter indicating whether the cursor can be modified.

RETURN: A new Cursor object, representing the results of the stored procedure.

DESCRIPTION: This method creates a Cursor object that contains the rows returned by the SQL `SELECT` statement of the stored procedure object. If the `SELECT` statement does not return any rows, the resulting `Cursor` object also has no rows.

The optional `updateable` parameter specifies whether the Cursor object created can be modified. If no value is specified in the `updateable` parameter, the cursor is created non-updateable.

If an updateable `Cursor` object is desired, the virtual table generated by the stored procedure must be updateable. For example, the `SELECT` statement cannot contain a `GROUP BY` clause. In addition, the query usually must retrieve key values from a table. For more information on constructing updateable queries, consult your database vendor's documentation.

SEE: Cursor object, Stproc.execute()

EXAMPLE:
```
// create a SQL stored procedure
SQL = "select id, cust_name, city from customer"
      "where (id >=?) and (id <=?)";
sp = database.storedProc(SQL);

// set the parameters
sp[0] = 1000;
sp[1] = 2000;

// create the cursor
custs = sp.cursor(true)
```

Stproc execute()

SYNTAX: stproc.execute()

RETURN: number - 0 if the call was successful; otherwise, a nonzero status code based on the error message produced by the database. If the method returns a nonzero status code, use the associated Database's `majorErrorCode` and `majorErrorMessage` methods to interpret the meaning of the error.

DESCRIPTION: This method executes the stored procedure of SQL statement. It allows execution of any stored procedure or SQL statement that uses data definition language (DDL) or data manipulation language (DML) statements supported by the database server and that do not return a cursor (such as `CREATE`, `ALTER`, or `DROP`).

Each database supports a standard core of DDL and DML statements. In addition, a database may support DDL and DML statements specific to that database vendor. Use `execute` to call any of those statements. However, a database vendor may provide functions that are not DDL or DML statements. Do not use `execute` to call a stored procedure using those functions. For example, do not call the Oracle `describe` function or the Informix `load` function from a stored procedure's `execute` method. Although the `execute` method can be used to perform data modification (`INSERT`, `UPDATE`, or `DELETE` statements), it is recommended that `Cursor` objects be used instead to achieve the same functionality. Using the `Cursor` object for these sorts of actions allows better database-type independence and also allows the use of binary large object (BLOb) data.

When using the `execute` method, the stored procedure's SQL statement must strictly conform to the syntax requirements of the database server. For example, some servers require each SQL statement be terminated with a semicolon. See the server documentation for more information. If a transaction has not been started with `beginTransaction`, the single statement is automatically immediately committed when the stored procedure's `execute` method is called.

SEE: Stproc.cursor(), Database.majorErrorCode(), Database.majorErrorMessage()

EXAMPLE:

```
// Create a new database object, and
// connect it to a data source
var a = new Database;
a.connect(DBEngine, DataSource, User, Password);

// execute the stored procedure 'SomeProc'
var sp = a.storedProc("SomeProc");
```

```
sp.ItemID = 123;
sp.execute();

// execute an SQL stored procedure
sp = a.storedProc("delete from Items where Weight =
?");
sp[0] = 1000;
sp.execute();

// clean up
sp.close();
a.close();
```

Socket Link Library

The Socket object is used to communicate between computers over the internet through sockets.

Socket Object

```
platform: All OS except DOS and OS2; All Versions of SE
  source: #link <sesock.dll>
```

Socket object instance methods

Socket() with hostname

SYNTAX:	new Socket(hostname, port)
WHERE:	hostname - Name of remote host to connect to
	port - Port of remote host to connect to
RETURN:	object - A new Socket object, or null on error
DESCRIPTION:	This method attempts to connect to the specified remote host. If the library is unable to connect to the remote host, then null is returned and error is set. Once the connection is established, the socket can be read from / written to until it is closed or the connection is lost.
SEE:	Socket.error()

EXAMPLE:

```
function connect( hostnamePort )
{
   var index;
   var port = 1000;  // Default port

   if( (index = hostnamePort.indexOf(":")) != -1 )
   {
      port = ToNumber(hostnamePort.substring(index,
                      hostnamePort.length));
      hostnamePort = hostnamePort.substring(0,index);
   }

   var socket = new Socket(hostnamePort, port);
   return socket;
}
```

Socket() with port

SYNTAX: new Socket(port)

WHERE: port - Port to listen on

RETURN: object - A Socket object, or null on error.

DESCRIPTION: There are two types of sockets, in general. One type is a socket which is an established connection between a client and a server. This socket can be read to and written from just like a file. The other type of socket is a listening socket, which is a server-side socket which is not connected to a specific client, but rather to a certain port. It is listening for any new requests on that port. Requests can be checked for using the select() method. Once it is established that there is a request waiting, the peer-to-peer connection can be established using the accept() method. This creates a new connection socket on another port, leaving the original socket still listening for incoming connections.

SEE: Socket.select(), Socket.accept()

EXAMPLE:

```
var listenSocket = new Socket( 1000 );

if( listenSocket != null )
{
   if( listenSocket.ready() )
   {
      var connectSocket = listenSocket.accept();
      if( connectSocket != null )
      {
         // Finally! we have the socket
         // ... do stuff with socket ...
         connectSocket.close();
      }
   }
}

/* This will create a socket to listen on port 1000
 * and wait for any incoming
 * connections.  The no-parameter form
 * of ready() uses an infinite
 * timeout, so the program waits indefinitely
```

```
                      * for a connection.  This is
                      * also equiavalent to
                      * "Socket.select(-1,listenSocket)", which is a
                      * generic form which allows for
                      * listening on multiple sockets.
                      */
```

Socket accept()

SYNTAX:	socket.accept()
RETURN:	object - A new socket object connected to the client of the incoming request, or null if there is an error.
DESCRIPTION:	If there is no incoming request waiting on the socket, or this socket is not listening on a certain port, then it is an error and null is returned. Otherwise, the method establishes a socket connection on another port and returns a socket object representing this connection. The returned socket can later be used for reading/writing and other communication between the client and server.
SEE:	Socket.select(), Socket.ready()

Socket blocking()

SYNTAX:	socket.blocking(flag)
WHERE:	flag - A boolean value indicating whether this socket is to be blocking or non-blocking.
RETURN:	boolean - Whether the operation was successful.
DESCRIPTION:	This method sets the state of the socket to be blocking if `flag` is true, and non-blocking otherwise. A blocking socket will wait indefinitely for data on reads, while a non-blocking socket will immediately exit with an error indicating that there is no data to be read. By default, all sockets are blocking when they are created.
SEE:	Socket.select(), Socket.read()

Socket close()

SYNTAX:	socket.close()
RETURN:	boolean - Whether the operation was successful or not
DESCRIPTION:	This method closes the specified socket and frees up any memory associated with the object. It must be called to appropriately dispose of the socket. If not explicitly called, then the socket will be automatically closed when the library is unloaded. If the socket is successfully closed, then true is returned, otherwise false. The nature of the error can be retrieved with Socket.error().
SEE:	Socket.error()

Socket linger()

SYNTAX:	socket.linger(flag[, timeout])
WHERE:	flag - A boolean value indicating whether this socket will linger or not timeout - A timeout, in seconds, to wait for when closing the socket, only used when linger is on. This defaults to 10 if not supplied.
RETURN:	boolean - Whether the operation was successful
DESCRIPTION:	If `flag` is true, then this socket is set to linger, otherwise it is not. A lingering socket will remain active after closing it if there remains data to be read or written. If linger is not set, then the socket will immediately close, but any remaining data will be sent, if possible, before closing the socket. If linger is active, and timeout 0, then the socket is immediately closed and any unsent data is lost, simulating a hard close. Otherwise, the socket remains open until the data is transferred or `timeout` is reached. By default, all sockets are non-lingering.
SEE:	Socket.close()

Socket read()

SYNTAX: socket.read(destination, description)

WHERE: destination - A destination variable which will be converted to the appropriate type based on description.

descript - A variable description, either one of the special Blob variables UWORD8, SWORD8, UWORD16, SWORD16, UWORD24, SWORD24, UWORD32, SWORD32, FLOAT32, FLOAT64, FLOAT80, a blobDescriptor object describing a structure, or a positive value indicating the length of a buffer to read.

RETURN: number - elements read.

DESCRIPTION: This method is almost identical to Clib.fread(), except that it reads from the current socket rather than a supplied file. The `description` variable acts in the same way as Clib.fread(). If it is a positive value, then `destination` is treated as a buffer and filled with raw data. Otherwise, one of the blob types or blob descriptors can be used to read data values. For buffers, the length of the buffer read is returned. For all other values, 1 is returned if the item is read successfully, -1 or 0 otherwise. Use Socket.error() to determine the nature of the error. Typically, -1 means the socket is non-blocking and no data is available to read. 0 usually indicates that the program at the other end of the socket closed it.

SEE: Clib.fread(), Socket.write(), Socket.error()

EXAMPLE:

```
function readInfo( socket )
{
   var description = new blobDescriptor();
   description.name = 12;
   description.age = UWORD8;
   description.extension = UWORD16;
   var info;

   if( !socket.read( info, description ) )
      return null;
   else
      return info;
}

/* The above function will read the special
 * info data structure from
 * the socket, returning null
 * if there is some sort of error.
 */
```

Socket ready()

SYNTAX: socket.ready([timeout])

WHERE: timeout - Maximum time to poll, in milliseconds, or -1 for no timeout

RETURN: boolean - Whether the socket is ready for reading in the specified time.

DESCRIPTION: This method is very similar to "Socket.select()", except that it only polls the current socket for input. If no timeout is specified or it is -1, then the socket is polled indefinitely, unless there is an error. This method is useful for simple applications, when there are just a few sockets open, or in special instances where the select() method is impractical or impossible.

SEE: Socket.select()

EXAMPLE:

```
var listenSocket = new Socket( 1000 );

// Assume 'done' is a global flag
if( listenSocket != null )
{
   while( !done )
   {
      if( listenSocket.ready(10) )
      {
         // Open connection with accept() ...
      }

      // Do other idle stuff...
   }

   listenSocket.close();
}

/* This code creates a socket listening
 * on port 1000, and continuously
 * polls it to see if there is
 * an incoming connection, alternatively doing
 * idle code when there is no connection ready.
 */
```

Socket remoteHost()

SYNTAX: socket.remoteHost()

RETURN: string - The host this socket is connected to, or null if it is not connected.

DESCRIPTION: For listening sockets that are only connected to a port, then this method returns null. Otherwise, it returns the name of the remote host, either the server that the socket connected to, or the client from an incoming request. This method can be used to tell whether or not a socket is listening or is connected.

EXAMPLE:

```
var listenSocket = new Socket( 1000 );

if( listenSocket != null )
{
  if( listenSocket.ready() )
  {
     var connection = listenSocket.accept();
     if( connection != null )
     {
        // Print out name of incoming request
        Screen.writeln( connection.remoteHost() );
        // .. do other stuff ...
        connection.close();
     }
  }
  listenSocket.close();
}
```

Socket write()

SYNTAX: socket.write(source, description)

WHERE: source - Source variable to write to the socket

description - Variable description describing how to write the source variable to the socket.

RETURN: number - The number of elements read.

DESCRIPTION: This method is almost identical to Clib.fwrite(), except that it writes to the current socket, rather than to a supplied file. The `description` variable acts in the same way as in Clib.fwrite(). If it is a positive value, then `source` is treated as a buffer of the specified length. Otherwise, `description` must be a Blob value (SWORD8, UWORD32, etc) or a blobDescriptor object describing how the data should be written to the socket. If `source` is a buffer, then the number of bytes written is returned, otherwise 1 is returned if the datum is successfully written, -1 otherwise. Use Socket.error() to determine the nature of the error.

SEE: Clib.fwrite(), Socket.read(), Socket.error()

EXAMPLE:

```
function writeInfo( socket, info )
{
   if( !socket.write( info.name, 12 ) ||
       !socket.write( info.age, UWORD8 ) ||
       !socket.write( info.extension, UWORD16 ) )
      return false;
   else
      return true;
}

/* This function will write the contents
 * of the info object to the
 * specified socket in a native data format.
 */
```

Socket object static methods

Socket.addressByName()

SYNTAX: Socket.addressByName(address)

WHERE: address - Address of host to look up

RETURN: string - The address of the specified host.

DESCRIPTION: This method attempts to find the address of the specified host through a reverse DNS lookup. If this lookup is successful, then the address is returned as a string. Otherwise, null is returned.

SEE: Socket.hostByName()

Socket.error()

SYNTAX:	Socket.error()
RETURN:	number - The last error from the socket library
DESCRIPTION:	When there is some sort of error within the socket library, the special errno value gets set indicating the error number. If a method returns a value indicating an error, this method can be used to determine the exact nature of the error. The actual meaning of the value depends what system is being run.

Socket.hostByName()

SYNTAX:	Socket.hostByName(name)
WHERE:	name - Name of host to look up
RETURN:	string - The name of the specified host
DESCRIPTION:	This method looks up the specified host through a DNS lookup and returns it. This method can be used to convert between numerical addresses and domain names, as well as resolving local names appropriately. If unable to find the host name, then null is returned.
SEE:	Socket.addressByName()
EXAMPLE:	`var hostname = Socket.hostByName("44.55.66.77");`

Socket.hostName()

SYNTAX:	Socket.hostName()
RETURN:	string - The name of the host
DESCRIPTION:	This method attempts to find the name of the local host. If the call is successful, then a string is returned with the name of the host. Otherwise, the empty string is returned.

Socket.select()

SYNTAX:	Socket.select([timeout ,] socket1[, socket2 ...])
WHERE:	timeout - Maximum time to poll, in milliseconds, -1 for no

timeout

socketN - A list of sockets to poll for data, or an array of sockets

RETURN: object - The first supplied socket object which is ready for reading, or null if none is ready before the timeout is reached.

DESCRIPTION: This method is an alternate form of "socket.ready()". The other ready method is a property of Socket instances, and only polls the current socket for data. This global method allows for polling of multiple sources, which is needed when multiple sockets are open. When any of the specified sockets are ready to be read from, then this method returns the first socket which is so ready. Note that these sockets can be either connected sockets or listening sockets. A listening socket that is ready to be read from means that a request is waiting. If no timeout is specified, then -1 (infinite) timeout is used.

SEE: Socket.ready()

EXAMPLE:

```
var listenSocket1 = new Socket( 1000 );
var listenSocket2 = new Socket( 1001 );

// Assume 'done' is a global flag somewhere
if( listenSocket1 != null && listenSocket2 != null )
{
   while( !done )
   {
      var acceptSocket;
      if((acceptSocket = Socket.select(100,
          [listenSocket1, listenSocket2])) != null)
      {
         // Connect with socket ...
      }

      // Do other stuff ...
   }
}

/* This code opens two sockets for listening,
 * and then continuously polls
 * these two sockets for incoming connections.
 *  Note that in a real
 * program, it would be better to create
 * a dynamic array which holds all
 * of the open sockets.
 */
```

Com Object Link Library

The Com Object consists of only one function to create a Com object link.

Com Object

```
    title: COM object link library
 platform: WINDOWS; All versions except WMLScriptEase
   source: #link <comobj.dll>
```

The COM object library provides utilities for using COM objects from within scripts.

COMCreateObject()

SYNTAX:	COMCreateObject(COMObject)
WHERE:	COMObject - the name of a COM object.
RETURN:	object - An instance of the specified COM object.
DESCRIPTION:	Create an instance of a COM object to be used in a script.
EXAMPLE:	`var excel1 = COMCreateObject("Excel.Application");`

Script Libraries

Script librariesscript libraries and library fileslibrary files are normal text script files with the extension *jsh*. The core language, both JavaScript and C, of ScriptEase is found in ScriptEase interpreters, such as *sewin32.exe* and *secon32.exe*. ScriptEase uses *dll* files, known as link libraries, to add objects, functions and power to the core language. The advantage of link libraries is that they are precompiled machine language libraries that execute as fast as internal routines. The disadvantages are that they are static and take much programming effort to develop. Script libraries are dynamic and can be developed quickly.

Script libraries are written using the same program statements and structure as normal scripts. They are included in scripts using the `#include`#include preprocessor directive. The following fragment is an example of including a script library:

```
#include <dlgobj.jsh>

function main()
{
   /*
      Put your normal code here.
      Use the Dialog object, defined in dlgobj.jsh, and
      call its methods as if they were a part of ScriptEase.
   */
}
```

Nombas ScriptEase products are based on the concept that the core language may be extended or expanded by writing scripts. For example, Nombas provides the ScriptEase Integration Software Developer's Kit (ISDKISDK) which allows programmers to add script and macro ability to their own applications, a scripting ability based on JavaScript. Nombas also provides the ScriptEase Web Server Edition (SEWSESEWSE) which allows the use of server-side JavaScript for CGICGI. As with ScriptEase Desktop, programming tasks may be faster, simplified, improved, and empowered by writing simple text scripts that enhance the use and power of ScriptEase. To learn more about other Nombas ScriptEase products, visit us at:

http://www.nombas.com/us/

Many useful and powerful objects, methods, and functions are implemented using scripts. These scripts have two obvious advantages. First, enhancements can be added to ScriptEase JavaScript quickly and easily. Two, programmers have access to the source code and can alter it to fit their unique needs.

The following descriptions provide summary information about some common and useful script library filesscript library files. The details about objects, properties, methods, and functions are documented in the files themselves. The documentation is in the form of special comments that are consistent with the reference tables used in this manual.

Common script libraries

Some script libraries are common to two or more platforms and operating systems. These libraries are found in *C:\SEdesk\include* (assuming the use of the default ScriptEase Desktop directory *C:\SEdesk*).

ansi.jsh	Allows the use of color and other enhancement to text output when using *ansi.sys*.
array.jsh	Enhances the use of array, both JavaScript arrays and ScriptEase automatic arrays. Provides methods that allow both types of arrays to be used more interchangeably.
cmdline.jsh	Provides command line handling and is used by ScriptEase shells.
copyfile.jsh	Routines for copying files on disks.
datetime.jsh	Routines to simplify working with date and time strings in special formats.
dspfile.jsh	Provides distributed scripting using file transfer protocol.
exec.jsh	Enhancements to calling and executing programs from a script.
file.jsh	Various file handling routines for different platforms and operating systems.
filename.jsh	Various routines to work with filenames on different platforms and operating systems.
fileobj.jsh	Defines a File object and its methods to enhance working with files.
filepack.jsh	Routines for packing files onto the end of some binary file

	and retrieving them from the binary file.
getopt.jsh	Routines for working with command line arguments using Unix-like getopt.
idsp.jsh	Provides distributed scripting using internet protocol.
inout.jsh	Routines for user input and output in a text window.
item.jsh	Defines an Item object which works with items of data as a delimited string or as an array. Both formats of the data are simultaneously updated. A programmer can use String or Array methods to work with data.
key.jsh	Defines a Key object with methods to work with keyboards.
lock.jsh	Use file locking for exclusive temporary access to resources.
mail.jsh	Perform TCP/IP Mailing tasks from scripts.
nntp.jsh	Routines for working with the Network News Transfer Protocol, that is, with newsgroups.
optparms.jsh	Routines for working with command line arguments using techniques different form Unix-like getopt.
setest.jsh	Routines to assist in testing scripts.
seutil.jshseutil.jsh	The base include file that defines common variables for use in all scripts. If a script includes any files, it should include this file first.
smdtset.jsh	Defines a SimpleDataset object and methods for working with data using this object.
sqlconst.jsh	Defines SQL constants useful for working with ODBC.
string.jsh	Many methods added to the String object and the Clib object for enhanced string handling.
struct.jsh	Routine for initializing and working with a structure array.
url.jsh	Routines getting and manipulating text pages from URLs.
winini.jsh	Routines for working with Windows like initialization/profile files from a non-Windows operating system, such as DOS.
write.jsh	Routines to provide enhanced `write` type operations, some,

such as `wlb()`, are useful for debugging.

Common utility and sample scripts

There are many utility and sample scripts in *C:\SEdesk\utility* and *C:\SEdesk\sample* (assuming the use of the default ScriptEase Desktop directory *C:\SEdesk*). The utility scriptsutility scripts are useful and ready to run script to perform tasks on your computer. The sample scriptssample scripts primarily illustrate how to use script libraries and to perform tasks that are useful to different scripters.

Win32 script libraries

Script libraries with objects and methods for working in a Win32 Windows environment are in *C:\SEdesk\win32\include* (assuming the use of the default ScriptEase Desktop directory *C:\SEdesk*).

bmp.jsh	Routines for working with bitmap (.bmp) files.
clipbrd.jsh	Functions for reading from and writing to the Windows clipboard.
colors.jsh	Routines to control colors in a ScriptEase text screen.
dlgobj.jsh	Defines the Dialog object and provides many methods and routines for programming dialog or GUI style windows for interacting with users.
dropsrc.jsh	Functions to facilitate drag-and-drop operations.
gdi.jsh	Wrappers for some of Windows' GDI graphics routines. For use in the WS_PAINT message handler function of a window
getit.jsh	The getItem routines are automated routines that allow the selection of an item in a listbox and the getLine routines are similar to input boxes. These routines are common dialogs.
hotkey.jsh	Library to simplify the creation of hot keys to perform arbitrary tasks when a key-code combination is pressed.

icon.jsh	Routines useful for working with icons.
inputbox.jsh	Provides useful InputBox and InfoBox functions that do not use the Dialog object.
keypush.jsh	Routines to control or mimic the pushing of keys on the keyboard for another application.
keypushg.jsh	Routines to control or mimic the pushing of keys on the keyboard for another application. In German.
menuctrl.jsh	Functions for creating and controlling window menus in another application.
message.jsh	ScriptEase code wrapper for the *`SendMessage()`* and *`PostMessage()`* Windows functions. With these routines, any message can be sent or posted to any window.
mouseclk.jsh	Routines to control or mimic mouse operations for another application.
msgbox.jsh	Various message box functions, not based on the Dialog object, to simplify user interaction.
pickfile.jsh	Routines for working with the open file dialog of the Common Dialog DLL.
profile.jsh	Routines for working with Windows initialization/profile files.
profobj.jsh	Defines the Profile object and methods for working with Windows initialization/profile files.
regobj.jsh	Defines the Registry object and methods for working with the Windows registry.
screen.jsh	Methods added to the Screen object that enhancement work with a ScriptEase text screen.
shortcut.jsh	Routines to create Windows shortcut or *lnk* files.
useful.jsh	A few general utility functions.
win32api.jsh	Defines for working with the Windows 32 API.
window.jsh	Common defines for creating and defining windows using the ScriptEase `MakeWindow()`, `BreakWindow()`, and

	`DoWindows()` functions.
winexec.jsh	Multiple functions for executing scripts and programs using various techniques in the Windows API.
winobj.jsh	Defines the Window object and methods for manipulating windows on the screen.
wintools.jsh	Functions for setting the state of windows.
winvers.jsh	Routines for working with version information.

Win32 utility and sample scripts

There are many utility and sample scripts in *C:\SEdesk\win32* and *C:\SEdesk\win32\sample* (assuming the use of the default ScriptEase Desktop directory *C:\SEdesk*). These scripts are geared toward a Win32 environment. The utility scriptsutility scripts are useful and ready to run script to perform tasks on your computer. The sample scriptssample scripts primarily illustrate how to use script libraries and to perform tasks that are useful to different scripters.

Appendix B
Instance and Static Notation

ScriptEase uses object properties which are integral to JavaScript. For clarity we refer to object **properties** and object **methods**, not just properties, though both properties and methods may be referred to by the general term property. When using the terms property and method, object properties refer to the variables and data of an object and object methods refer to the functions of an object. We have clarified one dimension of object properties and methods. But, to be precise, we must deal with another dimension.

Object properties and methods are either **instance**, belonging to an instance of an object, or **static**, belonging to an object itself. Thus, all properties and methods of an object may be classified according to two dimensions. Is a property of an object a property or a method, and is it an instance or a static property? The following examples illustrate

- **Instance property** `string.length`
- **Instance method** `string.indexOf()`
- **Static property** *`String.illus`*
- **Static method** `String.fromCharCode()`

Objects may have all four categories of methods and properties, but usually they do not. In this illustration, the String object has three of the categories, but not a static property, which is the reason why *`String.illus`* had to be made up for this example.

ScriptEase documentation uses a couple of style conventions to distinguish between properties and methods and between being instance or static. The four sections, following the bullet list of explanations, illustrate how these distinctions are made in reference sections of documentation.

- First, headings, such as "String instance properties" below, specifically identify whether the following reference information applies to instance properties, instance methods, static properties, or static methods.
- Second, properties do not have parentheses "()" but methods do.

- Third the top lines of reference tables vary in how they refer to instance and static properties and methods. Instance properties and methods have object names followed by a space, such as "String ", whereas static properties and methods have object names followed by a period, such as "String.".
- Fourth, the syntax line for instance properties and methods uses the object name in all lowercase, whereas, the syntax line for static properties and methods uses the object name precisely. The significance is that instance properties and methods actually use the variable name of an instance of an object, whereas, static properties and methods use the actual object name itself.
- Fifth, the use of lowercase for instance properties and methods is used consistently in text and descriptions, not just the reference tables themselves.

String instance properties

String length

SYNTAX:	string.length
DESCRIPTION:	
SEE:	
EXAMPLE:	

String instance methods

String indexOf

SYNTAX:	string.indexOf(substring[, offset])
WHERE:	
RETURN:	
DESCRIPTION:	
SEE:	
EXAMPLE:	

String static properties

String.illus

SYNTAX: String.illus

DESCRIPTION:

SEE:

EXAMPLE:

String static methods

String.fromCharCode()

SYNTAX: String.fromCharCode(char1[, char2 ...])

WHERE:

RETURN:

DESCRIPTION:

SEE:

EXAMPLE:

Prototype property

For the technically inclined, objects have a `prototype` property. Instance properties and methods are attached to the `prototype` property of an object. As an illustration, assume that two new methods and two new properties are added to the `String` object. The instance property and method are added to the `prototype` property of the `String` object, whereas, the static property and method are added to the `String` object itself.

The following two declaration lines illustrate an instance property and an instance method:

```
String.prototype.newInstanceProperty
String.prototype.newInstanceMethod()
```

The following two declaration lines illustrate a static property and a static method:

```
String.newStaticProperty
String.newStaticMethod()
```

The following code fragment illustrates the differences in using these properties and methods.

```
   // Begin an instance of a String object
var newStr = "an example string";
var instVal = newStr.newInstanceProperty;
newStr.newInstanceMethod();
   // Use the static property and method directly
var statVal = String.newStaticProperty;
String.newStaticMethod();
```

Appendix A
Grouped Functions

In the current section, the functions and methods of ScriptEase are organized according to purpose and operation and not according to object. Some functions and methods are specific to certain operating systems and do not exist in all versions of ScriptEase. For example, SElib.subclassWindow() does not apply to the DOS operating system.

Routines for arrays

For dynamic arrays

getArrayLength	Determines size of an array.
setArrayLength	Sets the size of an array.

For Array objects

Array.join	Creates a string from array elements.
Array.sort	Sorts array elements.
Array.reverse	Reverses the order of elements of an array.

Array properties

Array.length	Returns the length of array.

Routines for Buffers

Buffer methods

Buffer.getString	Returns a string starting from the current cursor position.
Buffer.getValue	Returns a value from a specified position.
Buffer.putString	Puts a string into a Buffer.
Buffer.putValue	Puts a specified value into a buffer.
Buffer.subBuffer	Returns a section of a buffer.

Buffer.toString	Returns string equivalent of the current state of buffer.

Buffer properties

Buffer.bigEndian	Boolean flag for bigEndian byte ordering.
Buffer.cursor	Current position within a buffer.
Buffer.data	Reference to the internal data of a buffer.
Buffer.size	Size of a Buffer object.
Buffer.unicode	Boolean flag for the use of unicode strings.

Routines for character classification

Clib.isalnum	Tests for alphanumeric character.
Clib.isalpha	Tests for alphabetic character.
Clib.isascii	Tests for ASCII coded character.
Clib.iscntrl	Tests for any control character.
Clib.isdigit	Tests for any decimal-digit character.
Clib.isgraph	Tests for any printing character except space.
Clib.islower	Tests for lower-case alphabetic letter.
Clib.isprint	Tests for any printing character.
Clib.ispunct	Tests for punctuation character.
Clib.isspace	Tests for white-space character.
Clib.isupper	Tests for upper-case alphabetic character.
Clib.isxdigit	Tests for hexadecimal-digit character.

Routines for console I/O

Clib.kbhit	Checks if a keyboard keystroke is available.
Clib.getch	Gets a character from the keyboard, no echo.
Clib.getchar	Gets character from standard input, keyboard.
Clib.getche	Gets character from the keyboard, with echo.
Clib.gets	Reads string from standard input, keyboard.
Clib.perror	Displays a message describing error in errno.
Clib.printf	Formatted output to standard output, screen.
Clib.putchar	Writes a character to standard output, screen.
Clib.puts	Writes a string to standard output, console.
Clib.scanf	Formatted input from standard input, keyboard.
Clib.vprintf	Formatted output to stdout, screen, variable args.

Clib.vscanf	Formatted input from stdin, keyboard, variable args.

Routines for conversion/casting

parseInt	Converts a string to an Integer.
parseFloat	Converts a string to a Float.
escape	Escapes special characters in a string.
unescape	Removes escape sequences in a string.
ToBoolean	Converts a value to a Boolean.
ToBuffer	Converts a value to a Buffer.
ToBytes	Converts a value to a Buffer, raw transfer.
ToInt32	Converts a value to a large Integer.
ToInteger	Converts a value to an Integer.
ToNumber	Converts a value to a Number.
ToObject	Converts a value to an Object.
ToPrimitive	Converts a value to a Primitive.
ToString	Converts a value to a String.
ToUint16	Converts a value to an unsigned Integer.
ToUint32	Converts a value to an unsigned large Integer.

Routines for data/variables

Methods for data

Blob.get	Reads data from specified location of a BLOb.
Blob.put	Writes data into specified location of a BLOb.
Blob.size	Determine size of a BLOb.
defined	Tests if variable has been defined.
getAttributes	Gets attributes of a variable.
isNaN	Determines if a value is Not a Number.
isFinite	Determines if a value is finite.
setAttributes	Sets attributes of a variable.
undefine	Makes a variable undefined.
SElib.getObjectProperties	Get name list of members of object/structure.

toString	Converts any variable to a string representation.
valueOf	Returns the value of any variable.

Properties for data

_BigEndianMode	Global variable, ScriptEase-data/memory-data.

Routines for date/time

Clib.asctime	Converts data and time to an ASCII string.
Clib.clock	Gets processor time.
Clib.ctime	Converts date-time to an ASCII string.
Clib.difftime	Computes difference between two times.
Clib.gmtime	Converts data and time to GMT.
Clib.localtime	Converts date/time to a structure.
Clib.mktime	Converts time structure to calendar time.
Clib.time	Gets current time.
Clib.strftime	Formatted write of date/time to a string.
Date.getDate	Returns the day of the month.
Date.getDay	Returns the day of the week.
Date.getFullYear	Returns the year with four digits.
Date.getHours	Returns the hour.
Date.getMilliseconds	Returns the millisecond.
Date.getMinutes	Returns the minute.
Date.getMonth	Returns the month.
Date.getSeconds	Returns the second.
Date.getTime	Returns date/time, milliseconds, in Date object.
Date.getTimezoneOffset	Returns difference, in minutes, from GMT.
Date.getUTCDate	Returns the UTC day of the month.
Date.getUTCDay	Returns the UTC day of the week.
Date.getUTCFullYear	Returns the UTC year with four digits.
Date.getUTCHours	Returns the UTC hour.
Date.getUTCMilliseconds	Returns the UTC millisecond.
Date.getUTCMinutes	Returns the UTC minute.
Date.getUTCMonth	Returns the UTC month.
Date.getUTCSeconds	Returns the UTC second.
Date.getYear	Returns the year with two digits.
Date.setDate	Set day of the month.

Date.setFullYear	Sets the year with four digits.
Date.setHours	Sets the hour.
Date.setMilliseconds	Sets the millisecond.
Date.setMinutes	Sets the minute.
Date.setMonth	Sets the month.
Date.setSeconds	Sets the second.
Date.setTime	Sets date/time, in milliseconds, in Date object.
Date.setUTCDate	Sets the UTC day of the month.
Date.setUTCFullYear	Sets the UTC year with four digits.
Date.setUTCHours	Sets the UTC hour.
Date.setUTCMilliseconds	Sets the UTC millisecond.
Date.setUTCMinutes	Sets the UTC minute.
Date.setUTCMonth	Sets the UTC month.
Date.setUTCSeconds	Sets the UTC second.
Date.setYear	Sets the year with two digits.
Date.toGMTString	Converts a Date object to a string.
Date.toLocaleString	Returns a string for local date and time.
Date.toSystem	Converts a Date object to a system time.
Date.toUTCString()	Returns a string that represents the UTC date.
Date.fromSystem	Converts system time to Date object time.
Date.parse	Converts a Date string to a Date object.
Date.UTC	Returns date/time, milliseconds, use parameters.

Routines for diagnostic/error

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

Routines for directory, file, and OS

Clib.chdir	Changes directory.
Clib.flock	File locking.
Clib.getcwd	Gets current working directory.

Clib.mkdir	Makes a directory.
Clib.rmdir	Removes a directory.
Clib.getenv	Gets an environment string.
Clib.putenv	Sets an environment string.
SElib.directory	Searches directory listing for file spec.
SElib.fullPath	Converts partial path spec to full path name.
SElib.splitFileName	Gets directory, name, and extension parts of a file spec.

Routines for display control

Screen.clear	Clears screen.
Screen.cursor	Gets/sets cursor position in the visible screen.
Screen.handle	Gets handle of current ScriptEase window.
Screen.setBackground	Sets background color of current ScriptEase screen.
Screen.setForeground	Sets foreground color of current ScriptEase screen.
Screen.size	Gets the height and width of the screen.
Screen.write	Displays a value.
Screen.writeln	Displays a value with automatic end-of-line characters.

Routines for execution control

Clib.abort	Terminates program, normally due to error.
Clib.assert	Test a condition and abort if it is false.
Clib.atexit	Sets function to be called at program exit.
Clib.exit	Normal program termination.
Clib.system	Passes a command to the command processor.
global.eval	Evaluate string as script code, like SElib.interpret.
SElib.compileScript	Compiles script into executable code.
SElib.inSecurity	Calls security manager initialization routine.
SElib.interpret	Interprets ScriptEase code or source file.
SElib.interpretInNewThread	Creates a new thread within a current process.
SElib.spawn	Runs an external executable.

SElib.suspend	Suspends program execution for a while.

Routines for file/stream I/O

Clib.fclose	Closes an open file.
Clib.feof	Tests if at end of file stream.
Clib.fflush	Flushes stream for open file(s).
Clib.fgetc	Gets a character from file stream.
Clib.fgetpos	Gets current position of a file stream.
Clib.fgets	Gets a string from an input stream.
Clib.fopen	Opens a file.
Clib.fprintf	Formatted output to a file stream.
Clib.fputc	Writes a character to a file stream.
Clib.fputs	Writes a string to a file stream.
Clib.fscanf	Formatted input from a file stream.
Clib.fread	Reads data from a file.
Clib.freopen	Assigns new file spec to a file handle.
Clib.fseek	Sets file position for an open file stream.
Clib.fsetpos	Sets position of a file stream.
Clib.ftell	Gets the current value of the file position.
Clib.fwrite	Writes data to a file.
Clib.getc	Gets a character from file stream.
Clib.putc	Writes a character to a file stream.
Clib.remove	Deletes a file.
Clib.rename	Renames a file.
Clib.rewind	Resets file position to beginning of file.
Clib.tmpfile	Creates a temporary binary file.
Clib.tmpnam	Gets a temporary file name.
Clib.ungetc	Pushes character back to input stream.
Clib.vfprintf	Formatted output to a file stream using variable args.
Clib.vfscanf	Formatted input from a file stream using variable args.

Routines for math

Math methods

Clib.abs	Returns the absolute value of an integer.
Clib.asin	Calculates the arc sine.

Clib.acos	Calculates the arc cosine.
Clib.atan	Calculates the arc tangent.
Clib.atan2	Calculates the arc tangent of a fraction.
Clib.atof	Converts ASCII string to a floating-point number.
Clib.atoi	Converts ASCII string to an integer.
Clib.atol	Converts ASCII string to an integer.
Clib.ceil	Rounds up.
Clib.cos	Calculates the cosine.
Clib.cosh	Calculates the hyperbolic cosine.
Clib.div	Integer division, returns quotient & remainder.
Clib.exp	Computes the exponential function.
Clib.fabs	Absolute value.
Clib.fmod	Modulus, calculate remainder.
Clib.floor	Rounds down.
Clib.frexp	Breaks into a mantissa and an exponential power of 2.
Clib.labs	Returns the absolute value of an integer.
Clib.ldexp	Calculates mantissa * 2 ^ exp.
Clib.ldiv	Integer division, returns quotient & remainder.
Clib.log	Calculates the natural logarithm.
Clib.log10	Calculates the base-ten logarithm.
Clib.max	Returns the largest of one or more values.
Clib.min	Returns the minimum of one or more values.
Clib.modf	Splits a value into integer and fractional parts.
Clib.pow	Calculates x to the power of y.
Clib.rand	Generates a random number.
Clib.sin	Calculates the sine.
Clib.sinh	Calculates the hyperbolic sine.
Clib.sqrt	Calculates the square root.
Clib.srand	Seeds random number generator.
Clib.tan	Calculates the tangent.
Clib.tanh	Calculates the hyperbolic tangent.

Math.abs	Returns the absolute value of an integer.
Math.acos	Calculates the arc cosine.
Math.asin	Calculates the arc sine.
Math.atan	Calculates the arc tangent.
Math.atan2	Calculates the arc tangent of a fraction.
Math.ceil	Rounds up.
Math.cos	Calculates the cosine.

Math.exp	Computes the exponential function.
Math.floor	Rounds down.
Math.log	Calculates the natural logarithm.
Math.max	Returns the largest of one or more values.
Math.min	Returns the minimum of one or more values.
Math.pow	Calculates x to the power of y.
Math.random	Returns a random number.
Math.round	Rounds value up or down.
Math.sin	Calculates the sine.
Math.sqrt	Calculates the square root.
Math.tan	Calculates the tangent.

Math properties

Math.E	Value of e, base for natural logarithms.
Math.LN10	Value for the natural logarithm of 10.
Math.LN2	Value for the natural logarithm of 2.
Math.LOG2E	Value for the base 2 logarithm of e.
Math.LOG10E	Value for the base 10 logarithm of e.
Math.PI	Value for pi.
Math.SQRT1_2	Value for the square root of **2**.
Math.SQRT2	Value for the square root of 2.
Number.MAX_VALUE	Largest number (positive)
Number.MIN_VALUE	Smallest number (negative)
Number.NaN	Not a Number
Number.POSITIVE_INFINITY	Number above MAX_VALUE
Number.NEGATIVE_INFINITY	Number below MIN_VALUE

Routines for memory manipulation

SElib.peek	Reads data from memory location.
SElib.pointer	Gets address of variable.
SElib.poke	Writes data to memory location.

Routines for miscellaneous

Clib.bsearch	Binary search for member of a sorted array.

Clib.qsort	Sorts an array, may use comparison function.

Routines for strings/byte arrays

Clib.memchr	Searches a byte array.
Clib.memcmp	Compares two byte arrays.
Clib.memcpy	Copies from one byte array to another.
Clib.memmove	Moves from one byte array to another.
Clib.memset	Copies from one byte array to another.
Clib.rsprintf	Returns formatted string.
Clib.sprintf	Formatted output to a string.
Clib.sscanf	Formatted input from a string.
Clib.strcat	Concatenates strings.
Clib.strchr	Searches a string for a character.
Clib.strcmp	Compares two strings.
Clib.strcmpi	Case-insensitive compare of two strings.
Clib.strcpy	Copies one string to another.
Clib.strcspn	Searches string for first character in a set of characters.
Clib.stricmp	Case-insensitive compare of two strings.
Clib.strlen	Gets the length of a string.
Clib.strlwr	Converts a string to lowercase.
Clib.strncat	Concatenates bytes of one string to another.
Clib.strncmp	Compares part of two strings.
Clib.strncmpi	Case-insensitive compare of parts of two strings.
Clib.strncpy	Copies bytes from one string to another.
Clib.strnicmp	Case-insensitive compare of parts of two strings.
Clib.strpbrk	Searches string for character from a set of characters.
Clib.strrchr	Searches string for the last occurrence of a character.
Clib.strspn	Searches string for character not in a set of characters.
Clib.strstr	Searches a string for a substring.
Clib.strtod	Converts a string to a floating-point value.
Clib.strstri	Case insensitive version of Clib.strstr.
Clib.strtok	Searches a string for delimited tokens.
Clib.strtol	Converts a string to an integer value.
Clib.strupr	Converts a string to uppercase.
Clib.toascii	Converts to ASCII.
Clib.tolower	Converts to lowercase.

Clib.toupper	Converts to uppercase.
Clib.vsprintf	Formatted output to string using variable args.
Clib.vsscanf	Formatted input from a string.
String.charAt	Returns a character in a string.
String.charCodeAt	Returns a unicode character in a string.
String.indexOf	Returns index of first substring in a string.
String.lastIndexOf	Returns index of last substring in a string.
String.split	Splits a string into an array of strings.
String.substring	Retrieves a section of a string.
String.toLowerCase	Converts a string to lowercase.
String.toUpperCase	Converts a string to uppercase.
String.fromCharCode	Creates a string from character codes.

Routines for variable argument lists

Clib.va_arg	Retrieves variable from variable list of args.
Clib.va_end	Terminates variable list of args.
Clib.va_start	Starts a variable list of args.
Clib.rvsprintf	Returns formatted string using variable args.
Clib.vfprintf	Formatted output to a file stream using variable args.
Clib.vfscanf	Formatted input from file stream using variable args.
Clib.vprintf	Formatted output to stdout, screen, using variable args.
Clib.vscanf	Formatted input from stdin, using var args.
Clib.vsprintf	Formatted output to string using variable args.
Clib.vsscanf	Formatted input from a string.

Index

!, 33
!=, 33
#include, 94
#optionoption, 77
%, 29
%=, 30
&, 31, 122
&&, 33
&=, 31
(Open windows list), 112
(Recent files list), 107
*, 29, 122
*=, 30
/, 29
/=, 30
[]Array initialzer, 50, 129;
  []initialzer:[], 50, 129
^, 32
^=, 32
95CON, 76
95WIN, 76
_argc, 234
_argv, 234
_call(...), 57
_canPut(property), 56
_construct(...), 56
_defaultValue(hint), 56
_delete(property), 56
DOS, 76
DOS32, 76
_get(), 55
_get(property), 55
_hasProperty(property), 56
MAC, 76
NTCON, 77
NTWIN, 77
NWNLM, 76
_OS2_, 76
_put(property, value), 56
SEDESKTOP, 80
SHELL, 77
UNIX, 76
WIN32, 76
WINDOWS, 76
`, 125
|, 31
||, 33
|=, 32
~, 32
+, 29
++, 31
+=, 30
<, 33
<<, 31
<<=, 31
=, 29, 30
==, 33
>, 33
>=, 33
>>, 31
>>=, 31
>>>, 31
>>>=, 31
0, 77
1, 78
abort(), 274
About ScriptEase Debugger...In toolbar,
  112
abs(), 223, 324
accept(), 449
acos(), 223, 325
Add, 103
Add breakpoint, 105
Add/Remove..., 111
Add/Remove...Toolbar, 111
addition, 29
addRecord(), 435
address arithmetic, 122

address operator, 122
address(), 253
addressByName(), 454
and, 33
andbitwise and, 31
Appendix A Grouped Functions, 471
Appendix B Instance and Static Notation, 467
apply(), 250
arc(), 382
argc, 47
arguments[], 45
argv, 47
arithmetic operators, 29
Arrange Icons, 111
Array arithmetic, 48
Array concat(), 132
Array constructor function, 128
Array join(), 132
Array length, 130
Array object, 127
Array object instance methods, 131
Array object instance properties, 130
Array pop(), 133
Array properties, 471
Array push(), 134
Array representation, 115
Array reverse(), 134
Array shift(), 135
Array slice(), 135
Array sort(), 136
Array splice(), 138
Array toString(), 138
Array type, 25
Array unshift(), 139
Array() with length, 131
Array() with list, 131
arrays, 115
asctime(), 269
asin(), 224, 325
asm(), 253
assert(), 275
assign addition, 30
assign division, 30
assign multiplication, 30
assign remainder, 30
assign subtraction, 30
assignment, 29, 30
Assignment arithmetic, 30
assignment bitwise and, 31
assignment bitwise or, 32
assignment bitwise xor, exclusive or, 32
assignment operator, 30
assignment shift left, 31
assignment shift left with zeros, 31
assignment shift right, 31
atan(), 224, 325
atan2(), 225, 326
atexit(), 275
atof(), 326
atoi(), 326
atol(), 327
Auto files, 67
autodecrement, 31
Autoincrement (++) and autodecrement (), 31
Automatic array allocation, 115
automatic arrays, 115
Automatic type conversion, 27
Automatic type declaration, 114
Back quote, 206
Back quote strings, 125
baseWindowFunction(), 149
Basic arithmetic, 29
Basics of ScriptEase, 14
beginTransaction(), 417
BIG_ENDIAN, 77
bigEndian, 187
Bind, 73
Bit operators, 31
bitwise xor, exclusive or, 32
Blob Object, 141
Blob object static methods, 141
Blob.get(), 141
Blob.put(), 142
Blob.size(), 144

blobDescriptor object, 145
blobDescriptor(), 145
blocking(), 449
blue(), 383
Boolean Object, 347
Boolean object instance methods, 347
Boolean type, 23
Boolean(), 347
Boolean.toString(), 347
boundsSafe(), 384
break, 37
Break when Expression, 104
Breakpoint, 110
Breakpoint dialog, 104
Breakpoints listing, 106
breakWindow(), 150
bsearch(), 295
Buffer bigEndian, 187
Buffer cursor, 187
Buffer data, 188
Buffer getString(), 191
Buffer getValue(), 191
Buffer methods, 471
Buffer Object, 187
Buffer object instance methods, 189
Buffer object instance properties, 187
Buffer properties, 472
Buffer putString(), 192
Buffer putValue(), 192
Buffer size, 188
Buffer subBuffer(), 194
Buffer toString(), 195
Buffer unicode, 188
Buffer(), 189
Buffer[] Array, 188
by reference, 121
by referencepassed by reference, 44, 122
by valuepassed by value, 44
call(), 251
Cascade, 111
case, 38
case expression, 119
Case sensitivity, 15
Case statements, 122
caseSensitive, 440
catch, 40
ceil(), 225, 327
cfunction, 45, 113
cfunction keyword, 47
cfunctions, 122
Change Variables, 111
Character classification, 299
charAt(), 208
charCodeAt(), 209
charUp(), 385
chdir(), 293
clear(), 197
clearerr(), 277
Clib, 113
Clib Object, 259
Clib.abort(), 274
Clib.abs(), 324
Clib.acos(), 325
Clib.asctime(), 269
Clib.asin(), 325
Clib.assert(), 275
Clib.atan(), 325
Clib.atan2(), 326
Clib.atexit(), 275
Clib.atof(), 326
Clib.atoi(), 326
Clib.atol(), 327
Clib.bsearch(), 295
Clib.ceil(), 327
Clib.chdir(), 293
Clib.clearerr(), 277
Clib.clock(), 269
Clib.cos(), 327
Clib.cosh(), 327
Clib.ctime(timeInt), 269
Clib.difftime(), 270
Clib.div(), 328
Clib.errno, 277
Clib.exit(), 276
Clib.exp(), 328

Clib.fabs(), 328
Clib.fclose(), 280
Clib.feof(), 281
Clib.ferror(), 277
Clib.fflush(), 281
Clib.fgetc(), 281
Clib.fgetpos(), 282
Clib.fgets(), 282
Clib.flock(), 293
Clib.floor(), 329
Clib.fmod(), 329
Clib.fopen(), 279
Clib.fprintf(), 282
Clib.fputc(), 283
Clib.fputs(), 283
Clib.fread(), 284
Clib.freopen(), 285
Clib.frexp(), 329
Clib.fscanf(), 286
Clib.fseek(), 287
Clib.fsetpos(), 288
Clib.ftell(), 288
Clib.fwrite(), 289
Clib.getc(), 290
Clib.getch(), 262
Clib.getchar(), 263
Clib.getche(), 263
Clib.getcwd(), 293
Clib.getenv(), 298
Clib.gets(), 263
Clib.gmtime(), 270
Clib.isalnum(), 300
Clib.isalpha(), 300
Clib.isascii(), 300
Clib.iscntrl(), 301
Clib.isdigit(), 301
Clib.isgraph(), 301
Clib.islower(), 301
Clib.isprint(), 302
Clib.ispunct(), 302
Clib.isspace(), 302
Clib.isupper(), 302
Clib.isxdigit(), 303
Clib.kbhit(), 264
Clib.labs(), 330
Clib.ldexp(), 330
Clib.ldiv(), 331
Clib.localtime(), 270
Clib.log(), 331
Clib.log10(), 331
Clib.max(), 332
Clib.memchr(), 322
Clib.memcmp(), 322
Clib.memcpy(), 323
Clib.memmove(), 323
Clib.memset(), 324
Clib.min(), 332
Clib.mkdir(), 295
Clib.mktime(), 272
Clib.modf(), 332
Clib.perror(), 278
Clib.pow(), 333
Clib.printf(), 259
Clib.putc(), 290
Clib.putchar(), 264
Clib.putenv(), 299
Clib.puts(), 264
Clib.qsort(), 297
Clib.rand(), 333
Clib.remove(), 291
Clib.rename(), 291
Clib.rewind(), 291
Clib.rmdir(), 295
Clib.rsprintf(), 303
Clib.rvsprintf(), 304
Clib.scanf(), 265
Clib.sin(), 333
Clib.sinh(), 333
Clib.sprintf(), 305
Clib.sqrt(), 334
Clib.sscanf(), 304
Clib.strcat(), 306
Clib.strchr(), 307
Clib.strcmp(), 307
Clib.strcmpi(), 308
Clib.strcpy(), 308

Clib.strcspn(), 308
Clib.strerror(), 278
Clib.strftime(), 272
Clib.stricmp(), 309
Clib.strlen(), 310
Clib.strlwr(), 310
Clib.strncat(), 311
Clib.strncmp(), 311
Clib.strncmpi(), 312
Clib.strncpy(), 312
Clib.strnicmp(), 313
Clib.strpbrk(), 314
Clib.strrchr(), 314
Clib.strspn(), 315
Clib.strstr(), 315
Clib.strstri(), 316
Clib.strtod(), 316
Clib.strtok(), 317
Clib.strtol(), 318
Clib.strupr(), 319
Clib.system(), 276
Clib.tan(), 334
Clib.tanh(), 335
Clib.time(), 274
Clib.tmpfile(), 292
Clib.tmpnam(), 292
Clib.toascii(), 320
Clib.tolower(), 320
Clib.toupper(), 321
Clib.ungetc(chr, filePointer), 292
Clib.va_arg(), 335
Clib.va_end(), 336
Clib.va_start(), 337
Clib.vfprintf(), 338
Clib.vfscanf(), 338
Clib.vprintf(), 266
Clib.vscanf(), 267
Clib.vsprintf(), 321
Clib.vsscanf(), 339
clock(), 269
CLOCKS_PER_SEC, 78
close(), 410, 429, 441, 450
closeConnection(), 375
CloseCtrl+W, 106
Color styles, 381
colorAllocate(), 385
colorClosest(), 386
colorDeallocate(), 386
colorExact(colorExact()red, 387
colorsTotal(), 387
colorTransparent(), 388
columnName(), 410
columns(), 411
Com Object, 459
Com Object Link Library, 459
COMCreateObject(), 459
Command-line switches, 72
Comments, 16
commitTransaction(), 418
Common script libraries, 462
Common utility and sample scripts, 464
compile(), 218
compileScript(), 150
complementBitwise not, complement, 32
Components of main MDI window, 100
Composite data types, 24
concat(), 132, 209
conditional expression, 32
Conditional operator, 40
connect(), 418
connected(), 419
Console I/O functions, 259
constructor function, 49
continue, 38
Conversion or casting, 233
Converting existing C code to ScriptEase, 125
copy(), 388
CopyIn toolbar and Ctrl+C, 108
copyResized(), 389
cos(), 226, 327
cosh(), 327
Creating a DSP object, 371
Creating arrays, 128
ctime(), 269

currentRecord(), 429
cursor, 187
Cursor close(), 410
Cursor columnName(), 410
Cursor columns(), 411
Cursor deleteRow(), 411
Cursor filter, 409
Cursor first(), 412
Cursor insertRow(), 412
Cursor Instance Methods, 410
Cursor Instance Properties, 408
Cursor last(), 413
Cursor next(), 414
Cursor Object, 407
Cursor previous(), 414
Cursor reload(), 415
Cursor sort, 409
Cursor updateRow(), 415
cursor(), 197, 420, 439, 442
CutIn toolbar and Ctrl+X, 107
dashedLine(), 390
data, 188
Data types, 21, 114
Data types in C and SE, 114
Database beginTransaction(), 417
Database commitTransaction(), 418
Database connect(), 418
Database connected(), 419
Database cursor(), 419
Database disconnect(), 420
Database execute(), 421
Database majorErrorCode(), 422
Database majorErrorMessage(), 422
Database minorErrorCode(), 423
Database minorErrorMessage(), 423
Database Object, 416
Database procedureName(), 424
Database procedures(), 424
Database rollbackTransaction(), 424
Database storedProc(), 425
Database table(), 425
Database tableName(), 426
Database tables(), 426
Date and time display, 82
Date getDate(), 351
Date getDay(), 351
Date getFullYear(), 351
Date getHours(), 351
Date getMilliseconds(), 351
Date getMinutes(), 352
Date getMonth(), 352
Date getSeconds(), 352
Date getTime(), 352
Date getTimezoneOffset(), 353
Date getUTCDate(), 353
Date getUTCDay(), 353
Date getUTCFullYear(), 353
Date getUTCHours(), 354
Date getUTCMilliseconds(), 354
Date getUTCMinutes(), 354
Date getUTCMonth(), 354
Date getUTCSeconds(), 354
Date getYear(), 355
Date object, 83, 349
Date object instance methods, 351
Date object static methods, 364
Date setDate(), 355
Date setFullYear(), 355
Date setHours(), 356
Date setMilliseconds(), 356
Date setMinutes(), 356
Date setMonth(), 357
Date setSeconds(), 357
Date setTime(), 358
Date setUTCDate(), 358
Date setUTCFullYear(), 358
Date setUTCHours(), 359
Date setUTCMilliseconds(), 359
Date setUTCMinutes(), 359
Date setUTCMonth(), 360
Date setUTCSeconds(), 360
Date setYear(), 361
Date toDateString(), 361
Date toGMTString(), 361
Date toLocaleDateString(), 362
Date toLocaleString(), 362

Date toLocaleTimeString(), 362
Date toString(), 363
Date toSystem(), 363
Date toTimeString(), 363
Date toUTCString(), 364
Date valueOf(), 364
Date.fromSystem(), 364
Date.parse(), 365
Date.UTC(), 366
Debug menu, 109
debugger, 10, 99
decisions, 34
decode(), 369
default, 38
Default Interpreter..., 109
DefaultLocalVars, 77
define, 61
defined(), 235
deleteAll(), 437
deleteRecord(), 436
deleteRow(), 411
Description of the Cursor object, 407
Description of the SimpleDataset object, 427
Description of the Stproc object, 440
destroy(), 391
difftime(), 270
Directory, 293
directory(), 154
disconnect(), 420
div(), 328
division, 29
do ... while, 36
Document window, 100
Documentation, 11
DOS batch files, 70
Dos Object, 253
Dos object static methods, 253
Dos.address(), 253
Dos.asm(), 253
Dos.inport(), 254
Dos.inportw(), 254
Dos.interrupt(), 255
Dos.offset(), 256
Dos.outport(), 256
Dos.outportw(), 256
Dos.segment(), 257
doWindows(), 156
drawChar(), 384
DSP dspClose(), 377
DSP dspCloseConnection(), 374
DSP dspGetValue(), 378
DSP dspLoad(), 376
DSP dspReceive(), 375
DSP dspSecurityGuard(), 379
DSP dspSecurityInit(), 378
DSP dspSecurityTerm(), 379
DSP dspSend(), 376
DSP dspService(), 377
DSP Link Library, 371
DSP Object, 371
DSP object instance methods, 374
DSP object static properties, 380
DSP(), 374
dsp.dspClose(), 377
dsp.dspService(), 377
DSP.remote, 380
dspGetValue(), 378
dspLoad(), 376
dspReceive(), 375
dspSecurityGuard(), 379
dspSecurityInit(), 378
dspSecurityTerm(), 379
dspSend(), 376
Dynamic links, 180
Dynamic objects, 55
dynamicLink() - OS/2, 183
dynamicLink() - Win16, 182
dynamicLink() - Win32, 180
E, 221
Edit menu, 107
elements, 127
else, 35
encode(), 369
Environment variables, 298
EOF, 78

equality, 33
errno, 277
Error, 277
Error checking for functions, 46
error(), 455
escape sequences, 125
Escape sequences for characters, 205
escape(), 236
eval(), 236
Exception handling, 40
exec(), 217
execute(), 421, 443
Executing a script, 66
Exit, 107
exit(), 276
EXIT_FAILURE, 78
EXIT_SUCCESS, 78
exp(), 226, 328
expression, 16, 103
Expressions, statements, and blocks, 16
fabs(), 328
false, 77
FATTR_ARCHIVE, 78
FATTR_HIDDEN, 78
FATTR_NORMAL, 78
FATTR_RDONLY, 78
FATTR_SUBDIR, 78
FATTR_SYSTEM, 78
fclose(), 280
feof(), 281
ferror(), 277
fflush(), 281
fgetc(), 281
fgetpos(), 282
fgets(), 282
File I/O, 279
File menu, 106
File Name for breakpoint, 105
File redirection, 66
fill(), 391
filledPolygon(), 392
filledRectangle(), 392
fillToBorder(), 393
filter, 409
finally, 40
find(), 431, 433
Find...Ctrl+F, 109
findAll(), 434
findDistinct(), 434
first(), 412
firstRecord(), 430
Floating point, 23
flock(), 293
floor(), 227, 329
Flow decisions statements, 34
fmod(), 329
Font specifications, 381
Font..., 108
fopen(), 279
for, 36, 54
For Array objects, 471
For dynamic arrays, 471
for/in, 54
for/in statement, 54
fork(), 341
Format String, 104
fprintf(), 282
fputc(), 283
fputs(), 283
fread(), 284
freopen(), 286
frexp(), 329
fromCharCode(), 216
fromGd(), 402
fromGif(), 403
fromSystem(time), 364
fromXbm(), 403
fscanf(), 286
fseek(), 287
fsetpos(), 288
ftell(), 288
fullpath(), 158
Function apply(), 250
Function call(), 251
Function identifier, 20
Function Object, 249

Function object instance methods, 249
Function property arguments[], 45
Function recursion, 46
Function return statement, 43
Function scope, 20
Function toString(), 252
Function with a return, 87
Function with parameters, 84
Function(), 249
Functions, 20, 42
fwrite(), 289
GD arc(), 382
GD blue(), 383
GD boundsSafe(, 384
GD charUp(), 385
GD colorAllocate(), 385
GD colorClosest(), 386
GD colorDeallocate(), 386
GD colorExact(), 387
GD colorsTotal(), 387
GD colorTransparent(), 388
GD copy(), 388
GD copyResized(), 389
GD dashedLine(), 390
GD destroy(), 391
GD drawChar(), 384
GD fill(), 391
GD filledPolygon(), 392
GD filledRectangle(), 392
GD fillToBorder(), 393
GD getInterlaced(), 393
GD getPixel(), 394
GD getTransparent(), 394
GD green(), 394
GD height(), 395
GD interlace(), 395
GD line(), 395
GD Link Library, 381
GD Object, 381
GD object instance methods, 382
GD object static methods, 402
GD polygon(), 396
GD rectangle(), 397
GD red(), 398
GD setBrush(), 398
GD setPixel(), 399
GD setStyle(), 399
GD setTile(), 400
GD string(), 400
GD stringUp(), 401
GD toGd(), 402
GD toGif(), 402
GD width(), 402
GD(), 382
GD.fromGd(), 402
GD.fromGif(), 403
GD.fromXbm(), 403
get(), 141
getArrayLength(), 237
getAttributes(), 238
getc(), 290
getch(), 262
getchar(), 263
getche(), 263
getcwd(), 293
getDate(), 351
getDay(), 351
getenv(), 298
getFullYear(), 351
getHours(), 351
getInterlaced(), 394
getLastError(), 439
getLastErrorCode(), 439
getMilliseconds(), 352
getMinutes(), 352
getMonth(), 352
getObjectProperties(), 159
getPixel(), 394
gets(), 263
getSeconds(), 352
getString(), 191
getTime(), 352
getTimezoneOffset(), 353
getTransparent(), 394
getUTCDate(), 353
getUTCDay(), 353

getUTCFullYear(), 353
getUTCHours(), 354
getUTCMilliseconds(), 354
getUTCMinutes(), 354
getUTCMonth(), 354
getUTCSeconds(), 354
getValue(), 191
getYear(), 355
global, 19
Global object, 233
global object methods/functions, 235
global object properties, 234
Global variables, 19
Global...Ctrl+Shft+G, 111
global._argc, 234
global._argv, 234
global.defined(), 235
global.escape(), 235
global.eval(), 236
global.getArrayLength(), 237
global.getAttributes(), 238
global.isFinite(), 237
global.isNaN(), 237
global.parseFloat(), 239
global.parseInt(), 239
global.setArrayLength(), 240
global.setAttributes(), 240
global.ToBoolean(), 243
global.ToBuffer(), 243
global.ToBytes(), 244
global.ToInt32(), 244
global.ToInteger(), 244
global.ToNumber(), 245
global.ToObject(), 245
global.ToPrimitive, 246
global.ToString(), 246
global.Uint16(), 247
global.Uint32(), 248
global.undefine, 242
global.unescape(), 247
Globals, 101
Globals window, 102
gmtime(), 270
GoCtrl+F5, 110
goto, 39
goto and labels, 39
greater than, 33
greater than or equal to, 33
green(), 395
GUI environment, 69
handle(), 198
hasOwnProperty(), 219
height(), 395
Help menu, 112
Help Topics...F1, 112
hostByName(), 455
hostName(), 455
Identifiers, 17
if, 34
if, ifdef, elif, else, endif, 63
in, 54
include, 62
indexOf(), 210
inequality, 33
Initialization code which is external to functions, 123
Initializers for arrays and objects, 129
Initializers for objects and arrays, 50
inport(), 254
inportw(), 254
inSecurity(), 160
insertRow(), 412
inside of functions, 123
Installation, 11
instance(), 160
instanceof operator, 34
instanceof(), 34
Integer, 22
Integrated Debugger, 99
interlace(), 395
INTERP_COMPILED_SCRIPT, 78
INTERP_FILE, 79
INTERP_LOAD, 78
INTERP_NOINHERIT_GLOBAL, 78
INTERP_NOINHERIT_LOCAL, 79
INTERP_TEXT, 79

interpret(), 161
interpreter, 10
interpretInNewThread(), 162
interrupt(), 255
Introduction, 9
isalpha(), 300
isascii(), 300
iscntrl(), 301
isdigit(), 301
isFinite(), 237
isgraph(), 301
islower(), 301
isNaN(), 237
isprint(), 302
isPrototypeOf(), 219
ispunct(), 302
isspace(), 302
isupper(), 302
isxdigit(), 303
join(), 132
kbhit(), 264
kill(), 342
label, 39
labs(), 330
last(), 413
lastIndexOf(), 211
lastRecord(), 431
ldexp(), 330
ldiv(), 331
length, 130, 208
less than, 33
less than or equal to, 33
Library and sample files, 94
Library file, 91
Library files, 10, 82
Line Number for breakpoint, 106
line(), 395
linger(), 450
link, 64
Link Libraries, 367
Literal strings, 116
Literal strings and assignments, 117
Literal strings and comparisons, 117
Literal strings and parameters, 118
Literal strings and returns, 118
Literal Strings and switch statements, 118
LN10, 221
LN2, 221
local, 19
Local...Ctrl+Shft+L, 112
Locals, 102
Locals window, 102
localtime(), 270
LOCK_EX, 79
LOCK_NB, 79
LOCK_SH, 79
LOCK_UN, 79
log(), 227, 331
log10(), 331
LOG10E, 222
LOG2E, 222
Logical operators, 32
Logical operators and conditional expressions, 32
Long Strings, 206
Macros, 124
Main menu bar, 106
main(), 47, 83
main() function, 47
majorErrorCode(), 422
majorErrorMessage(), 422
makeWindow(), 163
Math, 324
Math methods, 477
Math Object, 221
Math object static methods, 223
Math object static properties, 221
Math properties, 479
Math.abs(), 223
Math.acos(), 223
Math.asin(), 224
Math.atan(), 224
Math.atan2(), 225
Math.ceil(), 225
Math.cos(), 226

Math.E, 221
Math.exp(X), 226
Math.floor(), 226
Math.LN10, 221
Math.LN2, 221
Math.log(), 227
Math.LOG10E, 222
Math.LOG2E, 222
Math.max(), 227
Math.min(), 228
Math.PI, 222
Math.pow(), 228
Math.random(), 229
Math.round(), 229
Math.sin(), 230
Math.sqrt(), 231
Math.SQRT1_2, 222
Math.SQRT2, 223
Math.tan(), 231
Mathematical operators, 29
MathErrorWarnings, 77
max(), 228, 332
MAX_VALUE, 26
MAX_VALUENumber.MAX_VALUE, 79
MD5 Checksum Link Library, 405
md5 Object, 405
md5 object instance methods, 405
md5(), 405
MDI windows, 101
memchr(), 322
memcmp(), 322
memcpy(), 323
memmove(), 323
Memory manipulation, 322
memset(), 324
Menu bar, 100
messageFilter(), 167
Methods - assigning functions to objects, 51
Methods for data, 473
min(), 228, 332
MIN_VALUE, 26
MIN_VALUENumber.MIN_VALUE, 79
minorErrorCode(), 423
minorErrorMessage(), 423
mkdir(), 295
mktime(), 272
modf(), 332
modulo, 29
multiplication, 29
multiTask(), 168
NaN, 26, 79
NaNNumber.NaN, 79
NEGATIVE_INFINITY, 27
NEGATIVE_INFINITYNumber.NEGATIVE_INFINITY, 79
new constructorconstructor, 50, 129
NewIn toolbar and Ctrl+N, 106
next(), 414
nextRecord(), 429
not, 32, 33
null, 25, 77
Number constants, 26
Number Object, 147
Number object instance methods, 147
Number toLocaleString(), 147
Number toString(), 147
Number type, 22
numbers to strings, 27
Object hasOwnProperty(), 219
Object initializer, 50, 129; initializer:, 50, 129
Object isPrototypeOf(), 219
Object Object, 219
Object object instance methods, 219
Object operator, 28
Object propertyIsEnumerable(), 219
Object prototypes, 52
Object toLocaleString(), 220
Object toString(), 220
Object type, 24
Objects, 49
offset(), 256
Open...In toolbar and Ctrl+O, 106

Operating system command prompt, 68
Operators, 28
option, 65
Options, 108
or, 33
orbitwise or, 31
OS/2 and seos2pm.exe, 73
OS/2 batch file, 71
OS/2 REXX file, 72
outport(), 256
outportw(), 257
outside of functions, 123
P_NOWAIT, 79
P_OVERLAY, 79
P_SWAP, 79, 276
P_WAIT, 80
parameterName(), 442
parameters of cfunctions, 45
parameters(), 442
Parameters..., 110
parse(), 365
parseFloat(), 239
parseInt(), 239
pass by referenceby reference, 24
pass by valueby value, 21
pass variables to functions, 44
Passing information to cfunctions, 45
Passing information to functions, 44
Passing variables by reference, 120
PasteIn toolbar and Ctrl+V, 108
PATH, 60, 96
peek(), 168
perror(), 278
PI, 222
Platform, 76
Point specifications, 381
pointer, 122
Pointer operator *, 122
pointer(), 169
poke(), 171
polygon(), 396
pop(), 133
POSITIVE_INFINITY, 26
POSITIVE_INFINITYNumber.POSITIVE_INFINITY, 79
pow(), 228, 333
Predefined constants and values, 77
Predefined Values, 75
Predefining objects with constructor functions, 49
Preprocessor, 59
Preprocessor Directives, 61
Preprocessor values, 75
previous(), 414
prevRecord(), 430
Prewritten routines, 11
Primitive data types, 21, 122
Print Preview, 107
Print Setup..., 107
Print...In toolbar and Ctrl+P, 107
printf(), 259
procedureName(), 424
procedures(), 424
Prohibited identifiers, 18
Properties and methods of basic data types, 27
Properties for data, 474
propertyIsEnumerable(), 220
prototype, 52
Prototype property, 469
push(), 134
put(), 142
putc(), 290
putchar(), 264
putenv(), 299
puts(), 264
putString(), 192
putValue(), 192
qsort(), 297
Quick Start Tutorial, 81
rand(), 333
RAND_MAX, 80
random(), 229
read(), 451
ready(), 452
rectangle(), 397

recursion, 46
recursive function, 46
red(), 398
RegExp compile(), 218
RegExp exec(), 217
RegExp Object, 217
RegExp object instance methods, 217
RegExp test(), 218
RegExp(), 217
reload(), 415
remote, 380
remoteHost(), 453
Remove, 103
Remove All, 103
Remove allIn toolbar, 111
Remove allToolbar, 111
Remove breakpoint, 105
remove(), 291
rename(), 291
Replace...Ctrl+R, 109
replaceRecord(), 437
RequireFunctionKeyword, 77
RequireVarKeyword, 77
Restart, 109
return, 43, 88
return values, 88
reverse(), 134
rewind(), 291
rmdir(), 295
rollbackTransaction(), 424
round(), 230
Routines for arrays, 471
Routines for Buffers, 471
Routines for character classification, 472
Routines for console I/O, 472
Routines for conversion/casting, 473
Routines for data/variables, 473
Routines for date/time, 474
Routines for diagnostic/error, 475
Routines for directory, file, and OS, 475
Routines for display control, 476
Routines for execution control, 476
Routines for file/stream I/O, 477
Routines for math, 477
Routines for memory manipulation, 479
Routines for miscellaneous, 479
Routines for strings/byte arrays, 480
Routines for variable argument lists, 481
rsprintf(), 303
Run in DebuggerIn toolbar and F5, 110
Running a script, 68
rvsprintf(), 304
Sample files, 11
Save As..., 107
SaveIn toolbar and Ctrl+S, 107
scanf(), 265
scope, 20
Screen object, 82, 197
Screen object static methods, 197
Screen.clear(), 197
Screen.cursor(), 197
Screen.handle(), 198
Screen.setBackground(), 199
Screen.setForeground(), 200
Screen.size(), 201
Screen.write improved, 88
Screen.write(), 82, 201
Screen.writeln(), 82, 203
script, 82
Script execution, 274
Script Libraries, 461
ScriptEase 4.20, 1
ScriptEase Desktop, 9
ScriptEase JavaScript, 13
ScriptEase package, 10
ScriptEase Shell, 81
ScriptEase shell command prompt, 69
ScriptEase shell command-line, 66
ScriptEase versus C language, 113
SE_ESET, 61
Search menu, 109
SEDBC Link Library, 407
SEDESKPATH, 60, 96
SEDESKPREFS, 60, 95

SEEK_CUR, 80
SEEK_END, 80
SEEK_SET, 80
segment(), 257
select(), 455
SElib, 113
SElib Object, 149
SElib object static methods, 149
SElib.baseWindowFunction(), 149
SElib.breakWindow(), 150
SElib.compileScript(), 150
SElib.directory(), 154
SElib.doWindows(), 156
SElib.dynamicLink() - for OS/2, 183
SElib.dynamicLink() - for Win16, 182
SElib.dynamicLink() - for Win32, 180
SElib.fullpath(), 158
SElib.getObjectProperties(), 159
SElib.inSecurity(), 160
SElib.instance(), 160
SElib.interpret(), 161
SElib.interpretInNewThread(), 162
SElib.makeWindow(), 163
SElib.messageFilter(), 167
SElib.multiTask(), 168
SElib.peek(), 168
SElib.pointer(), 169
SElib.poke(), 171
SElib.ShellFilterCharacter(), 172
SElib.ShellFilterCommand(), 173
SElib.spawn(), 174
SElib.splitFilename(), 176
SElib.subclassWindow(), 177
SElib.suspend(), 178
SElib.windowList(), 179
semicolons, 123
setArrayLength(), 240
setAttributes(), 240
setBackground(), 199
setBrush(), 398
setDate(day), 355
setForeground(), 200
setFullYear(), 355
setgid(), 342
setHours(), 356
setMilliseconds(), 356
setMinutes(), 356
setMonth(), 357
setPixel(), 399
setSeconds(), 357
setsid(), 343
setStyle(), 399
setTile(), 400
setTime(), 358
Setting breakpoints, 104
Setting watches, 102
setuid(), 343
setUTCDate(), 358
setUTCFullYear(), 358
setUTCHours(), 359
setUTCMilliseconds(), 359
setUTCMinutes(), 359
setUTCMonth(), 360
setUTCSeconds(), 360
setYear(), 361
Shell, 77
ShellFilterCharacter(), 172
ShellFilterCommand(), 173
shift, 31
shift left with zeros, 31
shift right, 31
shift(), 135
Simple script, 82
SimpleDataset addRecord(), 435
SimpleDataset close(), 429
SimpleDataset currentRecord(), 429
SimpleDataset cursor(), 438
SimpleDataset deleteAll(), 437
SimpleDataset deleteRecord(), 436
SimpleDataset find() with clause, 433
SimpleDataset find() with template, 431
SimpleDataset findAll(), 434
SimpleDataset findDistinct(), 434
SimpleDataset firstRecord(), 430
SimpleDataset getLastError(), 439
SimpleDataset getLastErrorCode(), 439

SimpleDataset instance methods, 428
SimpleDataset lastRecord(), 431
SimpleDataset nextRecord(), 429
SimpleDataset Object, 427
SimpleDataset prevRecord(), 430
SimpleDataset replaceRecord(), 437
SimpleDataset static properties, 439
SimpleDataset(), 428
SimpleDataset.caseSensitive, 440
sin(), 230, 333
Single quote, 206
sinh(), 334
size(), 144, 201
slice(), 135, 212
Socket accept(), 449
Socket blocking(), 449
Socket close(), 450
Socket linger(), 450
Socket Link Library, 447
Socket Object, 447
Socket object instance methods, 447
Socket object static methods, 454
Socket read(), 450
Socket ready(), 452
Socket remoteHost(), 453
Socket write(), 453
Socket() with hostname, 447
Socket() with port, 448
Socket.addressByName(), 454
Socket.error(), 454
Socket.hostByName(), 455
Socket.hostName(), 455
Socket.select(), 455
sort, 409
sort(), 136
Sorting, 295
Source, 101
Source Mark, 108
spawn(), 174
Special values, 25
splice(), 138
split(), 212
splitFilename(), 176
sprintf(), 305
sqrt(), 231, 334
SQRT1_2, 222
SQRT2, 223
srand(), 334
sscanf(), 304
Start Debug Session, 109
statement, 16
statement blockblock, 16
Status bar, 100
Status Bar view, 109
stderr, 80
stdin, 80
stdout, 80
Step IntoIn toolbar and F9, 110
Step OutIn toolbar and F12, 110
Step OverIn toolbar and F10, 110
Step to CursorIn toolbar and F11, 110
StopIn toolbar, 110
storedProc(), 425
Stproc close(), 441
Stproc cursor(), 442
Stproc execute(), 443
Stproc instance methods, 441
Stproc instance properties, 440
Stproc Object, 440
Stproc parameterName(), 442
Stproc parameters(), 442
strcat(), 306
strchr(), 307
strcmp(), 307
strcmpi(), 308
strcpy(, 308
strcspn(), 309
strerror(), 278
strftime(), 272
stricmp(), 309
Strictness of interpretation, 77
String as data type, 205
String as object, 207
String charAt(), 208
String charCodeAt(index), 209
String concat(), 209

String indexOf(), 210
String instance methods, 468
String instance properties, 468
String lastIndexOf(), 211
String length, 208
String localeCompare(), 211
String manipulation, 303
String Object, 205
String object instance methods, 208
String object instance properties, 208
String object static methods, 216
String slice(), 212
String split(), 212
String static methods, 469
String static properties, 468
String substring(), 213
String toLocaleLowerCase(), 214
String toLocaleUpperCase(), 214
String toLowerCase(), 214
String toUpperCase()), 215
String type, 23
String valueOf(), 215
String(), 208, 401
String.fromCharCode(), 216
strings to numbers, 27
stringUp(), 401
strlen(), 310
strlwr(), 310
strncat(), 311
strncmp(), 311
strncmpi(), 312
strncpy(), 312
strnicmp(), 313
strpbrk(), 314
strrchr(), 314
strspn(), 315
strstr(), 315
strstri(), 316
strtod(), 316
strtok(), 317
strtol(), 318
Structures, 119
strupr(), 319
subBuffer(), 194
subclassWindow(), 177
substring(), 213
subtraction, 29
suspend(), 178
switch, 38
switch expression, 119
switch, case, and default, 38
system(), 276
table(), 425
tableName(), 426
tables(), 426
Tabs..., 108
tan(), 231, 334
tanh(), 335
term(), 405
Terminology, 86
test(), 218
The debugger, 10
The interpreter, 10
this, 50
throw, 40
Tile, 111
Time functions, 268
time(), 274
tmpfile(), 292
tmpnam(), 292
toascii(), 320
ToBoolean(), 243
ToBuffer(), 243
ToBytes(), 244
toGd(), 402
Toggle currentIn toolbar and F8, 110
toGif(), 402
toGMTString(), 361
ToInt32(), 244
ToInteger(, 244
Token replacement macros, 124
toLocaleDateString(), 362
toLocaleLowerCase(), 214
toLocaleString(), 83, 147, 220, 362
toLocaleTimeString(), 362
toLocaleUpperCase(), 214

tolower(), 320
toLowerCase(), 214, 215
ToNumber(), 245
ToObject(), 245
Tool bar, 100
Toolbar view, 109
ToPrimitive(), 246
toString(), 28, 246
toSystem(), 363
toTimeString(), 363
ToUint16(), 247
ToUint32(), 248
toupper(), 321
toUTCString(), 364
Trace On, 108
Trace over, 108
Trace Speed, 108
Transactions, 416
true, 77, 78
try, 40
tutorial, 81
Type declarations, 125
typeof operator, 34
typeof(), 34
undefine(), 242
undefined, 25
UndoCtrl+Z, 107
unescape(), 247
ungetc(), 292
unicode, 188
Unix, 72
Unix Object, 341
Unix object static methods, 341
Unix.fork(), 341
Unix.kill(), 342
Unix.setgid(), 342
Unix.setsid(), 343
Unix.setuid(), 343
Unix.wait(), 343
Unix.waitpid(), 344
Unnecessary tokens, 123
unshift(), 139
update(), 406
updateRow(), 415
Using a DSP object, 372
Using library files, 95
Using the ScriptEase Debugger, 100
UTC(), 366
UU object static methods, 369
UU.decode(), 369
UU.encode(), 369
UUCode Link Library, 369
va_arg(), 335
va_end(), 337
va_start(), 337
valueOf(), 28, 215
Variable argument lists, 335
Variable scope, 19
Variables, 18
Variables in the environment, 59
VERSION_MAJOR, 80
VERSION_MINOR, 80
VERSION_STRING, 80
vfprintf(), 338
View menu, 109
vprintf(), 266
vscanf(), 267
vsprintf(), 321
vsscanf(), 339
wait(), 343
waitpid(), 344
Watch, 111
Watch dialog, 103
Watch...Ctrl+Shft+W, 112
Watches, 102
Watches window, 102
while, 35, 36
White space characters, 15
width(), 402
Win32 script libraries, 464
Win32 utility and sample scripts, 466
Window menu, 111
windowList(), 179
with, 54
with statement, 54
write(), 82, 88, 201, 453

writeln(), 82, 203